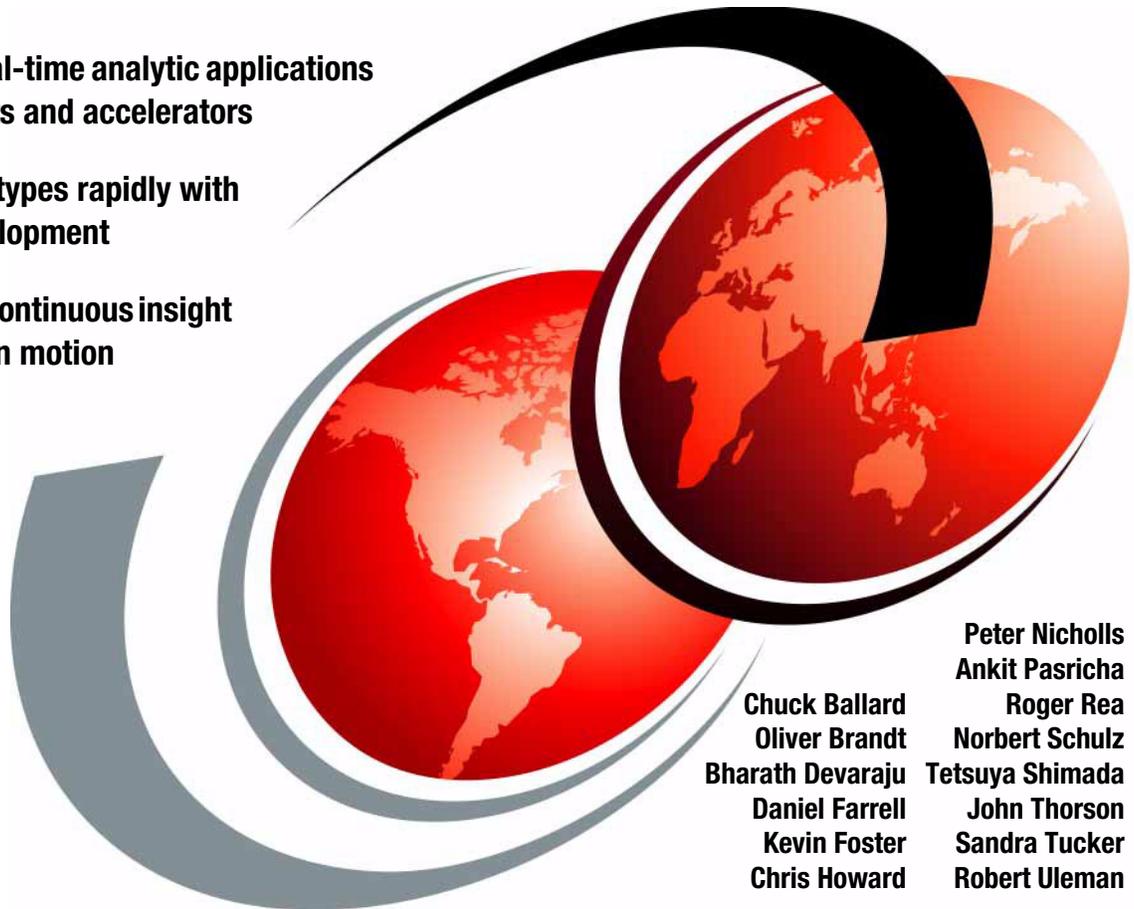


IBM InfoSphere Streams Accelerating Deployments with Analytic Accelerators

Develop real-time analytic applications
with toolkits and accelerators

Build prototypes rapidly with
visual development

Assemble continuous insight
from data in motion



Peter Nicholls
Ankit Pasricha
Roger Rea
Norbert Schulz
Chuck Ballard
Oliver Brandt
Bharath Devaraju
Daniel Farrell
Kevin Foster
Chris Howard
Tetsuya Shimada
John Thorson
Sandra Tucker
Robert Uleman



International Technical Support Organization

**IBM InfoSphere Streams: Accelerating
Deployments with Analytic Accelerators**

February 2014

Note: Before using this information and the product it supports, read the information in “Notices” on page ix.

First Edition (February 2014)

This edition applies to Version 3.0.0 of IBM InfoSphere Streams (Product Number 5724-Y95).

© Copyright International Business Machines Corporation 2014. All rights reserved.

Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	ix
Trademarks	x
Preface	xi
Authors	xiii
Now you can become a published author, too!	xix
Comments welcome	xx
Stay connected to IBM Redbooks	xx
Chapter 1. Introduction	1
1.1 The challenge of getting started	2
1.1.1 Accelerators and toolkits	4
1.2 Using this book	8
Chapter 2. Application programming using Streams Studio	9
2.1 SPL Graphical Editor	10
2.1.1 Adding operators to the graph	11
2.1.2 Connecting operators	13
2.1.3 Layout options and Outline view	14
2.1.4 Defining global types and schemas	16
2.1.5 Editing operator details	19
2.1.6 Generating SPL code	32
2.2 Application development use cases	34
2.2.1 Design: Sketching an application	34
2.2.2 Implementing an application	37
2.2.3 Composites: Reusable subgraphs	39
2.2.4 Making changes: Refactoring	43
2.2.5 Program understanding	44
Chapter 3. Visualizing stream data	51
3.1 Stream data, views, and charts	52
3.2 How data visualization works	53
3.3 Use cases	54
3.3.1 Debugging using Streams Studio	54
3.3.2 Application monitoring using Streams Console	64
3.3.3 Hints and tips	83

Chapter 4. Analytics entirely with SPL	85
4.1 Volume, variety, and velocity	86
4.2 Model view controller (MVC), and Streams	88
4.3 Flow rate sensor: Example application we create	90
4.3.1 Sample output from our flow rate sensor	91
4.3.2 Flow rate sensor: First ingest, heartbeat generator	92
4.3.3 Flow rate sensor: Actual ingest, parsing semi-structured data	94
4.3.4 Flow rate sensor: React phase (simulated)	97
4.3.5 Flow rate sensor: User defined functions, (read-only) maps	98
4.3.6 Flow rate sensor: Analyze, read-write maps, windows, other	103
4.4 Conclusion, how to proceed	109
Chapter 5. Streams and DataStage integration	111
5.1 Introduction to Streams processes	112
5.2 Runtime architecture	113
5.3 Metadata integration	115
5.3.1 Integration Setup Overview	115
5.4 Sample application	117
5.4.1 Importing Streams certificate to DataStage	117
5.4.2 Streams-to-DataStage application	118
5.5 DataStage Job design practices	133
Chapter 6. Streams integration with IBM BigInsights	137
6.1 Streams and big data challenges	138
6.1.1 Application scenarios	139
6.1.2 Large scale data ingest	141
6.1.3 Bootstrap and enrichment	141
6.1.4 Adaptive analytics model	141
6.1.5 Complex social and entity related analysis	143
6.1.6 Application development	146
6.1.7 Application interactions	147
6.1.8 Enabling components	148
6.2 BigInsights summary	149
Chapter 7. Complex event processing	151
7.1 The role of the CEP Toolkit	152
7.1.1 Adding the CEP Toolkit to your build path	152
7.2 Stock price watch example	155
7.2.1 First iteration of stock price watch application	156
7.2.2 Second iteration of stock price watch application	159
7.2.3 Third iteration of stock price watch application	160

Chapter 8. WebSphere MQ, XMSSource, XMSSink	163
8.1 WebSphere MQ Server, Message Service Client installation	164
8.2 Making the WebSphere MQ resident objects	165
8.3 Setting Streams environment variables, and adding toolkits.	171
8.4 The two Streams applications	175
8.5 Using JMS adapters from the Messaging Toolkit	181
8.5.1 Example use cases for the JMS adapters.	182
8.5.2 Installing and configuring of WebSphere MQ	183
8.5.3 Installing and configuring for Apache ActiveMQ	188
8.5.4 Compiling and running sample applications	190
8.5.5 Sample applications	197
8.5.6 Verifying the results.	199
Chapter 9. XML, XMLParse, XPath, and xquery	205
9.1 Scenario 1: Flat, single-tier XML, XMLParse	206
9.2 Scenario 2: Multitiered (list data).	212
9.2.1 Second iteration, XMLParse	221
9.2.2 Third iteration, XMLParse	224
9.2.3 Fourth (final) iteration, XMLParse	228
9.3 The spl-schema-from-xml utility, generating XMLParse code.	232
9.4 The xquery() function, including filter	236
9.5 CDATA: Topic that is not covered in detail	239
Chapter 10. Geospatial Toolkit	241
10.1 Concepts	242
10.1.1 Moving objects and location-based services.	242
10.1.2 Geospatial concepts and operations.	243
10.2 Toolkit organization	251
10.2.1 Namespaces	252
10.2.2 Types and enumerations.	252
10.2.3 Constructor functions	255
10.2.4 Accessor functions	256
10.2.5 Spatial production functions	257
10.2.6 Spatial relationship functions	259
10.2.7 Validation of input arguments	261
10.2.8 Metric conversion functions.	265
10.3 A location-based scenario: tracking vehicles	266
10.3.1 A vehicle simulator	266
10.3.2 Geofencing: detecting entry and exit.	269
10.3.3 Predictive geospatial analytics	274
10.4 Conclusion.	275

Chapter 11. TimeSeries Toolkit	277
11.1 Basics of time series analysis	278
11.1.1 Time series patterns	278
11.1.2 Detecting patterns using the TimeSeries Toolkit	282
11.2 Time series representation and operators overview	282
11.2.1 Time series representation	283
11.2.2 Control signals	286
11.2.3 Types of operators and overview	287
11.3 Preprocessing operators	291
11.3.1 ReSample operator	291
11.3.2 TSWindowing operator	294
11.3.3 IncrementalInterpolate operator	298
11.4 Analysis operators	299
11.4.1 DSPFilter operator	300
11.4.2 Fast Fourier Transform	303
11.4.3 Discrete Wavelet Transform operator	306
11.4.4 Seasonal Trend Decomposition operator	309
11.4.5 CrossCorrelate operator	311
11.4.6 Normalize operator	312
11.4.7 FunctionEvaluator operator	314
11.4.8 Distribution operator	317
11.5 Modeling operators	320
11.5.1 HoltWinters operator	321
11.5.2 ARIMA operator	324
11.5.3 FMPFilter operator	326
11.5.4 Kalman operator	328
11.5.5 GAM operators	331
11.5.6 GMM operator	337
11.5.7 LPC operator	339
11.5.8 VAR operator	341
11.5.9 RLSFilter operator	342
11.6 Time series functions	346
11.6.1 The generator functions	346
11.6.2 The Crosscorrelate function	347
11.6.3 The convolve function	348
11.6.4 The rms function	348
Chapter 12. Developing Java primitive operators	349
12.1 Operator lifecycle	350
12.2 Threading in a Java operator	351
12.3 Creating a simple operator	352
12.3.1 Operator code layout	352
12.3.2 Defining the operator model	353

12.3.3	Implementing the operator in Java	357
12.3.4	Compiling the operator code	366
12.3.5	Testing the operator code	367
12.3.6	Creating an SPL application	370
12.3.7	Adding custom metrics	372
12.3.8	Implementing a tuple consumer operator	376
12.4	Java development using Streams Studio	377
Chapter 13. Text Analytics, AQL		
13.1	Overview text analytics, by example	381
13.2	Installing and configuring text analytics tools	385
13.2.1	Installing text analytics tools from Streams Studio installation	386
13.2.2	Installing text analytics tools from a BigInsights install	392
13.2.3	Configuring your Streams project, add a BigInsights project	396
13.3	First AQL example, Apache HTTP log file	408
13.3.1	Improving the first example, Apache HTTP log file	414
13.4	Guided team-based AQL and using AQL tools	432
13.5	Regular expressions (regex)	436
13.5.1	AQL tools to aid in regex development and debug	438
13.5.2	Regex directly inside Streams; no AQL required	443
13.6	Additional AQL objects and techniques	451
13.6.1	The kitchen sink AQL script and Streams application	464
13.7	Topics not covered	471
Chapter 14. IBM Accelerator for Telecommunications Event Data Analytics V1.2		
14.1	Overview of TEDA	474
14.2	Installing TEDA	475
14.3	Understanding concepts and terms	477
14.3.1	Application components	477
14.3.2	Application infrastructure	479
14.3.3	Configuration	480
14.3.4	Fault tolerance	481
14.4	Customizing TEDA	483
14.4.1	Workflow overview	483
14.4.2	Preparation	483
14.4.3	Defining the sample use case	484
14.4.4	Exercise 1: Basic setup	488
14.4.5	Exercise 2: Writing to the database	501
14.4.6	Exercise 3: Adding an aggregation operator	504
14.5	Conclusion	510

Chapter 15. SPSS Toolkit	511
15.1 An overview of InfoSphere Streams and SPSS	512
15.1.1 Integrating InfoSphere Streams and SPSS.....	512
15.1.2 Roles and terminology	512
15.1.3 Example development process.....	513
15.2 Coordinating Data Analyst and Streams developer efforts	514
15.3 Building the predictive models.....	515
15.4 Configuring the SPSSScoring operator.....	521
15.5 Summary.....	526
Appendix A. Additional material	527
Locating the web material	527
Using the web material.....	528
Related publications	529
IBM Redbooks	529
Online resources	529
Help from IBM	530

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Trademarks

IBM, the IBM logo, and [ibm.com](http://www.ibm.com) are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at <http://www.ibm.com/legal/copytrade.shtml>

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

BigInsights™	Informix®	solidDB®
Cognos®	InfoSphere®	SPSS®
DataStage®	QualityStage®	System z®
DB2®	Rational®	Tivoli®
developerWorks®	Redbooks®	Unica®
Global Business Services®	Redbooks (logo)  ®	Velocity™
IBM®	Smarter Planet®	WebSphere®

The following terms are trademarks of other companies:

Netezza, and N logo are trademarks or registered trademarks of IBM International Group B.V., an IBM Company.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java, and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Other company, product, or service names may be trademarks or service marks of others.

Preface

An information revolution has been underway for more than 40 years. A key event in that revolution was the development of the relational database, which changed the world by simplifying data access and management for many simultaneous users. Now, another innovation has started a similar change. IBM® InfoSphere® Streams (“Streams”) is radically transforming the information industry by assembling continuous insights from data in motion, that is, from data that might not have yet been stored. This new technology was developed to handle the issues of high volume, velocity, and variety of data, which we call *big data*.

This approach, coupled with relational databases, enables higher expectations regarding the kinds of applications that can be deployed in the enterprise. In much the same way that technology, such as the assembly line, was the culmination of innovation in the Industrial Revolution, Streams is becoming the “assembler of continuous insight” of the Information Revolution. Streams captures and uses data in motion, or streaming data, by processing it as it passes through enterprise applications. With traditional data processing, you typically run queries against relatively static sources of data, which then provides you with a query result set for your analysis. With stream computing, you execute a process that continually analyzes data as it becomes available, with results updated continuously.

Based on the requirements, the analytic results can be stored for further processing or can be acted on immediately and not stored. As a result, operations can be scaled out to more servers or cores because each core can focus on its own operation in the overall sequence. Like an assembly line, steps that take longer can be further segmented into sub-steps and processed in parallel. Because the data is processed in an assembly-line fashion, high throughputs can be achieved, and because the data is processed as it arrives, initial results are available much sooner and the amount of inventory, or enrichment data, that needs to be readily available is decreased. IBM developed Streams to address the emerging requirement for more timely results coupled with greater volumes and varieties of data. Streams is the assembly line for insights of the Information Revolution, complementing the relational databases that began the revolution. What is new in Streams is the higher level of abstraction and the ability to make assembly lines available using a cluster or processing nodes.

Streams also includes many elemental building blocks, such as text analytics, geospatial analytics, machine learning, predictive analytics, statistical analytics, data mining, and relational analytics, so that enterprises can more easily build

these assembly lines for business insight. We believe this approach will become widely used, enable us to meet the coming business challenges, and, as with the manufacturing assembly line, will transform the information industry. Streams provides an execution platform and services for user-developed analytic applications that ingest, filter, analyze, and correlate potentially massive volumes of continuous streams of data. These analytic applications can be changed over time, without interrupting processing, based on intermediate results or application requirements. New applications can be dynamically added to extend existing applications, allowing multiple users to access and analyze this data, much like multi-user relational database management systems. Developing applications in this type of environment differs significantly from traditional approaches because of the continuous flow nature of the data.

In this IBM Redbooks® publication, we describe the toolkits and accelerators available with Streams to build real-time analytic applications. The Streams environment, architecture, and programming paradigms covered in *IBM InfoSphere Streams: Assembling Continuous Insight in the Information Revolution*, SG24-7970 still apply to the latest versions of Streams. This book is intended for professionals who require a deeper understanding of how to process high volumes of streaming data with the wide variety of available analytic tools. These tools include the drag-and-drop development environment and visualization capabilities, adapters to HDFS, IBM MQ and IBM InfoSphere DataStage®, and analytic toolkits for geospatial, time series (including machine learning and predictive analytics), and the Accelerators for Social Data Analytics and Telecommunications Event Data Analytics.

Imagine the business impact of being able to analyze more data at rates faster than ever before. What if it suddenly became cost effective to analyze data that was too time-sensitive or expensive to analyze in the past? You could more consistently identify new market opportunities before competitors, and be agile enough to respond to changes in the market.

This type of processing offers tremendous potential. Begin the evolution in your environment today and start to gain the business advantage.

Authors

This book was produced by the following team of product and solution specialists from around the world working with the International Technical Support Organization in San Jose CA.



Chuck Ballard is a Project Manager at the International Technical Support organization, in San Jose, California. He has over 35 years of experience, holding positions in the areas of Product Engineering, Sales, Marketing, Technical Support, and Management. His expertise is in the areas of database, data management, data warehousing, business intelligence, and process re-engineering. He has written extensively about these subjects, taught classes, and presented at conferences and seminars worldwide. Chuck has both a Bachelor's degree and a Master's degree in Industrial Engineering from Purdue University.



Oliver Brandt is a Software Developer with the IBM InfoSphere Streams team in Berlin, Germany, since 2011. He joined IBM in 2007 and has worked on several telecommunication related projects since then. Prior to joining IBM, Oliver worked for an international telecommunications equipment manufacturer for more than 12 years, where he developed software for mobile network elements such as home location registers. Oliver studied Computer Science at Technische Universität Berlin.



Bharath Devaraju has been with IBM since 2009. He has worked as a Developer on several first-of-a-kind and innovative projects such as Data Quality Rule Set for India, data as a service (IBM Cloud Computing offering) and InfoSphere Streams TimeSeries Toolkit. Bharath is an IBM DataStage and IBM QualityStage® certified professional and IBM developerWorks® contributing author. He has also worked extensively on customer proof of concept documents (POCs) and assists with pre-sales activities for growth markets.



Daniel Farrell is a Principle Software Engineer on the IBM North America Hadoop proof-of-concept team, with focus in areas such as real-time analytics, R, Map/Reduce, Hive, and HBase. In 1985, Daniel began working with IBM Informix® software products at a national retail and catalog company, a job that would set his direction for the next thirty or so years. He joined IBM as a result of the acquisition of Informix Software. Daniel has a Master's degree in Computer Science from Regis University and is pursuing a Ph.D in Computer Science. Both Daniel's thesis and dissertation concern research on fourth generation programming languages, analytics, business application delivery, and staff development.



Kevin Foster is a Big Data Solutions Architect at the IBM Silicon Valley Lab in San Jose, California. He has worked in the database software area for over 30 years. In the first phase of his career, Kevin worked as a system integrator building custom applications for a number of major corporations. He began the next phase of his career working in a variety of roles in software product development, engineering management, technical marketing, and consulting. His current focus is on IBM Business Partner and customer technical enablement on the IBM InfoSphere Streams product. Kevin has a Bachelor's degree in Mathematics from California State University Stanislaus and a Master's degree in Computer Science from Stanford University.



Chris Howard is an IBM Executive IT Specialist and is currently the Technical Leader for big data across IBM Growth Markets. He has been with IBM since 1998, having held a variety of regional, European, and worldwide roles across IBM Software Group and IBM Research. Chris has been involved with big data since before the commercial launch of IBM InfoSphere Streams in 2009, and was responsible for the initial concept, launch, and management of the EMEA Stream Computing Center of Competence, opened to serve as a hub of customer support, research, and advanced testing for stream-computing applications. Chris is originally from the UK, is a Chartered Fellow of the British Computer Society, and is currently on a two year assignment with IBM Australia.



Peter Nicholls is a Senior Technical Staff Member and the InfoSphere Streams Tooling Architect and Development Manager with IBM Software Group. He has 25 years experience in the areas of language runtimes and tooling with IBM. His expertise is in remote development tooling and debugging and has contributed to many of the IBM remote developer products for IBM platforms. As a member of the InfoSphere Streams product team, his focus is on simplifying the development of Streams applications and providing a superior developer experience.



Ankit Pasricha is Advisory Software Developer in the InfoSphere Streams SPL Runtime team in Toronto, Canada. Previously, he worked for several years as the Technical Team Lead of the TPF Toolkit and the IBM Rational® for System z® C++ teams. Over the last 10 years, he has developed extensive expertise in the Java programming language and Eclipse plug-in development. Ankit has a Bachelor's degree in Computer Engineering from the University of Toronto.



Roger Rea is the InfoSphere Streams Product Manager in the IBM Software Group. Previously, he has held a variety of sales, technical, education, marketing, and management positions with IBM, Skill Dynamics, and Tivoli® Systems. He has received four IBM 100% Clubs awards, one Systems Engineering Symposium award, numerous Excellence awards, and Teamwork awards. Roger earned a Bachelor of Science in Mathematics and Computer Science, Cum Laude, from the University of California at Los Angeles (UCLA), and a Master's Certificate in Project Management from George Washington University.



Norbert Schulz is a Software Developer with the InfoSphere Streams team in Berlin, Germany, which he joined in 2011. He has been with IBM since 2007 when he was working on software for managing the configuration of VoIP switches. Prior to that, Norbert worked for huge international telecommunications equipment providers giving him over fifteen years of experience in the field. Norbert earned a Diplom Ingenieur degree (comparable to the Master of Engineering) in Electrical Engineering from Technische Universität Berlin, Germany in 1997.



Tetsuya Shimada is a Senior Technical Architect in the Information Management Center of Excellence. He came to IBM with the Ascential Software acquisition and has been working in the data integration field for over 11 years. He has a Bachelor's degree in Electric Engineering from Nihon University. His areas of expertise include QualityStage, Parallel Framework Engine, Information Server Grid, and big data products.



John Thorson is a Senior Software Engineer in the IBM Software Group, SPSS®. He joined IBM with the acquisition of SPSS. John is on the SPSS Architecture team and has expertise in SPSS Collaboration and Deployment Services and the SPSS Modeler group of products. He is the technical lead of the SPSS Analytical Toolkit for InfoSphere Streams and has served as the technical Customer Advocate on many SPSS and Streams engagements.



Sandra Tucker is an Executive IT Specialist with the IBM Software Group Information Management Lab Services Center of Excellence, concentrating on services for emerging products and integrated solutions. She has been successfully implementing business intelligence and decision support solutions since 1990. Sandra joined IBM through the Informix Software acquisition in July, 2001 and is currently focused on the IBM InfoSphere Streams computing product and Data Warehousing solutions. Her experience includes work in areas such as healthcare's profile health and cost management, retail's sales analysis and customer profiling, telecommunication's churn analysis, and market segmentation.



Robert Uleman is a Streams expert on the Worldwide Information Management Technical Sales Acceleration team. Robert has been training IBMers and IBM Business Partners on Streams, and working on Streams-based customer engagements, since 2010. Previously, he focused on geospatial databases (Informix and IBM DB2®), both in the field and in product development. He has deep expertise in spatiotemporal computing and GIS, time series analysis, and image processing. Robert holds Master's degrees in Exploration Geophysics (Stanford University) and Applied Physics (Delft University of Technology). He is based in California, USA.

Other Contributors

In this section, we thank others who contributed to this book, in the form of written content, subject expertise, review, and support.

For written content

- Alain Biem: Big Data/Streaming Analytics R&D Lead and Research Scientist, IBM Research, Yorktown Heights, NY US.
- Samantha Chan: InfoSphere Streams Studio Development, IBM Software Group, Information Management, Markham, ON Canada.
- Rich Harken: Big Data Architect, Distinguished IT Specialist, IBM Software Group, Information Management, Southfield, MI US.
- Dattaram Aswathanarayana Rao: InfoSphere Streams Software Developer, IBM Software Group, Information Management, Bangalore, KA India.
- Manasa Rao: InfoSphere Streams Software Developer, IBM Software Group, Information Management, Bangalore, KA India.

For subject expertise and content review

- Naveen Dronavalli: Technical Solution Architect, Data Integration and Warehousing, IBM Software Group, Information Management, Dallas, TX US.
- Kevin Foster: InfoSphere Streams and IBM BigInsights™ Enablement, IBM Software Group, Enterprise Content Management, Silicon Valley Lab, San Jose, CA US.
- Anne Jane Gray: Senior Project Manager, Information Management Services, IBM Software Group, Information Management, Pittsburgh, PA US.
- Jeff Leake: Implementation Technical Lead, Technical Solution Architect, IBM Software Group, Information Management, Cleveland, OH US.
- Michael Nobles: Software Sales Executive, IBM Sales and Distribution, IBM Software Sales, Atlanta, GA US.
- Elizabeth Peterson: Solution Architect/Principal, SWG Lab Services, IBM Software Group, Information Management, Seattle, WA US.

Vishal Puri: Senior Managing Consultant, Big Data, Watson and Content Analytics, IBM Global Business Services®, Dallas, TX US.

Brandon Swink: Architect, InfoSphere Streams, Information Management Lab Services, IBM Software Group, Information Management, Dallas, TX US.

Chris White: Solutions Implementation Manager, IBM Software Group, Information Management, Columbia, SC US.

From the International Technical Support Organization

Mary Comianos: Publications Management
Ann Lund: Residency Administration
Ella Buslovich: Graphics Support
Diane Sherman: Editor

Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an ITSO residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks publications in one of the following ways:

- ▶ Use the online **Contact us** review Redbooks form found at:
ibm.com/redbooks
- ▶ Send your comments in an email to:
redbooks@us.ibm.com
- ▶ Mail your comments to:
IBM Corporation, International Technical Support Organization
Dept. HYTD Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400

Stay connected to IBM Redbooks

- ▶ Find us on Facebook:
<http://www.facebook.com/IBMRedbooks>
- ▶ Follow us on Twitter:
<http://twitter.com/ibmredbooks>
- ▶ Look for us on LinkedIn:
<http://www.linkedin.com/groups?home=&gid=2130806>
- ▶ Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:
<https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm>
- ▶ Stay current on recent Redbooks publications with RSS Feeds:
<http://www.redbooks.ibm.com/rss.html>



Introduction

In this chapter, we give you some background about the components of InfoSphere Streams that can help you accelerate your time-to-value. This is meant to be a practitioner's guide with information about what is available and how to leverage it using examples and exercises that demonstrate practical real world application of the components. We introduce the concepts of accelerators and toolkits and where to find them. In addition, we introduce functional components that can also be used to get you started quickly. Subsequent chapters go deeper into what each component, toolkit, accelerator, and guideline is and how you can use each to begin your journey to a deeper understanding of big data with InfoSphere Streams.

In exploring the challenge of getting started, the objective of this chapter is to provide insight into the following key questions and concepts:

- ▶ What are accelerators and toolkits and why do we need them?
- ▶ What are the currently available accelerators and toolkits for InfoSphere Streams and where can you find them?
- ▶ How can you use these tools to accelerate your development of an InfoSphere Streams application?
- ▶ How can you use this book as a guide to leveraging the components for accelerating development?

1.1 The challenge of getting started

By now most of us have heard the big data messages:

- ▶ In the last couple of years, the level of instrumentation in all aspects of our business and personal lives has grown exponentially.
- ▶ The proliferation of this instrumentation is driving a huge increase in the amount of data that is available for analysis.
- ▶ The key differentiator for successful businesses in the future will be their ability to tap into this new vast source of information and leverage it quickly and effectively to make strategic business decisions.

New technologies have appeared in the market, targeted specifically at using this new information resource both from the perspectives of data at rest and data in motion. The volume and velocity increase in the data available for analysis is at a rate that makes it difficult to know what you need to store or even if it can be stored. As such, most of us have grown to accept the value of analyzing data and taking actions quickly, even while data is still in motion. You might even be aware of key analytic solutions that are completely or partially comprised of stream computing applications. Stream computing brings the power to sift through the massive volume of data before committing it to storage and filter it down to the key analytic components. Stream computing also brings the ability to enrich the data as it is processed and take action in time to make a difference. Leveraging data in motion can be a critical entry point into the world of big data analytics.

So, if big data is such a valuable asset and stream computing is your ticket of admission in the world of big data, how best do you get started?

Many of us can envision uses for this vast amount of information, tasks we have long put aside because the realities of processing such amounts of data with conventional technology products of that time made any useful analysis in an appropriate time frame impossible. This new era of big data has spawned new technologies that have us rethinking the “possible.” One of the technologies that has risen to the task of helping organizations use big data in motion is IBM InfoSphere Streams. Still, selecting a product is only the beginning. Using that product to make your use cases come to life is not always an obvious path.

Consider for a moment, the following scenario. Baxter is a student with an opportunity to prepare a mixed media piece for the state-wide art competition. He has successfully competed in many art competitions but not in mixed media, but still he has a vision of what he wants to create. He even knows he needs paint and other supplies to make it happen, although he is not entirely sure what all of those supplies might be. Baxter asks his sister (because she has a car) to take him to an art supply store, where he finds himself confronted by the vast array of

supplies he might use. The choices are so vast he does not quite know how to begin to choose. Even with his past success as an artist and a clear vision of what he wants to create, he just does not know what he needs to begin. His sister is pressuring him to stop taking so long and make a decision, and his frustration is mounting. Trying to translate his vision into the pieces he needs to create it is too daunting a task in this environment, and so he leaves without purchasing any supplies, and therefore unable to start. However, he has not given up hope yet, because he really wants to compete. He goes to a second art supply store where he finds the materials are organized into categories that align with what he wants to accomplish. Based on the successful experience of others in the field, this store presents proven palettes of useful material for each kind of artistic endeavor. That organization allows him to quickly and confidently choose the supplies he needs to turn his vision into a reality. He creates his masterpiece and is handsomely rewarded by winning the art contest.

Developers getting started in stream computing are much like Baxter. They know how to code (most are highly skilled in application development using other paradigms) and they even generally know what they need. Faced with new technologies, they find themselves in that “unorganized store” unable to decide how to get started and they are under the pressure of tight deadlines and high expectations. They do not need someone to paint the picture for them, only a little edge of organizing the components into categories that align with what they are trying to achieve. In IBM InfoSphere Streams that organization is provided by the accelerators and toolkits component of the product.

1.1.1 Accelerators and toolkits

As you see in Figure 1-1, accelerators and toolkits are a key component of IBM Big Data platform. Accelerators and toolkits facilitate development of applications for key analytics for business and social initiatives toward a IBM Smarter Planet®.

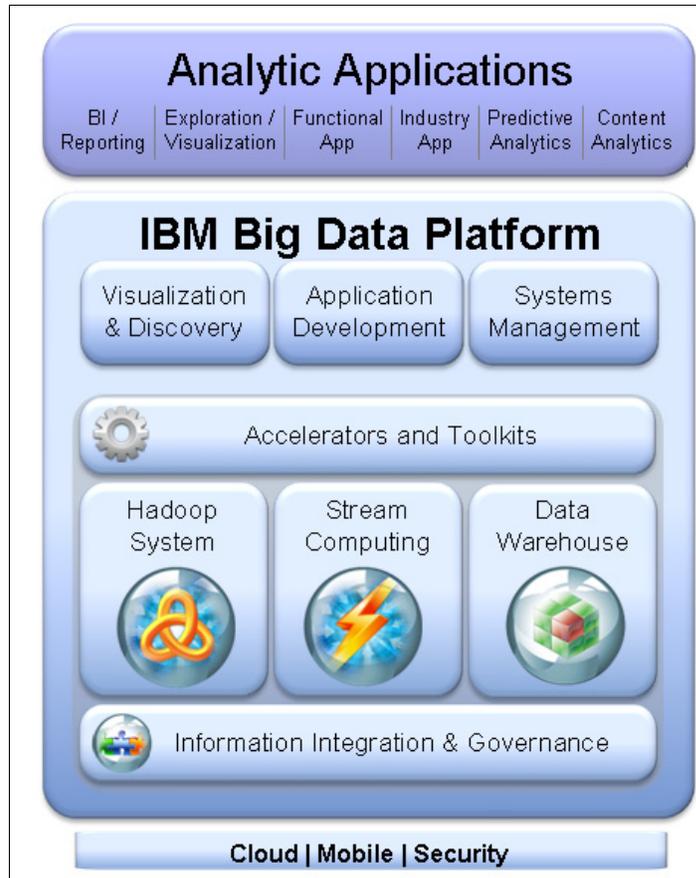


Figure 1-1 IBM Big Data platform

They provide that source of proven components that application developers can use to develop those critical analytic applications against big data. IBM has taken a holistic approach toward these assets that allows the developer to employ the same logical component across the applications developed for data at rest and data in motion. We believe this is an important differentiator. More than just providing application consistency, it eases the learning curve and shortens the time to value of application development.

From the time of their introduction, accelerators and toolkits have been one of the three key components of the IBM InfoSphere Streams product and the heart of the product's focus on providing sophisticated analytics, as shown in Figure 1-2.

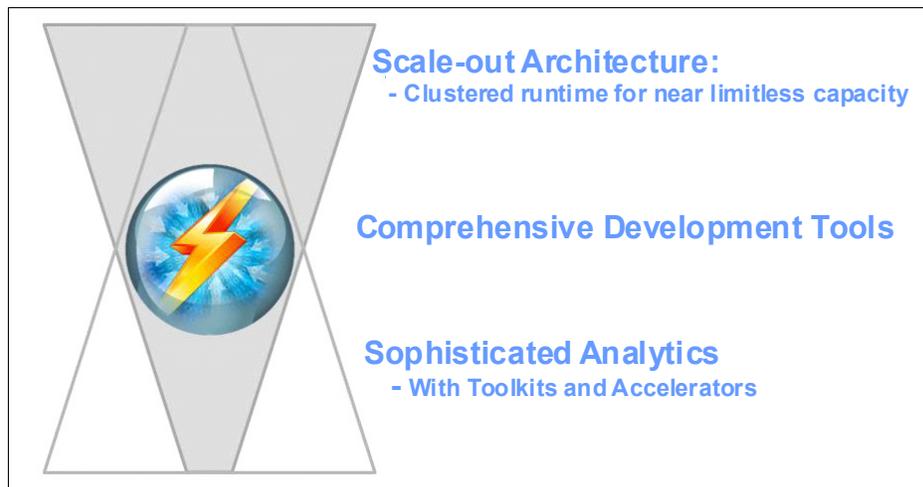


Figure 1-2 IBM InfoSphere Streams

Simply, accelerators and toolkits are collections of assets that facilitate the development of a solution for a particular industry or functionality. For Streams, these represent critical functionality that has proven useful each time. Honed by experience in delivering streaming applications, these assets are often born in IBM Research. The accelerators and toolkits can consist of simple operators and adapters, from complex, as sample parts, to whole applications. Because the groupings can be by industry (such as financial or telecommunications) or functionality (such as data mining or complex event processing (CEP)), and some assets can be useful to an industry and also a functionality, individual components can appear in more than one accelerator or toolkit. In addition, the collections are grouped based on prevalent experience, but could also be beneficial to another industry or functionality.

Figure 1-3 lists the current organization of accelerators and toolkits that are included with IBM InfoSphere Streams product.

▪ Big Data	▪ Internet
▪ Complex Event Processing	▪ Mining
▪ Database	▪ Messaging
▪ Data Explorer	▪ Text
▪ DataStage	▪ Time Series
▪ Finance	▪ Telco
▪ SPSS	▪ Machine Data
▪ Geospatial	▪ Social Data

Figure 1-3 Sophisticated Analytics provided by Accelerators and Toolkits

As with the more organized art supply store, aligning the components with their common uses helps developers more easily know where to find what they need and get started without frustrating delays. The understanding developers gain by applying the components in the most common scenario guides these developers toward understanding the value of the components, enabling them to use that knowledge to expand upon their uses.

The primary goal of the accelerators and toolkits is not to get you to think about new ways of doing tasks, but enabling you to do them. These components are focused on enabling you to achieve the goal of creating the solutions that use data in motion to make a difference to your business or your world. These assets are provided to give the new Streams developer working examples of what the language can do and how to program a working application to implement those capabilities. As a developer, these assets can be used in the following ways:

- ▶ As an example of what types of applications can be developed in Streams
- ▶ As an example of how a Streams application can be structured
- ▶ As a template to begin developing your own application
- ▶ To understand how to program an application in Streams
- ▶ To augment or include functionality into your Streams application

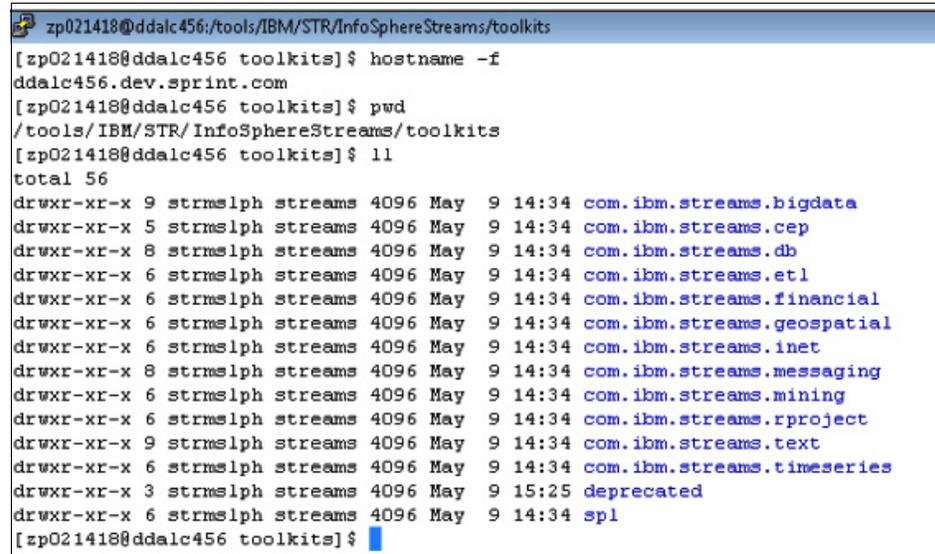
These assets do not include any implicit warranties. They are not guaranteed to provide optimal performance for your application or any specific environment. Performance tuning is not considered to be inherent when you use these assets alone or in conjunction with an application you have developed.

The accelerators and toolkits that are available with the product must be separately downloaded from the same site you used to download the software.

They are optional and each can be downloaded and installed as needed. The installation instructions for each are provided with the accelerators or toolkits.

The default toolkits that are installed with the product are installed under the toolkits directory, as in the following example (see Figure 1-4):

/tools/IBM/STR/InfoSphereStreams/toolkits



```
zp021418@ddalc456:/tools/IBM/STR/InfoSphereStreams/toolkits
[zp021418@ddalc456 toolkits]$ hostname -f
ddalc456.dev.sprint.com
[zp021418@ddalc456 toolkits]$ pwd
/tools/IBM/STR/InfoSphereStreams/toolkits
[zp021418@ddalc456 toolkits]$ ll
total 56
drwxr-xr-x 9 strmslph streams 4096 May  9 14:34 com.ibm.streams.bigdata
drwxr-xr-x 5 strmslph streams 4096 May  9 14:34 com.ibm.streams.cep
drwxr-xr-x 8 strmslph streams 4096 May  9 14:34 com.ibm.streams.db
drwxr-xr-x 6 strmslph streams 4096 May  9 14:34 com.ibm.streams.etl
drwxr-xr-x 6 strmslph streams 4096 May  9 14:34 com.ibm.streams.financial
drwxr-xr-x 6 strmslph streams 4096 May  9 14:34 com.ibm.streams.geospatial
drwxr-xr-x 6 strmslph streams 4096 May  9 14:34 com.ibm.streams.inet
drwxr-xr-x 8 strmslph streams 4096 May  9 14:34 com.ibm.streams.messaging
drwxr-xr-x 6 strmslph streams 4096 May  9 14:34 com.ibm.streams.mining
drwxr-xr-x 6 strmslph streams 4096 May  9 14:34 com.ibm.streams.rproject
drwxr-xr-x 9 strmslph streams 4096 May  9 14:34 com.ibm.streams.text
drwxr-xr-x 6 strmslph streams 4096 May  9 14:34 com.ibm.streams.timeseries
drwxr-xr-x 3 strmslph streams 4096 May  9 15:25 deprecated
drwxr-xr-x 6 strmslph streams 4096 May  9 14:34 spl
[zp021418@ddalc456 toolkits]$
```

Figure 1-4 Toolkits directory

Other accelerators and toolkits can be installed in any directory, but keeping an organizational standard will help you know where things are and facilitate setup.

1.2 Using this book

The remainder of this book can help you gain the most benefit from the accelerators and toolkits, and also the general functionality in IBM InfoSphere Streams to enable developers to accelerate development time. Each chapter focuses on an individual component of the product design to decrease the time to value. The chapters do not necessarily build upon one another. The book stands as a practitioner's guide to leveraging the product for your own use cases and as a trusted reference that you can refer to, as you continue to grow in proficiency as a developer. Some of the chapters assume a base level of knowledge in that area and this book is not meant as a replacement for the information that can be found in the product's documentation or installation instructions. The examples can help you get started by doing tasks; doing them can take the mystery out of how to begin building streaming applications for your needs.

The book begins with some of the standard functionality of InfoSphere Streams to help you get started quickly. Then, the following chapters can help you beyond getting started:

- ▶ Chapter 2 introduces the “drag and drop” editor feature that helps you assemble a Streams application graphically. You will be able to see the lines of Streams Processing Language (SPL) code that are generated, helping you become familiar with how a Streams application is constructed.
- ▶ Chapter 3 introduces the visualization functionality of Streams. After you construct a Streams application and get it running, you can use that visualization functionality to see the results.
- ▶ Chapter 4 has examples of developing analytical application functionality using only the standard SPL operators.
- ▶ Chapter 5 and subsequent chapters introduce and provide examples of how to use one of the accelerators or toolkits. These components are not presented in any particular order of significance, so you may read the chapters in any order. Not all accelerators and toolkits are covered, but by the time you finish this book, you can have a good idea of how to get started with several of them; that knowledge can help you use others.

Neither this book nor the accelerators code the applications for you. As with our friend, Baxter, you generally know how to paint your picture. Consider this book your floor plan to that organized art supply store and a key guide of how best to get the most from the supplies you choose.



Application programming using Streams Studio

In this chapter, we describe how to create an InfoSphere Streams application by using Streams Studio. Streams Studio is the integrated development environment for InfoSphere Streams. The focus of this chapter is in application development, the graphical editor, and related features within Streams Studio.

The first part of the chapter describes features and functions of the Streams Processing Language (SPL) Graphical Editor. The graphical editor is used to construct a Streams application within a graphical context.

The second part of the chapter describes specific use cases within an application development workflow. The use cases build upon the functional information by providing specific uses for the functions described in the first part of the chapter.

2.1 SPL Graphical Editor

Streams applications are, by definition, a directed flow graph where data streams flow through a series of operators that perform analytic operations and produce new streams, which flow to subsequent operators and eventually out of the application. These flows form an assembly line of continuous analytic processing.

Use the SPL Graphical Editor to create and edit a Streams application within a directed flow graph context. The application is rendered as a left-to-right directed flow graph where the nodes represent operators and the edges represent the streams that connect the operators. This graphical representation provides a more natural and simpler view of the application, which can then be used to edit the details of the application.

Figure 2-1 shows the basic components of the graphical editor view. One of the InfoSphere Streams sample applications, Vwap, is opened in the graphical editor. The palette is on the left, the canvas is on the right. To use the graphical editor, select objects in the palette and drag them to the canvas.

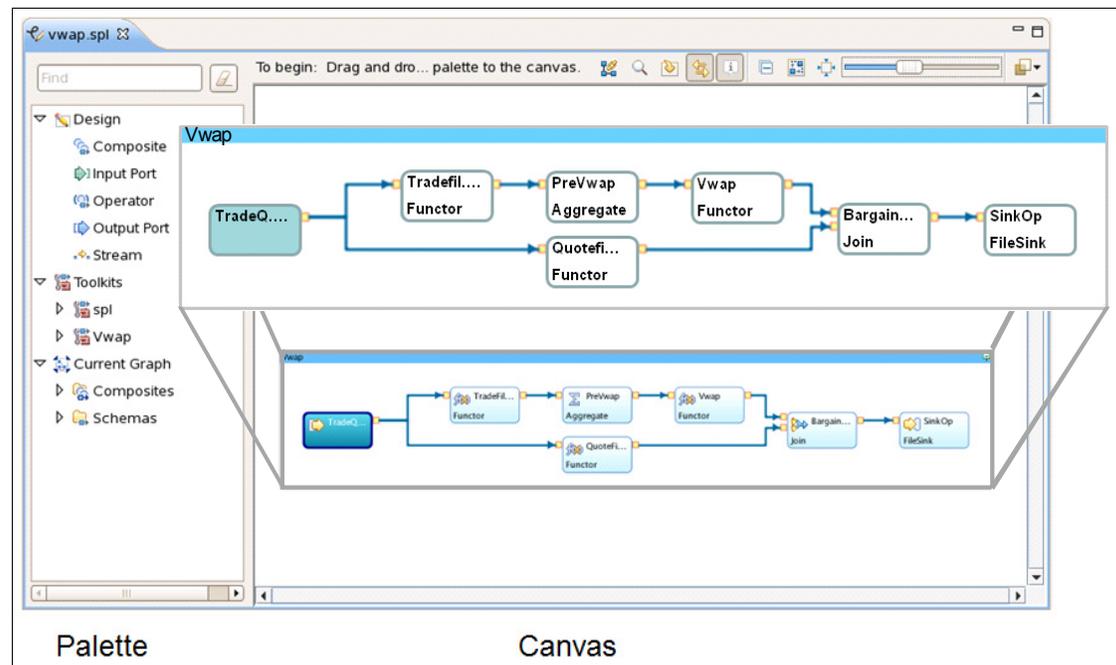


Figure 2-1 Basic components of the graphical editor

2.1.1 Adding operators to the graph

To add an operator to the graph, first select the operator from the palette, then press the left mouse button while you drag the operator to the canvas and drop the operator. As you drag an operator over elements in the graph, the elements change color based on whether the drop-point is valid or not.

- ▶ Green indicates a valid drop location.
- ▶ Yellow indicates a valid drop location, but might result in compiler errors or warnings.
- ▶ Red indicates that dropping is not allowed at this location.

The following drop points are valid for operators:

- ▶ In a composite definition, which adds the operator to the composite operator.
- ▶ On top of an existing operator, which replaces the existing operator with a new one. The SPL clauses for the existing operator are overwritten. The ports and schemas remain unchanged.
- ▶ In a blank part of the canvas, which creates a composite operator definition containing the operator.

In Figure 2-2 on page 12, the palette shows the operators and schema types that you can use to build your application. By default, the palette on the left shows all the toolkits that are listed as direct dependencies for the project. You might also see all the available toolkits that are available to the workspace. To see all of the available toolkits, right-click the **Toolkits** container and select the **Show All Toolkits** check box. The palette on the right shows all the available toolkits. The warning decorators indicate that a toolkit is not a project dependency.

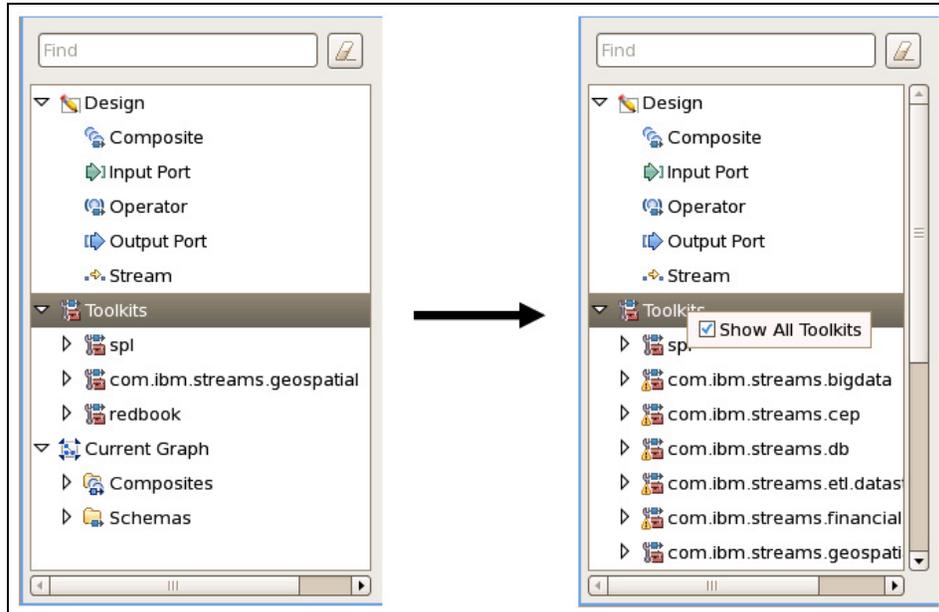


Figure 2-2 Graphical editor palette

Tip: You should only use operators that are from toolkits listed as dependencies. If you use an operator from a toolkit that is not listed as a dependency a compile error occurs, indicating that the operators cannot be resolved. You must then update project dependencies to include the new toolkit that you are using.

To select an operator, find it in the palette by using one of the following methods:

- ▶ Explore the toolkits by expanding the toolkit, and then the namespaces to see the operators available in each namespace.
- ▶ Use the Filter area at the top of the view. Use asterisk (*) for wildcard characters as you type the filter expression elements in the palette that match the filter are displayed as shown in Figure 2-3 on page 13.

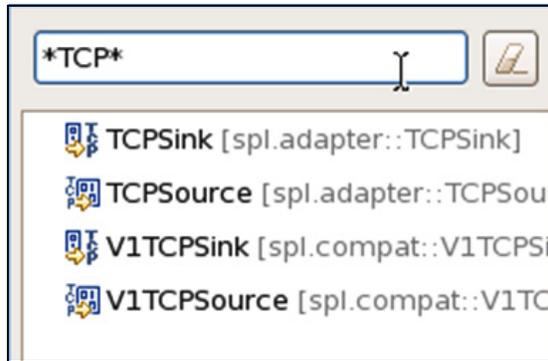


Figure 2-3 Filtering for TCP-related operators

Tip: You can use the generic operator from the Design section of the palette as a placeholder for a new operator or if you are unsure exactly which operator to use.

2.1.2 Connecting operators

After you place operators on the canvas, you must connect the operators. Operator connections originate at an output port and connect to an operator's input port. When you drop an operator onto the canvas, any required ports (per the operator's definition) are created. Ports are shown as small yellow rectangles attached to the sides of an operator. Input ports are attached to the left side, output ports to the right side. The Join operator is shown with two input ports and one output port in Figure 2-4.



Figure 2-4 Join operator

You can add more ports to an operator by selecting the input or output port element from the design palette and dragging it onto the operator as depicted in Figure 2-5 on page 14. If the operator allows additional ports (according to its operator model), the operator color becomes green to indicate you can add the port. An operator that becomes red indicates that you cannot add the port.

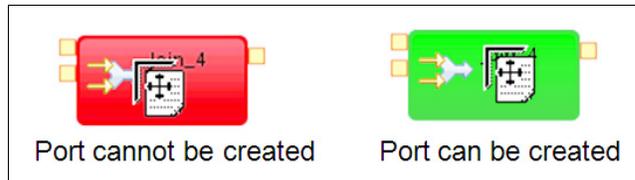


Figure 2-5 Dropping ports on operators

To connect operators, you start by selecting an output port. Press the left mouse button and start dragging the mouse cursor. This enables linking mode in the editor and you can then release the button. As you drag the mouse, you see the connection emanating from the output port. As you drag the connection to an input port, the port changes color: it becomes green if you are able to create the connection or red if the output port's schema does not match the expected input port schema. To create the connection, click the input port that you want the connection to flow in to. To cancel the connection and exit linking mode, press the Esc key.

When you are in linking mode, white ports might be displayed on the input side of operators. These are “ghost” ports. If you create a connection to one of these ports, a new port will be added to the operator and then the connection is made to the new port. This is provided as a convenience so that you do not have to explicitly add the port prior to making the connection.

You can also enable linking mode if you want to make several connections at once. To switch to linking mode, press the linking mode toggle button (shown in Figure 2-6). To turn off linking mode, press the toggle button again.



Figure 2-6 Link mode toggle

2.1.3 Layout options and Outline view

The graphical editor lays out composite definitions stacked vertically. Within each composite, elements are laid out in a left to right orthogonally connected flow. A set of controls along the top of the editor can be used to affect the layout of the graph, as shown in Figure 2-7 on page 15.

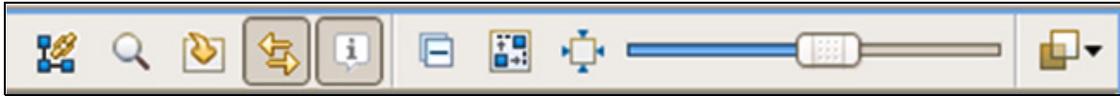


Figure 2-7 Graphical editor controls

The layout button (Figure 2-8) refreshes the graph display. Use to clean up the graph and line up the operators.

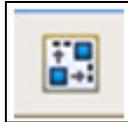


Figure 2-8 Graphical editor, layout button

The fit content button (Figure 2-9) will relayout the graph and adjust the zoom level so that all content fits within the canvas area.

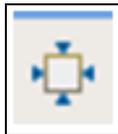


Figure 2-9 Graphical editor, fit content button

The zoom slider (Figure 2-10) adjusts the zoom level of the graph.

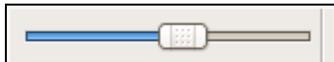


Figure 2-10 Graphical editor, zoom slider

With the graph zoomed out, the operators are displayed as simple graphics with no content. As you zoom in, the operator graphic will contain the operator's icon, name, and type. The fonts also get bigger as you zoom in. When the graph is zoomed in, use the outline view to navigate within the graph by dragging the visible area in the outline over the section of the graph you want to work on.

The graphical editor does not save any positional information. When a file is opened in the graphical editor, the layout is restored to the default. You can move objects around during the editing session, but clicking the layout button returns elements to their default position.

Outline view

The graphical editor Outline view (Figure 2-11) shows the entire graph, with the visible portion of the graph in the editor canvas highlighted in yellow. The Outline view shows the Vwap sample application. The yellow highlighted area indicates the part of the graph visible in the editor. You can move the highlighted area to reposition the visible area of the graph.

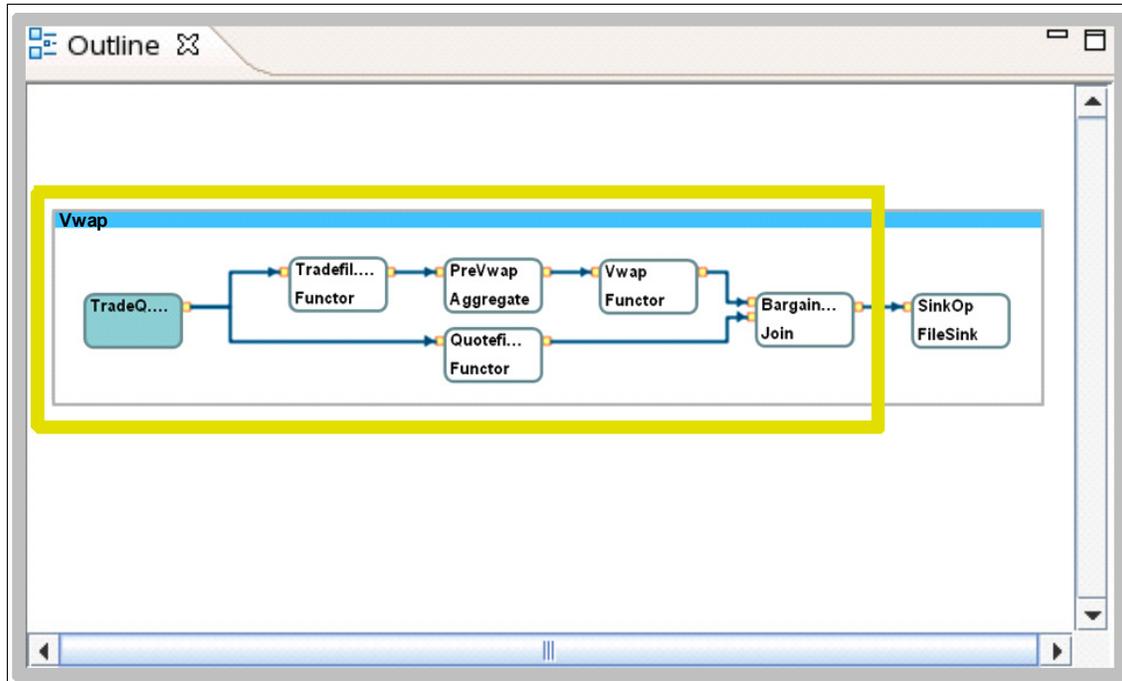


Figure 2-11 Graphical editor, Outline view

2.1.4 Defining global types and schemas

You can create global types that can then be used to define the schemas for connections within the graph.

Defining global types and use statements

In the graphical editor, you can define global types. To define global types in the SPL Graphical Editor, double-click anywhere in the blank area of the canvas (not within a composite) to open the Global properties window. In the properties window, select the **Types** tab to see the global types.

Add a type by clicking the **Add new type** item that is highlighted. You can edit existing types as shown in Figure 2-12.



Figure 2-12 Global properties, Types dialog

Click **Uses** to specify use statements for the SPL file. By invoking content assist (Ctrl+Spacebar), you see the list of potential use statements. The list is derived based on the project's direct dependencies. This is depicted in Figure 2-13.

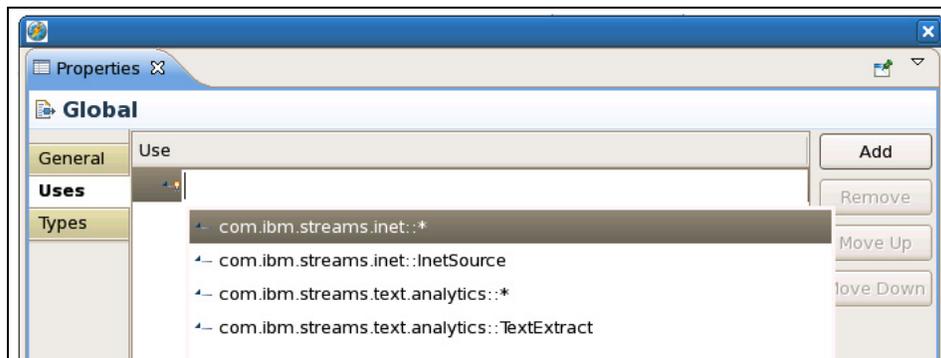


Figure 2-13 Global properties, Uses dialog

Defining schemas

An output port defines the schema for a connection. The schema provides the set of attributes (name and type) that make up the tuple that will flow over that connection. The graphical editor provides a visual clue to indicate which connections have schemas defined. A solid blue line connection indicates that the schema is defined. A dashed lighter blue line indicates that the connection does not have a defined schema. This is depicted in Figure 2-14.

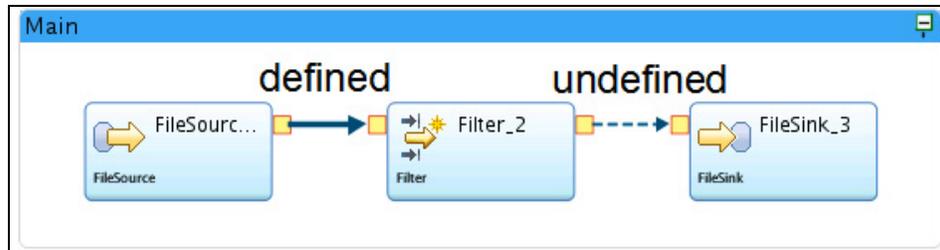


Figure 2-14 Defined and undefined schemas for connections

To define the schema, you can select either the output port or the connection, and double-click to open the Edit dialog. In the property edit dialog, you can provide the following information:

- ▶ Output stream name
You can click rename to provide a new name for the output stream.
- ▶ Schema table
The name column lists the attribute name; the type column provides its type. Content assist is available in the Type column and shows a list of SPL types you can use. Click **Add Attribute** at the bottom of the table to add a new attribute. The following examples show how to use the dialog:
 - Define the schema using an existing type. Enter <extends> in the name column and the existing schema type in the type column.
 - Define the schema with attributes of primitive types. Enter the attribute name in the Name column and the SPL type in the Types column.

You can manipulate the table using the Remove, Move Up, and Move Down buttons along the right side.

Tips:

- ▶ To open the Edit dialogs for any elements properties, select the element and either double-click or right-click, and click **Edit** in the context menu.
- ▶ Define your schemas in separate SPL files. This allows you to reuse them more easily.

The palette also shows all the schemas that are available for use in building your application. Under each toolkit, you see all the static schema types that are available for reuse. A list of all schema types is defined under the current graph section:

- ▶ In the file
- ▶ In your composite operator
- ▶ Implicitly by streams from your operators

You can select the schema and then drag it to either a connection or an output port. This will define the connection with the schema. If the connection already has a schema defined, dragging a schema to the connection will append the schema element at the end of the existing schema.

You can also copy a schema from a defined connection and paste it onto an undefined connection. This step defines the connection with the copied schema. To copy the schema, select a connection that has a defined schema (bold blue line), press Ctrl+C (to copy), then select an undefined schema and press Ctrl+V (to paste). If you paste the schema onto a connection where a schema is already defined, the schema of the connection will be overwritten with the pasted schema.

Tip: If you want the same schema definition for several connections, copy the schema and repeatedly paste it onto the connections to define them.

2.1.5 Editing operator details

To fill in the details for an operator from the graphical editor, open the properties window for the operator. Either double-click the operator or right-click the operator, and click **Edit**. The multi-tab Properties dialog opens (Figure 2-15 on page 20).

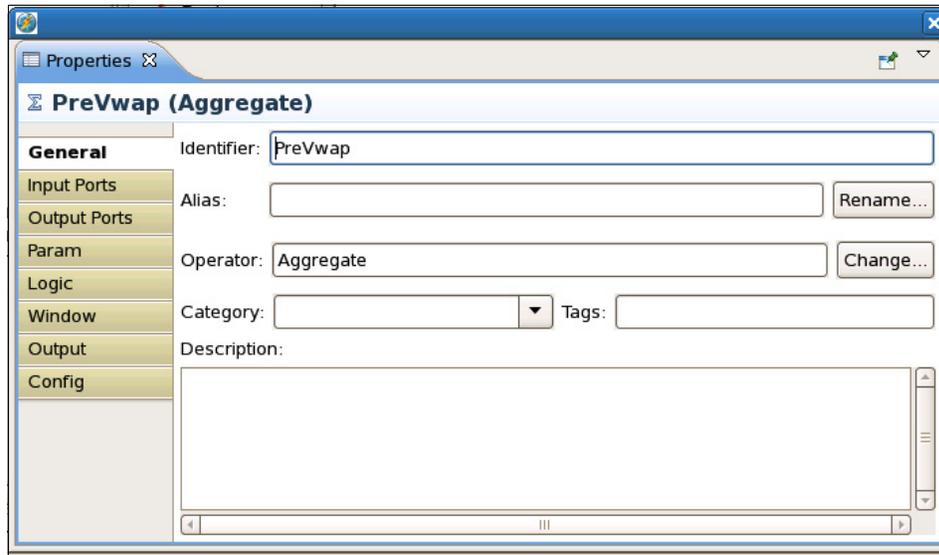


Figure 2-15 Properties dialog for an operator

Select each tab and fill in the details. The first three tabs define the operator and the ports. The remaining tabs reflect the SPL operator clauses. As you fill in the tabs and save the graph, the graphical editor generates SPL source code.

Tip: You can select **Edit** → **Undo** (or Ctrl+Z) to undo any changes you made in the operator's properties dialog.

General tab

The General tab has basic information about the operator in the following fields.

- ▶ Identifier

Either the operator's alias if defined, or the stream name of the first output port. This field is not directly editable. The identifier is shown as the name of the operator on the graph.
- ▶ Alias

The SPL Graphical Editor by default assigns all operators it creates an alias. Click the Rename button to change the alias of the operator.
- ▶ Operator

This is the SPL operator that will be invoked and corresponds to a defined operator from a toolkit. You can change the operator by clicking Change and selecting a new operator from the list. This will replace the current operator with the selected one.

Note that using the Change button is the same as dropping an operator from the palette on top of an existing operator. All the clauses will be overwritten. The ports and their schemas will remain unchanged.

► **Category**

This field is a string which identifies the category for this operator. Categories can be used in the instance graph to provide an alternative layout where operators from the same category are grouped together. These can also used to search for operators. More details about using Category are discussed in section on Program Understanding.

► **Tags**

You can add one or more tags (strings) separated by a comma to an operator. These are useful for searching and finding operators later. For example, adding a tag with a work item identifier to every operator changed for a particular work item allows you to search by that tag later and see all operators for that work item. Type directly in the box to enter the tag information.

► **Description**

This information is formatted as SPLDoc and is displayed in the hover text for an operator. To change the description, type in the description field.

Figure 2-16 shows the completed fields in the General tab for an operator.

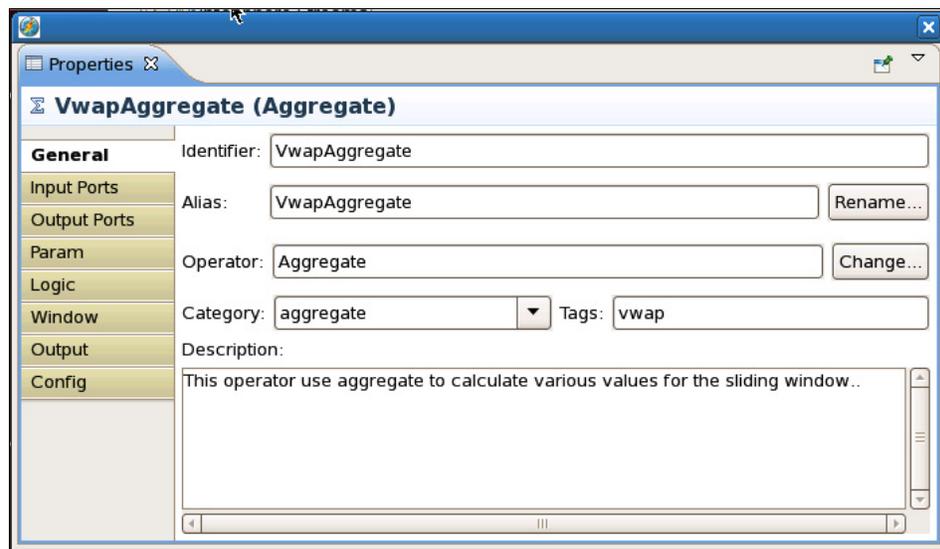


Figure 2-16 General tab, operator properties

Input Ports tab

On the Input Ports tab (Figure 2-17) you can add and delete input ports and provide an alias for the input port. The input port's schema is displayed if it has been defined.

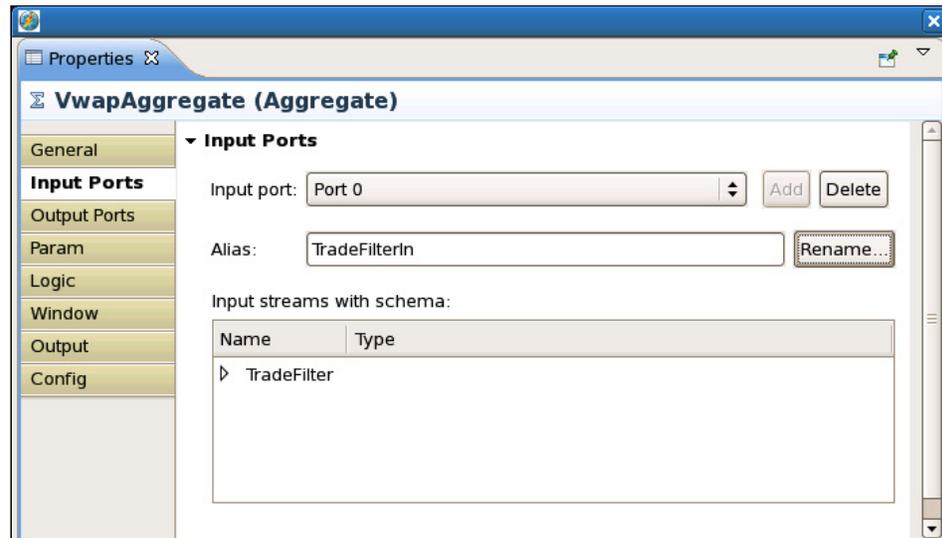


Figure 2-17 Input ports tab, operator properties

The Input Ports tab has the following fields to describe the port:

- ▶ Input port

Use this field to select the specific input port to edit. This will update the Alias field and the schema information to reflect the currently selected port. You can add or delete input ports by clicking **Add** or **Delete**. If the Add button is disabled, the operator's definition does not allow additional input ports to does not allow the removal of the selected input port.

- ▶ Alias

Specify an alias name for the input port. To change the alias, click **Rename** and enter the new alias. The code is then refactored using the new alias.

- ▶ Input streams with schema

This area shows a read-only view of the schema for the input port if it has been defined. The input port's schemas are defined by the connected output port of the feeding operator. You can expand the schema to see the attributes.

Tip: Always provide an alias for input ports. With this approach, any future refactoring becomes simpler because the editor is better able to identify the required changes.

Output Ports tab

Use the Output Ports tab (Figure 2-18) to add and delete output ports and provide an alias for the output port. The output port's schema can be defined.

The figure shows an output port for an aggregate operator. The <extends> name in the Name column is for reusing an existing schema type. Additional attributes are added by specifying a name and type.

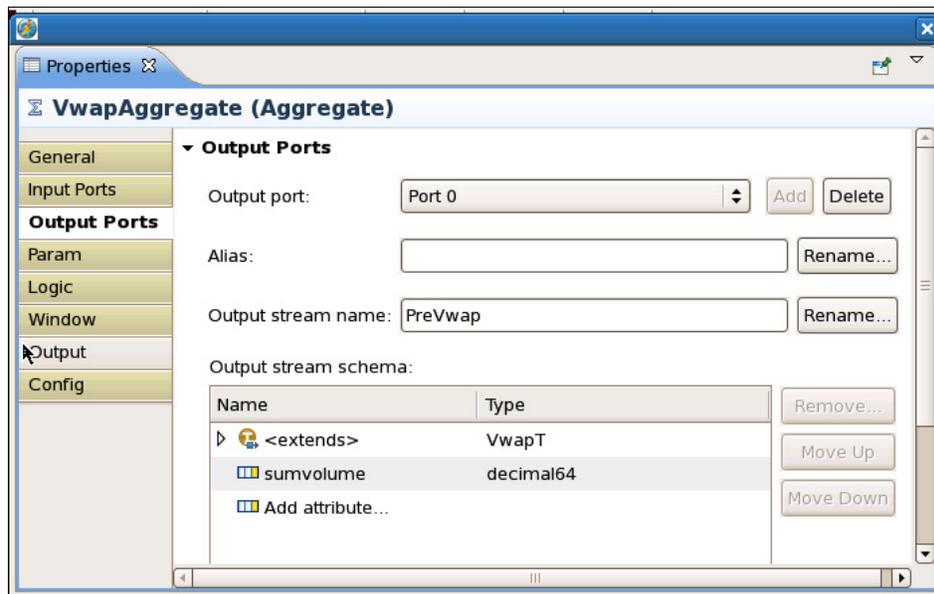


Figure 2-18 Output ports tab, operator properties

The Output Ports tab has the following fields to describe the port:

- ▶ Output port

In this field, select the specific output port to edit. This will update the Alias field and the schema information to reflect the currently selected port. You can add or delete output ports by clicking **Add** or **Delete**. If the Add button is disabled, the operator's definition does not allow additional output ports to be defined. If the Delete button is disabled, the operator's definition does not allow removal of the selected port.

- ▶ **Alias**
Specifies an alias name for the output port. This is used in the output clause. To change the alias, click **Rename** and specify a new alias. The code is then refactored using the new alias.
- ▶ **Output stream name**
This is the name of the output stream on this portclick **Rename** and provide the new name. The code is refactored with the new output stream name. The output stream name is displayed on the hovers for a connection on the graph.
- ▶ **Output stream schema**
The table defines the schema for the output port. Each output port must have a defined schema. Each row in the table defines an attribute. You can use <extends> for the name and provide an existing schema type for the type to reuse a schema. To add attributes, scroll to the bottom of the table and click **Add Attribute**. This will create a new row in the table. To edit an entry, click the cell in the table to switch to edit mode, type in the new information, and click outside the cell to leave edit mode. The Type column provides content assist (Ctrl+Spacebar) and displays a list of available types. Use the buttons on the right side to remove attributes or adjust the position of attributes within the tuple.

Parameters tab

The Param tab provides a list of the all the currently selected parameters and their values for the operator, as shown in Figure 2-19.

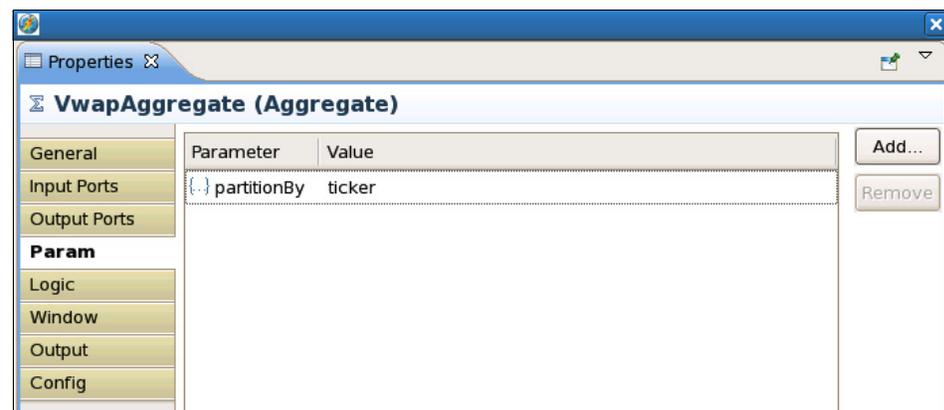


Figure 2-19 Parameters, operator properties

The dialog is used to add and remove parameters and set the values for the chosen parameters.

When an operator is dropped onto the canvas, the operator is prefilled with a list of the required parameters as specified by the operator's definition. To add optional parameters, click **Add**. A dialog of all the operators parameters is shown. Select those that you want and click **OK**. After the parameters are added, fill in the values. To remove a parameter, select it and click **Remove**.

The parameters are listed as a set of name and value pairs. To edit the value of a parameter, click the value cell to enter edit mode. Type in the new value and click outside the cell to leave edit mode. The next dialog is shown in Figure 2-20.



Figure 2-20 Add parameter dialog, operator properties

Tip: To see a description of the parameters, select the operator on the canvas and press Ctrl+F1. The context sensitive help for the operator includes information about the parameters.

Logic clause

The Logic tab provides a functional SPL text editor control including content assist to specify the logic clause for the operator. The one in Figure 2-21 is empty. You can enter your logic code directly in the view.

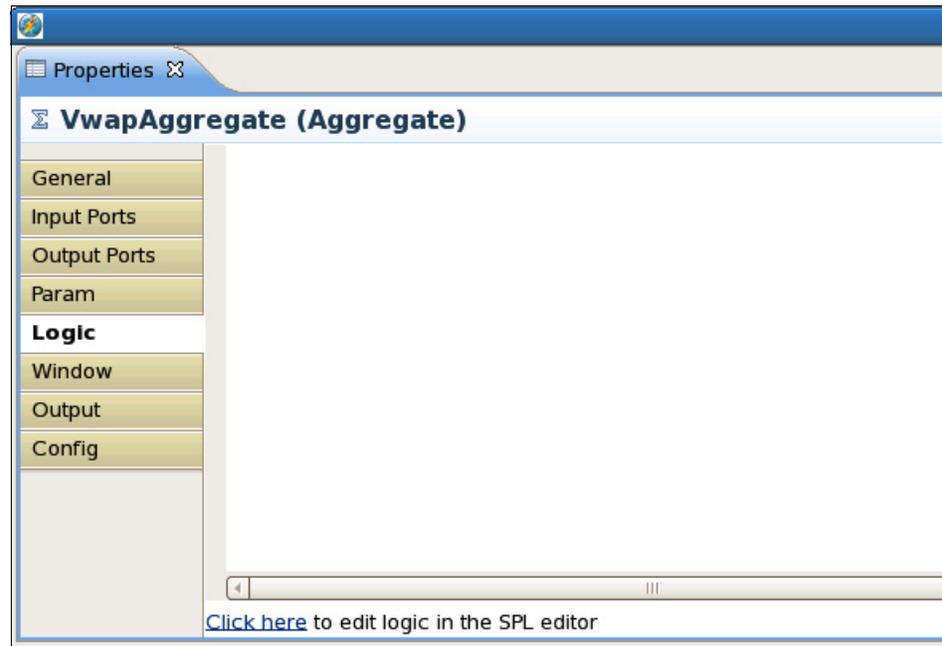


Figure 2-21 Logic clause, operator properties

The logic clause provides an SPL snippet text editor that you can use to enter the logic for your operator. Just type the logic clause code into the box. The editor provides content assist, and syntax highlight, the same as the SPL text editor. You can quickly jump to the SPL text editor by clicking the **Click here** link at the bottom of the page, as shown in Figure 2-21.

If the logic clause contains any syntax or compiler errors, the snippet editor will show the problem markers on the ruler on the left side.

The embedded SPL text editor in the logic tab supports syntax highlighting and content assist. Invoking content assist by pressing Ctrl+Spacebar, after entering `print`, shows the list of functions that start with `print`, as shown in Figure 2-22.

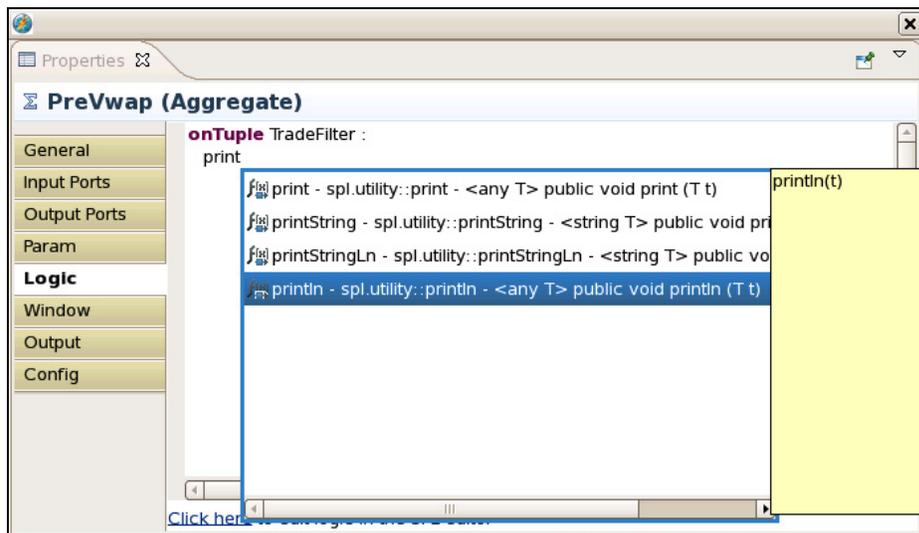


Figure 2-22 Content assist in logic clause, operator properties

Tip: If you enter a long logic clause, a better way is to use the SPL text editor because you can see more of the code and can more easily examine other parts of the code. Copying code snippets from other operators is easier using the SPL text editor.

Window clause

The window tab shows a list of the window definitions for the operator. The Window clause showing a window definition for the aggregate operator is depicted in Figure 2-23.



Figure 2-23 Window clause, operator properties

To change or add a window definition, click **Add** or **Edit**. The Add Window Mode dialog opens (Figure 2-24 on page 29). In this dialog, you can define the window.

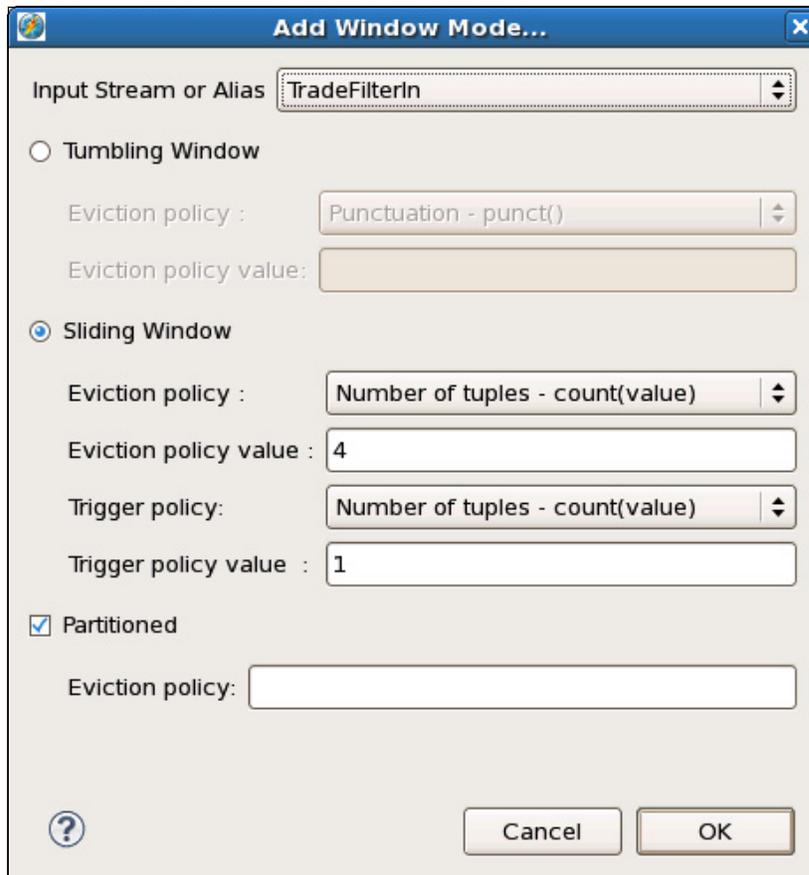


Figure 2-24 Add Window Mode dialog, operator properties

The fields that define a window are described in the following list.

- ▶ **Input Stream or Alias**
Use this control to select the input stream to define the window mode. Each input stream can have at most one window mode.
- ▶ **Tumbling Window**
Select this button if you want a tumbling window.
- ▶ **Sliding Window**
Select this button if you want a sliding window.
- ▶ **Eviction policy and value**
Select an eviction policy from the drop-down list and then enter the value for the policy.

- ▶ Trigger Policy and Value:
Select a trigger policy type from the drop-down list and then enter the value for the policy.
- ▶ Partitioned
Select this check box to create partitioned window.

Output clause

The Output tab for an operator from the Vwap sample application is shown in Figure 2-25.

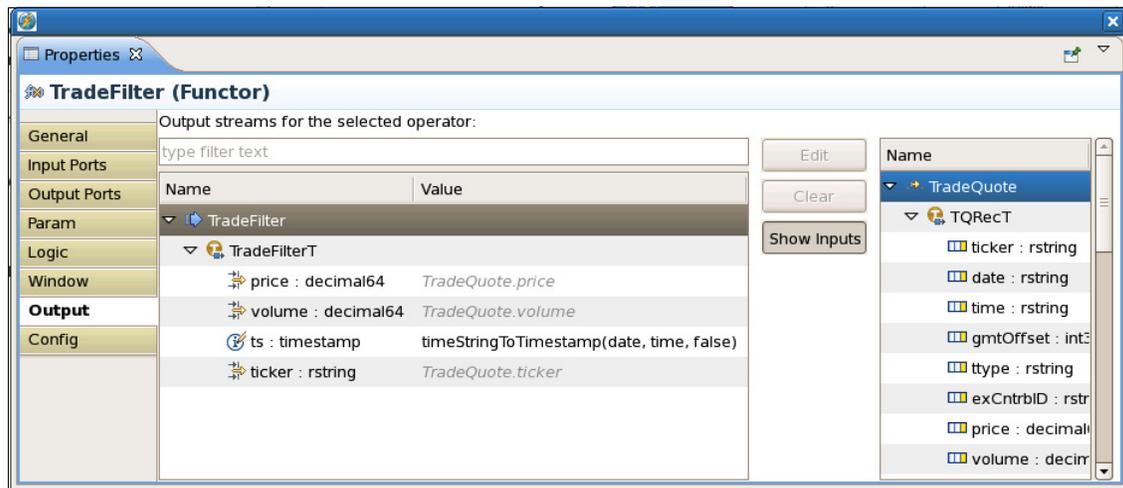


Figure 2-25 Output clause, operator properties

The left side shows all of the attributes for the output stream and their values. A value that is displayed but not activated indicates a default assignment to a matching attribute from the input stream. The attribute *ts* has been assigned from an expression.

For each output stream, the table shows the attribute name and the value. The value is the expression. By default, attributes in the outgoing stream with the same name and type as an incoming stream attribute will be assigned the incoming stream attribute. This convenience mechanism avoids having to assign all attributes and allows automatic forwarding of attributes.

To edit the value for an attribute, click in the value cell to enter edit mode or select the attribute and click **Edit**, and then type in the new expression. Content assist is available for the value.

If you click **Show Inputs**, the right side will list the input streams for the operator. This information can help you while you define the expressions for the output tuple attributes.

Config clause

The Config clause is a set of name-value pairs that provide configuration information to the run time for the operator. Figure 2-26 shows an example of the config clause for an aggregate operator.

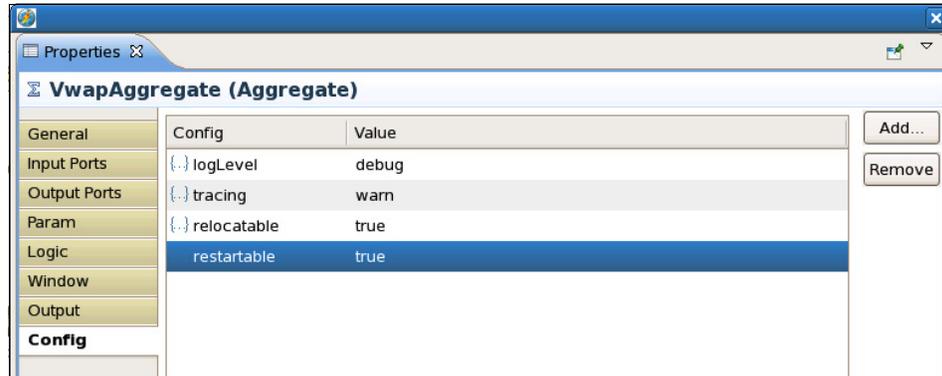


Figure 2-26 Config clause, operator properties

Click **Add** to see the list of properties available (Figure 2-27). Select the properties you want to add. After they are added to the list, provide values for each configuration property.

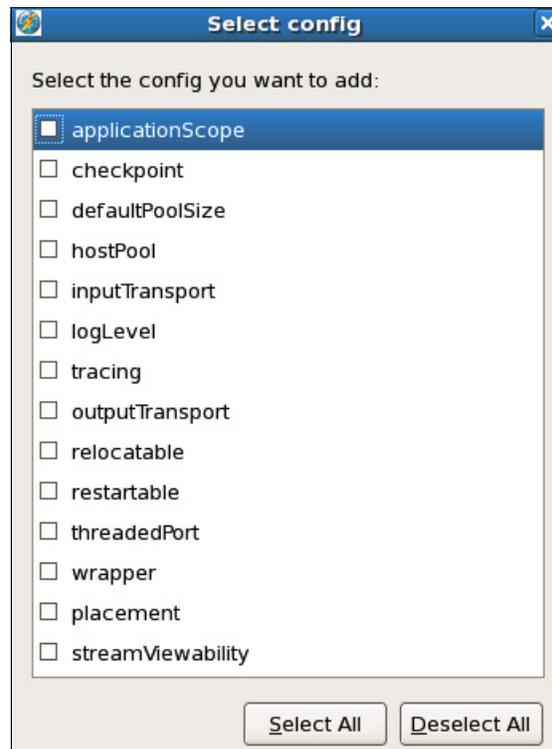


Figure 2-27 Add Config properties, operator properties

To provide a value, click the config property's value cell to switch to edit mode. In edit mode, if a set of predefined values for the config property exists, a drop-down control will list the valid values. For config properties such as `hostPool`, you simply type in the value.

To remove a config property, select the property and click **Remove**. You can multi-select properties to remove several at once.

2.1.6 Generating SPL code

Every time you save the graphical editor contents, SPL source code is generated. Because the editor will generate correct SPL source code, you do not have to understand all the nuances of the SPL syntax or worry about typographical errors while entering code. The graphical editor uses default

names and might insert placeholders for tuple attributes in order to generate SPL code. This results in compile-time errors while you are completing the details of the operators and connections within the application.

The code is always formatted when it is generated according to the formatting preferences specified for SPL source code.

SPL code might not be successfully generated if syntax errors are within the operator's details. Graph elements with syntax errors that prevent code generation have an error marker in the graph. Correct these syntax errors before making more changes. If you continue to make changes without correcting the syntax errors, the editor will make your changes but will be unable to generate code until the syntax errors are corrected. If you close the graphical editor but syntax errors exist, a temporary file is written. If you open the file again, this temporary file is reloaded and a prompt indicates there were unsaved changes.

Tip: Always consistently format your SPL source code. This helps the file utilities, which compare files for changes, and provide consistent results.

The graphical editor and the SPL text editor can both be opened with the same file. When changes are made in the graphical editor and saved, the SPL text editor automatically refreshes with the changes. When changes are made in the SPL text editor, the next time the graphical editor is given focus, you will be prompted to replace the graphical editor contents with the new SPL file contents.

The Replace Editor Content dialog (Figure 2-28) indicates that the SPL file has changed on the file system. This usually is a result of editing the file with both editors open. There is no merge support, the contents are replaced.

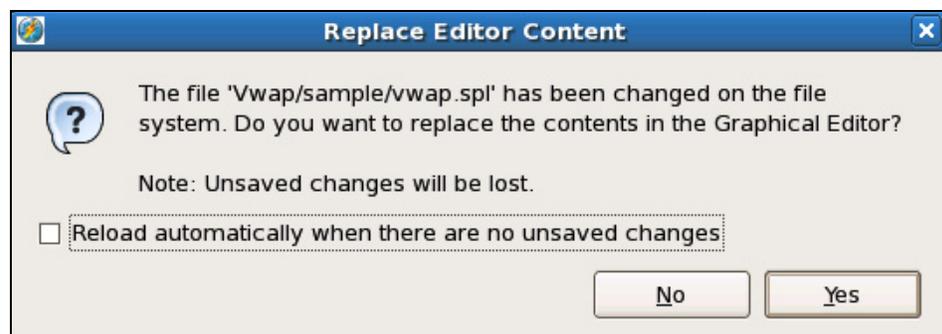


Figure 2-28 Replace editor content dialog

Tip: When editing in both the graphical and text editors, make changes only in one of them at a time and save to synchronize the files.

The Eclipse **Compare with** and **Replace with** commands can be used to compare versions or recover changes.

2.2 Application development use cases

In this section, we describe how to use the features and functions that are described in 2.1, “SPL Graphical Editor” on page 10. Each use case focuses on the specific features relevant to that use and provides additional details of how to use Streams Studio to accomplish these tasks.

2.2.1 Design: Sketching an application

The design phase for Streams applications typically involves creating a graphical representation of the Streams application (or applications) along with notes and details that describe how the graph’s operators and connections are expected to function after they are implemented. The SPL Graphical Editor in Streams Studio provides a convenient tool with which to capture the design of a Streams application.

To create a design, you need to create an SPL file and open it in the graphical editor. You can use this file as a blank canvas to start sketching your SPL application. The typical work flow is as follows:

1. Place your source and sink operators on the canvas (source on left, sink on far right).
2. Starting from the source, work left to right across the graph placing new operators and connecting them as you proceed until you have completed sketching the application.
3. Add annotations to provide details to consumers of the design.

Generic operators

Generic operators are used at design time as placeholders in the graph. These represent an operator without specifying an exact operator. Often when designing there are cases when new operators must be created, or it is unclear which operator will be used at that point in the graph. By using a generic operator and providing details within the operators description you are able to capture that an operator is needed without having to provide all of the details.

Figure 2-29 depicts a simple sketch of an SPL application. Note that the generic operator is shaded gray to indicate that it needs to be defined as a real operator. A developer starting with this design sketch is then able to easily distinguish generic operator placeholders.

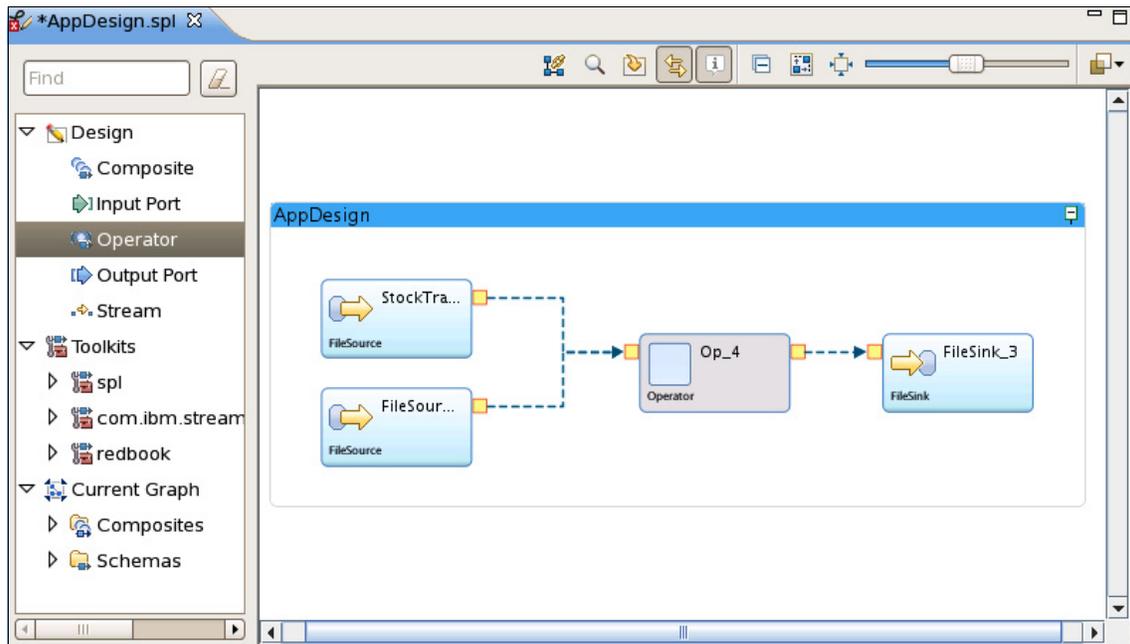


Figure 2-29 Basic design sketch

Validation Layer, hiding errors

Generic operators will generate SPL source code but the code will likely cause compile-time errors. You might also have errors because operators have not had all of their details specified. This is normal during the design phase because the application is not complete. You can turn off the validation layer on the graphical editor (by using the icon in the upper right corner of the editor) to hide the compiler errors.

If the validation layer is on, errors are visible (with a red composite title and error decorators). If the validation layer is turned off, you see the error decorators displayed but not active, indicating a stale marker. This indicates that the graph has changed since the build that created the markers.

Tip: At design time, give all generic operators a descriptive name. This name is displayed, in the graph providing insight into the operators function.

Descriptions and annotations

When adding operators to a design, use the Description field in the operator's General Properties (see 2.1.5, “Editing operator details” on page 19 for more information) to add details about the operator's function, expected inputs and outputs, and any other design details. This information is shown when the mouse pointer hovers on an operator in the graph as part of the hover information for the operator.

Annotations are notes that are anchored to an operator. These are used to call attention to specific operators or as a means to provide special notes regarding the operator. They are often temporary in nature and can be used to capture design comments when reviewing a graphical design. The annotations are captured as part of the source code when the graph is saved. Because the annotations can block parts of the graph and clutter the view, you can disable the view of the annotation layer by using the layer control. To add an annotation to an operator, select the operator, right-click, and select **Add Annotation**. When the annotation is present, to edit the contents, double-click the annotation object and enter the text. Hovering on the annotation will show the complete text.

Figure 2-30 shows that the generic operator (Op_4) has an associated annotation, which is the green note pinned to the graph. Hovering on the green note shows the complete annotation as depicted in the yellow hover control.

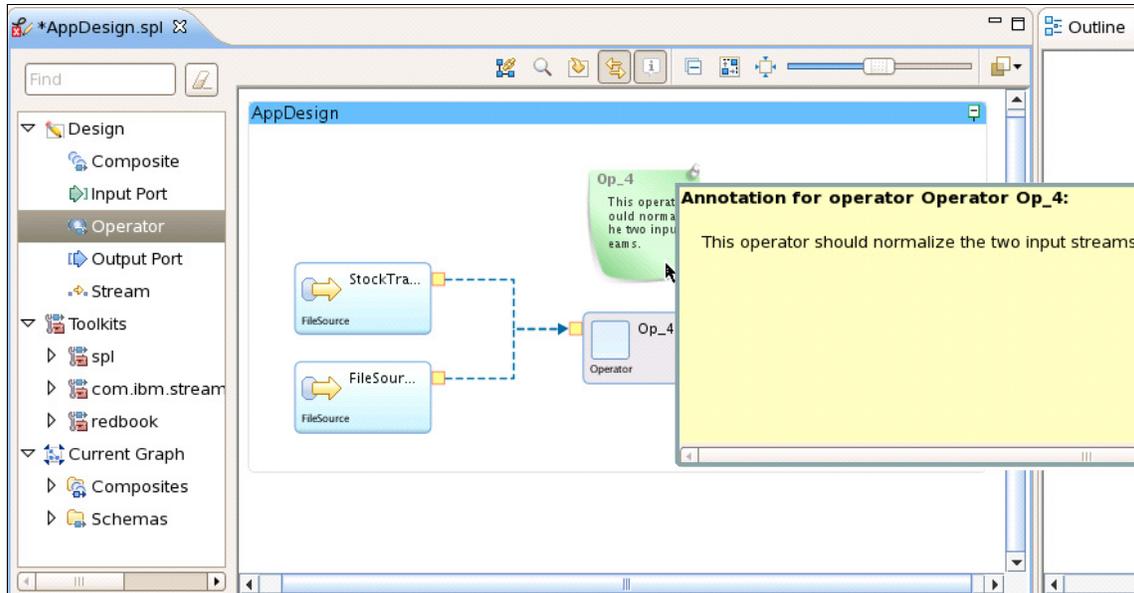


Figure 2-30 Generic operator annotation

Tip: When using annotations to indicate to-do items, delete the annotations when each to-do is completed.

Consistent graph

By saving the design graph as an SPL file, and using the same SPL file for implementing SPL applications, you get a consistent view of the application. In this way, reviewing and communicating designs is easier in a team environment because everyone is working from the same base SPL file and sees the same graph.

2.2.2 Implementing an application

This section describes two workflows for implementing an application:

- ▶ The first workflow assumes the starting point is a design sketch. See 2.2.1, “Design: Sketching an application” on page 34 for more information about creating a design sketch.
- ▶ The second workflow starts from a blank canvas.

The workflows are similar in the approach and contain many common steps. The steps make use of the functions and features described in the previous section; they are not described in detail again here.

Implementing from a design sketch

The following steps implement an application, starting from the design sketch:

1. Open the design sketch in the graphical editor. This will allow you to start with the graph from the sketch and proceed to fill in the details to generate the SPL code for the application. Make sure the validation layer and the annotation layer are both enabled.
2. Proceed from left to right, starting with one of the leftmost source operators. For each operator along the subflow, complete the following steps:
 - a. If the operator is a generic placeholder operator, change it to the actual operator. Remember to update the project’s dependencies if using an operator from a toolkit that is not listed in the dependencies.
 - b. Define the schema for each output port. By doing this, filling in the downstream operators detail is easier because the schema attributes are needed in the SPL clauses.
 - c. Fill in the remaining operator details by working through the tabs in the order they are listed on the left side of dialog.

3. Repeat step 2 on page 37 for each subflow. If the subflows merge at some point, a good practice is to implement all of the input subflows before proceeding to complete the details of the merged flow's operators.

You can delete any annotations that are no longer needed.

Tip: Make sure all inputs to an operator have defined schemas and that you define the output port schemas for an operator before filling in the remaining operator details.

Implementing an application from a blank canvas

To implement an application from a blank canvas, start by creating a new main composite in the project. This will produce a new SPL file. Open the file with graphical editor and an empty main composite. Then continue with the following steps:

1. Drag the source operators and the sink operators to the graph.
2. Proceeding from left to right, drag operators to create the flow and then connect them to create the applications flows from source to sink. At this point you have created a sketch of the application and the remaining steps are similar to the workflow that starts with a design sketch.
3. Proceed from left to right starting with one of the leftmost source operators. For each operator along the subflow, do the following steps:
 - a. Define the schema for each output port. By doing this, specifying the downstream operator detail is easier because the schema attributes are needed in the SPL clauses.
 - b. Complete the remaining operator details by working through the tabs in the order they are listed on the left side of dialog.
4. Repeat step 3 for each subflow. If the subflows merge at some point, a good practice is to implement all of the input subflows before proceeding to complete the details of the merged flow's operators.

Tip: If you want to insert an operator along a connection between two existing operators, you can drop the operator directly on the connection. The graphical editor will insert the operator and then refactor the surrounding operator.

If you want to see the generated code that is created with each save operation, you can have both the graphical and SPL text editors open at the same time. These are kept in sync as changes are made.

2.2.3 Composites: Reusable subgraphs

Composites are operators that contain a subgraph. They are used to create reusable subflows that can then be used as a single operator in an application.

To create a composite using the graphical editor, select the composite element from the design palette and drag it onto an empty part of the canvas. This will create a composite container graphic with no content. Figure 2-31 shows a new empty composite (Comp_0). Next add the operators to the composite.

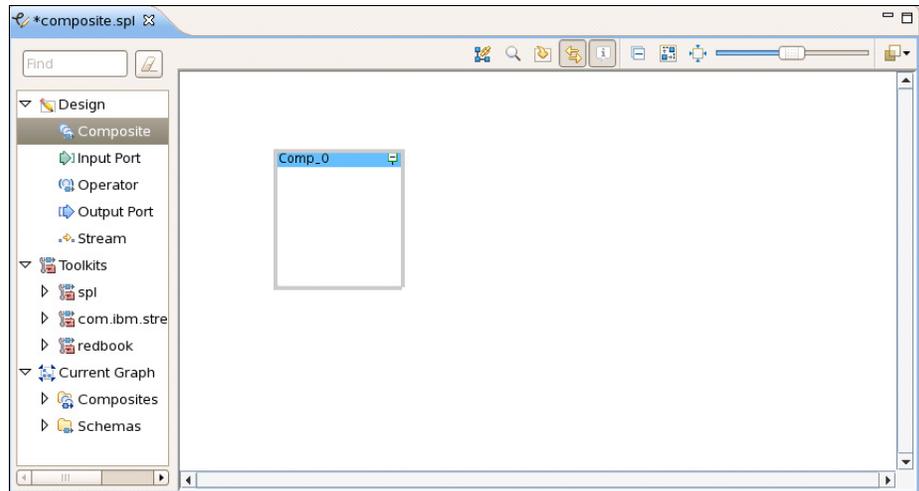


Figure 2-31 Newly created composite operator

You can then drag operators to the composite container. You can also populate a composite by multi-selecting a set of operators from an existing graph and cutting or copying them and then pasting them into the composite definition.

The operators within the composite container need to be connected to form the composite's subgraph. The subgraph then needs to be connected to the ports of the composite container.

To add ports to the composite container, you select the input port or output port element from the design palette and drop into a blank space in the composite container. This adds the port element to the container (yellow rectangle on left side for input port or right side for output port). You can now connect the input and output ports of the composite to the internal subgraph.

Figure 2-32 on page 40 shows a composite operator definition that reflects the data pipeline pattern. Notice that the input port's schema is undefined. Defining

the input port is optional for composite operators. When this operator is used, the connection to the composite's input port will define the schema.

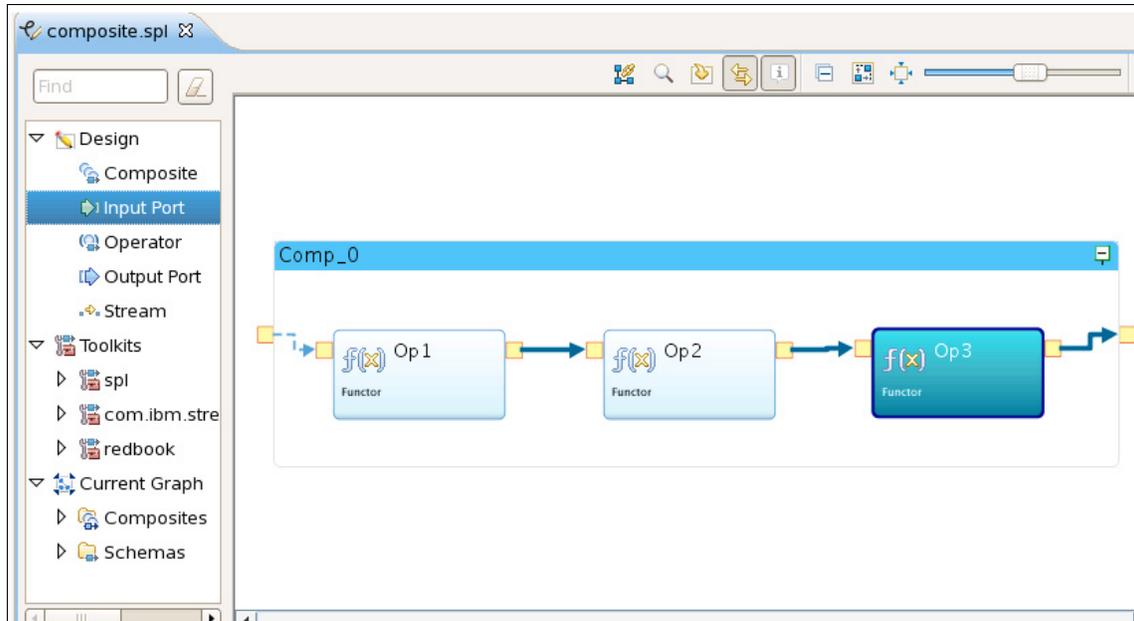


Figure 2-32 Composite operator, data pipeline pattern

To connect the input port of the composite to the internal operator, drag a link from the composite's input port toward the internal operator's input port. This action puts the graphical editor into linking mode and you click the internal operator input port you want to connect to. As a convenience mechanism the graphical editor will create new composite operator output ports if you drag a connection from an internal operators output port to the "ghost" output port (white port) that is displayed on the right side of a composite during linking mode.

You may define types in a composite operator's definition. By default, the types are visible only within the composite operator. To use these types outside the composite operator, define the type as static. To define types, use the Types tab on the composite operator's property page.

Composite operator parameters can be defined using the composite operator's properties dialog. To edit the composite operator's properties, double-click the composite operator. This dialog is similar to the properties for an operator. The Param tab lets you define the parameters for the composite operator.

The composite operator's properties Param tab, as shown in Figure 2-33 on page 41, defines a new parameter. The drop-down list shows the types of

parameters you can define. Select an element from the list and then provide a name and default value if necessary.

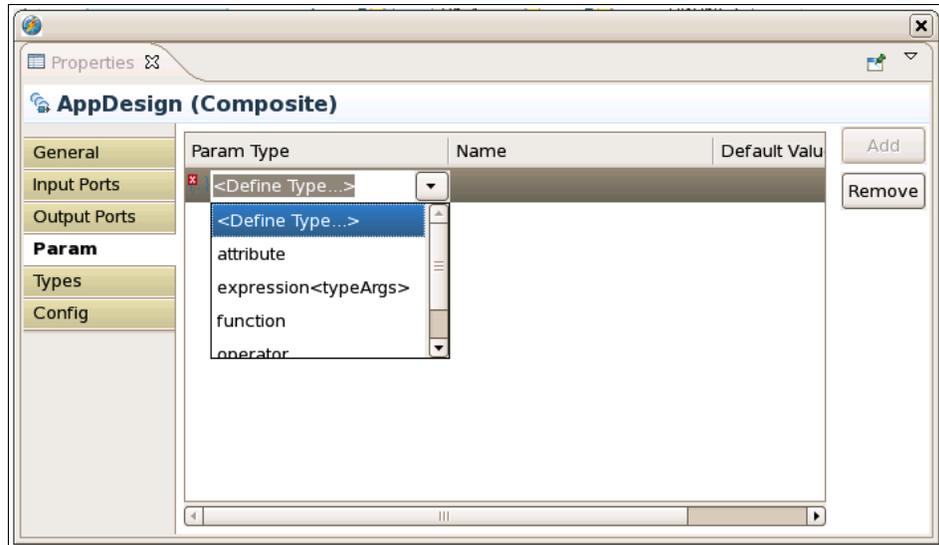


Figure 2-33 Param tab, composite operator properties

After you create a composite operator you can then use it in other parts of the graph. To use the composite, locate and select it on the graphical editor palette, and then drag it to where you want to use it. This will create a reference composite operator that invokes the composite operator. The operator is prefilled with the required number of input and output ports, and also any required parameters that are needed.

The reference composite operator is read-only. If the composite operator definition is within the same SPL file, you can expand the composite operator reference to see the internal subflow. The expanded reference to the composite operator is shaded gray to indicate it is read-only.

Figure 2-34 on page 42 shows a composite operator definition, Comp_0, and an expanded reference to the composite that is in the same SPL file. The reference composite is shaded gray to indicate it is read-only.

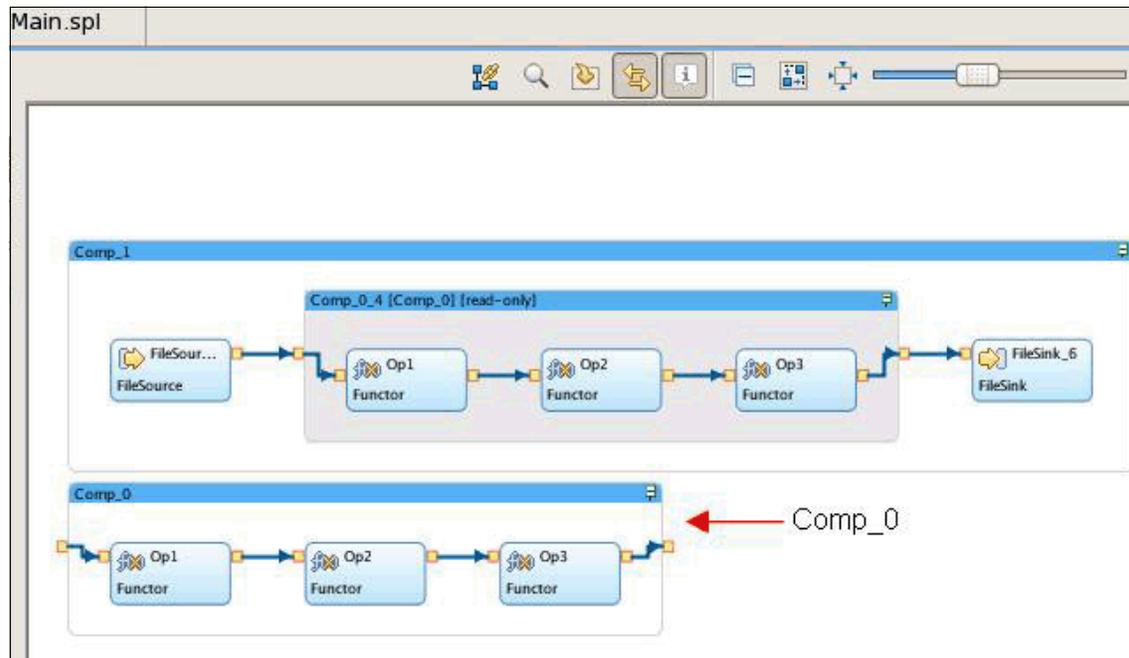


Figure 2-34 Expanded composite reference, same file

If the composite definition is located within a separate SPL source file, when you add a composite reference to an application, the composite will be shown as an operator and will not be expandable. To open the definition and see the subflow or make changes, select the composite reference and press F3 to open the declaration.

Figure 2-35 on page 43 shows a composite reference for a composite that is defined in a different SPL file. The second operator from the left in the graph is the composite reference. The composite reference appears the same as an operator and is not expandable. To see the composite definition and make changes to it, select the composite reference and press F3 to open the declaration.

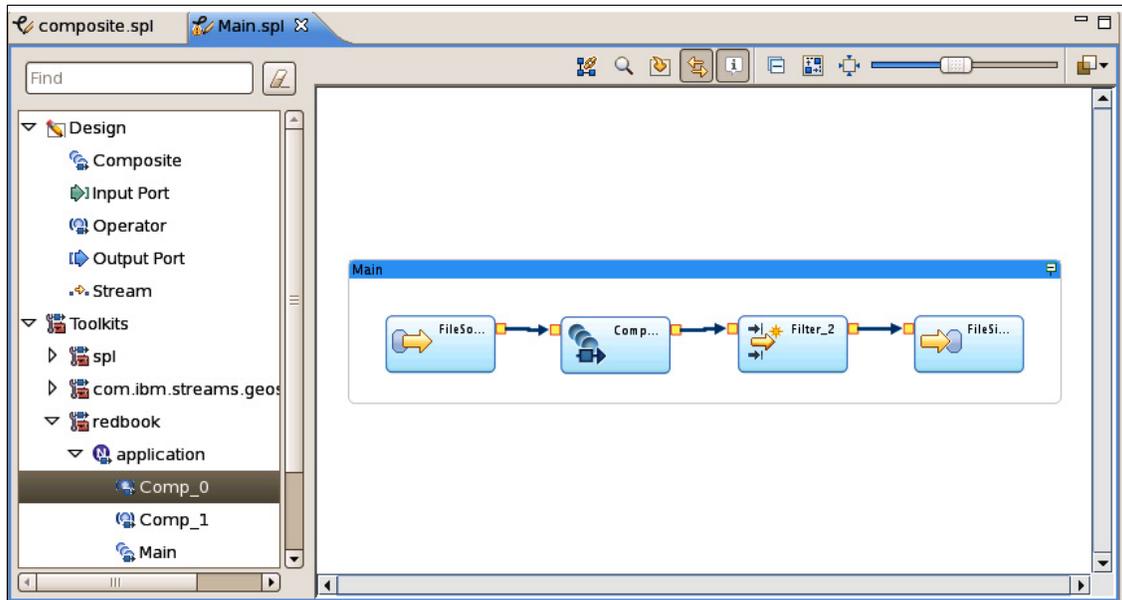


Figure 2-35 Composite reference, different file

To make changes, work with the composite definition. Any changes made in the composite definition will be reflected in all references. If you want to add or remove ports for a composite operator, you must make that change in the composite operator's definition. Adding or removing ports causes the editor to search for operator references within the same file and refactor to add or remove the port. If the operator reference is in a different file, you must manually make the changes in the references.

Tip: Put composite operator definitions in their own SPL file. Place commonly used composites in a toolkit to enable better reuse across applications.

2.2.4 Making changes: Refactoring

When refactoring stream schemas and attributes, a better approach is to use the SPL text editor. The text editor provides a wider view of the SPL source file and helps you to better understand the scope of the changes and also any places you will need to manually change.

Refactoring often involves renaming elements and updating any references to the renamed element. To rename an attribute, select the attributes definition, right-click and select **Rename element**. This allows you to type over the existing name with the new one. Press Enter when you finish renaming; the editor then

refactors the code using the new name. There may be cases where the editor is unable to determine that a change is required or might be unable to properly resolve name ambiguity. In these cases, the editor can miss changes that must occur. The text editor provides indicators of what was changed.

Always use aliases for input ports. By doing this, refactoring is easier because it eliminates some of the ambiguity that SPL permits. In general, using aliases for operators and ports provides more details and simplifies refactoring.

Always save before and after any refactoring operation. Saving provides two benefits: the code will compile prior to the refactoring and you can fix any compile errors before refactoring. This approach can help you more easily find places where the editor was unable to correctly refactor. Saving after the refactor triggers a build and any errors can be attributed to the refactoring operation.

Saving before and after creates revisions in the Eclipse file history and you can then use the **compare** and **replace** commands on the SPL file to get a more detailed understanding of the refactoring changes.

To undo refactoring operations, either use Ctrl+Z or select **Edit** → **Undo**.

2.2.5 Program understanding

The graphical editor can be used as an aid in program understanding. The description in this section is not meant to be comprehensive but instead give an idea of the various uses of several of the features and functions that are not directly related to editing.

Graph versus source code

It is often said that a picture is worth a thousand words. In the case of SPL applications, a graph can be worth a thousand lines of code. By presenting the SPL in a graphical context, understanding the application topology is much easier by showing a high level view of how data flows through the operators in the application. Even in a simple SPL application's source code, trying to mentally visualize the flows can become difficult. The text editor provides a detailed view of SPL code and is useful for more detailed understanding of the individual clauses and logic in the application.

Hovers

The graphical editor provides hover support for operators and connections within the graph. For operators, the hover shows the SPLDoc and the source code that is associated with the operator, providing a quick and simple way to study the details of an operator while still within the context of the graph. See Figure 2-36 on page 45.

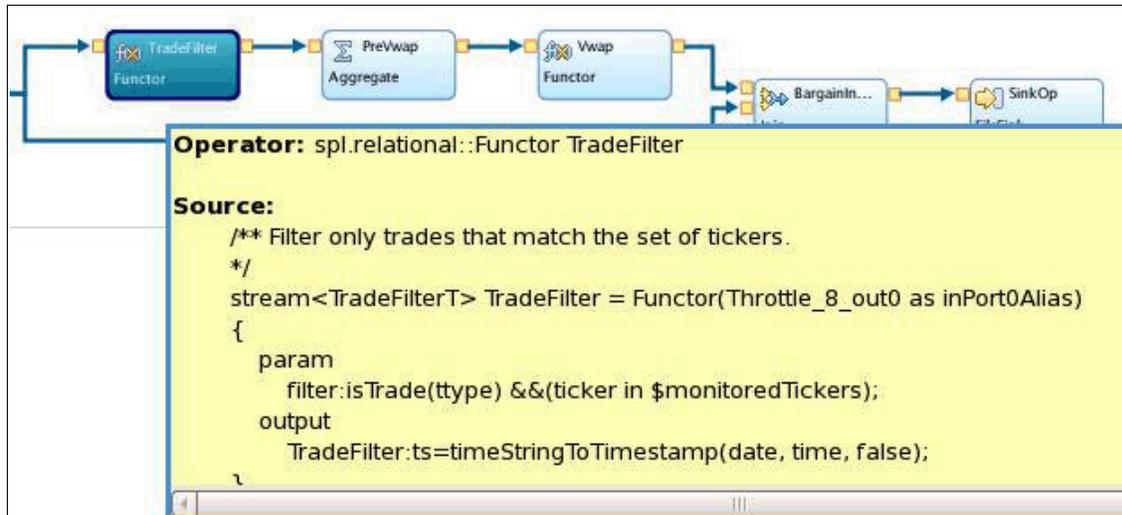


Figure 2-36 Hover for an operator

For connections, the hover function provides the schemas definition so you can see the attributes for the schema (Figure 2-37).

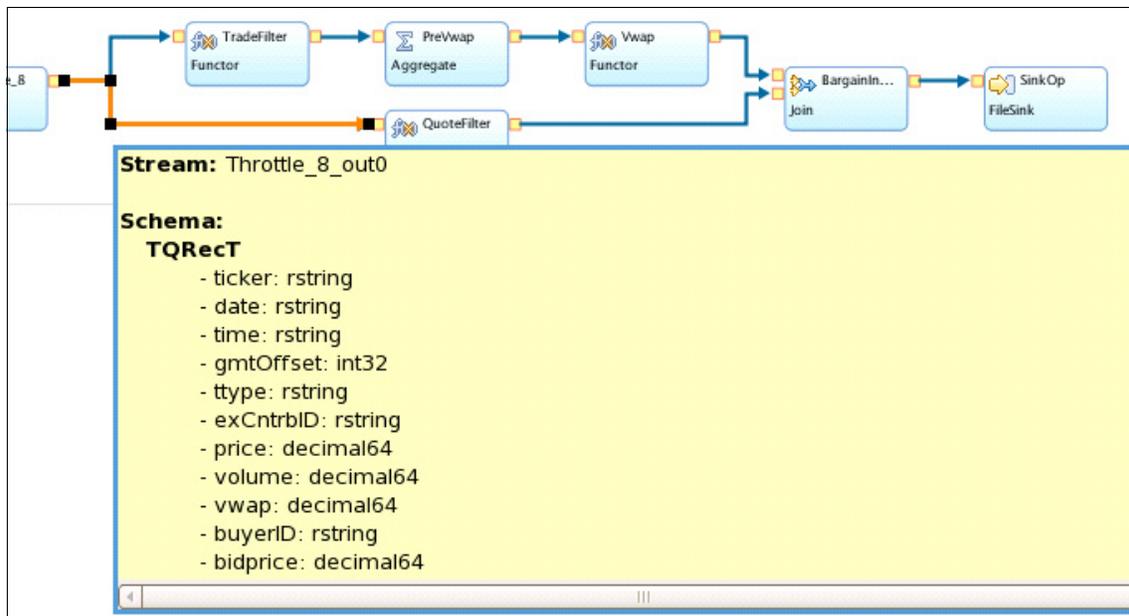


Figure 2-37 Hover for a connection

Notice that the selected connection is highlighted in orange. This helps you more easily determine, in larger graphs, which connection is selected.

Tip: Press F2 or click within the hover control to be able to scroll through the contents.

Categories

The graphical editor allows you to specify a category for an operator. Each operator can belong to one category. These are typically used to classify the operator and when the application is running, allow the instance-level graph to be laid out by category. This groups operators in the same category so you can visualize the applications in the context of the categories. Common approaches for this are as follows:

- ▶ Ingest

Creating a category for all ingest-related operators helps you more easily see all the sources of input to the applications. This can help you determine whether to create some common ingest applications that then export the streams to all the other applications.

- ▶ Output

Creating a category for the output operators can be useful in determining common sink operators, and creating import-based applications that sink the data.

- ▶ Common applications

In cases where several applications are related you can create a category for the set of applications and then all operators within the applications are part of the same category.

- ▶ Attribute or stream based

Categories created to highlight specific attributes or streams helps you see all operators related to a specific stream together.

The sample application, Vwap, with some basic categories assigned to the operators is running in an instance graph, as depicted in Figure 2-38 on page 47. Each category is a container and all operators that are part of the category are placed in the container.

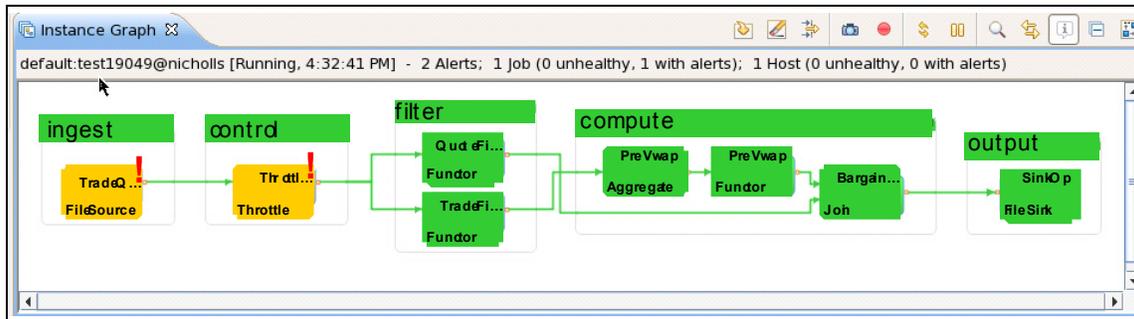


Figure 2-38 Instance graph, category layout

Tags

With the SPL Graphical Editor, you can attach tags to operators. Tags are simply a set of strings that can then be used to find related operators. There are many uses for tags. The most common cases for Steams are as follows:

- ▶ Work item identifier

When using Streams Studio with a work item-based system, a good practice is to add the work item identifier as a tag to any operator changed by the work item. This way allows searches on the tag to get a list of all operators affected by the work item.

- ▶ Attribute or streams identifiers

Adding a tag to operators that identifies the streams or specific attributes that are used by the operator helps you more easily find subflows that use a particular stream or attribute. This is helpful when refactoring or doing impact analysis and trying to determine which operators will be affected by changing a stream.

- ▶ Debug or error hints

Tags can also be used to help developers debug or do problem determination. For example adding tags that start with `checkfor_XXXXXX` lets someone quickly get a list of all the places they should check for `XXXXXX`.

Search and quick outline

With the search function, you can get a list of the elements in the graphical editor that match the search expression. This includes annotations, category, operators, export properties, input and output ports, schema elements, streams, and tags.

To use this function either click the magnifying glass icon on the editor toolbar (upper right side of the editor canvas) or press Ctrl+O to open the dialog. At the top of the dialog enter in the filter expressions using asterisk (*) for the wildcard character. The list is filtered based on the filter expression as shown in Figure 2-39.

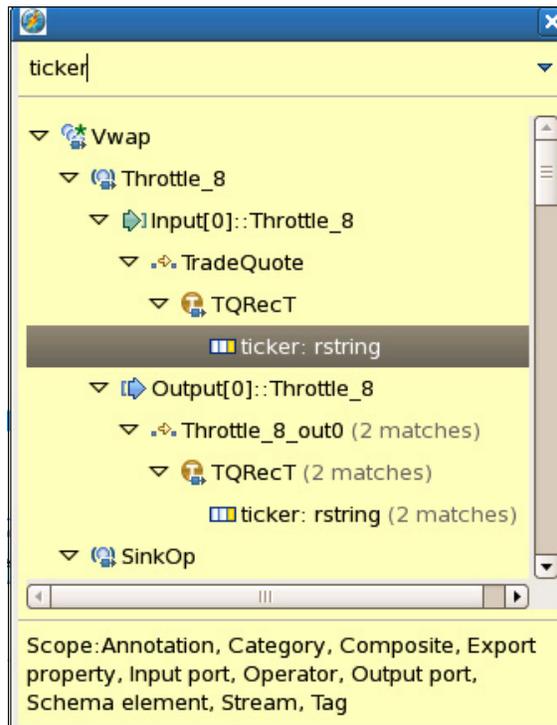


Figure 2-39 Search dialog

You can modify the search scope to further filter the results. Use this feature to find elements of a specific type such as schema elements. To open the dialog, click the drop-down arrow on the right side of the filter expression and click **Modify Scope**. Figure 2-40 shows the search scope dialog with only Tag selected. This will narrow the search scope to tags only.

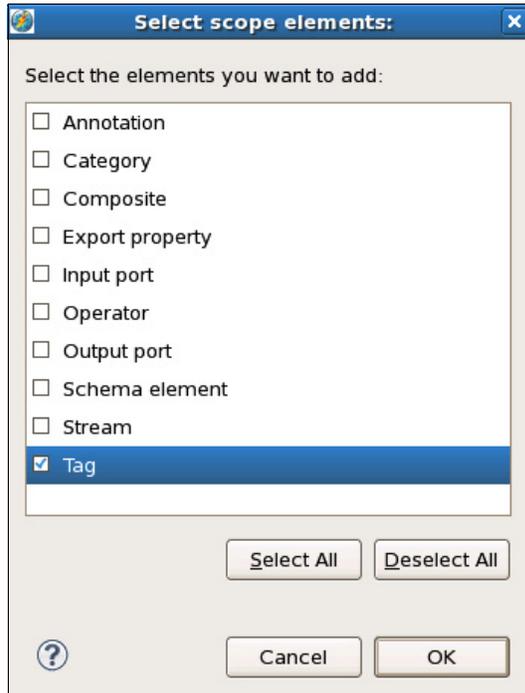


Figure 2-40 Search scope dialog

When you select an item from the dialog, all occurrences of the selection are highlighted in the graph as depicted in Figure 2-41. Notice that the scope has been narrowed to tag; by not entering anything on the filter line you can see a list of all the tags currently defined. The graph selects the two operators tagged with defect1.

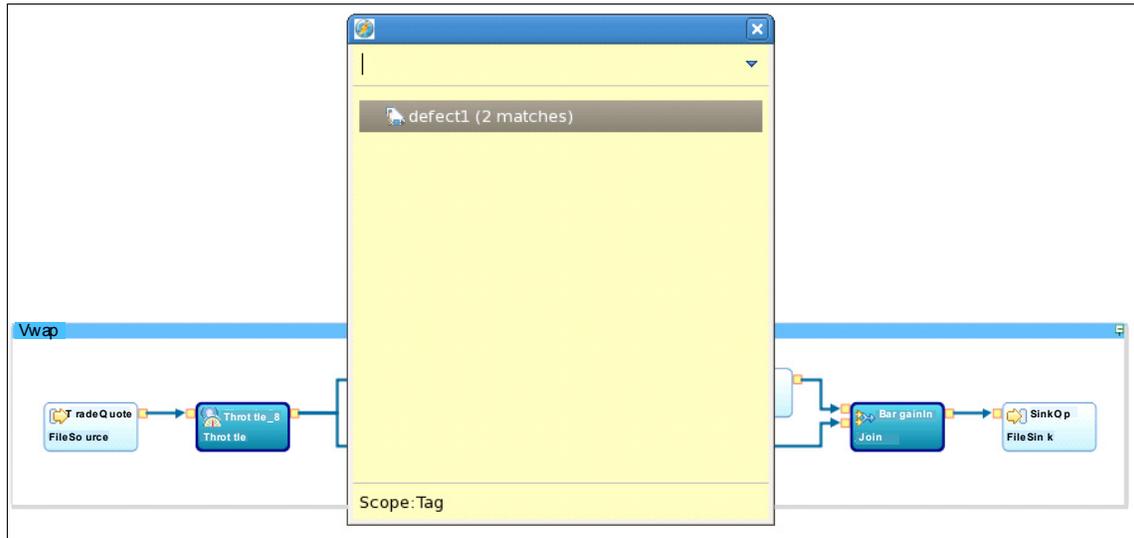


Figure 2-41 Selecting from search dialog



Visualizing stream data

Want to see the data that is flowing through your Streams application? Wonder if your application is doing the correct task? Need to check that the data is still what you expect? Worried about bad data? Need a fast, simple way to show results of your Streams application? Do not want to stop your application and add sink operators or expensive logging code?

In this chapter, we describe the InfoSphere Streams data visualizations feature, which helps you to answer those questions and more. We explain how the data visualization feature works and provide practical use cases of Streams Studio and Streams Console.

3.1 Stream data, views, and charts

Three components exist for visualizing data within InfoSphere Streams applications:

- ▶ Stream data

A stream consists of a continuous flow of data from a source. The data is in the form of tuples. Each tuple is composed of a fixed set of data attributes. An operator processes each tuple as it flows through. To use the data visualization service, the stream being visualized must be viewable. By default, all streams within a Streams application are viewable.

A stream might not be viewable if its streamViewability configuration has been set to false.

- ▶ View

A view defines how the Streams run time will sample the stream data. The definition consists of the following properties:

- Viewable stream: Identifies the stream whose tuples will be sampled.
- Attribute set: The subset of a tuple's attributes that the run time will capture for each tuple sampled.
- Buffer: Controls the number of tuples that the view will contain. The buffer size is controlled by either a count or time limit.
- Throttle: Determines the number of tuples sampled each second. It is always the first x tuples. Any tuples after the throttle limit is reached are ignored.
- Filter: The run time applies this filter to the sampled data and adds only tuples that match the filter to the buffer. The filter is applied after the sample is taken.

Together these define the view that provides the data for the charts.

- ▶ Chart

A chart defines how to present the data in the view's buffer. The types of charts supported are bar, line, and table. These are standard chart types and provide basic data visualization for stream data. The definition of a chart includes the following items:

- Type: Either bar, line, or table.
- View: Provides the data for the chart.
- Chart attributes: The titles, labels, data points and so forth. Depending on the chart, different attributes need to be specified. These are explained in detail in the examples in following sections.

3.2 How data visualization works

At its simplest level, tuples are sampled from the viewable stream and placed in the view buffer, which the chart then renders.

When a view is started, the Streams run time will begin to sample the tuples from the output port of the viewable stream for which the view is defined. Every second, the run time will sample the first x tuples of the stream. If a filter has been defined, the sampled tuples are compared against the filter, and only those that match the filter are placed in the view's buffer. If the buffer reaches its defined limit new tuples will push the oldest tuples out of the buffer.

The chart uses the view's buffer and renders a visualization based on the current content of the buffer. The chart will update based on its definition and is usually some time interval.

Figure 3-1 shows an operator's output port that has a view defined for it. As tuples flow through the output port, the first three for each second are copied into the view's buffer within the Streams Web Service (SWS). The chart associated with the view then gets the buffer contents and renders the chart.

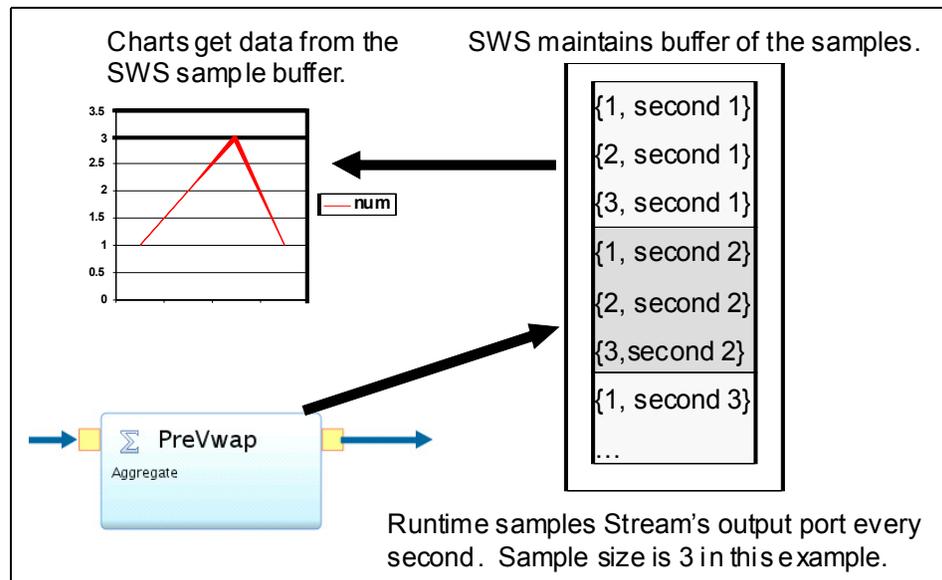


Figure 3-1 Overview of data visualization

The data visualization feature in Streams uses sampling because most charting engines cannot handle the high volume of tuples that streams applications are capable of processing in one second. The sample approach provides a

consistent sampling period that produces a consumable (both for the charting engine and for the user) amount of information. The samples provide insight into the data without having to inspect and capture every tuple.

3.3 Use cases

This topic describe two basic use cases for data visualization:

- ▶ Debugging using Streams Studio
- ▶ Application monitoring using Streams Console

Each use case describes the scenarios that apply, how to define the views and charts, and provides an example that demonstrates one of the scenarios.

3.3.1 Debugging using Streams Studio

Streaming applications by nature are data-centric; being able to sample the stream's data as it is flowing is a powerful debugging tool. By visualizing the data entering and exiting a subflow it becomes easier to validate that the subflow is correctly analyzing the data. If you are using operators from a toolkit or trying to understand an existing application, visualizing the data at various points within the application provides concrete information to help you understand how the data is being analyzed within the application flows.

Streams Studio provides the development environment for Streams applications and data visualization support has been added to the Streams instance graph view to support the debugging use case. The following scenarios use Streams Studio.

Subflow validation

Subflow validation is the basic debugging case. This typically involves visualizing data at the start and end points of a subflow within a streams application and at one or more points within the subflow. The length of the subflow can be as short as a single operator if you are validating operator function, or as long as the entire graph if you are validating application inputs and outputs. A table provides the best visualization for subflow validation.

Using the Vwap sample that is included with InfoSphere Streams, the following procedure demonstrates how to validate a subflow using a table in Streams Studio.

Vwap was modified to include a Throttle operator after the FileSource. This common debugging technique slows the streams to debug the data flowing

through them. Chapter 2, “Application programming using Streams Studio” on page 9 provides information about how to insert an operator using the Streams Studio graphical editor.

Complete the following steps:

1. Select the subflow to validate. Use the instance graph view in Streams Studio to determine the subflow. By selecting an operator or stream, you can then right-click and use the highlight functions to highlight upstream, downstream, or both directions. Figure 3-2 depicts the Vwap sample with a subflow highlighted.

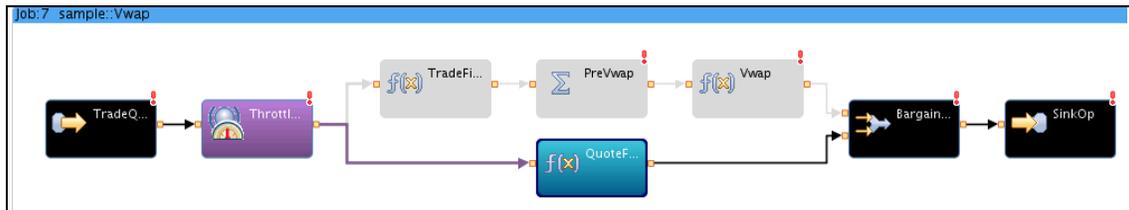


Figure 3-2 Vwap, subflow highlighted

2. Determine the starting point for validation. Select the inbound stream to the first operator in the flow you want to validate. Right-click and select **Show Data**. The dialog to define the table opens. Figure 3-3 on page 56 shows the dialog resulting from clicking **Show Data** on the output stream from Throttle operator (purple stream connection).

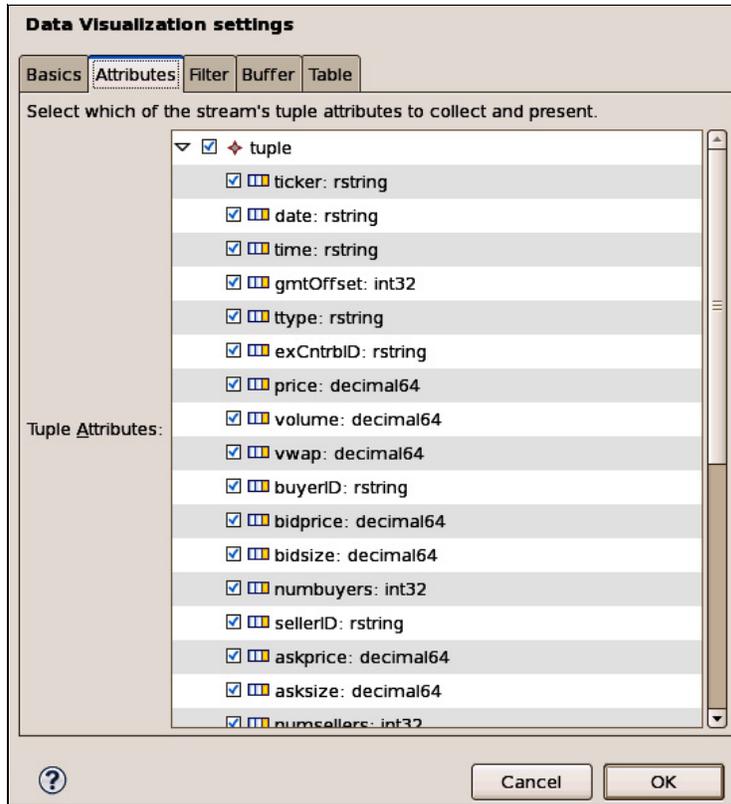


Figure 3-3 Streams Studio, Show Data dialog

If this is the first visualization, a dialog might prompt you to validate the credentials.

The Show Data dialog has tabs that provide control over the table. This is shown in Figure 3-4.

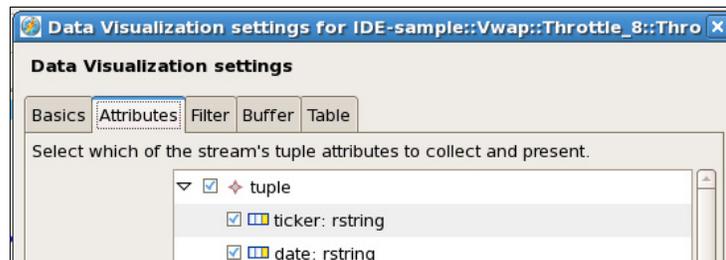


Figure 3-4 Streams Studio, Show Data dialog tabs

– Basics:

This tab is already completed with the default information based on the stream selected as shown in Figure 3-5. You should not need to change this.

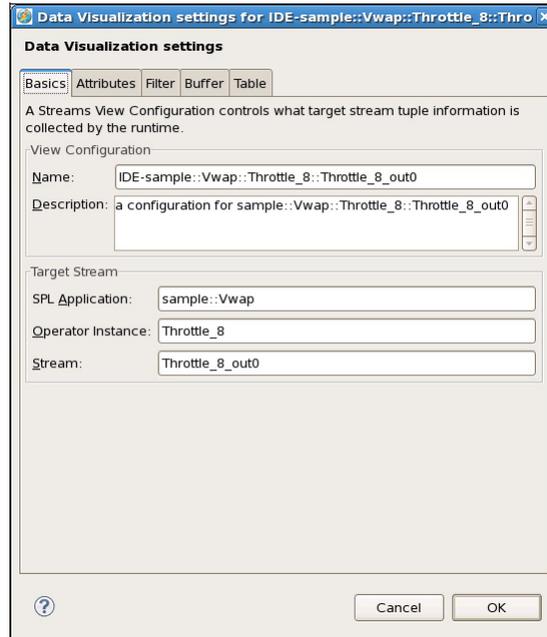


Figure 3-5 Streams Studio, Show Data Basics tab

– Attributes

This tab has focus when the dialog opens. It lists all attributes in the tuple. Select the subset of the attributes for the table to display. Each attribute will be a separate column in the table. A good practice is to select the fewest number of attributes required for validation. Too many attributes results in two problems: wide tables that require scrolling and extra overhead of transferring unnecessary attributes to the buffer.

Figure 3-6 shows a subset of the attributes selected from a tuple. In this case, only selecting ticker to validate that the downstream filter is working correctly.

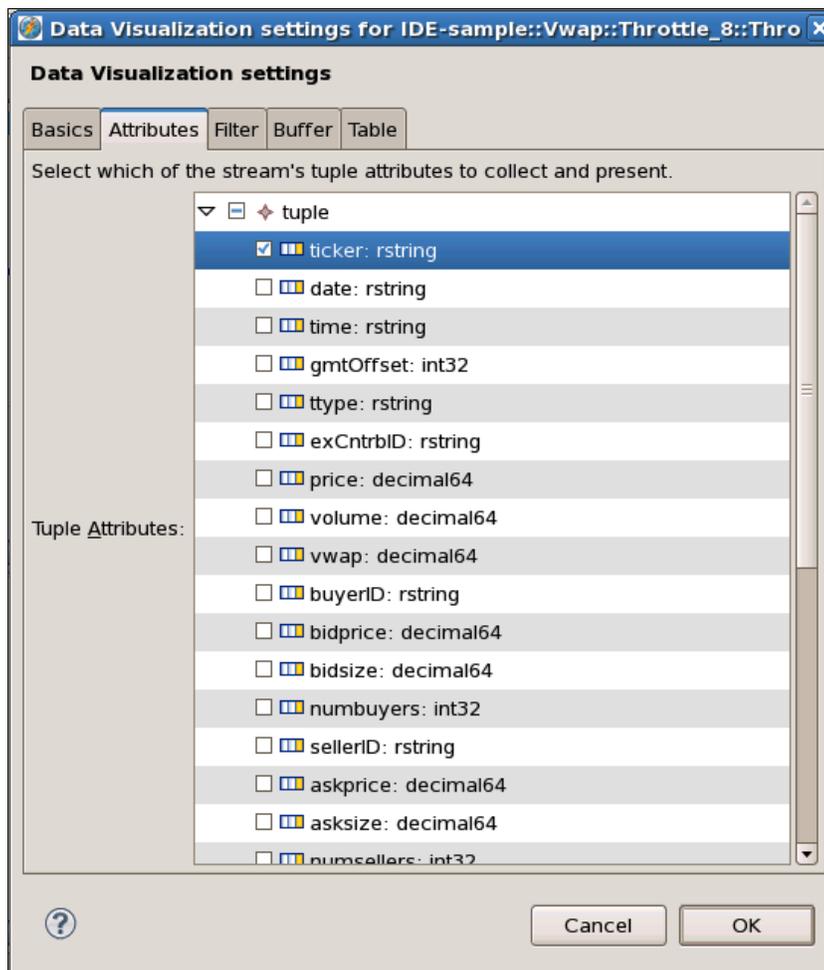


Figure 3-6 Subset of attributes

– Filter

On this tab, select an attribute and a filter expression. Only sampled tuples that match the expression are copied into the buffer. This is rarely used for subflow validation.

– Buffer

This tab controls the size of the sample and the number of tuples kept in the buffer. The buffer is a rolling set of tuples where newly sampled tuples are added to the beginning of the buffer and older tuples are ejected when the buffer reaches its limit. The size can either be a count (maximum number of tuples in buffer) or a time period in seconds. If it is a time limit, new tuples are added every sample period and expired tuples are ejected. Figure 3-7 shows a buffer with a size of 20 tuples.



Figure 3-7 Streams Studio, Show Data Buffer tab

– Table

This tab controls how often in seconds the table refreshes. For validation purposes refreshing the table every three seconds is usually sufficient.

3. Click **OK**. The table is added to the properties view for the stream. The view is detached and floats on top of the Streams Studio window. This is shown in Figure 3-8.

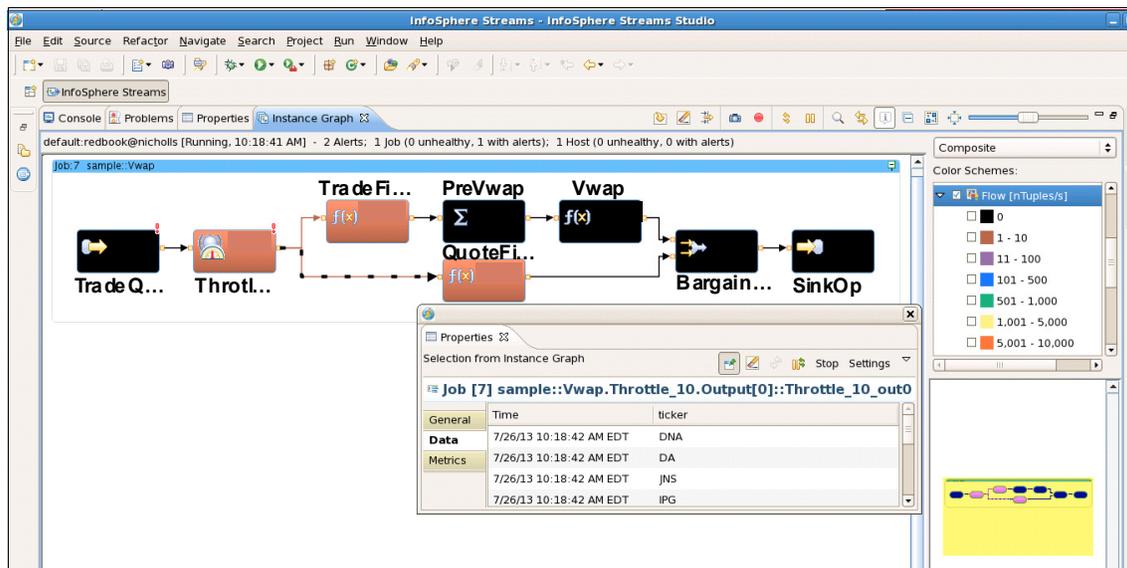


Figure 3-8 Streams Studio, monitoring data flowing into QuoteFilter

4. Repeat the procedure for the stream coming out of QuoteFilter. A second table opens and you can validate that the Filter operator is working.

Tip: When visualizing multiple streams, in this case the input and output streams to Vwap::QuoteFilter, arrange the tables on the right side of the Streams Studio window, as shown in Figure 3-9. This enables you to simultaneously validate the various flows.

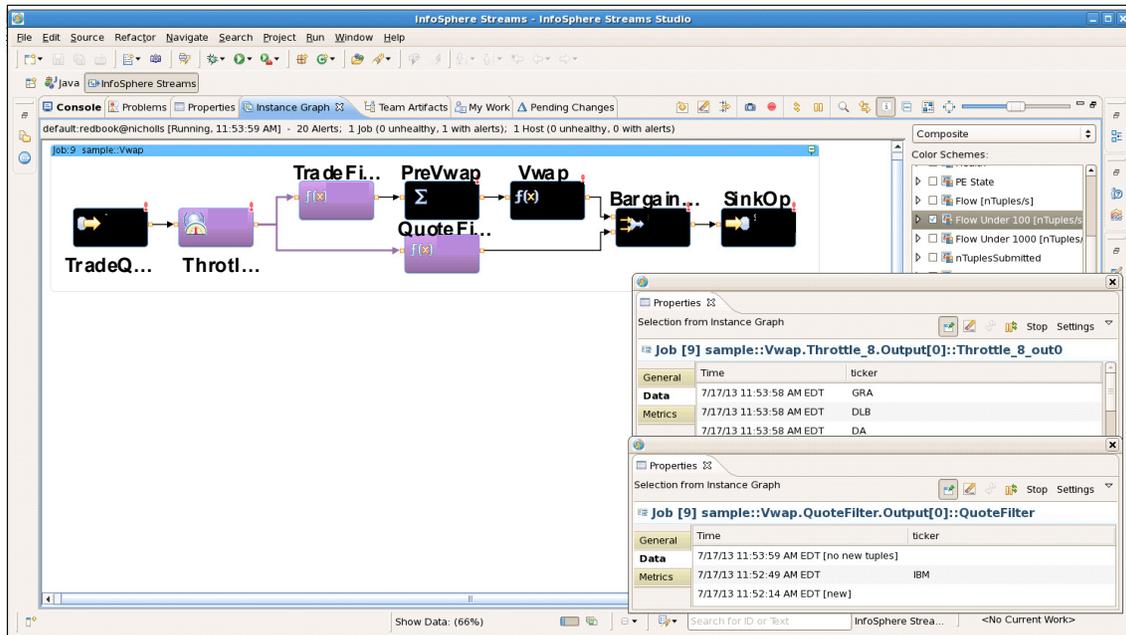


Figure 3-9 Streams Studio, monitoring input and output stream to QuoteFilter

Application understanding

Streams graphs show how the streaming data flows through the operators; you can easily determine the topology of the application. However, understanding what is happening to the data within each operator is more difficult. By using data visualization, you can inspect stream data coming into and going out of an operator. This provides a more detailed picture of what the operator does to the data.

The following procedure is one method you can use to gain better understanding of how an operator is analyzing the data:

1. Visualize the source data. Select an input stream, right-click and select **Show Data**. In the dialog that opens, click **OK**. A table is created that you can use to help you understand the input data. You can also modify the application and add a FileSink to capture the data. This file can then be copied and modified in step 4 on page 63.
2. Using the graphical editor in Streams Studio, modify your application by adding the following flow, **DirectoryScan** → **FileSource** (Figure 3-10 on page 62) as an input to the operator you want to better understand. Connect it to the input port you want to test. This will allow you to create small test files and copy them into the directory being scanned. The files are then read and

sent to the operator you are testing. The figure shows the InfoSphere Streams WordCount application with the modifications highlighted.

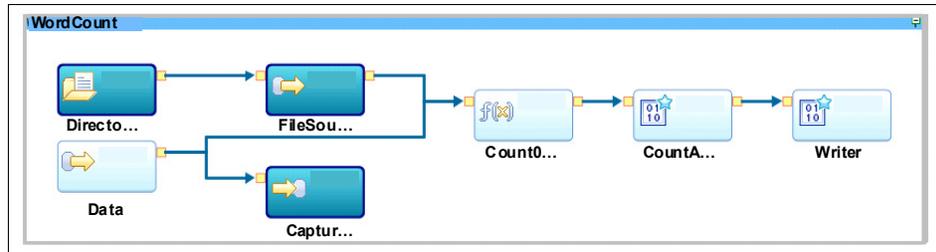


Figure 3-10 WordCount sample, modified to validate CountOneLine operator

3. After you compile and launch the application with the test subflow, select the input and output streams from the operator, right-click, and select **Show Data**. A table shows the input tuples and another table shows the output tuples from the operator. Figure 3-11 shows the input table at the top and the output table at the bottom, arranged on the right side of the Streams Studio window. The streams are shown selected on the graph.

General	Time	line
Data	7/26/13 1:39:44 PM EDT [no n...	
Metrics	7/26/13 1:37:40 PM EDT [new]	

General	Time	lines	wcs
Data	7/26/13 1:39:41 PM EDT [no n...		
Metrics	7/26/13 1:37:39 PM EDT [new]		

Figure 3-11 WordCount sample, showing data for input/output of CountOneLine

4. Create small files of test data based on your observations of the data from step 1 on page 61. In this case, each line of the file contains a string that will be the input tuple. Copy these files one at a time into the directory that the DirectoryScan operator is scanning. The test data then flows the operator and you can monitor the data in the input and output tables to gain a better understanding of how the operator analyzes various inputs. Figure 3-12 shows this technique being used to better understand the CountOneLine operator in the WordCount sample. The test is seeing what happens with special characters.

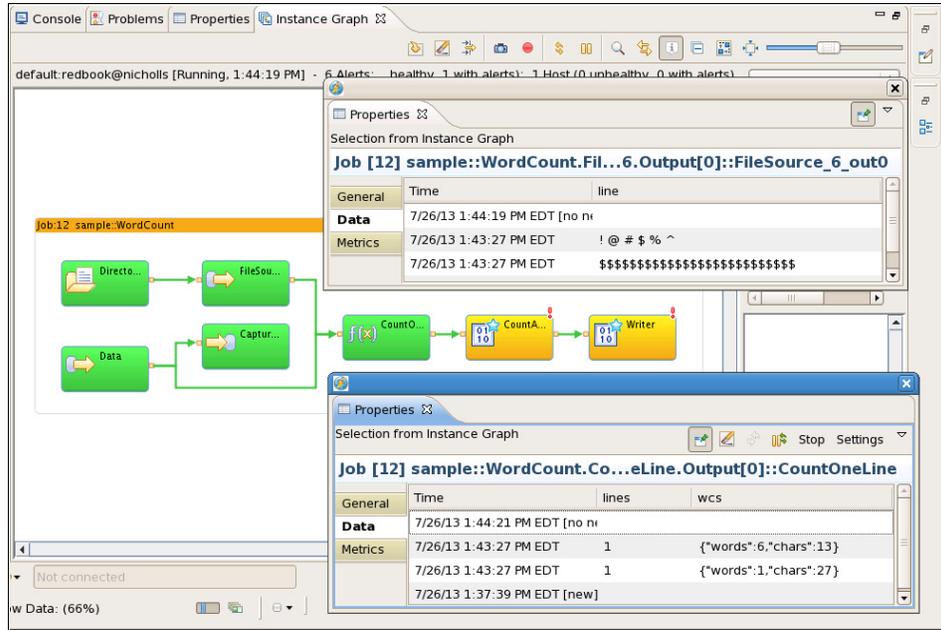


Figure 3-12 WordCount sample, data resulting from a test file

Data changes and anomalies

Determining if the data that is currently flowing through the graph is what you expected can also be done using data visualization. By visualizing the data you are able to detect data outliers or that the data currently flowing is not what is expected. This can be helpful determining that the application needs to be updated because the data has changed.

This is similar to the debugging and data understanding with the following differences:

- ▶ Select key points within the graph where the data flowing in the streams can be tested to determine if conditions are changing in the data. At these points, select a stream, right-click and select **Show Data**. Typically the entire tuple is displayed in the table, enabling more detailed checking for changing conditions.
- ▶ Filters are used extensively to try to detect anomalies in the data samples. The filter expressions are intended to match only in cases where the data no longer matches expected conditions or to indicate some sort of outlier or anomaly. The resulting tables are expected to remain empty, making it obvious if a tuple unexpectedly matches one of the filters.

3.3.2 Application monitoring using Streams Console

In this section, the data visualization is used to monitor a running Streams application. Basic graphs or a table can be used to indicate that data is flowing where it is supposed to flow. The Streams Console is used to define and start the views and charts that an administrator can use to monitor the application.

The process is similar for all the use cases. Details of the process are described in the following use case, Monitoring flow using a chart. Subsequent use cases in this section simply outline the process.

Monitoring flow using a chart

To monitor that an application is flowing correctly, key streams are identified and views are defined for those streams. Charts are then created and attached to the views to provide a simple and easy to understand monitor of the data flowing through the charted stream.

In this example, one of the InfoSphere Streams sample applications, Vwap, is used and a bar chart will be created that monitors the flow of tuples from the QuoteFilter operator.

The process for using a chart to monitor an application is as follows:

1. Create a view.
2. Start the view.
3. Define a chart.
4. Associate the chart to the view.
5. Open the chart.

The view may also be started when the chart is opened and requests data.

Create view

A view is used to define how the Streams run time will sample the stream. Use the following steps to create a view:

1. From Streams console, click the **Views** instance to open the Views page, as shown in Figure 3-13.

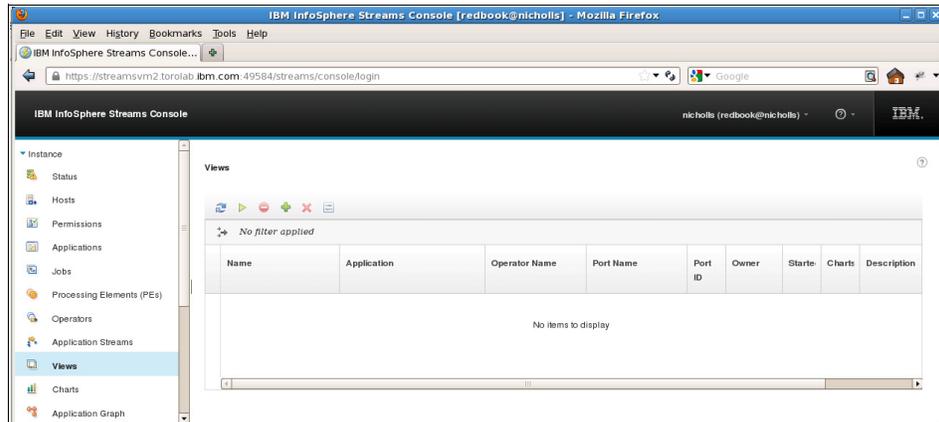


Figure 3-13 Streams Console, Views page (no views defined)

2. Click the green plus sign (+) to add a new view, as shown in Figure 3-14. This starts a dialog with a series of forms that provide the view definition. Click **Next** to proceed through the forms to complete the information to define the view.

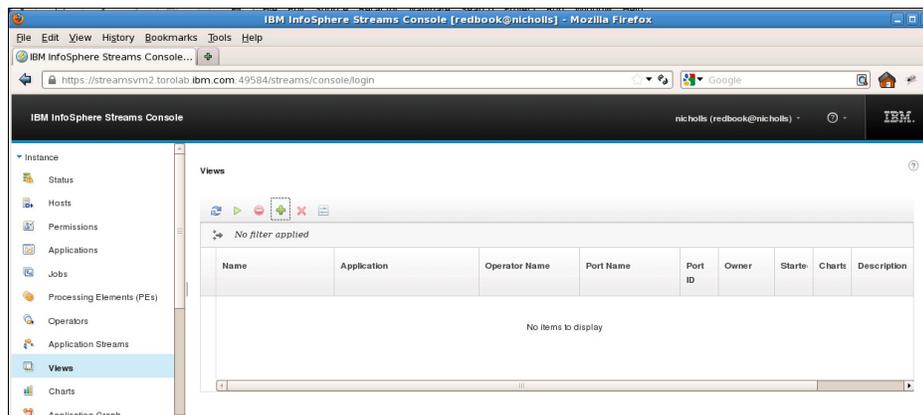
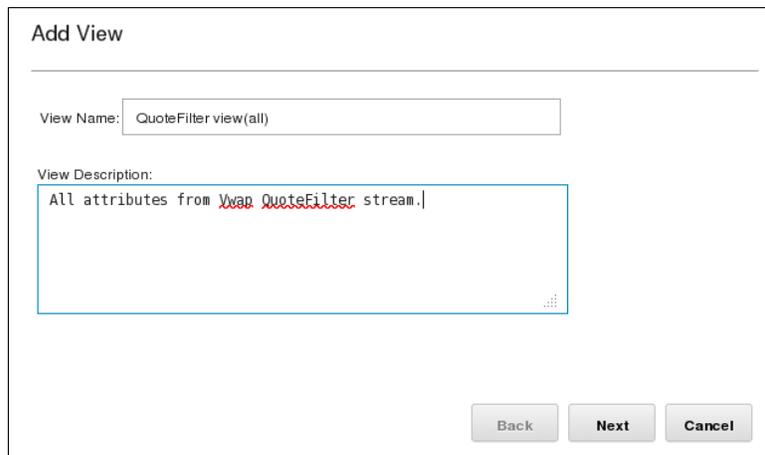


Figure 3-14 Views page, add a view

3. Give the view a name and optionally provide a description. Figure 3-15 shows an example. A good practice is to make the names descriptive and provide a small description to help identify the view.



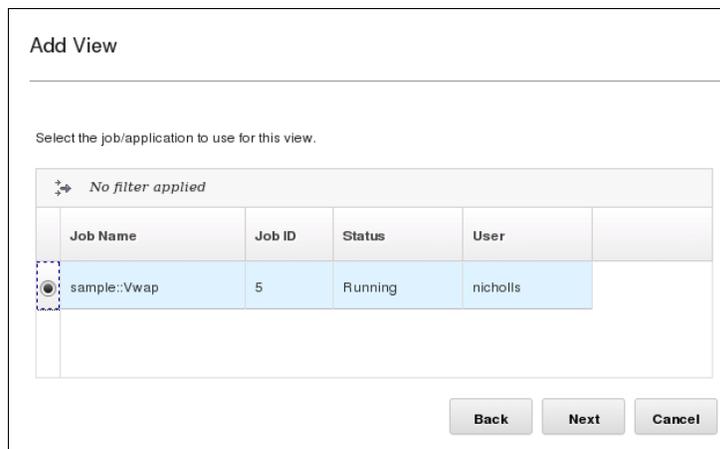
Add View

View Name:

View Description:

Figure 3-15 Add View, basics

4. Select the job that contains the stream to be viewed from the list of running jobs, as shown in Figure 3-16. A job must be running to create a view.



Add View

Select the job/application to use for this view.

Job Name	Job ID	Status	User
<input checked="" type="radio"/> sample::Vwap	5	Running	nicholls

Figure 3-16 Add View, select job

5. Select the stream from the list of streams in the job. Scroll through the list to find the stream. The streams are listed by operator and output port.

Figure 3-17 shows selection of the QuoteFilter stream.

The screenshot shows a dialog box titled "Add View". Below the title bar, there is a horizontal line and the instruction "Select the stream to use for this view." Below this is a table with a header row and three data rows. The first row is highlighted in light blue and has a radio button selected. The table has columns for "Operator Name", "Operator ID", "Port", and "Port Index". Below the table are three buttons: "Back", "Next", and "Cancel".

	Operator Name	Operator ID	Port	Port Index
<input checked="" type="radio"/>	TradeFilter	1	TradeFilter	0
<input checked="" type="radio"/>	QuoteFilter	2	QuoteFilter	0
<input type="radio"/>	PreVwap	3	PreVwap	0

Figure 3-17 Add View, select stream

6. Select the attributes from the stream that the run time will collect for each tuple sampled. If you want all the attributes, select the check the box next to the **Name** and **Type** row. You should select only the attributes you need for any charts associated with this view. This way helps limit the amount of data sampled. Figure 3-18 shows a view with all attributes of the stream selected.

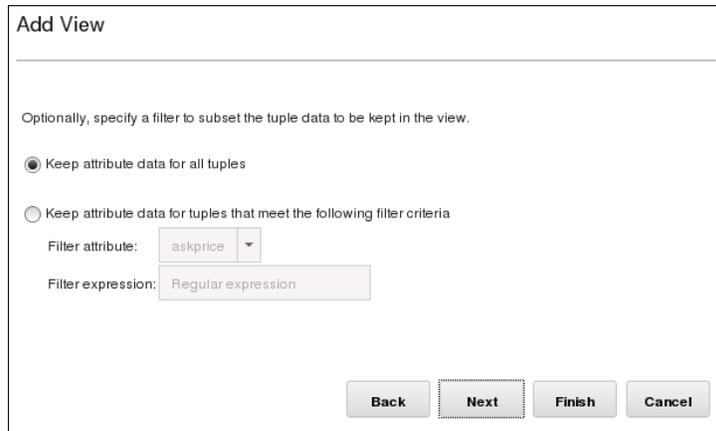
The screenshot shows a dialog box titled "Add View". Below the title bar, there is a horizontal line and the instruction "Select the attributes to include in the view." Below this is a table with a header row and three data rows. The first row is highlighted in light blue and has a checkmark in the first column. The table has columns for "Name" and "Type". Below the table are three buttons: "Back", "Next", and "Cancel".

	Name	Type
<input checked="" type="checkbox"/>	askprice	decimal64
<input checked="" type="checkbox"/>	asksize	decimal64

Figure 3-18 Add View, select attributes

7. Choose to keep all tuples in the sample, or filter the sample. You can filter by only one attribute. Enter the regular expression for the filter. Only tuples in the sample that match the filter will be available in the view. This is typically used

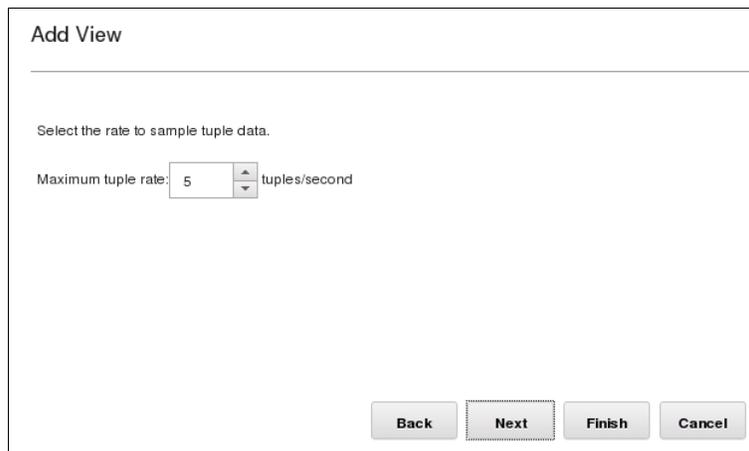
in use cases where the filter is not expected to match anything. The view should be empty and if not there is unexpected data in the stream. Figure 3-19 shows the filtering form.



The screenshot shows a dialog box titled "Add View". Below the title bar, there is a horizontal line. The text "Optionally, specify a filter to subset the tuple data to be kept in the view." is displayed. There are two radio button options: the first is "Keep attribute data for all tuples" (selected), and the second is "Keep attribute data for tuples that meet the following filter criteria". Below the second option, there is a "Filter attribute:" label followed by a dropdown menu showing "askprice". Below that is a "Filter expression:" label followed by a text input field containing "Regular expression". At the bottom right, there are four buttons: "Back", "Next", "Finish", and "Cancel".

Figure 3-19 Add View, filter

8. Set the size of the sample the run time will take. The value is the number of tuples that will be sampled each second. The sample is always the first n number of tuples in that second. Do not set this number to be large. The data visualization support is meant to provide small samples of the streams. Taking large samples can affect performance and cause charting problems. Figure 3-20 shows the form with the default values.

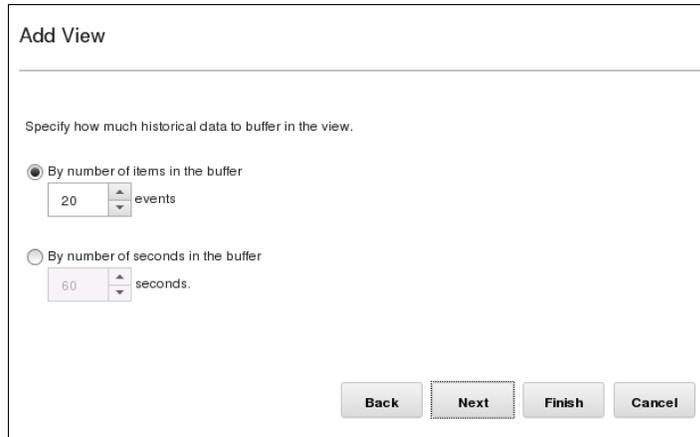


The screenshot shows a dialog box titled "Add View". Below the title bar, there is a horizontal line. The text "Select the rate to sample tuple data." is displayed. Below that, there is a "Maximum tuple rate:" label followed by a spinner control showing the value "5" and the unit "tuples/second". At the bottom right, there are four buttons: "Back", "Next", "Finish", and "Cancel".

Figure 3-20 Add View, sample size

9. Set the size of the view's sample buffer. You have two options:
- Select **By number of items in the buffer** to limit the buffer's size based on tuples. After the maximum number of tuples are in the buffer, new tuples are added to the start of the buffer and the oldest tuples are ejected.
 - Select **By number of seconds in the buffer** to limit the size of the buffer based on time. Any tuple older than the time period is ejected. This method can result in varying numbers of tuples in the buffer.

Figure 3-21 shows the default values for sample buffer.

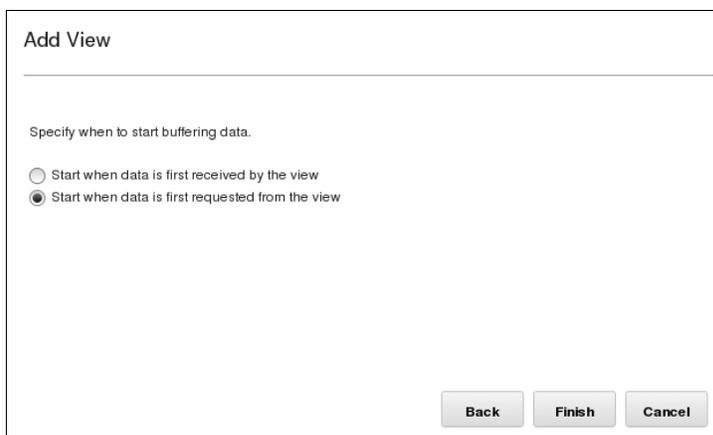


The screenshot shows a dialog box titled "Add View". Below the title bar, there is a horizontal line. The text "Specify how much historical data to buffer in the view." is displayed. There are two radio button options. The first option, "By number of items in the buffer", is selected. It has a text input field containing "20" and a dropdown menu showing "events". The second option, "By number of seconds in the buffer", is unselected. It has a text input field containing "60" and a dropdown menu showing "seconds". At the bottom right of the dialog box, there are four buttons: "Back", "Next", "Finish", and "Cancel".

Figure 3-21 Add View, buffer size

10. Select when the buffer will start to accumulate tuples:
- Starting when data is first received by the view causes the buffer to immediately begin to accumulate tuples after the view is started.
 - Starting when data is requested causes the buffer to begin accumulating data only when a chart is active and requesting data.

Figure 3-22 shows the default value.



The 'Add View' dialog box contains the following text and controls:

Add View

Specify when to start buffering data.

Start when data is first received by the view

Start when data is first requested from the view

Buttons: Back, Finish, Cancel

Figure 3-22 Add View, starting point

11. Click **Finish**. The view is defined and you can select the view and press the Start icon on the toolbar. This starts the run time, sampling the data from the stream. Figure 3-23 shows the view page with the created view selected; **Start** (the toolbar icon that is selected) was clicked.



The 'Views' page shows a toolbar with icons for refresh, play, stop, add, delete, and help. Below the toolbar, it indicates 'No filter applied'. A table lists the views:

Name	Application	Operator Name	Port Name	Port ID	Owner	Starts	Charts	Description
QuoteFilter view(all)	sample:Vwap	QuoteFilter	QuoteFilter	0	nicholls	1	0	All attributes from Vwap QuoteFilter stream.

Figure 3-23 Views page, created view, started

Define chart

A chart uses a view to get data, which it then renders in a line or bar chart based on the chart's definition. The steps for defining a chart are as follows:

1. From Streams Console, click **Charts** (as shown in Figure 3-24) to open the Charts page. Click the **Add chart** icon at the top to open the Add Chart dialog.

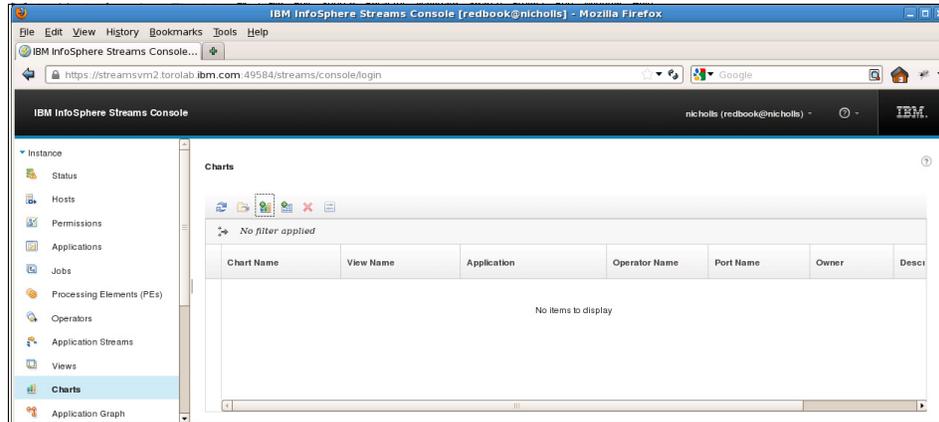


Figure 3-24 Streams Console, Charts page

2. The Add Chart dialog, first page, contains general information about the chart and the view from which the chart will get data. The fields are as follows:
 - Chart name: The name of the chart is not visible on the chart but is used on the Streams Console charts page to help identify the chart.
 - Chart title: Select either **Top** or **Bottom**, depending where you want the title of the chart to be displayed.
 - Chart type: Select either **Line** or **Bar** from the drop-down control.
 - View: Click **Select** and then choose the view from the list. This selection will be used to complete the **Application**, **Operator**, and **Stream** fields.
 - Chart description: Enter an optional description; the description will not be visible on the chart. It is used to help identify the chart on the charts page of the Streams Console.

Figure 3-25 shows the basic information completed for a bar chart.

The screenshot shows a dialog box titled "Add Chart" with the following fields and options:

- *Chart name: Vwap::QuoteFilter Monitor(Bar)
- Chart title: Vwap::QuoteFilter Prices (with radio buttons for Top and Bottom, where Top is selected)
- Chart type: Bar (dropdown menu)
- *View: QuoteFilter view(all) (with a Select... button)
- Application: sample::Vwap
- Operator: QuoteFilter
- Stream: QuoteFilter
- Chart description: Make sure only filtered stocks are shown (IBM, GOOG, MSFT) and both prices are positive and present. (text area)

* denotes a required field

Buttons: Back, Next, Cancel

Figure 3-25 Add Chart, basics

3. Specify information for the y-axis of the chart (Figure 3-26 on page 73):

- Title: This label is displayed vertically along the y-axis.
- Attributes to plot chart section, on the chart's y-axis. Select the attribute or attributes and click **Add**, **Edit**, or **Remove** to create the list of available tuple attributes from the view that is associated with this chart. Selecting more than one attribute produces multiple lines or bars.
- Data range options:
 - Automatically calculate data range: Causes the chart to adjust the y-axis data range as data points are plotted.
 - Data will normally fall between: You specify a range for the y-axis. Points outside the range are plotted off the graph.
- Show a horizontal line at: Places a horizontal line on the graph at the value specified on the y-axis. This setting is useful to better visualize the points distance from a specific value.

- Tick marks: These are shown along the y-axis of the graph. Unless you require specific spacing, use the default values.
- Refresh rate: Defines how often, in seconds, the chart will be refreshed to show new data points.

Figure 3-26 shows the chart's y-axis information.

The screenshot shows the 'Add Chart' dialog box with the 'Y Axis' section expanded. The 'Title' field contains 'Price (\$)'. Under '*Attributes to plot on chart:', there is a table with three rows: 'askprice' (Label: 'askprice', attrType: 'decimal64') and 'bidprice' (Label: 'bidprice', attrType: 'decimal64'). Below the table are options for 'Automatically calculate data range' (selected), 'Data will normally fall between' (0 and 1,000), and 'Show a horizontal line at' (0). The 'Tick marks' section has 'Major' and 'Minor' both checked, with 'Spacing' set to 'Calculate'. The 'Refresh rate' is set to 5 seconds, and 'Start when chart is opened' is checked. Navigation buttons 'Back', 'Next', and 'Cancel' are at the bottom.

Figure 3-26 Add Chart, y-axis

- Specify information for the x-axis of the chart (Figure 3-27 on page 74):
 - Title: The title is centered and displayed horizontally along the x-axis.
 - Events to plot options:
 - Scroll n most recent events: The chart will show the most recent events. The number of events specified will be the maximum number of points on the x-axis that are plotted. After the value is reached, a new point is displayed on the far right of graph and the oldest point at the y-axis is removed. This causes the graph to continually scroll, showing the most recent events.

- Only the single most recent event: This is used in cases where the attributes being plotted are summary or aggregate information. In these cases, only the most recent event is relevant.
- Label source options:
 - No labels: No labels are used on the x-axis.
 - Event time: Set the label to the time value for the event.
 - Attribute: Set the label to be the value of the tuple's attribute.
 - Labels: Specify a list of labels. This is used in bar charts where there are stacked bars for each tuple. You can also enter a single string that provides a more descriptive label than the attribute's label.
- Label orientation: Select the style (**Horizontal** or **Slanted**) of the label for the data points on the x-axis. Use slanted if you have large labels and many points of the x-axis.

Figure 3-27 shows the chart's x-axis information.

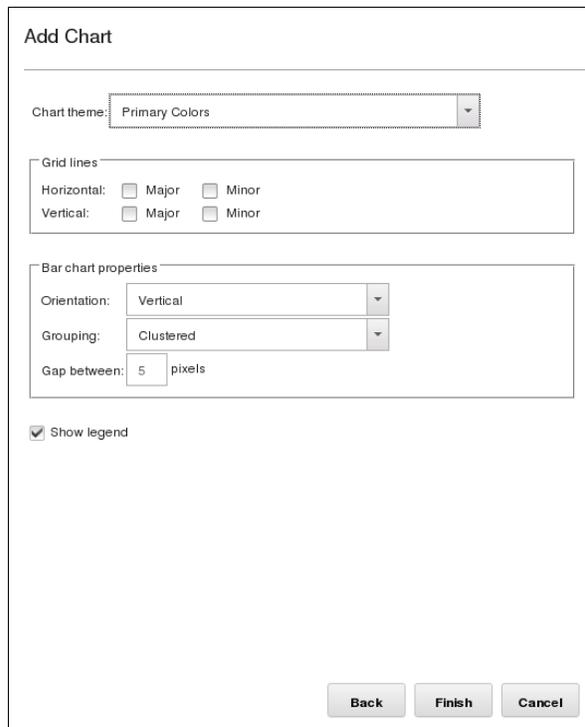
The screenshot shows a dialog box titled "Add Chart" with the following configuration options for the X Axis:

- X Axis Title:** A text input field containing the word "Stock".
- Events to plot:** A section with two radio button options:
 - Scroll: 20 most recent events (The number "20" is in a small input field).
 - Only the single most recent event
- Label source:** A section with four radio button options:
 - No labels
 - Event time
 - Attribute: ticker (The word "ticker" is in a dropdown menu).
 - Labels: Below this option is an empty list box with "Add...", "Remove", "Up", and "Down" buttons to its right.
- Label orientation:** Two radio button options:
 - Horizontal
 - Slanted
- Navigation buttons:** "Back", "Next", "Finish", and "Cancel" buttons are located at the bottom of the dialog.

Figure 3-27 Add Chart, x-axis

5. Specify additional options for the chart's appearance:
 - Chart theme: Select a theme from the drop-down list. The various themes control the color of the chart's lines and bars.
 - Grid lines: Indicates whether and where (horizontal and vertical) to place grid lines on the chart. These can help determine where points are on charts that contain data points that are far from the chart's axis.
 - Line/Bar chart properties: Either mark each data point, or have the line smoothly go through the point.
 - Show legend: Select this check box if you want the legend to be shown. By default it is selected.

Figure 3-28 shows the chart's additional appearance options.



The screenshot shows a dialog box titled "Add Chart". At the top, there is a "Chart theme:" label followed by a drop-down menu currently set to "Primary Colors". Below this is a "Grid lines" section with two rows of checkboxes: "Horizontal:" with "Major" and "Minor" options, and "Vertical:" with "Major" and "Minor" options. The next section is "Bar chart properties", which includes "Orientation:" set to "Vertical", "Grouping:" set to "Clustered", and "Gap between:" set to "5 pixels". At the bottom of the dialog, there is a checked checkbox labeled "Show legend". Three buttons are located at the bottom right: "Back", "Finish", and "Cancel".

Figure 3-28 Add Chart, additional appearance options

6. Click **Finish**. The chart is now defined.

To display a chart, select the chart from the chart page and click the **Open Chart** icon, as shown in Figure 3-29.

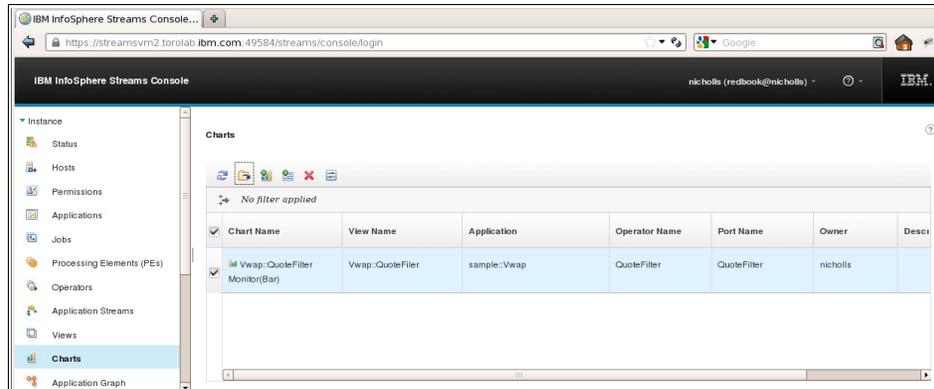


Figure 3-29 Streams Console, open chart

When the chart opens (Figure 3-30 on page 77), you can monitor it to ensure the data is actually flowing and appears correct. The view and chart that were created allow the monitoring of the QuoteFilter stream. The following information can be monitored in the sample:

- ▶ Only specific stock tickers are present. The Vwap sample application filters the transactions based on ticker symbol. From the chart, you can determine that not all tickers are present.
- ▶ There should always be a bidprice and askprice. If one is missing the entire bar is not present.
- ▶ You can determine that bidprice and askprice are close in value.
- ▶ Data is flowing because you can see the chart changing as new entries are added and old ones are removed.

There is fly-over support for the actual bar values as shown in Figure 3-30 on page 77.

Because many streams applications consume live data, the pattern of the data can change or the data can change over time. If the chart starts to contain data points that appear to be out of the expected ranges, or the chart stops producing new data points, this can be an indication that are problems with the application that require attention.



Figure 3-30 Open chart, Vwap QuoteFilter bar chart

Using error streams to monitor applications

Error streams are a useful programming pattern. An error stream is used to flow bad tuples or signal error conditions. These streams are not expected to have a large volume of data (and in some cases only produce tuples in error conditions). Having a data visualization table for these streams provides an excellent monitor of problems within the data. The table should be empty and any entry in the table provides an easy way to detect problems. Figure 3-31 shows a modified Vwap application. The highlighted operators, ErrorFilter and ErrorSink, are used to create an error stream and then write that stream to a file. The ErrorFilter will output only tuples whose transaction type is neither trade nor quote. By creating a view of that error stream and associating a table with it, you can monitor the stream to see if there are any tuples with an unexpected transaction type.

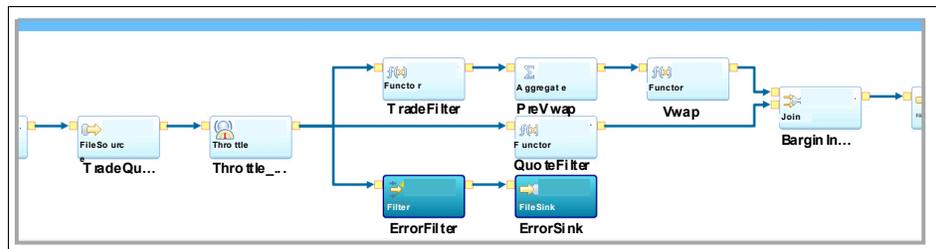


Figure 3-31 Vwap application, added error stream

The process for monitoring an error stream using a table in the Streams console is as follows:

1. Define view.
2. Define table.
3. Associate table to view.
4. Open table.

The process to define the view for the output stream from ErrorFilter is the same as was described in “Using error streams to monitor applications” on page 77. After the view is defined, you define the table that will use the view.

Define table

A table uses a view to get data, which it then renders in a table format based on the table’s definition. The steps for defining a table are as follows:

1. From Streams Console, click **Charts** to open the Charts page (Figure 3-32). Click the **Add Table** icon at the top of the page to open the Add Table dialog.

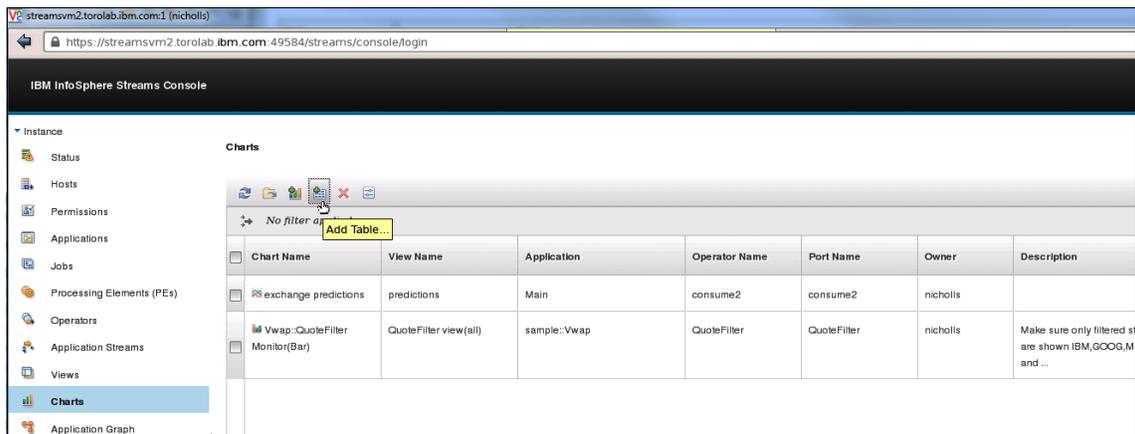


Figure 3-32 Chart page, Add Table

2. The Add Table dialog, first page, contains general information about the table and the view from which the table will get data. The fields are as follows:
 - Chart name: The chart name is shown at the top of the table and on the chart page in Streams console.
 - Chart type: This is set to Table and cannot be changed.
 - View: Click **Select** and choose a view from the list. Then, complete the Application, Operator, and Stream fields.

- **Chart description:** This field is optional and the information in it does not display in the table. It is used to help identify the chart on the charts page of the Streams Console.

Figure 3-33 shows the basic information completed for a table.

Add Table

*Chart name: Error Stream - invalid transaction type

Chart type: Table

*View: errorStream **Select...**

Application: sample::Vwap

Operator: ErrorFilter

Stream: Filter_8_out0

Chart description: Table should be empty. Any entries indicate invalid transactions in the input data.

* denotes a required field

Figure 3-33 Add Table, basics

3. On the next page, specify the remaining information for the chart:
 - Attributes to display as columns in the table: Use the control buttons (Add, Edit, or Remove) to create the list of available tuple attributes from the view that is associated with this table. Select the attributes to display. Each attribute will be displayed in a separate column in the table. Use the buttons (Up, Down) to arrange the order of the columns in the table.
 - Refresh rate controls how often the table is refreshed.

Figure 3-34 shows the completed page.

Add Table

Attributes to display as columns in the table:

<input checked="" type="checkbox"/>	Attribute	Column header
<input checked="" type="checkbox"/>	ticker	"ticker"
<input checked="" type="checkbox"/>	time	"time"
<input checked="" type="checkbox"/>	ttype	"ttype"

Refresh rate: seconds Start when chart is opened

Figure 3-34 Add Table, data details

4. Click **Finish**. The table is now defined.

To display a table, select the table from the chart page and click the **Open Chart** icon as shown in Figure 3-35.

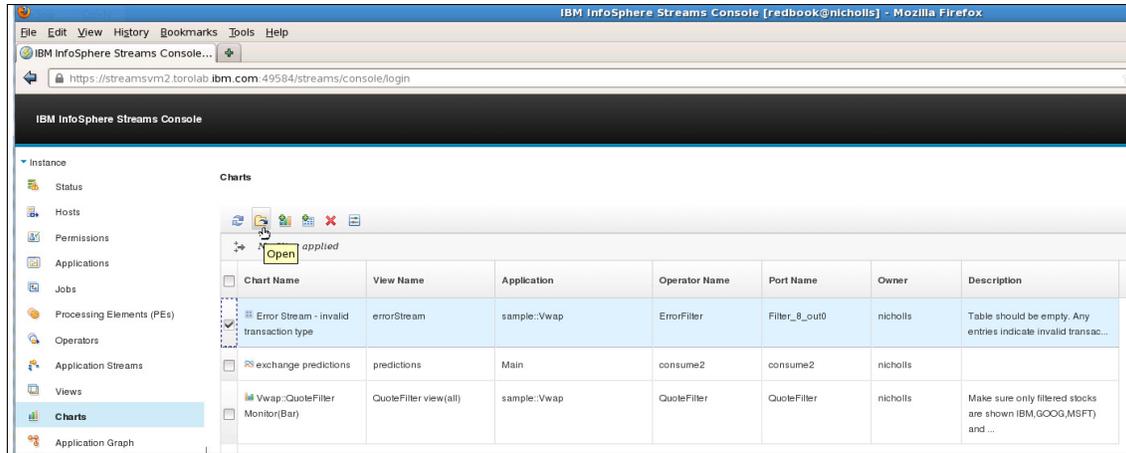


Figure 3-35 Chart page, open chart

After the table is open, as shown in Figure 3-36, you can then monitor it to watch for any tuples. The table should remain empty. If any tuples are present in the tuple, there is unexpected data in the input stream and you can take appropriate action.

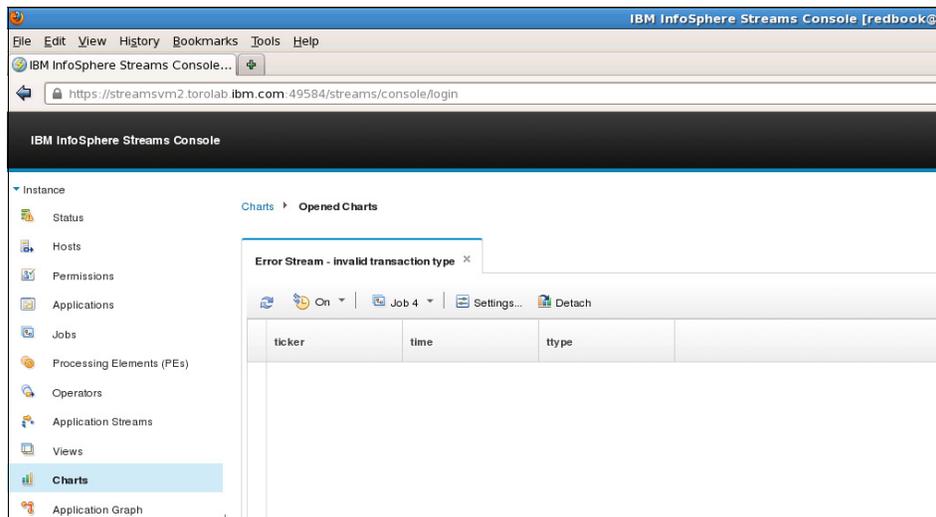


Figure 3-36 Table, monitoring an error stream

Displaying results

A streaming application has data flow from its sources through a series of operators to its sinks. This does not produce any visible results. The data visualization feature of the Streams Console can be used to provide basic display of a streams application's results.

The process for displaying results is similar to what is described in section 3.3.2, “Application monitoring using Streams Console” on page 64. The view is defined using the output stream that contains the result information. The best charting method (line, bar, or table) is chosen. The view and chart are started. The URL of the chart can be used to place the chart within a website.

Figure 3-37 uses one of the InfoSphere Streams TimeSeries Toolkit samples, ExchangeratePrediction, and creates a line chart showing the output of the consume2 operator.

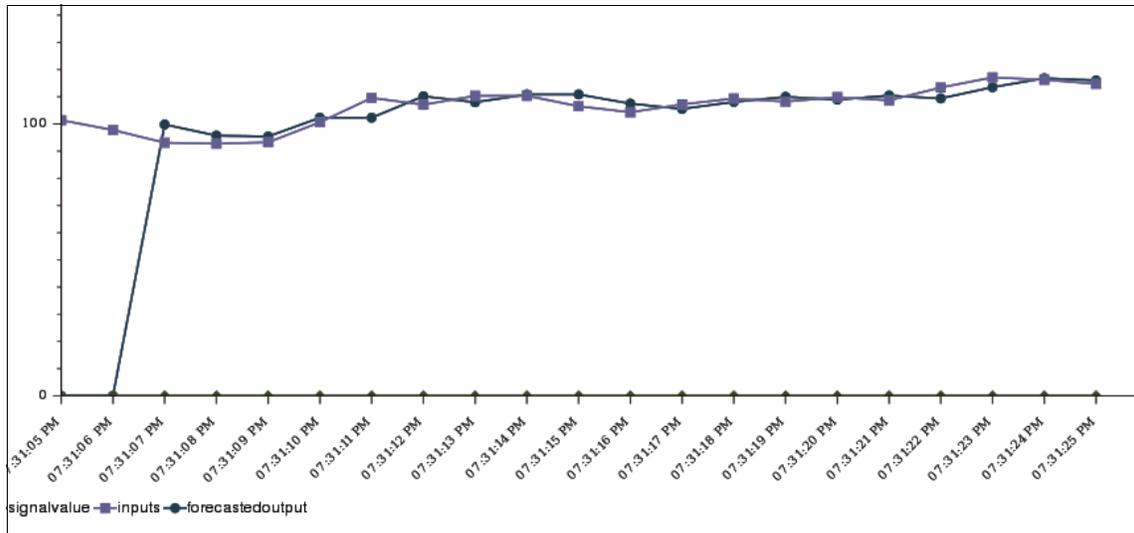


Figure 3-37 TimeSeries Toolkit, sample line chart

To use other charting engines, the best approach is to create custom chart output operators that are able to provide summary type data to the charting engine. Connecting a charting engine directly to a stream can cause issues when large volumes of information flow over the connection.

3.3.3 Hints and tips

Consider the following hints and tips for using the data visualization feature within InfoSphere Streams.

- ▶ In Streams Studio, stack the views that are related to each other vertically along one side of the window. This way helps you more easily compare the tuples in the tables.
- ▶ In Streams Studio, you can reorder the columns in the table to move more interesting attributes closer to the left. For tables defined in Streams Console, use the Up and Down control buttons to order the table's columns.
- ▶ Use views that should be empty as indicators of data changing or anomalies in the data. Using filter expression that should only match data anomalies provides a method to check the data. Error streams can work better in these cases because of the data visualization that is sampling only the first few tuples per second. Sometimes the anomalies or bad data are not present in the small samples.
- ▶ When debugging, use DirectoryScan and throttle operators to control data flowing through the application for better validation. Streams is fast; and when using file sources, the application can complete before you have a chance to open the tables for debugging. The DirectoryScan allows you to drop in new files as needed to flow more data through the application. Using a throttle rate that matches the sample size per second allows all tuples to be sampled.
- ▶ Create error streams to sink operators and monitor those streams for problems. Using an error stream out of a source operator enables you to more easily detect high volumes of bad data, which can indicate a need to update the parsers in the source operator to account for the changing source data.
- ▶ Detach charts in the Streams Console to see more than one chart at a time. By detaching the chart, you can also see the charts URL which can be copied and opened in another browser window.



Analytics entirely with SPL

The primary objective of this chapter is to help you become more aware of how easily you can do robust analytics entirely inside the Streams Processing Language (SPL). You do not have to write Java or C/C++, you do not have to use any free or additional toolkits or accelerators, you do not have to install any third-party products or other products. We often talk about these pieces to the Streams landscape, and although they might be available, you often do not need them.

In this chapter, we build a *flow rate sensor*, which is a general category of Streams application. Flow rate sensors are used on natural gas or other pipelines, oil wells, electrical grids, hospital beds, machine sensors, and machine data (database server logs, any other server logs), basically anything with a pulse that emits data you want to monitor and react to. If you need to calculate *key performance indicators* (KPIs), you probably have a form of flow rate sensor.

This chapter focuses on analytics using SPL exclusively (and more specifically describes flow rate sensors). However, you might also find benefit in this chapter because it can improve your Streams and SPL skill level from beginner to intermediate. If you understand and can implement the primary example detailed in this chapter, then you are at the intermediate skill level or beyond.

4.1 Volume, variety, and velocity

When introducing Streams to new audiences, we often use the concept of the three V's: *volume*, *variety*, and *velocity*.

- ▶ *Volume* means data that is so large that you cannot afford to capture it, whether it is a single data element or collection of data. Streams can respond to this data without data capture, or can aggregate the data before capture, and more.

Figure 4-1 (courtesy of Getty Images), shows a commercial aircraft in flight. Test flights of such aircraft, depending on the duration of the test, can generate hundreds of terabytes of data to be tracked. Such testing can easily result in a data volume tracking problem. And testing multiple flights presents an even greater data-tracking challenge.



Figure 4-1 Tractor that moves the US Space Shuttle, and other rockets

- ▶ *Variety* is meant to represent structured data (rows and columns, the type of data normally found in a relational database), semi-structured data (ASCII text data, but no standard or consistent formatting to this data), and unstructured data, whether it is binary data or entirely free-form text, or a mixture of the two. Streams handles all three categories of data mentioned here.

As the primary example in this chapter, we monitor the flow rate of four simulated Apache Web servers. The customer we were working with had 2000 Apache Web servers and more than 10,000 database servers. Although this certainly is a volume problem, we also had a variety problem; the data from all of these logs and log types was semi-structured, as we demonstrate.

- ▶ *Velocity* is meant to represent data that is moving so quickly that you cannot wait to capture this data before analyzing and responding to it. If you had a real-time sensor telling you to shut down a device before it fails, do you really

want to wait to write this to a database, commit and flush the transaction, and then query that data so that another part of the application can see it?

Figure 4-2 (courtesy of Steffes Corporation) displays a heater in a residential home. By monitoring the outside temperature, the average response rate of the heater to the outside temperature, and then any deviation from the average response, you could determine, as an example, whether a door to the outside might be open, causing excess heat loss. The Streams application you would create is a flow rate sensor. This can be a velocity problem because you want the door to be closed quickly to prevent excessive heat loss and added expense.



Figure 4-2 Home heater, front control panel has access to sensor data

4.2 Model view controller (MVC), and Streams

Model view controller (MVC) is a software architecture pattern (a design pattern) that has been widely adapted as the architecture for the Internet, and most business application development projects. Consider the following information:

- ▶ There are primarily three functionality groupings in a (business application).
 - The *model* is the persistence layer, the amount that persists data to a database or similar system. (Model gets its name from the term *data model*.)
 - The *view* is the amount of design, programming, or both, that the user or consumer views, that the user views and interacts with.
 - The *controller* is the piece that is in the middle, between the view and the model. The controller takes user-initiated events (button clicks, menu item selection, and other) and ties them to database routines, both read and write.
- ▶ The advantages of using MVC are numerous.
 - Assume you have a 1000-line application, written using MVC, and one-third of the lines are model, one-third are view, and one-third are controller. If you need to port this application from one database to another, or just test it for a new version of the same database software, effectively you have to test only one-third of your application, and not test all 1000 lines of it.
 - If you must port to a new category of viewing platform, for example a mobile device, again you only need to test or modify one-third of your application, not all of it.
 - Through localization by function, MVC promotes code reuse, ease of maintenance, and many other good, cost savings, and quality improvement aspects.

Streams has a design pattern similar to MVC. Streams has the design pattern of *ingest-(analyze/predict)-react*.

- ▶ A design pattern with Streams is to receive or ingest the data source, then make it available to one or more consumers inside the Streams run time. Ideally hundreds of data sources are ingested, and categorized to consumers by topics, keywords, or both.
- ▶ The ingest phase typically does not reject data. A consumer might ultimately want to monitor or react to obviously bad data. Or, you may choose to provide clean ingested data sources, and also raw sources.

- ▶ After ingest, numerous consumers can choose to analyze one or more (ingested) data sources. The analyze phase can also include predictive analytics (analyze/predict).
- ▶ Then react after we are aware of a condition or pattern: what action, real time preventive action or other, should be taken?

In the example used in this chapter, a flow rate sensor based on Apache Web server log events, we structure our Streams application using the design pattern of ingest-(analyze/predict)-react. After we ingest the simulated Apache Web logs (and parse/format it), we then make it available to any number of consumers by topic, and by descriptive keywords.

Figure 4-3 displays the analyze portion of our sample Streams application.

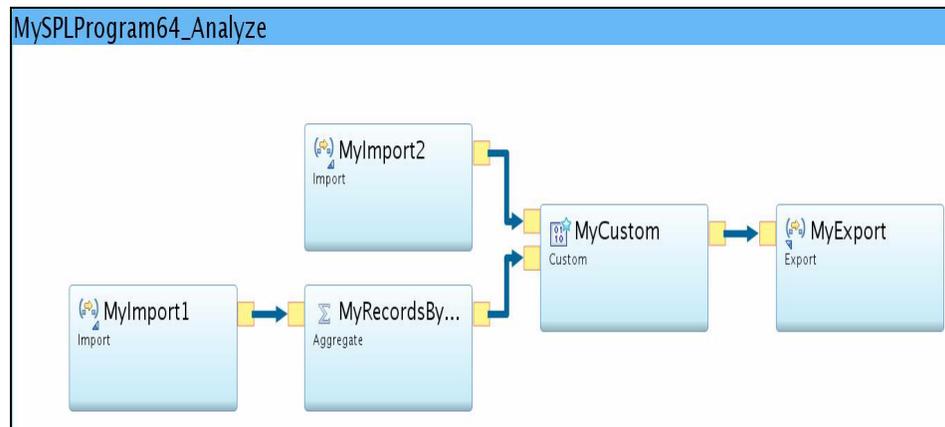


Figure 4-3 The analyze portion of the Streams application we create in this chapter

Consider the following information about Figure 4-3:

- ▶ Our Streams application is *de-coupled* from the data source that feeds it, and from any application (any targets) which might want to react to its findings, its analysis, or predictions. These data sources and targets can change or improve with little or no impact to our analysis routines. Also, we can add or drop more analysis routines at will, in full multi-user mode, to this same data source or others.
- ▶ After the data is analyzed, it flows down stream to any single, set, or Streams applications that respond, or that react to whatever we detect or want to have a response to. Again, these reacting Streams applications can change, or be added or dropped in full multi-user mode as we see as appropriate.
- ▶ The operators named MyImport1 and MyImport2 represent our sources and inputs. MyExport represents the endpoint of this routine, where any consumers can attach to our analysis routine displayed as MyCustom.

4.3 Flow rate sensor: Example application we create

Example 4-1 shows one input line from our expected Apache Web log data and the events it outputs. The example displays one physical line; however, because of its length, this one data line appears over several physical lines in the display. In our real-world test case, we had 2000 Apache Web servers all reporting to Streams. Apache Web logs have a semi-structured, albeit a reasonably well documented ASCII text format. See the code review after Example 4-1.

Example 4-1 One line of sample input data, Apache Web server.

```
10.14.246.11 122.93.102.53 - - [25/Oct/2011:01:41:00 -0500] "GET
http://www.Botera.com/index.jsp HTTP/1.1" 200 3067 "-" "Mozilla/5.0
(Windows; U; Windows NT 5.1; en-US; rv:1.9.0.19) datadatadate
Gecko/2010031422 Firefox/3.0.19"
```

Consider the following information about Example 4-1:

- ▶ The first column, containing value 10.14.246.11, represents the Apache Web server IP address; the address of the machine that hosts this Apache server. In the real world, this column is not present. We know the IP address of the Apache Web server by opening a socket and communicating with it. (A network socket contains an IP address and port number.) Streams has the inherent ability to perform network I/O to talk to this Apache Web server.

Note: The first column in Example 4-1, the server IP, will display four distinct server IPs (from the first four entries of our sample data shown in Example 4-2). An IP address is composed of four numbers (four quadrants), as separated by periods.

In our sample data, the last quadrant will contain a number in the range of 10 - 13. Server 12 (10.14.246.12) will output Apache Web server event records equal to or similar to the other three servers for a period of time, then server 12 will fail and will stop outputting data.

The flow rate sensor of our sample Streams application will detect this condition and report it to any consumers.

- ▶ The second through last columns are all standard in an Apache Web server log message. A standard Apache Web log message has a range of 8 - 80 columns, in an ASCII text, semi-structured format. Some fields are delimited by white space, some by commas, square brackets, parenthesis, a set of words contained within double quotation marks; Apache Web log records offer many fun and unique challenges.

4.3.1 Sample output from our flow rate sensor

Example 4-2 shows what was output from our flow rate sensor sample Streams application. See the code review after Example 4-2.

Example 4-2 Sample monitoring we want to output

```
"10.14.246.10",1,1,"NEW",14,14
"10.14.246.11",2,2,"NEW",14,14
"10.14.246.12",1,1,"NEW",14,14
"10.14.246.13",2,2,"NEW",14,14
"10.14.246.10",13,12,"EXISTING",15,15
"10.14.246.11",20,18,"EXISTING",15,15
"10.14.246.12",33,32,"EXISTING",15,15
"10.14.246.13",29,27,"EXISTING",15,15
"10.14.246.10",49,36,"EXISTING",16,16
"10.14.246.11",61,41,"EXISTING",16,16
"10.14.246.12",122,89,"EXISTING",16,16
"10.14.246.13",45,16,"EXISTING",16,16
... {Lines deleted}
"10.14.246.10",918,31,"EXISTING",40,40
"10.14.246.11",1142,47,"EXISTING",40,40
"10.14.246.13",1372,122,"EXISTING",40,40
"10.14.246.10",932,14,"EXISTING",41,41
"10.14.246.11",1200,58,"EXISTING",41,41
"10.14.246.13",1500,128,"EXISTING",41,41

"10.14.246.12",1537,0,"EXPIRED",36,42

"10.14.246.10",941,9,"EXISTING",42,42
"10.14.246.11",1245,45,"EXISTING",42,42
"10.14.246.13",1634,134,"EXISTING",42,42
"10.14.246.10",977,36,"EXISTING",43,43
"10.14.246.11",1265,20,"EXISTING",43,43
"10.14.246.13",1768,134,"EXISTING",43,43
... {Lines deleted}
"10.14.246.10",1995,25,"EXISTING",67,67
"10.14.246.11",2151,4,"EXISTING",67,67
"10.14.246.13",4317,22,"EXISTING",67,67
"10.14.246.10",1995,0,"EXPIRED",67,73
"10.14.246.11",2151,0,"EXPIRED",67,73
"10.14.246.13",4317,0,"EXPIRED",67,73
```

Consider the following information about Example 4-2 on page 91:

- ▶ For clarity purposes only, excess lines are deleted, and blank lines are added to the display in the example.
- ▶ Column 1 contains the Apache Web server, the server IP address is being reported on.
- ▶ Column 2 represents an integer count of the total number of requests served by this unique Apache Web server, the server IP.
- ▶ Column 3 represents an integer count of the new number of requests served since the last line output for this unique server IP.

Note: Why do we track total requests and only-new-request counters? With these numbers we can calculate rate of change over time. With additional data sources, we can track deviation over historical normals.

- ▶ Column 4 represents this server's status, as calculated by our Streams application. Valid values here include, NEW, EXISTING, or EXPIRED. An EXPIRED server is, in our definition, an Apache Web server that has stopped reporting, and has stopped serving pages.

Note: Again, this is how server 12 (10.14.246.12) behaves. According to our sample data, server 12 will stop reporting. Our job is to detect this condition and take action.

- ▶ Columns 5 and 6 are time stamps. To keep things simple, we use an integer counter as the time stamp; we could have also used a wall clock time stamp. Column 5 is the current time, and column 6 is the last time a given server reported as ACTIVE. As we create our Streams application, a given server has 5 (count) iterations of 2 seconds to report in, otherwise it is marked as status equal to EXPIRED.

4.3.2 Flow rate sensor: First ingest, heartbeat generator

Example 4-3 on page 93 begins the example of our flow rate sensor Streams application. As usual, we start with the ingest phase of our entire application. In total, our example flow rate sensor will consist of four Streams applications and one library: two applications for ingest, one for analysis/predict, and one for react. See the code review after Example 4-3 on page 93.

Example 4-3 Beacon operator, part of our ingest phase.

```
namespace MyNameSpace_01;

composite MySPLProgram61_Heartbeat {

graph

// *****

stream <int32 MyStamp> MyBeacon60 as MyOut = Beacon(){
  logic state:
  {
    mutable int32 v_Stamp = 0;
  }
  param
    period    : 2.0;
  output
    MyOut: MyStamp = v_Stamp++ ;
}

// *****

() as MyExport2 = Export(MyBeacon60) {
  param
    properties :
    {
      MyTopic    = "Ingest_MyPlatformServices",
      MySubTopic = "Beacon60"
    } ;
}

}
```

Consider the following information about Example 4-3:

- ▶ This Streams application, part of the ingest phase, has two operators: MyBeacon60 and MyExport2.

Note: As a decoupled application, using the ingest-(analyze/predict)-react design pattern, all of our example applications should have at least one import or export operator. Import and export operators are how we link these applications together; they are the operators that specify that these applications are available to communicate with one another.

- ▶ The first operator is a beacon. Generally beacon operators serve one of two purposes in a Streams application:
 - Beacon operators can either generate data for testing or be used to supply a sample application used for education. This is not our purpose here.
 - Beacon operators can also be used to generate a heartbeat for a Streams application. What is a heartbeat, and why might we need one? The general paradigm of a Streams application is that tuples flow, and Streams monitors or takes action on those flowing tuples. But, what if no tuples flow, what if all of our data sources have failed or are unreachable? When Streams receives no tuples, it takes no actions, has no flow to monitor. We create a heartbeat, a constant and guaranteed source of tuples, so that our application will always monitor, and will always be able to react.
- ▶ Our beacon is configured to output every two seconds, and to initialize and then increment an integer counter. Although we might use (or have need for) a real time stamp (a wall clock of actual time-of-day time stamp), an integer counter works as well for us here.
- ▶ We then export the heartbeat, making it available to any consumers. We labeled this export with two topics, to aid consumers in finding this data source by topic. One topic is named `Ingest_MyPlatformServices`, which, as you might imagine, would be a standard, librated service we make available to any users of our analytics application.

4.3.3 Flow rate sensor: Actual ingest, parsing semi-structured data

Example 4-3 on page 93 is small and is the first part of our ingest phase. Example 4-4 displays the bulk of our ingest, the actual reading and parsing of an Apache Web server log. See the code review after Example 4-4.

Example 4-4 Bulk of our ingest phase, reading the actual Apache Web log.

```
namespace MyNamespace_01;

composite MySPLProgram60_Ingest {

graph

    stream<rstring MyString> MyFileRead60 = FileSource() {
        param
            file          :
                "/POT/MyFiles/03a_POT/00.InputFile.WebLog.10000Lines.txt";
            format       : line;
            initDelay    : 10.0;
    }
}
```

```

// *****
// 10.14.246.11 122.93.102.53 - - [25/Oct/2011:01:41:00 -0500]
// "GET http://www.Botera.com/index.jsp HTTP/1.1" 200 3067 "-"
// "Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US;
// rv:1.9.0.19) Gecko/2010031422 Firefox/3.0.19"

stream<t_ApacheWebLog60> MyFullWebLog60 as MyOut =
  Functor(MyFileRead60 as MyIn) {
    logic state:
    {
      mutable list<rstring> l_MyAllFieldsBySpace;
      //
      mutable t_ApacheWebLog60 r_ApacheWebLog ;
      mutable t_DateAndTime60 r_DateAndTime ;
    }
    onTuple MyIn:
    {
      l_MyAllFieldsBySpace = tokenize(MyString, " ", false);
      //
      r_ApacheWebLog.ServerIP = l_MyAllFieldsBySpace[ 0] ;
      r_ApacheWebLog.ClientIP = l_MyAllFieldsBySpace[ 1] ;
      r_ApacheWebLog.ClientMachineID = l_MyAllFieldsBySpace[ 2] ;
      r_ApacheWebLog.ClientUserID = l_MyAllFieldsBySpace[ 3] ;
      //
      r_DateAndTime = MyDateAndTime_Func60(
        l_MyAllFieldsBySpace[ 4] );
      r_ApacheWebLog.RequestDate = r_DateAndTime.RequestDate;
      r_ApacheWebLog.RequestTime = r_DateAndTime.RequestTime;
      //
      r_ApacheWebLog.TimeZone = "abc";
      //
      r_ApacheWebLog.RequestMethod = "abc";
      r_ApacheWebLog.RequestUrl = "abc";
      r_ApacheWebLog.ReturnCode = "abc";
      r_ApacheWebLog.BytesSent = "abc";
      r_ApacheWebLog.Referrer = "abc";
      r_ApacheWebLog.UserAgent = "abc";
    }
    output
    MyOut :
      ServerIP = r_ApacheWebLog.ServerIP,
      ClientIP = r_ApacheWebLog.ClientIP,
      ClientMachineID = r_ApacheWebLog.ClientMachineID,

```

```

        ClientUserID    = r_ApacheWebLog.ClientUserID,
        RequestDate     = r_ApacheWebLog.RequestDate,
        RequestTime     = r_ApacheWebLog.RequestTime,
        TimeZone        = r_ApacheWebLog.TimeZone,
        RequestMethod   = r_ApacheWebLog.RequestMethod,
        RequestUrl      = r_ApacheWebLog.RequestUrl,
        ReturnCode      = r_ApacheWebLog.ReturnCode,
        BytesSent       = r_ApacheWebLog.BytesSent,
        Referrer        = r_ApacheWebLog.Referrer,
        UserAgent       = r_ApacheWebLog.UserAgent;
    }

// *****

stream<t_ApacheWebLog60> MyThrottle60 =
    Throttle(MyFullWebLog60) {
    param
        rate: 100.0;
    }

// *****

() as MyExport = Export(MyThrottle60) {
    param
        properties :
        {
            MyTopic    = "Ingest_ApacheWebLog",
            MySubTopic = "ThrottleApacheWebLog60"
        };
    }
}

```

Consider the following information about Example 4-4 on page 94:

- ▶ This Streams application begins with a FileRead operator. In the real world, we would do a network file read; Apache Web servers can be configured to output their log events to one or more destinations, including writing to a network socket.
- ▶ The next operator in this Streams application is a *functor*. Essentially we want this functor to parse our input tuple. (Why would we make every consumer of this ingest phase application do this parsing? If this was an issue, we might offer two exports from this one ingest application: one offering the raw feed,

and another offering the parsed, but still full, feed.) The `tokenize()` function is a Streams built-in function that is used to divide an `rstring` into a *list* based on a single character or set of characters. Here we split the input tuple, a log event record from Apache, into a list based on whitespace. Regarding lists, note the following information:

- A list is one of three data types in Streams, similar to what other environments might refer to as *arrays*. Streams refers to these three data types as *collections*, which include *list*, *set*, and *map*.
- Although each of list, map, and set have their intended use, here we use only detail list. Streams defines a list as a random-access, zero-indexed (subscripted elements of a list begin counting at zero, not one), unordered collection of entities that allows duplicates.

Note: In this section, we detail the list data type collection. Example 4-6 details a read-only map, and Example 4-7 details the full read-write use of a map.

- ▶ The variable named `I_MyAllFieldsBySpace` is a Streams list: a zero-based array of the various entities from one Apache log event record. In the remainder of this operator, we place various elements of this list into distinct variables, which we then output.
- ▶ Note the reference to `MyDateAndTime_Func60()`, which is a user-defined function. This function is detailed in Example 4-6 on page 98.
- ▶ Before the export operator, a last operator is used to throttle our output. In the real world, we would process input tuples in real time, as quickly as they are received. In this sample program, which might have a human observer, we delay output through the Throttle operator. This operator is configured to output 100 tuples per second. This is not required; again, we merely wanted to create an example that a human can easily observe.

4.3.4 Flow rate sensor: React phase (simulated)

Example 4-5 displays our react phase to this Streams application. See the code review after the example.

Example 4-5 Our react application, simply a file write.

```
namespace MyNameSpace_01;

composite MySPLProgram68_React {

graph
```

```

stream<t_ServerRecord60> MyImport = Import() {
    param
        subscription      :
        MyTopic           == "Analyze_ApacheServerRate" ;
    }

() as MySink1 = FileSink(MyImport) {
    param
        file      : "/POT/MyFiles/03a_POT/68.OutputFile.txt" ;
        format    : csv;
        flush     : 1u;
    }

}

```

Consider the following information about Example 4-5 on page 97:

- ▶ Although there is not much to learn from Example 4-5 on page 97 (it shows FileSink and Import operators), we include this application so our example is complete.
- ▶ This simple application should form our react phase: Now that you have observed something in your data, what do you want to do about it?

4.3.5 Flow rate sensor: User defined functions, (read-only) maps

Example 4-6 displays a user-defined function to our Streams application. This function takes the Apache Web server supplied date and time stamp, and converts it to a format we can more easily use. We also define tuples in this one central location for ease of maintenance (define once, use in many places, provide a central point of control and maintenance). See the code review after Example 4-6.

Example 4-6 Our sample user defined function, including a map data type.

```

namespace MyNameSpace_01;

type
    t_DateAndTime60 = tuple
        <
            rstring      RequestDate   ,
            rstring      RequestTime
        >;
type

```

```

t_ApacheWebLog60 = tuple
<
  rstring      ServerIP      ,
  rstring      ClientIP      ,
  rstring      ClientMachineID,
  rstring      ClientUserID   ,
  rstring      RequestDate    ,
  rstring      RequestTime    ,
  rstring      TimeZone       ,
  rstring      RequestMethod   ,
  rstring      RequestUrl     ,
  rstring      ReturnCode     ,
  rstring      BytesSent      ,
  rstring      Referrer       ,
  rstring      UserAgent
>;

type
  t_ServerRecord60 = tuple
  <
    rstring      ServerIP      ,
    int32        TotalCount    ,
    int32        ThisCount     ,
    rstring      IPStatus      ,
    int32        LastReport    ,
    int32        CurrentReport
  >;

type
  t_ServerInMap60 = tuple
  <
    int32        TotalCount    ,
    rstring      IPStatus      ,
    int32        LastReport
  >;

// *****

public t_DateAndTime MyDateAndTime_Func60(rstring arg1) {
  //
  map <rstring, rstring> MyMapOfMonths =
  {
    "JAN" : "01", "FEB" : "02", "MAR" : "03", "APR" : "04",
    "MAY" : "05", "JUN" : "06", "JUL" : "07", "AUG" : "08",
    "SEP" : "09", "OCT" : "10", "NOV" : "11", "DEC" : "12"
  };
  //

```

```

// The " = {}" initializes this structure, a
// required modifier since this is used as a
// return value.
//
mutable t_DateAndTime r_ReturnValue =
    {
        RequestDate = "-",
        RequestTime = "-"
    };
//
mutable rstring      v_CleanString      ;
mutable list<rstring> l_AllValuesInString ;
//
mutable rstring      v_JustYear         ,
                    v_JustMonthStr     ,
                    v_JustMonthNum     ,
                    v_JustDay          ;

// Remove "[", and "]" from input string. "-" could
// not be removed, because its not leading/trailing.
// (Trim() does leading and trailing only.)
//
v_CleanString = trim(arg1,"[]");

// Now break the string into pieces by the delimiters
// of "/", ":", and space.
//
l_AllValuesInString = tokenize(v_CleanString, " /: ", false);

// Index into list begins from zero.
//
v_JustYear  = l_AllValuesInString[2];
v_JustDay   = l_AllValuesInString[0];

// Re-coding the Month so we can sort on it if need be.
//
v_JustMonthStr = upper(l_AllValuesInString[1]);
v_JustMonthNum = MyMapOfMonths[v_JustMonthStr];

// Assemble the final date value. This is a string,
// so you are seeing string concatenations below.
//
r_ReturnValue.RequestDate = v_JustYear +
    v_JustMonthNum + v_JustDay ;

```

```

// Assemble the final time value.
//
r_ReturnValue.RequestTime =
    l_AllValuesInString[3] +
    l_AllValuesInString[4] +
    l_AllValuesInString[5];

// Ignored timezone. If you wish, complete that
// portion yourself. (Currently we aren't even
// passing that portion of the value into here.
//

// And then the call to return.
//

return r_ReturnValue;
}

```

Consider the following information about Example 4-6 on page 98:

- ▶ Example 4-6 on page 98 is not a Streams application. It is a reusable amount of program logic and variable definitions that can be referenced or used by a single Streams application or set of Streams applications (or even other functions, and more). Example 4-6 on page 98 is also not an ingest, (analyze/predict), or react phase of our application; it is merely reusable code for any purpose.
- ▶ This function definition begins with several tuple definitions that are only rstrings and not really new or unique.
- ▶ Next, our function definition begins: `MyDateAndTime_Func60()` receives a single input argument of type `rstring` (`arg1`), and returns a tuple of type `t_DateAndTime`, defined previously.
- ▶ A Streams *map* named `MyMapOfMonths` is defined and initialized with values. Note the following information about maps:
 - A map is one of three data types in Streams, similar to what other environments might refer to as *arrays*. Streams refers to these three data types as *collections*, which include *list*, *set*, and *map*.
 - Although each of list, map, and set have their intended use, here we use and detail only map. Streams defines a map as an unordered list of key/value pairs.

Imagine you have a list of state/province abbreviations and the full state/province names (for example NY, New York, FL, Florida, and so on). How is a map different from arrays that are in other environments? It is

value-based, meaning you do not need to loop through the entire (array) to find “NY.” You can simply say, I want array index NY (MyMap[NY]) to receive the value of “New York.”

Note: Here we define and use a map in a read-only manner. The next example Streams application Example 4-7, uses a map in a read-write manner, which is useful.

- ▶ Find the following line in Example 4-6 on page 98:

```
v_CleanString = trim(arg1, “[ ]” );
```

The trim() function is a built in to Streams. In this context, trim() removes the square bracket pairs that are at the start and end of the input variable. The trim() function affects only leading and trailing characters; trim() cannot remove strings from the middle of a variable.

- ▶ The tokenize() function is built in to Streams. It divides a single rstring into a Streams list. The tokenize() function can accept one or more delimiters on which to divide. Here we use three values: the UNIX/Linux slash character, a colon, and a space character.

- ▶ Find the following lines in Example 4-6 on page 98:

```
v_JustMonthStr = upper ( l_AllValuesInString [ 1 ] );  
v_JustMonthNum = MyMapOfMonths [ v_JustMonthStr ];
```

The upper() function is built in to Streams to fold a character value to all upper case. The second line reads our map (MyMapOfMonths) using the input value of JAN, FEB, MAR, and so on, to index into this map, and extract the month number, 01, 02, 03, and so on.

Why do we want numbers for the months and not three-character abbreviations? Numbers are easier to sort, and also Apache outputs English three-character abbreviations for months, which is not helpful when you prefer non-English.

- ▶ The remainder of this function assembles strings, return values, and so on.

Sorting: Because streaming data never terminates, and operations such as sort and others require an end-of-file marker, how does Streams sort? If the last tuple received was the tuple that should have sorted and released first, how do you know when it is okay to sort?

Streams and other real-time environments create the concept of waves or windows: essentially, marked subgroups of tuples that are okay to sort, that identify a unit of work, and ready to be sorted and released.

In Streams (and other real-time environments) you “sort as you go,” on subsets of data. These subsets can be identified by various means: by tuple count, by elapsed time, and by others.

Sorting affects, of course, sorting, joining, and aggregate calculation.

With the variety of switches to Streams windows, twenty or more permutations can be created. At a high level, we state the following information:

- ▶ Focus on only the two window types: tumbling or sliding. Both window types process tuples. The difference is tumbling windows empty themselves of tuples every time they fire (elapsed time, row count, and others); sliding windows can keep some tuples handy after they fire.
- ▶ Determine which window type to use. There is overlap. Sliding windows might be able to more easily do certain calculations because they keep rows on handy; that is, sliding windows might more easily calculate standard deviation, which requires a history of tuples. Tumbling windows are viewed by some as easier to initially understand.

4.3.6 Flow rate sensor: Analyze, read-write maps, windows, other

Example 4-7 is the most strategic application in our flow rate sensor example; it features windowing, read-write maps, a custom operator, looping, and more. See the code review after Example 4-7.

Example 4-7 Our analyze phase of this Streams application.

```
namespace MyNameSpace_01;

composite MySPLProgram64_Analyze {

graph
```

```

// *****

stream<t_ApacheWebLog60> MyImport1 = Import() {
    param
        subscription    :
            MyTopic      == "Ingest_ApacheWebLog"      &&
            MySubTopic   == "ThrottleApacheWebLog60"   ;
    }

stream<int32 MyStamp> MyImport2 = Import() {
    param
        subscription    :
            MyTopic      == "Ingest_MyPlatformServices" &&
            MySubTopic   == "Beacon60"                 ;
    }

// *****

stream <t_ServerRecord60> MyRecordsByServerIP as MyOut =
    Aggregate(MyImport1 as MyIn) {
        window
            MyIn          : tumbling, time(2) ;
        param
            groupBy       : MyIn.ServerIP      ;
        output MyRecordsByServerIP :
            ServerIP      = MyIn.ServerIP      ,
            TotalCount    = 0                   ,
            ThisCount     = Count()             ,
            IPStatus      = "-"                 ,
            LastReport    = 0                   ,
            CurrentReport = 0                   ;
    }

// *****

stream <t_ServerRecord60> MyCustom as MyOut = Custom
(
    MyImport2            as MyIn1 ;
    MyRecordsByServerIP as MyIn2
) {
    logic state :
    {

```

```

mutable t_ServerRecord60 r_ServerRecord;
    //
mutable int32          r_Stamp = 0    ;
    //
mutable rstring        v_KeyToMap    ;
mutable t_ServerInMap60 r_ServerInMap ;
    //
mutable map<rstring, t_ServerInMap60>
    m_MyMapOfServers                ;
    //
mutable map<rstring, rstring>
    m_MyDeleteMap                    ;
}
onTuple MyIn1 :
{
    r_Stamp = MyIn1.MyStamp ;
    //
    for (rstring v_KeyToMap in m_MyMapOfServers )
    {
        if (m_MyMapOfServers[v_KeyToMap].LastReport < r_Stamp -5 ){
            r_ServerRecord =
                {
                    ServerIP      = v_KeyToMap                ,
                    TotalCount     = m_MyMapOfServers[v_KeyToMap].TotalCount,
                    ThisCount      = 0                        ,
                    IPStatus       = "EXPIRED"                ,
                    LastReport      = m_MyMapOfServers[v_KeyToMap].LastReport,
                    CurrentReport  = r_Stamp
                } ;
            submit (r_ServerRecord, MyOut);
            m_MyDeleteMap[v_KeyToMap] = v_KeyToMap;
        }
    }
    for (rstring v_KeyToMap in m_MyDeleteMap)
    {
        removeM(m_MyMapOfServers,v_KeyToMap);
    }
    clearM(m_MyDeleteMap);
}
onTuple MyIn2 :
{
    v_KeyToMap = MyIn2.ServerIP ;
    //
    if (v_KeyToMap in m_MyMapOfServers ) {
        //
        // This is an IP we have seen before.
        //
        r_ServerInMap =
            {

```

```

        TotalCount =
            m_MyMapOfServers[v_KeyToMap].TotalCount +
            MyIn2.ThisCount
        IPStatus = "EXISTING"
        LastReport = r_Stamp
    } ;
}
else {
    //
    // This is not an IP we have seen before.
    //
    r_ServerInMap =
        {
            TotalCount = MyIn2.ThisCount,
            IPStatus = "NEW"
            LastReport = r_Stamp
        } ;
}
//
// This next call updates/inserts into the
// Map. Easy peezy. There is a means to
// delete from a Map using a method called
// delete(), not used anywhere here.
//
// From the "if in" code above, you can now
// query, insert, update, and delete from a
// Map.
//
m_MyMapOfServers[v_KeyToMap] = r_ServerInMap ;
//
r_ServerRecord =
{
    ServerIP = MyIn2.ServerIP
    TotalCount =
        m_MyMapOfServers[v_KeyToMap].TotalCount
    ThisCount = MyIn2.ThisCount
    IPStatus =
        m_MyMapOfServers[v_KeyToMap].IPStatus
    LastReport = r_Stamp
    CurrentReport = r_Stamp
} ;
//
// 'submit' is the call to output from this
// Operator. 'MyOut' is the alias for the
// Output Port, and you can have multiples.
//
// With Custom Operators, is there nothing
// you can't do ?!
//

```

```

        submit (r_ServerRecord, MyOut);
    }
}

// *****

() as MyExport = Export(MyCustom) {
    param
        properties :
        {
            MyTopic    = "Analyze_ApacheServerRate"
        };
}
}

```

Consider the following information about Example 4-7 on page 103:

- ▶ This Streams application begins with two import operators: heartbeat monitor (MyImport2) and ingest phase (MyImport1).
- ▶ The aggregate operator named MyRecordsByServerIP offers the following information:
 - As configured, this aggregate operator uses a tumbling window of two seconds; every two seconds it releases a set of aggregate calculations: tumbling, time(2).
 - These calculations are done for each unique ServerIP that is received (param/groupBy : MyIn.ServerIP). So, calculations are not done on the entire stream of input tuples (all Server IPs together, as a grand total), and are instead done only for each unique Server IP.
We will receive only four unique Server IPs, however; this aggregate operator can handle tens of thousands of unique Server IPs.
 - Aggregate operators and their associated windows support several built-in functions, such as the Count() function:
ThisCount = Count();
In this context, Count() produces a count of the tuples for each unique Server IP, for this tumble of the window, which is two seconds.
 - Most output values are default values that we increment or change downstream.

- ▶ The next operator in our Streams application is of type custom, and is named, MyCustom.
 - This operator accepts two input Streams: our two ingest streams, MyImport2 (the beacon and heartbeat monitor) and MyImport1 (the Apache log file event reader). This operator outputs one stream named, MyCustom, which outputs a tuple of our analysis results.
 - With two input streams, this custom operator has two onTuple event blocks.

Consider the following information about the onTuple MyIn1 event block:

- The onTuple MyIn1 block is probably easier to understand than MyIn2; it is certainly shorter. Recall that MyIn1 is the input stream for the heartbeat monitor. On receipt of a new tuple from this input stream, basically we check history, which we maintain in the form of a map named m_MyMapOfServers.

Note: MyIn1 comes from MyImport2, and MyIn2 comes from MyImport1. We did this to reinforce that these names are only identifiers, and the numbers are not actually significant.

- Our goal here is to check for EXPIRED servers, those we have not seen in a while. If we find an EXPIRED server, we output (reference to submit), and delete this server from our history map. We do not have to delete it; this was a design choice.
- Because we do not have an incoming tuple for EXPIRED servers, we must look for one; we have to loop through history, our map of Server IPs.
- The removeM() function is built in to Streams to delete an element from a map. The clearM() function is built in to Streams to delete all elements from a map.
- Consider the following information about the onTuple MyIn2 event block:
 - The onTuple MyIn2 block is the larger of the two event blocks. There is no looping here. The reason is because this onTuple event block receives a new ServerIP. This is the stream of input tuples that are Server IPs.

With the receipt of a Server IP, we can find it in the map of history, m_MyMapOfServers. A map is by definition indexed by a key value, and that key value is our Server IP.

 - The call to output (submit), is to report Server IP activity: NEW or EXISTING.

- We can check the history of Sever IPs here, check the map for EXPIRED servers here, but why do that if our heartbeat monitor does that for us. And, the heartbeat monitor is guaranteed to arrive. Regular new tuples of server IPs may stop arriving. This is unlikely, but possible.
- ▶ the final operator is a a call to export, MyExport. Any consumer receiving this stream is receiving a precalculated list of Apache Web server event records: are these servers alive and outputting, how fast, or have they expired?

4.4 Conclusion, how to proceed

Based on the exact type of work you perform, some percentage of your analysis will be done on flow rate sensors, and some percentage will be calculating key-performance-indicators on streaming data. Previously we detailed how to monitor Apache Web server log record events. In the real world, we find we use this type of Streams application repeatedly, whether is machine data, log records, the output from medical sensors to a hospital bed, or nearly any reason.

For your skills development, you might want to create and experiment with the sample Streams application detailed in this chapter. For information about how to access the sample Streams application see Appendix A, “Additional material” on page 527.



Streams and DataStage integration

In this chapter, we describe the details of using InfoSphere Streams (referred to as Streams) and InfoSphere Information Server DataStage (referred to as DataStage). First, the chapter includes an overview of use cases where you might need to integrate Streams and DataStage. Next, it details the runtime environment that is used in the examples that follow in this chapter; the runtime environment we expect you see in the real world. Last, the chapter creates two examples: Streams to DataStage and DataStage to Streams.

We assume that you know how to create a beginning-to-intermediate level of Streams and DataStage applications. Each example we present is simple by itself: for Streams, we read a plain flat file; for DataStage, we also read a flat file. On the receiving side, the idea is the same: we write to a flat file. The reason is because this chapter is really about the configuration between these two software systems, and not about the detailed use of either system by itself.

This section is based on the following product versions:

- ▶ Streams Version 3.0
- ▶ Information Server Version 9.1

5.1 Introduction to Streams processes

Streams processes represent “data in motion” without landing the data or tuples on the disk. The pipeline parallelism DataStage provides has similar capability. The records are passed through stages without having to land them on the disk. How these tools process data is similar in many ways. As you might imagine, Streams and DataStage are a great fit and would produce even more capability by complementing each other.

Streams offers means to extend its functionality by developing custom operators. Although you can add any function you want to Streams if it does not have a ready-to-use option, it will involve C/C++ or java coding. Your team might not have enough developers with sufficient skills or might spend a long period of time developing custom operators. If you have DataStage, your Streams application can easily be extended by using a wide variety of functions and connectivity capabilities that DataStage offers. Some functions can be achieved by developing custom operators but DataStage stages, such as Data Quality standardization or match stages, can be extremely difficult to implement within a reasonable amount of time in a project. The same applies to DataStage projects. Integration of Streams and DataStage provides increased productivity and functionality.

For example, if you have a Streams application that handles customer records, you might want to enrich customer name information by calling QualityStage standardization from the Streams application before loading it to the target data warehouse. The QualityStage Standardize stage parses and normalizes name representation for individuals and organizations. That can help the data analysis improve the identification of customers. Similarly, address information can be standardized and enriched by QualityStage Standardize stage. Name and address standardization is supported for major countries by DataStage, and is read to use.

If you need to output data in a complex XML format, such as financial industry schema, DataStage has a rich set of tools to handle XML data; examples are XML Assembly Editor, XML Transformer, and XML Input and Output stages. You might be able to create an XML document with standard operators if the format is simple or if you add an external component to Streams, but with DataStage it can be done by a ready-to-use function.

DataStage also provides native connectivity to database management systems such as Netezza® and Teradata, and other major databases, which can also increase your team productivity.

In addition, functions of Streams, such as the Text Toolkit or the Mining Toolkit, significantly enhance the DataStage capability. DataStage can use those toolkits in both batch and real time.

Streams and DataStage can be integrated through IBM InfoSphere DataStage Integration Toolkit from Streams, and InfoSphere Streams Connector from DataStage. Streams can pass tuples to DataStage as part of near real-time analytic processing (RTAP). DataStage can pass records to Streams utilizing a unique capability Streams provides as part of batch or real-time job. You can also create an application that round-trips between these two technologies.

5.2 Runtime architecture

When passing data from DataStage to Streams, Streams Connector in a DataStage job sends data sets to Streams through TCP/IP. In Streams, the DSSource operator, provided by InfoSphere DataStage Integration Toolkit, receives the data as tuple and passes it to subsequent operators (Figure 5-1).

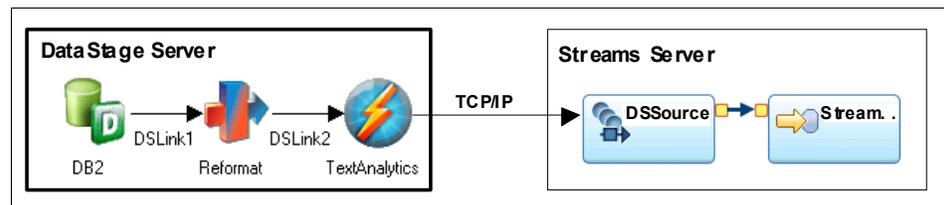


Figure 5-1 Runtime architecture: DataStage to Streams

When sending data from Streams to DataStage, the Streams application uses the DSSink operator, which is also part of InfoSphere DataStage Integration Toolkit, to send it with TCP/IP. In the DataStage job, the Streams Connector receives the data and passes it to the following stages (Figure 5-2).

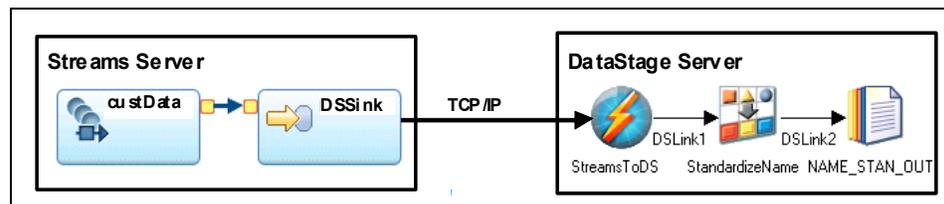


Figure 5-2 Run-time architecture: Streams to DataStage

The integration between Streams and DataStage does not necessarily have to be a one-way trip. You can also extend the architecture to have the data

round-trip between servers: Streams to DataStage and then to Streams, or DataStage to Streams and then DataStage.

Some overhead exists in sending the data across the software systems. The time spent by that overhead can be critical if a real-time application is sending the data without landing it on the disk to reduce the latency. Figure 5-3 and Figure 5-4 illustrate two such scenarios.

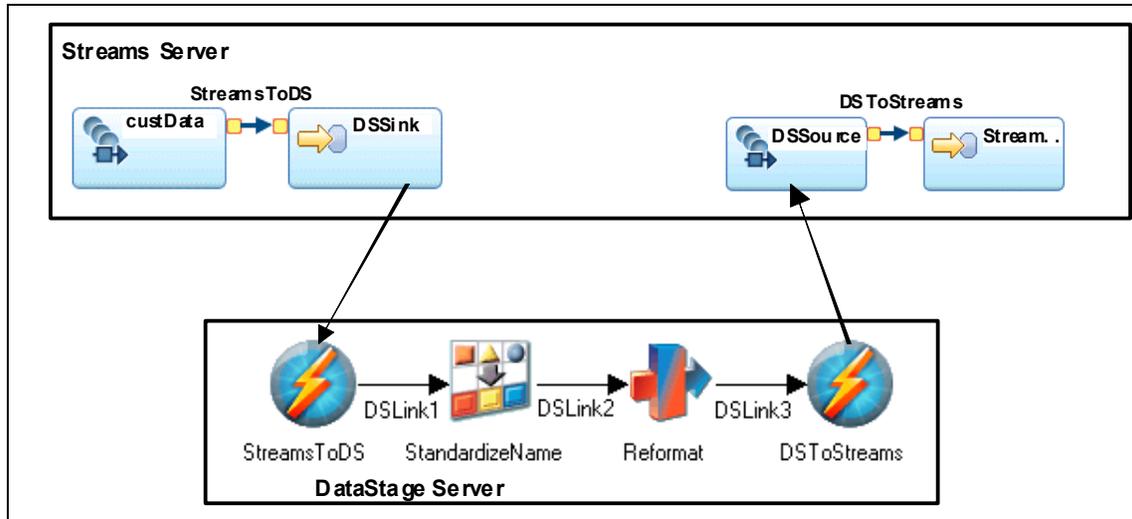


Figure 5-3 Round trip data, scenario 1

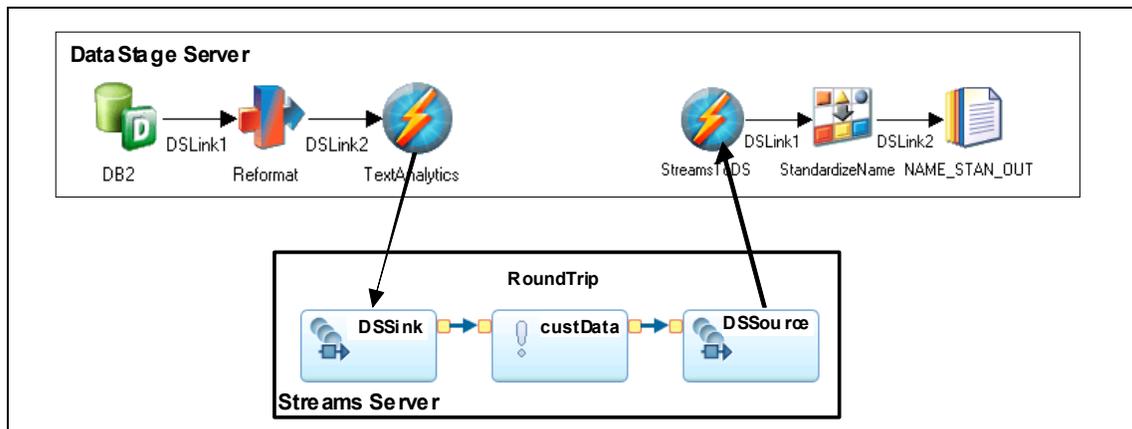


Figure 5-4 Round trip data, scenario 2

5.3 Metadata integration

The endpoint metadata of Streams applications can be imported to XMETA repository in DataStage as asset type of endpoint, which includes tuple and tuple attributes. Imported assets are shared to the repository and then can be used to configure Streams Connector for development.

On the reverse side, the DataStage job information can be retrieved in Streams to generate sample SPL application. This SPL application includes sample codes to set up DSSource or DSSink operators and also the schema definition created from the Streams Connector in the DataStage job.

To import Streams endpoint metadata to DataStage, a Streams application endpoint description file is generated from Streams application description language (ADL) files. The Streams application endpoint description file is then imported to XMETA repository in DataStage through InfoSphere Metadata Asset Manager. The Streams application endpoint description file can be created by the **generate-ds-endpoint-defs** command provided by the InfoSphere DataStage Integration Toolkit or the InfoSphere Streams Studio user interface (UI).

To import DataStage job metadata to Streams as a sample SPL, you can use the **generate-ds-spl-code** command provided by InfoSphere DataStage Integration Toolkit or the InfoSphere Streams Studio UI. Either way, you specify the DataStage project and the job that includes Streams Connector from which you want to create a sample SPL code.

5.3.1 Integration Setup Overview

Before providing details, we show you the overall steps to develop a Streams and DataStage integrated application to help you understand how the development work will go.

In addition, you must export the Streams Certificate and import it to DataStage to allow the Streams Connector to log in to the Streams name server. This can be done any time before developing the application and needs to be done only once. See 5.4, “Sample application” on page 117 for more detail about the procedure.

Steps to create a Streams-to-DataStage application

The steps are as follows:

1. Create a Streams SPL application that includes DSSink operator at the end of the stream.
2. Compile the SPL application and generate the endpoint description using the **generate-ds-endpoint-defs** command or from Streams Studio.
3. Import the endpoint description in InfoSphere Metadata Asset Manager.
4. Create the DataStage job that includes Streams Connector. Specify the Streams endpoint imported in the previous step in the Streams Connector properties.
5. Compile the DataStage job.

Steps to create a DataStage-to-Streams application

The steps are as follows:

1. Create the DataStage job that includes a Streams Connector at the end of that job. You define the columns in Streams Connector. The other stage properties for the Streams Connector can remain empty. Those properties can be completed after importing the endpoint description.
2. Create a sample SPL code in Streams by using the **generate-ds-spl-code** command or Streams Studio.
3. Create the Streams SPL application that includes DSSource operator as a start of the stream, using the sample code generated in the previous step for the schema definition and the operator setup.
4. Compile the SPL application and generate an endpoint description using the **generate-ds-endpoint-defs** command or from Streams Studio.
5. Import the endpoint description in InfoSphere Metadata Asset Manager.
6. Complete the Streams Connector properties using the imported Streams endpoint.
7. Compile the DataStage job.

5.4 Sample application

In this section, we create two basic applications. The first application reads data from a flat file in Streams and passes it to DataStage, then DataStage writes the data to a flat file. The second application does the reverse: DataStage job reads data from a flat file and passes it to Streams. Streams writes the data in a flat file. Both applications are simple for learning purposes and cover necessary steps to get you started.

5.4.1 Importing Streams certificate to DataStage

First, import the Streams certificate to DataStage to allow Streams Connector in DataStage to log in to the Streams name server. This step must be done only once.

1. Log in to the Streams server as Streams Administrator on a terminal. In this example, we use the user ID for the Streams Administrator "streamsadmin" and the "streams1" as instance name.
2. Run the **keytool** command (Example 5-1), which is in the jre directory of the InfoSphere Streams server installation, to export the certificate. The alias must be lwiks and cannot be configured. The keystore uses "ibmpassw0rd" as the default password.

Example 5-1 Run keytool

```
[streamsadmin@SEA-STREAMS1 ~]$ $STREAMS_INSTALL/jre/jre/bin/keytool
-keystore
~/streams/instances/streams1\@streamsadmin/sws/security/keystore/ib
mjsse2.jts -export -alias lwiks -file streams_certificate
Enter keystore password:
Certificate stored in file <streams_certificate>
[streamsadmin@SEA-STREAMS1 ~]$ ls -ltr -rw-rw-r-- 1 streamsadmin
streamsadmin 565 Jun 25 10:17 streams_certificate
```

3. Transfer the certificate to the DataStage server.
4. Log in to the DataStage server as an appropriate user. We use the root user in this example.
5. Run the **keytool** command to import the certificate. In this example, we create a keystore named `datastage_keystore`. See Example 5-2 on page 118.

Example 5-2 Create keystore

```
[root@SEA-IS-SMP ASBNode]#
opt/IBM/InformationServer/ASBNode/apps/jre/bin/keytool -import
-alias lwiks -file /SHR/Projects/REDBOOK/etc/streams_certificate
-keystore datastage_keystore.

Enter keystore password:
Re-enter new password:
Owner: CN=www.ibm.com, OU=STG, O=IBM, L=Austin, ST=TX, C=US
Issuer: CN=www.ibm.com, OU=STG, O=IBM, L=Austin, ST=TX, C=US
Serial number: 48e9afee
Valid from: 10/5/08 11:27 PM until: 5/27/33 11:27 PM
Certificate fingerprints:
    MD5: 30:1E:7E:67:B8:E4:CD:D9:97:26:DF:7D:47:2C:24:E9
    SHA1:
2C:A4:C8:AA:73:05:45:A8:94:B1:73:F4:FE:21:F6:17:C2:FE:01:05
Trust this certificate? [no]: yes
Certificate was added to keystore
```

5.4.2 Streams-to-DataStage application

We create a simple application that reads a flat file in Streams and passes the data to DataStage. The DataStage job writes the data to a flat file.

Creating Streams SPL application

Complete the following steps:

1. Before opening the Streams Studio, set up the STREAMS_SPLPATH environment variable to point to the InfoSphere DataStage Integration Toolkit. See Example 5-3. The InfoSphere DataStage Integration Toolkit is in the following directory:

```
$STREAMS_INSTALL/toolkits/com.ibm.streams.etl
```

Example 5-3 Set up STREAMS_SPLPATH

```
export
STREAMS_SPLPATH=/opt/ibm/InfoSphereStreams/toolkits/com.ibm.streams.etl
```

2. Open Streams Studio from the command line and create a new project. In this example, we invoke Streams Studio by the **streamsStudio** command under StreamsStudio directory in streamsadmin home directory (Example 5-4 on page 119).

Example 5-4 Invoke Streams Studio

```
[streamsadmin@SEA-STREAMS1 ~]$ pwd
/SHR/home/streamsadmin
[streamsadmin@SEA-STREAMS1 ~]$ StreamsStudio/streamsStudio
```

3. Add the InfoSphere DataStage Integration Toolkit to the project. We add the toolkit from the UI. Alternatively, the toolkit can also be added to the project by specifying it on the build command (sc) parameter. To add the toolkit, right-click the project and select **Edit Toolkit Information**, as shown in Figure 5-5.

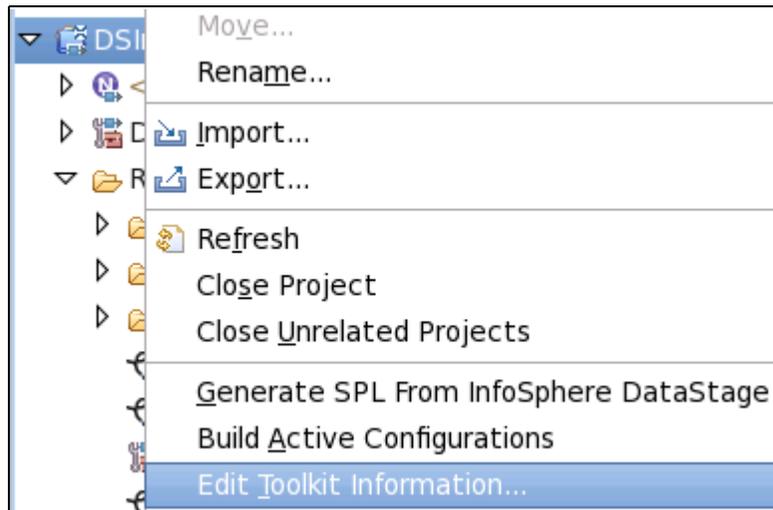


Figure 5-5 Add the Toolkit

4. Select the **Dependencies** tab and click **Add** on the Toolkit Dependencies window.
5. Click **Browse** to locate and specify the toolkit.
6. Select **com.ibm.streams.etl.datastage** and click **OK**. Click **OK** again to return to the Streams Studio main window.
7. Create a Streams SPL code. We create an application that reads data from a file and passes it to the DSSink operator. The code is shown in Example 5-5 on page 120.

Example 5-5 Streams SPL code

```
1 use com.ibm.streams.etl.datastage.adapters::*;
2
3 composite StreamsToDS {
4 graph
5     stream<int32 customerID, rstring name> sourceData =
6     FileSource() {
7         param
8             file      : "customer_list.txt";
9             format    : csv;
10            initDelay : 20.0;
11        }
12    () as DSOut = DSSink(sourceData) {
13        param connectionName : "DSOUT1";
14    }
```

Line 1 imports the definitions of DSSource and DSSink operators from the InfoSphere DataStage Integration Toolkit. Line 9 gives enough time to the DSSink operator and the DataStage job that is receiving the data from this code to get ready. The connection name in the line 12 is also used in the Streams Connector properties in DataStage job.

8. Compile the SPL code.
9. Find the ADL file created by Streams Studio. The ADL file is created in the following folder, by default (see Figure 5-6 on page 121):

`Resources/output/<application name>/<build mode>`

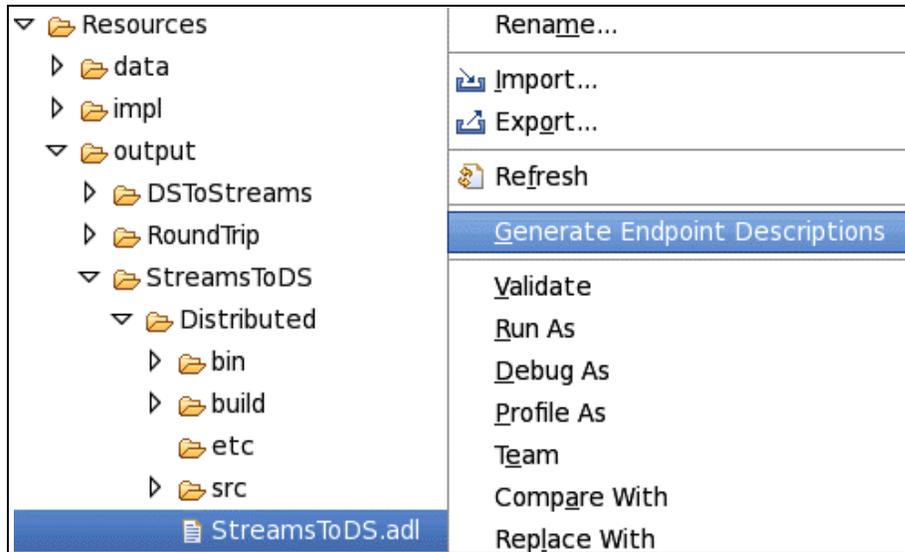


Figure 5-6 ADL file

10. Right-click the ADL file and select **Generate Endpoint Descriptions**.
11. Specify the name of the endpoint definition file. The SPL application ADL file is completed automatically, as shown in Figure 5-7.

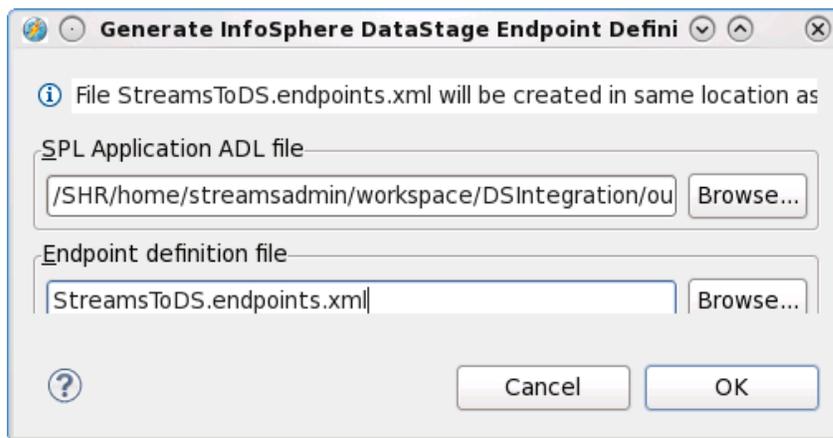


Figure 5-7 Endpoint definition file

12. Copy this file to the machine where you run InfoSphere Metadata Asset Manager.

Importing endpoint description to InfoSphere Metadata Asset Manager

Complete the following steps:

1. Open the browser to start InfoSphere Metadata Asset Manager and log in with an appropriate user ID. We use Information Server Administrator user ID (isadmin). The default URL is `http://<host_name>:9080/ibm/imam/console`.
2. Select the **Import** tab and click **New Import Area** (Figure 5-8).



Figure 5-8 New Import Area

3. Enter the import area name and select the metadata interchange server. You see the list of Bridges and Connectors in the tree view.
4. Select **IBM InfoSphere Streams** and click **Next**, as shown in Figure 5-9 on page 123.

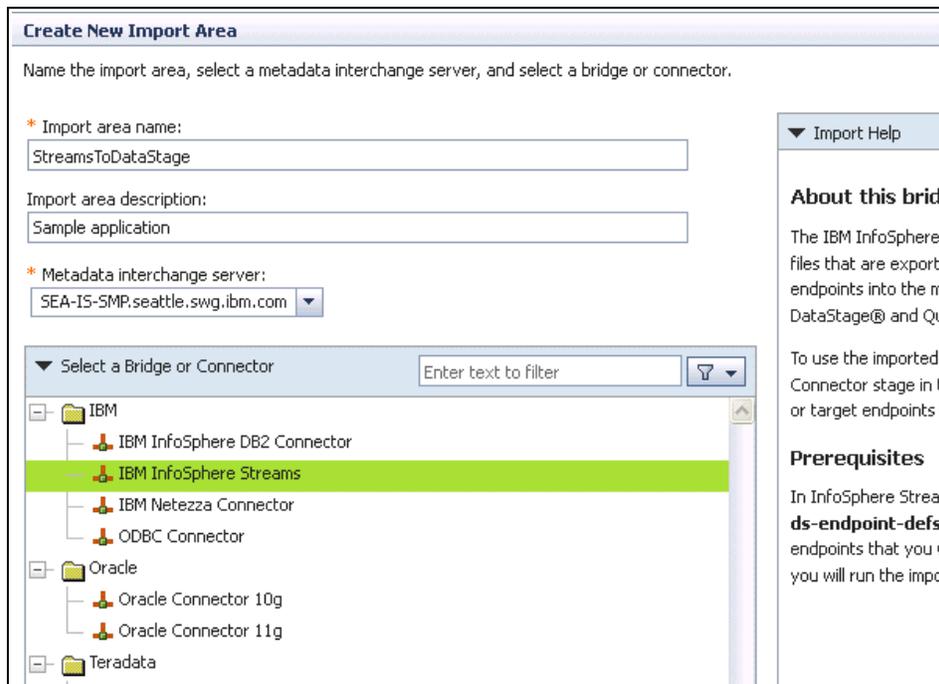


Figure 5-9 Bridges and Connections

5. Choose **Express Import** and click **Import**. If you want to manually import the endpoint, select **Managed Import**. See Figure 5-10 through Figure 5-13 on page 124.

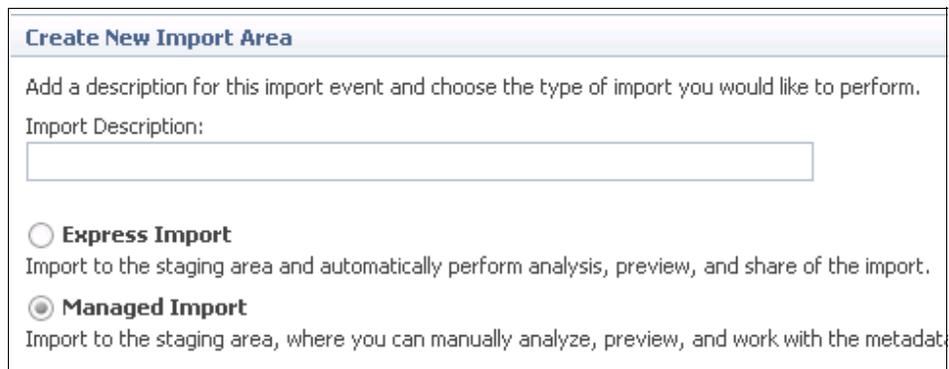


Figure 5-10 Select import area type

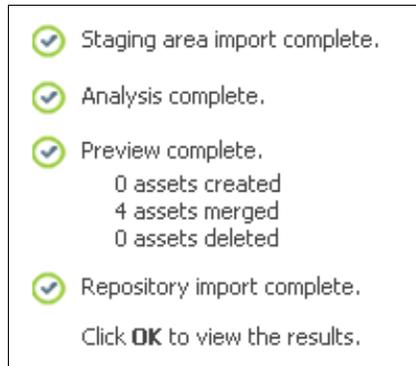


Figure 5-11 Status tracking

Summary				
Shared:	2013-08-25 at 10:53:21 by isadmin			
Staged import:	StreamsToDataStage 001			
Statistics				
Asset Types	Total	Created	Merged	Deleted
All	4	4	0	0
Endpoint	1	1	0	0
Tuple	1	1	0	0
Tuple attribute	2	2	0	0

Resulting Assets	
Endpoint	DSOUTPUT1
DSOUTPUT1	customerID
DSOUTPUT1	name

Figure 5-12 Results summary

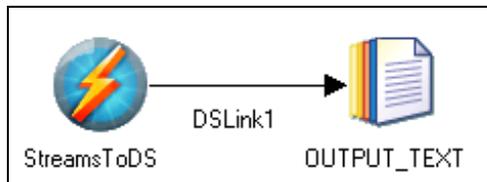


Figure 5-13 Final output text

Creating DataStage Job

Complete the following steps:

1. Create a new parallel job and place Streams Connector and Sequential File stages on the canvas.
2. Open the stage properties of Streams Connector and click **Configure**.
3. Select the endpoint you just imported in the previous step, as shown in Figure 5-14 on page 125.

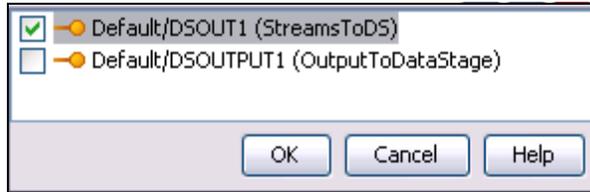


Figure 5-14 Endpoint selection

4. Click **OK**. The Connection name, Application scope, and Application name properties are completed automatically (Figure 5-15).

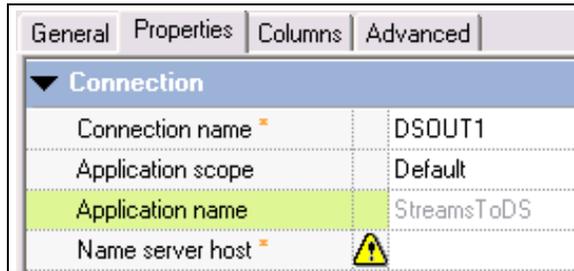


Figure 5-15 Endpoint properties

Columns are also populated based on the endpoint description. Click the **Columns** tab to see the columns, as in Figure 5-16.

General Properties Columns Advanced									
	Column name	Key	SQL type	Extended	Length	Scale	Nullable	Data element	Description
1	customerID	<input type="checkbox"/>	Integer				No		(int32)
2	name	<input type="checkbox"/>	VarChar				No		(rstring)

Figure 5-16 Columns

5. Specify the name server host and port number. To find the name server information, get on the Streams server as the Streams administrator and run the **streamtool geturl** command (Example 5-6).

In Example 5-6, sea-streams1.seattle.swg.ibm.com is the host name and 8443 is the port number.

Example 5-6 Run the geturl command

```
[streamsadmin@SEA-STREAMS1 ~]$ streamtool geturl -i streams1
https://sea-streams1.seattle.swg.ibm.com:8443/streams/console/login
```

6. Enter the user ID and password that is valid to run the Streams application. Here we use the streamsadmin user.
7. For the keystore file, specify the file created when you imported the Streams server certificate, as shown in Figure 5-17.

▼ Connection	
Connection name *	DSOUT1
Application scope	Default
Application name	StreamsToDS
Name server host *	sea-streams1.seattle.swg.ibm.com
Name server port *	8443
User name *	streamsadmin
Password *	*****
Keystore file *	/opt/IBM/InformationServer/ASBNode/datastage_keystore

Figure 5-17 Specify keystore file

8. Click **OK** to save and close the properties. You can keep the defaults of the other properties. The Test button in the Connection property is not enabled because our Streams application has not been started yet.
9. Set up the Sequential File stage.
10. Compile the job.

Running the application

Complete the following steps:

1. Return to Streams Studio and launch the application.
2. Start the DataStage job. When the DataStage job is started, it accesses the Streams name server to look up the host and port number that the Streams Connector must connect. After connection is established, the DSSink operator in Streams starts sending the data to the Streams Connector in DataStage, as shown in Figure 5-18 on page 127.

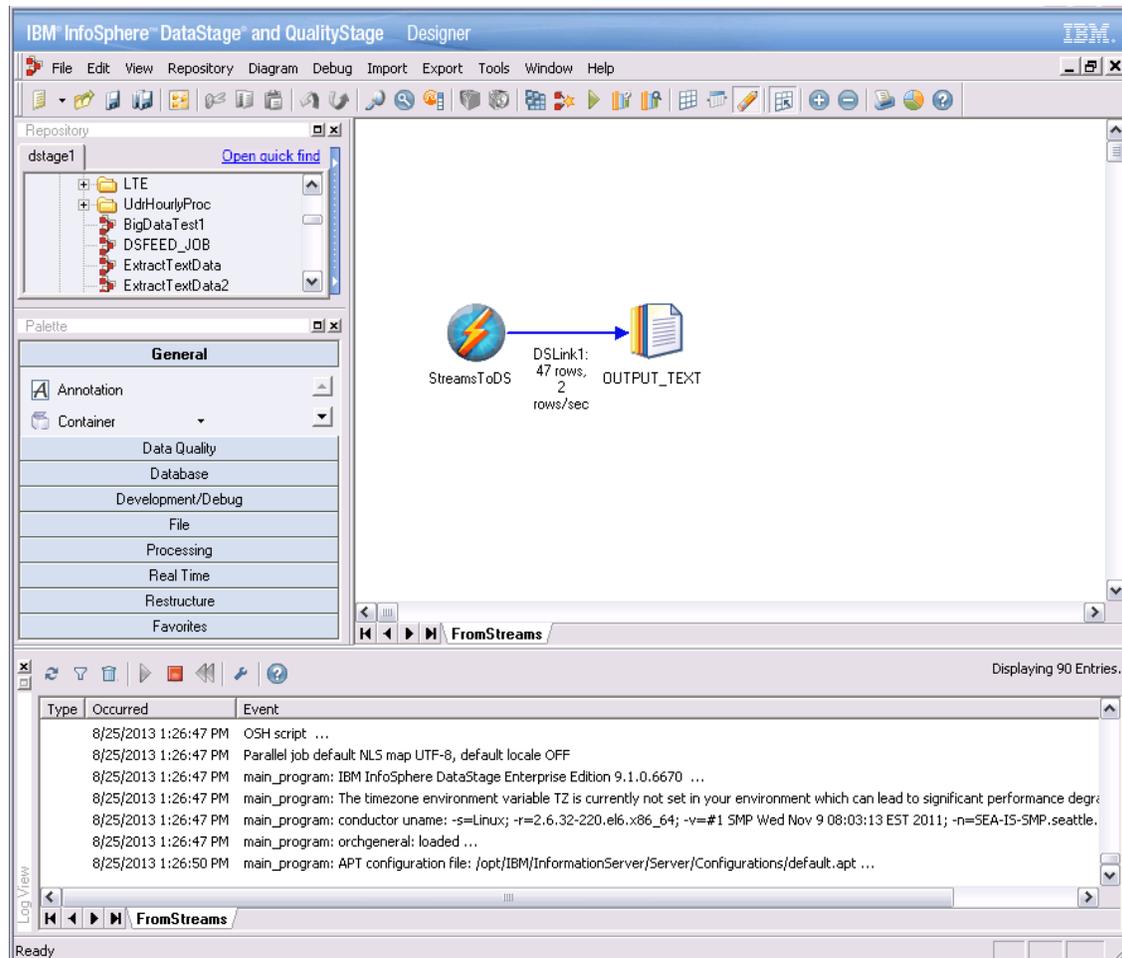


Figure 5-18 Streams connector to DataStage

DataStage-to-Streams Application

In this section, we create an application where the data flows in the opposite direction. In DataStage, the job reads data from a flat file in a Sequential File stage and passes it to the same Streams Connector stage we used in the previous example. How you configure Streams Connector is similar to when it receives data. In Streams, we use DSSource operator to receive data from DataStage and pass the data to FileSink operator to save it to a file.

We first create the DataStage job, but do not complete the configuration for Streams Connector. We only define the columns and keep the other properties blank because we want to import the metadata of this job to Streams to create

SPL code. Then, we return to DataStage and complete the configuration for Streams Connector.

Creating DataStage job

Complete the following steps:

1. Create a new parallel job and place Sequential File and Streams Connector stages, as shown in Figure 5-19.
2. Define the schema for the Sequential File stage and save the job. Keep the other properties in Streams Connector blank. We will return to this step later.

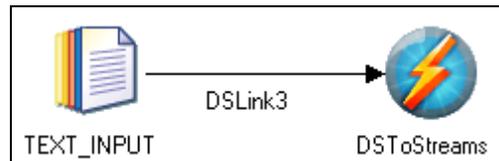


Figure 5-19 Creating the DataStage Job

Importing DataStage Metadata to Streams

Complete the following steps:

1. In Streams Studio, right-click the project name in the Project Explorer. Select **Generate SPL From InfoSphere DataStage**, as shown in Figure 5-20 on page 129
2. Enter the InfoSphere DataStage Services information. For server name, add the port number at the end of the server name string, separated by colon (:).
3. Enter the InfoSphere DataStage Connection information; you can click **Browse** to locate the names.

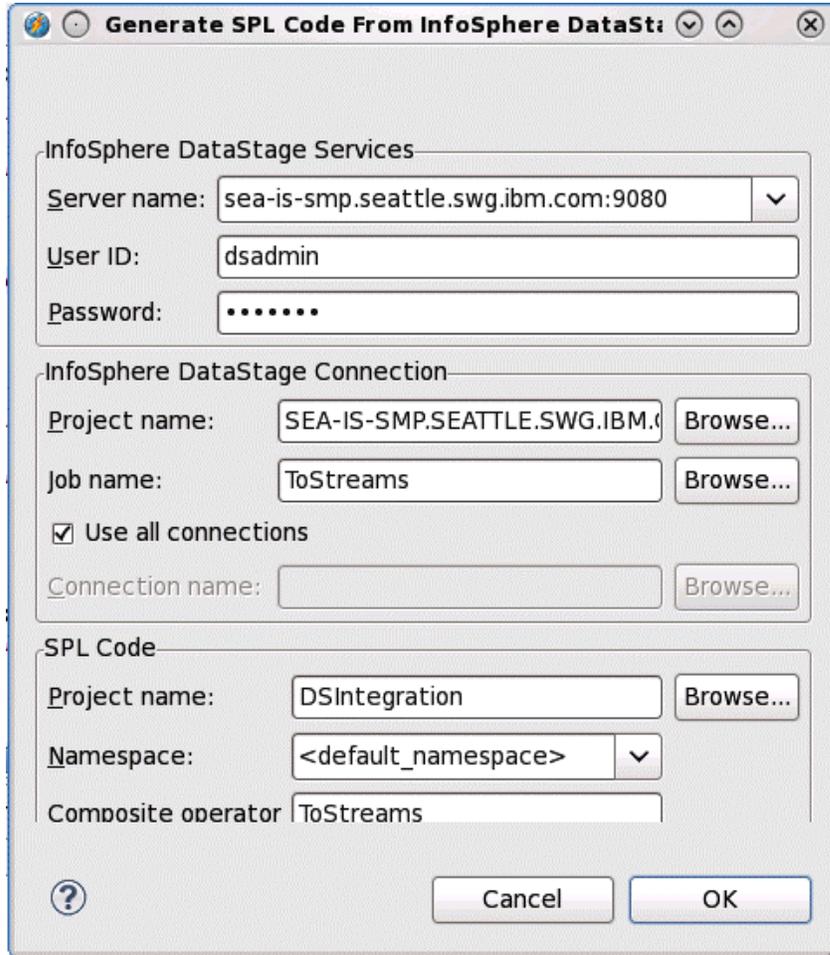


Figure 5-20 Generate SPL code

4. Keep the other information by default. Click **OK** to generate the sample SPL code. You might see a message similar to the one in Figure 5-21, but that is okay. You need only the schema definition and part of the sample code to copy, as shown in Figure 5-21.

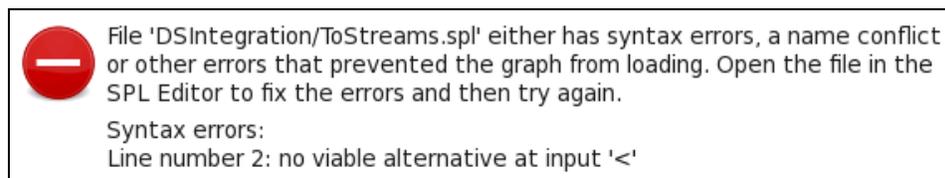


Figure 5-21 Fix errors with the SPL Editor

Example 5-7 is a snippet of the sample code. Only the code we are interested in is copied in the example. The code in lines 3 - 7 illustrates how you configure the DSSource operator. The connection name should be replaced by a valid string.

Example 5-7 Sample code snippet

```
1 namespace <default_namespace>;
2 use com.ibm.streams.etl.datastage.adapters::*;
3
4 type
5     DSToStreams_Schema =
6     int32 ID,
7     rstring text;
```

At the beginning of the code (line 4 to 7), you see the schema definition imported from the DataStage job. We will use this part later.

```
1 /* Start of Connection : DSToStreams */
2 /* The following DSSource operator is used to receive data from a datastage
job. */
3 stream<DSToStreams_Schema> DSToStreams_source_stage_outputStream =
DSSource() {
4     param
5         connectionName : "null";
6         outputType      : DSToStreams_Schema;
7     }
8 /* The following Custom can be used to show the data sent to the DSSource */
9 () as DSToStreams_source_stage_outputStream_Custom =
ToStreams_datastage_Custom(DSToStreams_source_stage_outputStream) {}
10 /* End of Connection : DSToStreams */
11 // Warning: applicationScope in DataStage was not set by connector name:
DSToStreams
12 // a default applicationScope was indicated by connector name: DSToStreams
so the config applicationScope : <scope> clause will be left off
13 }
```

Creating Streams SPL code

Complete the following steps:

1. Create the SPL code. Example 5-8 on page 131 shows the SPL code to receive the data from DataStage job. We copied the schema definition created in the sample code to lines 3 - 6 and the DSSource operator to lines 10 - 14. We specify "DSIN1" as the connectionName parameter.

Example 5-8 SPL code sample

```
1 use com.ibm.streams.etl.datastage.adapters::*;
2
3 type
4   DStreams_Schema =
5     int32 ID,
6     rstring text;
7
8 composite DStreams {
9   graph
10    stream<DStreams_Schema> dsData = DSource() {
11      param
12        connectionName : "DSIN1";
13        outputType      : DStreams_Schema;
14    }
15
16    () as StreamsSink = FileSink(dsData) {
17      param
18        file      : "streams_out.txt";
19        format    : csv;
20    }
```

2. Compile the code.
3. Create the endpoint description file from the ADL file created in the previous step.
4. Copy the endpoint description file to the machine where you run InfoSphere Metadata Asset Manager, if necessary.

Importing Endpoint description

Import the endpoint description file in the same way you did when you imported the other endpoint description file in the previous step. Figure 5-22 depicts an example of the import result.

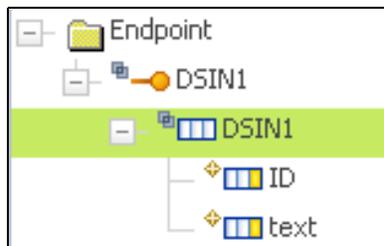


Figure 5-22 Sample code snippet

Completing DataStage job

Complete the following steps:

1. Return to the DataStage job and open the Streams Connector properties.
2. Click **Configure** and select the endpoint you just imported (Figure 5-23).



Figure 5-23 Endpoint selected

3. Click **OK** to apply it to the Streams Connector properties. You see the connection properties are completed, as shown in Figure 5-24.

General	Properties	Columns	Advanced	Partitioning
Connection				
Connection name		DSIN1		
Application scope		Default		
Application name		DSToStreams		

Figure 5-24 Connection properties

4. Specify the name server host, name server port, user name, password and keystore file. Use the same values for these as you entered in the other Streams Connector for Streams-to-DataStage application.
5. Save and compile the job.

Running the application

Complete the following steps:

1. Create some test data. This does not have to be many.
2. Run the Streams application first, as shown in Figure 5-25 on page 133. Whether the Streams application starts first actually does not matter. In our case, however, we do not have much test data. If we start the DataStage job first, it will end before we start the Streams application.

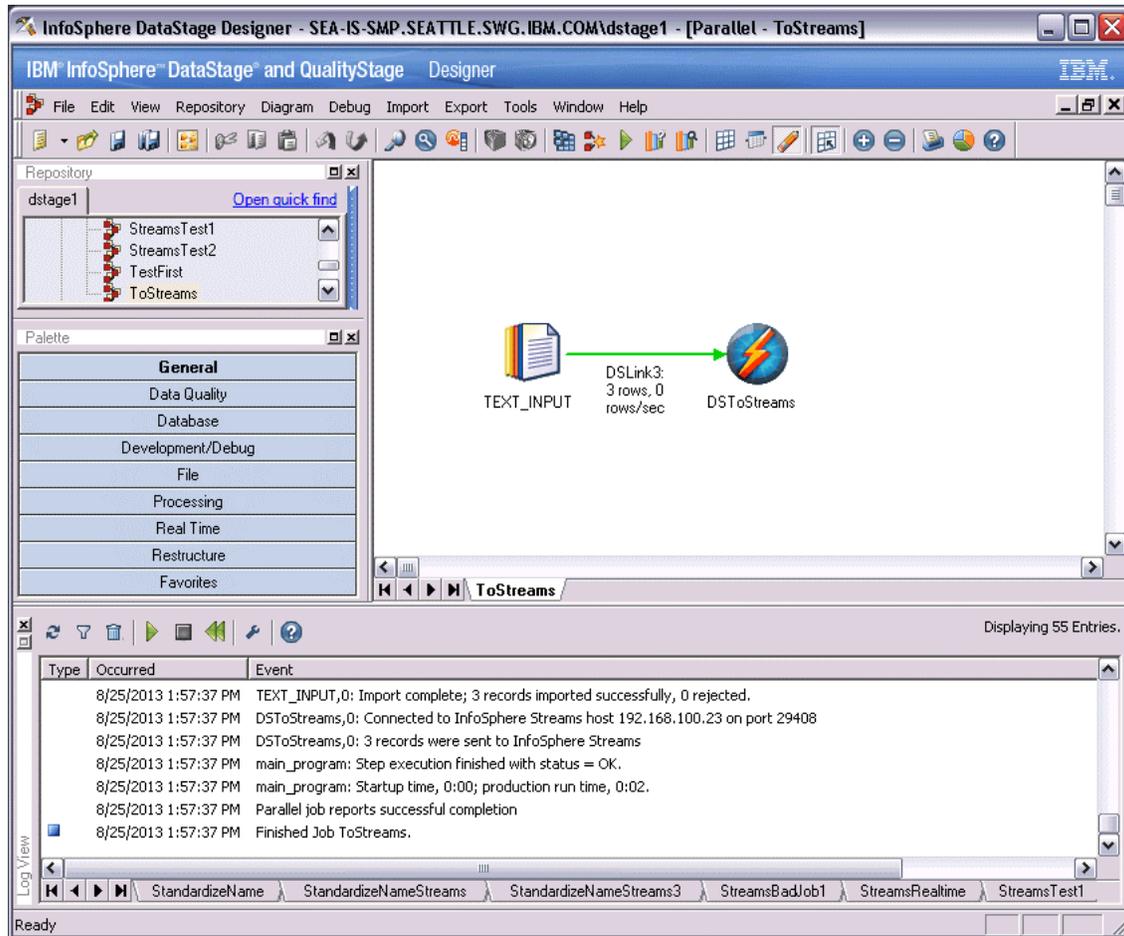


Figure 5-25 Run the Streams application

See the output file created in Streams to verify the result.

5.5 DataStage Job design practices

When using Streams Connector as a source, the DataStage job design can be somewhat similar to a real-time job, in which case the job starts with Streams connector, followed by some stages to perform data transformation, as depicted in Figure 5-26 on page 134. The DataStage job continues receiving data from Streams until Streams sends final *punctuation* or the job aborts. The DataStage job becomes an “always-on” job.

If the Streams application starts with an operator that sends final punctuation such as FileSource operator, the DataStage job is batch rather than real time. The DataStage job ends when Streams sends the final punctuation.

If your DataStage job can become “always-on,” consider the job data flow, especially when it includes source stages other than Streams Connector. See Figure 5-26. If the source in Streams was an operator that produces a set of data only once, such as FileSource, this DataStage job will work without any problems, as you expect. However, if the source continuously populates data into DataStage, you might have a problem.

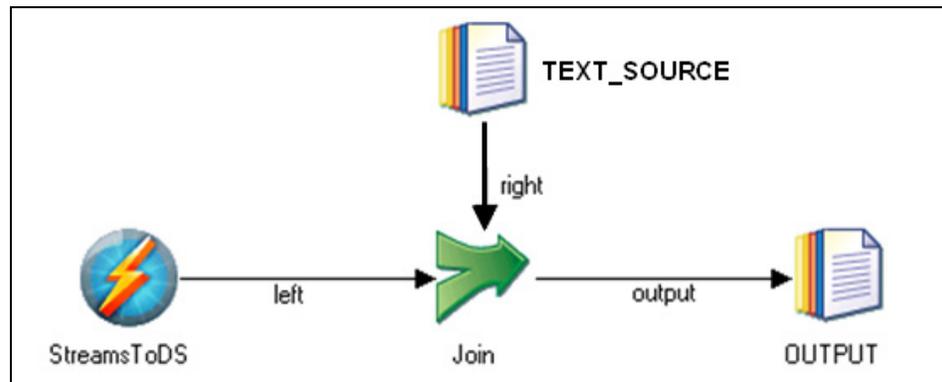


Figure 5-26 Streams Connector source

What can happen depends on other configurations in Streams and DataStage. However, one possible result is that the input link of the Join stage connected to Streams continues buffering the data and the Join stage does not produce any output. If you set up the Streams Connector to generate end-of-wave (which is the DataStage equivalent to the Streams punctuation) by converting window punctuation from Streams or by certain record counts, you will start seeing the output from the Join stage, but the records from the Sequential File stage will be exhausted by the first end-of-wave and will not be used again for the second wave, and so on. That is, there is no reference data that the records from Streams Connector stage can join, and therefore, no match from the second wave.

If, for example, you change the Join stage to the Lookup stage, the reference data deployed on the memory remains during the job run and the incoming records from Streams stage can always look up the reference. There are other scenarios that can result in a similar situation. What you should consider when designing a DataStage job with Streams connector is exactly same as designing a real-time DataStage job. The design practices for real-time jobs are described in *InfoSphere DataStage Parallel Framework Standard Practices*, SG24-7830. See the information about real-time data flow design and job topologies.

Streams Connector as a target

The Streams Connector runs sequentially, by default, regardless of whether it is a source or target. If the Streams Connector is the source, the fact that the Streams Connector runs sequentially does not cause issues in most cases, in terms of latency. However, if the Streams Connector is the target in a DataStage job that is intended as a batch job, be aware that this Streams Connector might become a bottleneck for the data flow. Even if your DataStage job runs in parallel, all the data from all partitions are collected together into one partition and passed to Streams. This might potentially be a bottleneck. In addition, Streams operators run in single thread unless configured otherwise. Even if you configure the Streams application to run in multi-threads, it might not perform as fast as DataStage because Streams is intended for a low-latency, not a high-throughput, application.

If you must use Streams as a target, do so if no alternatives are available. However, be aware of the difference in the nature of data processing between Streams and DataStage.

Punctuation and end-of-wave

The *punctuation* in Streams is a control signal between the tuples in a stream. A window punctuation creates a boundary in a stream and a final punctuation indicates the end of a stream. The *end-of-wave* in DataStage is similar to window punctuation in Streams. When DataStage receives an end-of-wave signal, it flushes the records in the buffer. It also limits the set of records to perform data processing in a stage, which applies to Sort and Aggregate stages. Going back to 5.5, “DataStage Job design practices” on page 133, that bad job example actually had a few issues. One issue is that the parallel framework might insert a sort operator in the input links to the Join stage. If the framework inserted a sort operator and the Streams Connector does not generate end-of-wave signal, the sort operator keeps accumulating records in a buffer without producing any output because it does not see the end of records. If the Streams application generates punctuation, you can map it to end-of-wave so that the sort operator can complete its operation when it receives an end-of-wave so that the sort operator starts producing the output. Another way is to generate an end-of-wave by specific record count if doing so makes sense to you.

However, punctuation can be generated by the event of failure. Although rare, if it happens, you do not know if the punctuation was intentionally created by the SPL program or was caused by the failure. Also, the use of end-of-wave can affect the DataStage job performance because it adds overhead. The best approach is to avoid such DataStage job design. If you cannot avoid that design, use it with caution.

DataStage real-time application

For real-time applications, a critical approach is to reduce the latency of the service. If the DataStage real-time job needs to use the Streams functionality, you might want to create a job that starts with ISDInput stage, followed by a Streams Connector, and finally an ISDOutput stage; or, a job starts with ISDInput and ends with Streams Connector. However, the Streams Connector cannot have an input link and output link at the same time. It must be only one of them. An Information Services Director (ISD) job must end with ISDOutput stage and ISDOutput stage only. How do you create the job? You must have two Streams Connector stages in your ISD job. The job should start with the ISDInput stage, and possibly another stage for data transformation, followed by a Streams Connector to send the data to Streams, as depicted in Figure 5-27. Within the same job, place another Streams Connector, with no link to the previous stages, followed by the ISDOutput stage.



Figure 5-27 ISD job with two Streams Connector stages

Although the example in Figure 5-27 appears as two separate data processing flows in a job, these Streams Connector stages are connected by the SPL application, as shown in Figure 5-28. This SPL application receives the data from the DataStage job by the DSSource operator, followed by some other operators, and then finally sends the data back to DataStage job by the DSSink operator.

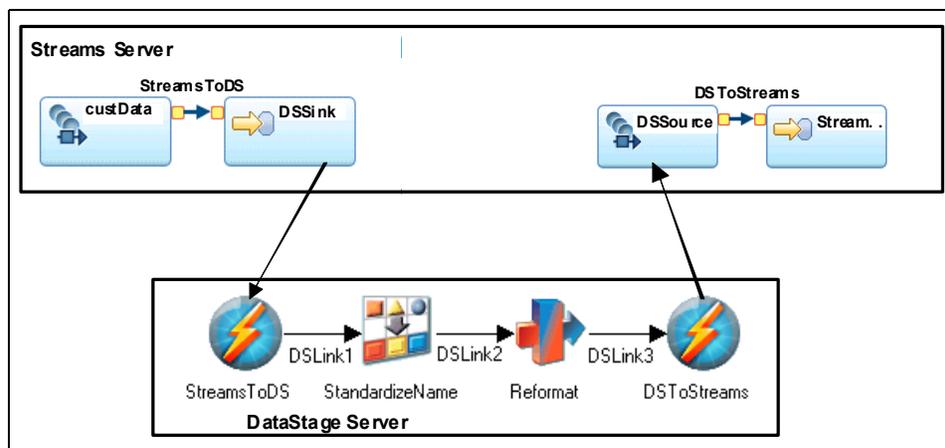


Figure 5-28 Connecting stages with an SPL application



Streams integration with IBM BigInsights

In this chapter, we focus on the IBM leading big data products, namely IBM InfoSphere Streams and IBM InfoSphere BigInsights, which are designed to address those current challenges. InfoSphere BigInsights delivers an enterprise-ready big data solution by combining Apache Hadoop, including the MapReduce framework and the Hadoop Distributed File Systems (HDFS), with unique technologies and capabilities from IBM. Both products are built to run on large-scale distributed systems, designed to scale from small to very large data volumes, handling both structured and unstructured data analysis.

6.1 Streams and big data challenges

With the growing use of digital technologies, the volume of data generated by our society is exploding into the exabytes. With the pervasive deployment of sensors to monitor everything from environmental processes to human interactions, the variety of digital data is rapidly encompassing structured, semi-structured, and unstructured data. Finally, with better “pipes” to carry the data, from wireless to fiber optic networks, the velocity of data is also exploding (from a few kilobits per second to many gigabits per second). We call data with any or all of these characteristics *big data*. Examples include sources such as the Internet, web logs, chat, sensor networks, social media, telecommunications call detail records, biological sensor signals (such as ECG and EEG), astronomy, images, audio, medical records, military surveillance, and eCommerce.

Many readers are familiar with the acronym ROI, defined as return on investment. Here, we propose a change to that traditional definition, and extend ROI to be return on information, a new paradigm that focuses on driving better business outcomes through several objectives, such as these:

- ▶ Improving efficiencies and reducing operational costs: doing more with less
- ▶ Improving the confidence and trustworthiness of the data: recognizing and reducing risk and exposure
- ▶ Improved decision-making, with access to timely insight across all sources of data

Time is often a common and limiting dimension with these objectives. How can you make the best decision, while reducing the time to make a decision? In a fast-paced business world, you cannot always afford to wait too long for answers. There are several aspects to this requirement from a resource, risk, and value perspective:

- ▶ Without undermining the decision process, ideally you should aim to minimize the resources (such as people and computation) consumed while making a decision, which should also subsequently minimize the cost.
- ▶ Consider the risk of making a decision based on untruthful or incomplete data. Ideally you should verify the truthfulness of the data (minimize risk) in a time frame that gets you to a point of comfort. However you should recognize the need to balance the time and effort required to achieve this, versus the risk level for the decision to be made. That is, you can spend more time, if required, to be more confident.
- ▶ Recognize the value of the data and how it changes over time. Much of the data around us is transient in nature, with data value having a half-life characteristic, such as decaying over time. What is useful one second, might be useless the next.

The quicker you can respond, the more opportunity there is to influence the outcome of an event that might still be in motion (such as a fraudulent transaction). This can enable you to move from an “after-the-fact” model (time frame $t+1$) to an “in-the-moment” model (time frame t).

You can also leverage historic data and learning from prior decisions that have been made, moving from a reactionary stance (time frame of t and $t+1$), to one of predicting outcomes and guiding the response ahead of time (time frame $t-1$).

InfoSphere Streams has exceptional capability to handle decisions in the moment: real-time data-in-motion analytics. However, there are occasions when a deeper analysis is required (as examples, correlation, modeling, and entity integration) over complex data sets, over extended periods of time. The InfoSphere Streams windowing capability is powerful but there are practical limits for how long you can buffer data in memory.

This situation requires the seamless functioning of data-in-motion (current data) and data-at-rest (historical data) analysis, operating on massive volumes, varieties, and velocities of data. How to bring the seamless processing of current and historical data into operation is a technology challenge faced by many businesses that have access to big data.

In the next section (6.1.1, “Application scenarios” on page 139), we describe various scenarios where data analysis can be performed across the two platforms to address the big data challenges.

6.1.1 Application scenarios

The integration of data-in-motion (InfoSphere Streams) and data-at-rest (InfoSphere BigInsights) platforms addresses several application scenarios:

- ▶ Scalable data ingest: Continuous ingest of data through Streams into BigInsights, exploiting a highly parallel approach with the following characteristics:
 - Volume: Handling the volume of data arriving and being ingested, such as data reduction or condensing and signals from noise
 - Velocity: Reacting to events in the moment, with the ability to perform far deeper analysis through BigInsights when finding data of interest
 - Variety: Data types beyond traditional structured sources, to semi-structured and unstructured data, such as text, images, video and audio
 - Veracity: improving the accuracy and truthfulness of the data through fact discovery, corroboration, correlation, and spatial and temporal reasoning.

- ▶ **Bootstrap and enrichment:** Historical context generated from BigInsights to bootstrap analytics and enrich incoming data on Streams, then focusing in on what is important and using prior insight to guide and direct future analysis.
- ▶ **Adaptive analytics model:** Models generated by analytics, such as data mining, machine-learning, and statistical-modeling on BigInsights used as a basis for analytics on incoming data in Streams and updated based on real-time observations. You should also recognize that your view of what constitutes normal changes drifts over time and therefore it is necessary to trigger a refresh of the underlying model.
- ▶ **Complex social- and entity-related analysis:** Identifying entities and entity attributes, and allowing for integration and resolution of the entities and attributes. These interactions are shown in Figure 6-1 and explained in greater detail in the subsequent sections.

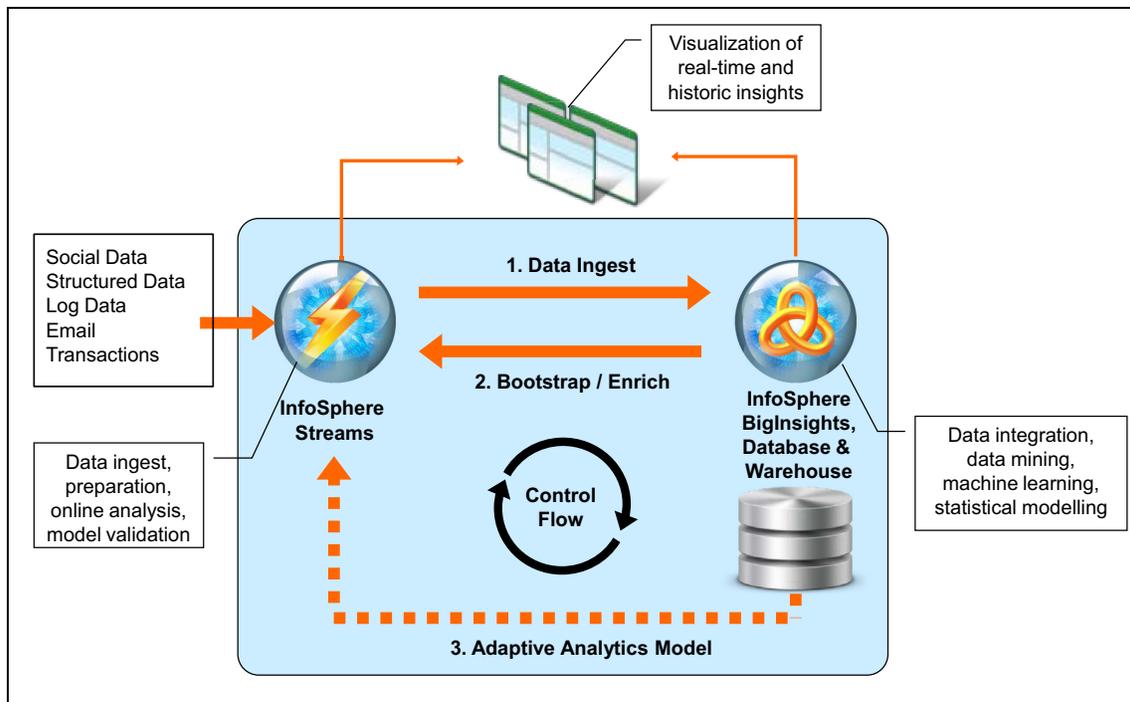


Figure 6-1 Big Data application scenarios

6.1.2 Large scale data ingest

Data from various systems arrives continuously, that is, as a continuous stream, a periodic batch of files, or by other means. Data must first be processed to extract all the required data for consumption by downstream analytics. Data-preparation steps include operations such as data-cleansing, filtering, feature extraction, deduplication, and normalization. These functions are performed on InfoSphere Streams. Data is then stored in BigInsights for deep analysis, and also forwarded to downstream analytics on Streams. The parallel pipeline architecture of Streams is used to batch and buffer data and, in parallel, load it into BigInsights for best performance.

An example of this function is the call detail record (CDR) processing use case. CDRs come in from the telecommunications network switches periodically as batches of files. Each of these files contains records that pertain to operations, such as call initiation, and call termination for telephones. The most efficient way is to remove the duplicate records in this data as it is being ingested, because duplicate records can be a significant fraction of the data that will needlessly consume resources if post-processed. Additionally, telephone numbers in the CDRs must be normalized and data must be appropriately prepared to be ingested into the back end for analysis. These functions are most efficiently performed using Streams.

6.1.3 Bootstrap and enrichment

BigInsights can be used to analyze data over a large window of time, and that it has assimilated and integrated from various continuous and static data sources. Results from this analysis provide contexts for various online analytics and serve to bootstrap them to a well-known state. They are also used to enrich incoming data with additional attributes required for downstream analytics.

As an example from the CDR processing use case, an incoming CDR might list only the phone number to which that record pertains. However, a downstream analytic might want access to all phone numbers a person has ever used. At this point, attributes from historical data are used to enrich the incoming data to provide all the phone numbers. Similarly, deep analysis results in information about the likelihood that this person will cancel their service. Having this information enables an analytic to offer a promotion online to keep the customer from leaving the network.

6.1.4 Adaptive analytics model

Integration of the Streams and BigInsights platforms enables interaction between data-in-motion and data-at-rest analysis. The analysis can use the same analytic

capabilities in both Streams and BigInsights. It not only includes data flow between the two platforms, but also control flows to enable models to adapt to represent the real-world accurately, as it changes. The two interactions are as follows:

- ▶ BigInsights to Streams control flow
- ▶ Streams to BigInsights control flow

BigInsights to Streams control flow

Deep analysis is performed using BigInsights to detect patterns on data collected over a long period of time. Statistical analysis algorithms or machine-learning algorithms are compute-intensive and run on the entire historical data set, in many cases making multiple passes over the data set, to generate models to represent the observations. For example, the deep analysis might build a relationship graph showing key influencers for products of interest and their relationships. After the model is built, it is used by a corresponding component on Streams to apply the model on the incoming data in a lightweight operation. For example, a relationship graph built offline is updated by analysis on Streams to identify new relationships and influencers based on the model, and take appropriate action in real time. In this case, there is control flow from BigInsights to Streams when an updated model is built, and an operator on Streams can be configured to pick up the updated model mid-stream and start applying it to new incoming data.

Streams to BigInsights control flow

After the model is created in BigInsights and incorporated into the Streams analysis, operators on Streams continue to observe incoming data to update and validate the model. If the new observations deviate significantly from the expected behavior, the online analytic on Streams may determine that it is time to trigger a new model-building process on BigInsights. This situation represents the scenario where the real world has deviated sufficiently from the model's prediction that a new model needs to be built. For example, a key influencer identified in the model might no longer be influencing others, or an entirely new influencer or relationship can be identified. Where entirely new information of interest is identified, the deep analysis can be targeted to just update the model in relation to that new information, for example, to look for all historical context for this new influencer, where the raw data had been stored in BigInsights but not monitored on Streams until now. In this situation, the application does not have to know everything that it is looking for in advance. It can find new information of interest in the incoming data and get the full context from the historical data in BigInsights and adapt its online analysis model with that full context. Here, an additional control flow from Streams to BigInsights is required in the form of a trigger.

6.1.5 Complex social and entity related analysis

The IBM Accelerator for Social Data Analytics (SDA) is a set of end-to-end applications that extracts insight from a variety of social media sources (tweets, boards, and blogs) and then builds rich social profiles of users based on several specific use cases.

Performing this combination of both in-motion and deeper at-rest analysis against social media data can be challenging because of the complexity of unstructured textual data, and because of the significant volume and velocity of social data being generated.

Typical use cases for the accelerator include the following examples:

- ▶ Brand management: Assess the nature and amount of feedback around a company or product and make more informed decisions about how to position, market, and sell their products.
- ▶ Lead generation: Identify and generate leads in the finance and retail industries.

Also possible is to customize the generic IBM Accelerator for Social Data Analytics applications to meet the needs of your specific use cases and industries.

Entity extraction and integration

Social media can be used as a rich source of information that can be segmented by several measures, such as the following examples:

- ▶ Buzz: The noise or volume of discussion on a topic
- ▶ Sentiment: An expression of a person's opinion toward a product or service
- ▶ Intent: The action a person might take in relation to a product or service, such as buy, subscribe, attend, and cancel

Other examples might include interest and ownership.

IBM SDA adds more value to social media by the following measures:

- ▶ Identifying entities, such as people and companies, from the social data
- ▶ Constructing 360 degree views (attribute-rich profiles) of these entities that can be leveraged for timely decision-making

These profiles might include user dimensions, such as gender, location, parental status, and marital status. By combining the social measures, such as sentiment and intent, with comprehensive entity profiles, it is possible to provide detailed insight that is segmented into micro-segments, and use this to enhance use cases such as brand management and lead generation.

SDA: How it works

To achieve this level of analysis, the SDA uses both InfoSphere Streams and InfoSphere BigInsights to provide in-motion and at-rest analysis of social data.

Two distinct pipelines, or flows, of data are processed to perform analysis of the social data in real time with Streams (providing immediate buzz and sentiment insight), and in a parallel through BigInsights for more complex entity analysis and profile building. Both flows support each another: with the online ingest processes feeding the offline analysis, and the offline analysis feeding profile data back to the online flow for micro-segmentation of the social insight.

The two pipelines are shown in Figure 6-2.

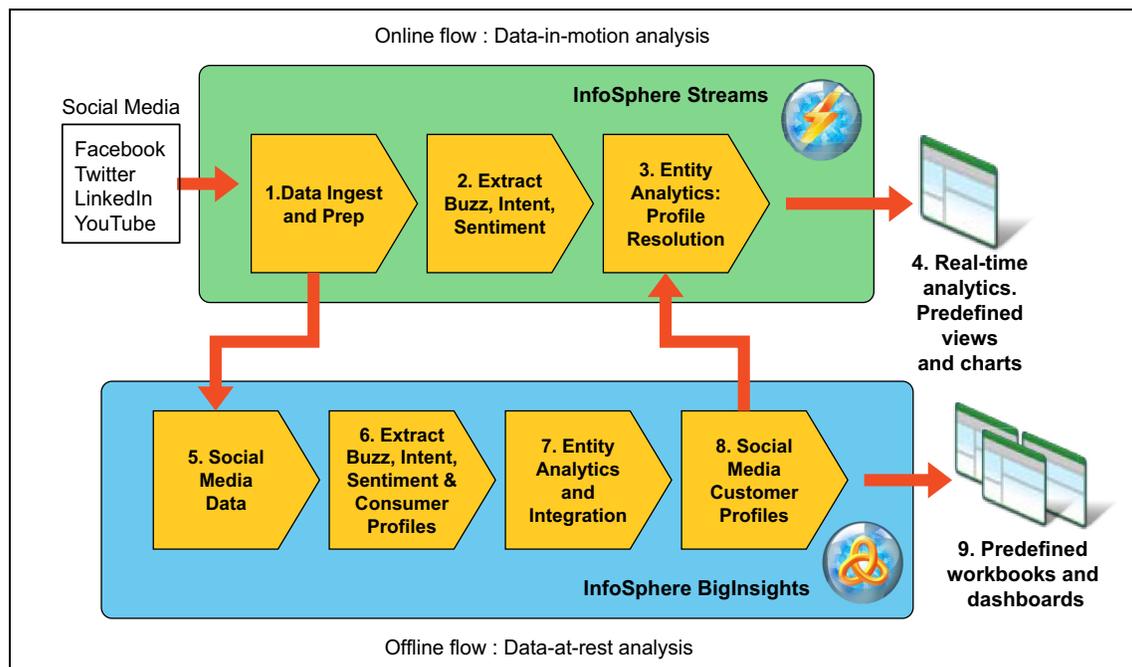


Figure 6-2 IBM Social Data Accelerator

Online flow (data-in-motion analysis) operates as follows:

1. Raw social media is initially consumed from sources such as the BoardReader search engine from BoardReader.com and APIs from Gnip, Inc. before being formatted to a common and consistent format. This common format data is also shared at this stage with the second offline data pipeline (list item 5 on page 145).
2. The initial text analysis stage is where the first level of timely insight is extracted based on the content of each document analyzed in isolation. The

output is the feedback (buzz, intent, sentiment, customer of, ownership), analyzed for each document (tweet, board, or blog).

3. Profile data received from the offline flow (list item 8) is used to perform Entity Profile Matching, allowing micro-segmentation reporting of the social data by profile attribute.
4. This is dashboard-based visualization of the social insight, updating in real-time, based on continuous feeds of social data.

Offline Flow (data-at-rest) operates as follows:

5. Commonly formatted social data is received from the online flow.
6. The text analysis stage is completed using global analysis across all documents, all social media sources, and over time.
7. Comprehensive entity profiling is used to identify entities (such as people and companies) within the social data and to begin integrating and resolving (disambiguating) occurrences of the same entity.
8. These entities are then used to seed the creation of social media customer profiles, which contain attributes of interest about each customer or entity. The richness of the entity profile improves over time, with significant volumes of social data (both across source, post or document and time) being used to incrementally contribute toward the profile attribute data. Profiles are shared across both the online and offline pipelines.
9. A series of offline reports are produced showing the following information:
 - User profiles: Comprehensive user profiles are built based on analysis over time and across social media sources, and are saved as a comma-separated values (CSV) file, a BigSheets Workbook, and charts in the BigInsights Dashboard.
 - User profiles and feedback: Comprehensive feedback with profiles is built based on analysis over time and across social media sources and are saved as a CSV file, a BigSheets Workbook, and charts in the BigInsights Dashboard.
 - Feedback: The data includes cumulative feedback (buzz, intent, sentiment, customer ownership) and is saved as a CSV file, a BigSheets Workbook, and charts in the BigInsights Dashboard.

6.1.6 Application development

This section describes how an application developer can create an application spanning two platforms to give timely analytics about data in motion while maintaining full historical data for deep analysis. We describe a simple scenario to the SDA example in “SDA: How it works” on page 144, showing how an application can use interactions between Streams and BigInsights. This simple application tracks the positive and negative sentiment being expressed about products of interest in a stream of emails and tweets (it does not provide the in-depth entity profiling of the full SDA solution). An overview of the application is shown in Figure 6-3.

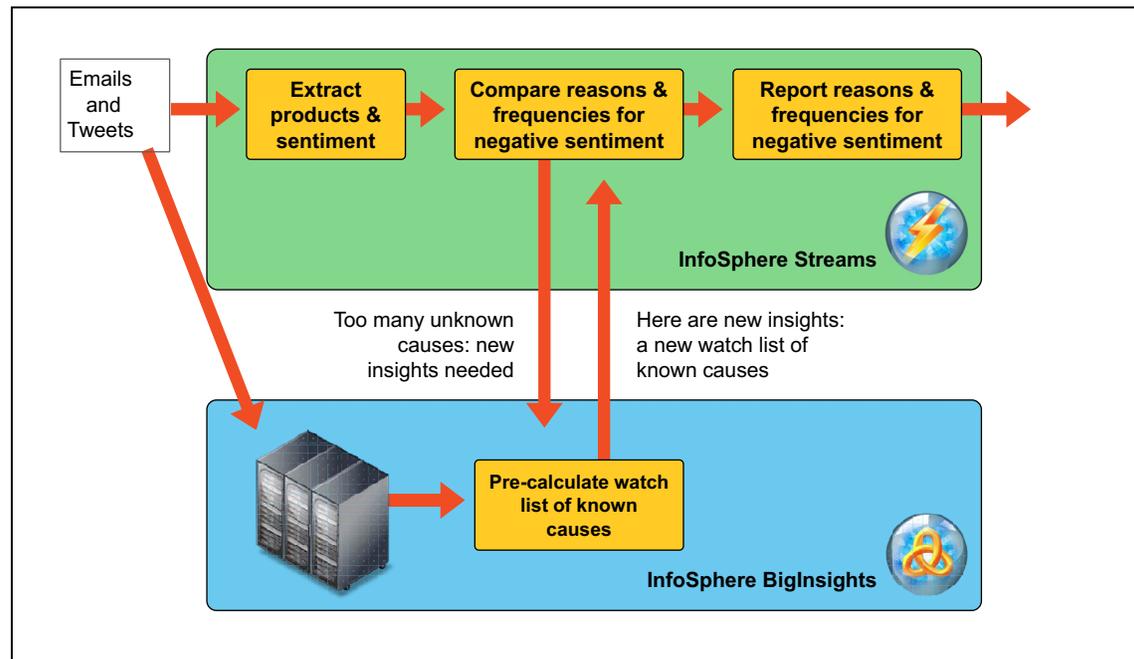


Figure 6-3 Streams and BigInsights application example

Each email and tweet on the input streams is analyzed to determine the products mentioned and the sentiment expressed. The input streams are also ingested into BigInsights for historical storage and deep analysis. Concurrently, the tweets and emails for products of interest are analyzed on Streams to compute the percentage of messages with positive and negative sentiment being expressed. Messages with negative sentiment are further analyzed to determine the cause of the dissatisfaction based on a watch list of known causes.

The initial watch list of known causes can be bootstrapped using the results from the analysis of stored messages on BigInsights. As the stream of new negative

sentiment is analyzed, Streams checks if the percentage of negative sentiment that has an unknown cause (not in the watch list of known causes) has become significant. If it finds that a significant percentage of the causes are unknown, it requests an update from BigInsights. When requested, BigInsights queries all of its data using the same sentiment analytics used in Streams and recalculates the list of known causes. This new watch list of causes is used by Streams to update the list of causes to be monitored in real time. The application stores all of the information it gathers but only monitors the information currently of interest in real time, thereby using resources efficiently.

6.1.7 Application interactions

Although the following example is simple, it demonstrates the main interactions between Streams and BigInsights and reflects the same principles as used by the full blown Social Data Accelerator:

- ▶ Data ingest into BigInsights from Streams
- ▶ Streams triggering deep analysis in BigInsights
- ▶ Updating the Streams analytical model from BigInsights

The implementations of these interactions for this simple demonstration application are discussed in more detail in the following sections.

Data ingest into BigInsights from Streams

Streams processes data using a flow graph of interconnected operators. The data ingest is achieved using a Streams BigInsights Sink operator to write to BigInsights (see 6.1.8, “Enabling components” on page 148). The complexities of the BigInsights distributed file system used to store data are hidden from the Streams developer by the Streams BigInsights Sink operator. The Sink operator batches the data stream into configurable sized chunks for efficient storage in BigInsights. It also uses buffering techniques to decouple the write operations from the processing of incoming streams, allowing the application to absorb peak rates and ensure that write operations do not block the processing of incoming streams. As with any operator in Streams, the Sink operator writing to BigInsights can be part of a more complex flow graph allowing the load to be split over many concurrent Sink operators that could be distributed over many servers.

Streams triggering deep analysis in BigInsights

Our simple example triggered deeper analysis in BigInsights using the same Streams BigInsights Sink operator as mentioned in “Data ingest into BigInsights from Streams” on page 147. BigInsights does deep analysis using the same sentiment extraction analytic as used in Streams and creates a results file to update the Streams model. For more advanced scenarios, the trigger from

Streams might also contain query parameters to tailor the deep analysis in BigInsights.

Updating the Streams analytical model from BigInsights

Streams updates its analytical model from the result of deep analysis in BigInsights. The results of the analysis in BigInsights are processed by Streams as a stream that can be part of a larger flow graph. For our simple example, the results contain a new watch list of causes for which Streams will analyze the negative sentiment.

6.1.8 Enabling components

The integration of data-in-motion and data-at-rest models is enabled by the following three main types of components:

- ▶ Common analytics
- ▶ Common data formats
- ▶ Data exchange adapters

The components used for this simple demonstration application are available on the Streams Exchange at the IBM developerWorks website:

<https://www.ibm.com/developerworks/mydeveloperworks/groups/service/html/communityview?communityUuid=d4e7dc8d-0efb-44ff-9a82-897202a3021e>

Common analytics

The same analytics capabilities can be used on both Streams and BigInsights. In this simple example, (described in 6.1.7, “Application interactions” on page 147) IBM BigInsights System T text analytic capabilities are used to extract information from the unstructured text received in the emails and tweets. Streams uses a System T operator to extract the product, sentiment, and reasons from the unstructured text in the feeds. The feeds are stored in BigInsights in their raw form and processed using the System T capabilities when the deep analysis is triggered. System T uses AQL, a declarative rule language with a SQL-like syntax to define the information extraction rules. Both Streams and BigInsights use the same AQL query for processing the unstructured text.

Common data formats

Streams formatting operators can transform data between the Streams tuple format and data formats used by BigInsights. In this simple example, we use JavaScript Object Notation (JSON) as the data format for storage in BigInsights. The TupleToJSON operator is used to convert the tuples in Streams to a JSON string for storage in BigInsights. The JSONToTuple operator is used to convert the JSON string read from BigInsights to a tuple for Streams to process.

Common data exchange adapters

Streams source and sink adapters can be used to exchange data with BigInsights. In this simple example, HDFSSource, HDFSSink, and HDFSDirectoryScan adapter operators are used to exchange data with BigInsights. These adapters have similar usage patterns to the fileSource, fileSink, and directoryScan adapter operators provided in the Streams Processing Language (SPL) Standard Toolkit. The HDFSSink operator is used to write data from streams to BigInsights. The HDFSDirectoryScan operator looks for new data to read from BigInsights using the HDFSSource operator.

6.2 BigInsights summary

IBM big data platforms, InfoSphere Streams, and InfoSphere BigInsights, enable businesses to operationalize the integration of data-in-motion and data-at-rest analytics at large scales to gain current and historical insights into their data, allowing faster decision making without restricting the context for those decisions. In this chapter, we describe various scenarios in which the two platforms interact to address the big data analysis problems.



Complex event processing

In this chapter, we explore the IBM InfoSphere Streams Complex Event Processing Toolkit. For brevity, we refer to it as the CEP Toolkit in the remainder of this chapter.

The following quotation is from the Complex Event Processing Toolkit overview in the IBM information center:

“Complex event processing (CEP) uses patterns to detect composite events in streams of tuples. For example, CEP can be used to detect stock price patterns, routing patterns in transportation applications, or user behavior patterns in web commerce settings. You can perform complex event processing in IBM InfoSphere Streams by using the Complex Event Processing Toolkit.”

See the information center:

<http://pic.dhe.ibm.com/infocenter/streams/v3r1/topic/com.ibm.swg.im.infosphere.streams.cep-toolkit.doc/doc/cep-overview.html>

7.1 The role of the CEP Toolkit

With the CEP Toolkit, you can build applications in InfoSphere Streams that implement patterns to detect specific sequences of tuples in a stream. The resulting output of the event detection is itself a tuple that can then be published externally (in a message, database, or dashboard as examples) or can be consumed inside of Streams for additional analytics.

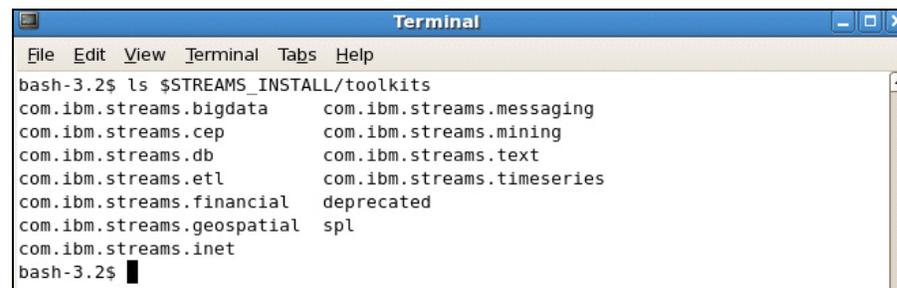
What types of sequences can be found? That is entirely up to the person defining the pattern. Consider the following examples:

- ▶ Up and down changes in numeric values or measurements
- ▶ Direction or location changes in a two- or three-dimensional space
- ▶ On-off changes of switches or sensors within a limited period of time
- ▶ Any other change of status or state of any value or measurement

The role of CEP Toolkit is to provide a high level grammar to express the patterns of interest, and then provide the underlying algorithms that perform the pattern detection in the correct and most efficient manner possible.

7.1.1 Adding the CEP Toolkit to your build path

As with many other toolkits, the CEP Toolkit is installed with the InfoSphere Streams product and is in `$STREAMS_INSTALL/toolkits` location as `com.ibm.streams.cep` file. See Figure 7-1.



```
Terminal
File Edit View Terminal Tabs Help
bash-3.2$ ls $STREAMS_INSTALL/toolkits
com.ibm.streams.bigdata      com.ibm.streams.messaging
com.ibm.streams.cep         com.ibm.streams.mining
com.ibm.streams.db          com.ibm.streams.text
com.ibm.streams.etl         com.ibm.streams.timeseries
com.ibm.streams.financial   deprecated
com.ibm.streams.geospatial spl
com.ibm.streams.inet
bash-3.2$
```

Figure 7-1 CEP Toolkit in Streams

Assume that you are using the Eclipse-based Streams Studio tooling for your application development.

To get started with the CEP Toolkit, include it in your toolkit lookup path:

1. Navigate to the Streams Explorer tab, right-click **Toolkit Locations**, and select **Add Toolkit Location** from the menu, as shown in Figure 7-2.

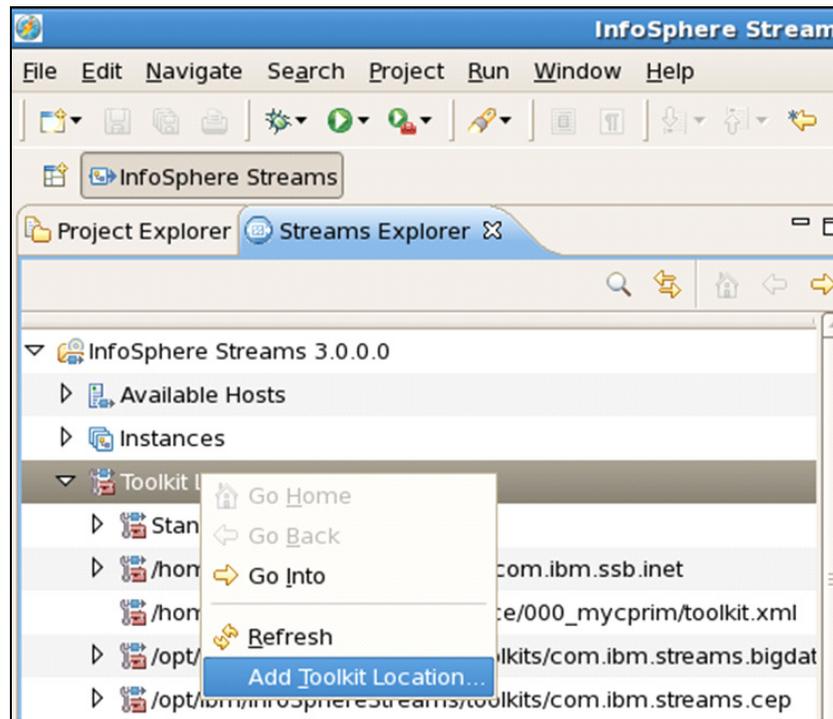


Figure 7-2 Streams Explorer

2. In the dialog that opens (Figure 7-3), click **Directory**.

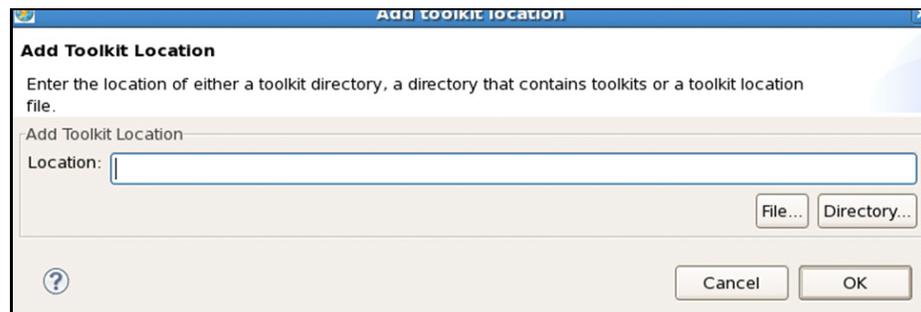


Figure 7-3 Streams Explorer navigation

3. In the Select the Toolkit location dialog box (Figure 7-4), navigate to your \$STREAMS_INSTALL/toolkits folder.

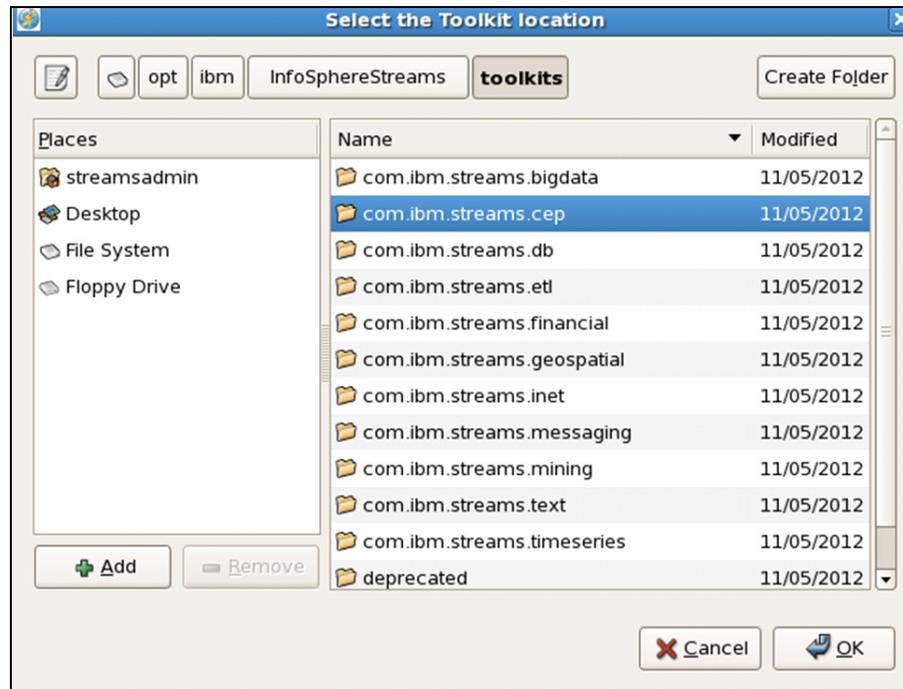


Figure 7-4 Toolkit location

4. Select **com.ibm.streams.cep** and click **OK**.
5. In the Add toolkit location dialog box, click **OK**.

For more information, see “Adding toolkit locations” topic in the IBM InfoSphere Streams information center:

<http://pic.dhe.ibm.com/infocenter/streams/v3r0/topic/com.ibm.swg.im.infosphere.streams.studio.doc/tasks/tusing-working-with-toolkits-adding-to-olkit-locations.html>

7.2 Stock price watch example

In this section, we explore the CEP Toolkit through a stock price watch application. We define the business objective and then iterate through several versions of the SPL applications.

The purpose of the application is to detect an M-shape (double-top) pattern in company stock prices, as shown in the example in Figure 7-5. In that figure the green arrow (G, on the left side) points to the beginning of an M-shape pattern in the input data and the red arrow (R, on the right side) points to the end of the pattern.

The M-shape pattern is a well-known pattern in the analysis of stock trading history. Early detection of this pattern can be important information when making trading decisions, and is well-suited to detection by the CEP Toolkit.

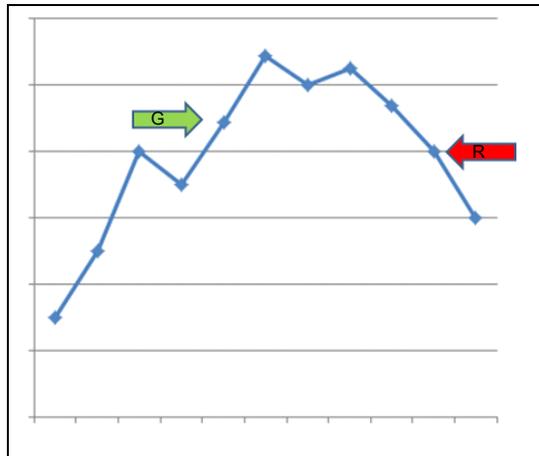
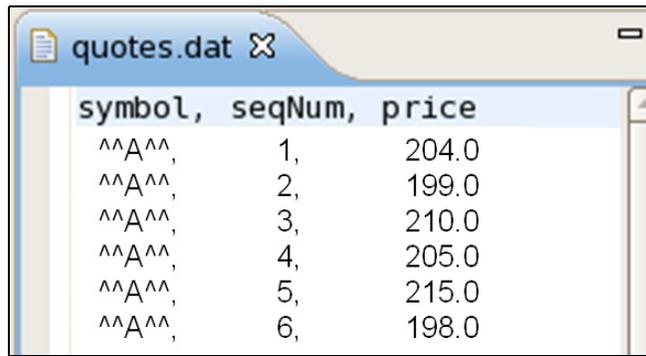


Figure 7-5 Stock price watch example

7.2.1 First iteration of stock price watch application

Figure 7-6 shows sample data for a stock that exhibits the behavior that we need to detect.



symbol	seqNum	price
^^A^^	1	204.0
^^A^^	2	199.0
^^A^^	3	210.0
^^A^^	4	205.0
^^A^^	5	215.0
^^A^^	6	198.0

Figure 7-6 Stock Price Watch iteration number 1

You can see in this data that starting from a value of 199.0 in the second tuple, the price rises to 210.0, drops to 205.0, rises again to 215.0, then deeply drops to 198.0, which is below the starting value of 199.0. That series of values completes the M-shape pattern that we want to detect. We write code and test a Streams application that will use the MatchRegex operator of the CEP Toolkit.

To design, implement, and test the pattern logic, we create a simple application consisting of the operators FileSource, MatchRegex, and FileSink, as shown in Figure 7-7.

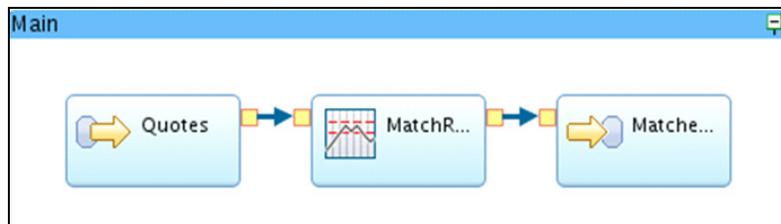


Figure 7-7 Sample operators

The invocation of the MatchRegex in the application is shown in Figure 7-8.

```
stream<rstring symbol, int32 seqNum, decimal32 maxPrice, int32 count>
  MatchResults as Matches = MatchRegex([Quotes])
{
  param
    pattern : ". rise drop rise deep" ;
    partitionBy : symbol ;
    predicates : {
      rise = price > First(price) && price >= Last(price),
      drop = price >= First(price) && price < Last(price),
      deep = price < First(price) && price < Last(price)
    } ;

  output
    Matches : symbol = symbol,
            seqNum = First(seqNum),
            maxPrice = Max(price),
            count = Count() ;
}
```

Figure 7-8 Definition of the MatchRegex

Now we explore the details of this implementation.

We start by defining a pattern to be detected by the MatchRegex operator, as shown in Figure 7-9.

```
param
  pattern : ". rise drop rise deep" ;
```

Figure 7-9 Match Pattern

With this definition, we are inventing the terms *rise*, *drop*, and *deep* for this application. We are specifying that, from any tuple in the input stream (identified with the dot (.) character), we are looking for the pattern of a rise in the next tuple, followed by a drop in the next tuple, followed by a rise in the next tuple, and then followed by a deep in the tuple after that.

Next, we implement the definitions of these terms as a set of rules:

- rise** Current tuple price is higher than starting tuple price and previous tuple price.
- drop** Current tuple price is higher than starting tuple price but lower than previous tuple price.
- deep** Current tuple price is lower than starting tuple price and previous tuple price.

We define these rules as Boolean expressions in the predicates parameter of the operator, as shown in Figure 7-10.

```
predicates : {  
    rise = price > First(price) && price >= Last(price),  
    drop = price >= First(price) && price < Last(price),  
    deep = price < First(price) && price < Last(price)  
};
```

Figure 7-10 Boolean expressions

Finally, we do not want to accidentally combine two tuples from two different symbols into the same trend analysis, so we separate the symbols using partitioning, as shown in Figure 7-11.

```
partitionBy : symbol ;
```

Figure 7-11 Partitioning

Running this application using the sample data confirms that it produces the expected output:

```
"^A^",2,215.0,5
```

That output is defined as follows:

- ▶ `^A^`: Symbol of the stock
- ▶ `2`: Sequence number of the tuple that started the pattern
- ▶ `215.0`: Maximum price value of all tuples in the pattern
- ▶ `5`: Number of tuples that made up the pattern

Success! Now we can get more sample data for further testing.

7.2.2 Second iteration of stock price watch application

We have been given additions to the sample data, as shown in Table 7-1.

Table 7-1 Sample data for the second iteration

Symbol	seqNum	Price
^D^	1	199.0
^D^	2	206.0
^D^	3	210.0
^D^	4	207.0
^D^	5	205.0
^D^	6	215.0
^D^	7	198.0

Now when we run the sample application we are not detecting the M-shape for this new input data, as shown in Figure 7-12. What went wrong?

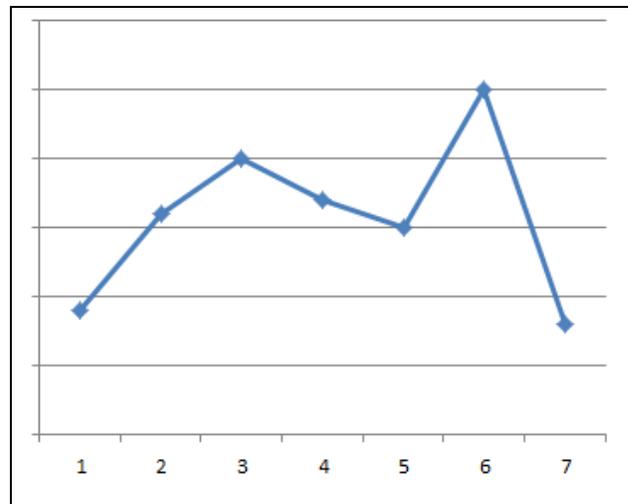


Figure 7-12 Plot of expanded data file

On closer examination of the data, we see that the rise and drop trends do not occur with just one single tuple. In the first rise sequence, the rise is to 206.0 and then another rise to 210.0. Similarly, we see that it is not a single tuple for the drop. Instead this is also two steps. That is, a drop to 207.0 and then a further drop to 205.

With further study we learn that yes, the rise and drop sequence can be just one tuple, or can happen over more than one tuple and still fit the definition of an M-shape pattern. So, we must modify our application to improve the pattern detection.

Luckily, the CEP operator provides syntax to specify one-or-many trends using the plus sign (+) character as part of the pattern definition. So we modify the pattern to the following line:

```
pattern : ". rise+ drop+ rise+ deep";
```

After recompiling the application, we test again with the updated sample data file and receive the expected output:

```
"^A^",1,215.0,5  
"^D^",1,215.0,7
```

This shows that we still correctly detected the M-shape pattern on fictional stock `^A^` over a sequence of five records, and also detected the M-shape on fictional stock `^D^` over a sequence of seven records.

7.2.3 Third iteration of stock price watch application

For final testing we received additions to the sample data file, shown in Table 7-2.

Table 7-2 Additional data for third iteration

Symbol	seqNum	price
<code>^X^</code>	1	199.0
<code>^X^</code>	2	206.0
<code>^X^</code>	3	210.0
<code>^X^</code>	4	207.0
<code>^X^</code>	5	205.0
<code>^X^</code>	6	215.0
<code>^X^</code>	7	204.0
<code>^X^</code>	8	198.0

Again, we are not recognizing an M-shape pattern in this data with the application, as seen in Figure 7-13. What went wrong?

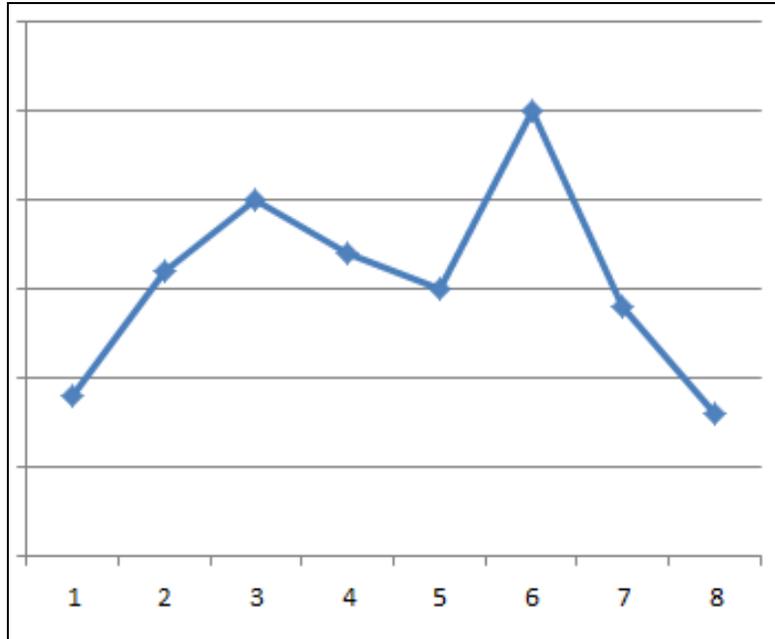


Figure 7-13 Plot of additional data

On closer examination of the data, we see that the final value 198.0 did not occur immediately after a rise, but instead was preceded by a drop. With further study, we learn that yes, there can be zero or more drops before a deep instance and still meet the criteria for the M-shape pattern.

The MatchRegex operator's pattern syntax includes the asterisk (*) character for this zero-or-more occurrences trend, so we modify the pattern to the following line:

```
pattern : ". rise+ drop+ rise+ drop* deep";
```

We can read this pattern as follows:

1. Any tuple in the stream
2. Followed by one or more rises
3. Followed by one or more drops
4. Followed by one or more rises
5. Followed by zero or more drops
6. Followed by one deep drops below the starting tuple's value

Sequences of tuples in the input stream that meet these criteria should be detected as valid M-shape patterns.

After recompiling the application, we test again with the updated sample data file and receive the following expected output:

```
"^A^",2,215.0,5  
"^D^",1,215.0,7  
"^X^",1,215.0,8
```

This output shows that we correctly detected the M-shape pattern on the following items:

- ▶ Stock `^A^` over a sequence of five records
- ▶ Stock `^D^` over a sequence of seven records
- ▶ Stock `^X^` over a sequence of eight records

With this third set of test data, we have completed the testing for the usage of the MatchRegex operator, and can now use that SPL code in our actual application.



WebSphere MQ, XMSSource, XMSSink

This chapter details how to use InfoSphere Streams with IBM WebSphere® MQ.

Interoperability between InfoSphere Streams and WebSphere MQ is provided through the InfoSphere Streams Messaging Toolkit. This toolkit is installed automatically when you install Streams. You do have to tell Streams where this or any toolkits are located, and you do have to add a toolkit to any specific Streams project, but the Messaging Toolkit is always present. The Messaging Toolkit provides an *XMSSource* operator to read from WebSphere MQ, and an *XMSSink* operator to write to WebSphere MQ.

Reading from or writing to WebSphere MQ is nearly identical to reading to or writing from any outside software server using Streams, whether it is a SQL database, HTTP server, or similar. The Messaging Toolkit provides interoperability only to WebSphere MQ. To achieve connectivity to these other software server types (SQL databases, HTTP, or other), use other Streams toolkits designed for that purpose.

Although this chapter expects that you can create, compile, and run a simple Streams application, this chapter requires that you have WebSphere MQ skills prerequisites. This chapter shows how to create every WebSphere MQ resident object that is required to run the two simple Streams applications that follow, even if the need for each WebSphere MQ object is covered only at a high level.

8.1 WebSphere MQ Server, Message Service Client installation

The example described here was created and tested on a single-tier Red Hat Enterprise Linux version 6.3, 64-bit system, with the Linux KDE desktop. We ran Streams version 3.0, WebSphere MQ Server version 7.5, and an additional (included with WebSphere MQ) software package named IBM Message Service Client for C/C++, version 2.0.2.

Generally, we show the command-line options to create the WebSphere MQ required objects to save time and space. Because we had a single-tier system, we installed WebSphere MQ Server. In a multitier environment, you likely would install the WebSphere MQ Client distribution, co-located on any Streams node that requires any WebSphere MQ interoperability. The Streams Client software is meant to forward requests to a WebSphere MQ Server installation, if properly configured. Although this chapter does not detail this more advanced client/server installation of WebSphere MQ, we expect that with the information that is in the next list, an experienced WebSphere MQ system administrator can accomplish the more advanced installation and configuration.

Note: We ran version 7.5 of WebSphere MQ because it was readily available to us. We know through trial that version 7.1.0.3 and version 7.0 of WebSphere MQ Server also worked, and worked with earlier versions of the Message Service Client. But, be sure you run versions that are certified and supported by IBM. Check the following resources:

- ▶ IBM InfoSphere Streams Information Center:
<http://pic.dhe.ibm.com/infocenter/streams/v3r0/index.jsp>
- ▶ IBM technical support
http://www.ibm.com/software/lotus/passportadvantage/remote_technical_support.html

Related to the installation of WebSphere MQ Server version 7.5, and the Messaging Service Client version 2.0.2, consider the following information:

- ▶ The steps were done as the user named, root. Unless otherwise stated, in each case we chose the default values as prompted.
- ▶ The single software distribution file for WebSphere MQ Server must be extracted (with `gunzip` or `untar`).
- ▶ Run the `mqlicense.sh` license program.
- ▶ We installed the (*n*) RPM files with a single command: `rpm -ivh *.rpm`

- ▶ A new user, `mqm`, is created during the installation process. We set the password for user `mqm`.
- ▶ The distribution file for the Messaging Service Client must be extracted. To avoid conflict, we did this second part of the installation from a new, empty directory.
- ▶ We ran `setup.bin`.
- ▶ We did like that the installation process defaulted to an installation directory of root's home directory, so we chose `/opt` when prompted for the "Install GSK SSL" parent directory. Every other piece of software referred to in this chapter was in `/opt` location by default, so we preferred this option.

At this point we are done as `root`. The next set of steps are done as the user named, `mqm`. The next section (8.2, "Making the WebSphere MQ resident objects" on page 165) offers a series of Bash(C) shell scripts, that make assorted WebSphere MQ resident objects, set WebSphere MQ permissions, start WebSphere MQ resident listening daemons, and other related tasks.

8.2 Making the WebSphere MQ resident objects

In this section, we list the creation of every WebSphere MQ resident object that is required to support the two Streams applications that follow. (The first Streams application writes to WebSphere MQ, the second reads from WebSphere MQ, forming a round trip, and a complete unit of work that you can use for testing.) Although we list every WebSphere MQ command that is required, we do not necessarily define what an WebSphere MQ channel object is, what an WebSphere MQ queue object is, or why you need one.

The steps that follow are contained inside several Bash(C) shell scripts, although one larger script would have worked just fine. Example 8-1 creates an WebSphere MQ queue manager, WebSphere MQ queue, and WebSphere MQ channel. See the code review after Example 8-1.

Example 8-1 Create the queue manager, queue, and channel

```
export PATH=$PATH:/opt/mqm/bin:/opt/mqm/samp/bin

echo "Make the queue manager"
crtmqm RHHOST_QM

echo "Start the queue manager"
strmqm RHHOST_QM

echo "Start the listener. You might use a different port number here."
```

```

runmq1sr -t tcp -p 1425 -m RHHOST_QM &
echo "Pid is "$!

#####

echo "Create the queue, and a channel."

runmqsc RHHOST_QM <<!EOF

def q1 ( RHHOST_Q )
dis q1 ( RHHOST_Q )

def channel ( RHHOST_CHANNEL ) chltype ( SVRCONN ) TRPTYPE ( TCP )
dis channel ( RHHOST_CHANNEL )

end

!EOF

```

Consider the following information about Example 8-1 on page 165:

- ▶ Using the **crtmqm** command-line program, we make a queue manager named, RHHOST_QM.

Note: Generally, in this section, for everything you name and create, use RHHOST_, and then follow with a new set of characters for the object type.

- ▶ Using the **strmqm** command-line program, the queue manager is started.
- ▶ The **runmq1sr** command calls to start a new connection listening daemon for this queue manager. You might need to use a different value for the port number, which is 1425. This value will be referred to later.

Note: If you must reboot your operating system, the **strmqm** and **runmq1sr** commands must be run again, to start and restart the queue manager listener for this queue manager.

None of the other WebSphere MQ related commands must be rerun; all objects are persisted in perpetuity.

- ▶ The **runmqsc** command enters an interactive WebSphere MQ command environment where we create a queue and a channel; **def** defines; **dis** displays.

Example 8-2 executes commands to alter several WebSphere MQ system related permissions. See the code review after Example 8-2.

Example 8-2 Setting WebSphere MQ related permissions

```
export PATH=$PATH:/opt/mqm/bin:/opt/mqm/samp/bin

#####

echo "Setting permissions"

setmqaut -m RHHOST_QM -t qmgr -p mqm +all
setmqaut -m RHHOST_QM -t qmgr -p streamsadmin +all

setmqaut -m RHHOST_QM -t q -n RHHOST_Q -p mqm +all
setmqaut -m RHHOST_QM -t q -n RHHOST_Q -p streamsadmin +all
```

Consider the following information about Example 8-2:

- ▶ Essentially we give all permissions to both the mqm and streamsadmin user. In a production environment, we would grant a lesser set, a more secure set of permissions.
- ▶ Permissions are granted to the queue manager, and then the lower queue object levels.

Example 8-3 creates an WebSphere MQ topic, and copies a sample WebSphere MQ configuration file that we must then manually edit. See the code review after Example 8-3.

Example 8-3 Creating an MQ topic, preparing to edit JMSAdmin.config

```
export PATH=$PATH:/opt/mqm/bin:/opt/mqm/samp/bin
export MQ_ROOT=/opt/mqm
export PATH=$PATH:$MQ_ROOT/java/bin:$MQ_ROOT/samp/jms/samples

# 'Source' this next script, do not run it directly.
. $MQ_ROOT/java/bin/setjmsenv64

export CLASSPATH=$CLASSPATH:/opt/mqm/java/lib64/jms.jar

#####

runmqsc RHHOST_QM <<!EOF

def topic ( RHHOST_TOPIC ) topicstr ( 'TOPICTEST' )
```

```
end

!EOF

#####

cp /opt/mqm/java/bin/JMSAdmin.config .
chmod 777 JMSAdmin.config

# Edit file above, change provider Url ..

# PROVIDER_URL=file:///POT/MyFiles/21_MQ/
```

Consider the following information about Example 8-3 on page 167.

- ▶ After setting a large number of environment variables, we again run **runmqsc**, this time to create a WebSphere MQ topic.

Note: We need these environment variables only as user mqm, and to create the WebSphere MQ resident objects.

Streams and the streamsadmin user do not need these environment variables. The streamsadmin environment variables requirements are covered in 8.3, “Setting Streams environment variables, and adding toolkits” on page 171.

- ▶ We copy a sample WebSphere MQ configuration file used in the area of JMS administered objects. (JMS administered objects are how Streams and WebSphere MQ talk to one another.) In our example here, we must manually change one line of this source file (the line that sets the value for PROVIDER_URL). We used the /POT/MyFiles/21_MQ/ directory. Notice the precise formatting of this value, because it is required.

Note: The directory that is pointed to by the value of PROVIDER_URL is significant; it will be used later in this example. If you change the value of PROVIDER_URL, you must rerun a portion of the steps that follow.

Minus a last set of changes to permissions, Example 8-4 forms the last set of steps, the last set of objects to create within WebSphere MQ. See the code review after Example 8-4.

Example 8-4 Nearly final set of MQ related steps

```
export PATH=$PATH:/opt/mqm/bin:/opt/mqm/samp/bin
export MQ_ROOT=/opt/mqm
export PATH=$PATH:$MQ_ROOT/java/bin:$MQ_ROOT/samp/jms/samples:$MQ_ROOT/
java/jre/bin
export PATH=$PATH:/opt/mqm/java/jre64/jre/bin
export CLASSPATH=$CLASSPATH:/opt/mqm/java/lib64/jms.jar
export MQ_JAVA_INSTALL_PATH=/opt/mqm/java
export MQ_JAVA_DATA_PATH=/var/mqm
export MQ_JAVA_LIB_PATH=/opt/mqm/java/lib64

L=$MQ_JAVA_INSTALL_PATH/lib
CLASSPATH=$CLASSPATH:$L/com.ibm.mq.jar:$L/com.ibm.mqjms.jar
CLASSPATH=$CLASSPATH:/opt/mqm/samp/jms/samples:/opt/mqm/samp/wmqjava/sa
mples
export CLASSPATH

#####

/opt/mqm/java/bin/JMSAdmin -v -cfg JMSAdmin.config <<!EOF

DEF CF(RHHOST_CF) QMGR(RHHOST_QM) TRANSPORT(CLIENT) HOSTNAME(localhost)
PORT(1425)
DEF Q(RHHOST_DEST) QMGR(RHHOST_QM) QU(RHHOST_Q)

end

!EOF
```

Consider the following information about Example 8-4:

- ▶ After setting a large number of environment variables (all default paths), we run the **JMSAdmin** command, with reference to the sample configuration file we manually edited (JMSAdmin.config).
- ▶ The bulk of what we do inside JMSAdmin is we make a *connection factory* (referred to as CF) and a *destination* (referred to as Q). This set of steps will output a .bindings file, which is one of two important ASCII text configuration files required to get this example to work.

Note: As stated, the `.bindings` file is one of two critical files required to get the two example Streams applications to work:

- ▶ The `.bindings` file is an WebSphere MQ required file, and is created here as detailed in Example 8-4.
- ▶ The `connections.xml` file is a Streams required file, which is detailed in 8.4, “The two Streams applications” on page 175.

Where was the `.bindings` file output, and where should it reside?

The value of `PROVIDER_URL`, in the `JMSAdmin.config` file, specified this output directory. We found if we moved the `.bindings` file, we had to rerun these **JMSAdmin** commands to create or re-create the connection factory and destination. And we could not (re-create) these objects until we first deleted them.

To delete these or other objects, we used the WebSphere MQ Explorer graphical administration program. These objects resided under the specific WebSphere MQ queue manager, then under JMS Administered Objects.

Example 8-5 makes one final change to WebSphere MQ resident object permissions. See the code review after Example 8-5.

Example 8-5 Last change to MQ permissions

```
export PATH=$PATH:/opt/mqm/bin:/opt/mqm/samp/bin

runmqsc RHHOST_QM <<!EOF

ALTER QMGR CHLAUTH(DISABLED)

!EOF
```

Consider the following information about Example 8-5:

- ▶ Again, we are essentially granting all permissions, or removing restrictive permissions from an WebSphere MQ resident object or objects. Here we open the associated WebSphere MQ queue manager channel authorizations (permissions).
- ▶ Our goal here was to get the examples that follow to work. In a production environment, a skilled WebSphere MQ administrator implements real-world permissions; a topic we did not want to detail here.

Thus completes the creation and configuration of all WebSphere MQ resident objects, and all of the work we must perform as the mqm user. The remainder of work takes place inside Streams.

8.3 Setting Streams environment variables, and adding toolkits

For the Streams run time to properly locate the WebSphere MQ client libraries, several Streams resident environment variables must be set:

- ▶ With a default installation directory for Streams, there is a specific Streams system file where environment variables are set. The default full path name to this file is as follows:

```
/opt/ibm/InfoSphereStreams/bin/streamsprofile.sh
```

- ▶ So that the two example Streams applications can work with WebSphere MQ, we must set three environment variables in this file:

```
– export XMS_HOME=/opt/IBM/XMS
– export MQ_HOME=/opt/mqm
– export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/mqm/lib64
```

Note: This assumes all default values for directory installation path names.

- ▶ We placed these three new lines at the bottom of `streamsprofile.sh` file, above the return statement. To ensure full observance of the changes, we logged out and logged back in as `streamsadmin`.

We must also make the location of the Streams Messaging Toolkit known to the Streams Developer's Workbench, and to any specific Streams projects that want to use a single or set of toolkits. Figure 8-1 on page 172 displays the Streams Explorer view, before any Streams toolkits locations are specified.

Note: In Figure 8-1 you see only the Standard Toolkit, which forms the core of Streams. From the display in Figure 8-1, we can determine that this installation of Streams does not currently point to the additional toolkit, which offers access to the Messaging Toolkit and more, a condition we will address.

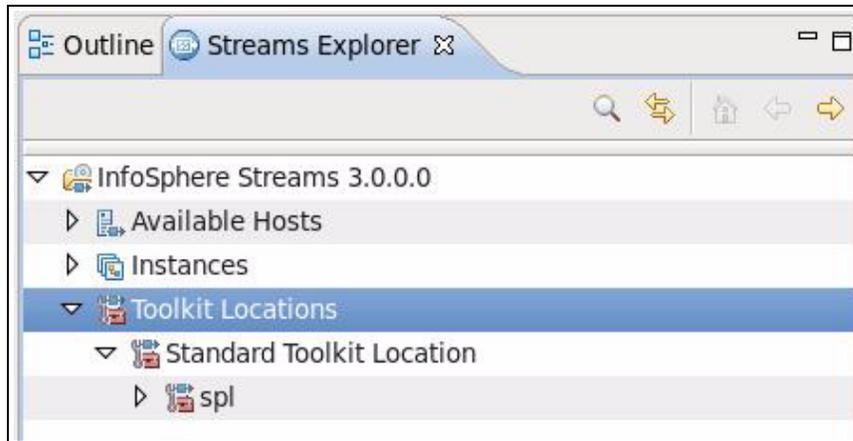


Figure 8-1 Default Streams Explorer view, no toolkit locations specified

Complete the following steps:

1. In the Streams Explorer view, right-click **Toolkit Locations** and select **Add Toolkit Locations**.
2. Assuming default installation directories, browse to the `/opt/ibm/InfoSphereStreams/toolkits` location, as shown in Figure 8-2.
3. Click **OK** to complete.

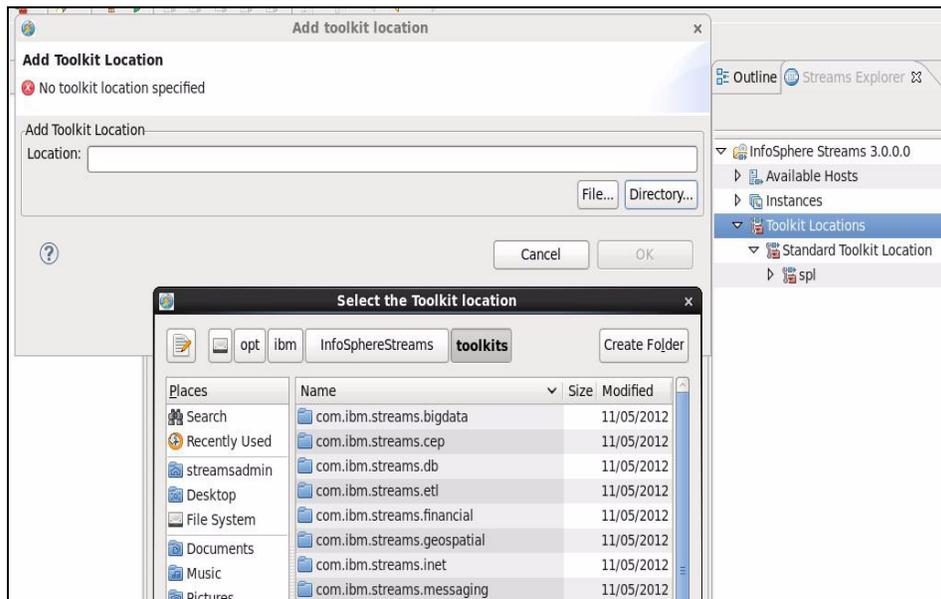


Figure 8-2 Adding toolkit locations to the Streams Developer's Workbench

Figure 8-3 shows a Streams Developer's Workbench with the Streams toolkits successfully identified.

Note: These steps make the toolkits known only to the Streams Developer's Workbench. You must still add any individual toolkits to each specific Streams project, as detailed in Figure 8-4.

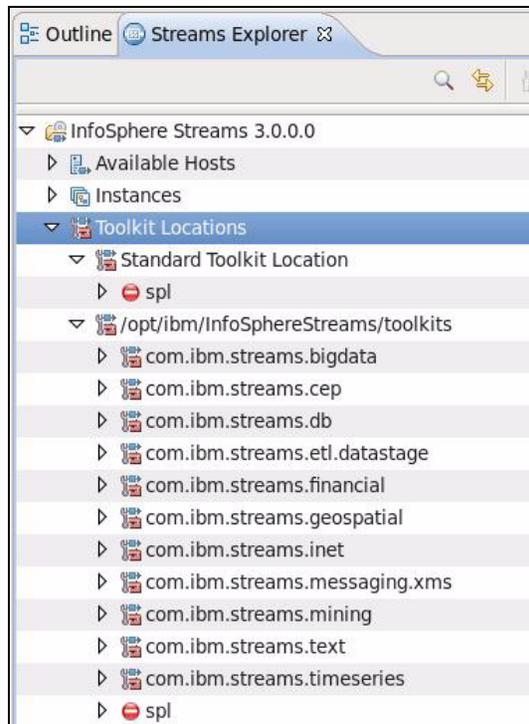


Figure 8-3 Streams Developer's Workbench

Figure 8-4 shows how to add a specific Streams toolkit to a Streams project.

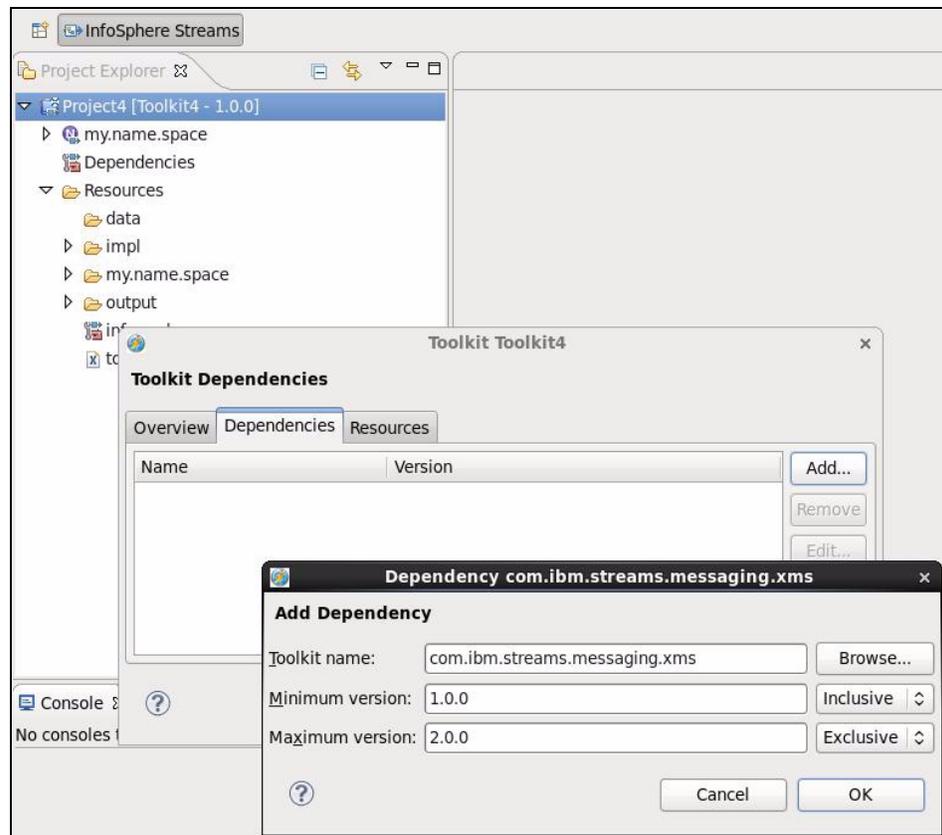


Figure 8-4 Adding toolkits to a specific Streams project

Complete the following steps:

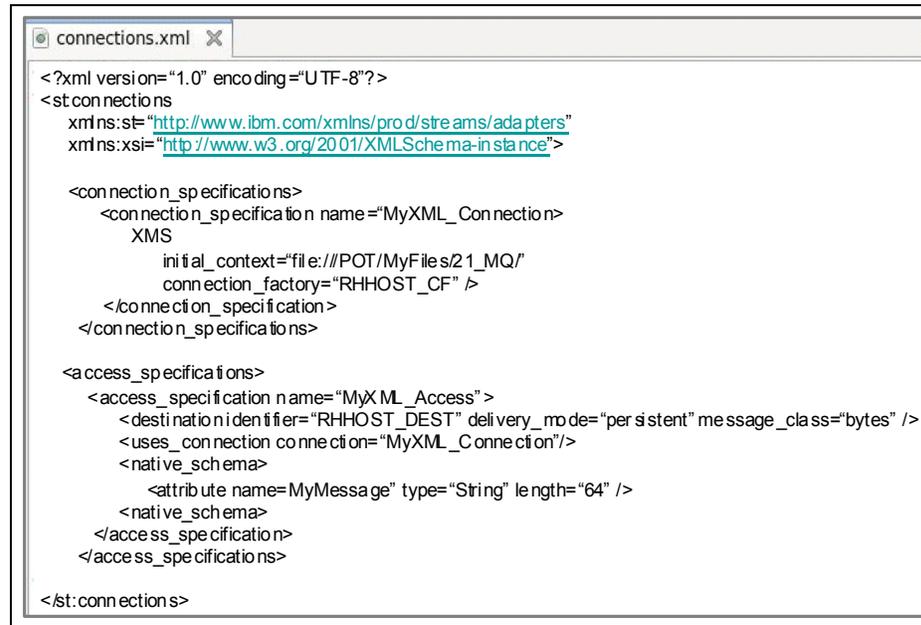
1. In the Streams Project Explorer view, right-click the Streams project in which you want to complete these examples. From the context sensitive menu that is produced, select **Edit Toolkit Information**, then select the **Dependencies** tab.
2. Select the Messaging Toolkit, and Click **OK**.

You are now ready to complete the two example Streams applications that demonstrate interoperability with WebSphere MQ.

8.4 The two Streams applications

Previously, we stated that two ASCII text configuration files are critical to achieve Streams and WebSphere MQ interoperability. The first file, `.bindings`, is generated as the result of creating given MQ resident objects. The second file, `connections.xml`, is simple and is easily created by hand.

Figure 8-5 displays the sample `connections.xml` file. See the code review after Figure 8-5.



```
<?xml version="1.0" encoding="UTF-8"?>
<st:connections
  xmlns:st="http://www.ibm.com/xmlns/prod/streams/adapters"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <connection_specifications>
    <connection_specification name="MyXML_Connection">
      XMS
      initial_context="file://POT/MyFiles/21_MQ/"
      connection_factory="RHHOST_CF" />
    </connection_specification>
  </connection_specifications>

  <access_specifications>
    <access_specification name="MyXML_Access">
      <destination_identifier="RHHOST_DEST" delivery_mode="persistent" message_class="bytes" />
      <uses_connection_connection="MyXML_Connection"/>
      <native_schema>
        <attribute name="MyMessage" type="String" length="64" />
      </native_schema>
    </access_specification>
  </access_specifications>

</st:connections>
```

Figure 8-5 Streams resident `connections.xml` file, used by the Messaging Toolkit

Minus the standard XML markings for namespace and related information, there are essentially two sections of importance in Figure 8-5:

- ▶ `<connection_specifications>`
 - The value for `connection_specification name` is a value we chose (`MyXML_Connection`), and will be referred to in our Streams applications.
 - The value for `initial_context` specifies the directory where we generated and placed our `.bindings` file, detailed previously. Notice the specific formatting, because it is required.
 - The value for `connection_factory` is a value we chose (`RHHOST_CF`), when creating objects inside WebSphere MQ.

- ▶ <access_specifications>
 - The value for `access_specification` name is a value we chose (MyXML_Access), and will be referred to in our Streams applications.
 - The value for `destination` identifier is a value we chose (RHHOST_DEST), when creating objects inside WebSphere MQ.
 - The value for `uses_connection` connection (MyXML_Connection) relates this entry with the connection specification.
 - Within the `native_schema` block, we specify the format of the message that we will send and receive. Our example, as displayed, sends and receives one column of type string (WebSphere MQ), rstring (Streams). This one column is named, MyMessage, which we chose.

Note: In this example we chose to send and receive one column of type string, length 64. If we retrieve a message that is longer than 64 characters, the message will be truncated. If the message is shorter than 64 characters, it will be padded.

There is a variable length string type capability in the operability between Streams and WebSphere MQ; however, we suggest you choose the longest string length for string values that you require.

To make use of the variable length string capability, and when using the byte message class type, specify a length of -2, -4, or -8. These values allow for a variable length string value within certain length ranges. These -2, -4, -8 values work only for byte class messages. The other message class types allow for variable length string and messages inherently.

There are message classes other than type bytes, XML for example. Certainly, you can choose to send and receive more than one column. After the example we provide is working for you, the Streams online help offers more information to help you develop new and more complex (message) types.

Figure 8-6 displays the first Streams application we create, an application that writes to WebSphere MQ. First we write to WebSphere MQ so that we have something to read in the next example.

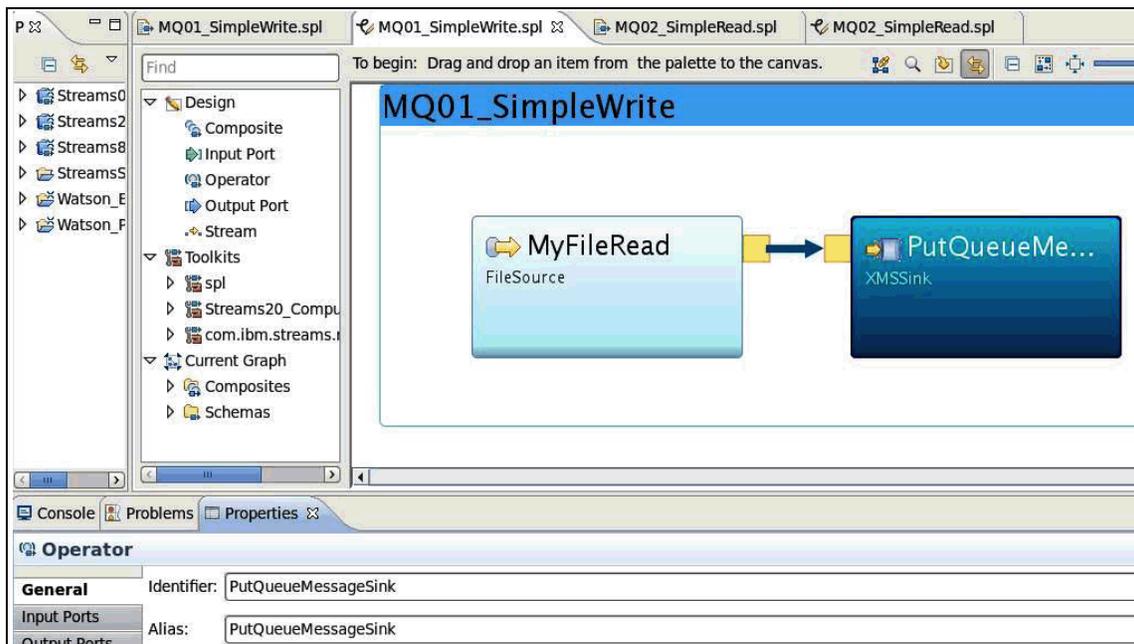


Figure 8-6 The first Streams application we create, write to MQ

Although we can entirely paint (drag, drop, and draw) the Streams application that displayed in Figure 8-6, a quicker and easier way is probably to display its Streams Processing Language (SPL) specification, as shown in Figure 8-7 on page 178. See the code review after Figure 8-7 on page 178.

```
MQ01_SimpleWrite.spl  MQ02_SimpleRead.spl

namespace Just02_MQ;

use com.ibm.streams.messaging.xmls::XMSSink ;

composite MQ01_SimpleWrite {
graph

stream<rstring MyMessage> MyFileRead = FileSource() {
param
file : "/POT/MyFiles/21_MQ/MQ01.Input.txt";
format : line;
}

() as PutQueueMessageSink = XMSSink(MyFileRead) {
param
connectionDocument : "/POT/MyFiles/21_MQ/connections.xml" ;
connection : "MyXML_Connection" ;
access : "MyXML_Access" ;
}
}
```

Figure 8-7 Example 1, write to MQ, XMSSink

Consider the following information about Figure 8-7:

- ▶ The use statement declares which toolkit we intend to use, and more specifically, which operator contained within we need to make reference to. XMSSink is the Streams Messaging Toolkit operator used to write to WebSphere MQ.
- ▶ A FileSource operator is used to read text that we will send to WebSphere MQ.
- ▶ XMSSink is the Streams Messaging Toolkit operator to write to WebSphere MQ:
 - The connectionDocument parameter refers to the absolute path name to the connections.xml document we created.
 - The connection parameter is the name of the specific entry from the connection_specifications section, from the connections.xml document, we want to make reference to.

- Similarly, the access parameter makes reference to the specific entry from the access_specifications section.

Figure 8-8 displays the WebSphere MQ and Streams object hierarchy, including objects we created in this chapter so far. See the code review after Figure 8-8.

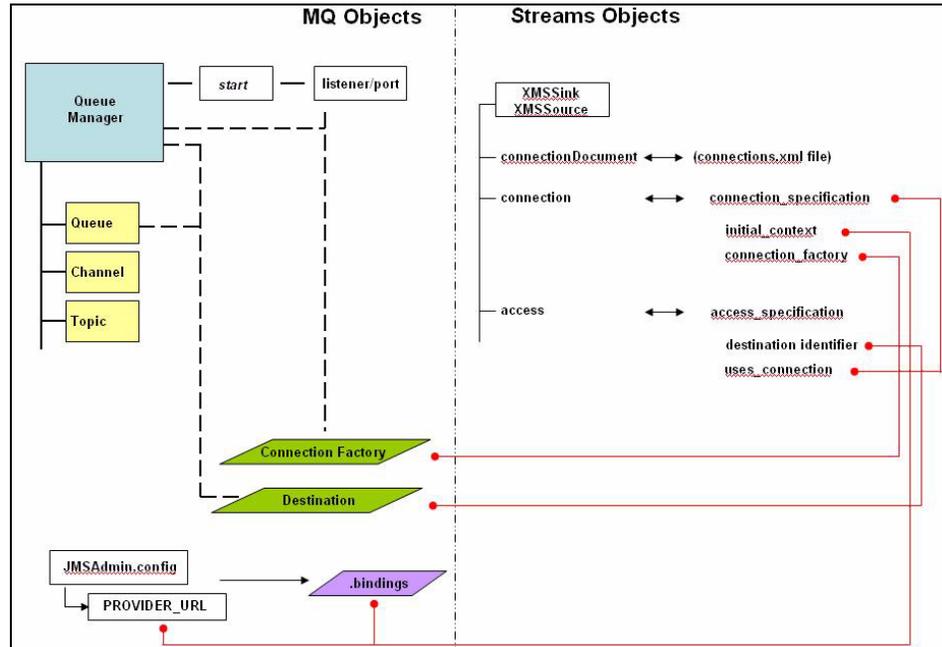


Figure 8-8 WebSphere MQ and Streams object hierarchy

Consider the following information about Figure 8-8:

- ▶ WebSphere exists as a server, and any Streams application is a client to the WebSphere MQ server. So although the WebSphere MQ resident objects must exist and be accessible, ultimately we are concerned with the ability for Streams to locate and decode reference to any WebSphere MQ resident objects.
- ▶ Parameters in the Streams operators, XMSSink and XMSSource, specify the following information:
 - The full path name to the connections.xml file. All subsequent information is read from that file.
 - The connection parameter relates to an entry inside connections.xml:
 - Here we retrieve the initial_context, which is used to locate and read the .bindings WebSphere MQ configuration file.

- Here we retrieve the identifier for the WebSphere MQ object connection factory.
- The destination value gives us the identifier of the destination MQ object.

Figure 8-9 displays the Streams application to read the message we placed in WebSphere MQ in Figure 8-7 on page 178. See the code review after Figure 8-7 on page 178.

```

MQ02_SimpleRead.spl

namespace Just02_MQ;

use com.ibm.streams.messaging.xmls::XMSSource ;

composite MQ02_SimpleRead {
graph
    stream<rstring MyMessage> MyXMLMessage = XMSSource() {
    param
        connectionDocument : "/POT/MyFiles/21_MQ/connections.xml" ;
        connection         : "MyXML_Connection" ;
        access              : "MyXML_Access" ;
        initDelay           : 2.0 ;
    }

    () as MySink = FileSink(MyXMLMessage) {
    param
        file      : "/POT/MyFiles/21_MQ/MQ02.Output.txt";
        format    : csv;
        flush     : lu;
    }
}

```

Figure 8-9 Example 2, read from MQ, XMSSource

Consider the following information about Figure 8-9:

- ▶ Although this example uses an XMSSource operator to read from MQ (versus the XMSSink operator to write), this Streams application offers almost no difference from our first application, which writes to WebSphere MQ.

- ▶ Installing and configuring for Apache ActiveMQ: Describes in detail the installation and configuration steps on client and server machines when Apache ActiveMQ is used as the messaging system.
- ▶ Compiling and running sample applications: Describes the environment variables to be set and steps to import and run the sample applications demonstrating the capability of JMSSource and JMSSink for both JMS providers.

Other topics include sample applications and verifying the results

We assume that you have basic to moderate skills on both InfoSphere Streams and the messaging system used (WebSphereMQ or Apache ActiveMQ).

8.5.1 Example use cases for the JMS adapters

This section has two example use cases in which JMS adapters can be used.

Real-time campaign management

Consider the case of a large bank that has many credit card customers. Imagine if someone performs a credit card purchase in an expensive mall. We know where the transaction was committed, which enables the bank to offer various cross-sell and up-sell products. The transaction committed is captured by Streams for processing and posting the message to the Messaging queue layer. IBM Marketing Operations OnDemand, formerly known as IBM Unica® Marketing Operations OnDemand, is used to connect to the messaging queue and instantiates the campaign. This example illustrates how organizations are using service-oriented architecture (SOA) technologies to integrate their enterprise systems and how Messaging Toolkit adapters fit into the ecosystem seamlessly to respond in real time, and increase efficiency and timeliness of the processes.

Tracking stolen vehicles

Department of motor vehicles (DMV) issues licences to many vehicles. Electronic toll collection allows for electronic collection of tolls by identifying cars and electronically debiting the account of registered car owners. Information from electronic toll collection can be used for smarter crime detection and prevention. The following scenarios are examples:

- ▶ Tracking stolen devices: When a device theft is reported to enforcers, an entry is made to systems with details about stolen devices. When a car containing the stolen device passes a toll, it can be flagged and sent to business performance management (BPM) systems. Because BPM systems are mostly compliant with WebSphere MQ or Active MQ, data from Streams can be sent through JMSSink to them, which in turn are coupled to BPM systems.

These can issue necessary alert to respective officials about the last whereabouts of the stolen car and thus assists them to zero in on location.

- ▶ Tracking stolen cars: Assuming all toll information is continuously sent and filtered against the stolen car list by Streams, when the stolen car passes a toll, it can be flagged and sent to BPM systems. Because BPM systems are mostly compliant with WebSphere MQ or Active MQ, data from Streams can be sent through JMSSink to these, which in turn are coupled to BPM systems. These systems can issue a necessary alert to respective officials about the last location of the stolen car and thus assists them to zero in on location.

8.5.2 Installing and configuring of WebSphere MQ

This section describes the necessary steps for using WebSphere MQ as the messaging system. The machine where Streams is installed and from where JMS adapters are used is referred to as a *client machine*. The machine where WebSphere MQ Server will be installed is referred to as the *server machine*.

Client machine: Installation and configuration steps

If the JMS provider is WebSphere MQ, the adapters require that the WebSphere MQ client libraries must be installed on the same machine as IBM InfoSphere Streams. Download the WebSphere MQ Client from the following site:

<http://www.ibm.com/software/integration/wmq/clients/>

At this site, click the WebSphere MQ client version that you want to download. On the resulting page, scroll to the “Download package” section. Click the HTTP or HTTPS option for the version that you want to download. Read the terms and conditions carefully, and when you agree click **I agree** to proceed with the steps. Click the installable file for the platform and operating system you want. If you have not already logged in, you are prompted to log in with your IBM ID. Select a preferred method of download and download the installable file. The name of the file might differ from the name in our example.

The following steps outline the installation of WebSphere MQ v7.5 Client libraries on a Red Hat Enterprise Linux (RHEL) v5.6, 64-bit machine, using the WebSphere MQ v7.5.0.2 client as an example:

1. Extract the installable file:

```
tar -xzf WS_MQ_CLIENT_LIN_X86_64_7.5.0.2.tar.gz
```
2. Run `mqlicense.sh` and accept the license agreement:

```
./mqlicense.sh
```
3. Install MQSeriesRuntime:

```
rpm -ivh MQSeriesRuntime-7.5.0-2.x86-64.rpm
```

4. Install MQSeriesClient:

```
rpm -ivh MQSeriesClient-7.5.0-2.x86-64.rpm
```

5. Install WebSphere MQSeriesJava libraries:

```
rpm -ivh MQSeriesJava-7.5.0-2.x86_64.rpm
```

The installation of WebSphere MQ client is now complete.

Server machine: Installation and configuration steps

Complete the following steps on the server machine. In this example, we describe the use of two separate servers.

Installing WebSphere MQ Server

The installation of WebSphere MQ server need not be on the same machine as InfoSphere Streams. Learn more about downloading WebSphere MQ Server at the following address:

<http://www.ibm.com/software/integration/wmq/>

The following steps outline the installation of WebSphere MQ v7.5 Server on a RHEL 5.6, 64-bit machine, using WebSphere MQ v7.5.0.2 Server as an example:

1. Extract the installable file:

```
tar -xzf WS_MQ_LINUX_ON_X86_64_V7.5.0.2.tar.gz
```

2. Run `mqlicense.sh` and accept the license agreement if installing on a machine separate from the one where WebSphere MQ Client is installed.

3. Install MQSeriesRuntime, if installing on a machine separate from the one where WebSphere MQ Client is installed:

```
rpm -ivh MQSeriesRuntime-7.5.0-2.x86_64.rpm
```

4. Install MQSeriesServer:

```
rpm -ivh MQSeriesServer-7.5.0-2.x86-64.rpm
```

Installation of WebSphere MQ v7.5 Server is now complete.

Installing MQ Explorer

Do the following steps on the machine that contains the MQ Server (perform the following steps as root user ID):

1. Install MQSeriesJRE:

```
rpm -ivh MQSeriesJRE-7.5.0-2.x86_64.rpm
```

2. Install MQSeriesExplorer:

```
rpm -ivh MQSeriesExplorer-7.5.0-2.x86_64.rpm
```

WebSphere MQ Explorer is now installed and ready to use. To launch WebSphere MQ Explorer, go to the `/opt/mqm/bin` directory and run the `./MQExplorer` command as a user of the `mqm` group.

Creating WebSphere MQ objects

If the JMS provider is WebSphere MQ, then a set of MQ objects must be created that will be used by the JMS adapters to configure connection to the MQ Server. MQ objects can be created either through the command line or through the more interactive MQ Explorer. In this section, we create the MQ objects through the MQ Explorer.

Note: In the following steps, for simplicity, we accept the default values for most parameters while creating the MQ objects. An expert MQ administrator can modify these to suit requirements.

1. Create a Queue Manager:
 - a. Expand IBM WebSphere MQ, right-click **Queue Managers**, and select **New** → **Queue Manager**.
 - b. In the Enter basic values window, provide a Queue Manager name. Click **Next** three times. Our example uses `TestQueueManager` as the name for the Queue Manager.
 - c. In the Enter listener options window, ensure that the Listen on port number is set to a port value that is unassigned. We set this to 1440.
 - d. Click **Finish**.
2. Disable authentication on the queue manager:
 - a. Open a command prompt.
 - b. In the `/opt/mqm/bin` directory, run `runmqsc` for the Queue Manager previously created:

```
./runmqsc TestQueueManager
```
 - c. Set the authentication to disabled for that Queue Manager:

```
ALTER QMGR CHLAUTH(DISABLED)
```
 - d. Type `END` to exit from the `mqsc` prompt.

Security: Although for simplicity in this book, we disable authentication on the Queue Manager, from a security perspective, doing so is not advisable because this gives access to the Queue Manager to everyone. In a real world scenario, an expert MQ administrator can modify authentication records to suit requirements.

If writing to or reading from a queue, do the following step:

- ▶ Create a queue:
 - a. Expand the TestQueueManager created in step 1 on page 185 (Create a Queue Manager), right-click **Queues**, and select **New** → **Local Queue**.
 - b. In the Create a Local Queue window, provide a queue name. Our example uses TestQueue as the name for the queue.
 - c. Click **Finish**.

If writing to or reading from a topic, use the following steps:

1. Create a topic:
 - a. Expand the TestQueueManager created in step 1 on page 185 (Create a Queue Manager), right-click **Topics**, and select **New** → **Topic**.
 - b. In the Create a Topic window, provide a topic name. Our example uses TestTopic as the topic name. Click **Next**.
 - c. In the Change Properties window, provide a topic string. We use TestTopicIdentifier as the value for the Topic string.
 - d. Click **Finish**.
2. Create a subscription queue for the topic:

Create a local queue on similar lines as described in “Create a queue:” on page 186 (but with a different name); this will act as a subscription queue for this topic. We created a queue named TestTopicQueue, which is our subscription queue.
3. Create a subscription:
 - a. Expand the TestQueueManager created in step 1 on page 185 (Create a Queue Manager), right-click **Subscriptions**, and select **New** → **Subscription**.
 - b. In the Create a Subscription window, provide a subscription name. We use TestSubscription as the name for the subscription. Click **Next**.
 - c. In the Change properties window, under Topic, select the topic previously created, **TestTopic**.
 - d. Enter the topic string TestTopicIdentifier provided while creating the topic for Topic string under Topic.
 - e. In the Destination name field, provide the name of the Subscription queue: TestTopicQueue, for this topic.
 - f. Click **Finish**.

For both reading to or writing from a queue or topic, use the following steps:

1. Add initial context:
 - a. Under IBM WebSphere MQ, right-click **JMS Administered objects**, and select **Add Initial Context**.
 - b. In the Connection details window, select **File system** for Where is the JNDI namespace located? Provide a valid value for the bindings directory under JNDI Namespace Location. The bindings directory should point to a directory where the user ID creating this initial context has write permissions. We use `/var/mqm/Bindings` as the location of the bindings directory.
 - c. Click **Finish**.
2. Add a connection factory object:
 - a. Expand the initial context labeled `file:/var/mqm/Bindings` created in step 1a (JMS Administered Objects), right-click **Connection Factories**, and select **New** → **Connection Factory**.
 - b. In the Enter the details area of the Connection Factory window, provide a name for the connection factory. For example, we use `TestConnFac` as the name for the connection factory. Click **Next** twice.
 - c. In the “Select the transport” that the connections will use area of the window, select **MQ Client** under transport. Click **Next** twice.
 - d. On the Change properties page, under Connection, specify `<host>(<port>)` in the Connection list. For example, we provide `f0819b10.pok.hpc-ng.ibm.com(1440)` as the value. The number 1440 is the port we mentioned while creating the queue manager.

When the client and server are two separate machines, it is important that the host mentioned here is a host name IP address that the client machine can resolve appropriately and connect to this server. Hence using `localhost` is discouraged.
 - e. Click **Finish**.
3. Add a destination object:
 - a. Expand the initial context labeled `file:/var/mqm/Bindings` (located under JMS Administered Objects), right-click **Destination**, and select **New** → **Destination**.
 - b. On the Create a Destination window, provide a name for the destination. For example, we created two destination objects: `TestDest` and `TestDestTopic`. Click **Next** twice.

Complete these steps:

- i. On the Change properties page, under the General tab, for Queue Manager, select the **Queue Manager TestQueueManager** that was previously created.
- ii. On the same page, for Queue, do the following steps based on whether you want to read from and write to *queue*, read from and write to a *topic*, or *both*:
 - *For reading from and writing to a queue:*
Select the **Queue TestQueue** that was previously created.
 - *For reading from or writing to a topic:*
Select the subscription queue **TestTopicQueue** that was previously created
 - *For reading from or writing to both the queue and topic:*
Based on the requirement, create two destination objects with different names, one for queue and one for topic. In our example, the destination object named TestDest is the destination object that has been created for reading from and writing to a queue; the destination object named TestDestTopic is the destination object created for reading from and writing to the topic.
- c. Click **Finish**.

A `.bindings` file is now created in `/var/mqm/Bindings` and contains the connection factory and destination object information. This file will be needed by the client machine to connect to the queues or topics that have been created. Copy this file to the client machine in a location accessible to Streams. We copied it to the `/tmp/Bindings` location.

8.5.3 Installing and configuring for Apache ActiveMQ

This section describes the steps for using Apache ActiveMQ as the messaging system. The machine where Streams is installed and from where JMS adapters will be used is referred to as a *client machine*. The machine where Apache ActiveMQ Server will be installed is referred to as the *server machine*.

Note: All the steps in this section have been performed as root.

Client and server machines: Installation of Apache ActiveMQ is required

If the JMS provider is Apache ActiveMQ v5.7, then the Apache ActiveMQ libraries must be installed on same machine as IBM InfoSphere Streams because the JMSSink and JMSSource operators require the libraries that are installed with Apache ActiveMQ. However, the ActiveMQ instance to which a message is sent can be on a separate machine. The following steps outline installation of Apache ActiveMQ v5.7 on a RHEL 5.6, 64-bit machine. These steps must be done on both the client and the server machines:

1. Extract the installable file:

```
tar -xzf apache-activemq-5.7.0-bin.tar.gz
```

2. The Apache ActiveMQ should now be installed along with the libraries needed by JMS adapters.

Server machine: Configuration steps

Follow these steps only on machines to which messages must be sent to start the Apache ActiveMQ server:

1. Set JAVA_HOME if it is not set already. Our example uses the following setting:

```
export JAVA_HOME=/opt/ibm/java-x86_64-60
```

2. In the bin folder inside the extracted folder, start **activemq**:

- a. `cd apache-activemq-5.7.0/bin`
- b. `./activemq start`

The Apache ActiveMQ server is now running.

8.5.4 Compiling and running sample applications

The JMS adapters include sample SPL applications that demonstrate the capability of these operators. The samples can be run either with WebSphere MQ as the JMS provider or with Apache ActiveMQ as the JMS provider, or with both. This section describes the environment variables to set, and steps to import the sample, and configure, compile and run it.

Setting the environment variables

Set the variables as follows:

- ▶ When the JMS provider is Apache ActiveMQ:

The `STREAMS_MESSAGING_AMQ_HOME` environment variable must be set to the installation directory of Apache ActiveMQ, as in the following example:

```
export STREAMS_MESSAGING_AMQ_HOME=/opt/ActiveMQInstallables/apache-activemq-5.7.0
```

- ▶ When the JMS provider is WebSphere MQ:

The environment variable `STREAMS_MESSAGING_WMQ_HOME` must be set to the installation directory of WebSphere MQ Client, as in the following example:

```
export STREAMS_MESSAGING_WMQ_HOME=/opt/mqm
```

- ▶ If an application contains multiple JMS adapters, where some have Apache ActiveMQ as the JMS providers and some have WebSphere MQ as the JMS providers, both of the environment variables must be set.

Making the Messaging Toolkit available in Streams Studio

Use these steps to make the toolkit available in the Streams studio interface:

1. In the Streams Explorer view, right-click **Toolkit Locations** and select **Add Toolkit Location**.
2. Browse to the toolkits directory under the InfoSphere Streams installation directory and select **com.ibm.streams.messaging**.

After successfully adding the toolkit, the Streams Explorer view should look as depicted in Figure 8-11 on page 191.

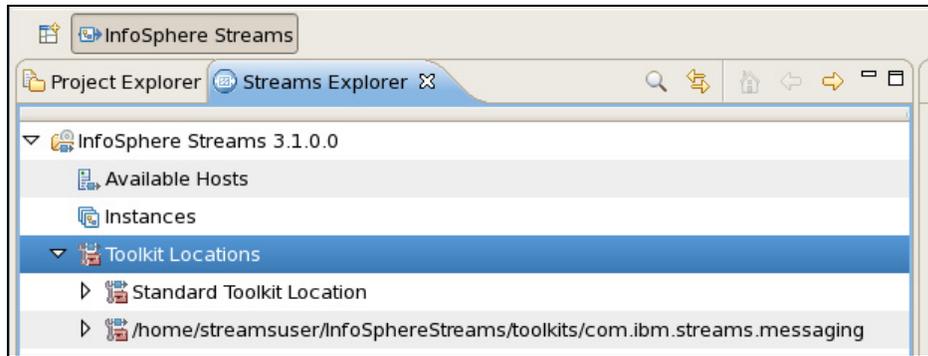


Figure 8-11 Streams Explorer view

Importing the samples

The JMSSource and JMSSink samples are located in the following folder:

`$STREAMS_INSTALL/toolkits/com.ibm.streams.messaging`

To import the samples to the workspace, complete the following steps:

1. In the Project Explorer view, click **File** → **Import**.
2. In the import window (Figure 8-12 on page 192), select **Samples SPL Application** under InfoSphere Streams Studio.
3. Expand **Toolkit Locations**, expand **com.ibm.streams.messaging 2.0.0**, and then select **JMSSource** and **JMSSink**.

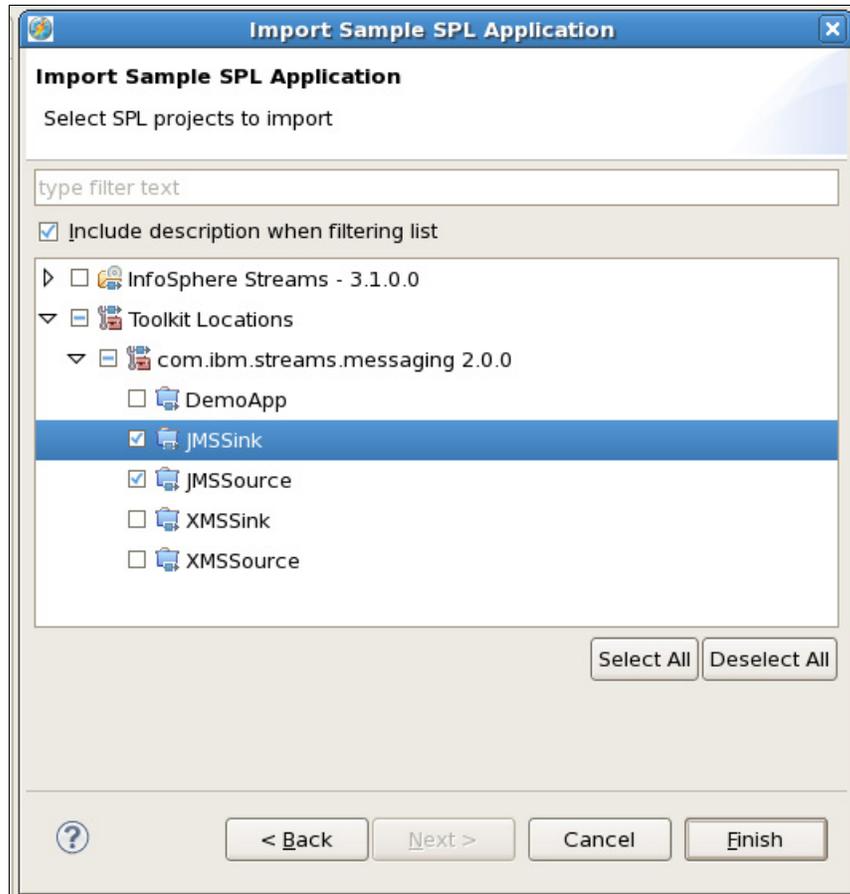


Figure 8-12 Import application

4. Click **Finish**.

After importing, the Project Explorer view is similar to Figure 8-13.

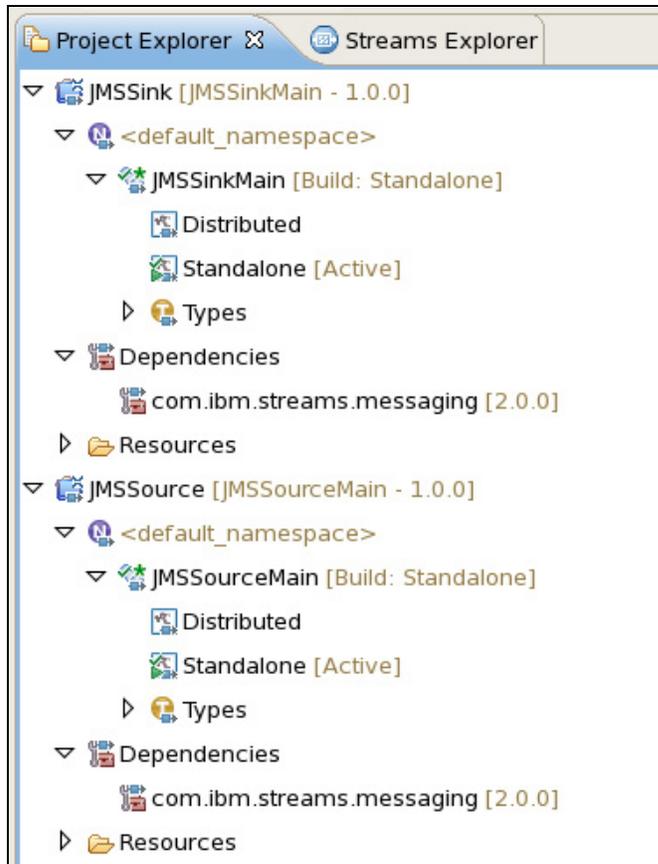


Figure 8-13 Project Explorer view

Modifying the connection specification document

The connection specification document is an XML file that details the connection information to the JMS providers. The optional `connectionDocument` parameter specifies the path name of this XML file. If the parameter is absent, the operator uses the default location file path `/etc/connections.xml` (regarding the application data directory).

The general guidelines for modifying the connection specification document are described in the following list. For a more specific example of connection specification document, see the modified connection specification document for WebSphere MQ (Listing 1) or the modified connection specification document for Apache ActiveMQ (Listing 2).

- ▶ The value for `connection_specification` name is a user-chosen string and is used in the SPL application as a value for the connection parameter to refer to the connection information to be used by the particular operator instance. We use `conn1` as the value in our examples.
- ▶ The value for `access_specification` name is a user-chosen string and is used in the SPL application as a value for the access parameter to refer to the format and content of the message being sent or received by the particular operator instance. We use `access1` as the value in our examples.
- ▶ The value for `uses_connection` in the `access_specification` element is used to refer to the `connection_specification` to be used by this `access_specification` element. Because we use `conn1`, we specify `conn1` as the value for `uses_connection`.
- ▶ The `native_schema` lists name and type of all the attributes of the message either to be sent or to be expected to receive.
- ▶ The `message_class` in the destination element specifies how a stream tuple is to be serialized into a MQ message. We set this to `map` in our example.

Note: `Message_class` is an involved topic and is outside the scope of this article. For details about message class values supported by JMS adapters, see the IBM InfoSphere Streams Information Center:

<http://pic.dhe.ibm.com/infocenter/streams/v3r1/index.jsp?topic=%2Fcom.ibm.svg.im.infosphere.streams.homepage.doc%2Fdoc%2Fic-homepage.html>

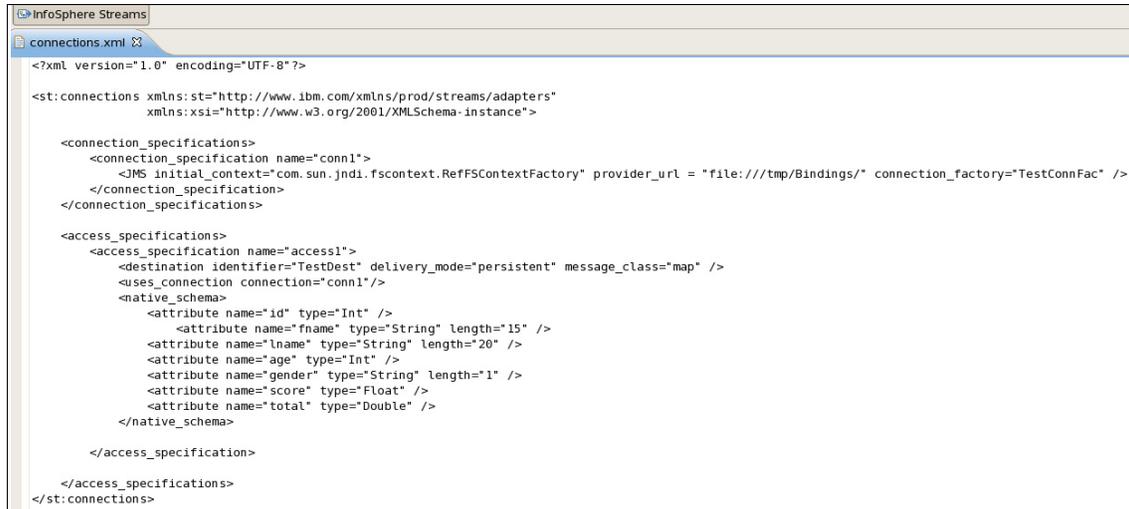
Table 8-1 summarizes the modification guidelines for the other elements in the connection specification document.

Table 8-1 Modification guidelines

Element name	General description	When JMS provider is WebSphere MQ	When JMS provider is Apache Active MQ
<code>initial_context</code>	Represents the class name of the factory class that will create an initial context.	If the JMS provider is WebSphere MQ and is using a bindings file for connection, then the value of the <code>initial_context</code> is <code>com.sun.jndi.fscontext.RefFSContextFactory</code> .	If the JMS provider is Apache ActiveMQ, then, the <code>initial_context</code> can be <code>org.apache.activemq.jndi.ActiveMQInitialContextFactory</code> .

Element name	General description	When JMS provider is WebSphere MQ	When JMS provider is Apache Active MQ
provider_url	This is a URL that points to the directory service containing the administered objects or the JNDI URL string.	If the JMS provider is WebSphere MQ, if bindings file is being used, then this is set to <code>file:///homes/abc/xyz/wef/</code> if the directory <code>/homes/abc/xyz/wef</code> contains a <code>.bindings</code> file with information about the administered objects. In our example, if we move the <code>.bindings</code> file in <code>/var/mqm/Bindings</code> from the server machine and place it at <code>/tmp/Bindings</code> on the client machine, then this is set to <code>file:///tmp/Bindings/</code> .	If the JMS provider is Apache ActiveMQ, then the provider_url will be similar to <code>tcp://machinename.com:61616</code> . In our example, it is set to <code>tcp://charak.in.ibm.com:61616</code> .
connection_factory	This is the name of the ConnectionFactory administered object (within the directory service context specified by the initial_context attribute) to be used to make the connection.	If the JMS provider is WebSphere MQ, then this should be set to the connection factory created through the process in the Add a connection factory object (step 2) in the "Creating WebSphere MQ objects" on page 185. We set this to <code>TestConnFac</code> in our example.	If the JMS provider is Apache ActiveMQ, then a default connection factory by the name ConnectionFactory is available for use.
destination_identifier	This is the name of the Destination administered object.	For WebSphere MQ, this is the name of the Destination administered object (within the directory service context specified by the connection specification's initial_context attribute) to be used. We set this to <code>TestDest</code> when we interact with a queue, and to <code>TestTopicDest</code> when we interact with a topic in our example.	For Apache ActiveMQ, because creating destinations at the beginning is unnecessary, this will refer to a queue name that must be created, updated, or read through this operator. In our example, we set this to <code>dynamicQueues/MapQueue</code> .

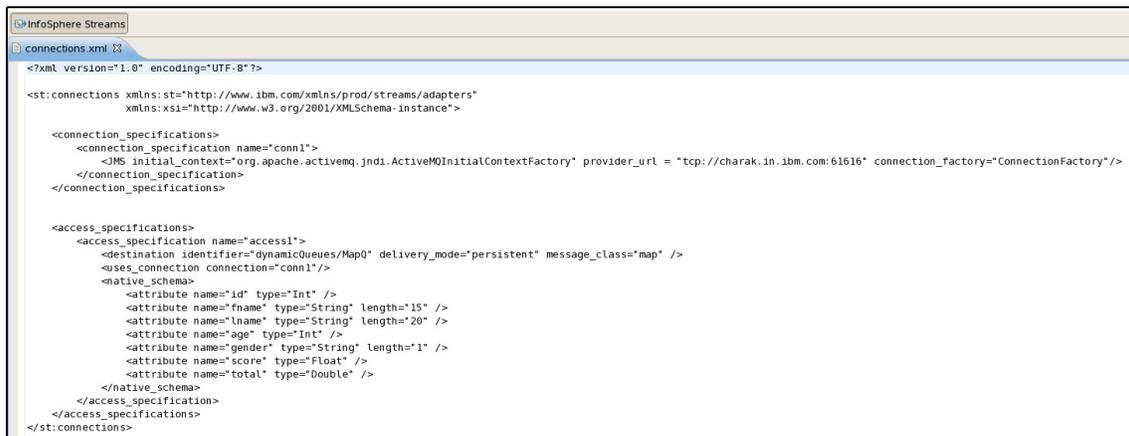
The modified connection specification document for WebSphere MQ as the JMS provider is shown in Figure 8-14.



```
<?xml version="1.0" encoding="UTF-8"?>
<st:connections xmlns:st="http://www.ibm.com/xmlns/prod/streams/adapters"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <connection_specifications>
    <connection_specification name="conn1">
      <JMS_initial_context="com.sun.jndi.fscontext.RefFSContextFactory" provider_url = "file:///tmp/Bindings/" connection_factory="TestConnFac" />
    </connection_specification>
  </connection_specifications>
  <access_specifications>
    <access_specification name="access1">
      <destination_identifier="TestDest" delivery_mode="persistent" message_class="map" />
      <uses_connection connection="conn1"/>
      <native_schema>
        <attribute name="id" type="Int" />
        <attribute name="fname" type="String" length="15" />
        <attribute name="lname" type="String" length="20" />
        <attribute name="age" type="Int" />
        <attribute name="gender" type="String" length="1" />
        <attribute name="score" type="Float" />
        <attribute name="total" type="Double" />
      </native_schema>
    </access_specification>
  </access_specifications>
</st:connections>
```

Figure 8-14 Connection document for WebSphere MQ

The modified connection specification document for Apache ActiveMQ as the JMS provider is shown in Figure 8-15.



```
<?xml version="1.0" encoding="UTF-8"?>
<st:connections xmlns:st="http://www.ibm.com/xmlns/prod/streams/adapters"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <connection_specifications>
    <connection_specification name="conn1">
      <JMS_initial_context="org.apache.activemq.jndi.ActiveMQInitialContextFactory" provider_url = "tcp://charak.in.ibm.com:61616" connection_factory="ConnectionFactory"/>
    </connection_specification>
  </connection_specifications>
  <access_specifications>
    <access_specification name="access1">
      <destination_identifier="dynamicQueues/Map0" delivery_mode="persistent" message_class="map" />
      <uses_connection connection="conn1"/>
      <native_schema>
        <attribute name="id" type="Int" />
        <attribute name="fname" type="String" length="15" />
        <attribute name="lname" type="String" length="20" />
        <attribute name="age" type="Int" />
        <attribute name="gender" type="String" length="1" />
        <attribute name="score" type="Float" />
        <attribute name="total" type="Double" />
      </native_schema>
    </access_specification>
  </access_specifications>
</st:connections>
```

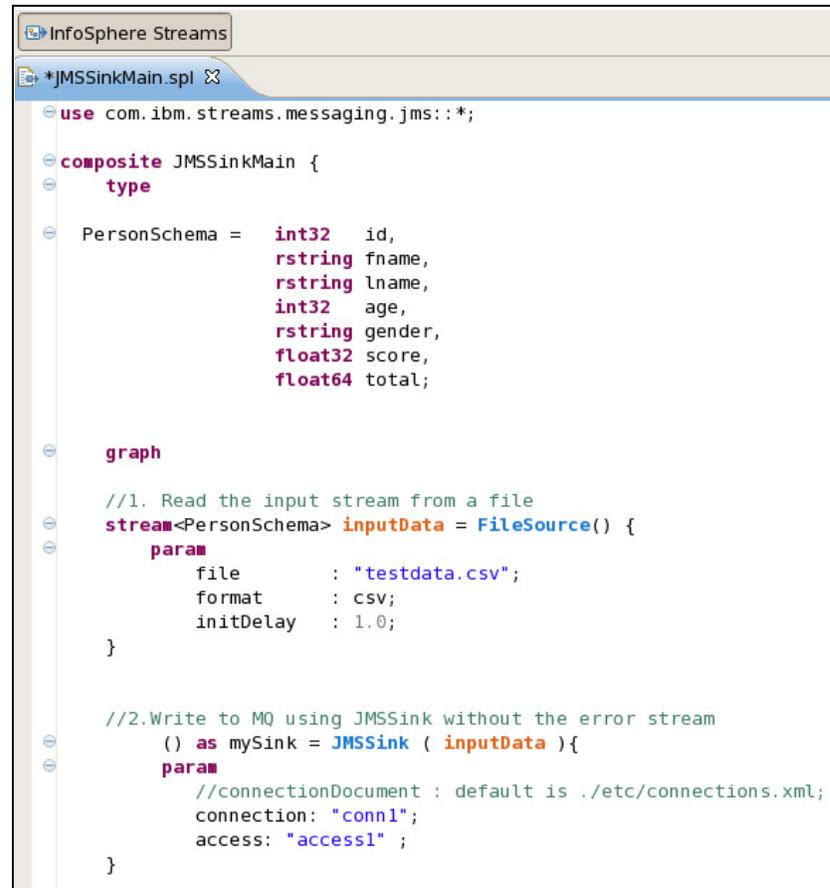
Figure 8-15 Connection specification document for ApacheActiveMQ

8.5.5 Sample applications

Several sample applications are described in this section.

The JMSSink SPL application

A sample JMSSink SPL application is shown in Figure 8-16.

The image shows a screenshot of the InfoSphere Streams IDE. The window title is "InfoSphere Streams" and the active file is "*JMSSinkMain.spl". The code is written in SPL and defines a composite application named JMSSinkMain. It includes a use statement for the messaging library, a composite definition with a type section for PersonSchema, and a graph section. The graph contains two main steps: reading input data from a file and writing it to MQ using JMSSink. The code is as follows:

```
use com.ibm.streams.messaging.jms::*;

composite JMSSinkMain {
  type
  PersonSchema = int32 id,
                 rstring fname,
                 rstring lname,
                 int32 age,
                 rstring gender,
                 float32 score,
                 float64 total;

  graph

  //1. Read the input stream from a file
  stream<PersonSchema> inputData = FileSource() {
    param
      file      : "testdata.csv";
      format    : csv;
      initDelay : 1.0;
  }

  //2. Write to MQ using JMSSink without the error stream
  () as mySink = JMSSink ( inputData ){
    param
      //connectionDocument : default is ./etc/connections.xml;
      connection : "conn1";
      access     : "access1" ;
  }
}
```

Figure 8-16 JMSSink SPL application

The JMSSink SPL application reads a `testdata.csv` file using the `FileSource` operator and generates tuples of the type `PersonSchema`, which is sent to the `JMSSink` operator. The `JMSSink` operator converts these tuples to JMS messages and sends them to the JMS provider as configured through the connection and access parameters in conjunction with the connection specification document.

JMSSource SPL application

The sample JMSSource SPL application is shown in Figure 8-17.



```
InfoSphere Streams
*JMSSinkMain.spl
use com.ibm.streams.messaging.jms:*;

composite JMSSinkMain {
  type
  PersonSchema = int32 id,
                 rstring fname,
                 rstring lname,
                 int32 age,
                 rstring gender,
                 float32 score,
                 float64 total;

  graph

  //1. Read the input stream from a file
  stream<PersonSchema> inputData = FileSource() {
    param
      file      : "testdata.csv";
      format    : csv;
      initDelay : 1.0;
  }

  //2. Write to MQ using JMSSink without the error stream
  () as mySink = JMSSink ( inputData ){
    param
      //connectionDocument : default is ./etc/connections.xml;
      connection: "conn1";
      access: "access1" ;
  }
}
```

Figure 8-17 JMSSource SPL application

The JMSSource SPL application listens to the queue topic as configured through the connection and access parameters in conjunction with the connection specification document. It reads the JMS messages, converts them into tuples, and sends them downstream to a FileSink operator that writes this data to the persons.dat file.

The JMSSource and JMSSink SPLs must be modified so that the connection parameter refers to the connection_specification name and the access parameter refers to the access_specification name as mentioned in the connection document. Both the JMSSink and JMSSource applications are now ready to be compiled and run.

Figure 8-18 shows the instance graph for the JMSSink sample:

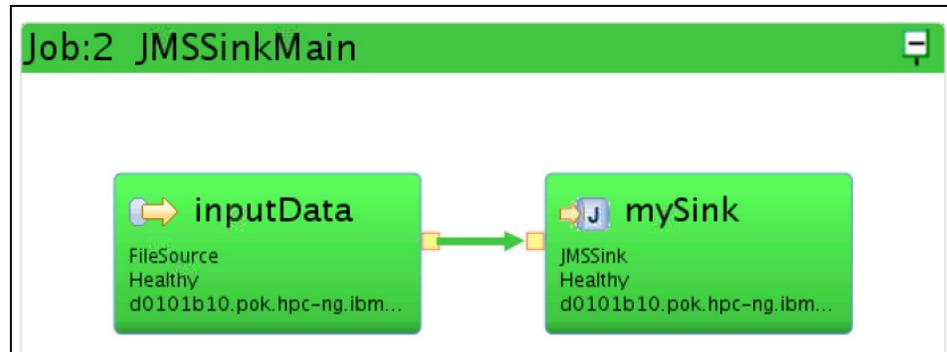


Figure 8-18 JMSSink instance graph

Figure 8-19 shows the instance graph for JMSSource sample:

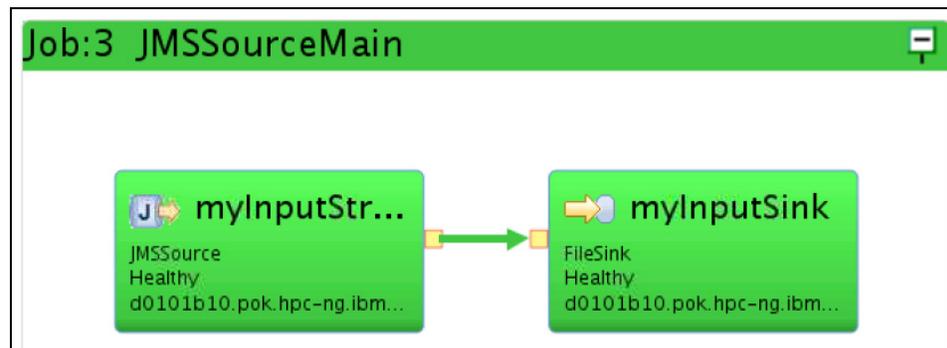


Figure 8-19 JMSSource instance graph

8.5.6 Verifying the results

The execution of the JMSSink operator can be verified by opening the console of the respective JMS provider and looking in the queue topic to which the connection was configured.

For verifying the results in Apache ActiveMQ, go to the admin console (<http://servername:8161/admin>) and click **Queues**. Select the queue to which the data was sent using the JMSSink operator. In our example, we verify results by going to this location:

<http://charak.in.ibm.com:8161/admin/>

An example of the resulting queue is depicted in Figure 8-20.

Browse MapQ								
Message ID ↑	Correlation ID	Persistence	Priority	Redelivered	Reply To	Timestamp	Type	Operations
ID:charak.in.ibm.com-48195-1378150967446-1:1:1:1:1		Persistent	4	false		2013-09-03 01:12:48:652 IST		Delete
ID:charak.in.ibm.com-48195-1378150967446-1:1:1:1:10		Persistent	4	false		2013-09-03 01:12:48:988 IST		Delete
ID:charak.in.ibm.com-48195-1378150967446-1:1:1:1:11		Persistent	4	false		2013-09-03 01:12:48:994 IST		Delete
ID:charak.in.ibm.com-48195-1378150967446-1:1:1:1:12		Persistent	4	false		2013-09-03 01:12:49:001 IST		Delete
ID:charak.in.ibm.com-48195-1378150967446-1:1:1:1:13		Persistent	4	false		2013-09-03 01:12:49:024 IST		Delete
ID:charak.in.ibm.com-48195-1378150967446-1:1:1:1:14		Persistent	4	false		2013-09-03 01:12:49:028 IST		Delete
ID:charak.in.ibm.com-48195-1378150967446-1:1:1:1:15		Persistent	4	false		2013-09-03 01:12:49:032 IST		Delete
ID:charak.in.ibm.com-48195-1378150967446-1:1:1:1:16		Persistent	4	false		2013-09-03 01:12:49:036 IST		Delete
ID:charak.in.ibm.com-48195-1378150967446-1:1:1:1:12		Persistent	4	false		2013-09-03 01:12:48:732 IST		Delete
ID:charak.in.ibm.com-48195-1378150967446-1:1:1:1:13		Persistent	4	false		2013-09-03 01:12:48:737 IST		Delete
ID:charak.in.ibm.com-48195-1378150967446-1:1:1:1:14		Persistent	4	false		2013-09-03 01:12:48:748 IST		Delete
ID:charak.in.ibm.com-48195-1378150967446-1:1:1:1:15		Persistent	4	false		2013-09-03 01:12:48:753 IST		Delete
ID:charak.in.ibm.com-48195-1378150967446-1:1:1:1:16		Persistent	4	false		2013-09-03 01:12:48:757 IST		Delete
ID:charak.in.ibm.com-48195-1378150967446-1:1:1:1:17		Persistent	4	false		2013-09-03 01:12:48:761 IST		Delete
ID:charak.in.ibm.com-48195-1378150967446-1:1:1:1:18		Persistent	4	false		2013-09-03 01:12:48:765 IST		Delete
ID:charak.in.ibm.com-48195-1378150967446-1:1:1:1:19		Persistent	4	false		2013-09-03 01:12:48:982 IST		Delete

Figure 8-20 Queue for JMSSink operator

By clicking any message, you can see the content of the message under Message details, as shown in Figure 8-21.

Headers ↑	
Correlation ID	
Destination	queue://MapQ
Expiration	0
Group	
Message ID	ID:charak.in.ibm.com-48195-1378150967446-1:1:1:1:1
Persistence	Persistent
Priority	4
Redelivered	false
Reply To	
Sequence	0
Timestamp	2013-09-03 01:12:48:652 IST
Type	

Message Actions

Delete

Copy

Move

-- Please select --

Message Details

```
{fname=Tasos, gender=M, score=78.3, total=349.3, lname=Kementsietsidis, id=1, age=35}
```

Figure 8-21 Message details

For verifying the results in WebSphere MQ, go to the MQ Explorer. Select **Queues**, and then select the queue to which the operator was configured. In our example, the queue is depicted in Figure 8-22.

Queue name	Queue type	Open in	Open o	Current queu
TestQueue	Local	0	0	16
TestTopicQueue	Local	0	0	0

Figure 8-22 Queue verification

Right-click on a queue name, and select **Browse** to see the messages. Our example is shown in Figure 8-23.

Queue Manager Name: TestQueueManager
 Queue Name: TestQueue

Position	Put date/time	User identi	Put application name	Format	Data length	Message data
1	Sep 24, 2013 : mqm		WebSphere MQ Client	MQSTR	408	<map><elt name="a
2	Sep 24, 2013 : mqm		WebSphere MQ Client	MQSTR	398	<map><elt name="a
3	Sep 24, 2013 : mqm		WebSphere MQ Client	MQSTR	395	<map><elt name="a
4	Sep 24, 2013 : mqm		WebSphere MQ Client	MQSTR	396	<map><elt name="a
5	Sep 24, 2013 : mqm		WebSphere MQ Client	MQSTR	395	<map><elt name="a
6	Sep 24, 2013 : mqm		WebSphere MQ Client	MQSTR	400	<map><elt name="a
7	Sep 24, 2013 : mqm		WebSphere MQ Client	MQSTR	403	<map><elt name="a
8	Sep 24, 2013 : mqm		WebSphere MQ Client	MQSTR	400	<map><elt name="a
9	Sep 24, 2013 : mqm		WebSphere MQ Client	MQSTR	402	<map><elt name="a
10	Sep 24, 2013 : mam		WebSphere MQ Client	MOSTR	403	<map><elt name="a

Scheme: Standard for Messages
 Last updated: 11:59:27 (16 items)

Figure 8-23 Messages

Double-click a message to see more information about the message. Our example is shown in Figure 8-24.

The screenshot shows a message details window with a sidebar on the left containing the following menu items: General, Report, Context, Identifiers, Segmentation, Named Properties, and Data (which is highlighted in blue). The main area is titled "Data" and contains the following fields:

- Data length: 408
- Format: MQSTR
- Coded character set identifier: 1208
- Encoding: 273
- Message data: `<map><elt name="age" dt='i4'>35</elt>`
- Message data bytes: A hex dump showing the byte representation of the XML data, with each line containing a hex value and its corresponding ASCII character.

```
61 6D 65 3D 22 |<map><elt name="|
27 3E 33 35 3C |age" dt='i4'>35<|
61 6D 65 3D 22 |/<elt><elt name="|
6E 74 73 69 65 |lname">Kementsie|
3E 3C 65 6C 74 |tsidis</elt><elt|
65 72 22 3E 4D | name="gender">M|
6E 61 6D 65 3D |</elt><elt name=|
27 72 38 27 3E |"total" dt='r8'>|
3C 65 6C 74 20 |349.3</elt><elt |
22 20 64 74 3D |name="score" dt=|
65 66 74 20 3E |</elt></map></|
```

Figure 8-24 Message details

The execution of JMSSource operator can be verified by looking in the `persons.dat` file.



XML, XMLParse, XPath, and xquery

In this chapter, we detail InfoSphere Streams, and its support for processing of XML-formatted data sources. Version 3.0 of Streams adds a native XML data type, an XMLParse operator, and several XML-related functions. By *native*, we mean that these objects are part of the SPL language or the standard toolkit in Streams, and are not made available using any additional toolkit or accelerator. These native items are built-in, immediately available, and highly performant objects within Streams. The phrase *data source* is carefully chosen. As a real-time analytics platform, Streams can ingest and then process XML-formatted data. There is no inherent ability and no single operator for Streams to originate or newly format any data in XML, and then output it.

This chapter uses two primary examples to explain XML processing within Streams. The first example uses a flat, single-tier XML-formatted document, and serves as the introduction to the topic. The second example uses a multitiered document, which demonstrates processing of most XML-formatted documents you might encounter. Because of the hierarchical nature of XML-formatted data, processing XML with Streams might lead you to define and then use variables of intermediate and higher complexity. For that reason, the second example is created over four iterations, allowing you to build skills in this area safely and carefully.

9.1 Scenario 1: Flat, single-tier XML, XMLParse

The first example XML-formatted input data file we work with is listed in Example 9-1. This scenario is meant to model a fictitious configuration file for a piece of software, and is single-tiered (no *nested elements* or *sub-elements*). See the code review after Example 9-1.

Example 9-1 First scenario, XML01_Simple.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:MyServerConfig
  xmlns:tns="http://www.example.org/MyXSD"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.example.org/MyXSD MyXSD.xsd">
  <tns:ServerIP enable="true">192.168.1.14</tns:ServerIP>
  <tns:LogLevel>normal</tns:LogLevel>
  <tns:SysAdmin>junkmail@w3.blahblah.com</tns:SysAdmin>
</tns:MyServerConfig>
```

Consider the following information about Example 9-1:

- ▶ XML lines that begin with `<?` and end with `?>` characters are called (*XML processing instructions*). This line identifies this file to be of type XML, and identifies its XML version number and other declaratives. Generally, in the extraction of data from an XML-formatted file using Streams, lines of this type are ignored.
- ▶ Line 2 in this example is relatively long, and displays over several physical lines. The line includes references to one or more XML *namespaces*. In this context we state the following information:
 - XML is extensible. With XML, you can declare new data types, new data type rules and more. XML namespaces can be thought of as libraries (library addresses), or the ability to make remote references to this type of extensible programming. Generally as we seek to extract data from any (local) document, we can ignore these new data types or new data type rules; XML data files are inherently textual, and that is all we need to know to extract data.
 - XML can be validated. The references the `tns` namespace. In real-world XML processing, we can use or ignore these namespace references, or perform XML documentation validation. Unless we are “validating” documents, we can generally choose to ignore the single namespace or set of namespaces.
 - Line 2 also declares the name of the *root element* in this XML document; in the example it is named `MyServerConfig`. Any data we want to extract

from this XML-formatted document will be relative to the MyServerConfig element. (An XML root element is most similar to the root directory of any file system.)

- ▶ Line 3 defines an *element* to this XML document named ServerIP, with its associated value of 192.168.1.14. This same element has an *attribute* named enable; the attribute value is set to true.
- ▶ Lines 4 and 5 define two new elements, named LogLevel and SysAdmin with the associated values of normal and junkmail@w3.blahblah.com. Neither of these elements has any attributes.

Example 9-2 displays what we want to output from the XML-formatted input data file in Example 9-1 on page 206. Really we only want to parse the input XML, place the values of interest into distinct Streams variable structures, and then continue our real-world analytic processing. The examples we use in this chapter merely parse and print; we print as a proof of the results we want.

Example 9-2 Sample output we want to create, XML01_Simple.out.txt

```
192.168.1.14,true,normal,junkmail@w3.blahblah.com
```

The example Streams application that can parse Example 9-1 on page 206 may be painted (created with drag-and-drop operations) using the Streams Studio Graphical Editor, and is displayed in Figure 9-1 on page 208. See the code review after Figure 9-1 on page 208.

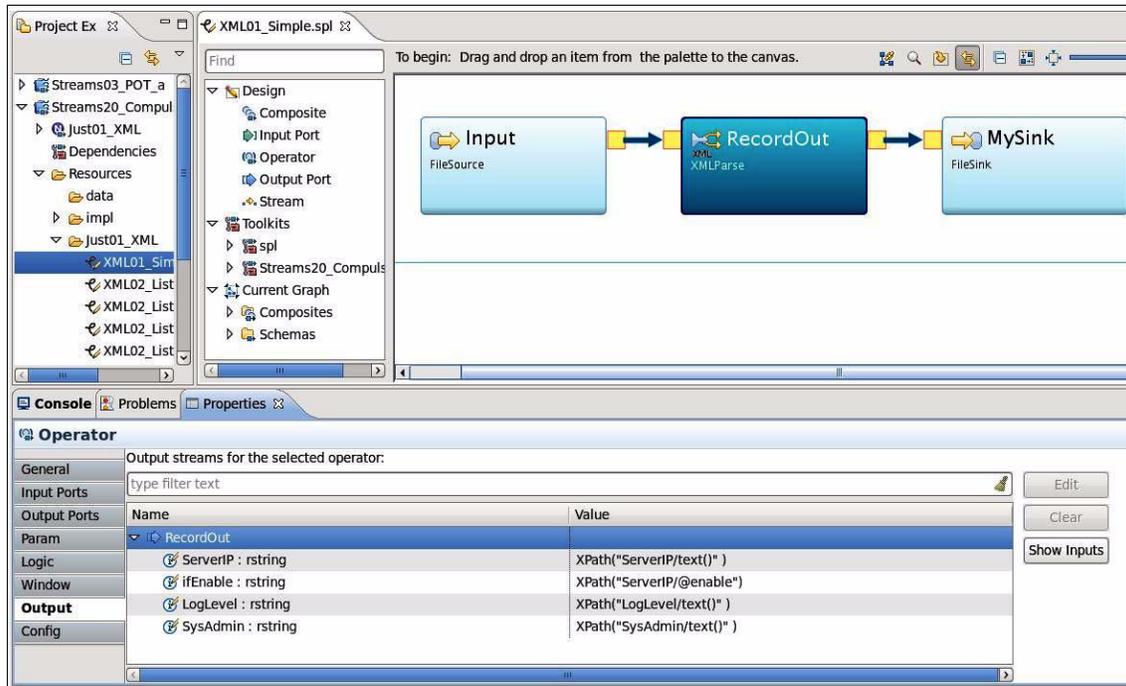


Figure 9-1 SPL Graphical Editor view of XML01_Simple.spl

At this point in the book, you can probably paint, compile, run, examine the output of a standard Streams file read, do something, and then file-write this style of Streams application. You can also read and understand a Streams application source code listing. Consider the following information about Figure 9-1:

- ▶ The canvas of the Streams Graphical Editor view displays three operators: FileSource (input, left), FileSink (output, right), and in the middle, XMLParse. *XMLParse* is that operator that allows us to read and retrieve the values of XML elements, XML attributes, and related information. XMLParse reads, it does not format or write. XMLParse is the primary object we cover in this chapter.
- ▶ The properties view, and more specifically its output tab, displays several XPath expressions. *XPath* is a topic that is from the public domain, and is a larger topic than Streams; XPath is the means by which we identify a given XML element or XML attribute, and how we want to extract its value.

Figure 9-2 on page 209 displays the source code listing for the Streams application that parses Example 9-1 on page 206. See Appendix A, “Additional material” on page 527 for instructions for how to access downloadable (text, editable) copy of this and all source code and input data files used in this chapter.

```

composite XML01_Simple {
  graph
    stream<rstring InputLine> Input = FileSource() {
      param file : "/POT/MyFiles/20_XML/XML01_Simple.in.xml";
      format : line;
    }

    stream<rstring ServerIP, rstring ifEnable,
      rstring LogLevel, rstring SysAdmin>
      RecordOut = XMLParse(Input) {
      param
        trigger      : "/MyServerConfig";
        parsing      : permissive;
        ignoreNamespaces : true;
      output RecordOut :
        ServerIP = XPath("ServerIP/text()" ),
        ifEnable = XPath("ServerIP/@enable" ),
        LogLevel = XPath("LogLevel/text()" ),
        SysAdmin = XPath("SysAdmin/text()" );
    }

    () as MySink = FileSink(RecordOut) {
      param
        file      : "/POT/MyFiles/20_XML/XML01_Simple.out.txt";
        format    : csv;
        quoteStrings : false;
    }
  }
}

```

Figure 9-2 Source code listing for XML01_Simple.spl

Consider the following information about Figure 9-2:

- ▶ The first operator, named Input, is a standard Streams input file reader, a Streams FileSource operator. Note that the XML-formatted input data file is read by line, and is passed line by line using an rstring data type to XMLParse. The data we are reading is from Example 9-1 on page 206.

Note: In this chapter, you see XML-formatted input data files read by using two basic means:

- ▶ **XMLParse:** In this first example, Streams is reading an ASCII text file line by line, and parsing that file using the Streams operator named XMLParse. Input data is passed into XMLParse by the Streams data type called rstring. In this manner, XML-formatted data can be parsed as it is received, piece by piece, line by line. You can parse one document or several, and output results immediately as they are observed.
- ▶ **xquery:** In 9.4, “The xquery() function, including filter” on page 236, XML data is parsed using a built-in Streams function named xquery(). The xquery() function must receive *whole* XML-formatted documents; processing begins as the whole XML-formatted document is sent to the xquery() function at invocation time. As a whole document, this (input data) is not sent using rstring, but rather with the Streams data type xml. The xquery() function cannot process an XML-formatted snippet (subset) of data; xquery() must always receive whole XML documents.

When do you use XMLParse versus xquery()?

- ▶ Generally, XMLParse is considered more performant than xquery().
 - ▶ xquery() should gain total performance when you need to run multiple parses, or multiple extractions, from a single XML document. (XMLParse might require multiple operators, depending on the exact requirements.) As a guideline, view the task by the total amount of I/O that is processed, once or multiple times.
 - ▶ In this release of Streams, XMLParse effectively supports the XPath expressions for text() and @attr. The xquery() function supports xquery language, which is a superset of XPath, and supports more functionality.
 - ▶ Streams XMLParse uses an embedded Simple API for XML (SAX) processor; xquery() uses the **xqilla** open source XQuery processing facility. To most successfully query xquery() topics and syntax, use a search engine query that makes reference to xqilla.
 - ▶ When do you most commonly receive XML-formatted data as whole documents? Perhaps when reading blobs (binary large objects) from a database, or any means where the XML-formatted data arrives as a complete document.
- ▶ The last operator, named MySink is a standard Streams output file writer, a Streams FileSink operator. We choose to write by CSV file so that we can pass a Streams tuple to it; tuple is the Streams object that allows bundling of

variables and other data elements. When you want to write a tuple using a Streams file writer, you commonly write that file as CSV. The data we are writing is displayed in Example 9-2 on page 207.

- ▶ Next, you can use XMLParse, the Streams operator for XML-formatted data ingest, for XML parsing.
 - This operator, as it is displayed, receives a series of single input values, in the form of an rstring. For our use, it outputs four values, which are the 4 values we want d to extract from the sample XML. This operator outputs when it receives a full element as specified by the value of the trigger parameter.
 - Three parameters (param) are listed in Figure 9-2 on page 209:
 - The trigger parameter: Specifies the *reference element level* at which we want to investigate the XML document. This example lists MyServerConfig, which is the root element of the sample XML document. The sample XML document has only a one-tier XML format, meaning; all elements and attributes are available at the root level (directly) of this XML document.
 - The parsing parameter: Specifies whether this Streams operator should allow nearly well-formed XML, or allow only perfectly validated XML-formatted data. Generally we use permissive throughout.

Note: To be more precise; XML can be *well-formed, but not valid*. Validity is determined by an XML schema. The permissive parameter specifies how the XMLParse operator behaves when it receives an XML element or attribute that cannot convert to the desired SPL variable.

- The ignoreNamespaces parameter: Allows this Streams operator to act as though no namespace prefixes are relevant or present. Generally this reduces the bulk and complexity of XML XPath expressions we need to provide as input to this operator.
- The output clause of the XMLParse operator offers this information:
 - An XML element within this XML document named /MyServerConfig/ServerIP has a value, represented by the absolute XPath expression of /MyServerConfig/ServerIP/text(). An attribute named enable to this same element, is represented by the absolute XPath expression /MyServerConfig/ServerIP/@enable. Recall that we do not use absolute XPath expressions inside the XMLParse operator. After the trigger parameter, which is absolute, all XPath expressions are then relative.

Note: How do you create or know the values for the XPath expressions referenced here?

Essentially, the XPath expressions supported by XMLParse end with `text()` or `@attr`, which is the attribute name, if an attribute is present. As a function, `text()` returns the value of the XML element to which it refers; `@attr` is a reference to the attribute value that it identifies.

The prefix to the XPath expression is *relative* to the reference element level, and it *must be relative*. That is, the XPath expression *must not be absolute*. If the XPath expression is absolute, you will get a Streams application compiler error. The trigger parameter to the XMLParse operator must be absolute, or you will get a Streams application compiler error.

Examples are as follows:

- ▶ Given an element with the absolute reference of `/root/foo/bar/text()`, and an XMLParse operator trigger value of `/root`, this element's value has the XPath expression of, `foo/bar/text()`. The XPath expression for an attribute of this same element, attribute name `MyAttribute`, would have an XPath expression of, `foo/bar/@MyAttribute`.
- ▶ Given an element of `/root/foo/bar/text()`, and a trigger of `/root/foo`, the XPath expression to `/root/foo/bar` is `bar/text()`. This same element's attribute XPath expression would be `bar/@attr`, where `attr` is the name of the attribute.

- Two more elements are output in this example: named `LogLevel` and `SysAdmin`. These elements offer no new information to this example.

9.2 Scenario 2: Multitiered (list data)

This second XML parsing scenario has a more complex XML-formatted input data file, and represents the majority of the XML parsing cases you might encounter. Generally, the XMLParse operator is configured in exactly the same manner as the first scenario; the primary change in this example is reflected in the Streams variables receiving the output of XMLParse. XML-formatted data is hierarchical in nature, and the related Streams variables generally reflect this. Because these more complex Streams variables might be unfamiliar to you, we build this second example over four iterative steps.

The second XML-formatted input data file we work with in this second scenario is listed in Example 9-3. This example models a fictitious retail point of sale terminal, and is multitiered (*containing a list of nested elements or sub-elements*). See the code review after Example 9-3.

Example 9-3 Second scenario, XML02_List.in.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:RegisterOrders RegID="Register-100"
  xmlns:tns="http://www.example.org/MyXSD"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    "http://www.example.org/MyXSD MyXSD.xsd ">
  <tns:ATransaction TxID="101" TxType="sale">
    <ItemName count="2">Helmet      </ItemName>
    <ItemName count="4">Cap        </ItemName>
  </tns:ATransaction>
  <tns:ATransaction TxID="102" TxType="sale">
    <ItemName count="1">Jersey     </ItemName>
  </tns:ATransaction>
  <tns:ATransaction TxID="103" TxType="return">
    <ItemName count="3">Swim Fin  </ItemName>
    <ItemName count="6">Swim Mask </ItemName>
  </tns:ATransaction>
  <tns:ATransaction TxID="104" TxType="sale" >
    <ItemName count="5">Football  </ItemName>
    <ItemName count="7">Baseball  </ItemName>
    <ItemName count="9">Frisbee   </ItemName>
  </tns:ATransaction>
</tns:RegisterOrders>
```

Consider the following information about Example 9-3:

- ▶ By means of example, the absolute XPath expression to the primary data element is `/RegisterOrders/ATransaction/ItemName/text()`. By *primary data element*, we mean the element that we care most about. If we trigger on `/RegisterOrders/ATransaction`, the XPath expression becomes `ItemName/text()`.
- ▶ This example has attributes at all three element levels. Later, we filter on the `/RegisterOrders/ATransaction/@TxType` attribute.

Example 9-4 displays the output. Although we build the second example over four iterations, this is generally the data we will output.

Example 9-4 Output from second example, generally the same over four iterations.

```
101,sale ,Helmet      ,2
101,sale ,Cap         ,4
102,sale ,Jersey      ,1
103,return,Swim Fin   ,3
103,return,Swim Mask ,6
104,sale ,Football    ,5
104,sale ,Baseball    ,7
104,sale ,Frisbee     ,9
```

Figure 9-3 on page 215 through Figure 9-6 on page 217 display the source code listing for the Streams application that parses the input data shown in Example 9-3 on page 213. See the code review after Figure 9-6 on page 217.

```
*XML02_List_a.spl ⌵

namespace Just01_XML;

composite XML02_List_a {

type
  MyTxList = tuple
  <
    rstring      TxID      ,
    rstring      TxType    ,
    list<rstring> ItemName,
    list<rstring> ItemCount
  >;
  MyOneTx = tuple
  <
    rstring      TxID      ,
    rstring      TxType    ,
    rstring      ItemName,
    rstring      ItemCount
  >;

graph
  stream<rstring InputLine> Input = FileSource() {
    param file : "/POT/MyFiles/20_XML/XML02_List.in.xml";
    format : line;
  }
}
```

Figure 9-3 Second scenario, XML02_List_a.spl (image 1 of 4)

```
*XML02_List_a.spl ⌵

stream<MyTxList> RecordOut = XMLParse(Input) {
  param
    trigger      : "/RegisterOrders/ATransaction";
    parsing      : permissive;
    ignoreNamespaces : true;
  output RecordOut :
    TxID      = XPath("@TxID"),
    TxType    = XPath("@TxType"),
    ItemName  = XPathList("ItemName/text()"),
    ItemCount = XPathList("ItemName/@count");
}

//
// The above gives us,
//
// 101,sale,["Helmet      ", "Cap          "],["2", "4"]
// 102,sale,["Jersey      "],["1"]
// 103,return,["Swim Fin   ", "Swim Mask   "],["3", "6"]
// 104,sale,["Football    ", "Baseball    ", "Frisbee     "],["5", "7", "9"]
//
// So to 'flatten' this, we run the operator below
//
```

Figure 9-4 Second scenario, XML02_List_a.spl (image 2 of 4)

```

*XML02_List_a.spl
stream <MyOneTx> MyCustom as MyOut =
    Custom (RecordOut as MyIn) {

    logic state :
    {
        mutable MyOneTx l_MyTx;
    }

    onTuple MyIn :
    {
        l_MyTx.TxID = MyIn.TxID ;
        l_MyTx.TxType = MyIn.TxType;
        //
        mutable int32 l_cnr = 0 ;
        mutable int32 l_sizeof = size(MyIn.ItemName);
        //
        while (l_cnr < l_sizeof)
        {
            l_MyTx.ItemName = MyIn.ItemName [l_cnr];
            l_MyTx.ItemCount = MyIn.ItemCount[l_cnr];
            //
            ++l_cnr;
            //
            submit (l_MyTx, MyOut);
        }
    }
}

```

Figure 9-5 Second scenario, XML02_List_a.spl (image 3 of 4)

```

() as MySink = FileSink(MyCustom) {
    param
        file      : "/POT/MyFiles/20_XML/XML02_List.outa.txt";
        format    : csv;
        quoteStrings : false;
}

```

Figure 9-6 Second scenario, XML02_List_a.spl (image 4 of 4)

Consider the following information about Figure 9-3 on page 215:

- ▶ We placed this SPL application in a directory under the Streams project root directory, hence the reference to a Streams namespace. This was not required. We did this to better organize this SPL source code from all others.

Note: Streams namespaces are entirely unrelated to XML namespaces. It is merely coincidence that these two environments use the same keyword, namespace.

- ▶ The first operator, FileSource, reads the XML-formatted input data file by line, and passes its value to the next operator by an rstring data type. No new information is offered here from the first example displayed in Figure 9-2 on page 209.
- ▶ Two tuples are defined:
 - MyOneTx is a simple tuple offering the four values we want to extract from the XML-formatted input data file. This is the Streams variable we use to pass values to the final operator, a FileSink.
 - MyTxList is the tuple that the XMLParse operator will populate. This tuple has two rstring variables, and two list-of-rstring variables. Consider the following information about lists:
 - A *list* is one of three data types in Streams, similar to what other environments might refer to as *arrays*. Streams refers to these three data types as *collections* and includes *list*, *set*, and *map*.
 - Although each data type of list, map, and set have their intended use, here we use and describe only the list type. Streams defines a list as a random access, zero indexed (sub-scripted elements of a list begin counting at zero, not one), unordered collection of entities that allows duplicates.
 - Although a list can contain only one data type by itself (a list of a single rstring variable, a list of a single int32 variable, and others), a list can contain a list of tuples. If you need a list to contain, for example, an rstring and two (count) int32 variables, create a tuple of the definition, and then create a list of that tuple type.
 - In our example, we create two lists, each of type rstring. These lists will contain the ItemName and ItemCount from the supposed retail register list of transactions.

Note: We are building this second example over four iterations. The first three iterations all have various issues. Only the fourth iteration is preferred in its use.

Why not then just display the fourth iteration?

First, you might not be accustomed to using variables that have lists with embedded lists, with embedded lists and tuples. Second, we want to clearly demonstrate why you should use the fourth iteration, and not perhaps, the apparently easier first through third iterations.

The fourth iteration is more advanced, but is also the best practice.

Consider the following information about Figure 9-4 on page 216:

- ▶ This is the XMLParse operator. As we have configured it, this operator receives an rstring, and outputs a tuple with four variables.
- ▶ The trigger to this XMLParse operator is `/RegisterOrders/ATransaction`. The sample XML-formatted input data file in Example 9-3 on page 213 has four `ATransaction` elements present, therefore, the XMLParse operator will output four tuples.
- ▶ Other parameters to this operator include `parsing = permissive`, and `ignoreNamespace = true`. The XML formatted data passed to this operator should be fully formed (permissive), and we need not create the XPath expressions with concern for XML namespaces (`ignoreNamespaces = true`). Unless you have a specific set of needs, we find these settings work well in most cases.
- ▶ Upon receipt of a whole trigger element (`/RegisterOrders/ATransaction`), this operator will output the expected tuple:
 - `TxID = XPath("@TxID")` should not be unfamiliar at this point. There is a single XML attribute at this trigger level of the input d XML-formatted data. This is also true for `TxType`.
 - `ItemName = XPathList("ItemName/text()")` is new to us. `ItemName` is a variable of type `list/rstring`. Because a trigger, `ATransaction`, might have multiple `ItemNames`, we need the following information:
 - We must use `XPathList`, not `XPath`. `XPath` returns a single value where `XPathList` returns a list; otherwise, `XPath` and `XPathList` should be considered identical in function.
 - The receiving variable (`ItemName`) must be of type `list`.
 - `ItemCount` is the same as `ItemName`, except we are retrieving a list of attributes, instead of a list of elements and their associated values.

- The sample data output by this operator is displayed in Figure 9-4 on page 216 as comments. (See also Figure 9-7 on page 221 for the final results.)

Note: Having the two separate lists as displayed in Figure 9-4 is the first issue we want to avoid.

This example code works properly because every trigger element in `ATransaction` has both an `ItemName` and a given attribute (`ItemCount`); in this case, both lists will have four values.

If one of the four `ATransaction` elements was missing the given attribute (possible with XML), then you would receive one list with four values and another list with three values, and, no way to align these lists. Because a missing given attribute might be in the middle of the list, you cannot adequately determine which attribute belongs to which element.

We begin to address this issue in the next iteration of this example.

Given the variable output from the `XMLParse` operator in Figure 9-4 on page 216, we might want to flatten these values, produce something more similar to a database two-dimensional record. (You are about to start observing more complex Streams variables, lists with embedded lists, with even more embedded lists.)

Consider the following information about Figure 9-5 on page 217:

- ▶ This is a Streams custom operator. *Custom operators* are handy because they can accept one or more input streams, output one or more streams, are fully programmable, and more. In this case, we use the custom operator merely to loop through the lists of XML element values and attributes, and produce a flattened record.
- ▶ The bulk of the work in this operator is done inside the `onTuple` block:
 - As configured, this block will execute on the receipt of every `ATransaction`. Because we have four of these input tuples, we might expect that this operator outputs four tuples of its own. But, the `submit` function is embedded inside a `while` loop, which counts every element of the `ItemName` list; therefore, we will output (`submit` for output) at least four times, once for every `ItemName` for each of those four times (four `ATransactions`).
 - The `size` function returns a count of the number of elements in the list.
 - The remainder of this operator is mostly math to count through the list and so on.

Note: Find the following line in Figure 9-5:

```
l_MyTx.ItemCount = MyIn.ItemCount[l_cntr];
```

It is the line we do not recommend in context. This line is dependant on every ATransaction having both an ItemName and an associated attribute for ItemCount. If you have a smaller number of ItemCounts, then the index into this list becomes out of bounds, out of range. Not only will you experience a runtime error, bad data will be output: attributes (ItemCount) for ItemNames to which they do not belong.

Figure 9-6 on page 217 displays the FileSink, which offers no new techniques. At this point in the iterative examples, we stop highlighting the FileSource and FileSink operators.

Figure 9-7 displays the correct output from this Streams application.



Figure 9-7 Sample, and correct output

9.2.1 Second iteration, XMLParse

We want to correct several items in the first iteration of this example, displayed in Figure 9-3 on page 215 through Figure 9-6 on page 217:

- ▶ We want to better handle missing ItemCount attributes, for example, XML-formatted data that is uneven (sometimes referred to as *jagged*).
- ▶ We want to output XML elements and attributes above the current trigger of /RegisterOrders/ATransactions. For instance, XML-formatted data that is multitiered.
- ▶ Each of the next three iterations to this second example use the same XML-formatted input data file, as shown in Example 9-3 on page 213. The next iteration to the second example produces the same output file as in Figure 9-7.

Example 9-5 displays the second iteration to this second example. See the code review after Example 9-5.

Example 9-5 Second scenario, second iteration, XML02_List_b.spl

```
namespace Just01_XML;

composite XML02_List_b {

type
  MyItem = tuple
  <
    rstring      ItemName,
    rstring      ItemCount
  >;
  MyTxList = tuple
  <
    rstring      TxID      ,
    rstring      TxType   ,
    list<MyItem> ItemList
  >;
  MyOneTx = tuple
  <
    rstring      TxID      ,
    rstring      TxType   ,
    rstring      ItemName,
    rstring      ItemCount
  >;

graph

  stream<rstring InputLine> Input = FileSource() {
    param
      file      : "/POT/MyFiles/20_XML/XML02_List.in.xml";
      format    : line;
  }

  stream<MyTxList> RecordOut = XMLParse(Input) {
    param
      trigger      : "/RegisterOrders/ATransaction";
      parsing      : permissive;
      ignoreNamespaces : true;
    output RecordOut :
      TxID        = XPath("@TxID"  ),
```

```

TxType    = XPath("@TxType" ),
ItemList  = XPathList
    ( "ItemName",
      {
        ItemName = XPath("text()"),
        ItemCount = XPath("@count")
      }
    )
}

stream <MyOneTx> MyCustom as MyOut =
  Custom (RecordOut as MyIn) {
    logic state :
    {
      mutable MyOneTx l_MyTx;
    }
    onTuple MyIn :
    {
      l_MyTx.TxID    = MyIn.TxID ;
      l_MyTx.TxType = MyIn.TxType;
      //
      mutable int32 l_cnr    = 0 ;
      mutable int32 l_sizeof = size(MyIn.ItemList);
      //
      while (l_cnr < l_sizeof)
      {
        l_MyTx.ItemName = MyIn.ItemList[l_cnr].ItemName ;
        l_MyTx.ItemCount = MyIn.ItemList[l_cnr].ItemCount;
        //
        ++l_cnr;
        //
        submit (l_MyTx, MyOut);
      }
    }
  }

() as MySink = FileSink(MyCustom) {
  param
  file      : "/POT/MyFiles/20_XML/XML02_List.outb.txt";
  format    : csv;
  quoteStrings : false;
}

}

```

Consider the following information about Example 9-5 on page 222:

- ▶ There are now three tuples defined at the head of Example 9-5 on page 222:
 - MyItem is new. We need MyItem because we want a single list of both ItemName and ItemCount, but lists support only variables of the same type. So, we create a tuple of the variable we want (MyItem), and then list on that.
 - MyTxList has changed from Figure 9-3 on page 215. This tuple now contains a single list of tuples, versus two lists of rstrings.
 - MyOneTx has not changed and offers no new functionality.
- ▶ The FileSource operator has not changed.
- ▶ The XMLParse operator has changed in the area of its assignments, specifically in the area of XPath and XPathList:
 - ItemList is populated with a single XPathList. The previous iteration of this example used two XPathLists and two lists.
 - The specific syntax to accomplish this assignment are parenthesis, curly braces, their placement, and so on. This is what SPL refers to as *tuple literal format*.

Note: This is what we wanted to fix from second example, iteration one. By populating the list in one call, one XPathList invocation, we now handle missing ItemCount attributes. (Missing values are represented by an empty string, and do not cause a runtime error as before.)

- ▶ The custom operator is not new. The custom operator has been updated slightly to handle the improved variables.
- ▶ The FileSink operator has not changed.

9.2.2 Third iteration, XMLParse

Nothing is functionally or tactically wrong with Example 9-5 on page 222. It is, however, limited to one tier of XML element or attributes in the XML-formatted input data. In Example 9-6 on page 225, we begin to access multiple levels in the same XML document, using the XMLParse operator. This involves raising the XML element level of the trigger value, and creating and using more capable user-defined Streams variables.

See the code review after Example 9-6 on page 225.

```
namespace Just01_XML;

composite XML02_List_c {

type
  MyItem = tuple
  <
    rstring      ItemName,
    rstring      ItemCount
  >;
  MyTxList = tuple
  <
    rstring      TxID      ,
    rstring      TxType    ,
    list<MyItem> ItemList
  >;
  MyOneTxWithRegID = tuple
  <
    rstring      RegID     ,
    rstring      TxID      ,
    rstring      TxType    ,
    rstring      ItemName,
    rstring      ItemCount
  >;

graph

stream<rstring InputLine> Input = FileSource() {
  param
    file   : "/POT/MyFiles/20_XML/XML02_List.in.xml";
    format : line;
}

stream<MyTxList> RecordOut1 = XMLParse(Input) {
  param
    trigger      : "/RegisterOrders/ATransaction";
    parsing      : permissive;
    ignoreNamespaces : true;
  output RecordOut1 :
    TxID      = XPath("@TxID" ),
    TxType    = XPath("@TxType" ),
    ItemList  = XPathList
```

```

        ("ItemName",
        {
            ItemName = XPath("text()"),
            ItemCount = XPath("@count")
        }
    );
}

stream<rstring RegID> RecordOut2 = XMLParse(Input) {
    param
        trigger          : "/RegisterOrders";
        parsing          : permissive;
        ignoreNamespaces : true;
    output RecordOut2 :
        RegID           = XPath("@RegID");
}

stream <MyOneTxWithRegID> MyCustom as MyOut =
    Custom (RecordOut1 as MyIn1 ; RecordOut2 as MyIn2) {
    logic state :
    {
        mutable MyOneTxWithRegID l_MyTx;
    }
    onTuple MyIn2 :
    {
        l_MyTx.RegID = MyIn2.RegID;
    }
    onTuple MyIn1 :
    {
        l_MyTx.TxID = MyIn1.TxID ;
        l_MyTx.TxType = MyIn1.TxType;
        //
        mutable int32 l_cnr = 0 ;
        mutable int32 l_sizeof = size(MyIn1.ItemList);
        //
        while (l_cnr < l_sizeof)
        {
            l_MyTx.ItemName = MyIn1.ItemList[l_cnr].ItemName ;
            l_MyTx.ItemCount = MyIn1.ItemList[l_cnr].ItemCount;
            //
            ++l_cnr;
            //
            submit (l_MyTx, MyOut);
        }
    }
}
}

```

```

() as MySink = FileSink(MyCustom) {
  param
    file      : "/POT/MyFiles/20_XML/XML02_List.outc.txt";
    format    : csv;
    quoteStrings : false;
  }
}

```

Consider the following information about Example 9-6 on page 225:

- ▶ The tuple definitions offer no new information. A new variable has been added to carry RegId, which is at a higher level in the XML document than we have processed before.
- ▶ The FileSource operator is not new.
- ▶ Now there are two XMLParse operators:
 - RecordOut1 is nearly identical to RecordOut from Example 9-5 on page 222.
 - RecordOut2 is entirely new:
 - This operator has a trigger of /RegisterOrders, the root level element to the XML document. This operator is constructed to capture the RegId element.
 - RegId is an attribute to RegisterOrders, as denoted by a "/RegisterOrders" trigger and an "@RegId" XPath expression.
 - The custom operator has been changed to accept two input Streams, RecordOut1 and RecordOut2, and two XMLParse operators:
 - Essentially this operator has one output tuple namely, I_MyTx, and places values in its various variables using the onTuple MyIn2 and onTuple MyIn1 event blocks. In effect, we use the two inputs to this operator to join or merge two input streams.
 - The remainder of this operator offers no new techniques.

Note: Using the custom operator here as shown is not considered best practice. (We are effectively parsing the same XML document twice.)

Only during application start up, it may happen that the processing element (in this context, processing element equals a Linux process) for RecordOut2 may still be initializing, and not outputting tuples, while the processing elements for FileSource and RecordOut1 are. This is a phenomena during application start only. You could put an `initDelay` attribute to FileSource and make the problem go away, but that is also not our first choice. (Calls to input ports are, by nature, asynchronous, and you cannot rely on starting things in any specific order.)

The fourth iteration to this design is best practice, and is the one you should implement, should you need to output multiple levels of elements from an XML-formatted data source.

Why show this third iteration example then?

First, we want you to be aware of this phenomenon so that you may avoid it. Second, custom operators with two or more inputs are hugely functional and fun. Third, the fourth iteration is challenging, if you are newer to Streams and do not have these three initial iterations to observe.

- ▶ And the FileSink offers no new techniques.

9.2.3 Fourth (final) iteration, XMLParse

Excellent so far. When you get this far and understand the fourth, and final, iteration to this second example, you are then ready to parse most of the XML you see in the real world. Example 9-7 offers the final version of the second example.

Example 9-7 Second scenario, fourth (final) iteration, XML02_List_d.spl

```
namespace Just01_XML;

composite XML02_List_d {

    type
        MyItem = tuple
            <
                rstring      ItemName,
```

```

        rstring      ItemCount
    >;
MyTxList = tuple
<
    rstring      TxID      ,
    rstring      TxType    ,
    list<MyItem> ItemList
>;
MyRegList = tuple
<
    rstring      RegID     ,
    list<MyTxList> TxList
>;
MyOneTxWithRegID = tuple
<
    rstring      RegID     ,
    rstring      TxID      ,
    rstring      TxType    ,
    rstring      ItemName,
    rstring      ItemCount
>;

```

graph

```

stream<rstring InputLine> Input = FileSource() {
    param
        file      : "/POT/MyFiles/20_XML/XML02_List.in.xml";
        format    : line;
}

```

```

stream<MyRegList> RecordOut = XMLParse(Input) {
    param
        trigger      : "/RegisterOrders";
        parsing      : permissive;
        ignoreNamespaces : true;
output RecordOut :
    RegID = XPath("@RegID"),
    TxList = XPathList

    (
        "ATransaction",
        {
            TxID      = XPath("@TxID" ),
            TxType    = XPath("@TxType"),
            ItemList = XPathList
        }
    )
}

```

```

        "ItemName",
        {
            ItemName = XPath("text()"),
            ItemCount = XPath("@count")
        }
    )
}
);
}

```

```

stream <MyOneTxWithRegID> MyCustom as MyOut =
    Custom (RecordOut as MyIn) {
        logic state :
        {
            mutable MyOneTxWithRegID l_MyTx;
        }
        onTuple MyIn :
        {
            l_MyTx.RegID = MyIn.RegID;
            //
            mutable int32 l_cntrT = 0;
            mutable int32 l_cntrI = 0;
            //
            mutable int32 l_sizeofT = size(MyIn.TxList);
            mutable int32 l_sizeofI = 0;
            //
            while (l_cntrT < l_sizeofT)
            {
                l_MyTx.TxID = MyIn.TxList[l_cntrT].TxID ;
                l_MyTx.TxType = MyIn.TxList[l_cntrT].TxType;
                //
                l_cntrI = 0;
                l_sizeofI = size(MyIn.TxList[l_cntrT].ItemList);
                //
                while (l_cntrI < l_sizeofI)
                {
                    l_MyTx.ItemName =
                        MyIn.TxList[l_cntrT].ItemList[l_cntrI].ItemName ;
                    l_MyTx.ItemCount =
                        MyIn.TxList[l_cntrT].ItemList[l_cntrI].ItemCount;
                    //
                    ++l_cntrI;
                    //
                    submit (l_MyTx, MyOut);
                }
                ++l_cntrT;
            }
        }
    }
}

```

```

}

() as MySink = FileSink(MyCustom) {
  param
    file      : "/POT/MyFiles/20_XML/XML02_List.outd.txt";
    format    : csv;
    quoteStrings : false;
  }
}

```

Consider the following information about Example 9-7 on page 228:

- ▶ Regarding variables and tuples, we finally have a list of lists. A list of lists is required if you are parsing XML-formatted input data, and are receiving XML elements, attributes, or both, at multiple levels in the XML object hierarchy. Note this information:
 - If you have a flat XML document (single-tier, Example 9-1 on page 206), then you do not need lists.
 - If you have a single-tier or multitier XML document with a repeating set of elements or attributes, and you are reading at only one level in the XML hierarchy, then you need only a single list (input data Example 9-3 on page 213, and the same applies to Example 9-5 on page 222).
 - If you have a multitier XML document or documents, retrieving elements or attributes at multiple levels, then you need Example 9-7 on page 228 with a list of lists.
- ▶ Note this information about MyItem, MyTxList, and MyRegList:
 - MyItem is the lowest level tuple (receives elements or attributes from the XML-formatted input data that is the lowest in the element hierarchy), and contains ItemName and ItemCount.
 - MyTxList adds TxID, and TxType, and a list of MyItem. Here we have a tuple with a list.
 - MyRegList adds RegId, and contains a list of the tuple MyTxList, which itself has a list.
 - How to use these variables is detailed in the following text.
- ▶ The RecordOut XMLParse operator has a specified trigger of RegisterOrders, the root element of this XML-formatted data. As a result, every element and attribute in the XML data is available to us. Consider this information:
 - The output block has the variable assignments.
 - RegId is at the root element level (root to the trigger). There is only one RegId, so we extract its value into a simple variable with XPath.

- TxList represents a list. There can be many of these (many ATransactions in the RegisterOrders); we have four. Consider this information:
 - A single TxList value can have only one TxID and one TxType, so we extract these using XPath into simple rstrings. (We are already in a list, MyRegList).
 - ItemName and ItemCount can have multiple values, and are retrieved using XPathList. Here we are in a list of lists: a MyRegList of MyTxList.
- Note all of the parenthesis and curly braces, because they are critical to getting a correct answer. At this point you should be able to read and understand the example, but can you write it from scratch? In 9.3, “The spl-schema-from-xml utility, generating XMLParse code” on page 232, you learn how to generate this code using a Streams utility, and not have to write it.
- ▶ And the custom operator presents some new techniques, namely a set of two nested while loops, to “walk through” the list of lists and correctly output the flattened data values.

Find the following line:

```
l_MyTx.ItemName = MyIn.TxList[l_cntrT].ItemList[l_cntrI].ItemName.
```

The two square bracket pairs refer to a list inside a list (l_cntrT and l_cntrI are the zero-based indexes into those two lists), and the specific element of each we want to address at a single time.

The second example using XMLParse is now complete. From this experience, we believe that you now have the techniques to parse most of the XML-formatted input data you see in the real world. The next section in this document details how to generate most, if not all of the SPL code used in Example 9-7 on page 228.

9.3 The spl-schema-from-xml utility, generating XMLParse code

You should complete and understand Example 9-7 on page 228, so that you can work with the Streams XMLParse operator effectively and accurately. Not having to write that code, or the associated variables, might be convenient. The Streams utility, **sp1-schema-from-xml**, does this for you. It is run from the Linux command-line prompt.

Example 9-8 on page 233 details the specific use of **sp1-schema-from-xml**. We are using the same input file as in Example 9-3 on page 213. We want to create a Streams application similar to Example 9-7 on page 228.

Example 9-8 spl-schema-from-xml to generate the previous example

```
spl-schema-from-xml -t /RegisterOrders -o XML08_Generated.spl
-c --composite Parse --mainComposite XML08_Generated
-f attributes --ignoreNamespaces /tmp/farrell10/XML02_List.xml
```

Consider the following information about Example 9-8:

- ▶ The command is entered on one line. Because of the length of the line, its display appears to span several lines.
- ▶ The command name is **spl-schema-from-xml**. The remainder of the line configures (changes) how this command behaves.
- ▶ The **-t /RegisterOrders** argument specifies the trigger to XMLParse. The value RegisterOrders is from Example 9-3 on page 213.
- ▶ The **-o <filename>** argument specifies the name of the SPL application file we want this utility to create.
- ▶ The **composites** and **mainComposites** arguments affect how and what SPL code is generated.
- ▶ The **ignoreNamespaces** argument calls to ignore XML namespaces.
- ▶ And the final argument is the sample XML-formatted input data file on which to base the Streams tuples and generated code.

Important: The XML sample should be complete and fully representative of the final XML, or the code that will be generated might not be correct.

Example 9-9 displays the nearly unaltered output from the `spl-schema-from-xml` command in Example 9-8. Only the input and output file names were changed. See the code review after Example 9-9.

Example 9-9 XML08_Generated.spl, generated solution.

```
namespace Just01_XML;

use spl.XML::*;

composite Parse(input input0; output output0) {

type
    static RegisterOrders_type = tuple
        <
            rstring schemaLocation,
```

```

    rstring RegID      ,
    list
    <
        RegisterOrders_MyTransaction_type
    > MyTransaction
>;

static RegisterOrders_MyTransaction_type = tuple
<
    rstring TxType      ,
    rstring TxID        ,
    list
    <
        RegisterOrders_MyTransaction_ItemName_type
    > ItemName
>;

static RegisterOrders_MyTransaction_ItemName_type = tuple
<
    rstring count,
    rstring _text
>;

```

graph

```

stream<RegisterOrders_type> output0 = XMLParse(input0) {
    param
        trigger      : "/RegisterOrders";
        parsing       : permissive;
        ignoreNamespaces : true;
    output output0 :
        schemaLocation = XPath("@schemaLocation"),
        RegID           = XPath("@RegID" ),
        MyTransaction  = XPathList
            (
                "ATransaction",
                {
                    TxType = XPath("@TxType"),
                    TxID   = XPath("@TxID" ),
                    ItemName = XPathList
                        (
                            "ItemName",
                            {
                                count = XPath("@count"),
                                _text = XPath("text()")
                            }
                        )
                }
            )
}

```

```

        }
    );
}

}

composite XML08_Generated() {

graph

    stream<rstring s> Input = FileSource() {
        param
            file   : "/POT/MyFiles/20_XML/XML02_List.in.xml";
            format : line;
    }

    stream<Parse.RegisterOrders_type> X0 = Parse(Input) {
    }

    () as 00 = FileSink(X0) {
        param
            file   : "/POT/MyFiles/20_XML/XML08_Generated.out.txt";
    }

}

```

Consider the following information about Example 9-9 on page 233:

- ▶ In effect, **sp1-schema-from-xml** generates three sections of SPL code:
 - The tuple type definitions, including one or two tuple attributes that represent elements of the XML-formatted input data file that we did not reference earlier, for example; `schemaLocation`.
 - The `XMLParse`.
 - A separate composite for input and output.

By having these three sections, you can more easily edit this SPL code to your own requirements.
- ▶ Minus some of the generated variable and tuple names, the `XMLParse` operator should not differ too much from Example 9-7 on page 228.

- ▶ The generated examples use *aliases* for the input and output ports, otherwise that code should be familiar also.

Note: The only item to be concerned with about `sp1-schema-from-xml` is when the source XML-formatted input data file has elements or attributes with names that are Streams keywords. In that case, you can edit the SPL-generated code to repair, and continue.

9.4 The `xquery()` function, including filter

So far in this chapter, we have parsed XML-formatted input data exclusively using the Streams operator, `XMLParse`. We also describe when to use the Streams `XMLParse` operator versus the Streams `xquery()` function. Although `xquery()` can do some tasks that `XMLParse` cannot, the reverse is also true.

Example 9-10 does not use an XML-formatted input data file. The `xquery()` function needs to receive data using the Streams *xml data type*; `xquery()` needs to receive whole XML documents, not a stream of rstrings that together form an XML document. Therefore, and for simplicity, in this example we use a constant rstring, cast it as XML (the Streams XML data type), and continue.

In the real world you might receive this value, this whole XML document, as a blob from a database, or other. Or, you can read from a file, and do string operations to build a whole XML document, then submit to `xquery()`. See the code review after Example 9-10.

Example 9-10 XML12_XQuery.spl

```
namespace Just01_XML;

composite XML12_XQuery {

type
  MyItem = tuple
  <
    rstring      ItemName
  >;

graph

  stream <int32 i> MyBeacon as MyOut = Beacon() {
```

```

    param
        iterations: lu ;
    output
        MyOut: i = 1 ;
}

stream <MyItem> MyCustom as MyOut =
    Custom (MyBeacon as MyIn) {
    logic state :
    {
        mutable MyItem      l_MyItem ;
        //
        mutable rstring      MyStr =
            "<?xml version='1.0' encoding='UTF-8'?>
            <RegisterOrders RegID='Register-100'>
            <ATransaction TxID='101' TxType='sale' >
                <ItemName count='2'>Helmet      </ItemName>
                <ItemName count='4'>Cap          </ItemName>
            </ATransaction>
            <ATransaction TxID='102' TxType='sale' >
                <ItemName count='1'>Jersey      </ItemName>
            </ATransaction>
            <ATransaction TxID='103' TxType='return'>
                <ItemName count='3'>Swim Fin    </ItemName>
                <ItemName count='6'>Swim Mask   </ItemName>
            </ATransaction>
            <ATransaction TxID='104' TxType='sale' >
                <ItemName count='5'>Football    </ItemName>
                <ItemName count='7'>Baseball    </ItemName>
                <ItemName count='9'>Frisbee     </ItemName>
            </ATransaction>
            </RegisterOrders>";
        mutable xml          MyXML      ;
        //
        mutable list<rstring> MyResult  ;
        mutable int32        MyCntr = 0;
        mutable int32        MySizeof  ;
    }
    onTuple MyIn :
    {
        convertToXML(MyXML,MyStr);
        MyResult = xquery(MyXML,
            "/RegisterOrders/ATransaction[@TxType =
'return']/ItemName/text()");
        MySizeof = size(MyResult);
        //
        while (MyCntr < MySizeof)
        {

```

```

        l_MyItem.ItemName = MyResult[MyCntr];
        submit ( l_MyItem, MyOut );
        //
        MyCntr++;
    }
}

() as MySink = FileSink(MyCustom) {
    param
        file          : "/POT/MyFiles/20_XML/XML12_List.out.txt";
        format        : csv;
        quoteStrings  : false;
}
}

```

Consider the following information about Example 9-10 on page 236:

- ▶ MyItem is a tuple of just one variable type, rstring. This is the variable for which we generate a value in the custom operator, and pass to the FileSink.
- ▶ A *beacon* operator is used to send a single input tuple to the custom operator. This is done to cause the custom operator to fire one time. The custom operator, as constructed, already has all of the data it needs, now we just need an input tuple so that it fires.
- ▶ The custom operator then executes its onTuple block one time. This onTuple block has a while loop that outputs one tuple for each while-loop iteration.
- ▶ Specific to the custom operator, note the following information:
 - MyStr contains the XML document, and is actually one long line; it displays over several lines, ending with </RegisterOrders>.
 - If MyStr does not contain one well-formed XML document, the invocation of xquery() will generate a runtime error.
 - The rstring, MyStr, is converted to a Streams xml data type named, MyXML. This is required because xquery() expects an XML input data type.

Note: Although we placed the XML document in an rstring, then converted to the SPL xml data type, we could have immediately placed the XML document in the SPL xml data type. The example here details how to convert from rstring to xml.

- Note the information about just the xquery() invocation:
 - MyXML is passed to xquery(). The result of this invocation is a list named MyResult.
 - Also passed to xquery() is an xquery expression (recall xquery expressions are a superset of XPath) of greater complexity and capability than we can pass to XMLParse. This XPath expression contains a filter. The [`@TxType = 'return'`] substring indicates that of all the ATransactions we receive, we want to output only those with TxType = 'return' attribute type.
 - Filtering is a capability that xquery() has and XMLParse does not. We can accomplish the same goal with XMLParse and a subsequent operator, but filtering is inherent to xquery().

Note: XPath considers the reduction in the number of elements produced or referenced to be a form of filtering. The filtering we detailed is more exactly referred to as *fine grained filtering*.

- While we are filtering at the ATransaction level, we choose to output `ItemName/text()`.
- ▶ The sample output of this operator will contain two tuples: one column each, with the values `Swim Fin` and `Swim Mask`. If you reference the value of `MyStr` from Example 9-10 on page 236, you see these are the two `ItemNames` from an ATransaction with a 'return' TxType.

9.5 CDATA: Topic that is not covered in detail

There is a concept in XML named, CDATA. One use for CDATA is when you have an XML document, and want to embed another XML document or XML subfragment within it. This occurs often when machine data is XML-encoded, and is placed inside another XML record. XMLParse can easily handle this. From the example in Figure 9-8 on page 240, you trigger or use XPath on `ItemName`, receive the “message” XML fragment, and then use XMLParse again to receive any subelements to the “message.”

```

<?xml version="1.0" encoding="UTF-8" ?>
- <tns:RegisterOrders RegID="Register-100" xmlns:tns="http://www
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" x
- <tns:ATransaction TxID="101" TxType="sale">
  <ItemName count="2">Helmet</ItemName>
  <ItemName count="4">Cap</ItemName>
- <ItemName>
  <![CDATA[ <message>Embedded XML 1</message> ]]>
  </ItemName>
- <LogRecord>
  <![CDATA[ <message>Embedded XML 1</message> ]]>
  </LogRecord>
</tns:ATransaction>
- <tns:ATransaction TxID="102" TxType="sale">
  <ItemName count="1">Jersey</ItemName>
</tns:ATransaction>
- <tns:ATransaction TxID="103" TxType="return">
  <ItemName count="3">Swim Fin</ItemName>
  <ItemName count="6">Swim Mask</ItemName>
- <ItemName>
  <![CDATA[ <message>Embedded XML 2</message> ]]>
  </ItemName>
</tns:ATransaction>
- <tns:ATransaction TxID="104" TxType="sale">
  <ItemName count="5">Football</ItemName>
  <ItemName count="7">Baseball</ItemName>
  <ItemName count="9">Frisbee</ItemName>
- <ItemName>
  <![CDATA[ <message>Embedded XML 3</message> ]]>
  </ItemName>
</tns:ATransaction>
</tns:RegisterOrders>

```

Figure 9-8 Handling CDATA with XMLParse



Geospatial Toolkit

In this chapter, we introduce the Geospatial Toolkit (referred to as “toolkit” in this chapter), a collection of types and functions that are useful in creating applications that process and analyze geographic location data. With the advent of low-cost GPS devices, smartphones, location-aware vehicles, and ubiquitous wireless communications, the number of *location-based services* (LBS), both professional and consumer-oriented, is exploding. Uses range from advertising, taxi dispatching and emergency response to route planning, crime prevention, and national security.

The toolkit, as introduced in Streams 3.0, provides solid fundamentals and basic functionality. Its internal architecture is tailored to real-time location processing and includes a high-performance, robust geometry engine. The initial functionality provided is enough for a large class of common location-based scenarios, but we expect this toolkit to evolve toward more complete functionality over successive releases.

10.1 Concepts

This section lays the foundation for understanding the toolkit. To understand it, you must have basic familiarity with maps and geometrical concepts, but it does not attempt to make you an expert in computer-based geography. The scenarios, main concepts, and terminology should be accessible to anyone who has used digital mapping technology on the web and on hand-held devices.

Geospatial data is anything that represents shapes and locations on the surface of the Earth. (One can easily extend this to other planets and celestial bodies, but we will leave that to the reader's imagination.) In the computer, these shapes are abstract representations of physical things in the real world: a *model*. Some models and approximations are more realistic than others, and the right level of detail and fidelity depends on the application. For example, for many applications, a point in two dimensions is a perfectly adequate representation of a vehicle or a building or a city; for others, more complex geometric shapes in two or three dimensions are needed. What is most important is that, given a model, we have a set of operations and rules that are not only mathematically consistent, but also computationally efficient, robust, and predictable.

Those shapes and locations are referenced to some explicit or assumed *coordinate reference system*; relationships and operations are defined by theorems and techniques from geometry, topology, set theory, and other branches of mathematics. For most users, it is not necessary to delve deeply into these scientific fundamentals, but occasionally some of it seeps to the surface and must be discussed. For example, Euclidean geometry, which works on flat surfaces such as maps, is different from the geometry of curved surfaces such as a globe or the Earth itself. When you are trying to locate an object in a factory or a warehouse, you would stay with Euclid, but when you are tracking moving ships, airplanes, animals, or vehicles anywhere on Earth, you must do something different.

10.1.1 Moving objects and location-based services

The purpose of Streams is to process and analyze streaming data in near-real time. This implies that there is a time factor involved in all of its data feeds: data packets (tuples, records, messages, events) arrive in a specific order. Whether the order is important or not, this time-sequential nature is one of stream processing's key characteristics—one that is generally not present in, say, database applications. This makes Streams ideal for handling a particular kind of geospatial data: the kind that involves objects that move.

It is perhaps useful to point out what this chapter is not about. A well-known and long-established geospatial discipline is that of *geographic information systems*,

or *GIS*. Think of a GIS as an intelligent map (one you can query), or as a database with a mapping front-end. GIS implementations are generally concerned with such things as land ownership, land use patterns, soil types, natural resources, topography, political boundaries, demographics, transportation networks, and so on. They are often the domain of government authorities, and support sophisticated analyses and highly developed visualization techniques involving maps. Streams has little to add to systems like that, because in general they are static representations of the world, with map (database) updates happening slowly, as a result of human-generated transactions such as surveys and property sales. As with other types of data stores, a geospatial Streams application typically uses a GIS as reference data to augment (“enrich”) streaming data, but does not directly participate in its data management and visualization processes.

As a term, *location-based services (LBS)*, is closer to describing a typical scenario for Streams. The term came to be associated with a telecom bubble and crash over a decade ago, when cellular operators spent billions of dollars on spectrum and were hoping to roll out myriad services for business, travelers, and consumers based on their instantaneous location and also their daily, weekly, and seasonal patterns. But the public and the technology were not ready, and viable business models were hard to find. Now, however, technology and public expectations have evolved; location-enabled smartphones and other devices are ubiquitous, and many professional, government, social-media, and consumer-oriented services have been invented and successfully deployed.

In light of this, the time is right for a powerful, efficient, open, location-aware, streaming-data platform: Streams with the Geospatial Toolkit.

10.1.2 Geospatial concepts and operations

Before we can describe the Geospatial Toolkit, we must introduce several terms and concepts.

Coordinates: straight and curved (hint: the Earth is round)

Locations and shapes are generally defined by starting with point locations, given by two (or more) *coordinates*: pairs (or triplets) of numbers that represent, for example, distance from some origin along some set of axes. That set of axes, or whatever else we use, is called a *coordinate reference system*. Just like we have to choose a character set for text values, a currency for money amounts, and units for physical measures, so we have to choose a coordinate reference system for geospatial objects.

The most obvious and simplest choice of coordinate reference goes back to secondary-school geometry: two axes at right angles in a plane. (This may also

be called a Cartesian coordinate system or orthonormal, rectilinear reference system in a two-dimensional Euclidean space.) Figure 10-1 shows such a reference system.

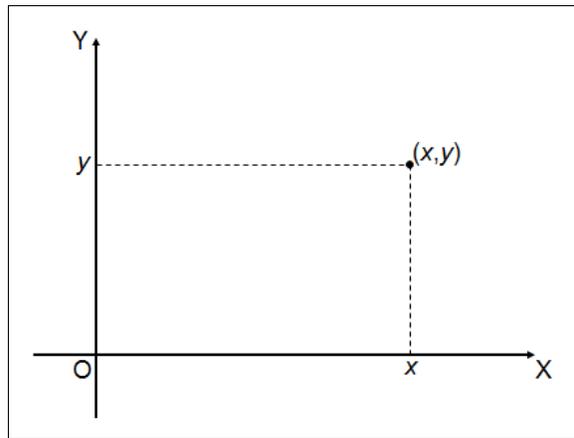


Figure 10-1 A point (x,y) in a Cartesian coordinate reference system

In this coordinate system, roughly speaking, the coordinates x and y of a point represent the straight-line distance to each of the two axes along the direction of the other. This is the natural coordinate system of a flat piece of paper: a map. Those of us who are old enough to have grown up with paper maps know well how useful such devices are; in particular, a map is much easier to carry around than a globe of the same scale.

The problem is, however, that the Earth does not really look like that: its surface is not flat. Unlike a map, it has no edges and contains no straight lines, and it is not possible to create a flat representation of the Earth's surface (this process is called *projection*) without introducing distortions in distances, angles, and areas. In many cases, and in particular for traditional GIS applications, this problem is not as bad as it sounds. On the scale of a city or county, the curvature of the Earth is unlikely to cause any problems in software that represents coordinates as X and Y in a well-chosen map projection. Over a large area, however, the distortions introduced by projecting the round globe onto a flat map become intolerable. For example, Greenland, which has about one-quarter the land area of Brazil, looks as big as all of South America on many world maps. Likewise, connecting two locations anywhere in the world and computing their distance can produce absurd results.

If a flat plane is not a good representation of the surface of the Earth, what is? The answer is that something resembling a sphere (a *spheroid*) is much better. Like the Earth's surface, a sphere has no edges; no straight line can be drawn on it. From a physics perspective, a sphere is almost correct: because of gravity, a body of sufficient size and weight in space will become a sphere in time (even rocks flow like a liquid on cosmological timescales). The only wrinkle is that this rock rotates, which causes the sphere to become flattened at the poles and widened at the equator: it becomes an *oblate spheroid*, which can be mathematically described as an *ellipsoid of rotation*, or the shape described in space when an ellipse is rotated about its shorter axis. Because the Earth's flattening is fairly small, not all applications require the use of an ellipsoidal Earth model; calculations on a sphere are simpler and faster, and might be good enough. Other applications, however, require more accurate, ellipsoidal calculations; in these, the assumed shape and position of the ellipsoid (how much it is flattened and where its center is relative to the Earth) is an important factor. Over the years, many ellipsoids (*datums*) have been defined.

With the popular adoption of the Global Positioning System (GPS), its datum, the World Geodetic System of 1984 (*WGS84*) has become the de-facto standard. Unless otherwise mentioned, the toolkit uses WGS84.

The coordinate reference system on the spheroid may not be as simple as the familiar Euclidean one of Figure 10-1 on page 244, but it should be familiar: instead of linear measures, it uses angles called *latitude* and *longitude*, also known as *geographic coordinates*. For a point on the surface, these are defined as depicted in Figure 10-2 on page 246: latitude is the angle between the vertical direction at that point and the equatorial plane; longitude is defined as the angle between the projection of the vertical on the equatorial plane and an axis through an arbitrarily chosen *Prime Meridian*, where the longitude is zero. (A meridian is a line of constant longitude. The current Prime Meridian is the one passing through the Royal Observatory in Greenwich, England. This is purely by convention, and a relatively recent one.) Historically and in human communication, latitude and longitude are expressed in degrees, minutes, and seconds along with a cardinal direction: north or south for latitude, east or west for longitude. For computing purposes, however, they are usually expressed as a signed floating-point number representing degrees (or, sometimes, radians) with a decimal fraction, with negative numbers denoting southern latitudes and western longitudes. For example, IBM Headquarters in Armonk, New York, USA, is at this location:

41° 6' 29.9" North, 73° 43' 13.7" West

In the computer, that location gets this numerical representation:

41.10831, -73.72047

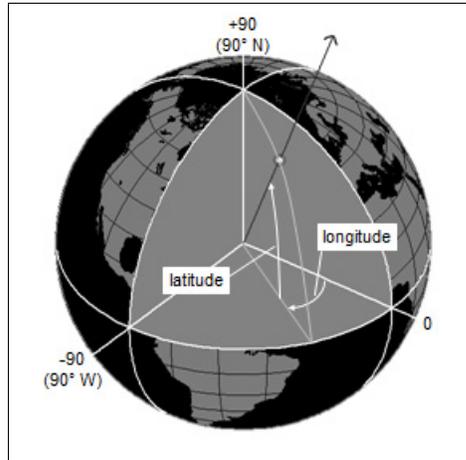


Figure 10-2 Geographic coordinates: latitude and longitude

Because modern mobile technology, such as GPS devices, reports locations in geographic coordinates, the toolkit uses these exclusively; it does not deal with map projections and planar coordinates.

Besides a reference system for locating points on the globe, a consistent definition is needed for the path connecting two points. This is important for the definition of geometric shapes such as lines and polygons, and for calculating distances and areas. On the map, the obvious choice is the straight line segment, which is unique and denotes the shortest path between the end points. On the sphere, there are no straight lines, but the shortest path between two points is a piece (or *arc*) of a *great circle*, meaning a circle whose center is the center of the sphere. On the ellipsoid things are more complicated, but a good approximation of the shortest path is the equivalent of a great circular arc, namely a *great elliptic arc*. The toolkit calls this a *line segment* and uses it as the basis of its geometric constructions and distance calculations, with various optional degrees of approximation for computational performance.

An example of a great elliptic arc is in Figure 10-3.

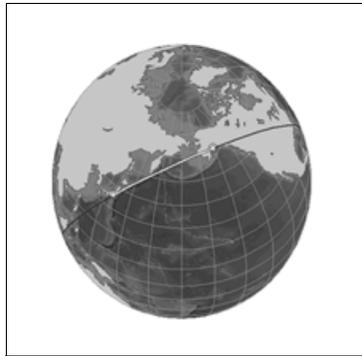


Figure 10-3 The path between Tokyo and Anchorage: a great elliptic arc

Feature types and geometry: points, lines, and polygons

To represent real-world objects (entities) as geometric shapes in the computer (features), we need at a minimum to be able to represent point locations, which have no extent in any dimension, linear shapes, which have an extent (length) in one dimension, and regions, which have an extent (area) in two dimensions. The simplest representations of such shapes are generally called points, lines, and polygons. The toolkit concerns itself mainly with points and line segments, and has rudimentary support for lines consisting of multiple segments (known as *line strings*) and for polygons.

In the toolkit, a *point* is fully defined by its latitude and longitude coordinates. A *line segment* does not follow the mathematical definition (in which a line segment is by definition straight) but is a great elliptic arc (see Figure 10-3), defined by its start and end points. There is no support for multi-segment lines (line strings) values; instead, one of the functions simply takes a list of line segments. It is the responsibility of the application to enforce the requirement that the line segments connect end-to-end. In geospatial systems, a *polygon* is defined by its boundary, which is a special kind of line string called a *ring*. A ring is a line string that does not cross itself (it is *simple*) and whose first and last points are the same (it is *closed*).

Figure 10-4 shows the requirements for a line string to form a valid ring, and for a valid *simple* polygon, which is a polygon defined by a single ring: no holes, no islands. Figure 10-5 shows an example of a simple polygon on the globe. In the toolkit, there is no data type or other support for polygons; instead, one of the functions takes a list of points, leaving it to the application to enforce the requirement that they appear in order to form a ring.

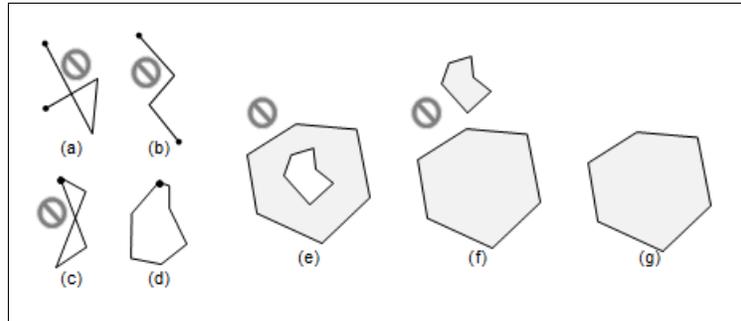


Figure 10-4 Line string, ring, and simple polygon.
 (a) - (d) Not all line strings form a ring: only (d) is simple and closed and thus a ring; others are either not simple (c), not closed (b), or both (a).
 (e) - (f) A simple polygon has a single ring as its boundary. Only (g) is simple; the others have an additional ring to define a hole (e) or island (f).

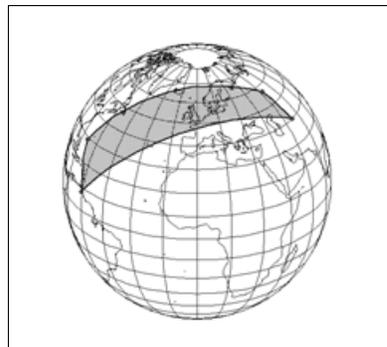


Figure 10-5 A simple, four-point polygon whose boundary is a simple, closed line string consisting of great-elliptic-arc "line segments"

Operations on points: distance and azimuth

Having defined a few types of geometric features, we describe what can be done with them. Aside from the ability to construct values of type point, line segment, or polygon, one of the most basic operations is to compute the *distance* between two points (the length of the path, or *line segment*, connecting them). In Figure 10-3 on page 247, that would be the distance between Tokyo and Anchorage.

Distance calculation may be a basic operation, but it makes possible several powerful types of analysis, as in the following examples:

- ▶ *Proximity* analysis: Determining how close moving objects are to each other or to a fixed location
- ▶ *Local search*: Finding all moving objects within a given search radius of a fixed point
- ▶ *Nearest-neighbor* analysis: Ranking objects by their distance to a given moving or fixed location

Closely related to the distance between two points is the direction of travel at which the connecting great elliptic path leaves the starting point; this is known as the *azimuth*, *bearing*, or *heading* (all three terms are used interchangeably in the toolkit). It is measured as the angle between north (along the local meridian) and the starting direction along the path to the destination point, as illustrated in Figure 10-6. In the toolkit, this is expressed in radians: $\pi/2$ is due east, π is due south, and so on. Note that the azimuth changes along the path; the computed azimuth is for the starting point only.

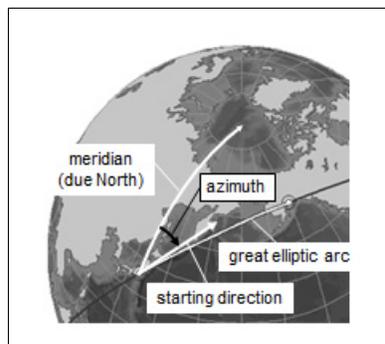


Figure 10-6 Azimuth is the angle from due north

The inverse of computing distance and azimuth given two points is to compute the *destination point*, given a starting point, distance, and azimuth. The starting point and azimuth uniquely determine the great ellipse, and the distance simply specifies how far to travel along that ellipse. This has fewer obvious applications,

but is important in predictive geospatial analytics, where it may be necessary to extrapolate from a currently known trajectory to find the future position of a moving object. For vehicles on a road network this may not work well, but for airplanes and missiles it may.

Operations on points and lines: nearest point

An operation that has direct relevance to the tracking of moving vehicles in a road network is the ability to project a point location onto a line string. A vehicle can be tracked using its GPS position, but such positions have inherent uncertainty and limited precision. It is often important not to take the GPS coordinates as reported, but to adjust (or “snap”) them to a location along a known nearby road (under the assumption that vehicles are moving along roads, not inside buildings or in undeveloped fields). The moving vehicle is represented by a point; the road by a collection of line segments. The logical choice for a location along that road is the nearest point: in most cases this point is found by traveling along the perpendicular to the nearest line segment. This is called the *perpendicular projection* of the point onto the line string; see Figure 10-7. In other cases, the nearest point is simply the start and end point of one of the line segments. In addition to finding the nearest point on the nearest line segment, this process can also yield the measured *cross-track distance* between the original point and its projection (d in the figure) and the *along-track distance*, meaning the distance from the start point of the first line segment to the projected point, measured along the sequence of line segments (s in the figure).

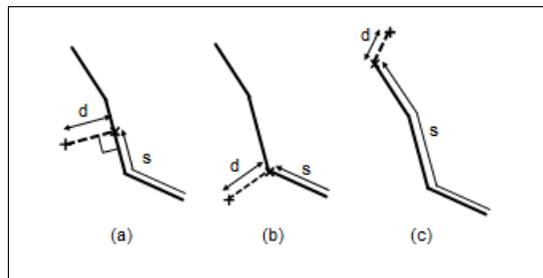


Figure 10-7 Projection (x) of a point (+) onto a string of line segments; d is the cross-track distance; s is the along-track distance. In (a), the projection is perpendicular onto a line segment; in (b) and (c), the nearest point is the start and end point of one or two line segments.

Relative spatial relationships: containment

A key process in geospatial systems is to determine the spatial relationship between geometric shapes: whether a point is on a line or inside a polygon, whether two lines cross or two polygons overlap, and so on. This is generally analyzed in terms of *set theory*: line segments, lines, polygons, and polygon boundaries are (infinite) sets of points, and it is possible to reason about the various kinds of intersection between such point sets.

The toolkit supports one specific type of intersection: containment, which determines whether one point set wholly contains another. It does this for three combinations of feature types (see Figure 10-8):

- ▶ A list of points and a point: Determine whether a point is in a (simple) polygon.
- ▶ A list of line segments and a point: Determine whether a point is on a line string.
- ▶ Two lists of points: Determine whether one polygon wholly contains another.

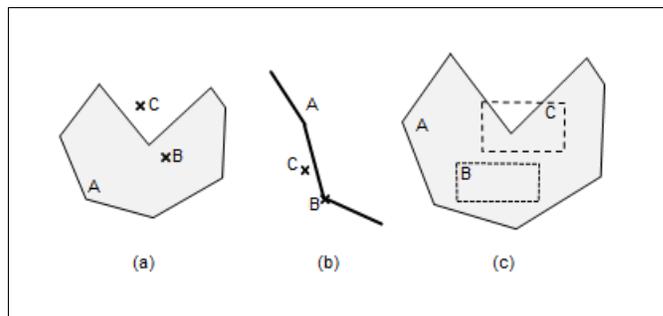


Figure 10-8 Containment. In each figure, A contains B and A does not contain C. (a) point in polygon; (b) point on line string; (c) polygon in polygon.

10.2 Toolkit organization

The first thing to notice is that this toolkit has no operators. It consists solely of types and functions. This makes the toolkit's functionality available anywhere you can write SPL expressions: in SPL functions and in the logic, param, and output clauses in operator invocations. This section provides an overview of the types and functions.

Note: All code listed in this chapter is available for download as a project that you can extract and import into Streams Studio. See Appendix A, "Additional material" on page 527 for links and instructions.

10.2.1 Namespaces

The toolkit's artifacts are organized in namespaces. The structure might seem somewhat elaborate for the limited amount of functionality, but the intent is to allow for much future enhancement while keeping things organized.

com.ibm.streams.geospatial.conversions

This namespace has a few convenience functions for converting linear units. Because few modern geospatial systems use traditional units such as miles and feet, these are of limited use.

com.ibm.streams.geospatial.twodimension.geometry

All core types and functions are in this namespace. To use the toolkit, put a use directive in each SPL source file that references its types and functions.

10.2.2 Types and enumerations

This section describes feature types, helper types, and the metric option (enumeration).

Feature types

As described in “Feature types and geometry: points, lines, and polygons” on page 247, the toolkit has data types only for points and line segments; the other feature types, line strings and polygons, are constructed as lists of line segments and points, respectively.

The geospatial types, and many of the functions, are available in three levels of precision, designated by a numeric suffix: point128, lineSegment64, and so on. They determine how the latitude and longitude coordinates of each point are represented, allowing developers to reduce tuple size in exchange for reduced precision.

Table 10-1 summarizes the options.

Table 10-1 Three levels of coordinate precision

Suffix	Coordinate representation	Precision (bits)	Precision (spatial)	Comment
128	2 separate float64 values	53	arbitrary	Useful in development
64	Compressed into one doubleword	~33	5 mm	Opaque ^a , efficient
32	Compressed into one word	~18	305 m	Opaque, too coarse

a. *Opaque* means that the individual coordinates cannot be read or manipulated directly.

Note: Do not mix precision levels in a project; that would require conversions everywhere, and lead to needless complexity and performance degradation.

As a guideline, you want the coordinate precision to be one or two orders of magnitude better than the external requirements for your application; the internal data representation must not become the limiting factor. For example, if your GPS data collection is precise to about 50 cm, the 5 mm spatial precision of the 64-bit representation is fine. But if you need to locate objects with centimeter precision, you may need the full, 128-bit representation. However, if all locations are only approximate to a few km and the bandwidth requirements are extreme, the 32-bit representation may suffice.

The 128-bit representation is especially useful during development, as it is easy to read the raw data values in Streams Console charts and tables or in Streams Studio “Show Data” views. For production use, however, pick the precision that best matches the requirements.

Because the precision suffixes violate the principle of information hiding (they mix implementation details into the public interface), a good practice is to create covers over the toolkit data types, and use only those cover types in your application code. This way, the choice of implementation is confined to one location, and if you want to change it (for example, when switching from development to production, as suggested in the preceding paragraph), only one source file is affected. Because the toolkit types are defined inside composites (named `PointPropertyType` and `CurvePropertyType`), covering them also lets you shorten the type names considerably, as shown in Example 10-1 on page 254.

Example 10-1 Cover types that hide implementation details

```
use com.ibm.streams.geospatial.twodimension.geometry::*;

// Cover the toolkit types with more manageable names
// Choose a precision and hide it: 128, 64, or 32
type PointT      = PointPropertyType.point64;
type LineSegmentT = CurvePropertyType.lineSegment64;
```

Helper type

One of the toolkit functions finds the point on a line string that is nearest to a given point; see “Nearest point on line string” on page 258. The wrinkle in this function is that it not only returns the nearest point, it also reports the along-track and cross-track distance, as shown in Figure 10-7 on page 250. The three results are wrapped in another tuple type, which, like those for point and line segment, is best covered by a redefinition. For example, for precision suffix 64, the toolkit defines this type inside a composite called `CurvePropertyType`:

```
static mapPointToLineString64 = float64 crossTrackDistance,
                                float64 alongTrackDistance,
                                PointPropertyType.point64 pointOnLineString;
```

A possible cover for this type is in Example 10-2.

Example 10-2 Cover type for use with `mapPointToLineString()`

```
// Hide long name and precision detail
type NearestResultT = CurvePropertyType.mapPointToLineString64;
```

Metric option (enumeration)

Several functions in the toolkit return measures such as distance and bearing, or use those measures to produce new geospatial values. See “Destination point” on page 257. As in the size and precision of the feature types, an implementation detail intrudes into these functions: the approximation (here called *metric*) used for calculating the result.

Why is this needed? Because complete, accurate distance and bearing calculations using the best model for the planet (an ellipsoid) are computationally highly complex and not necessarily robust (meaning that they do not arrive at a consistent result in all cases). Thus, approximate methods are needed that are faster and more stable, and accurate enough. What is “accurate enough” depends on the nature of the problem (for example, the maximum distance spanned by any two features in the project area), so a single method might not be enough.

The enumerated values for the various metric options are described in Table 10-2.

Table 10-2 Values for the metric argument to the DestinationPoint function

Metric value	Description	Comment
Metric.Spherical	Assumes a spherical shape of the Earth and applies one of two spherical algorithms, depending on distance	Fast and fairly accurate. Works for all distances. Haversine for distances < 10 km, great-circle calculations otherwise.
Metric.EuclideanSphere	Applies a local Mercator projection using a spherical shape of the Earth, followed by distance calculations in the plane	For short to moderate distances (< 10 km), faster and more accurate than Spherical. Not accurate for longer distances.
Metric.EuclideanEllipsoid	Applies a local Mercator projection using an ellipsoidal (WGS84) shape of the Earth, followed by distance calculations in the plane	For short to moderate distances (< 10 km), more accurate but slower than EuclideanSphere. Not accurate for longer distances.
Metric.Ellipsoidal	Assumes the WGS84 ellipsoid and applies full ellipsoidal calculations (Vincenty formulae)	Most accurate and works for all distances, but comparatively extremely slow. Use to check other results, not for production.

For more information about great-circle and haversine calculations and a link to the original Vincenty paper, go to the following web address:

http://en.wikipedia.org/wiki/Great-circle_distance

10.2.3 Constructor functions

The geospatial types have their own constructor functions for creating values of these types. Like the types themselves, however, the constructor functions carry a precision suffix; here, too, a good practice is to create cover functions to hide this implementation detail and localize the dependencies, as Example 10-3 on page 256 shows.

Example 10-3 Cover functions for the type constructors

```
// Constructor functions with implementation-neutral names
// Precision “64” version; change suffix as needed
// This version lacks input argument validation
public PointT point(float64 latitude, float64 longitude)
{
    return createPoint64(latitude, longitude);
}

public LineSegmentT lineSegment(PointT start, PointT end)
{
    return createLineSegment64(start, end);
}
```

Note: SPL is strongly typed, yet calls to the toolkit’s constructor functions with cover-type arguments (PointT, LineSegmentT) succeed, without explicit casts, even though their signatures are not defined in terms of those types. This is because SPL applies *structural equivalence*: when two tuple types have the same underlying structure, they are considered identical and do not require casting.

As mentioned in “Feature types and geometry: points, lines, and polygons” on page 247, the toolkit has no types for line strings and polygons. Instead, it relies on the support for collection types in SPL; specifically, a line string is represented by a list of line segments, and a (simple) polygon by a list of points.

Note: There is no validation of input arguments in the constructor functions of Example 10-3. This is discussed and remedied in 10.2.7, “Validation of input arguments” on page 261.

10.2.4 Accessor functions

Data types commonly have accessor functions, which can “get” and “set” the individual members of the underlying implementation. For the point data types, this means the latitude and longitude, and the toolkit duly provides getLatitude() and getLongitude() functions. No cover functions are needed for these.

The toolkit does not provide accessors for the line segment data types, so creating them might be useful, again to promote the localization of implementation-dependent details:

Example 10-4 Accessor functions for the line segment type

```
// For completeness, line segment accessors
public PointT getStart(LineSegmentT lineseg)
{
    return lineseg.start;
}

public PointT getEnd(LineSegmentT lineseg)
{
    return lineseg.end;
}

// Note that in some future release, length and bearing may be computed
// on the fly rather than stored
public float64 getLength(LineSegmentT lineseg) // Notice: renamed
{
    return lineseg.distance;
}

public float64 getBearing(LineSegmentT lineseg)
{
    return lineseg.bearing;
}
```

10.2.5 Spatial production functions

Some geospatial operations work on geospatial values and produce new geospatial values. (This description excludes the constructor and accessor functions, which create geospatial values by nongeometric means.) These are sometimes termed *production functions*. The toolkit provides two; because of various complications, each merits its own section.

Destination point

The purpose of the `destinationPoint()` function is described in the last paragraph of “Operations on points: distance and azimuth” on page 249. The function signature is mostly what you would expect: a start point, a heading (bearing), and a distance. In addition, it takes an extra argument: the approximation (here called *metric*) used for calculating the result, as described in “Metric option (enumeration)” on page 254.

Again, good practice suggests wrapping this function in another to hide this implementation dependency. A cover function is shown in Example 10-5 on page 258.

Example 10-5 Cover function for destinationPoint()

```
// Cover function for destinationPoint()
// Choose a metric and hide it
public PointT destinationPoint(PointT start, float64 heading,
                               float64 distance)
{
    return destinationPoint(Metric.Spherical, start, heading, distance);
}
```

Nearest point on line string

The `mapPointtoLineString()` function implements the process of finding the point on a line string nearest to a given point, described in “Operations on points and lines: nearest point” on page 250. Its result is not just a point value, but a tuple whose type is described in “Helper type” on page 254.

The `MapPointToLineString()` toolkit function differs from the others in another aspect also: rather than providing the result as a function return value, it does so in a mutable argument. A cover function lets you change that, if you prefer; and you might want to change the name also. All this is shown in Example 10-6. To illustrate, the example provides two variants of the function: one gives the result in the return value, the other in a mutable argument. The two functions have the same name but their signatures (the types of their arguments) distinguish them.

Example 10-6 Cover functions for MapPointToLineString()

```
// Cover function with return value
public NearestResultT nearestPoint(list<LineSegmentT> string,
                                   PointT point)
{
    mutable NearestResultT result = {};
    mapPointToLineString(string, point, result);
    return result;
}

// Cover function with mutable argument
public void nearestPoint(list<LineSegmentT> string, PointT point,
                        mutable NearestResultT result)
{
    mapPointToLineString(string, point, result);
    return;
}
```

Use with care: As indicated in “Feature types and geometry: points, lines, and polygons” on page 247, the toolkit does not provide a data type for line strings; instead, it uses a list of line segments. In addition, the internal calculations are performed in the Euclidean space of a projected map (equivalent to the EuclideanSphere metric from Table 10-2 on page 255); this means that it only works well and accurately when the area covered or distance spanned by the input geospatial values is limited.

10.2.6 Spatial relationship functions

The most commonly used functions in geospatial applications are those that measure aspects such as distance, length, azimuth, circumference, area, and so on; and those that determine geospatial relationships between two features, such as whether they intersect. The toolkit provides three functions, for azimuth, distance, and containment.

Azimuth and distance

The azimuth and distance functions measure two aspects of the path connecting two points: azimuth (bearing) and distance, as described in “Operations on points: distance and azimuth” on page 249. Like the preceding `mapPointToLineString()` (`nearestPoint()`), these two functions take an extra *metric* argument, described in “Metric option (enumeration)” on page 254, to indicate which approximation to use in the calculations. Example 10-7 shows suggested covers for these functions.

Example 10-7 Cover functions for azimuth and distance

```
// Choose a "metric" type and hide it
public float64 azimuth(PointT start, PointT end)
{
    return azimuth(Metric.Spherical, start, end);
}

public float64 distance(PointT start, PointT end)
{
    return distance(Metric.Spherical, start, end);
}
```

The distance function has additional variants, `distanceMetric()`, where *Metric* is `Ellipsoidal`, `GreatCircle`, or `Haversine`. The correspondence with the other signature and its metric options is shown in Table 10-3 on page 260. The various signatures provide more options than you need; if you hide your choice inside a cover function, which style you choose does not matter.

Table 10-3 Equivalent distance functions

<code>distanceMetric(p0, p1)</code>	Equivalent distance(<i>metric</i> , p0, p1)
<code>distanceEllipsoidal(p0, p1)</code>	<code>distance(Metric.Ellipsoidal, p0, p1)</code>
<code>distanceGreatCircle(p0, p1)</code>	<code>distance(Metric.Spherical, p0, p1)</code> , <i>distance > ~10 km</i>
<code>distanceHaversine(p0, p1)</code>	<code>distance(Metric.Spherical, p0, p1)</code> , <i>distance < ~10 km</i>

Containment

The `isContained` function provides the three types of containment test described in “Relative spatial relationships: containment” on page 251. Its return type is `boolean`: `true` if the first argument fully contains the second, otherwise `false`. There is no exposure of implementation details in this case: the function name does not reveal which precision suffix is used for its point and line-segment arguments, and signatures are provided for all possibilities (the precision levels of the two arguments must be the same).

There is, however, scope for confusion in this function. Note that the containment relationship is not commutative: “A contains B” does not imply “B contains A”. For this reason, in most products and standards, this function comes in a pair: typically, `Contains()` and `Within()`, where:

- ▶ `Contains(A, B)` returns `true`, if A contains B
- ▶ `Within(A, B) ≡ Contains(B, A)`, `true`, if B contains A

The name and signature chosen in this toolkit are different when compared to similar, well-known functions in other products and industry standards. The function `isContained(A, B)` returns `true` if A contains B—like `Contains()`—while the name suggests the other relationship—like `Within()`. For this reason, it is still prudent to create cover functions and hide the unusual implementation, as shown in Example 10-8 on page 261.

Use with care: As indicated in “Feature types and geometry: points, lines, and polygons” on page 247, the toolkit does not provide a data type for polygons; instead, it uses a list of points to construct a simple polygon, whose boundary consists of a single ring. In addition, the internal calculations are performed in the Euclidean space of a projected map (equivalent to the `EuclideanSphere` metric from Table 10-2 on page 255); this means that it works well and accurately only when the area covered or distance spanned by the input geospatial values is limited.

Example 10-8 Cover functions for isContained()

```
// Cover isContained() with standard function signatures
public boolean contains(list<PointT> p1, list<PointT> p2)
{ return isContained(p1, p2); } // Polygon in polygon

public boolean contains(LineSegmentT p1, PointT p2)
{ return isContained(p1, p2); } // Point on line segment

public boolean contains(list<PointT> p1, PointT p2)
{ return isContained(p1, p2); } // Point in polygon

public boolean within(list<PointT> p1, list<PointT> p2)
{ return isContained(p2, p1); } // Polygon in polygon

public boolean within(PointT p1, LineSegmentT p2)
{ return isContained(p2, p1); } // Point on line segment

public boolean within(PointT p1, list<PointT> p2)
{ return isContained(p2, p1); } // Point in polygon
```

10.2.7 Validation of input arguments

The cover functions presented so far ignore the possibility of invalid input arguments. What to do about it depends on the type of function and the type of arguments, as the following sections explain.

Floating point arguments

As noted in 10.2.3, “Constructor functions” on page 255, the example cover functions lack input validation. The floating-point types of the coordinate arguments can take on any value, but not all values are legitimate coordinates. The toolkit’s point constructors (`createPoint64()` and others) do perform a simple domain check on the input coordinates, as does `destinationPoint()` on the input heading and distance. This, along with the error message from the exception that is raised when a value is out of range, is summarized in Table 10-4 on page 262.

Table 10-4 Floating-point argument domains and error message when out of range

Argument	Domain	Error message
latitude (°)	[-90.0, 90.0]	Input latitude out of bound. Valid range is -90 to 90.
longitude (°)	[-180.0, 180.0] ^a	Input longitude out of bound. Valid range is -180 to 180.
heading (radians)	[0, 2 π) actually: [0, 6.283185307]	Input radian out of bound. Valid range is 0 to 6.283185307.
distance (m)	[0, ∞)	Input distance out of bound. It should be more than 0 ^b

- a. In most systems, longitude is restricted to (-180.0, 180.0] (180° west is not allowed) to avoid ambiguous assignment of longitude to the 180th meridian.
- b. The error message is misleading: a zero distance *is* valid. There is no upper bound, because continuing to travel along a great elliptic arc for more than a single circumference is valid.

Unfortunately, relying on the validation performed by the toolkit functions is often not sufficient, as any violation raises an exception that terminates the operator's processing element (PE). This means that in real applications, where you usually must allow for the possibility of invalid input, the calling code must perform its own validation, and take a different action in case of bad values. The problem is how to signal invalid input back to the caller: SPL has no exception handling system or "null" values. One of the simplest solutions is to designate a special value to indicate "invalid," such as -999.0, that can apply to all four arguments of Table 10-4. Because you do not want magic numbers to pervade your code, a useful approach is to create a function that returns this value (you do need a function, because SPL does not have named constants with namespace scope), along with functions that test whether a coordinate or other floating-point value is valid. A sample is shown in Example 10-9.

Example 10-9 Validation support functions

```
// Invent a special "invalid" value for floating-point arguments
// Hide it in invalidXxx() functions that return the value and
// isValidXxx() functions to test against it.
float64 invalidFloat() { return -999.0; } // Magic number in one place
public float64 invalidLatitude() { return invalidFloat(); }
public float64 invalidLongitude() { return invalidFloat(); }
public float64 invalidAzimuth() { return invalidFloat(); }
public float64 invalidDistance() { return invalidFloat(); }

public boolean isValidLatitude(float64 lat)
{ return lat >= -90.0 && lat <= 90.0; }
```

```

public boolean isValidLongitude(float64 lon)
{ return lon > -180.0 && lon <= 180.0; }    // (-180,180]

public boolean isValidAzimuth(float64 az)
{ return az >= 0.0 && az <= 6.283185307; }

public boolean isValidDistance(float64 dist) { return dist >= 0.0; }

```

With these validation functions, the constructor functions from Example 10-3 on page 256 may be enhanced as in Example 10-10. It takes advantage of the “extra” longitude value of -180.0° (see footnote “a” in Table 10-4 on page 262) to construct a special point value.

Example 10-10 Geospatial type constructors with validation

```

// Longitude -180 can be used to indicate a special point value, if
// it is disallowed by isValidLongitude(). Any point on the 180th
// meridian can always be given a longitude of +180.
// Precision “64” version
public PointT invalidPoint() {return createPoint64(-90.0, -180.0);}

public boolean isValid(PointT point) {return point != invalidPoint();}

// The key type constructor for point; application must validate result
// before using it in other function calls
public PointT point(float64 lat, float64 lon)
{
    if (isValidLatitude(lat) && isValidLongitude(lon))
        return createPoint64(latitude, longitude);
    else
        return invalidPoint();
}

// Take advantage of implementation visibility
public LineSegmentT invalidLineSegment()
{
    return {start = invalidPoint(), end = invalidPoint(),
           distance = invalidDistance(), bearing = invalidAzimuth()};
}

public boolean isValid(LineSegmentT lseg)
{ return lseg != invalidLineSegment(); }

// Validation in the line segment constructor is not strictly required // but
// gives more coding flexibility.
public LineSegmentT lineSegment(PointT start, PointT end)
{
    if (isValid(start) && isValid(end))

```

```

        return createLineSegment64(start, end);
    else
        return invalidLineSegment();
}

```

If you always use these constructor functions to create geospatial values, then the only additional validation needed is any time a function argument is of type float64. This applies only to the destinationPoint() function, as in Example 10-11.

Example 10-11 destinationPoint() with input argument validation

```

public PointT destinationPoint(PointT start, float64 heading,
                               float64 distance)
{
    if (isValidHeading(heading) && isValidDistance(distance))
        return destinationPoint(Metric.Spherical, start, heading, distance);
    else
        return invalidPoint();
}

```

Geospatial arguments

The toolkit provides SPL types for only two feature types: point and line segment. Other feature types are represented as lists: a line string is a list of line segments, and a (simple) polygon is defined by a ring given as a list of points. No input validation is done on such values by the functions that accept them: mapPointToLineString() for line strings and IsContained() for polygons. The requirements for each of these geospatial values, and the function's behavior when the requirements are not met are summarized in Table 10-5.

Table 10-5 Validity requirements for geospatial arguments

Function	Feature type	Input type	Requirement	Result on invalid input
mapPointToLineString()	line string	list of line segments	Line segments connect end-to-end	Unpredictable; exception if empty
isContained()	simple polygon	list of points	Points, connected in order, form a ring; implicitly closed (need not repeat first point)	False

The behavior of IsContained() is relatively safe: if a polygon is not valid, a point or other polygon cannot be considered to be inside it. So, it might not tell you whether the polygon is valid or not in the case of non-containment, but you will

not get false positives. Moreover, writing an independent validation function without using a more general geometric operations library is difficult.

With `mapPointToLineString()`, you have to be more careful because you cannot trust the result if the list of line segments does not form a contiguous line string. And avoiding the exception that an empty line string can cause is important. This is relatively simple to validate, however; Example 10-12 shows how to do this.

Example 10-12 Line string validation function

```
public boolean isValidLineString(list<LineSegmentT> string)
{ // Empty line string is invalid
  if (string == (list<LineSegmentT> [])) return false;

  mutable PointT previousEnd = {};

  for (LineSegmentT lseg in string)
  {
    if (previousEnd != (PointT) {})
      if (lseg.start != previousEnd)
        return false;

    previousEnd = lseg.end;
  }

  return true;
}
```

10.2.8 Metric conversion functions

The toolkit contains several general-purpose conversion functions for converting traditional linear units (inches, feet, yards, and miles) to meters. These conversions are trivial; and because only three countries in the world officially use nonstandard units and such units are absent from modern geospatial data systems (although locale-based conversions are often applied at visualization time), we do not describe these functions here.

10.3 A location-based scenario: tracking vehicles

Cars, buses, and other vehicles are becoming data factories on wheels. Futuristic developments like driverless cars aside, entertainment systems, integrated smartphones, diagnostic sensors, and location devices are producing continuous flows of data that are now being collected in bulk by auto manufacturers, telecom providers, insurance companies, and public authorities. With appropriate regulation and privacy safeguards, great benefits can be derived from all this data.

As suggested in 10.1.1, “Moving objects and location-based services” on page 242, Streams is an ideal platform for dealing with this deluge of real-time data. In this section, we explore how one specific type of processing and analysis can be supported by the toolkit. Before we look at the actual use case, however, we need to set up some basics.

Note: In all code snippets in this section, source code lines containing geospatial types and functions are highlighted in **bold**.

10.3.1 A vehicle simulator

The examples shown next rely on input data representing moving vehicles. In a development setting, these might be from a file with either real data captured from the original feed, or entirely artificial data that is invented for simulation purposes. Or they might be from a live feed. We rely on one or more exported streams from one or more ingest or simulation applications, and focus on the processing, not the input mechanism. In this section, we present one possible ingest application: one that reads a data file, previously captured or generated by a simulation program. Any application that produces tuples of the same type can be substituted.

Define the following tuple type:

```
type RawLocT = rstring id, rstring time, float64 latitude,  
              float64 longitude, float64 speed, float64 heading;
```

Then, assume that you have a file in the project’s data directory (for example, named `simcars.csv`) containing records that look like Example 10-13 on page 267.

Example 10-13 Sample records from simulated locations file: id, time (seconds since arbitrary epoch start), latitude, longitude, speed (km/h), heading (°)

```
...
C127,1363818327,37.7845496,-122.3963803,46,45.671
C128,1363818327,37.7826509,-122.4003815,47,135.287
C131,1363818327,37.7894759,-122.4066017,41,261.317
C132,1363818327,37.7822139,-122.3969715,31,315.288
C133,1363818327,37.8014483,-122.4161462,53,350.877
C134,1363818327,37.7810743,-122.3972691,30,44.956
...
```

Notice that this record format involves no specific geospatial type or format; all it has are coordinates as separate floating-point numbers.

Creating a simulator that produces a steady flow of such tuples from this file involves four steps:

1. Monitor a directory for the presence of location data files: DirectoryScan
2. Read any qualifying files that appear in the directory: FileSource
3. Regulate the tuple frequency to something manageable: Throttle
4. Export the stream for consumption by processing applications: Export

The resulting application graph is in Figure 10-9; the code is in Example 10-14 on page 268. The tuple frequency, set by default to 40 tuples per second, can be set at launch time. When the data in the file runs out, simply add copies of the file to the same directory to keep the data flowing.

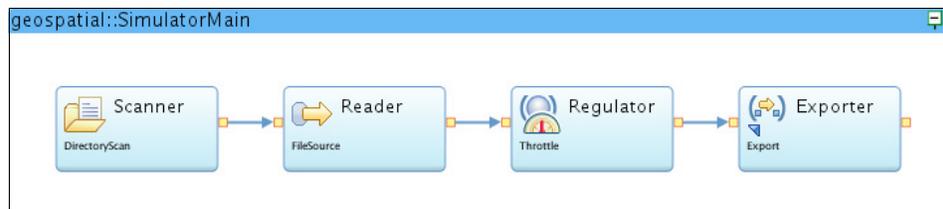


Figure 10-9 Simulator application graph

Example 10-14 Simulator application for vehicle location observations

```
namespace com.ibm.streams.redbook.geospatial;

type RawLocT = rstring id, rstring time,
  float64 latitude, float64 longitude, float64 speed, float64 heading;

composite SimulatorMain
{
  graph
    stream<rstring file> Files = DirectoryScan()
    {
      param
        directory : ".";
        pattern   : "simcars.*";
    }

    stream<RawLocT> Observations = FileSource(Files)
    { param format: csv; }

    stream<RawLocT> Throttled = Throttle(Observations)
    { param rate: (float64) getSubmissionTimeValue("rate", "40.0"); }

    () as Exporter = Export(Throttled)
    {
      param properties: {category = "vehicle positions",
                        feed = "sample file"};
    }
  }
}
```

All that is required for an application to consume this feed is to import it with a subscription that matches the exported stream's properties and type, as in Example 10-15.

Example 10-15 Importing the simulated feed

```
stream<RawLocT> RawLocations = Import()
{
  param subscription: category == "vehicle positions";
}
```

Hint: In Example 10-15, you make type `RawLocT` available either by repeating the definition from Example 10-14 on page 268, or by referring to it, which requires that the source file be in the same namespace as where it is defined, or include a use directive for that namespace. So, have either of the following lines at the top of the source file:

- ▶ `namespace com.ibm.streams.redbook.geospatial;`
- ▶ `use com.ibm.streams.redbook.geospatial::*;`

10.3.2 Geofencing: detecting entry and exit

One common process that relies on geospatial information is *geofencing*: given one or more fixed areas, detect when moving objects enter or leave that area. Commonly in geospatial applications, area features that are used in searches and filters are called *regions of interest (ROIs)*. The boundaries of the ROIs form, in effect, a virtual fence. This technique has obvious security applications, but can also be used, among other things, for traffic congestion management, emergency management (alerting or immobilizing vehicles entering a cordoned-off zone), shipping (detecting when containers leave a prescribed trade lane or storage area), and location-based marketing (to potential customers within a given distance of a store). Figure 10-10 illustrates the basic geometry of geofencing.

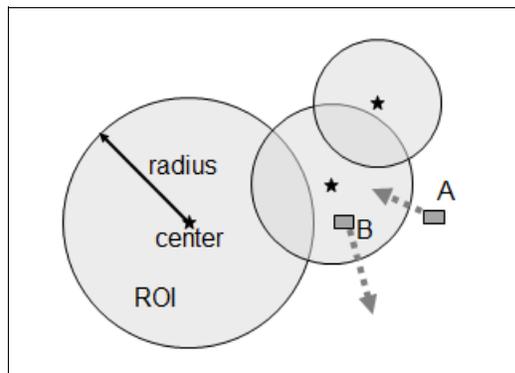


Figure 10-10 Geofencing with three ROIs; vehicle A enters the middle ROI; vehicle B leaves it

From the Streams perspective, geofencing is a special case of a more general pattern for filtering by externally defined (that is, not hard-coded) criteria; in this case, the external criteria are the definitions of the virtually fenced areas, and the filter predicate is a geospatial one.

This pattern has three main components:

- ▶ Read the criteria and maintain them in memory (most commonly, in a window) for lookup and filtering
- ▶ Apply the criteria to incoming data
- ▶ Optionally perform postprocessing to detect changes in conditions or otherwise transform results

The operator of choice for this pattern is the *Join*. It provides window management allowing for the use of anywhere from one to thousands of comparison criteria, maximum flexibility in defining the join predicate (including the use of geospatial functions). It also provides the ability to use parts of the matched criterion in the output (unlike, say, *Filter* or *DynamicFilter*). In this way, it is possible to take further action that depends on exactly which of multiple criteria was matched.

With the toolkit, geofencing can be implemented using either circular or simple polygonal ROIs. The principles and code are almost the same, but because reading in variable numbers of coordinate pairs raises the level of complexity, this example uses circles, defined by two coordinates and a radius.

Read and prepare filter criteria

Assume the ROIs are specified in a file with comma-separated values (CSV). A simple code pattern for reading file data, employing a *DirectoryScan* and a *FileSource* as in Example 10-15 on page 268, supports the ability to change the criteria at any time simply by editing the file or adding a new file to the same directory. Example 10-16 shows an ROI file.

Example 10-16 Sample ROI file: id, latitude, longitude, radius (m)

```
ROI1,37.786216,-122.409074,500
ROI2,37.791134,-122.398774,250
ROI3,37.787776,-122.40122,300
```

Two additional steps are needed:

- ▶ Convert the separate coordinates to a single point value for use in a geospatial join predicate.
- ▶ Add a file sequence number, needed for *Join* window maintenance.

Both steps are easily accomplished in a single *Functor* operator invocation. Code for this part of the application is shown in Example 10-17 on page 271.

Example 10-17 Reading and preparing the ROI criteria

```
namespace com.ibm.streams.redbook.geospatial ;

use com.ibm.streams.geospatial.twodimension.geometry::*;

composite GeofenceMain
{
  graph
    stream<rstring file> Files = DirectoryScan()
    {
      param
        directory: ".";
        pattern : "roi.*";
    }

    stream<rstring id, float64 latitude, float64 longitude,
          float64 radius> RawROIs = FileSource(Files)
    { param format: csv; }

    // Add the file sequence number to the ROI tuple. End-of-file
    // is detected by the punctuation marker produced by FileSource.
    // Also assign the center attribute as a geospatial point
    // representation of the latitude and longitude values.
    stream<rstring id, PointT center, float64 radius,
      int32 fileSeqNo> ROIs as 0 = Functor(ROIs as I)
    {
      logic
        state: mutable int32 seq = 0;
        onPunct I: seq++;
        output 0:
          center = point(latitude, longitude),
          fileSeqNo = seq;
    }
    // ...
  }
}
```

Ingest and prepare the location data

Example 10-15 on page 268 showed how to bring in simulated location data. To prepare the data for geospatial operations, first convert the separate latitude and longitude coordinates to a single point value, using a Functor. At the same time, convert the units of speed from km/h to m/s, and of heading from degrees (°) to radians. Example 10-18 shows how to apply these conversions in a Functor.

Note: Converting speed and heading units is useful if further geospatial processing is applied to these attributes. If not, leave them in the units used by the external systems providing and consuming the geospatial data.

Example 10-18 Preparing the location data

```
// ...
stream<rstring id, rstring time, PointT location,
        float64 speed, float64 heading> Locations as 0
    = Functor(RawLocations)
{
output 0:
    location = point(latitude, longitude),
    speed    = speed / 3.6,           // km/h to m/s
    heading  = heading * PI() / 180.0; // degrees to radians
}
// ...
```

Performing the geospatial join

The Join operator is powerful and complex, and this is not the place to explore all the possibilities. The key aspects of the geofencing join are as follows:

- ▶ Use a one-sided join, that is, one in which the mobile location data is not remembered (use a zero-length window).
- ▶ Keep the ROI tuples in memory indefinitely, until replaced by a new batch, as detected by a new file sequence number.
- ▶ Apply an inner join with a geospatial distance predicate that compares the distance between the mobile location and the ROI's center point to the ROI's search radius.
- ▶ Produce multiple output tuples if a given mobile location matches more than one ROI (this can happen if ROIs overlap).
- ▶ Add the ROI's ID and the computed distance to the output tuple, along with all the attributes from the mobile location tuple.

This leads to the code in Example 10-19.

Example 10-19 Join operator invocation

```
// ...
/* Maintain the ROI window for batch updates: the contents of one
 * ROI file form a batch. The arrival of the first ROI tuple
 * from the next file evicts the entire previous batch.
 * The join predicate with a circular ROI is a simple distance test.
 * Duplicate results are possible if there are multiple ROIs and some
 * of them overlap.
 */
stream<Locations, tuple<rstring idROI, float64 distance>> Matched
    as 0 = Join(Locations as L ; ROIs as R)
{
    window
        L: sliding, count(0);
        R: sliding, delta(fileSeqNo, 0);
    param
        match: distance(L.location, R.center) < R.radius;
    output 0:
        idROI    = R.id,
        distance = distance(L.location, R.center);
}
// ...
```

Hint: When the ROI is a polygon, replace the distance-based match criterion with one that is based on point-in-polygon analysis. Assume the shape argument contains a list of points defining a simple polygon:

```
match: contains(R.shape, L.location);
```

In this case, including a distance in the output tuple might not make sense.

Combining the preceding sample code fragments, the partial application graph for geofencing is depicted in Figure 10-11.

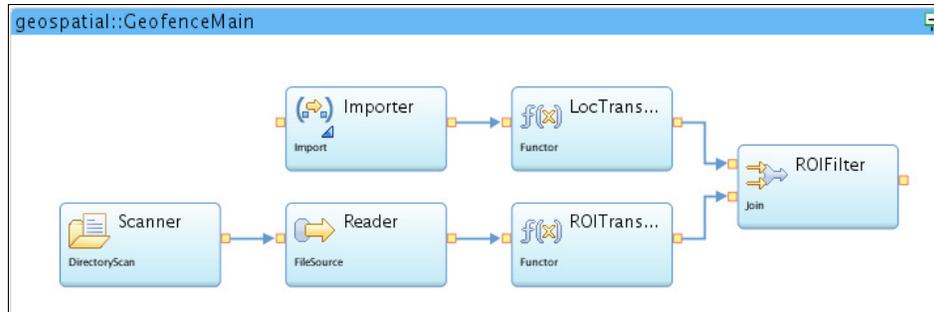


Figure 10-11 Partial geofencing application graph

Postprocessing: Fence crossings and multiple matches

The geospatial join tells you, for every tracked location, whether it is inside the fence or not. In some cases, this is enough; in others, knowing how long an object lingers inside an ROI is important. The most common requirement, however, is to detect when an object enters or leaves the ROI. For example, you might have to ensure that any vehicle gets a message only once upon entering the ROI and not repeatedly for as long as it remains inside the ROI. For this, you need a postprocessing step that detects changes in a vehicle's status.

Similarly, you might need to collapse the multiple results for a location that is inside multiple overlapping ROIs into a single result (for example, only for the ROI whose center is nearest).

Because neither of these postprocessing steps involves any geospatial functions, describing them is beyond the scope of this chapter.

10.3.3 Predictive geospatial analytics

Instead of responding to observations of current position, an application can apply more sophisticated analysis based on a model. Statistical models on numerical data might be more familiar, but similar techniques are possible with geospatial data. As a simple example, we enhance the geofencing application by not using the current position of the mobile object but a predicted one some time in the future. The "model" might be the known street map and current overall traffic speeds (in the case of vehicles) or it might be a simple extrapolation using the current position, speed, and heading (in the case of objects that are not constrained to a road network and do not change their speed and direction so rapidly, such as airplanes). In this example, we do the simple extrapolation.

All that is required to change the geofencing application from one that reacts to current observations to one that applies model-based prediction is to change the simple distance-based match condition to one that computes the predicted location and applies the distance measure to that. Example 10-20 shows this for predictions that are 30 seconds into the future:

Example 10-20 Geofencing with predicted locations 30 seconds from now

```
stream<Locations, tuple<rstring idROI, float64 distance>> Predicted
    as 0 = Join(Locations as L ; ROIs as R)
{
    window
        L: sliding, count(0) ;
        R: sliding, delta(fileSeqNo, 0);
    param
        match: distance(destinationPoint(L.location, L.heading,
                                30.0*L.speed), R.center) < R.radius;
    output 0:
        idROI    = R.id,
        distance = distance(destinationPoint(L.location, L.heading,
                                30.0*L.speed), R.center);
}
```

10.4 Conclusion

As mentioned in the introduction, the Geospatial Toolkit provides a limited set of geospatial data types and functions. This chapter devotes a considerable amount of space to explaining the concepts and principles behind it, and then shows a recommended approach toward preparing it for easy and productive use, while exploring the contents of the toolkit. Finally, the chapter explores several types and functions in the context of a specific use case.

By understanding a few simple types and functions, many powerful applications can be built that provide real value and take full advantage of the unique capabilities of InfoSphere Streams. Building on what you read here, you might envision many further applications of varying degree of complexity. Several are described here.

Traffic monitoring

Using a function such as `nearestPoint()`, from Example 10-6 on page 258, in a *Join* operator similar to Example 10-19 on page 273, “snap” the reported locations to the correct street segment. Then use an *Aggregate* to compute the number of vehicles and their average speed per street segment. This can help

detect congestion, alter the preferred routes and the associated drive times suggested by navigation systems, and more.

Real-time charging

For each vehicle, accumulate a list of street segments traveled over the course of a trip. Although the actual list aggregation is not a geospatial process, the street segment assignments can use the process described in the previous example. Toll collection and insurance premium calculation are likely to depend increasingly on this kind of actual driving pattern data.

Weather data from telematics

Using the additional telematics data often transmitted by vehicles along with GPS locations, such as *wipers-on* or *rain-detected*, the proximity of multiple vehicles reporting the same conditions can improve confidence in weather observations, such as determining if it is raining. This requires applying the `distance()` function in a similar process as described in 10.3.2, “Geofencing: detecting entry and exit” on page 269, except that the join is between moving vehicles rather than between a vehicle and a set of static locations. This is more challenging from a performance point of view, and something that few other systems can do efficiently.

Collisions and rendezvous

By analyzing locations at a point in time (present, past, or future) Streams can predict and help avoid collisions, alert on impending missile strikes, and detect whether two ships spend too much time together near the same spot at sea (possibly indicative of illegal activity). This is the realm of *spatiotemporal* computing, a field that is still developing and that requires a highly capable stream processing platform: Streams.



TimeSeries Toolkit

Streams processing is inherently the processing of time-dependent data. The TimeSeries Toolkit enhances IBM InfoSphere Streams with the capabilities to handle the time-dependent data that are numerical in nature. The toolkit has various algorithms for signal processing, time series analysis, and modeling.

A *time series* is a sequence of numerical data representing the values or measurements of an object or multiple objects over time. Time series data is pervasive across many industries and domains. It can be generated by physiological readings (electrocardiography (ECG), heart-rate measurements, or natural speech), natural phenomena (weather readings, seismic waves), or physical infrastructure (data center daily usage), or socioeconomic activity (trade volume, stock values).

The goal of the TimeSeries Toolkit is to enable the real-time analysis and modeling of time series data processing on the IBM InfoSphere Streams. The toolkit includes many operators for data preprocessing, signal processing, time series data analysis and transformation, and look-ahead and predictive analysis. The variety of operators available in the toolkit provides the means to interpolate missing data, remove noise, track changing patterns, detect or predict anomalies, forecast the future, and much more.

This chapter describes how to use the toolkit. It does so by including details about the workings of each operator, each operator's characteristics, and typical use cases. The chapter starts with by a basic course about time series to help

you become familiar with the terminology that is associated with time series analysis and modeling.

11.1 Basics of time series analysis

In this section, we provide a basic course of time series analysis. Readers familiar with time series analysis can skip to the later part of the chapter.

A *time series* is a sequence of numerical data representing the values or measurements of an object or multiple objects over time.

Time series can be regular or irregular. Regular time series values are generated at regularly spaced intervals of time. Irregular time series values are generated at arbitrary time intervals.

The following examples are of regular time series data:

- ▶ Climate data, such as hourly humidity levels, pressure, and temperature
- ▶ Yearly sunspot observations
- ▶ Financial market data, such as prices and volumes of a stock at 5-minute intervals
- ▶ Electrocardiography (ECG) and electroencephalography (EEG) readings from a patient
- ▶ Tide levels recorded nightly
- ▶ Seismograms readings recorded every 30 seconds

The following examples are of irregular time series data:

- ▶ Notes in a musical composition
- ▶ Dripping faucet

11.1.1 Time series patterns

Time series data displays a variety of patterns when plotted against time, examples of which are shown in the following list:

- ▶ Trend
- ▶ Seasonality
- ▶ Cyclicality
- ▶ Irregularity

Trends

The trend in a time series reflects the long-term alteration of the level of the data. The trend can have one of the following characteristics:

- ▶ Positive: The time series moves upward over an extended period of time, as depicted in Figure 11-1.
- ▶ Negative: The time series moves downward over an extended period of time.
- ▶ Stationary: There is a positive or a negative trend as depicted in Figure 11-2 on page 280.

Note: Detrending a time series is standard preprocessing technique to facilitate analysis. For example, for outlier detection, removing the trends can help detection performance as it reduces the variance of data. The DSPFilter operator can be used to remove the trend on a time series.

As you can see in Figure 11-1, the mean value of the time trends upwards.

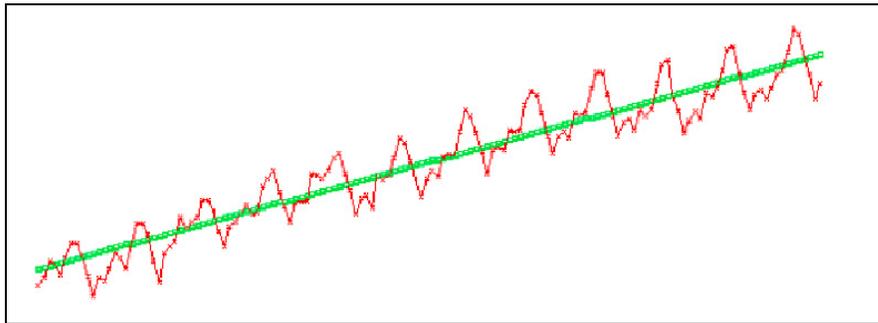


Figure 11-1 Positive trend in a time series

As you can see in Figure 11-2, the mean value of the time series evolves around a constant value.

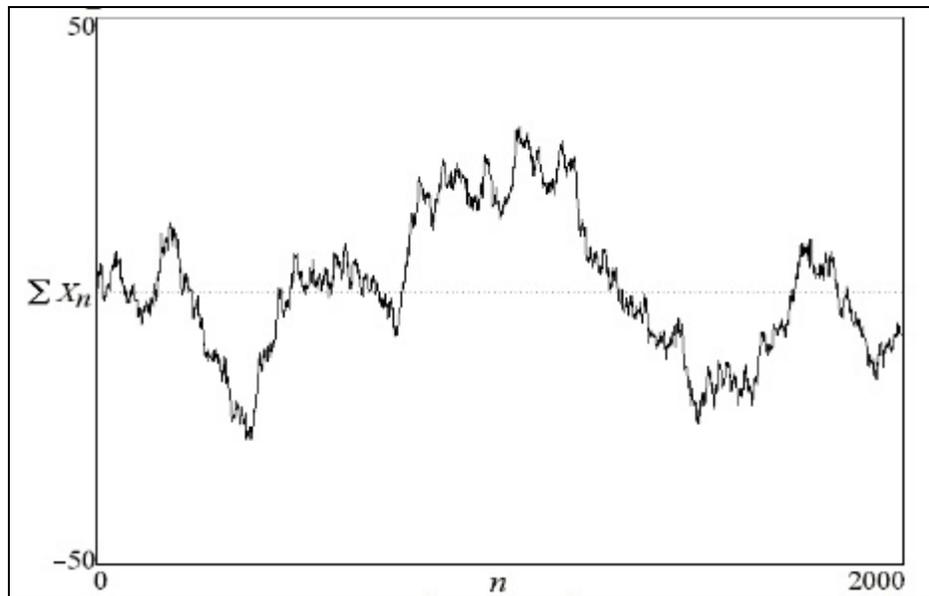


Figure 11-2 A stationary time series

Seasonality

The seasonal pattern in time series data represents “regular” data that repeats itself at the same time over a period of cycle (such as every year or every month) as depicted in Figure 11-3 on page 281.

The following examples are of seasonal patterns:

- ▶ Every year, retail sales tend to peak in the November and December time frame.
- ▶ Every year, the housing market is stronger in the spring and summer seasons compared to fall and winter.
- ▶ The average speed of vehicles on highways slows during peak hours at 8 am and 5 pm daily.

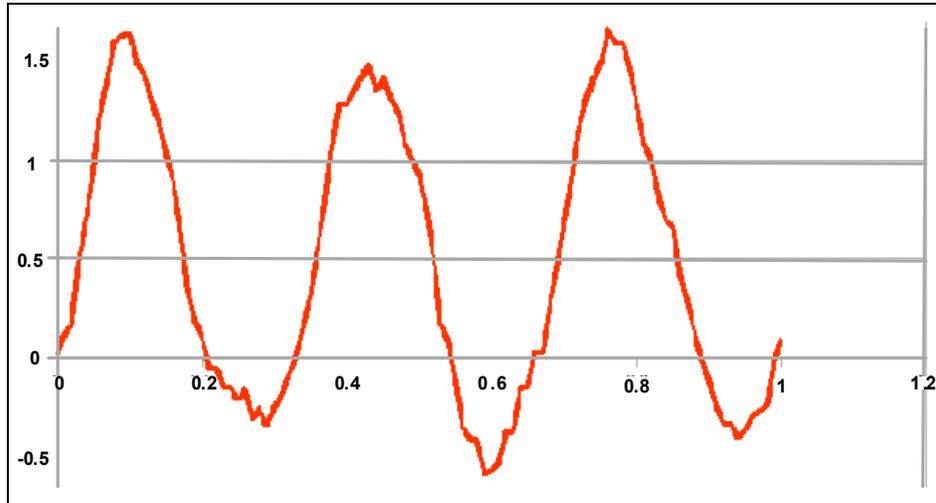


Figure 11-3 Time series with a seasonal pattern

Cyclicality

The cyclical pattern in a time series is presented in a wave, such as an upward and downward movement of the data.

Cyclical patterns are normally of longer duration and occur less regularly than seasonal fluctuations. The causes of cyclical fluctuations are usually less apparent than the seasonal variations. For example, the mineral extraction industry (mining, oil, and gas) tends to be highly cyclical (“boom and bust”).

Irregularity

An irregular pattern in a time series reflects the fluctuations that are not attributed to any of the other three patterns (trend, seasonal, cyclical).

11.1.2 Detecting patterns using the TimeSeries Toolkit

Detecting patterns is challenging in the context of streaming time series. The following guidelines can be used to determine which operators to use to estimate or remove a given time series pattern:

- ▶ Trend estimation: Trend estimation is done by tracking the evolution of the average mean value of a time series over time. This can be achieved by the following operators:
 - STD (Seasonal Trend Decomposition): The STD operator directly estimates the trend of a time series.
 - Normalize: The Normalize operator can be used to estimate the mean value of the time series overtime.
 - DSPFilter: The DSPFilter operator can be used to implement a moving average overtime.
 - FMPFilter: The FMPFilter operator produces a smooth version of the time when used with a large value of the FMPFilter's memoryLength parameter.
- ▶ Trends removal: The trend can be removed by differentiating two consecutive values of a time series. This process can easily be implemented by the DSPFilter operator (high pass filtering).
- ▶ Season/Cyclical components estimation: The STD operator can extract the seasonal component but assume a given season length.
- ▶ The FFT operator can be used to estimate underlying cycles in the time series (Fast Fourier Transformation).
- ▶ Season/Cyclical components removal: The seasonal component can be removed using the DSPFilter operator by differentiating the current value and last season value.

11.2 Time series representation and operators overview

This section describes how time series are represented in the toolkit and introduces a few key terms used to describe time series characteristics and operator's behavior.

11.2.1 Time series representation

Various time series schemas are supported in the toolkit as listed in Table 11-1. The minimal requirement is to have a numerical data within the incoming tuples. Timestamps might or might not be included. However, some operators require time series to be regular. This prerequisite can be verified only when timestamps are provided.

Table 11-1 Supported schema types for various time series types in the toolkit

Schema type of time series represented	Type of time series represented	Ingestion mode	Irregularity verification
tuple<float64 value>	univariate (single)	infinite	no
tuple<timestamp time, float64 value>	univariate (single)	infinite	yes
tuple<list<float64> values>	univariate (single)	finite	no
tuple<list<timestamp> times,list<float64> values>	univariate (single)	finite	yes
tuple<timestamp time, list<float64> values>	Vector	sample	yes
tuple<list<float64> values>	Vector	sample	no

Ingestion mode

Depending on the operator, time series data can be ingested in two ways:

- ▶ Sample-based ingestion: The operator ingests one sample at a time. A sample represents the value of a time series at one point in time. The sample can be either a single value (univariate time series) or a vector (vector time series)
- ▶ Window-based ingestion: The operator ingests a window of the data representing consecutive values of the time series.

Univariate time series

A univariate or single time series is a sequence of scalar data that represents the evolution of one single numerical variable over time. For example, a time series representing the daily temperature in New York is a univariate time series.

The supported schemas for univariate time series are depicted in Table 11-1 on page 283. Univariate time series can be represented as one of the following items:

- ▶ tuple<float64 value>
For data without timestamp and for operators with a sample-based ingestion mode.
- ▶ tuple<timestamp time, float64 value>
For data with timestamp and for operators with a sample-based ingestion mode.
- ▶ tuple<list<float64> values>
For data without timestamp and for operators with a window-based ingestion mode.
- ▶ tuple<list<timestamp> times,list<float64> values>
For data with timestamp and for operators with a window-based ingestion mode.

Most operators with a window-based ingestion mode have windowing capability, meaning that aggregation of data can be done directly by the operator from either of the following representations:

- ▶ tuple<float64 value>
- ▶ tuple<timestamp time, float64 value>

Vector time series

A vector time series is a sequence of a collected scalar data sharing the same timestamp. For example, the daily temperature and humidity level in New York are collected as a two-component vector time series. Vector time series are typically used to provide analysis of the same entity or event, seen through various perspectives and measurements.

The supported schemas for vector time series are depicted in Table 11-1 on page 283. Vector time series can be represented as the following items:

- ▶ tuple<list<float64> values>
For data without timestamp and for operators with a sample-based ingestion mode.
- ▶ tuple<timestamp time, list<float64> values>
For data with timestamp and for operators with a sample-based ingestion mode.

Expanding time series

The dimension of the time series is the number of distinct variables that are being represented by the time series. For example, a univariate time series that contains a single float64 value in each sample is a single dimension time series because it represents the evolution of a single variable. Similarly, a univariate time series that contains a window of consecutive values as `list<float64>` is a single dimension time series.

A vector time series that contains a `list<float64>` in each sample representing various variables is a multi-dimensional time series. For example, a vector time series containing a `list<float64>` of four values in each sample is of dimension 4 because it represents the evolution of four variables.

An expanding time series is a vector time series whose dimension increases over time; the number of represented variables increases over time. Support for expanding time series is available for a selected set of operators with the `maxDimension` parameter.

The `maxDimension` parameter does two tasks:

- ▶ Triggers the expanding time series mode of operation
- ▶ Specifies the maximum supported dimension of the vector time series.

For example, a `maxDimension` of 100 tells the operator to expect of vector time series of up to 100 elements. If a vector time series has more than 100 elements at some point, in this scenario, an exception is raised.

Interleaved time series

Interleaved time series represent a multiple time series with similar schema sharing the same stream of data. An example is streams of multiplexed time series of ECG value representing multiple patients.

Most operators in the toolkit support the independent processing of interleaved time series through the `partitionby` parameter. The `partitionby` parameter tells the operator to process each time series independently based on a specified attribute in the streams of data. For example `partitionby patientID` in the ECG of data will use the value of `patientID` in the incoming tuples to identify a time series data from a patient and then process each patient data independently of other patient data.

11.2.2 Control signals

A few operators in the TimeSeries Toolkit use an internal model in order to process the data. Typically those operators are used for prediction or forecasting. These operators are referred to as modeling operators (see 11.5, “Modeling operators” on page 320). A control port is an optional input port where control signals can be sent to modify the behavior of such operators including, retraining a model, loading a trained model, and suspend or resume the training of a model.

Control signals are particularly useful to synchronize the operations with the characteristics of the data or external events. For example, when the characteristics of the input data that were used during the training cycle to estimate the model change, or the trend of the input data drift significantly, the operator can no longer predict values accurately, and therefore a control signal can be sent to the operator to retrain the model.

In InfoSphere Streams 3.1, this feature is supported in several modeling operators such as ARIMA, LPC, DSPFilter, and VAR operators.

The control port supports the following control signals:

- ▶ **Retrain:** Use incoming time series data to rebuild the model. The current model is submitted to an optional monitor output port.
- ▶ **Load:** Initialize the model with provided model’s coefficients. If incorrect coefficients are provided, the operator logs a warning message in the log file and continues operation with available coefficients.
- ▶ **Monitor:** Write the coefficients to the optional monitor output port for later use or for further analysis.
- ▶ **Suspend:** Stop operation temporarily until a Resume signal is received.
- ▶ **Resume:** Restart operations from a suspended state.

Figure 11-4 illustrates the use of control port for sending the previously described control signals.

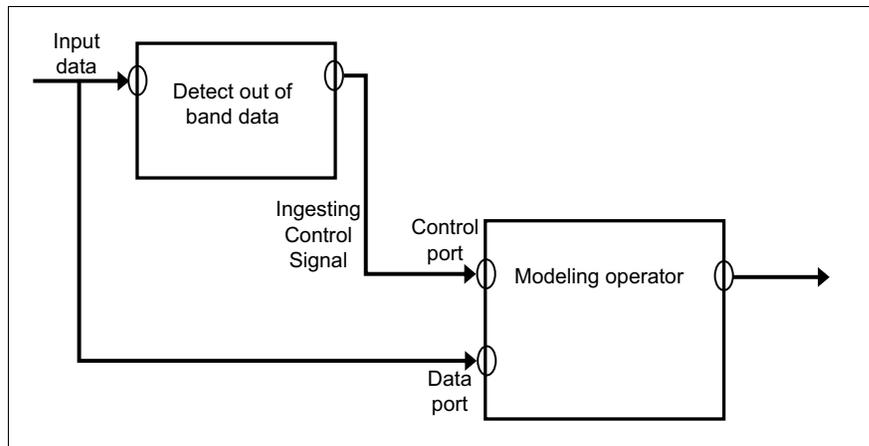


Figure 11-4 Use of control port for modeling operators

11.2.3 Types of operators and overview

The time series operators are broadly classified into two types, based on how they handle vector time series:

- ▶ Univariate operators
- ▶ Multivariate operators

Univariate operator

A *univariate operator* processes an index in a vector time series as a set of independent time series. All consecutive scalar value sharing the same index in the vector are treated as a univariate time series. The effect is similar to processing multiple univariate (single) time series in parallel, each with its own operator. The sequence of data that shares the same index in the input time series has its output at the same index in the output list.

Multivariate operator

A *multivariate operator* treats the entire vector that makes up each sample as a single object, which is transformed into an output sample. The output sample is either a vector or a scalar value, depending on the operator. Examples of multivariate operators include Kalman, Multivariate Autoregressive model (VAR), Discrete Wavelet Transform (DWT), and FFT operators.

Classes of operators

The set of operators available in the toolkit are categorized into three classes:

- ▶ Time series preprocessing: These operators deal with data conditioning for further processing and analysis. They perform tasks such as interpolating, segmenting, or resampling a time series.
- ▶ Time series analysis: These operators deal with exhibiting various aspects of time series including statistics, decomposition, or spectral transformations.
- ▶ Time series modeling: These operators create an internal model of the time series that is then used for prediction or forecasting. The model is generated either through regression analysis using a target time series or through an assumption of the time series behavior or correlations.

Table 11-2 summarizes the TimeSeries Toolkit operators, including description and usage.

Table 11-2 A summary of the time series operators

Time series generation	generate_square_wave generate_sawtooth_wave generate_triangular_wave generate_pulse_train_wave generate_sine_wave	Signal generators Generate test time series of various form and shape Good for testing and prototyping
Muti-rate processing	Resampling	Change time series sampling rate Upsample to higher sampling Downsample lower sampling
Segment analysis	TSWindowing	Time series tapering/windowing Select and weight a portion of the time series using a given shape (Hamming, Cosine, Hann, Triangle) Good for Spectral Analysis of time series segments
Interpolation	IncrementalInterpolate	Fill-in missing value incrementally over time Good for handling bad quality data handling
Interpolation	FunctionEvaluator	Estimate function values at given points Missing value estimation, bad quality data handling

Time series generation	generate_square_wave generate_sawtooth_wave generate_triangular_wave generate_pulsetrain_wave generate_sine_wave	Signal generators Generate test time series of various form and shape Good for testing and prototyping
Convolution	Convolution	Convolve two time series Good for estimating co-evolution
Correlation	CrossCorrelate	Estimate correlation between two time series Good estimate two time series co-evolution and non-co-evolution
Decomposition	DSPFilter	Digital filtering Low-pass and high-pass filter, Smoothing, noise removal, cyclic component extraction
Decomposition	STD	Break time series into season, trend, residuals Good for removing season or trend effect
Transformation	DWT	Good for time series compression, noise removal, fine-resolution analysis
Transformation	FFT	Fast Fourier Transform Spectral Analysis, short-term analysis, cycle or periodic estimation
Statistics	Distributor	Estimate histogram and quartiles Good for real-time statistics estimation
Statistics	Normalize	Normalize time series to be of zero means and unit variance Estimate means and variance Good changing the range of time series Making Multiple time series of difference change to be of the same range
Statistics	GMM	Gaussian Mixture Model (GMM) Estimate the probability of the data Estimate the anomaly in data

Time series generation	generate_square_wave generate_sawtooth_wave generate_triangular_wave generate_pulse_train_wave generate_sine_wave	Signal generators Generate test time series of various form and shape Good for testing and prototyping
Regression	GAM (Scorer and learner)	General Additive Model (GAM) Powerful regression technique Good for predictions using various input variables
Regression	RLSFilter	Recursive least square filter (RLSFilter) Good for tracking and short-term prediction Good for noise removal
Adaptive filtering Tracking	FMPFilter	Polynomial filter Smoothing, tracking and anomaly detection
Adaptive filtering Tracking	Kalman	Kalman filtering Powerful tracking technique (used in most GPS tracking device and radar) Good for prediction, anomaly detection, data smoothing
Forecasting	ARIMA	Forecasting by ARIMA Good for short-term and long-term forecasting
Forecasting	VAR	Vector Autoregression (VAR) Can predict multiple time series value using multiple inputs
Forecasting	HoltWinters	Forecasting by Holt-Winters Long-term forecasting of the time series
Forecasting Coding	LPC	Linear predictive coding (LPC) Good for time series compression, spectral smoothing, time series forecasting

11.3 Preprocessing operators

Preprocessing operators are used for preparing the input time series for further analysis.

11.3.1 ReSample operator

In signal processing, *resampling* is the process of changing the sampling rate of a time series. The sample rate is the rate at which samples are generated at the source. It is the inverse of the interval of time in seconds between two consecutive samples. It is typically measured in Hertz (Hz).

Downsampling is the process of decreasing the sampling rate of a time series. It is typically used to compress the time series into fewer elements or produce a slower time series for analysis. *Upsampling* increases the sampling rate. This can be used to synchronize two time series of different rate before processing.

The ReSample operator enables the downsampling and upsampling of a time series. As an example, consider an input time series whose samples are generated at a rate of 10 times per second (the input sampling rate is 10 Hz). The ReSample operator can double the rate to simulate a time series generated at 20 samples per second rate. Similarly, it can reduce the rate by transforming the time series into 5 times a second rate.

The SPL program to implement this upsampling scenario is shown in Example 11-1 on page 292; the example's corresponding input versus output is listed in Table 11-3 on page 293.

Characteristics of the ReSample operator

Several key features of the ReSample operator are as follows:

- ▶ The operator internally uses a polynomial function for interpolating the missing values during upsampling. The order of the polynomial function is three to locally fit a minimum of four consecutive samples.
- ▶ The operator is windowed supports tumbling window configurations; the window must have at least four tuples for the output tuples to be generated.
- ▶ Resampling is done on a per-window basis.
- ▶ Upsampling introduces some approximation to rounding affect in the polynomial interpolation as can be seen of output Table 11-3 on page 293.
- ▶ If an input timestamp is provided, the ReSample can estimate the new timestamp as a function of input timestamp.

- ▶ The ReSample operator does not support irregular time series.
- ▶ Key parameters to be set are as follows:
 - `samplingRate` as `uint32`: incoming sampling rate in Hertz.
 - `newSamplingRate` as `uint32`: new sampling rate in Hertz.

Example 11-1 shows how upsampling is done with the ReSample operator.

Example 11-1 ReSample operator at work

```
// in this example we perform upsampling of input time series from
// sampling rate of 10 //Hz to 20 Hz
use com.ibm.streams.time series.preprocessing::ReSample;
stream <list<float64> outputAudioSample,list<timestamp>
outputTimestamp> CalculateReSample = ReSample(ReadTime series)
{
  window
    ReadTime series : tumbling , count(11);
  param
    inputTime series : dataList;
    samplingRate : 10u;
    newSamplingRate : 20u;
  output
  CalculateReSample:
    outputAudioSample = reSampledTIme series(),
}
```

Table 11-3 illustrates the input and output of Example 11-1 on page 292.

Table 11-3 Input and output of Example 11-1 on page 292

Timestamp (seconds)	Input time series at 10 Hz sampling rate	Output time series at 20 Hz sampling rate
1372650000.00	562.97501	563.47501
1372650000.05		641.1903256
1372650000.10	491.78811	492.28811
1372650000.15		264.1991804
1372650000.20	103.854354	104.354354
1372650000.25		295.3167066
1372650000.30	578.620279	579.120279
1372650000.35		791.1436081
1372650000.40	933.416282	933.916282
1372650000.45		969.6786876
1372650000.50	854.176966	854.676966
1372650000.55		379.6174392
1372650000.60	2.859951	3.359951
1372650000.65		331.3783408
1372650000.70	714.036944	714.536944
1372650000.75		558.6692503
1372650000.80	303.841637	304.341637
1372650000.85		231.8444187
1372650000.90	227.339273	227.839273
1372650000.95		277.3950368
1372650001.00	365.080547	365.580547
1372650001.05		477.4646404
1372650001.10	352.1907052	

Figure 11-5 shows the input versus output plot using the ReSample operator. The input time series samples are depicted as boxes; the resampled values are depicted as diamonds.

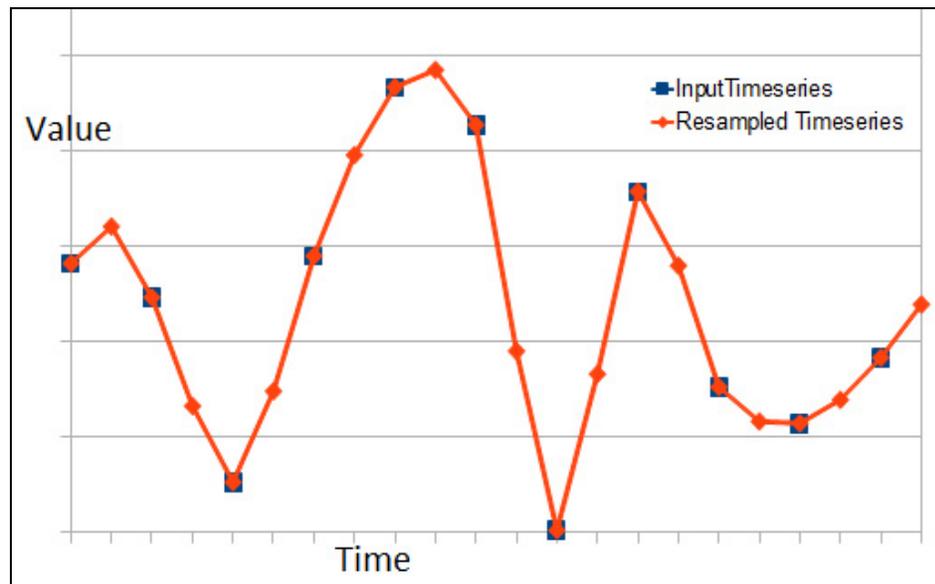


Figure 11-5 Output of ReSample producing a 20 Hz time series from 10 Hz time series

11.3.2 TSWindowing operator

Time series windowing or tapering is the process of isolating and weighting a truncation of the time series for local analysis and processing. The purpose is to attenuate the effect of the truncation when analyzing the isolated segment. Applying a weighting function that tapers to zero at the edge does this.

Various shape weighting functions are available for this purpose, each with its advantage and inconvenient. The standard Streams window is equivalent to applying a rectangular windowing function. The rectangular window is equal to 1 for each tuple in the window, and equal to 0 everywhere else.

Time series windowing is a powerful technique to help study the dynamics of a time series while applying local window-based analysis. For instance, it is widely used to analyze the evolution of a time series frequency spectrum overtime through Fourier transform.

The TSWindowing operator implements the time series windowing function.

Characteristics of the TSWindowing operator

Some of the key features of the TSWindowing operator are as follows:

- ▶ It is a windowed operator supporting tumbling and sliding window configurations.
- ▶ The operator outputs the windowed time series segment as a list.
- ▶ It needs at least 4 tuples to work.
- ▶ It supports only single time series.
- ▶ It supports a variety of windowing functions, which can be chosen using the algorithm parameter.
- ▶ The windows shown in Figure 11-6 on page 296 are features:
 - Hann window (Figure 11-6 on page 296, a), is useful for analyzing transients longer than time duration of window and for general purpose applications.
 - Hamming window (Figure 11-6 on page 296, b) is a modified version of Hann window. Hamming window does not get as close to zero near the edges as Hann window but it is widely used in speech processing.
 - The Blackman window (Figure 11-6 on page 296, c) is useful for single tone measurement.
 - The cosine window (Figure 11-6 on page 296, d) sets the samples to zero at the boundaries without reducing the gain.
 - The Triangular window (Figure 11-6 on page 296, e) a mathematical function that is zero-valued outside of some chosen interval.

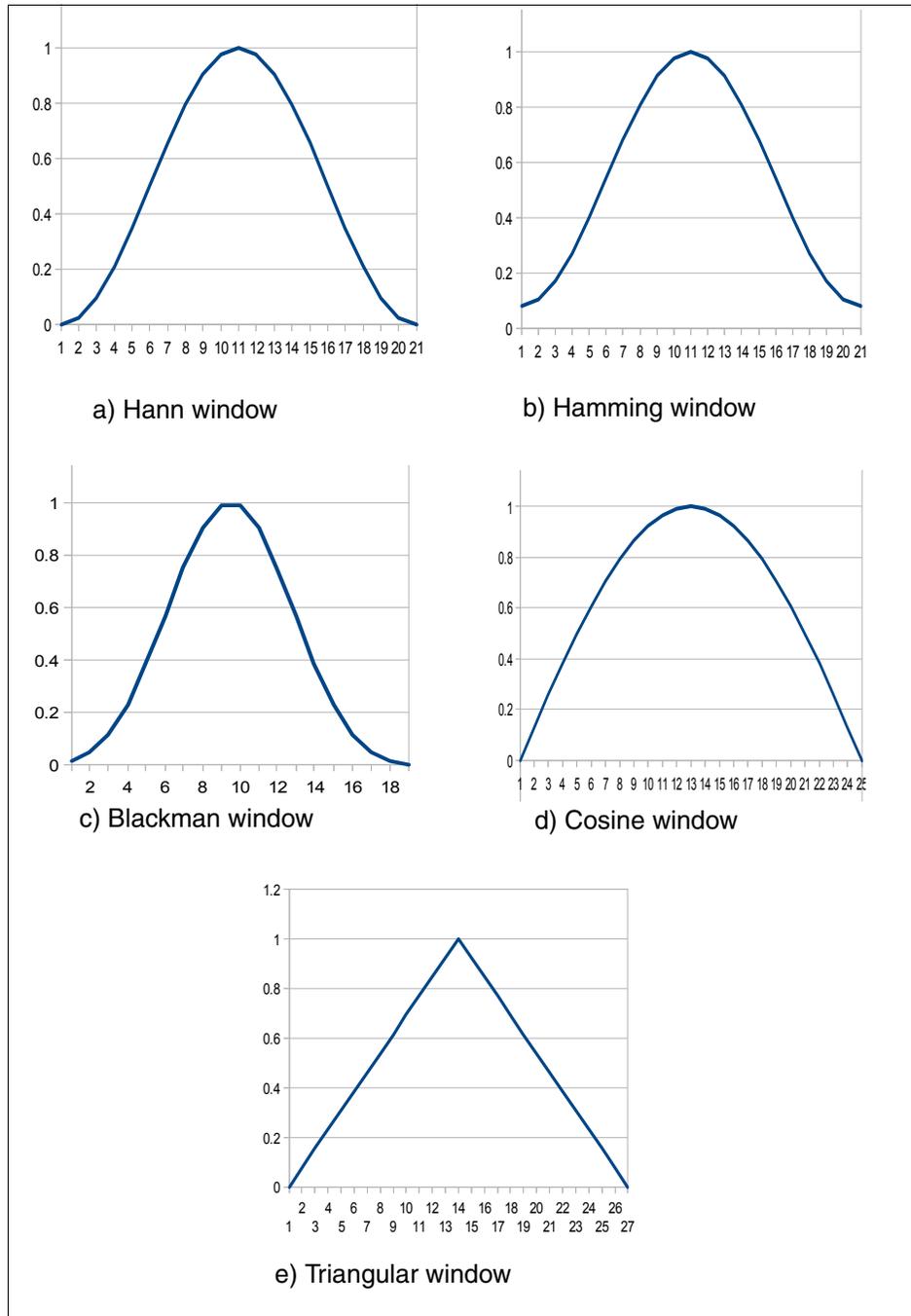


Figure 11-6 Windowing functions supported

Example 11-2 demonstrates how to apply the Hann window function using TSWindowing operator.

Example 11-2 TSWindowing operator at work

```
// in this example we apply Hann window function on input time series  
representing ECG //readings of a patient
```

```
use com.ibm.streams.time series.preprocessing::TSWindowing;  
stream <list<float64> newTS> HannStream = TSWindowing  
(rawECGStream)  
{  
  window rawECGStream:  
    tumbling,count(10);  
param  
  inputTime series : rawECGStream.ECGValue;  
  algorithm: Hann;  
  flushOnFinal: true;  
output  
  HannStream: newTS = windowedTime series();  
}
```

Figure 11-7 shows input time series and how it is transformed after applying the Hann window function.

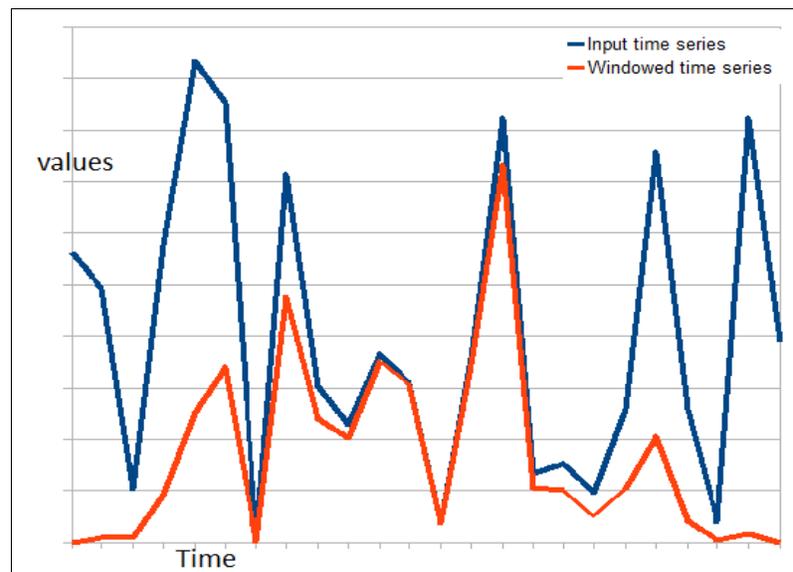


Figure 11-7 Shows how Hann window has transformed the input time series

11.3.3 IncrementalInterpolate operator

Interpolation is the process of replacing unavailable data with synthetic but representative data. It is a standard technique to deal with missing values. Missing values can be caused by glitches in the data collecting devices or have been discarded values due unwanted characteristics such a being noisy or out of range.

The IncrementalInterpolate operator replaces missing values in a time series by an estimated value. Because empty lists are not allowed in SPL, a special numerical code should be placed in lieu of the missing value. This can be done at collection time or through a preprocessing process. That missing code is then supplied to the operator by means of the `missingValueCode` parameter. The operator replaces the supplied code by interpolated values.

If the input is a vector time series, the interpolated value observed at a particular index replaces the missing value at that index. The `incrementalInterpolate` is a univariate operator because interpolation only uses data sharing of the same index in the sample.

As an example, a process identifies outlier values and replaces these values by a special code such as `-999999999.999` in the time series. By specifying the `missingValueCode` parameter as `999999999.999`, the operator will replace all values of `999999999.999` in the time series by the interpolated value. Timestamps have no influence on the output of the operator.

Characteristics of the IncrementalInterpolate operator

The `algorithm` parameter enables the choice of the following methods for interpolation:

- ▶ **Last:** This option replicates the most recent observed value. This algorithm is simple but computationally is the most efficient. It is appropriate for stable and slow-variation time series.
- ▶ **Average:** This option estimates the missing data value using a moving average method. This method is slightly more expensive computationally than the Last option but produces smoother results.
- ▶ **Predictive:** This option estimates the missing data value using a polynomial predictive filter. For more details about polynomial filter, see 11.5.3, “FMPFilter operator” on page 326. The predictive method produces most accurate results but it is computationally expensive.

The `IncrementalInterpolate` operator provides two output functions:

- ▶ The `missingDataRatio` function: This is the ratio of the number of interpolated values to the total number of samples processed, or $N_{\text{interpolated}}/N_{\text{total}}$.
- ▶ The `interpolatedTime series` function: Returns the interpolated time series.

`IncrementalInterpolate` operator usage depicts the `Incremental` operator usage, using the last algorithm and retrieving missing data statistics during run time. See Example 11-3.

Example 11-3 IncrementalInterpolate operator usage

```
//Incremental operator used with last algorithm
stream <list<float64> KPIs, list<float64> missingRatio>
    interpolatedTimeSeriesOut = IncrementalInterpolate
(TimeSeries)
{
    param
        inputTimeSeries: KPIs;
        algorithm       : last;
        missingValueCode : -999999999.9;
    output
        interpolatedTimeSeriesOut : KPIs = interpolatedTimeSeries(),
        missingRatio = missingDataRatio();
}
```

11.4 Analysis operators

Time series analysis, in the context of this toolkit, refers to the process of gathering information about the behavior of the time series. This includes gathering time series statistics, decomposition of the time series into underlying constituents, or performing transformations to have a view of some unseen aspect of the data.

Various analysis operators are currently available in the `TimeSeries Toolkit` including operations to perform signal processing algorithms, such as digital filtering, Fourier or Wavelet Transform, operations to correlate or convolve time series, and operations to estimate the time series statistics such means, median, or histogram.

11.4.1 DSPFilter operator

Digital filtering is a signal processing technique whose goal is to attenuate or amplify certain aspect of the time series. For example, digital signal processing can be used to attenuate high-varying components to generate a smoother time series, or it can be used to remove slow-varying components, such as long-cycles, or to exhibit the un-periodic components in the time series. This process is done in the time domain (the input time series itself). The digital filter transforms inputs into outputs using the formula depicted in Figure 11-8. For an input, the output is the sum of two components: a weighted sum of the input data and weighted sum of the past output data, From an input time series sample $x(n)$, at time n , produces an output time series sample $y(n)$. The output is a weighted sum of current and past inputs up to a lag, and weighted sum of past outputs up to a lag.

$$y(n) = \underbrace{\sum_{k=0}^{M} b(k)x(n-k)}_{\text{Linear combination of input current and (optionally) past inputs}} - \underbrace{\sum_{k=1}^{N} a(k)y(n-k)}_{\text{Linear combination of past outputs}}$$

Figure 11-8 The DSPfilter formal

A digital filter is completely specified by its coefficients: the set of weights for current and past inputs, the $b(k)$, and the set of weights for past outputs, the $a(k)$. Note that the weights of the current output, $a(0)$, is always 1.0.

A digital filter introduces a phase change in the output. Digital filter design is a well-studied area in signal processing.

Characteristic of the DSPFilter operator

The DSPFilter operator performs digital filtering. The DSPFilter operator can be used for the following tasks:

- ▶ Perform noise removal through data smoothing as depicted in Figure 11-9 on page 303.
- ▶ Eliminate seasonal variations present in the input time series. For example during anomaly detection, it is important that seasonal variations be removed to avoid false alarms.
- ▶ Extract cyclical components in the time series, such season variations.

- ▶ Estimate the trend in the time series by performing moving average
- ▶ Detrend a time series to generate a stationary version of the time series for further analysis.

The main features of the DSPFilter operator are as follows:

- ▶ It is a univariate operator: each time series index is filtered independently of other indices.
- ▶ It supports single time series and vector time series.
- ▶ It supports partitionBy parameter and expanding time series.
- ▶ Filter coefficients can be provided by the user or be estimated automatically. Automatic estimation of a filter's coefficients are provided only for standard 1-order high pass and low pass. Consider the following information:
 - Coefficients have to be passed using xcoef and ycoef parameters, which are of type `SPL::map<uint32, float64>`. The key is the lag (the index k) and the value is the coefficient. Indexing starts at 0.
 - The xcoef parameter represents $b(k)$ coefficients in the equation in Figure 11-8 on page 300. The value of `xcoef[k]` is $b(k)$.
 - The ycoef parameter represents $a(k)$ coefficients in the equation in Figure 11-8 on page 300. The value of `ycoef[k]` is $a(k)$ with `ycoef[0]` always equal 1 (explicitly stated).
 - If the coefficients are not available, you can set the filter Type parameter to either high pass or low pass. You must provide the samplingRate and cutOffFrequency parameters to facilitate the automatic estimation of coefficients. Example 11-4 shows various values that can be provided to xcoef and ycoef parameters for various standard filter types.

Example of DSPFilter usage

This section illustrates the coefficients setting for several typical usage samples of the DSPFilter.

Various samples of digital filters are illustrated in Example 11-4.

Example 11-4 Multiple examples of digital filter

- ▶ Examples of lowpass (smoothing) filter:
 - Simple 4-lag moving average used for removal of random noise:


```
xcoef:{0u:0.25, 1u:0.25, 2u:0.25,3u:0.25}
ycoef:{0u:1.0}
```
 - Simple 2-lag weighted moving average with triangular weight:


```
xcoef:{0u:0.75, 1u:0.5, 2u:0.25}
```

```
ycoef:{0u:1.0}
```

- Implementation of exponential smoothing, widely used for smoothing and prediction: $y(t) = ax(t) + (1 - a) y(t-1)$ (with $a=0.75$):

```
xcoef:{0u:0.75}
```

```
ycoef:{0u:1.0, 1u:-0.25}
```

► Examples of high-pass filter

- Simple detrending coefficients to create a stationary time series (first derivative):

```
xcoef:{0u:1.0, 1u:-1.0}
```

```
ycoef:{0u:1.0}
```

- Remove seasonal of component made of 100 samples per season:

```
xcoef:{0u:1.0, 99u:-1.0}
```

```
ycoef:{0u:1.0}
```

Example 11-5 illustrates a simple DSPFilter code that performs moving average.

Example 11-5 DSPFilter operator at work

```
//DSPFilter used for removing random noises by performing moving
average
stream <rstring Date, float64 Open, float64 MAOpen> filteredStockStream
=DSPFilter(stockStream)
{
param
  xcoef:{0u:0.2,1u:0.2,2u:0.2,3u:0.2,4u:0.2}; //moving average
  ycoef:{0u:1.0};
  inputTime series: Open;
output
  filteredStockStream:
  MAOpen=filteredTime series();
```

Figure 11-9 depicts the plot of input time series against the filtered output of Example 11-5 on page 302.

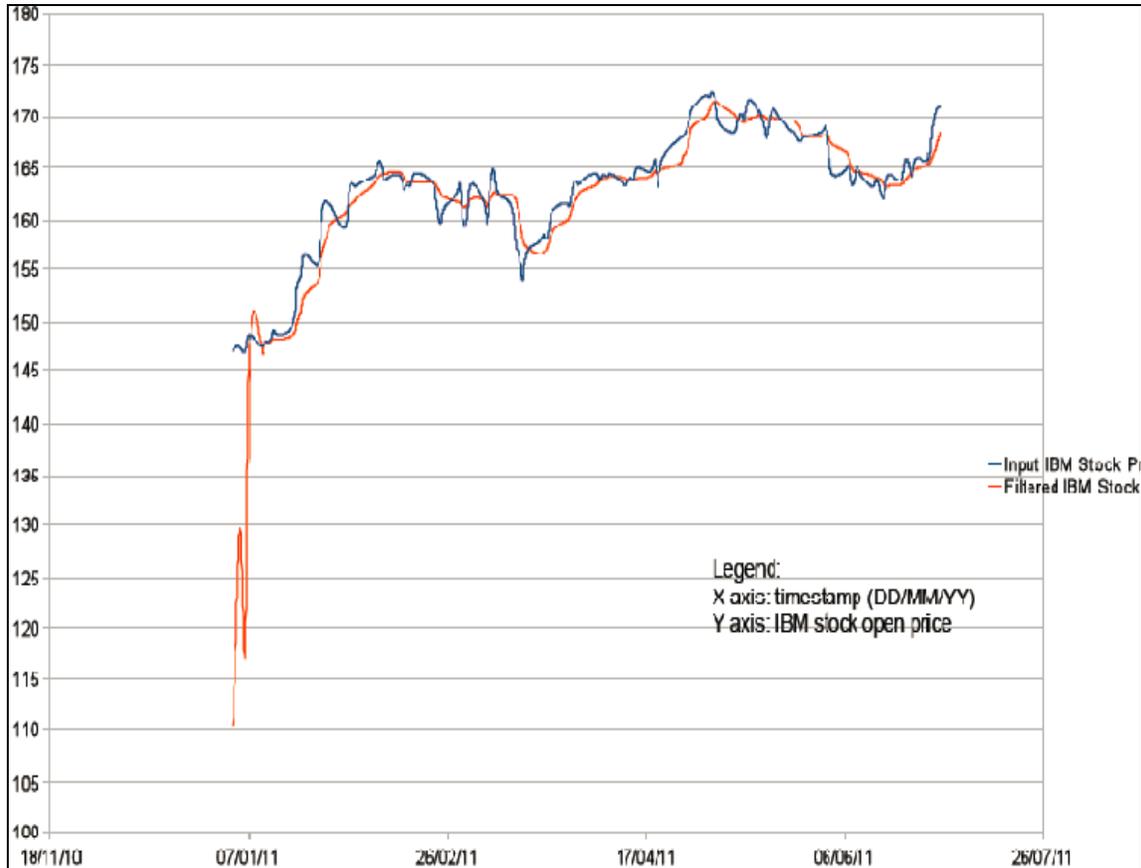


Figure 11-9 Shows data smoothing performed using DSPFilter operator

11.4.2 Fast Fourier Transform

Any time series is made of sine waves of various frequencies. For a particular time series, the number of its sine waves can be finite or infinite. The Fourier Transform is a signal processing technique used to estimate the number of sine waves that comprises a time series. This is usually referred to as spectral analysis and the result is a set of complex-valued sine waves, representing frequencies, amplitudes, and phases that comprise the time series.

Many algorithms are available to perform spectral analysis. The Discrete Fourier Transform(DFT) is a computational technique widely used to perform spectral analysis on a finite set of data. The DFT transforms a sequence of N input time

series samples into another sequence of N complex-valued samples using a matrix multiplication with predetermined DFT coefficients (complex-valued). It is customary to apply a time series windowing, such as Hamming windowing, on the data prior to DFT transform so as to minimize segmentation effect on the results (for details about windowing see 11.3.2, “TSWindowing operator” on page 294).

The Fast Fourier Transform (FFT) is a computationally efficient way to compute the DFT. If the number of samples is a power of 2, then the FFT makes use of the underlying symmetries in the DFT coefficients to increase speed of the DFT computation. This is the radix-2 FFT algorithm. When the number of samples is not a power of 2, it is customary to add a sequence of zeros at the end of the sample set to realize a power of 2 number. This process is called *zero-padding* and has minimal effect on the result of the spectral analysis.

Characteristics of the FFT operator

The FFT operator implements spectral analysis and makes available, on Streams, various algorithms for DFT and FFT. The FFT operator has the following features:

- ▶ It is a multivariate operator
- ▶ It expects either a vector time series or a single time series with a window parameter:
 - The vector time series can represent sample of data from a single time series gathered over a segment of time, or data from a multiple time series, where all values in the list are sharing the same time stamp.
 - A single time series can analyze by the FFT using a window parameter.
- ▶ The algorithm parameter enables the choice of varieties of the DFT implementations. The supported algorithms are as follows:
 - ComplexFFT: Transforms the input time series to a complex number recording both magnitude and phase information and then perform spectral analysis using radix-2 FFT on the complex data.
 - realFFT: Implements an efficient FFT algorithm for real values on the provided sample data, producing real spectrum. This is the default mode of the FFT operator and the fastest.
 - realDFT: Implements the direct DFT algorithm using direct method producing only real component of the spectrum; the imaginary component is discarded. Not zero-padding is done.
 - inverseComplexFFT: Implements the inverse process of the complex FFT producing complex numbers as output.
 - inverseRealFFT: Implements the inverse process of the realFFT producing only real numbers as output.

Table 11-4 lists output functions that the FFT operator supports.

Table 11-4 Custom Output Functions provided by FFT operator

Output function	Description
FFTsAsComplex()	Returns the FFT as list of complex64.
magnitude()	Returns the magnitude spectrum as list<float64>.
power()	Returns the power spectrum as list<float64>.

Example 11-6 illustrates how to extract frequency components of an input ECG data using the FFT operator.

Example 11-6 Extracting the magnitude spectrogram using FFT operator

```
//Given an ECG input time series FFT can be used to convert it to
frequency domain to //extract power and magnitude components from ECG
values which can give more //insight about a patient
use com.ibm.streams.timeseries.analysis::FFT;
stream<list<float64> magnitude,list<float64> power> spectrogramStream =
FFT
(sineStream) {
window
    sineStream:
    sliding, count(128), count(10);
param
inputTimeSeries: sine;
useHamming: true;
resolution:128u;
algorithm: realDFT;
output
spectrogramStream:
magnitude=magnitude(),
power=power();
}
```

Figure 11-10 on page 306 FFT spectrogram computed from a time series that is made of the two sine waves: a sine wave at 20 Hz and a sine wave at 50 Hz.

Similarly, Figure 11-10 on page 306 shows the output of the FFT operator when processing a time series composed of two sine waves at 50 and 20 Hz. The output clearly depicts the presence of frequency components at 50 Hz and 20 Hz. The multiples curves represent the magnitude of frequency components corresponding to sliding window positions across the time series.

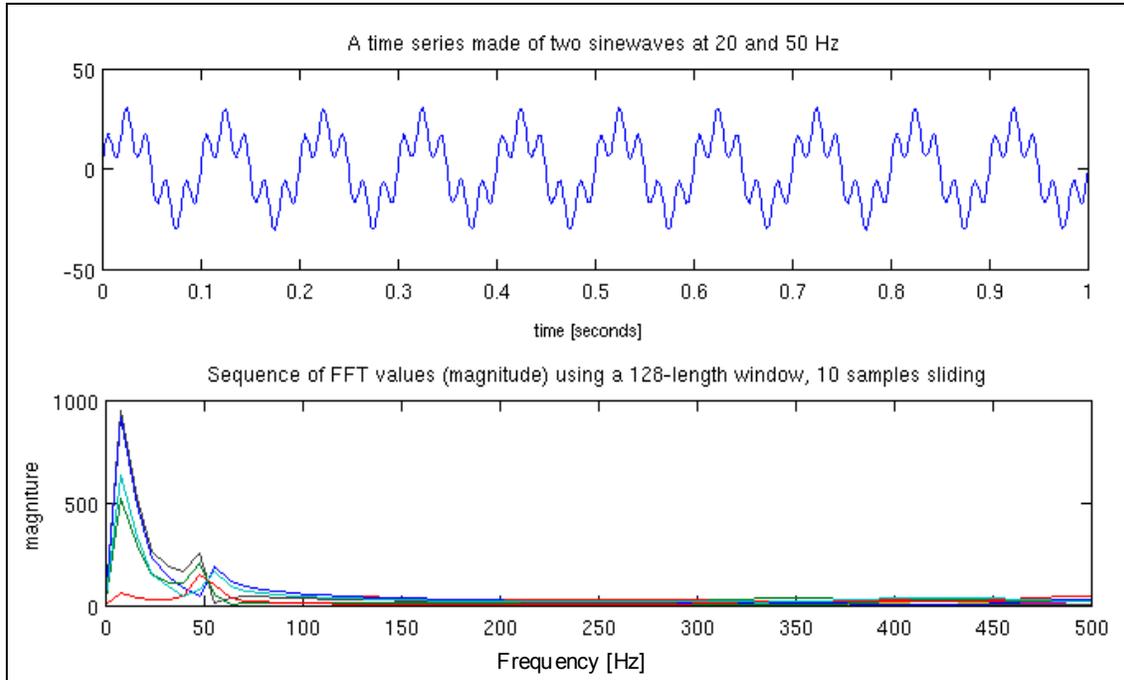


Figure 11-10 FFT spectrogram computed from a time series of two sine waves: 20 Hz and 50 Hz

11.4.3 Discrete Wavelet Transform operator

Wavelet stands for small waves. More precisely, a wavelet is a type of signal that has a short duration, is localized at some point in time with amplitude that tapers off at the edge. Wavelet analysis is the process of decomposing the time series into its wavelet components, similar to the way the Fourier transform decomposes a time series into its sets of sine waves. However, unlike a sine wave that has an infinite duration across time, a wavelet has a finite duration and it is well localized at some point time. This allows the wavelet to have a better time and frequency resolution than the Fourier Transform.

A better time-frequency resolution explains the wide use of the wavelet transform in detecting pulses or blips or any event that happens at a particular point in time. It also enjoyed a widespread use in the image compression.

The performance of the wavelet transform depends on the type or shape of the wavelet. The standard choice is the Daubechies wavelet, which is the one used in the TimeSeries Toolkit.

The Discrete Wavelet Transform (DWT) is a computational technique that transforms a set of N samples into a set of N wavelet coefficients, where the

lower-indices coefficients are approximation or aggregate representation of the time and higher indices are details of time series. Consequently, the DWT can exhibit fine-details of the time series or can provide powerful approximation or compression capabilities.

Advantages of DWT are as follows:

- ▶ It provides more efficient time and frequency localization than FFT.
- ▶ The output contains fine-grained details of input time series for example the first indices of the transform contains the average (trend) of the signal, the higher indices contains the most detailed information about the time series.
- ▶ Time series compression means time series can be represented by a few coefficients of lower indices.
- ▶ Smoothing of time series by removing higher indices coefficients

Characteristics of the DWT operator

The features of DWT operator are as follows:

- ▶ DWT is a multivariate operator that expects either a vector time series or a single time series with window parameter.
- ▶ Supported algorithms are Daubechies wavelet of order 2 (also known as Haar wavelet) and order 4.
- ▶ DWT requires the size of the input vector to be a power of 2. Zero-padding is performed internally when the size of the input is not a power of 2.

Figure 11-11 on page 308 illustrates the compression capabilities of the DWT operator. It depicts 3 time series graphs. The input time series is a noisy sine wave transformed using DWT with Daubechies 4. The two outputs are obtained by applying the inverse DWT on just 16 lower indices (bold curve) and the 8 lower indices.

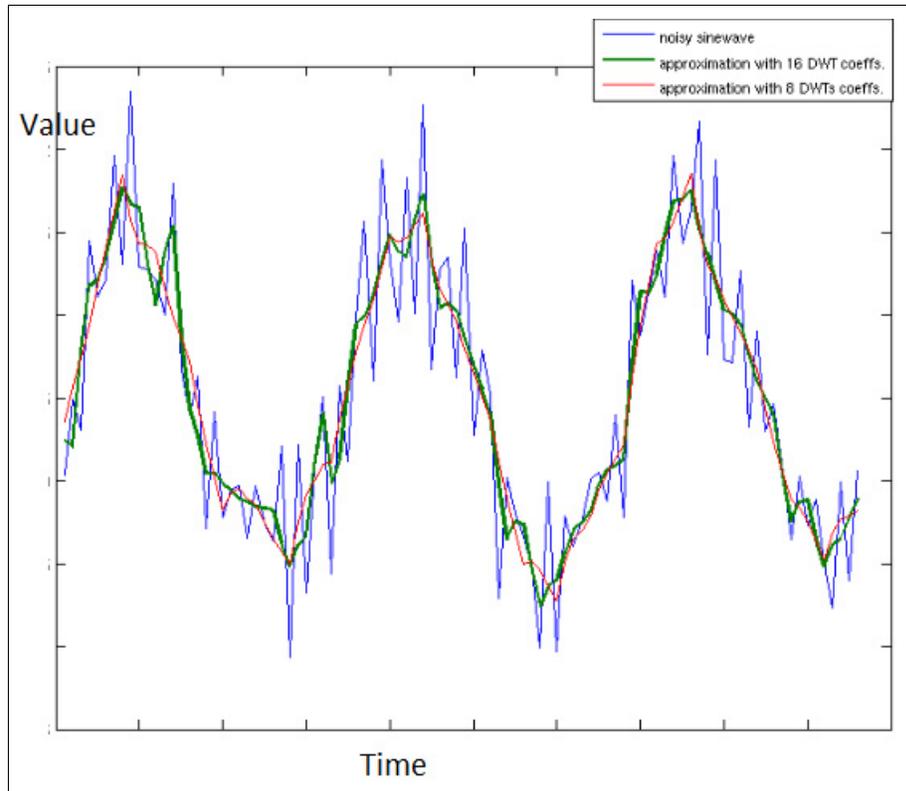


Figure 11-11 The amazing compression properties of the DWT

Example 11-7 shows DWT operator usage for performing transformation of various parameters which need to be set.

Example 11-7 DWT operator usage for compression

```
// do DWT transform on sequence of time series
stream<list<float64> sineWavelet> sineWaveletStream2 = DWT
(sineStream2) {
window
sineStream2:
tumbling, count(32);
param
inputTimeSeries:sineWave;
order:four
output
sineWaveletStream2: sineWavelet=DWTTransform();
}
```

11.4.4 Seasonal Trend Decomposition operator

The Seasonal Trend Decomposition (STD) operator decomposes a time series into three constituents: trend, season, and irregular component.

The STD assumes two models for the time series behavior as illustrated in Table 11-5.

- ▶ Additive model where the time series is sum of the trend, season, and the irregular component
- ▶ Multiplicative model: the season time series is the product of season to the trend and the irregular component

Table 11-5 Additive and multiplicative model of the STD algorithm

Model	Constituents
STD additive	$x = \text{factor} * (\text{trend} + \text{season} + \text{irregularity})$
STD multiplicative	$x = \text{factor} * \text{season} * (\text{trend} + \text{irregularity})$

The current implementation of the algorithm assumes a given length of the season, which is used as the basis for estimating the trend and the residuals.

Characteristics of the STD operator

The following steps describe how to use the STD operator:

1. The STD does not estimate the season length. Provide the length of season in the number of samples. The season can represent any time period such as an hour, a day, a week, or a month. Use the `samplesInSeason` parameter.
2. Provide an optional window configuration when the input is a single time series. However, only the tumbling window configuration is supported.
3. Provide the type of season model, multiplicative or additive, using the algorithm parameter as described in Table 11-5.

The operators provide three output functions:

- ▶ The `season()` function: Returns data for one season (a shape).
- ▶ The `trend()` function: Returns the trend over time.
- ▶ The `residuals()` function: Returns the residuals or irregular part of the (time series without trend and season).

Example 11-8 on page 310 illustrates how the STD operator is used to extract the trend and season information from input.

Example 11-8 STD operator usage

```
// In this example we use STD operator to extract season and trend of
// airline passenger count.
stream<list<float64> ts,list<float64> season,list<float64>
trend,list<float64> residual> STDOut = STD(fileIp)
{
  param
    inputTimeSeries:passenger_count;
    samplesInSeason:12;
    algorithm:Additive; // assume that trends and season are additive
  output
  STDOut:
    season=season(),
    trend=trend(),
    residual=residuals();
}
```

Figure 11-12 depicts the output of STD operator (with an estimated trend and seasonal component given an input time series) when in additive mode; the seasonal component detected is plotted as a curve close to y-axis. It shows that the increasing trend has not affected the season produced in output; the multiplicative seasonal component has grown with the trend.

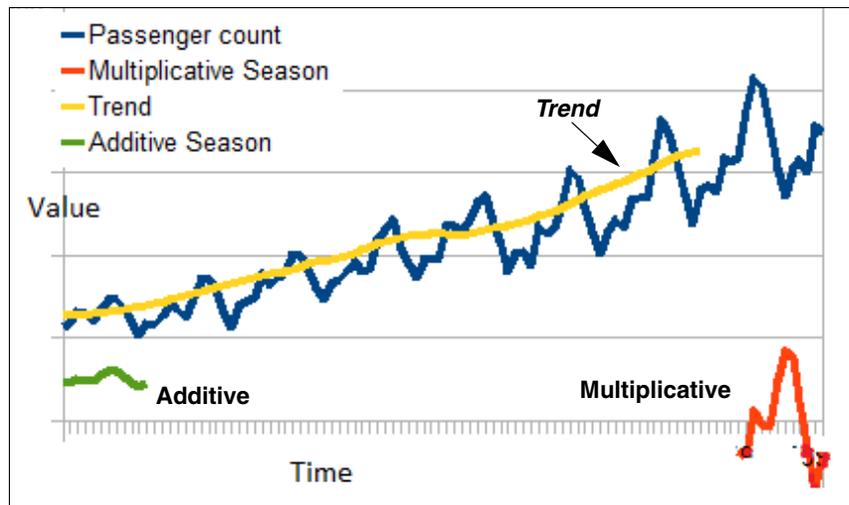


Figure 11-12 Estimated trend and seasonal component

11.4.5 CrossCorrelate operator

In signal processing, cross-correlation is a measure of similarity of two time series as a function of a time-lag. Cross-correlation is also known as a sliding dot product or a sliding inner-product,

Consider two lists of time series, x and y . The CrossCorrelate function is defined by the formula in Figure 11-13. In this formula, x and y are two input time series windows. An equivalent way to perform this computation is through Fast Fourier Transformation (FFT).

$$\begin{aligned} x &= [x(0), \dots, x(M)] \\ y &= [y(0), \dots, y(N)], \quad N \geq M \\ r(i) &= \sum_{j=0}^{M-N-i} x(j)y(i+j), \quad 1 \leq i \leq N-M+1 \end{aligned}$$

Figure 11-13 CrossCorrelate function

Characteristics of the CrossCorrelate operator

The CrossCorrelate operator has the following characteristics:

- ▶ It has two input ports, both are windowed.
- ▶ It inserts tuples from each input port into the corresponding window. When both windows are full, the CrossCorrelate operator calculates the cross correlation between the tuples by using either the Fast Fourier Transform (FFT) or by using the standard cross-correlation formula. The method is specified by the algorithm parameter.

Example 11-9 illustrates a typical usage of the CrossCorrelation operator, using a different window size.

Example 11-9 CrossCorrelation performed on timeseries ingested from two input ports

```
stream <list<float64> correlatedTimeSeries> Out1 = CrossCorrelate(ts1; ts2) {
  window
    ts1 : tumbling,count(19);
    ts2 : tumbling,count(3);
  param
    inputTimeSeries : ts1.data,ts2.data;
    algorithm       : FFT;
  output
    Out1: correlatedTimeSeries = crossCorrelateTimeSeries();
}
```

Figure 11-14 illustrates output of the CrossCorrelate operator as a function of time, showing the time of maximum correlation between the two time series.

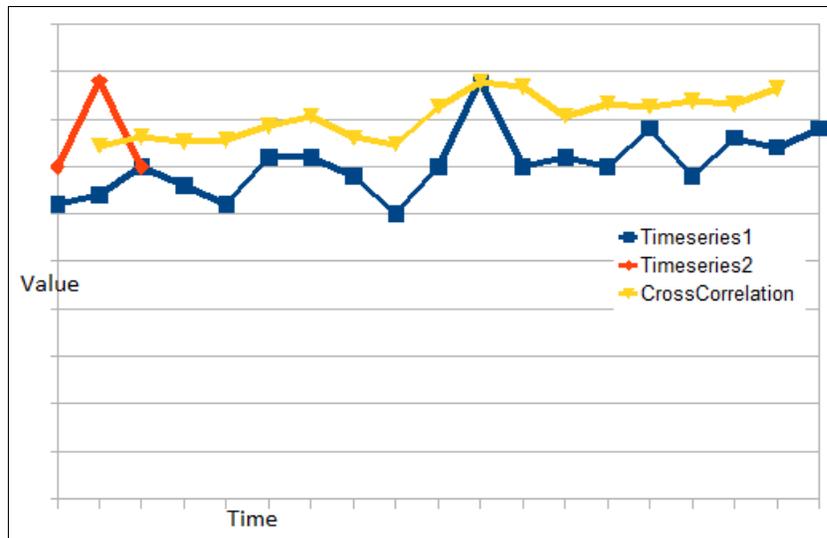


Figure 11-14 Cross-correlation of two input time series samples as a function of time

11.4.6 Normalize operator

Time series normalization is the process of scaling a time series value to have zero mean and unit variance. This is typically desirable when the user wants to reduce the range of the time series for better post-processing or when scaling multiple time series to be of the same range. The time series pattern remains unchanged; only the range of values are affected. The range is the difference between the maximum values and minimum values.

The toolkit version of the algorithm applies normalization incrementally to the streaming. It also incrementally estimates the mean and variance of the time series (single time series or vector time series) either indefinitely or up to a user-specified duration. The real-time estimate of the mean and variance is done for each new input x , as illustrated in Figure 11-15 on page 313.

$$\text{Mean update at time } n : \mu_n = \mu_{n-1} + \frac{1}{n}(x_n - \mu_{n-1})$$

$$\text{Variance update at time } n : \sigma_n^2 = \frac{S_n}{n}$$

$$\text{with } S_n = S_{n-1} + (x_n - \mu_{n-1})(x_n - \mu_n)$$

$$\text{The normalized time series : } y_n = \frac{1}{\sigma_n}(x_n - \mu_n)$$

Figure 11-15 Time series normalization

Characteristics of the Normalize operator

The normalize operator uses the equation to normalize a time series in real time. The normalize operator has the following characteristics:

- ▶ It is a univariate operator; each index of the vector time series has its own processing algorithm.
- ▶ It supports single and vector time series.
- ▶ It support partitionBy parameter.
- ▶ The incremental estimate of the mean and variance is done for each incoming tuple, up to a specified internal as provided by the initTime parameter. If initTime is not provided, then means and variance are estimated indefinitely for all incoming tuples.

Table 11-6 lists the output functions of the Normalize operator. The output functions enable the user to obtain the normalized time series and the estimated mean and variance of the time series, as seen so far. (Note that these are the estimated mean and variance of all data seen so far.)

Table 11-6 Output functions available in Normalize operator

Output function	Description
variance()	Returns the estimated variance so far
normalizedTimeSeries()	Returns the normalized input time series
means()	Returns the estimated means so far

Example 11-10 illustrates the use of the normalize operator. The first 14 samples are used to estimate the mean and variance (initSamples set to 14u).

Example 11-10 Normalize operator at work

```
// normalize the each time series to zero means and unit variance
stream<uint64 timepoint,list<float64> normalizedTS,list<float64> inpTS>
normalizedStream = Normalize(InpTS)
{
    param
        initSamples:240u;
        inputTimeSeries: inpTS;
    output
        normalizedStream: normalizedTS=normalizedTimeSeries();
}
```

Figure 11-16 illustrates the functionality of Normalize operator. The input time series represents statistics from the data center memory usage and the output is the normalized memory usage.

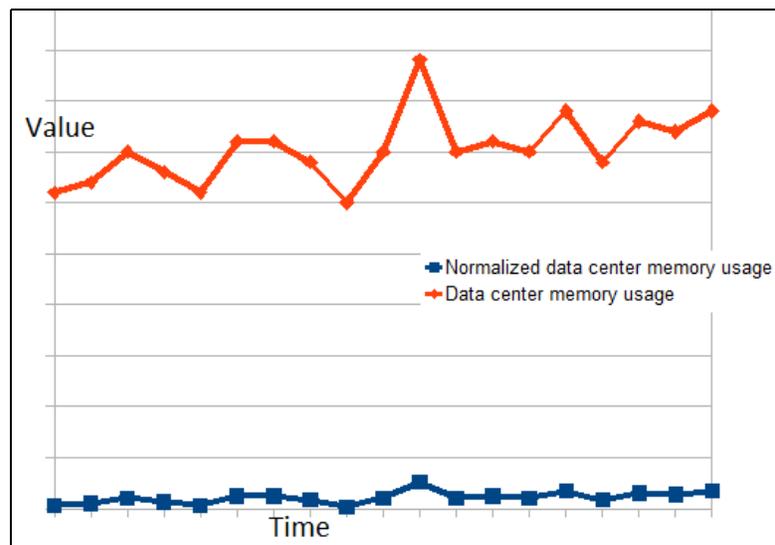


Figure 11-16 Normalized time series and its corresponding unnormalized input

11.4.7 FunctionEvaluator operator

The FunctionEvaluator operator estimates non-specified values of a function. You provide known values of the function (control-points) for a specified set of the input variables (knots). The FunctionEvaluator is then able to estimate values of

the function for unspecified input variables. Furthermore, the FunctionEvaluator can provide a mapping from a one-knot to several of control points, basically estimating multiple functions at once.

FunctionEvaluator can be used to interpolate missing values or estimate the value of the shape of the time series in real time.

The FunctionEvaluator internally uses linear interpolation to estimate missing values.

Characteristic of the FunctionEvaluator operator

The control and knot points are specified by means of the functionSpecification parameter, which is of type map. The user specifies a list of control points (values) per knot (key), which means evaluating a vector time series with one control per index.

The functionSpecification parameter has the following format; the parameter provides mapping from one knot to a series of control points:

```
functionSpecification:
{
<knot_1>:[<control_11>, <control_12> ... <control_1d>],
<knot_2> : [<control_21>, <control_22>... <control_2d>],
...
<knot_n>:[<control_n1>,<control_n2>... <controlling>]
}
```

Example 11-11 depicts use of FunctionEvaluator operator for estimating missing values of one single time series. The evaluateTimeseries () output function is used for extracting the evaluated values.

Example 11-11 FunctionEvaluator operator usage

```
// This example illustrate how the FunctionEvaluator can be used to
// compute water pressure in water network pipes from the water flow.
// The input are hourly measurements of the water flow.
// The FunctionEvaluator applies the following function:
// y = alpha * x^ beta
// where x is the input (water flow), alpha = 2.5, beta = 1.8, and
// y is the output (water pressure).
stream<In, tuple<list<float64> y>> Out = FunctionEvaluator(In)
{
param
inputTimeSeries: x;
functionSpecification: {
0.0:[0.0],// mapping points
```

```

1.0:[2.5],
2.0:[8.7055],
3.0:[18.0617],
4.0:[30.3143],
5.0:[45.2987],
6.0:[62.8944],
7.0:[83.0073],
8.0:[105.5606],
9.0:[130.4898],
10.0:[157.7393],
11.0:[187.2608],
12.0:[219.0112],
13.0:[252.9520],
14.0:[289.0483]
};
output
Out:
y= evaluateTimeSeries();
}

```

Example 11-12 shows how a sine wave is approximated using the FunctionEvaluator. Values are provided every 0.1 seconds up to 1 second. The FunctionEvaluator provides estimates of values every 0.01 seconds, providing an effective interpolation of the time series.

Example 11-12 Approximating a sine wave using FunctionEvaluator

```

// provide selected value of the sine at 0, 0.1, 0.2....
// let the function evaluator provide missing value in between:
// (here the function takes 2-dimensional values, which were
randomly generated)
stream<In, tuple<list<float64> y> > OutSine =
FunctionEvaluator(InSine)
{
    param
        inputTimeSeries:InSine.x;

        functionSpecification:{
            0.0000:[0.0000],
            0.1000:[11.7557],
            0.2000:[19.0211],
            0.3000:[19.0211],
            0.4000:[11.7557],
            0.5000:[0.0000],
            0.6000:[-11.7557],

```

```

        0.7000: [-19.0211],
        0.8000: [-19.0211],
        0.9000: [-11.7557],
        1.0000: [-0.0000],
        1.10000: [11.7557]
    };
    output
    OutSine:y=evaluateTimeSeries();
}

```

The approximated sine wave is illustrated in Figure 11-17.

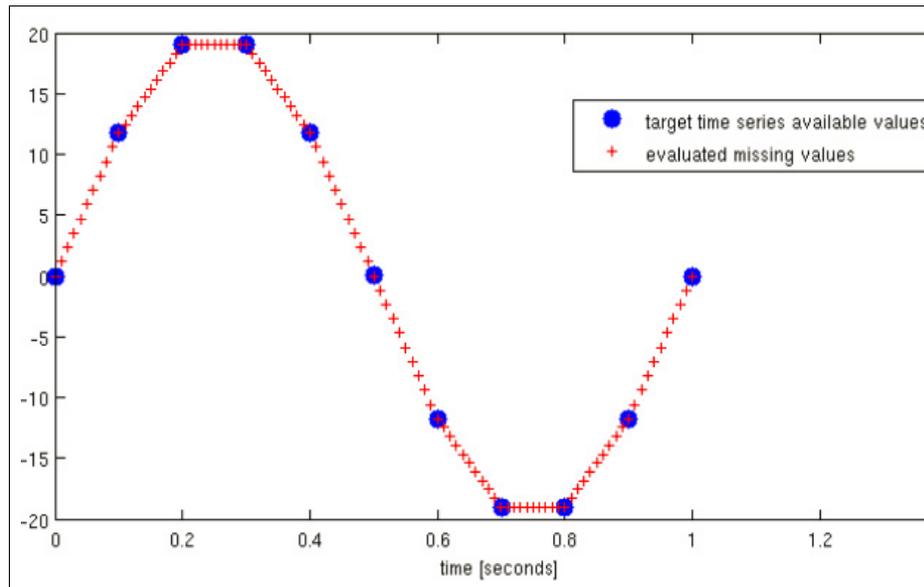


Figure 11-17 interpolation of the sine wave using function evaluator

11.4.8 Distribution operator

The Distribution operator estimates the distribution of the time series values overtime. The distribution refers to a count of the number of time a particular value is seen in incoming data. To do this, the time series value is approximated with the nearest integer. For each integer, counts are registered of how many times that value has been seen. This called a *histogram*. The histogram can then be used to estimate some statistics incrementally, such as median values or outliers. The Distribution operator is helpful in calculating quartiles, median and outliers, such as in Figure 11-18 on page 318.

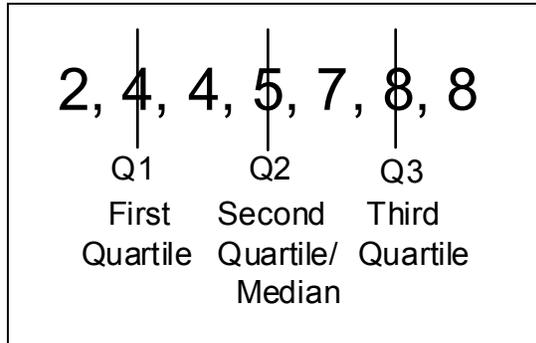


Figure 11-18 Quartiles

Tip: Quartiles are the values that divide a list of numbers into quarters by splitting the sorted list of numbers into four equal parts.

This operator can help you classify data based on how close or how far it is with respect to the median.

Characteristics of the Distribution operator

The Distribution operator has the following features:

- ▶ It estimates the histogram in real time.
- ▶ It is a windowed operator and supports only sliding window configurations.
- ▶ It supports the partitionBy feature. When partitionBy is enabled, individual histograms are created for every partition. For example, if a separate histogram must be created to monitor traffic speeds at various times of the day, partitionBy can be used. In that case, the key to partitionBy is the index of the hour of the day (a value in the range of 0 - 23).
- ▶ The bounds for the distribution values should be specified using the minValue and the maxValue parameters, which refer to the expected minimum and maximum values in a time series. This does not mean the exact value of the minimum or the maximum but rather a range of values that contains the time series. These values are provided by the user, based on the expected range of values of the time series.

Table 11-7 on page 319 describes various output functions supported by the Distribution operator. The functions enable the user to retrieve the entire distribution seen so far or to obtain current statistics including median and quartiles. In addition, you can use the functions to detect outliers because they provide the values of the largest and smallest non-outliers seen so far; values outside that range qualify as potential outliers.

Table 11-7 Distribution operator output functions

Output function	Description
distribution()	Returns the current distribution as a histogram in the form a list of values as sequence of value, count.
firstQuartile()	Returns the first quartile of the distribution.
median()	Returns the median of the distribution.
thirdQuartile()	Returns the third quartile of the distribution.
largestNonOutlier()	Returns the largest non outlier value of the distribution.
smallestNonOutlier()	Returns the smallest non outlier value of the distribution.

Example 11-13 illustrates the use of the Distribution operator. In the example, the Distribution operator is used to generate a distribution of road speeds (in miles per hours) and to identify the median speed and minimum speed in real time. The SPL program also uses the partitionBy feature to maintain a separate histogram for each partition based on an index of the hour of the day.

Example 11-13 Distribution operator usage

```

stream <float64 median,float64 firstQuartile,list<float64> distribution
> Out1 = Distribution(Sample1) {
    window
    Sample1: sliding,count(11),partitioned;
    param
    inputTime series: Sample1.data;
    partitionBy: index;
    minValue:0;
    maxValue:300;
    Out1:
    median = median(),
    firstQuartile=firstQuartile(),
    distribution=distribution();
}

```

11.5 Modeling operators

Modeling operators create an internal model, estimated from the data, and then use this model to perform tracking, prediction, forecasting, or probability estimation. Depending on the operator, the internal model is created incrementally as data comes in, or from a batch of accumulated early data. The interface of the operator enables the control of the model-building process either through a set of parameters or through the use of a control port. The control port enables runtime control of the operator's behavior. This is useful to synchronize and calibrate the model-building process and the model-exploitation process with the characteristics of the current incoming data.

Modeling operators can be used for a variety of time series processing schemes, as in the following examples:

- ▶ **Forecasting:** This is a process of extrapolating future time series values given the past and present time series data. Algorithms such as Auto Regressive Integrated Moving Average (ARIMA), Holt-Winters, and Vector Auto Regressive (VAR) can be used for forecasting time series data. Forecasting is practiced every day on financial values, such as stocks or bonds, or economic indicators, such as gross domestic product (GDP) and trade metrics.
- ▶ **Adaptive filtering:** Adaptive filtering incrementally use errors in prediction at run time to adapt the internal model to changing patterns in the time series. It is often used for tracking, filtering, short-term prediction, or parameter estimation. Adaptive filtering operators include the Kalman operator, the FMPFilter operator, and the RLSFilter operator. Tracking by using the Kalman filter is widely used in navigation systems such as GPS, radar, and missile-tracking devices.
- ▶ **Regression:** This statistical technique is used to estimate the functional relationship between some input variables (called *covariates*) and target variables (called the *dependent variables*). After the relationship is estimated, it can be used to predict the target variables, given new input variables. For instance, a regression model can be used to estimate the relationship between hourly temperature and hourly electricity consumption from available past data. Given a weather forecast for the day, that model can be used to predict hourly electricity consumption for that day. The toolkit supports the General Additive Model (GAM), which is a generic model able to simulate a variety of regression functions, and recursive least-square filter (RLSFilter), which uses an incremental linear regression.
- ▶ **Probabilistic estimation:** Probabilistic estimation uses a model to estimate the number of occurrence of time series values. Probabilistic estimation is useful for areas such as detection, classification, or statistic estimation. The toolkit provides the Gaussian Mixture Model (GMM), a generic model able to represent a variety of probabilistic functions.

11.5.1 HoltWinters operator

Holt-Winters is a well-known forecasting algorithm that exploits the evolution of trend, season, and level to estimate future values of the time series. It belongs to the family of exponential smoothing techniques. The Holt-Winters model can be additive or multiplicative depending on the effect of seasonality on the time series data.

Additive Holt-Winters

The additive Holt-Winters model assumes that season, trend, and level are additive in generating the time series. An additive model is appropriate when seasonality effects across the calendar years are constant in amplitude. An example of additive seasonality is illustrated in “Seasonality” on page 280.

Equations that govern the Holt-Winters additive model are shown in Figure 11-19.

Level equation:	$L_t = (x_t - S_t) (1 - \alpha)(L_{t-1} + T_{t-1})$
Trend equation:	$T_t = (L_t - L_{t-1}) (1 - \beta)T_{t-1}$
Season equation:	$S_t = (x_t - L_t) (1 - \gamma)S_t$
Forecast equation:	$x_{t+h} = L_t + hT_t + S_{t+h}$
L_t : the level at time t	
T_t : the trend at time t	
S_t : the season at time t	
m : length of season (user-supplied)	
h : time value ahead	
α, β, γ , 1, the smoothing factors	

Figure 11-19 Holt-Winters additive model

Multiplicative Holt-Winters

A multiplicative model assumes that the season has a multiplicative effect on the trend and the irregular components. It is an appropriate model when the seasonality effect across the calendar years is proportionally increasing. The equations that govern the Holt-Winters multiplicative model are shown in Figure 11-20 on page 322.

Level equation :	$L_t = \frac{x_t}{S_t} (1 - \alpha)(L_{t-1} + T_{t-1})$
Trend equation :	$T_t = (L_t - L_{t-1}) + (1 - \beta)T_{t-1}$
Season equation :	$S_t = \frac{x_t}{L_t} (1 - \gamma)S_t$
Forecast equation :	$x_{t+h} = (L_t + hT_t)S_{t+h}$

L_t : the level at time t
 T_t : the trend at time t
 S_t : the season at time t
 γ : length of season (user -supplied)
 h : time value ahead
 $\alpha, \beta, \gamma, 1$, the smoothing factors

Figure 11-20 Holt-Winters multiplicative model

HoltWinters operator characteristics

The list of features of the HoltWinters operator are as follows:

- ▶ It supports univariate and vector time series.
- ▶ It is a univariate operator: Each index in the vector time series has its own model.
- ▶ It supports partitionBy.
- ▶ Key parameters are as follows:
 - seasonPerSample: The length of the season (in samples).
 - initSeason: The number of seasons to be used for bootstrapping the algorithm.
 - stepAhead: The number of steps in the future to be forecasted. As shown in Example 11-14 on page 323, 24, time series values are forecasted.
 - algorithm: The choice of multiplicative or additive (which is the default).

Example 11-14 illustrates the use of HoltWinters for forecasting the numbers of passengers of an airlines, using the additive model. The results are shown in Figure 11-21.

Example 11-14 Forecasting using HoltWinters operator

```
stream <float64 webTraffic, list<list<float64>> predictedWebTraffic>
HWAdditive =
HoltWinters(iTime series )
{
param
inputTime series:passengerCounts;
stepAhead: 24u;
samplesPerSeason: 24u;
initSeason: 4u;
algorithm: additive ;
output
HWAdditive: predictedWebTraffic=forecastedAllTime seriesSteps();
}
```

Figure 11-21 illustrates forecasting using the HoltWinters operator. The blue line, which extends across the entire graph, indicates the input time series data. The red line, extending from value 49 across the remainder of the graph, indicates the forecasted data.

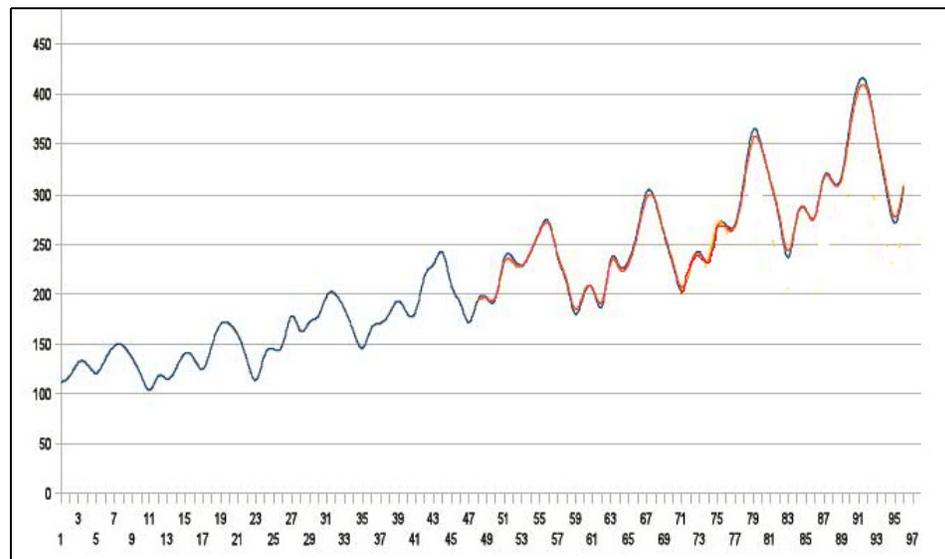


Figure 11-21 Forecasting using HoltWinters operator

11.5.2 ARIMA operator

Autoregressive integrated moving average (ARIMA), also known as the Box-Jenkins method, is a linear model for representing stationary and non-stationary time series. ARIMA is made up of an autoregressive (AR) component, an integrator (I) component that is able to integrate the differentiation of the time series in the model, and a moving average (MA) component. The following list describes the ARIMA components:

- ▶ AR (autoregressive) component: The current time series value is a linear combination of "p" lagged input time series. Figure 11-22 shows the AR equation.

$$x_t = a_0 + a_1 x_{t-1} + a_2 x_{t-2} + \dots + a_p x_{t-p} + e_t$$

Figure 11-22 AR equation

- ▶ MA (moving average) component: Models the input data as a linear combination of lagged past errors. Figure 11-23 shows the MA equation.

$$x_t = \theta_0 + \theta_1 e_{t-1} + \theta_2 e_{t-2} + \dots + \theta_p e_{t-p} + e_t$$

Figure 11-23 MA equation

- ▶ Differentiator: Differentiator allows ARIMA to model non-stationary data. Differentiation converts a non-stationary data (variable mean and variance) to a stationary data (constant mean and variance.) Figure 11-24 shows the differentiator.

$$\begin{aligned} \text{A first order difference is } x'_t &= x_t - x_{t-1} \\ \text{A second order difference is } x''_t &= x'_t - x'_{t-1} \end{aligned}$$

Figure 11-24 Differentiation

The combination of AR and MA and the differentiator provides an ARIMA model.

The ARIMA algorithm has the following characteristics:

- ▶ ARIMA captures the lag correlation between samples and uses them to predict future behavior of the time series.
- ▶ It does not assume any particular pattern in the time series.

- ▶ The general (non-seasonal) model is known as ARIMA (p, d, q):
 - p: This is the number of autoregressive terms.
 - d: This is the order of differences.
 - q: This is the number of moving average terms.
- ▶ ARIMA can be used to represent various models:
 - ARIMA(0,0,0): This is white noise.
 - ARIMA(2,0,0) = AR(2): This is an AR model (no MA and no differentiator).
 - ARIMA(0,0,3) = MA(3): This is an MA model (no AR and no differentiator).

Characteristics of the ARIMA operator

The ARIMA operator has the following key features:

- ▶ ARIMA supports single time series and vector time series
- ▶ ARIMA is a univariate operator; each time series index uses its own optimized model.
- ▶ The ARIMA operator uses early data to estimate the model's coefficients. When the user does not specify the model order (no specified p, d, q values), the p, d, q values are automatically estimated from the data and also the associated coefficients. In this mode, the `initSamples` parameter refers to the number of samples to be used for training the model. If the model cannot be trained with the `initSamples` value, the model will continue to add new training data until training can be done. During the training period, forecasting is not carried out and no output will be produced.
- ▶ Optionally, the user can also provide a model size by providing either `AROrder` (p), `MAOrder` (q), and `Differentiator` (d) parameters.
- ▶ The ARIMA operator can be used as a scoring operator. In this mode, the user provides the ARIMA coefficients along with differentiator as parameters. The user also provides some training context in the form of the most recent data used to train the model (history) and errors produced at training time (residuals). The size of the most recent data should be at least equal to the maximum value of p, q and d.
- ▶ ARIMA supports `partitionBy`.

Example 11-15 on page 326 illustrates the use of the ARIMA operator for forecasting. The model is automatically estimated using the least number of samples specified. The ARIMA operator usage is in Training mode, using only the `initSamples` parameter. AR and MA order are calculated automatically.

```
stream<list<list<float64>> forecast> ARIMALearn = ARIMA(InputTime
series)
{
param
inputTime series: ts;
initSamples:60u;
output
ARIMALearn: forecast = forecastedAllTime seriesSteps();
}
```

11.5.3 FMPFilter operator

The faded-memory polynomial filter is the adaptive filter that uses a polynomial fit to track the evolution of the time series.

FMFilter has the following characteristics:

- ▶ The filters are characterized by what is called *fading memory*, which means that older data samples are progressively weighted less, during the fitting process; the filter puts more weights on recent samples than the last sample.
- ▶ The weights (coefficients) of the polynomial are automatically estimated for each new data.
- ▶ The filter outputs are the estimated local mean values of the time series at each point time. In effect, it is a smoothing filter, which removes noise and outlier artifacts.
- ▶ The algorithm fits a polynomial, and therefore it is able to extrapolate and predict future values accordingly.
- ▶ It is well-suited for real-time outlier detection, because it is able to estimate the variance of the data around the estimated mean value of the time.

FMPFilter has the following applications, among others:

- ▶ Tracking
- ▶ Real-time data smoothing
- ▶ Prediction
- ▶ Anomaly detection.

Characteristics of the FMPFilter

The important features of FMPFilter operator are as follows:

- ▶ Supports univariate and vector time series.
- ▶ Is a univariate operator: each time series index has its own model.
- ▶ Supports partitionBy.
- ▶ Produces smoothed estimate of the future values.
- ▶ Can also flag outlier.

Other key parameters are as follows:

- ▶ degree: Determines the order of the polynomial. Example 11-16 uses a polynomial of degree 2.
- ▶ memoryLength: Indicates the length of fading memory in past values used to control weighting of the past value of the time series. Example 11-16 uses a memoryLength of 25 samples.
- ▶ integration: Indicates the number of samples used for variance estimation. Example 11-16 uses a value of 3 samples.
- ▶ thresholdFactor: Controls the threshold for anomaly detection. It determines the tolerable range of outlier detection around the estimated mean (predicted value). Example 11-16 uses a value of 7.
- ▶ maxDimension: FMPFilter operator supports expanding time series. This parameter specifies the cut off value for the expanding time series.

Example 11-16 illustrates an application of the FMPFilter to anomaly detection to predict and identify anomalies if present.

Example 11-16 Predicting next expected memory usage of a computer

```
//The memory readings are read every 10 milliseconds .FMPFilter tracks //the next
//expected trend.If input is outside the allowable threshold //it is flagged as
//anomaly
stream<list<float64> prediction, list<int32> flags, float64 var>
PredictedStream = FMPFilter(TSInput)
{
  param
  inputTime series: Open;
  degree: 2u;
  memoryLength: 25u;
  integration: 3u;
  thresholdFactor: 7u;
  maxDimension: 25u;
  cleanFrom: 25u;
  output
  predictedStream: prediction=predictedTime series(),
  flags=anomalousFlags();
}
```

The output of Example 11-16 on page 327 is shown in Figure 11-25. The figure shows that FMPFilter detects the synthetically generated anomaly.

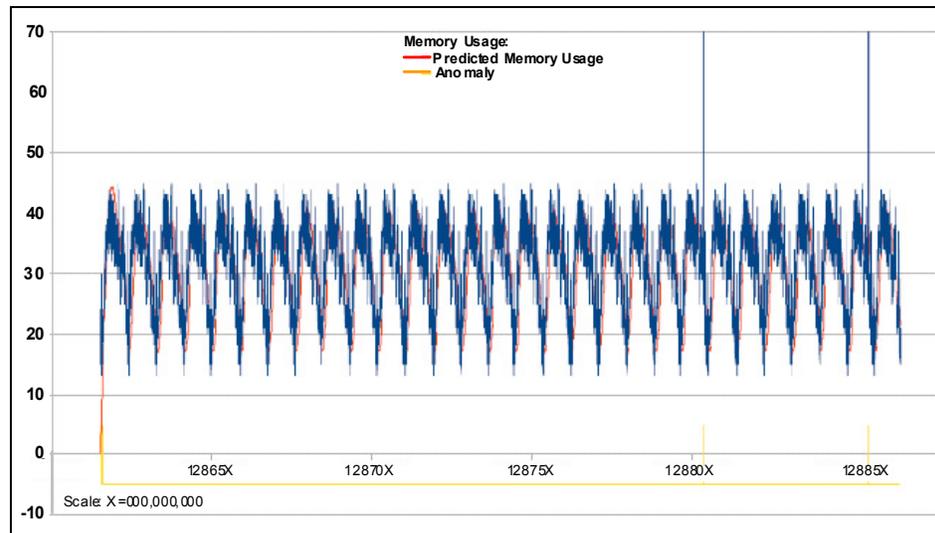


Figure 11-25 illustrates the application of the FMPFilter for anomaly detection

11.5.4 Kalman operator

The Kalman operator is an adaptive filter and is used for the purpose of estimation and tracking. The Kalman filter takes the noisy time series as the input and predicts the true state time series value. It adaptively estimates the true state of the system and variance of the errors committed. The Kalman filter is a standard technique used in a variety of applications ranging from GPS, radar, system estimation, object tracking, or anomaly detection.

The Kalman filter has the following characteristics:

- ▶ Is best suited for smoothing, filtering and short-term prediction. Kalman estimates the state variables by minimizing the estimation error.
- ▶ Is adaptive in nature; it continuously estimates the state variables based on the noisy input data.
- ▶ Is robust for estimating and replacing missing or invalid data.

The Kalman model is described in Figure 11-26 on page 329.

State equation: $x_{t+1} = Ax_t + Bu_t + w_t, w_t \sim (0, Q)$
 Observation equation: $z_t = Hx_t + v_t, v_t \sim (0, R)$
 x_t : the state vector at time t with dimension n
 z_t : the output vector at time t with dimension p
 u_t : the control vector at time t of dimension q
 A : the state transition matrix of dimension $n \times n$
 B : the control matrix of dimension $n \times q$
 H : the output matrix of dimension $p \times n$
 w_t : is the state model noise with zero mean and variance Q
 v_t : is the observation model noise with zero mean and variance R

Figure 11-26 The Kalman model

$W_t \sim N(0, Q)$, the noise around the state is normally distributed with mean zero and variance equal to state noise covariance value. $V_t \sim N(0, R)$: V_t is normally distributed with mean zero and variance equal to observation noise covariance value R . For the V_t parameter too, you can pass either a matrix, vector, or a single value for state noise covariance.

Specifying Kalman matrices

Most parameters in the Kalman operator are matrices. As such, they are specified as `list<list<float64>`.

State transition matrix and the observation matrix are specified as a `list<list<float64>`. All elements in the matrix are specified. When they are not specified, they are assumed to be the unit matrix.

The control matrix is optional. When specified, it should be `list<list<float64>>`. Otherwise, it is assumed equal to a single matrix of value 1. The control matrix should be specified when the a control time series is used.

The state noise covariance and observation noise covariance are optional. The matrix can be specified in many ways, depending on the matrix type:

- ▶ As `list<list<float64>>`: In that case, the whole matrix is specified.
- ▶ As simple vector `list<float64>`: In that case, only diagonal elements are specified and the rest is nil.
- ▶ As a `float64`, which means that all diagonal elements in the matrix have the same specified value; the rest is nil.

When those matrices are not specified, they are assumed to nil.

The dimension of the state matrix and the state noise covariance is either estimated from the `initialState` parameter or explicitly specified by the user using `stateVectorDimension` parameter.

Characteristics of the Kalman operator

The Kalman operator has the following features:

- ▶ Supports single and vector time series
- ▶ It is multivariate operator; it treats the vector as whole entity.
- ▶ Predefined state matrix is unity (per default)
- ▶ When not specified, the predefined controlMatrix is unity.
- ▶ When not specified, the initial state is estimated internally with the first tuples.
- ▶ It supports partitionBy

Example 11-17 illustrates the application of the Kalman operator in tracking an object in a plane. This is a two-dimensional setting. The object moves in the plane as constant speed. The time series is the noisy position of the object at regular intervals of time at two-dimensional time series (x- and y-axis). The state is a 4-dimensional vector comprising the position of the object (x, y) and the velocity (dx, dy). The observation noise covariance and state noise covariance are passed as a single value (all the diagonal elements share the same value). The dimension of the state transition matrix and observation matrix is 4x4 as shown. The Kalman filter will estimate the real position, velocity, and the next predicted position of the object.

Example 11-17 Kalman filter at work for tracking an object in 2D plane

```
// tracking a truck moving on a 2D plane with constant velocity.
stream<list<float64> movingObjectMetrics,list<float64>
predictedObservation, list<float64> observe>
KalmanTracking=Kalman(GPSSrc)
{
    param
    inputTime series: movingObjectMetrics;
    controlDimension: 4u;
    initialState:[ 10.0, 10.0, 1.0,0.0];
    parameterFile:"Parameters.dat";
    observationNoiseCovariance:0.3;
    stateNoiseCovariance:0.3;

stateTransitionMatrix:[[1.0,0.0,1.0,0.0],[0.0,1.0,0.0,1.0],[0.0,0.0,1.0
,0.0],[0.0,0.0,0.0,1.0]];
```

```

observationMatrix: [[1.0,0.0,0.0,0.0],[0.0,1.0,0.0,0.0],[0.0,0.0,1.0,0.0
],[0.0,0.0,0.0,1.0]];
    output
    KalmanTracking:
        predictedObservation=predictedTime series();

}

```

11.5.5 GAM operators

General additive model (GAM) is a regression model developed by Trevor Hastie and Robert Tibshirani of Stanford University in 1990. Its purpose was of blending the properties of generalized linear models with additive models.

It maps the input variables, also called covariates, into a target value, also referred to as dependent variable. The mapping is general and can theoretically approximate any function.

Use of GAM involves two steps:

1. Estimate the GAM model using a training data set. This process can be done offline using any other third-party tool, or done incrementally using the `GAMLearner` operator from a skeleton template Predictive Model Markup Language (PMML). The R-project GAM package can directly train a GAM and generate a compatible PMML file.
2. Predict unknown value using the model recorded in the PMML file. This step can use the *GAMScorer* operator or the *GAMLearner* operator.

The GAM algorithm has several advantages:

- ▶ It can accommodate both categorical data and numerical data. Such categorical variables as weekday or weekend can be used as input data.
- ▶ It can theoretically represent any regression function. That means the standard regression model, such as linear regression or logistic regression, can be seen as a special case of the GAM.
- ▶ The `GAMLearner` operator includes a variation of the optimization algorithm including incremental optimization of the model. Examples include penalized recursive least square (PRLS), PRLS-FF (PRLS with forgetting factor).

The GAM model is defined by the equation in Figure 11-27.

GAMModel :

$$g(y_t) = E \left(f_1(x_t(1)) + f_2(x_t(2)) + \dots + f_M(x_t(M)) \right)$$

$x_t(k)$ is the k - the component of the time series at time t (multivariate)

y_t is the target time series value at time t

$f_k()$ is the k - th transfer function with following type :

- constant value
- categorical value (0 and 1)
- linear function or smooth splines

$g()$ is the output transfer function

$E()$ is the expectation operator

Figure 11-27 GAM model

Example

Consider the example of predicting electricity load using GAM. The SPL program for the example is depicted in Figure 11-28. To predict the electricity load, which is the dependent variable, several covariates or categorical values are also considered, current electricity consumption, temperature, type of day, day of the year.

Electricity forecast example:

$$y_{t+1} = E \left(f_{lag}(x_{t-48}) + f_{DayOfYear}(x_t) + f_{TimeOfDay}(x_t) \right)$$

x_t is actual electricity consumption at the current time t

y_{t+1} is the prediction of electricity consumption at time $t + 1$

x_{t-48} is the electricity consumption 48 hours ago

$f_{lag}()$ is a transfer function that capture lag effect

$f_{DayOfYear}()$ is a transfer function that captures day of year effect to account for seasonal variation

$f_{TimeOfDay}()$ is a transfer function that captures time of day effect to account for daily variation

Figure 11-28 GAM example

Characteristics of the GAMLearner operator

GAMLearner applies GAM to categorical and continuous time series data. If observed data is given, the operator adapts the model parameters and outputs the filtered data.

The model must be contained in the PMML file whose name is provided as a value to the modelID2file parameter. The description of the PMML model is illustrated in Figure 11-29.

```
<PMML>
  <DataDictionary>
    <!-- Define covariates -->
  </DataDictionary>

  <TransformationDictionary>
    <!-- Define transfer functions -->
  </TransformationDictionary>

  <RegressionModel modelName="GAM" functionName="regression" algorithmName="GAM">
    <MiningSchema>
      <!-- Specify which covariates and transfer functions are used -->
    </MiningSchema>

    <RegressionTable intercept="0">
      <!-- Specify complete GAM -->
    </RegressionTable>

    <Extension name="smoothingparameters">
      <!-- Optional: specify spline smoothing parameters -->
    </Extension>
  </RegressionModel>
</PMML>
```

Figure 11-29 Description of PMML model

GAMLearner can process multiple GAMs at once; each PMML must be tagged with a unique modelID, as depicted in Example 11-18.

Example 11-18 Processing multiple pmml files

```
modelID2file:
  {
    1u1 : "ElectricityLoadCity1.pmml",
    2u1 : "ElectricityLoadCity2.pmml"
  };
```

Incoming tuples are assigned to the correct model using the modelID parameter. The modelID parameter acts as a partitionBy parameter; it refers to the input

tuple attribute that contains a reference to a particular time series of an entity. For example, if modelID contains a value of 1, the ElectricityLoadCity1.pmm1 file will be used to process that input.

The GAMLearner operator implementation is shown in Example 11-19 and is described in the following list:

- ▶ The operator supports univariate and vector time series.
- ▶ The operator has several key parameters:
 - modelID2file: Specifies the file that contains the GAM model in PMML format.
 - spl2pmm1: Specifies the mapping between input tuple variables and the covariates in the GAM model.
 - algorithm: Specifies the iterative learning algorithm to be used to learn the model
 - forgettingFactor: Specifies the forgetting factor, which controls the weighting of coefficients in the learning process; older time series data are weighted less during.
 - learningRate: Specifies the learning rate for adaptive forgetting factors.

Example 11-19 GAMLearner operator for electricity load prediction

```
// Apply the GAMLearner to the data stream. In this example, data will arrive only at the
second input port, which will update the GAM model and produce a filtered output value.
Input = tuple<float64 Time, float64 TimeOfDay, rstring DayType, float64 Temperature, float64
Load, float64 FilteredLoad>;

// Format of the data tuples, extended with an identifier for the model:
InputModelID = Input, tuple<uint64 modelID>;

graph

    (stream<Input, tuple<float64 PredictedLoad>> ScoreOutputStream; stream<Input,
tuple<float64 PredictedLoad>> LearnOutputStream)
    = GAMLearner(ScoreInputModelIDStream; LearnInputModelIDStream)
    {
        param
            method: "PRLS-FF"; // model update using Penalized Recursive Least Squares with a
forgetting factor:
                // Use a forgetting factor of 0.9999429 (corresponding to a time window of
approx. 1 year):
            forgettingFactor: 0.9999429f;
            penalizer: "diagonal"; // Use a diagonal penalizer with value lambda = 1.0 on the
main diagonal:
                lambda: 1f;
            // Use an adaptive regularizer with value epsilon = 1.0 on the main diagonal:
```

```

regularizer: "adaptive";
epsilon: 1f;
    // Location of the PMML file specifying the GAM model:
modelID2file:
{
    1u1 : "ElectricityLoad.pmml"
};
// Mapping between the attribute names in SPL and PMML:
spl2pmml:
{
    "Time" : "Time",
    "TimeOfDay" : "TimeOfDay",
    "DayType" : "DayType",
    "Temperature" : "Temperature"
};
// Names of the attributes containing the modelID, the observed variable, and the
filtered variable:
modelID: modelID;
inputTime series: Load;
output
ScoreOutputStream: PredictedLoad=predictedValue();
}

```

Characteristics of the GAMScorer operator

GAMScorer applies GAM to score the input time series values. The model is stored in a PMML file. Scoring means using an already-trained model to predict a new value. As such, the GAMScorer uses a model trained either by the GAMLearner or by third-party applications.

As with GAMLearner, the operator is capable of managing several GAMs at the same time using the ModelID parameter.

The GAMScorer operator has the following characteristics:

- ▶ Supports univariate and vector time series data.
- ▶ Several key parameters of the operator are as follows:
 - modelID2file: Specifies the name of the PMML files containing the GAMs associated with different modelID.
 - spl2pmml: Specifies a mapping between SPL input tuples and the covariates in the GAM.

Example 11-20 illustrates the application of the GAMScorer operator for the electricity load prediction using an already-trained model.

Example 11-20 GAMScorer usage on predicting

```
// Apply the GAMScorer to the data stream. for predicting electricity usage
stream<InputModelID, tuple<float64 PredictedLoad>> ScoreOutputStream =
GAMScorer(ScoreInputModelIDStream)
{
  param
  // Location of the PMML file specifying the GAM model:
  modelID2file:
  {
    1u1 : "ElectricityLoad.pmml"
  };

  // Mapping between the attribute names in SPL and PMML:
  spl2pmml:
  {
    "Time" : "Time",
    "TimeOfDay" : "TimeOfDay",
    "DayType" : "DayType",
    "Temperature": "Temperature"
  };

  // Names of the attributes containing the modelID and the filtered variable:
  modelID: modelID;
  output
  ScoreOutputStream: PredictedLoad=predictedTime series();
}
```

11.5.6 GMM operator

Gaussian mixture model (GMM) is a probability estimation technique. The GMM operator estimates the probability density function (a smoothed histogram) of a time series. It models the density as a sum of normal distributions, as shown in Figure 11-30.

The Gaussian Mixture Model :

$$GMM(x) = \sum_{i=1}^M w_i N(x; \mu_i, \Sigma_i)$$

w_i is the i -th mixture weight
 μ_i is the i -th means
 Σ_i is the i -th covariance of
 $N()$ is the i -th normal distribution

Figure 11-30 Gaussian mixture models

GMM can be used in a variety of applications including these examples:

- ▶ Anomaly detection of outliers
- ▶ Detecting a probability of events

GMM operator has the following characteristics:

- ▶ It supports single time series and vector time series
- ▶ It is a univariate operator: each time series index has its own model.
- ▶ You specify the number of mixtures explicitly with the mixture parameter.
- ▶ It trains only on early data, then starts producing outputs. You specify the number of early samples to be used for training with the trainingSize parameter.

Table 11-8 describes the output functions available in the GMM operator. The probability of a given sample and the probability of the that sample being an outlier is equal to 1.

Table 11-8 GMM operator custom output functions

Output function	Description
probability()	Returns the probability of given input sample.
outlierProbability()	Returns the probability that the given data is an outlier.
PDFValue()	Returns the probability density function.

Example 11-21 shows the use of the GMM operator for an outlier detection problem. In this example, computer usage data was generated for one week and then duplicated for other weeks. An outlier was manually included to illustrate the GMM usage. The results are illustrated in Example 11-21.

Example 11-21 GMM operator usage to detect outliers in memory usage

```
stream<float64 prob, float64 outlierProb, float64 pdfValue>
outputStream =
GMM(KPIStream)
{
param
  inputTime series: KPIs;
  trainingSize: 3400u;
  mixtures: 1u;
output
  outputStream:
  prob=probability(),
  outlierProb=outlierProbability(),
  pdfValue=PDFValue();
}
```

Figure 11-31 on page 339 depicts an anomaly detection application of the GMM operator. GMM is constructed using memory usage patterns; any sample with an outlier probability above the 0.8 threshold value will be classified as an outlier. The top graph depicts the input time series; the bottom graph indicates outlier probabilities.

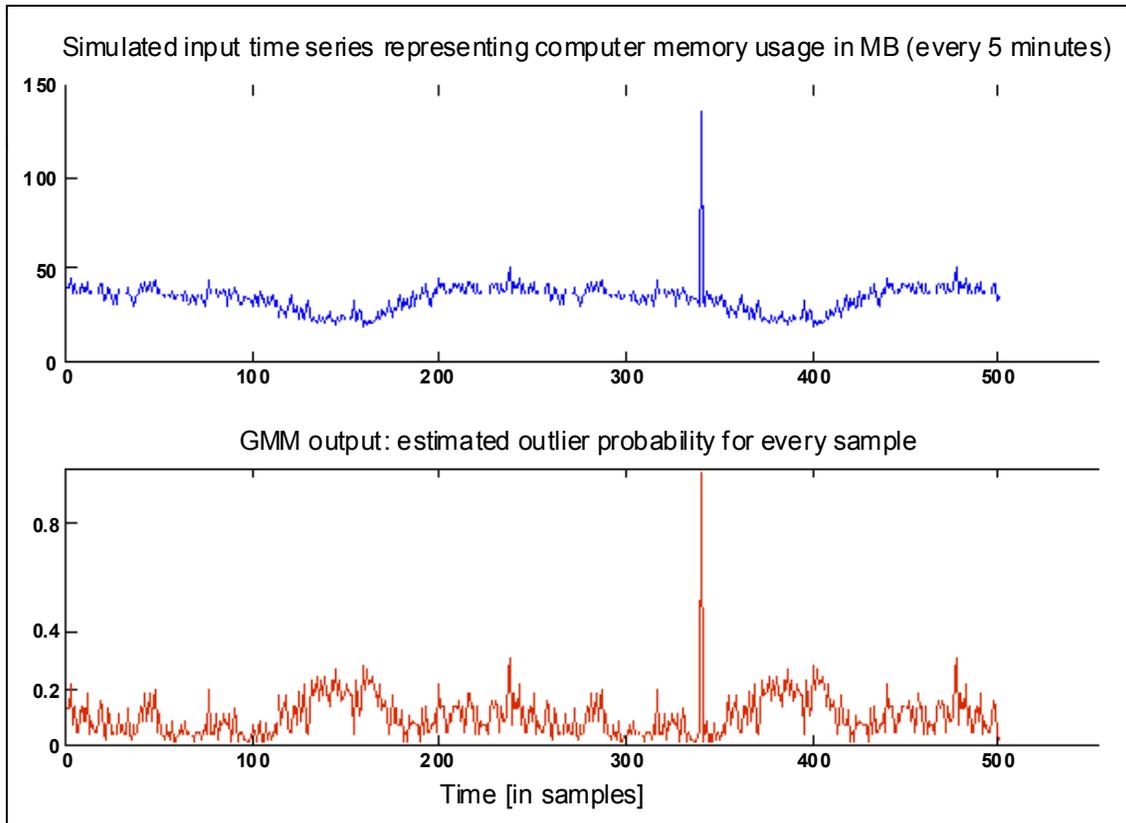


Figure 11-31 Detecting anomaly in memory usage using GMM operator

11.5.7 LPC operator

Linear Predictive Coding (LPC) is new with IBM InfoSphere Streams Version 3.1 and is a standard technique for coding signals, such as speech, for compression, transmission, or analysis. The LPC operator uses an autoregressive technique to encode the data, which can then be a basis for transmission, analysis, or forecasting.

Application of the LPC includes the following items:

- ▶ Signal encoding: Encodes a segment of signal into a set of parameters for binary transmission. This technique was standard in early telephony.
- ▶ Signal compression: A segment of the signal is represented by a few parameters.

- ▶ Spectral estimation: The autoregressive model used in LPC can represent the parameters of a Fourier spectrum model.
- ▶ Forecasting: LPC is used in forecasting of time series; it can be used in forecasting financial data, such stock market data.

Characteristics of the LPC operator

The LPC operator has the following characteristics:

- ▶ It is a univariate operator; each index in a vector time series has its own model.
- ▶ It supports single time series and vector time series
- ▶ It supports partitionBy; interleaved time series can be processed independently.
- ▶ It is semi-incremental, in the sense that a model is created for every new sample of data and reflects the local statistics at every time.

The LPC operator has several key parameters:

- ▶ InitSamples: Specifies the number of input time series values to be used for initialization of the model.
- ▶ order: Specifies the AR order, which LPC internally uses for model's parameters estimation

Example 11-22 shows the use of an 8-order LPC operator for forecasting the next hour of electricity consumption. A stepAhead of 1 is adopted using 144 early samples to initialize the model.

Example 11-22 Example 11-22 LPC operator usage

```
//Predict next hour electricity usage
stream <timestamp predTimestamps, <list<float64>> predTime series>
predStream = LPC(KPIStream)
{
  param
  inputTime series: KPIStream.KPIs;
  initTime: 144u;
  order: 8u;
  stepAhead: 1u;
  output predStream:
    predTimeseries =forecastedTimeseriesStep(),
    predTimestamps=forecastedTimestamps();
}
```

The plot of the output of Example 11-22 on page 340 is shown in Figure 11-32.

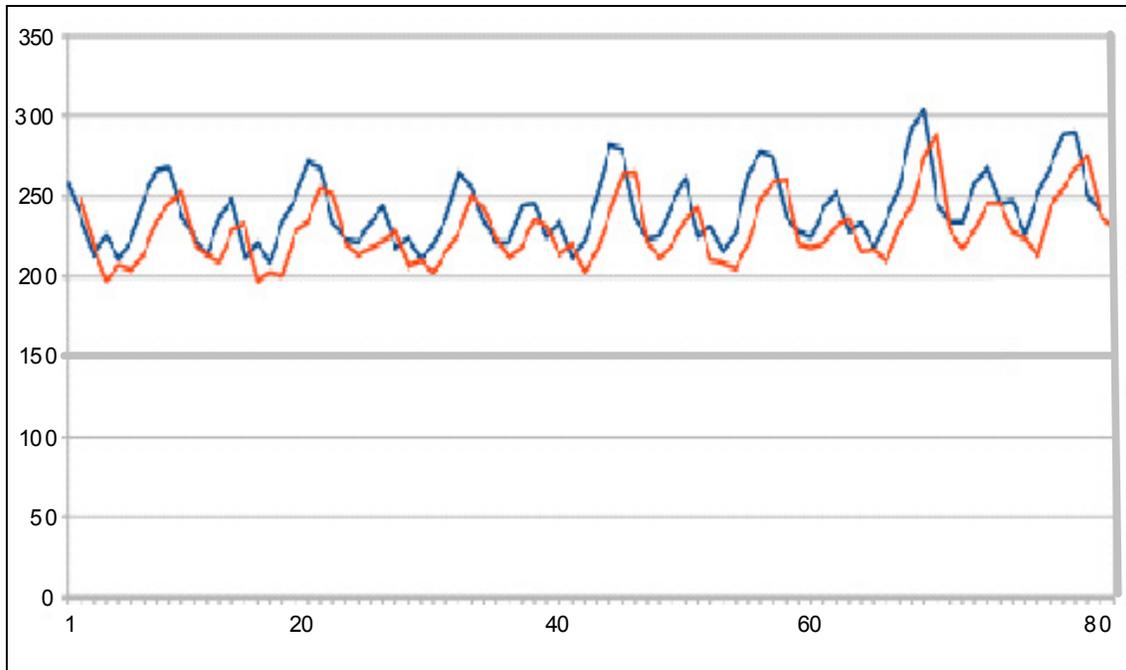


Figure 11-32 Used in predicting next-hour electricity usage

11.5.8 VAR operator

Multivariate autoregressive model (VAR) operator tracks data movement and predicts the next expected time series. It uses a lagged cross-correlation of multiple time series data to determine the movement of data-points in order to predict the next expected time series.

VAR is a powerful modeling technique that is able to track the relative cross-behavior of multiple time series in real time and use them to predict future values. Internally, VAR uses an autoregressive (AR) algorithm but applies that algorithm to vector data.

Characteristics of the VAR operator

The VAR operator has the following characteristics:

- ▶ VAR is a multivariate operator and expects time series values represented as a list.
- ▶ Two of the key parameters of the VAR operator are as follows:
 - `initSamples`: Specifies the number of input time series values to be used for initialization of the model.
 - `order`: Specifies the VAR order

Example 11-23 shows the use of the VAR operator on the multivariate prediction of IBM stock price and the trading volume. Stock and price are both predicted by tracking their relative joint-movements using the VAR operator.

Example 11-23 Multivariate prediction of IBM stock price and trading volume

```
//Given input stock price and trading volume predict the next expected
stock price and trading volume
stream<list<float64> predictions> varOut = VAR(stockInp)
{
  param
  inputTime series:inp;
  initSamples:12u;
  updateTime:6u;
  output
  varOut: predictions=predictedTime series();
}
```

11.5.9 RLSFilter operator

The RLSFilter (recursive least squares filter) operator fits a linear regression model to a series of input time series (covariates) and to track dependent variables. It is a linear regression whose parameters are incrementally estimated from errors produced in predicting the target value. The fitting is done adaptively using the recursive least squares (RLS) method, an iterative process that reduces the errors incrementally over time. The fitted model can be used later to predict the target variable, given new covariates. The mathematical equation that describes the RLSFilter is shown in Figure 11-33 on page 343.

The RLS filter model:

$$\hat{y}(n) = \sum_{k=0}^{M-1} b_k(n)x(n-k)$$

$\hat{y}(n)$ is the estimated output and $x(n-k)$ is the input data at lag k (in the past).

For each new sample $x(n)$, the goal is to estimate recursively the parameter $b_k(n)$ so as to minimize the error $e(n)$ between the prediction $\hat{y}(n)$ and desired output $y(n)$: $e(n) = y(n) - \hat{y}(n)$

This is done by incrementally minimizing the cost function:

$$C(b) = e(n) + \lambda e(n-1) + \lambda^2 e(n-2) \dots + \lambda^{n-1} e(1) + \lambda^n e(0)$$

$0 < \lambda < 1$, is the **forgetting factor**: recent samples are weighted more than old ones.

Figure 11-33 Mathematical equation that describes the RLSFilter

Several applications of the RLSFilter are as follows:

- ▶ Noise or echo cancellation
- ▶ DSPFilter's parameter estimate
- ▶ Prediction of outcome based on input

Characteristics of the RLSFilter operator

The RLSFilter operator has the following characteristics:

- ▶ It is a univariate time series operator: each index in a vector sample has its own model.
- ▶ It supports single time series and vector time series values.
- ▶ It has two output ports that can be used to obtain the predicted values and the model coefficients respectively.
- ▶ The RLSFilter has two key parameters:
 - dependentVariable: This attribute expression specifies the name of the attribute that contains the dependent variable (target) in the input tuple for input ports 0 and 1.
 - forgettingFactor: This parameter specifies a forgetting factor; the value is between 0 and 1. Recent samples are weighted more than the old samples.

Example 11-24 shows the use of the RLSFilter operator for prediction of the number of housing starts as a function of the interest rate and construction cost.

Example 11-24 Prediction of housing starts using interest rate and construction cost

```
//Given an input hosing starts , interest rate and construction
operator understands relation between three and predicts the next
expected housing start
type Covariate tuple<float64 y, list<float64> x>

(stream<Covariate, tuple<float64 y_hat>> modelAndDependentVariable;
stream<list<float64> coefficients> RegressionCoefficients;
stream<Covariate, tuple<float64 y_hat>> PredictedVariable)
= RLSFilter(CovariateAndDependentVariablePunct; Covariate)
{
param
inputCovariates: Covariate.x,CovariateAndDependentVariablePunct.x;
dependentVariable: CovariateAndDependentVariablePunct.y;
forgettingFactor: 1f;
output
UpdateModelAndFilterDependentVariable:
y_hat=estimatedValue();
RegressionCoefficients:
coefficients=coefficients();
PredictDependentVariable:
y_hat=estimatedValue();
}
```

Figure 11-34 shows output of the RLSFilter for the prediction of housing starts, using interest rate and construction costs. Prediction of the RLSFilter is adaptive over time to the data.

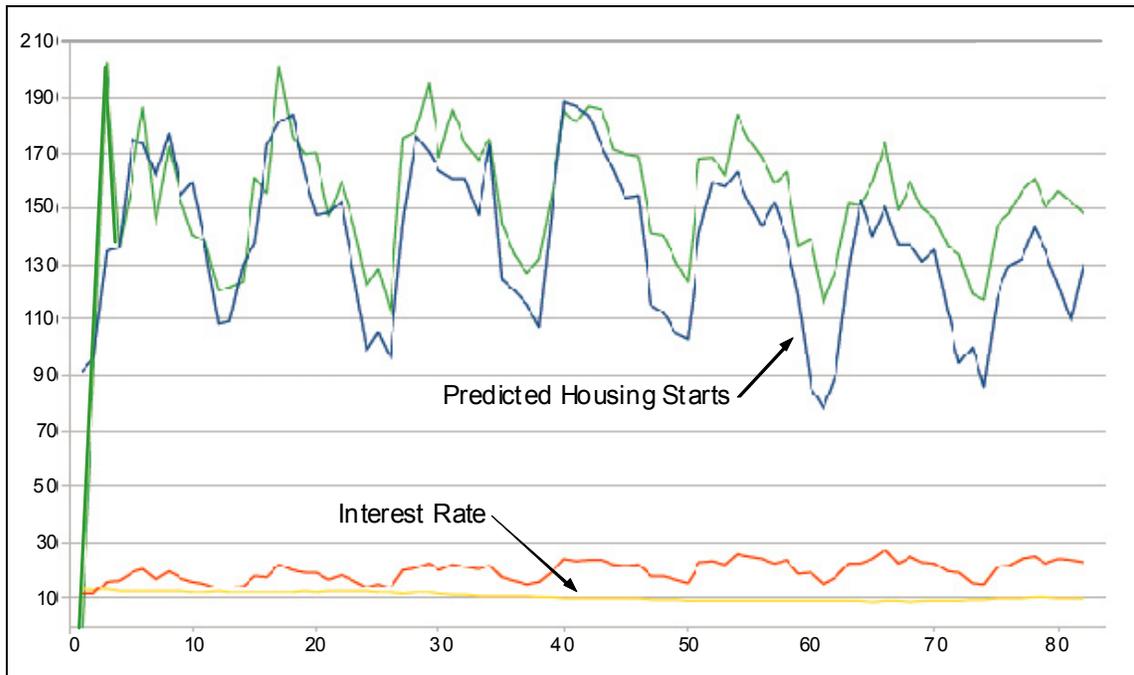


Figure 11-34 RLSFilter operator application; time series functions

11.6 Time series functions

The TimeSeries Toolkit provides a list of functions that can be used within Functor or Custom operators. These include functions to generate time series for test purposes, or to convolve or cross-correlate a time series.

11.6.1 The generator functions

A series or generator functions can generate time series of various shapes and for a user-specified duration. This utility is good for testing time series applications.

- ▶ `generate_triangular_wave(float64 frequency, uint32 duration, uint32 samplingRate)`
 - Generates a triangular wave with a given frequency, duration, and sampling rate.
 - Returns a `list<float64>` as a univariate time series representing the triangular wave.
 - The list has `samplingRate*duration` number of samples. It is good for testing and prototyping.
- ▶ `generate_square_wave(float64 frequency, uint32 duration, uint32 samplingRate)`
 - Generates a square wave with a given frequency, duration, and sampling rate.
 - Returns a `list<float64>` as a univariate time series representing the square wave.
 - The list has `samplingRate*duration` number of samples. It is good for testing and prototyping.
- ▶ `generate_sine_wave(float64 frequency, uint32 duration, uint32 samplingRate)`
 - Generates a sine wave with a given frequency, duration, and sampling rate.
 - Returns a `list<float64>` as a univariate time series representing the sine wave.
 - The list has `samplingRate*duration` number of samples. It is good for testing and prototyping.

- ▶ `generate_sawtooth_wave(float64 frequency, uint32 duration, uint32 samplingRate)`
 - Generates a sawtooth wave with a given frequency, duration, and sampling rate.
 - Returns a `list<float64>` as univariate time series representing the sawtooth wave.
 - The list has `samplingRate*duration` number of samples. It is good for testing and prototyping.
- ▶ `generate_pulse_train_wave(float64 frequency, uint32 duration, float64 humpsize, uint32 samplingRate)`
 - Generates a pulse train wave with a given frequency, duration, sampling rate and hump size.
 - Returns a `list<float64>` as univariate time series representing the pulse train wave.
 - The list has `samplingRate*duration` number of samples. It is good for testing and prototyping.

11.6.2 The Crosscorrelate function

The `crosscorrelate` function estimates the lagged cross-correlation between two lists representing the sequence of values of the two time series at some point in time. It is functional equivalent of the `CrossCorrelate` operator (see 11.4.5, “`CrossCorrelate` operator” on page 311). However, the function version of the `CrossCorrelate` operator returns only raw correlation using direct computation.

The `crosscorrelate` function is also known as a sliding dot product or sliding inner-product, which is defined as follows:

```
list<float64> crosscorrelate (list<float64> firstSignal, list<float64>
secondSignal)
```

The `crosscorrelate` function implements the raw, unbiased, and unnormalized cross-correlation.

11.6.3 The convolve function

The convolution between two time series is the time-domain representation of the product of the Fourier Transform of the two time series. It is a process that can help modify the behavior of a time series based on another time series. Modification consists of taking the product of the two FFT representation of the two time series. For example, convolving a time series with a sine wave is the same as isolating the component in the time series that mimics a sine waves.

The convolution function is represented as follows:

```
list<float64> convolve(list<float64> firstSignal, list<float64>
secondSignal)
```

11.6.4 The rms function

The root mean square (RMS) function is the square root of the average means of the list. It represents the average power of a signal. It is a widely used measure in electrical engineering and physics to measure the power of a signal.

The rms function is defined as follows:

```
T rms(list<T> data)
```



Developing Java primitive operators

In this chapter, we describe how you can develop primitive operators using the Java programming language. The development process is as follows:

1. Design the operator. This involves creating the operator model by describing the operator and its ports, parameters, and dependencies.
2. Create the operator implementation in Java based on the operator model described in the previous step.
3. Compile the operator code using the Java compiler and then fix any errors.
4. Test the operator using the Java Operator testing framework.

Then, you can start using the operator in your SPL applications.

We describe each step of this process, and cover the intricacies of developing a Java operator with parameters and the process of updating custom metrics.

12.1 Operator lifecycle

Before we begin the process of developing an operator, it is important to understand the Java Operator API and the operator lifecycle.

All operators written in Java must implement the following interface and implement a no-argument constructor:

```
com.ibm.streams.operator.Operator
```

Example 12-1 shows the definition for the interface. It provides the basic protocol that must be supported by every operator. Operators can choose to extend the `AbstractOperator` abstract class that provides default implementations for the `Operator` interface and allows you to override only those methods for which custom implementations are to be provided.

Example 12-1 Operator interface

```
public interface Operator {  
  
    public void initialize(OperatorContext context) throws Exception;  
  
    public void allPortsReady() throws Exception;  
  
    public void process(StreamingInput<Tuple> port, Tuple tuple) throws  
Exception;  
  
    public void processPunctuation(StreamingInput<Tuple> port,  
Punctuation mark) throws Exception;  
  
    public void shutdown() throws Exception;  
}
```

An operator instance starts its runtime lifecycle with a call to its `initialize(OperatorContext)` method. This can be used to initialize the state of the operator.

The operator cannot receive or send tuples or punctuations until its ports are ready. After the operator ports are ready, the operator will start receiving calls to its tuple and punctuation processing methods and can submit tuples if needed (after transforming or filtering them) to one or more of its output ports. The `allPortsReady` method is called as a notification to the operator that the ports are ready to receive and submit tuples.

All incoming tuples are sent to the operator through the `process` method. It is passed an instance of the `StreamingInput` class (describing the port on which the tuple arrived) and the tuple as an instance of the `Tuple` class.

Similarly, all incoming punctuations are sent to the operator through the `processPunctuation` method. It is passed an instance of the `StreamingInput` class describing the port on which the punctuation arrived and the actual punctuation as an instance of the `Punctuation` enumeration.

Some of these classes are described in greater detail later in the chapter.

When the processing element (PE) that hosts the operator is being shut down, the operator will receive a call to its `shutdown` method, which can take place while tuple and punctuation processing methods are active. The `shutdown` method completes any asynchronous activity and releases any resources.

12.2 Threading in a Java operator

A Java operator must always be written to be thread-safe because its methods can be called by different threads and can be called concurrently. Standard Java threading concepts apply, including synchronization, locking, and visibility.

With the exception of operator initialization, all the other methods may be called concurrently. Because the methods may be called concurrently they must be made thread-safe with respect to data integrity and state visibility. The specific synchronization needs will be driven by the functional and performance requirements of the operator implementation.

In addition to the *synchronized* keyword, Java provides several classes for multi-threaded programming in the following packages:

- ▶ `java.util.concurrent`
- ▶ `java.util.concurrent.atomic`
- ▶ `java.util.concurrent.locks` packages

For more information, see the Java Platform Standard Edition documentation:

https://publib.boulder.ibm.com/infocenter/ieduasst/v1r1m0/topic/com.ibm.iea.was_v7/was/7.0/ProgrammingModels/WASv7_JavaSE6/player.html?dmuid=20080925182031508907

12.3 Creating a simple operator

Here we develop, a Java operator named `FileSystemMonitor`. This operator monitors a particular location on the local file system for addition, deletion, and other changes, and produces tuples describing each event. The *interval* (in milliseconds) at which the location is scanned for changes will be a configurable parameter with the *location* to be scanned.

We use the Apache Commons IO framework to monitor the file system and show how third-party libraries (in this case, `commons-io-2.4.jar`) can be incorporated when developing an operator.

12.3.1 Operator code layout

Before we start the operator implementation, we explore how the artifacts that make up the operator implementation are organized in the file system.

The operator artifacts are grouped within a package called a *toolkit*. A toolkit is a directory on the file system that groups one or more SPL artifacts (primitive operators, native functions, and others) that are logically related.

In our example, we create a toolkit named `com.ibm.ssb.filemonitor` that will store all the artifacts for the `FileSystemMonitor` operator. The folder structure under the `com.ibm.ssb.filemonitor` directory is shown in Example 12-2.

Example 12-2 The `com.ibm.ssb.filemonitor` toolkit folder structure

```
/com.ibm.ssb.filemonitor
  info.xml
  toolkit.xml
  /com.ibm.ssb.filemonitor
    /FileSystemMonitor
      FileSystemMonitor.xml
  /impl
    /java
      /src
        /com
          /ibm
            /ssb
              /filemonitor
                FileSystemMonitor.java
  /opt
    /commons-io-2.4
      commons-io-2.4.jar
    ...
```

The description of each artifact is as follows:

- ▶ The `info.xml` and the `toolkit.xml` files describe the toolkit. The structure of these files is outside the scope of this chapter. For more details about these, see the SPL Toolkit development reference:

<http://ibm.co/LsjYzY>

- ▶ We put the `FileSystemMonitor` operator in the `com.ibm.ssb.filemonitor` namespace. Hence, we create a subdirectory by that name that will contain folders for each operator in that namespace. In this example, we have only one operator name `FileSystemMonitor`. Therefore, we create a folder by the operator name that will contain a `FileSystemMonitor.xml` file. This XML file is the operator model for this operator. We describe operator model in 12.3.2, “Defining the operator model” on page 353.
- ▶ We put our Java source code in the `impl/java/src` folder. We also use `com.ibm.ssb.filemonitor` as the package name for our Java source file. Therefore, our operator’s implementation code will be in the `impl/java/src/com/ibm/ssb/filemonitor` folder. Note that the package name and the SPL namespace for the operator can differ.
- ▶ Finally, we put the third-party library package (Apache Commons IO, in this case) within the `opt/` folder under the main toolkit directory.

12.3.2 Defining the operator model

Now we can implement the files listed in the structure described in 12.3.1, “Operator code layout” on page 352. The first step is to consider the design for our operator.

In our example, the `FileSystemMonitor` operator is a source operator, which means that it generates tuples for consumption by others instead of receiving tuples and transforming or filtering them. Hence, it will have no input port and one output port. The output port will be used to send tuples indicating the events that are occurring in the location on the file system.

Similarly, the operator will also take two pieces of information as input, which will both become parameters for the operator:

- ▶ The *location* to monitor
- ▶ The *interval* at which to poll for file system changes

This design information is put into the operator model. It is an XML document (`FileSystemMonitor.xml` in this example) that describes the basic syntactic and semantic properties of the operator that you are trying to create.

The following properties can be listed in the model:

- ▶ Class for the operator
- ▶ Library dependencies for the operator
- ▶ Input and output port cardinality
- ▶ Parameters for the operator

For a full list of properties that can be specified in this file, see the *IBM Streams Processing Language Operator Model Reference* at the following location:

<http://ibm.co/1fTGDz9>

Example 12-3 shows the operator model for the FileSystemMonitor operator. We describe each section of the model in detail.

Example 12-3 FileSystemMonitor operator model

```
<?xml version="1.0" encoding="UTF-8"?>
<operatorModel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://www.ibm.com/xmlns/prod/streams/sp1/operator"
xmlns:cmn="http://www.ibm.com/xmlns/prod/streams/sp1/common"
xsi:schemaLocation="http://www.ibm.com/xmlns/prod/streams/sp1/operator
operatorModel.xsd">
  <javaOperatorModel>
    <context>
      <description>This operator monitors the file system for changes
and sends tuples for each change event.</description>
      <executionSettings>

<className>com.ibm.ssb.filemonitor.FileSystemMonitor</className>
      </executionSettings>
      <libraryDependencies>
        <library>
          <cmn:description>Java operator class
library</cmn:description>
          <cmn:managedLibrary>
            <cmn:libPath>../../impl/java/bin</cmn:libPath>
          </cmn:managedLibrary>
        </library>
        <library>
          <cmn:description>Dependency for implementing monitoring
functionality</cmn:description>
          <cmn:managedLibrary>

<cmn:libPath>../../opt/commons-io-2.4/commons-io-2.4.jar</cmn:libPath>
          </cmn:managedLibrary>
        </library>
      </libraryDependencies>
    </context>
  </javaOperatorModel>
</operatorModel>
```

```

    </libraryDependencies>
</context>
<parameters>
  <description/>
  <parameter>
    <name>location</name>
    <description>Location to monitor</description>
    <optional>>false</optional>
    <type>rstring</type>
    <cardinality>1</cardinality>
  </parameter>
  <parameter>
    <name>interval</name>
    <description>The polling interval for the monitor. The default
value is 10 millisecs.</description>
    <optional>>true</optional>
    <type>int64</type>
    <cardinality>1</cardinality>
  </parameter>
</parameters>
<inputPorts/>
<outputPorts>
  <outputPortSet>
    <windowPunctuationOutputMode>Free</windowPunctuationOutputMode>
    <windowPunctuationInputPort>-1</windowPunctuationInputPort>
    <cardinality>1</cardinality>
    <optional>>false</optional>
  </outputPortSet>
</outputPorts>
</javaOperatorModel>
</operatorModel>

```

Context

We start with the *context* element. This covers the basic settings that must be provided for every Java operator. Each child element is covered.

description

This element provides a textual description of the operator and its functionality. Streams Studio also shows this description to developers using your operator in SPL applications.

executionSettings

This element describes the core of the operator functionality. You must specify a fully qualified (with package name) `className` of the operator Java class. In our example, `com.ibm.ssb.filemonitor` is the package in which the Java file will reside, and `FileSystemMonitor` is the Java class.

libraryDependencies

This element lists all the libraries (including the compiler output of the Java operator) that are required for this operator to function. In this example, two libraries are required. The compile output is generated into the `impl/java/bin` folder under the main operator folder, and the `commons-io-2.4.jar` is an Apache Commons input/output framework that we use to implement the folder monitoring functionality. The `libPath` is specified relative to the operator model XML file location.

Parameters

The *parameters* element describes all the parameters that can be set on the operator when the application is being written. By using parameters, application developers can customize the behavior of the operator to suit their needs.

Each parameter element has several properties that describe the parameter and its behavior.

name

This element provides the name for the parameter. In our example, we define two parameters: `location` and `interval`.

description

This element can be used to describe the usage of the parameter. Streams Studio also shows this description to developers using your operator in SPL applications.

optional

This element takes a boolean `true` or `false`. If set to `true`, then the parameter is optional and does not need to be specified when the operator is used in an application. In our example, the `location` parameter must be specified because the operator cannot function without it. However, the `interval` parameter is optional and if it is not specified then we assume a default interval of 10 milliseconds. The default value is set in the implementation code that we discuss later.

type

This element indicates the SPL type of the parameter and determines the values that the parameter can assume. In our example, the `location` parameter (which is the absolute path of the location to monitor) is of type *rstring*; the `interval` parameter (which is a time in milliseconds) is of type *int64*.

cardinality

This element indicates the number of values allowed for a parameter. A value of minus 1 (-1) indicates that the parameter can take any number of values. In our example, both parameters can take at most one value.

Input and output ports

The next two elements in the operator model XML file are `inputPorts` and `outputPorts`.

Because there are no input ports, the `inputPorts` element is empty. However, the `outputPorts` element describes the one and only output port for the operator and that it is not optional.

Tip: You can use Streams Studio to create the operator model XML file graphically instead of creating it manually. The development environment also validates your XML document for errors. For more information, see the “Developing streams applications using Streams Studio” documentation:

http://pic.dhe.ibm.com/infocenter/streams/v3r0/nav/5_0

12.3.3 Implementing the operator in Java

Now that we have defined the operator model, the next step is to write the Java file that will contain the implementation of the operator.

As indicated previously, we develop the `FileSystemMonitor` class to monitor a location on the file system for changes using the Apache Commons IO framework.

Example 12-4 shows the implementation of the `FileSystemMonitor` class. We describe each section of the implementation.

Example 12-4 FileSystemMonitor operator implementation

```
package com.ibm.ssb.filemonitor;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.
```

```

import org.apache.commons.io.monitor.FileAlterationListener;
import org.apache.commons.io.monitor.FileAlterationObserver;

import com.ibm.streams.operator.AbstractOperator;
import com.ibm.streams.operator.OperatorContext;
import com.ibm.streams.operator.OutputTuple;
import com.ibm.streams.operator.model.Parameter;

/**
 * Java operator implementation
 */
public class FileSystemMonitor extends AbstractOperator {

    private FileAlterationObserver observer;

    private String location;

    private long interval = 10;

    @Parameter
    public void setLocation(String location) {
        this.location = location;
    }

    @Parameter(optional=true)
    public void setInterval(long interval) {
        this.interval = interval;
    }

    /**
     * Initialize this operator. Called once before any tuples are processed.
     * @param context OperatorContext for this operator.
     * @throws Exception Operator failure, will cause the enclosing PE to
    terminate.
     */
    @Override
    public void initialize(OperatorContext context) throws Exception {
        super.initialize(context);
        File file = new File(location);
        if(!file.exists()) {
            throw new FileNotFoundException("Monitoring location is not
    valid");
        }
        observer = new FileAlterationObserver(file);
        observer.addListener(new FileAlterationListener() {

            @Override
            public void onStop(FileAlterationObserver arg0) {

```

```

    }

    @Override
    public void onStart(FileAlterationObserver arg0) {
    }

    @Override
    public void onFileDelete(File file) {
        submitTuple(FilesystemEvent.Delete, file);
    }

    @Override
    public void onFileCreate(File file) {
        submitTuple(FilesystemEvent.Create, file);
    }

    @Override
    public void onFileChange(File file) {
        submitTuple(FilesystemEvent.Change, file);
    }

    @Override
    public void onDirectoryDelete(File dir) {
        submitTuple(FilesystemEvent.Delete, dir);
    }

    @Override
    public void onDirectoryCreate(File dir) {
        submitTuple(FilesystemEvent.Create, dir);
    }

    @Override
    public void onDirectoryChange(File dir) {
        submitTuple(FilesystemEvent.Change, dir);
    }
    });
}

@Override
public void allPortsReady() throws Exception {
    super.allPortsReady();
    getOperatorContext().getScheduledExecutorService()
        .scheduleAtFixedRate(new Runnable() {
            @Override
            public void run() {
                observer.checkAndNotify();
            }
        }, 0, interval, TimeUnit.MILLISECONDS);
}

```

```

private void submitTuple(FilesystemEvent event, File file) {
    try {
        OutputTuple tuple = getOutput(0).newTuple();
        tuple.setEnum(FilesystemEvent.class, 0, event);
        tuple.setString(1, file.getCanonicalPath());
        getOutput(0).submit(tuple);
    } catch (Exception e) {
    }
}

@Override
public void shutdown() throws Exception {
    observer.destroy();
    super.shutdown();
}

public enum FilesystemEvent {
    Create,
    Delete,
    Change
}
}

```

All Java operators must implement the `com.ibm.streams.operator.Operator` interface. Our class extends the `AbstractOperator` class, which in turn implements the `Operator` interface.

The class also declares three fields:

- ▶ `observer`

This field represents an instance of the following class that checks for file system changes and notifies listeners:

```
org.apache.commons.io.monitor.FileAlterationObserver
```

- ▶ `location`

This string stores the value of the location parameter that is specified when the operator is used in an application.

- ▶ `interval`

This long value stores the value of the interval parameter, in milliseconds. However, because this parameter is optional, the default value of this field is 10 ms.

Parameters

Recall that the two parameters needed by the operator were declared in the operator model XML file. Example 12-5 shows the snippet of the operator model XML file where the parameters are declared.

Example 12-5 Operator model: parameters

```
<parameters>
  <description/>
  <parameter>
    <name>location</name>
    <description>Location to monitor</description>
    <optional>>false</optional>
    <type>rstring</type>
    <cardinality>1</cardinality>
  </parameter>
  <parameter>
    <name>interval</name>
    <description>The polling interval for the monitor. The default
value is 10 millisecs.</description>
    <optional>>true</optional>
    <type>int64</type>
    <cardinality>1</cardinality>
  </parameter>
</parameters>
```

The `location` and `interval` class fields are meant to be set from the values for these parameters in the application code. To facilitate the setting of these class fields automatically when the application is run, we provide two setter methods for these fields. Example 12-6 shows these setter methods extracted from the full source code in Example 12-4 on page 357.

Example 12-6 Setting up parameters

```
@Parameter
public void setLocation(String location) {
    this.location = location;
}

@Parameter(optional=true)
public void setInterval(long interval) {
    this.interval = interval;
}
```

Both setter methods are fairly straightforward and set up the values of the class fields. For these setter methods to be called by the Java run time automatically with the values that are set in the SPL application, we annotate both methods with the `@Parameter` annotation. The `@Parameter` annotation informs the run time that these methods are special setter methods for SPL parameters.

The `@Parameter` annotation can optionally take two attributes:

- ▶ `name`
This attribute is the name of the parameter as specified in the operator model. If not specified or set to empty string then the name is assumed to be same as the field name.
- ▶ `optional`
This attribute indicates whether the parameter is optional or mandatory. The default value is false.

In Example 12-6 on page 361, the location field is assumed to match the operator model parameter name and is considered mandatory; the interval field is also assumed to be the same as the operator model parameter name and is considered optional.

At run time, both these fields are set automatically and can be queried and used by the operator for its operation. Next, we look at the core implementation of the operator starting with the initialization.

Writing the monitoring code

This section describes initialization, starting the monitoring operation, and shut down.

Initialization

The next step in operator development is to implement the initialize method. This method is called after the parameters are set and can be used to initialize the state of the operator. You can use this method to initialize the state of class fields or make calls to third-party libraries to set up their state.

In our example, we use the initialize method to set up the file system observer that will be called to monitor the file system for changes. We call the Apache Commons IO framework to initialize an instance of the following class:

```
org.apache.commons.io.monitor.FileAlterationObserver
```

This class takes the location to monitor as a parameter.

We also add a listener to the `FileAlterationObserver` class instance, which will be notified for each file system event at the location being monitored. This listener

will receive each event, create tuples with the event information and submit the tuple to the output port. The output tuple that is generated will contain two attributes:

- ▶ `eventType`
This is an enumeration (SPL enum), indicating the type of event that occurred. It can take three values: create, delete, and change.
- ▶ `location`
This is a string (SPL rstring), indicating the file/directory where the event took place.

Example 12-7 shows the tuple submission code.

Example 12-7 Tuple submission

```
private void submitTuple(FilesystemEvent event, File file) {
    try {
        OutputTuple tuple = getOutput(0).newTuple();
        tuple.setEnum(FilesystemEvent.class, 0, event);
        tuple.setString(1, file.getCanonicalPath());
        getOutput(0).submit(tuple);
    } catch (Exception e) {
    }
}
```

The `submitTuple` code takes two parameters:

- ▶ `event`
This is the type of change that occurred on the file system. It can take one of three values as specified in the `FilesystemEvent` enumeration: create, delete and change.
- ▶ `file`
This is the file or directory within the monitored location that changed.

Each output port for the operator can be retrieved using the `getOutput` method. The index of the output port is passed to the method and the output port is returned as an instance of the `StreamingOutput<OutputTuple>` interface. This interface provides methods that operate on the output port of the operator.

For example, here are three commonly used methods:

- ▶ `newTuple`
This method returns a new output tuple (an instance of the `OutputTuple` interface) whose attributes can be set with new information.
- ▶ `submit`
This method takes an output tuple and submits it downstream through the operator port.
- ▶ `punctuate`
This method takes a punctuation and submits it downstream through the operator port.

For more information, see the SPL Java Operator API documentation:

<http://pic.dhe.ibm.com/infocenter/streams/v2r0/index.jsp?topic=%2Fcom.ibm.swg.im.infosphere.streams.javadoc.api.doc%2Fdoc%2Findex.html>

Consider the following additional information about Example 12-7 on page 363:

- ▶ We first create a new tuple by calling the `newTuple` method.
- ▶ As mentioned, the new tuple is returned as an instance of the `OutputTuple` interface. This interface allows us to call several setter methods to set the value of attributes of this tuple. |
- ▶ We call `setEnum`, passing the enumeration type (`FilesystemEvent.class`), index of the attribute to be set and the enumeration value (*event*) to be set. In this example, the first attribute (index 0) will be set with the enumeration value of the event.
- ▶ We then call `setString` to set the second attribute to the file or directory where the event took place.
- ▶ At this point, our output tuple is complete. We call `submit` to send this tuple downstream from the output port.

Now that the foundation of the location monitoring code is set up, we are ready to create and start the asynchronous monitoring operation. However, this cannot be done in the *initialize* code because the ports are not yet set up.

Starting the monitoring operation

The run time will call the `allPortsReady` after the ports are initialized. We wait until the ports are ready before we start our monitoring operation. Example 12-8 on page 365 shows the code for the monitoring operation.

```
@Override
public void allPortsReady() throws Exception {
    super.allPortsReady();
    getOperatorContext().getScheduledExecutorService()
        .scheduleAtFixedRate( new Runnable() {
            @Override
            public void run() {
                observer.checkAndNotify();
            }
        }, 0, interval, TimeUnit.MILLISECONDS);
}
```

The monitoring operation is fairly straightforward. It is started using the following method:

```
getOperatorContext().getScheduledExecutorService().scheduleAtFixedRate()
```

This starts an operation that runs a piece of code at regular intervals. In our example, it calls the `FileAlterationObserver.checkAndNotify()` method every `interval` milliseconds. This, in turn, will check the location for changes and notify the listener that was set up in the initialize method. The listener then generates tuples for each file system event and submits them to the output port.

Shutdown

When the processing element running our operator is shut down, the run time calls the shutdown method in our operator. This allows us to perform any cleanup operation needed before the PE is shut down. In our example, we destroy the `FileAlterationObserver` instance by calling the `observer.destroy()` method, and then exit.

Summary

That's it! The steps help you create a simple operator that generates tuples. For additional information about any of the classes or interfaces mentioned previously, see the SPL Java Operator API documentation at the Streams information center:

<http://pic.dhe.ibm.com/infocenter/streams/v2r0/index.jsp?topic=%2Fcom.ibm.swg.im.infosphere.streams.javadoc.api.doc%2Fdoc%2Findex.html>

Tip: Streams Studio provides several features to assist you in operator development including a wizard to create a new Java primitive operator and content assist. See the “Developing streams applications by using Streams Studio” website for more information:

http://pic.dhe.ibm.com/infocenter/streams/v3r0/nav/5_0

12.3.4 Compiling the operator code

So far, we have completed the following steps:

1. We created an operator model.
2. We created the Java code for our operator.

The next step is to compile our operator code so that it is ready to be integrated into an SPL application. The compilation process is straightforward because the operator behaves like any other Java code. We can run the `javac` Java compiler and generate class files for our operator. Example 12-9 shows the command to run to compile the operator code.

Example 12-9 Operator Java compile command

```
javac -cp
$STREAMS_INSTALL/lib/com.ibm.streams.operator.jar:opt/commons-io-2.4/co
mmons-io-2.4.jar
impl/java/src/com/ibm/ssb/filemonitor/FileSystemMonitor.java -d
impl/java/bin/
```

Note the following information about the command:

- ▶ The `FileSystemMonitor` operator depends on the SPL Java Operator API and also the Apache Commons IO framework for its work. Hence, the class path to the compiler is set to the location of the Java Operator API JAR file (`com.ibm.streams.operator.jar`) and also the Apache Commons IO JAR file (`commons-io-2.4.jar`). All operators that you develop will have a dependency on the Java Operator API JAR in addition to any third-party frameworks that you are using.
- ▶ We are putting the class files into the `impl/java/bin` folder. This location is the same location as referenced in the operator model.

At this point, the operator is ready to be used in an SPL application.

Tip: If you are using Streams Studio for developing operators, your Java code is compiled automatically every time you save your source file. The class files are generated to your output directory. For more information, see the “Developing streams applications by using Streams Studio” documentation:

http://pic.dhe.ibm.com/infocenter/streams/v3r0/nav/5_0

12.3.5 Testing the operator code

Before incorporating the operator that we created in SPL applications, we must test to be sure that the operator behaves as expected. In InfoSphere Streams 3.1, a Java operator testing framework was added to allow testing of individual, or a graph of, operators. In this section, we explore how this can be used to test the `FileSystemMonitor` operator that we created.

Example 12-10 shows the source for `FileSystemMonitorTester.java` test driver program that can be used to test the behavior of the `FileSystemMonitor` operator. In this example, the Java file is assumed to be located in the `impl/java/src/test` folder within the toolkit directory. The program starts the operator in a temp location (`./testfiles` folder), makes five file system changes on that location, and makes sure that five tuples are generated at the output port of the operator.

Example 12-10 `FileSystemMonitorTester.java`

```
package test;

import java.io.File;
import java.util.concurrent.Future;

import com.ibm.ssb.filemonitor.FileSystemMonitor;

import com.ibm.streams.flow.declare.OperatorInvocation;
import com.ibm.streams.flow.declare.OutputPortDeclaration;
import com.ibm.streams.flow.handlers.StreamCounter;
import com.ibm.streams.flow.javaprimitives.JavaOperatorTester;
import com.ibm.streams.flow.javaprimitives.JavaTestableGraph;
import com.ibm.streams.operator.Tuple;

public class FileSystemMonitorTester {

    private static final String TEST_LOCATION = "./testfiles";

    public static void main(String[] args) throws Exception {
        JavaOperatorTester tester = new JavaOperatorTester();
        OperatorInvocation<FileSystemMonitor> opInvoke =
            tester.singleOp(FileSystemMonitor.class);
```

```

        OutputPortDeclaration outputPort =
opInvoke.addOutput("tuple<enum{Create,Delete,Change}eventType, rstring file>");
        opInvoke.setStringParameter("location", TEST_LOCATION);
        opInvoke.setIntParameter("interval", 10);

        JavaTestableGraph graph = tester.tester(opInvoke);
        StreamCounter<Tuple> counter = new StreamCounter<Tuple>();
        graph.registerStreamHandler(outputPort, counter);

        deleteTestFiles();
        Future<JavaTestableGraph> future = graph.execute();
        generateTestFiles();
        future.cancel(true);

        if(counter.getTupleCount() == 5) {
            System.out.println("Passed!");
        }
        else {
            System.out.println("Failed!");
        }
    }

private static void generateTestFiles() throws Exception {
    File file = new File(TEST_LOCATION);
    if(file.isDirectory()){
        File temp = new File(file, "test.txt");
        temp.createNewFile();
        Thread.sleep(30);

        temp = new File(file, "testdir");
        temp.mkdir();
        Thread.sleep(30);

        temp = new File(file, "test2.txt");
        temp.createNewFile();
        Thread.sleep(30);

        temp = new File(file, "test.txt");
        temp.renameTo(new File(file, "renamed.txt"));
        Thread.sleep(30);

        temp.delete();
        Thread.sleep(30);
    }
}

private static void deleteTestFiles() {
    File file = new File(TEST_LOCATION);

```

```

        if(file.exists()) {
            if(file.isDirectory()){
                String[] myFiles = file.list();
                for (int i=0; i<myFiles.length; i++) {
                    File myFile = new File(file, myFiles[i]);
                    myFile.delete();
                }
            }
        }
        else {
            file.mkdir();
        }
    }
}

```

Here are the steps we followed and the Java operator testing framework we used:

1. Because we plan to test a single Java operator, we create an instance of the `JavaOperatorTester` class and then call the `singleOp()` method with our `FileSystemMonitor` operator class. This returns an instance of `OperatorInvocation`, which provides methods to set operator attributes similar to a real SPL application.
2. We set various properties on the `OperatorInvocation` object including the output port and the tuple schema it supports and also the values for the `location` and `interval` parameters. After the operator is declared, an instance of this operator can be created using the `JavaOperatorTester.tester()` method. This returns an instance of the `JavaTestableGraph` class.
3. We register a stream handler on the output port. This can be done using the `JavaTestableGraph.registerStreamHandler()` method. A stream handler will receive tuples and punctuations that are being from the associated output port, and perform actions with them. In our example, we use the `StreamCounter` class, which is designed to receive incoming tuples and punctuations, and to store a count of each.
4. Now that the handler is set up, we can actually programmatically run the operator code, make changes to that location (create new files, delete files, rename files) and then determine if the number of changes matches the number of generated tuples.

We can start the operator using the `JavaTestableGraph.execute()` method. At this point, our operator's `initialize` and `allPortsReady` methods are called and the monitoring thread is started. The started operator thread will be returned as an instance of the `java.util.concurrent.Future` class. Any changes now made to the monitored location (`./testfiles` folder)

will now be sent as tuples to the output port. The attached stream counter will receive these tuples and keep count.

5. Note that our operator is not designed to end automatically. It keeps monitoring the location until it is stopped; usually when the PE is stopped. In the case of the test framework, we can end the operator by using the `Future.cancel(true)` method.
6. The final step is to compare the count with the expected value. The count of tuples can be retrieved using the `StreamCounter.getTupleCount()` method.

Example 12-11 shows the command that can be used to compile the test program. After compiled, this program can be run to test the correct operation of the `FileSystemMonitor` operator.

Example 12-11 `FileSystemMonitorTester.java` compile command

```
javac -cp $STREAMS_INSTALL/lib/com.ibm.streams.operator.jar
FileSystemMonitorTester.java
```

In Example 12-11, we wrote a simple Java application to test our operator. However, the testing code can be incorporated into automated testing frameworks such as JUnit also.

12.3.6 Creating an SPL application

Now that the operator has been developed and tested, the next step is to incorporate the operator into an SPL application. From an application's point of view, a Java operator behaves like any other operator and does not require any specific set up.

The `com.ibm.ssb.filemonitor` toolkit that we developed previously can be used by one or more SPL applications that require file system monitoring functionality. Toward that goal, the application code must be kept separate from the toolkit artifacts so that the toolkit can be shared. For more information, see the *SPL Toolkit development reference* at the following location:

<http://ibm.co/1j7xQhY>

Example 12-12 on page 371 shows a simple application that uses the `FileSystemMonitor` operator with a Custom operator to log changes on the file system to standard output.

```
namespace application ;

type EventType = enum {Create, Delete, Change};
type EventTuple = EventType event, rstring location;

composite Main
{
  graph
    stream<EventTuple> FileSystemMonitor =
      com.ibm.ssb.filemonitor::FileSystemMonitor()
      {
        param
          location : "/some/location";
      }

    () as Custom_2 = Custom(FileSystemMonitor)
    {
      logic
        onTuple FileSystemMonitor: println("Event : " +
      (rstring)event + " Location : " + location);
    }
  }
}
```

The following list explains the application in the example:

- ▶ We declare 2 SPL types:
 - EventType: An SPL enum that maps directly to the FilesystemEvent Java enumeration that we used in the operator implementation code.
 - EventTuple: The type of the tuple that is produced by the FileSystemMonitor operator. It has two attributes:
 - The event (EventType)
 - The location where the event occurred (rstring)
- ▶ We create a new main composite (Main) with a graph of two operators. The first operator is an instance of FileSystemMonitor and its output port is connected to a Custom operator. The Custom operator will print the tuples it receives to standard output. We use `<namespace>::<operator name>` notation to refer to the FileSystemMonitor operator.
- ▶ The FileSystemMonitor operator can take two parameters. However, location is the only required parameter. Hence, we specify a value (/some/location) for that parameter in the application. The interval parameter, because unspecified, will take a default value of 10 milliseconds.

You can now compile the application and submit the job as you would with any other Streams application.

Tip: Streams Studio can help you develop your SPL applications. It provides an editor that allows you to create an application graph with operators and streams graphically. For more information, see “Developing streams applications by using Streams Studio” documentation:

http://pic.dhe.ibm.com/infocenter/streams/v3r0/nav/5_0

12.3.7 Adding custom metrics

Metrics are simple counters, maintained at run time, that can be read from outside of a running job to monitor statistics of interest. Two types of metrics are provided by the SPL language run time:

- ▶ System metrics are predefined and are maintained by the SPL run time.
- ▶ Custom metrics are created and maintained by the operators.

The next step in the evolution of our operator is to add support for a new custom metric: `nCreationEvents`. It will store the number of resource creation events processed by the operator at any given time.

Updating the operator model

The first step in adding support for this custom metric is to update the operator model to describe the metric. Example 12-13 shows the `context` section of the operator model and the newly added `metrics` element.

Example 12-13 Operator model with metrics support

```
<context>
  <description>Java Operator FileSystemMonitor</description>
  <metrics>
    <metric>
      <name>nCreationEvents</name>
      <description>The number of resource creation events processed
by the operator</description>
      <kind>Counter</kind>
    </metric>
  </metrics>
  <executionSettings>

<className>com.ibm.ssb.filemonitor.FileSystemMonitor</className>
  </executionSettings>
  <libraryDependencies>
```

```

        <library>
          <cmn:description>Java operator class
library</cmn:description>
          <cmn:managedLibrary>
            <cmn:libPath>../../impl/java/bin</cmn:libPath>
          </cmn:managedLibrary>
        </library>
        <library>
          <cmn:description>Dependency for implementing monitoring
functionality</cmn:description>
          <cmn:managedLibrary>
            <cmn:libPath>../../opt/commons-io-2.4/commons-io-2.4.jar</cmn:libPath>
          </cmn:managedLibrary>
        </library>
      </libraryDependencies>
    </context>

```

The *metrics* element shown in bold in Example 12-13 on page 372 can contain one or more metric definitions. Each component of a metric is as follows:

- ▶ **name**
This is the name of the custom metric being defined. In our example, this is `nCreationEvents`.
 - ▶ **description**
This is a textual explanation of what the metric provides.
 - ▶ **kind**
This describes the type of metric that is being provided. It can take one of three values:
 - Counter indicates that this metric represents a count of occurrence of some event.
 - Gauge indicates a value that is continuously variable with time.
 - Time indicates a metric that represents a point in time.
- In our example, because we are counting occurrences of file system events, we will set `kind` to `Counter`.

Updating the operator code

To add support for custom metrics in our operator, we must also slightly modify the `FileSystemMonitor` Java code that we developed in this chapter.

Example 12-14 shows how the operator code can be changed to incorporate the `nCreationEvents` metric that we defined in the previous model.

Example 12-14 Operator code with metric additions

```
public class FileSystemMonitor extends AbstractOperator {

    private FileAlterationObserver observer;

    private String location;

    private long interval = 10;

    private Metric nCreationEvents;

    @CustomMetric(kind=Metric.Kind.COUNTER)
    public void setnCreationEvents(Metric nCreationEvents) {
        this.nCreationEvents = nCreationEvents;
    }
    ...
    ...
    private void submitTuple(FilesystemEvent event, File file) {
        try {
            OutputTuple tuple = getOutput(0).newTuple();
            tuple.setEnum(FilesystemEvent.class, 0, event);
            tuple.setString(1, file.getCanonicalPath());
            if(event == FilesystemEvent.Create)
                nCreationEvents.increment();
            getOutput(0).submit(tuple);
        } catch (Exception e) {
        }
    }
    ...
    ...
}
```

The example shows only the relevant portions of the `FileSystemMonitor` class. The new code that was added to support the new metric is shown in bold font. We next examine the new code in detail.

The first step in adding the new metric is to declare a class field that will store a reference to the metric and allow us to interact with it and update it. All metrics are instances of the `com.ibm.streams.operator.metrics.Metric` interface. It provides access to methods for retrieving and updating metric properties. In our

example, we define a class field `nCreationEvents` of the following type that will be used to update the custom metric

```
com.ibm.streams.operator.metrics.Metric
```

The next step allows the Java run time to initialize the `nCreationEvents` field automatically. Therefore, we define a setter method (`setnCreationEvents`) for our `nCreationEvents` field and annotate it with the `@CustomMetric` annotation. This informs the run time to call the setter method to initialize the `nCreationEvents` custom metric.

The `@CustomMetric` annotation can accept up to four parameters:

- ▶ `name`
This is the name of the metric as specified in the operator model. If not provided or is set to blank, the name is assumed to be the same as the field name.
- ▶ `description`
This is text indicating what the metric is supposed to be measuring.
- ▶ `kind`
This can take values as defined in the `Metric.Kind` enumeration. It must match the kind element for the metric in the operator model. In our example, we set `kind` to `Metric.Kind.COUNTER` to match the operator model and to indicate that this is a counter type metric.
- ▶ `mxbean`
This Boolean indicates whether to register this metric into the platform's MBean server. JMX and MBeans are outside the scope of this chapter. The default value of this is *false*.

The final step is to actually change our code so that the `nCreationEvents` metric is updated whenever a `FilesystemEvent.Create` event is received by the operator. This is done by calling the `nCreationEvents.increment()` method in the operator's `submitTuple` method before a tuple is submitted for the event.

In these steps, we have just created a new custom metric in our operator that can be monitored externally using Streams Studio or the Streams Console. With slight modifications, the steps can be followed to implement other metrics such as one that returns the number of deletion events.

12.3.8 Implementing a tuple consumer operator

In Example 12-14 on page 374, we show the process of creating a source operator. That is, the operator monitors the file system and then generates new tuples. However, in some situations, you might want to create an operator that consumes tuples coming into its input port (or ports) and filtering or transforming them before optionally sending them out through one or more output ports.

In such situations, the implementation details of developing an operator outlined previously are still valid but with several different steps:

- ▶ Because the tuples are not generated by the operator but received at the input port (or ports), the operator needs to override the process method. Example 12-15 shows how the process method can be implemented by an operator.
- ▶ In the case of an operator that can receive punctuations and needs to handle or forward them to its output port, it needs to override the processPunctuation method.

Example 12-15 Operator.process method implementation

```
@Override
public void process(StreamingInput stream, Tuple tuple) throws
Exception {
    OutputTuple outputTuple = getOutput(0).newTuple();

    // Copy across all matching attributes.
    outputTuple.assign(tuple);

    long timestamp = tuple.getLong("timestamp");
    float temperature = tuple.getFloat("temperature");

    // calculate data from timestamp & temperature
    double threshold = ...
    outputTuple.setDouble("threshold", threshold);
    getOutput(0).submit(outputTuple);
}
```

The *process* method takes two parameters as input:

▶ `stream`

This is an instance of `StreamingInput` interface and provides information about the port and its associated schema.

▶ `tuple`

This is the incoming tuple as an instance of the `Tuple` interface. Attributes can be queried from this parameter using the provided getter methods.

The code snippet performs the following steps:

1. A new output tuple is created with the `getOutput(0).newTuple()` method.
2. All the attributes from the incoming tuple are copied over to the output tuple using the `assign` call.
3. We get the values of the timestamp and temperature attributes from the incoming tuple.
4. A threshold is calculated using these values. The actual calculation is irrelevant to this example.
5. This calculated value is passed to the output tuple through the threshold attribute.
6. The final output tuple is submitted to the output port.

An important aspect about the `process` and `processPunctuation` methods is that they can be called by multiple threads at the same time. Therefore, you must make sure that the code in both these methods is thread-safe.

The example shows how the Java Operator API can be used to access ports, their schemas, get and set attributes on tuples, and submit tuples. For details about additional functionality that is available, see the SPL Java Operator API documentation at the Streams information center:

<http://pic.dhe.ibm.com/infocenter/streams/v2r0/index.jsp?topic=%2Fcom.ibm.swg.im.infosphere.streams.javadoc.api.doc%2Fdoc%2Findex.html>

12.4 Java development using Streams Studio

In previous sections, we covered how you can design, implement, compile, and test a simple operator in Java. To make your development easier, some of the steps outlined could have been performed using the Streams Studio development environment that is bundled with InfoSphere Streams. In this section, we cover features in Streams Studio that can help you with your Java operator development.

If you already have an existing SPL project in Streams Studio, use the following steps to create a new Java operator:

1. Right-click the SPL project and select **New** → **SPL Primitive Operator**.
2. In the New SPL Primitive Operator wizard dialog, you can select the namespace and name for the operator. You can also select the language for the operator. In this case, select **Java** as the language type. Click **Next** to continue.
3. Because the Java language is chosen for the operator, the Operator Implementation Class wizard page allows you to provide details about the Java code that is to be generated.

You can specify the package, class name, and source folder for the Java class that will be generated. In addition to that, you can choose to let the wizard generate an operator that fits a particular operator pattern.

4. Click **Finish** to create the operator code.

Upon successful completion of the wizard, your project has these artifacts:

- ▶ The Java source code for the operator.
- ▶ A basic operator model XML file.
- ▶ The SPL Java Operator framework JAR, added as a dependency.

Now, you may edit your source code and operator model within Streams Studio and take advantage of all the Java tooling available including content assist, find references, type, and call hierarchy. As you modify and save your source code, the Java compiler automatically generates class files into the output directory. To change the location of the output directory and also modify other Java build settings, use the following steps:

1. Right-click your SPL project and select Properties.
2. Select the **Java Build Path** property page. Use this page to change source and class file locations and manage dependencies.
3. Select the **Java Compiler** property page. Use this page to customize the behavior of the Java compiler.

For additional assistance with Java operator development, Streams Studio contains a “cheat sheet” that provides a step-by-step guide for creating primitive Java operators. To access select **Help** → **Cheat Sheets**.

For more details about features that are available in Streams Studio for Java development, see “Developing streams applications using Streams Studio” documentation:

http://pic.dhe.ibm.com/infocenter/streams/v3r0/nav/5_0



Text Analytics, AQL

This book introduces the concept of the three V's: volume, variety, and velocity. Variety is meant to represent data that is in one of three forms:

- ▶ **Structured data:** Like data found to exist in a relational database, generally; a rigid, homogenous, spreadsheet-like, row and column format to data.
- ▶ **Semi-structured data:** Like data from software server log files, machine data, HTML pages, data from many types of real-time sensors (heart monitors, weather stations), and related.
- ▶ **Unstructured data:** Data from blogs, social media, RSS news feeds, comment fields in databases (databases normally being thought of as structured data only), PDF files, images, digitized sound, and related.

In Chapter 4, “Analytics entirely with SPL” on page 85, we build a Streams application entirely in Streams Processing Language (SPL), using no Streams toolkits, or external tools or libraries. That example processes semi-structured data.

In this chapter, we move further into semi-structured and unstructured data, primarily using the Streams Text Analytics toolkit. See the following website:

<http://www.ibm.com/developerworks/library/bd-streamstextanalytics/>

At the core of the Text Analytics toolkit is IBM Annotation Query Language (AQL), a declarative language purposely similar to the widely used relational database

Structured Query Language (SQL). What SQL does for structured data, AQL does for semi-structured and unstructured data.

AQL contributes a significant portion of the IBM Streams and IBM BigInsights Social Data Analytics (SDA) accelerator, Machine Data Analytics (MDA) accelerator, and others. Learning the Text Analytics toolkit, and thus AQL, allows you to easily process semi-structured and unstructured data, and serves as a strong foundation to using SDA and MDA.

Note: Before Streams can perform text analytics on PDF files, Microsoft Word files, digitized speech, or related (each of these file formats being stored natively as binary data), we most commonly use an open source, or third-party piece of software, to output ASCII text characters. AQL operates on text, not on binary data.

In this chapter, we do the following tasks:

1. Provide an overview of the text analytics process, using the example of an Apache HTTP server log file.
2. Adjust our software installation.

The runtime software to *run* text analytics is fine. However, we install the text analytics *tools* through one of two methods:

- We change the settings inside our Streams project to be able to run text analytics.
 - We create a BigInsights project to be able to develop and debug text analytics.
3. Complete the following steps in our first AQL example:
 - Create a dictionary.
 - Create a view with extract dictionary.
 - Create a view with extract regex.
 - Create a view with extract pattern.
 - Use a select case statement.
 - Use a Not ContainsDict() predicate to handle values not found.
 - Use a unioned select to manage multiple lists of values.
 - Use a return group clause with extract pattern.
 4. Introduce additional training materials, and instructional videos that detail use of the (AQL) Extraction Tasks, and Extraction Plan view.
 5. Explain regular expressions (regex), provide examples, and introduce the Regular Expression Builder Wizard, Regular Expression Generator Wizard, and Streams regexMatch() standard function.

6. Provide details about external dictionaries, AQL tables (in an AQL script, and external), joins, where clauses (filter expressions), multiple output view AQL scripts, and integration with Streams applications, the minus clause, and more.
7. Introduce topics and constructs not previously described: AQL parts of speech, sentiment, utilities, and more.

13.1 Overview text analytics, by example

In the information technology industry, one can generally assume that most people have at least a basic understanding of relational databases (tables, rows, columns, indexes, column data types, SQL SELECT statements, and so on), or a basic understanding of programming languages (variables, assignment operands, flow control, other). There is, however, almost no guarantee that these same people will have ever encountered text processing or text analytics.

Note: In this context, we present the terms text processing, text analytics, and even text mining as synonyms. The are other terms like, machine learning, natural language processing, sentiment analysis, and others, that offer important distinctions.

To serve as an introduction to text analytics, we again introduce the example from Chapter 4, “Analytics entirely with SPL” on page 85. In that chapter we perform text processing on the log file from an Apache HTTP server. This log file format begins with ten to twelve reasonably well defined, reasonably well formatted columns. an Apache HTTP server log file entry is highly unstructured; after these first ten to twelve columns, you might see an additional two to eighty columns, with highly variable format and content, with specific data being presented in almost any order (any sequence). Processing data in this format (or non-format) is easy with AQL.

Example 13-1 shows entries from an Apache HTTP server log file. See the code review after the example.

Example 13-1 The last few fields from an Apache Web server log file

```
"mozilla/5.0 (x11; u; linux x86_64; it; rv:1.9.1.9) gecko/20100402
ubuntu/9.10 (karmic) firefox/3.5.9"
```

```
"mozilla/5.0 (x11; u; linux i686; en-us; rv:1.8.1.11) gecko/20070914
mandriva/2.0.0.11-1.lmdv2007.0 IDS is best (2007.0) firefox/2.0.0.11"
```

```
"mozilla/5.0 (macintosh; u; intel mac os x 10_6_3; en-us)
applewebkit/531.22.7 (KHTML, like Gecko) version/4.0.5 safari/531.22.7"
```

```
"mozilla/5.0 (windows; u; windows nt 5.1; zh-cn; rv:1.9.2.3)
gecko/20100401 firefox/3.6.3"
```

```
"mozilla/5.0 (x11; u; linux x86_64; en-us) applewebkit/533.4 (KHTML,
like Gecko) chrome/5.0.375.55 safari/533.4"
```

Consider the following information about the sample data in Example 13-1 on page 381:

- ▶ Five Apache HTTP server log file entries presented. To save space, these entries contain only columns thirteen and higher (we deleted columns one through twelve to save space, and to focus this discussion), which are collectively referred to as the *user agent*. The user agent is meant to describe the web browser (or web crawler) software that the client used to make a request from the HTTP server. In the sample entries, these five lines are reasonably consistent, and you might even see obvious patterns in the data.
- ▶ From an investigation of a larger representative data file, we made the following observations:
 - The operating system from which the client was making the HTTP request would contain only one of the three following words 99% of the time:
 - Linux (11.4%)
 - Windows (84.2%)
 - MAC (3.3%)

For 1.1% of the time, the request contained none of these three words.

We call this column the *OS_Major*.

Note: From this point forward in Example 13-1, we discuss only Linux sample entries from the Apache HTTP server log.

- For entries identified as Linux, the entry always contains one of the three values:
 - linux i686 (a 32-bit client)
 - linux x86_64 (a 64-bit client)
 - linux (a client of unknown bitness, generally a web crawler, or robot)

We call this column the *OS_Bitness*.

– We received these user language values:

- en-us: English
- zh-cn: Chinese
- it: Italian

We call this column the *User_Language*.

– Although the operating system in this discussion is Linux, the distribution of Linux is identified by an additional entry from the list: Fedora, Ubuntu, Mandriva, CentOS, Linux Mint, AppleWebKit, and Spinn3r.

We call this column the *OS_Minor*.

– The browser type was indicated by these values: Firefox, Safari, Namoroka, or Spinn3r Robot.

We call this column the *Browser_Major*.

► The keywords can appear multiple times in an entry, and can lead us to incorrectly make an identification. For example, you will see entries similar to Browser-X compatible with Firefox, and thus might incorrectly identify the browser as Firefox. How we handle scenarios like this is to deepen the patterns we look for:

– As a word, i686 occurred often. When i686 was listed and prefixed with the word linux, as in linux i686, our analysis displayed with near certainty that we had identified the OS_Bitness.

– The Browser_Major value was always followed by a numeric value for software version, as displayed at the end of each line in Example 13-1 on page 381. These are some examples:

- Firefox/2.0.0.11
- Firefox/3.6.3
- Safari/533.4

Thus, we look for the Browser_Major value, but only when followed by a numeric, which we also extract as the software version number.

– In Example 13-1 on page 381, you see a similar presentation for OS_Minor, and software version number, for example, Ubuntu/9.10.

An approach to analytics

The observations and techniques we offer outline our common manner of execution:

- ▶ We examine the sample input data, with a subject matter expert if available, or with the result of web search engine research. We often “nibble” at this sample data file; iteratively running tests that extract more and more text, and then correcting to receive more and more accurate text by looking for distinct patterns.
- ▶ First we extract the smallest word units, including words that might be extracted out of context. (From Example 13-1 on page 381, i 686 was often found not in error, but out of context, and Firefox was often found, but out of context.)
 - Why do we say *out of context* versus *in error*?

The text we retrieved was exactly the text we asked for; it just was not the word we wanted in exactly the correct place, or *in the correct context*.

It is helpful to think of context versus error (being wrong). Extract all candidate text, then improve the context and the higher the likelihood that you have they word as you want it.
 - If the list of these available words is known and finite, we extract using *one technique*. (Browser_Major, for example Firefox, is from a relatively well known, short list of valid values.)
 - If the list of words is variable or too large to manage (for example, the Browser_Major software version number, or OS_Minor software version number can be from thousands of valid values), we extract using *another technique*.
- ▶ After we have all of our candidate words, we remove false positives by looking at distinct pattern combinations: Browser_minor followed immediately by a number (for example, Firefox/3.6.3 rather than just the word Firefox).

13.2 Installing and configuring text analytics tools

The first AQL example we create delivers the text extraction outlined previously. AQL runs in the Streams Studio or within a Streams instance with no modification to the software installation, because the runtime libraries are present when the Text Analytics toolkit is present. (The Text Analytics toolkit runtime software is always present.) To develop AQL with any software assistance, you might want to install other tools that are included at no charge with Streams Studio. These installation steps are described here.

Figure 13-1 displays the result of selecting **Help** → **About** → **Installation Details**, in the Streams Studio developer's workbench. The figure shows a Streams Studio installation that does not have the text analytics tools installed.

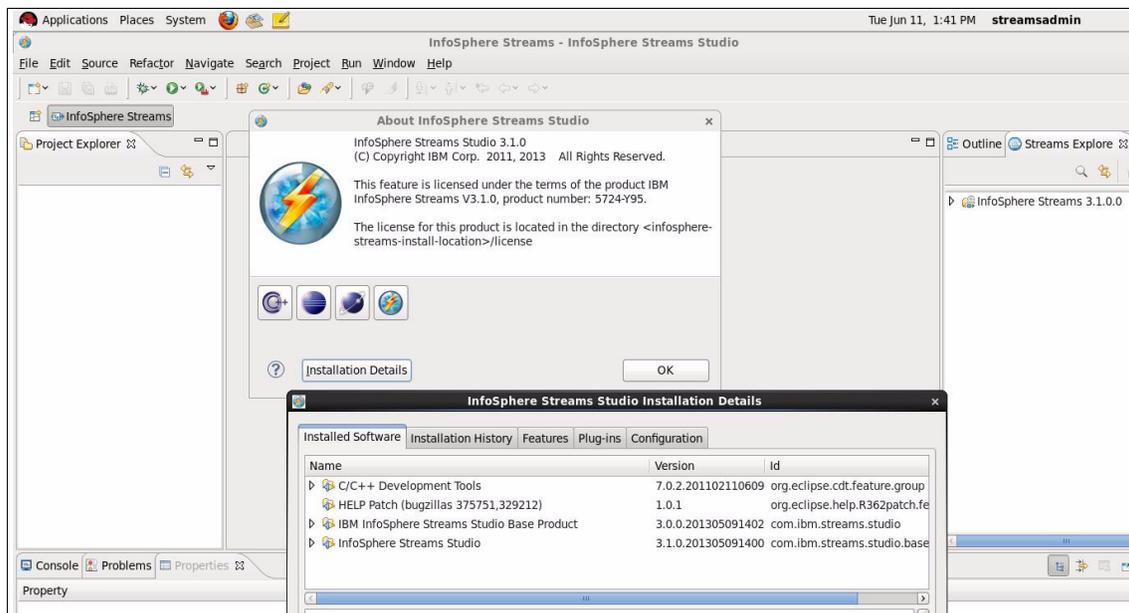


Figure 13-1 Results of Help → About → Installation Details

The additional text analytics tools we want to install differ slightly by release number (software version number) and by the procedure we choose to install from (from the local hard disk, or a new or older remote file; these two means also determine version).

A Streams Studio installation with the text analytics tools installed is displayed in Figure 13-2 on page 386. We are looking for these tools: IBM InfoSphere BigInsights, IBM InfoSphere BigInsights Runtime libraries.

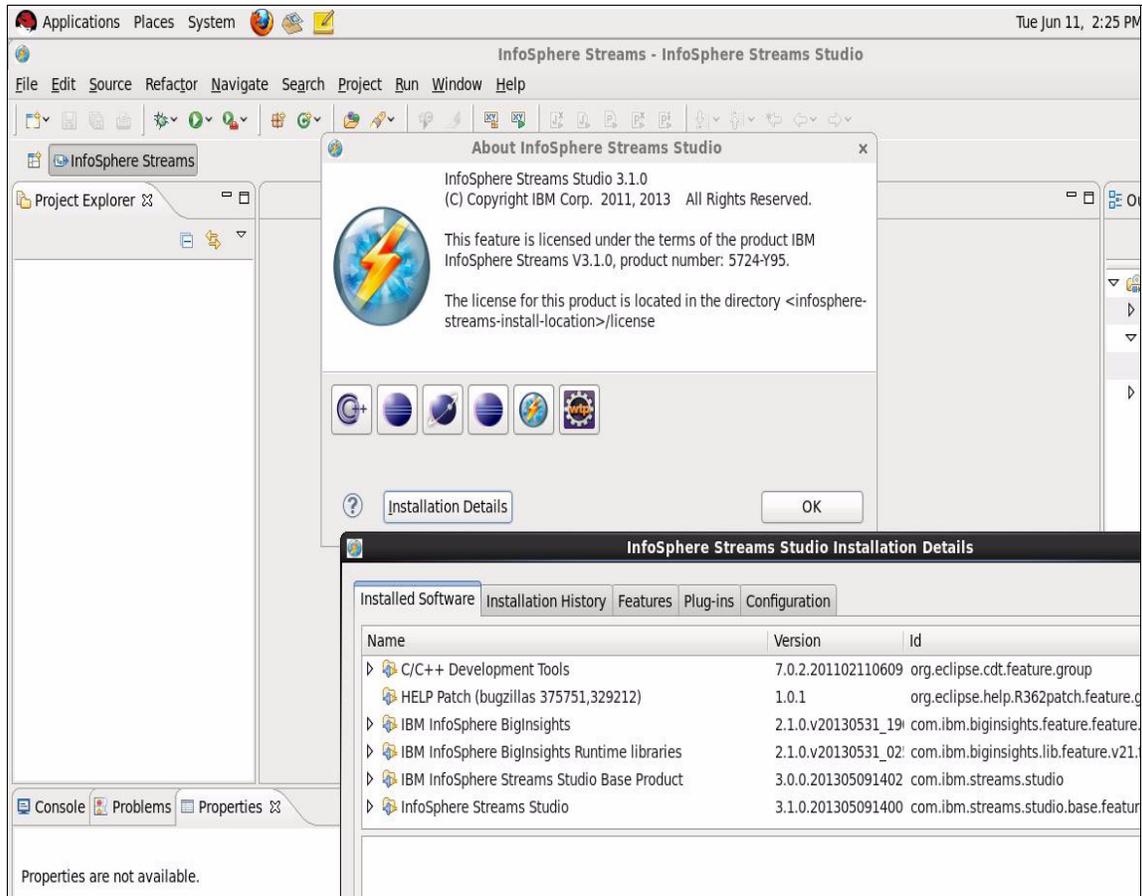


Figure 13-2 Streams Studio installation, with text analytics tools

13.2.1 Installing text analytics tools from Streams Studio installation

The text analytics tools software is already present on a Streams Studio installation, but are not yet installed. This software is located under the Streams Studio parent installation directory, `/etc/BIUpdateSite`, as Figure 13-3 on page 387 shows.

Use either section but not both: If you want to install the text analytics tools, follow the instructions in this section *or* the next section only. You do not need to follow the instructions in both this section and next, because that would be redundant.

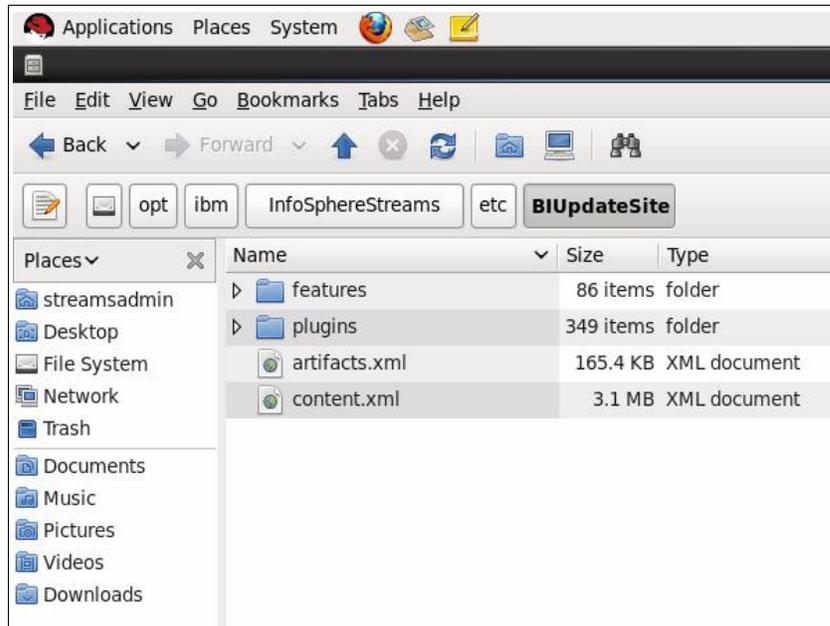


Figure 13-3 /etc/BIUpdateSite under the Streams Studio installation directory

To install the text analytics tools software from this location, complete the following steps:

1. From the menu bar select, **Help** → **Install New Software**.
2. If you get the No Updates Found message displayed in Figure 13-4, click **Yes**. (This message is issued only if you have not previously configured an update site; the message is not an error.)

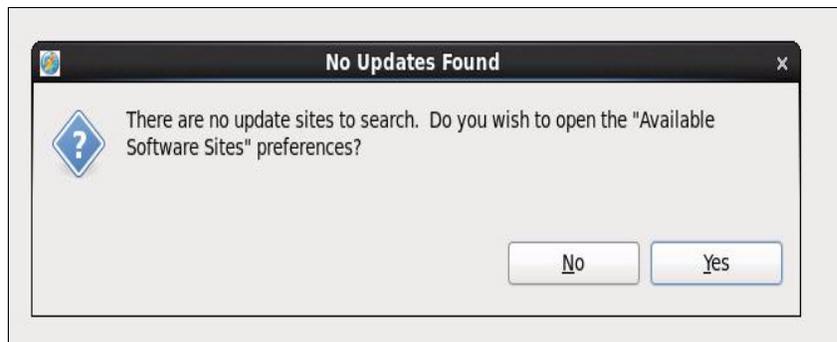


Figure 13-4 Result of first attempt, no update sites currently configured

3. In the Available Software Sites dialog box click **Add**.
4. The Add Repository dialog window opens (Figure 13-5).

Attention: Do not click Local or Archive here, until you understand why.

- ▶ Click **Archive** when you are installing a ZIP file or JAR file (or a URL to a ZIP file or JAR file).
- ▶ Click **Local** when installing from a directory.

If you make a mistake here, you may install only a subset of the files you need. You will not be able to complete your task accurately.

This ZIP or JAR file or directory issue is something we inherit from Eclipse.

We want to install from a directory, so we click **Local**.

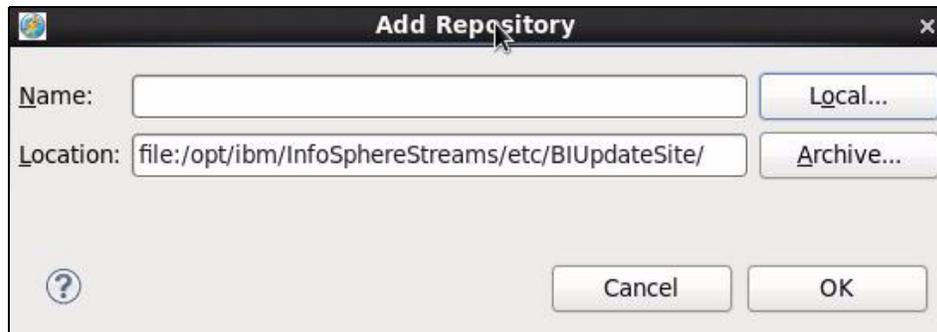


Figure 13-5 It matters whether you choose Local or Archive here

5. After clicking **Local**, browse to the Streams Studio parent installation directory (/etc/BIUpdateSite). Click through to produce the display in Figure 13-6 on page 389.

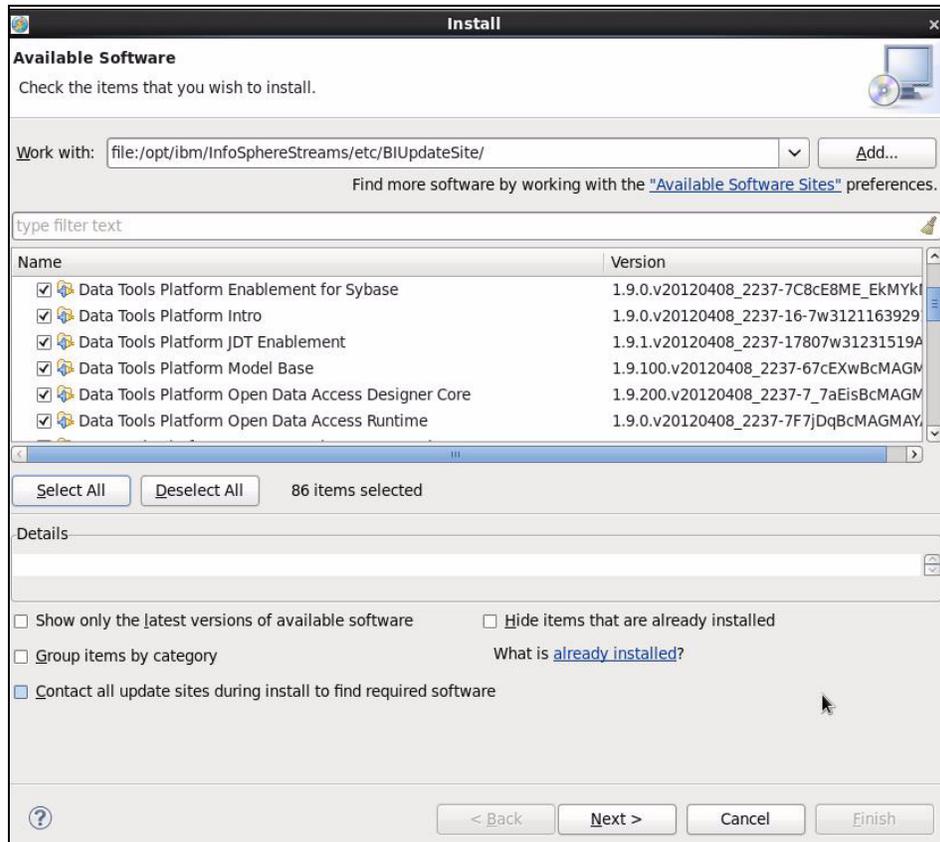


Figure 13-6 The prior step produces these options

6. Click **Select All**, and then click **Next**. This action can produce the display in Figure 13-7 on page 390. Figure 13-7 on page 390 has Eclipse removing duplicate values, checking dependencies, and other corrections.

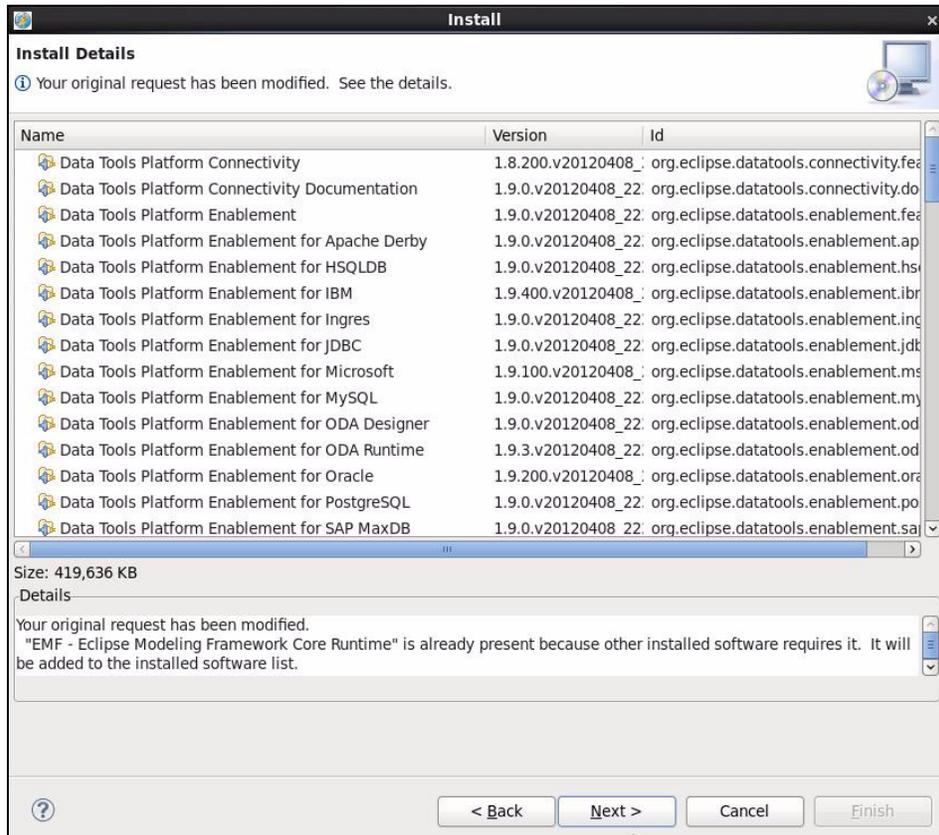


Figure 13-7 Eclipse checking dependencies

7. Figure 13-8 on page 391 shows the final step before committing changes.

Click, **I accept the terms of the license agreements**, and then click, **Finish**.

Note: You might receive more prompts after you click **Finish**, during the installation process. Generally each of these prompts are asking if you are sure you want to proceed.

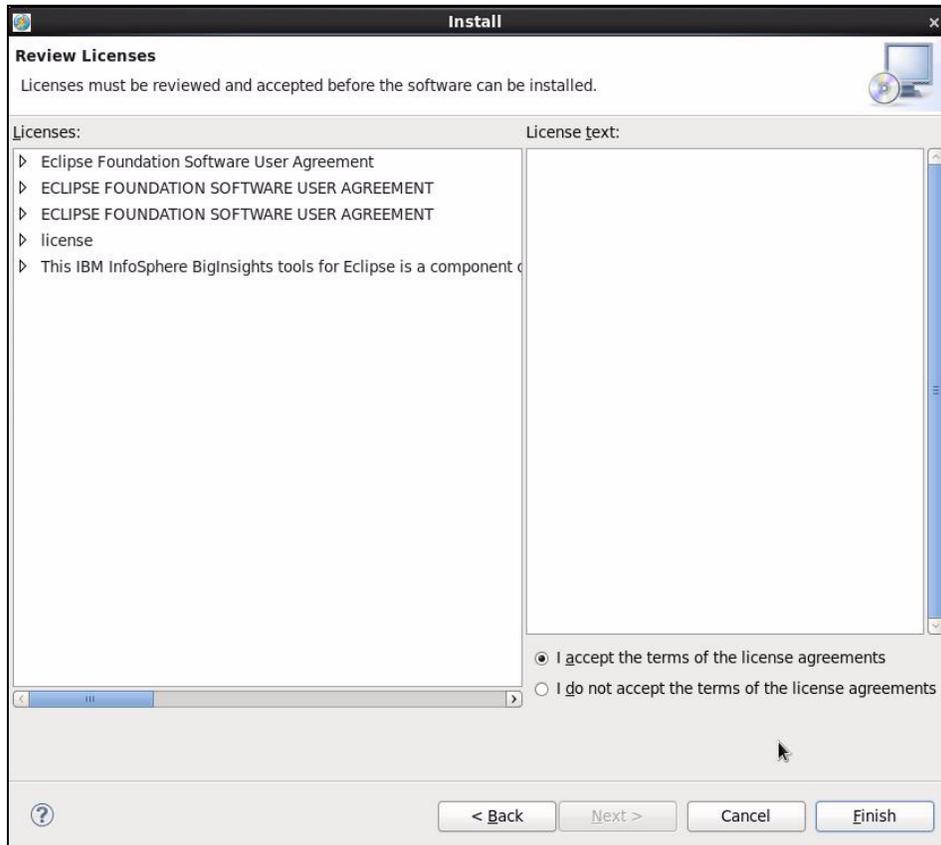


Figure 13-8 Final step before actually committing changes

The results of a successful installation are displayed in Figure 13-9.

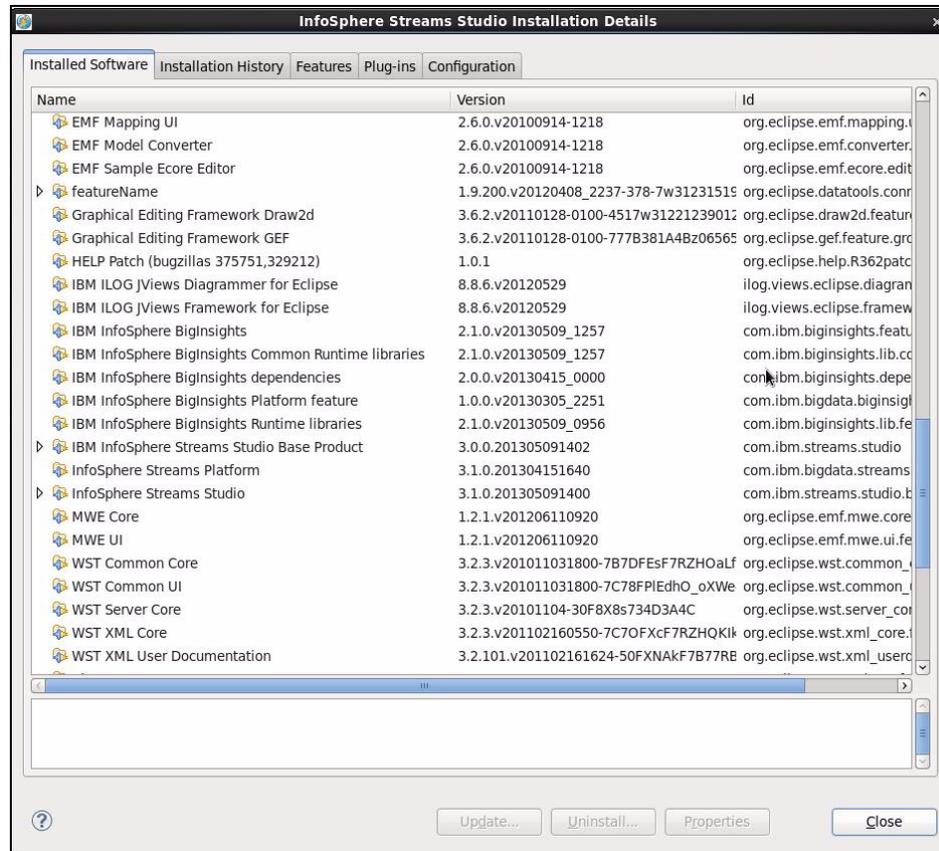


Figure 13-9 Results of successful install

13.2.2 Installing text analytics tools from a BigInsights install

We installed the text analytics tools software from the Streams Studio installation. The advantage there is you are certain to install the exact version of text analytics tools software that was tested and included with Streams.

In this section, we install the text analytics tools software from an IBM BigInsights installation. The advantage here is you are installing the same version of tools used by the BigInsights developers that might be active at your site. Text analytics assets, AQL assets, can be shared by Streams and BigInsights developers.

You might have these questions:

- ▶ Why not use the same version of product?

The disadvantage is you may be mixing versions of products that are not supported.

- ▶ Why do we sometimes force this issue?

The BigInsights distribution of text analytics tools may be many months newer, depending on the observed software release cycle.

Note: As always, you are welcome and encouraged to check with IBM technical support for guidance.

Figure 13-10 shows a page from the IBM BigInsights online information center (the web based documentation center to IBM BigInsights, the IBM Hadoop related product.) This information center entry details how to install the text analytics tools using a variety of methods.

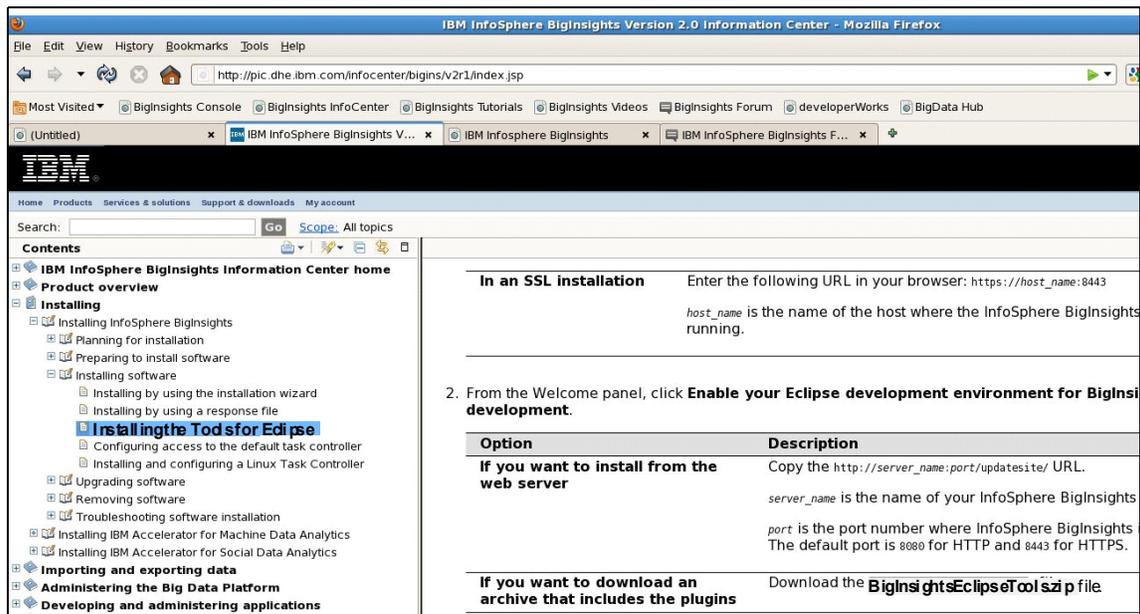


Figure 13-10 IBM BigInsights online information center

To install the text analytics tools software from a BigInsights installation file distribution, complete the following steps:

1. In the BigInsights web console Welcome page (Figure 13-11), click **Enable your Eclipse development environment for BigInsights application development**.

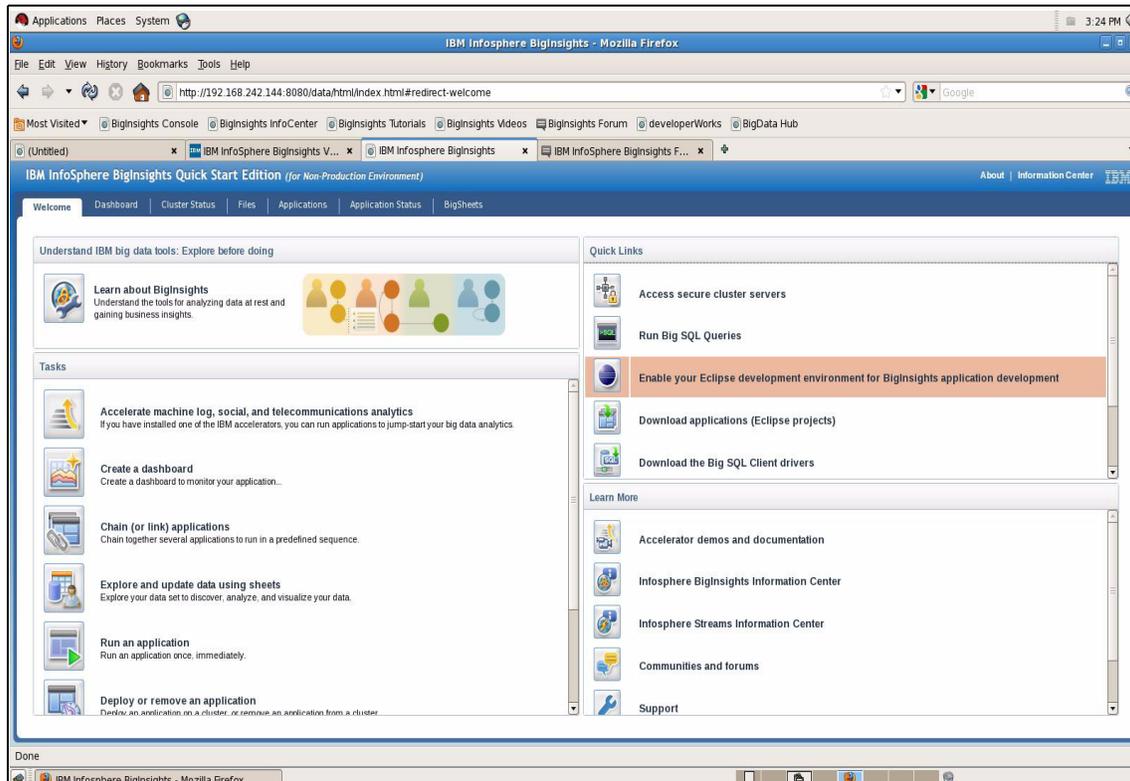


Figure 13-11 Big Insights web console, Welcome page

2. In the next window (Figure 13-12), click the **BigInsightsEclipseTools.zip** link (to download the file), and then navigate to the directory where you want to save this file.

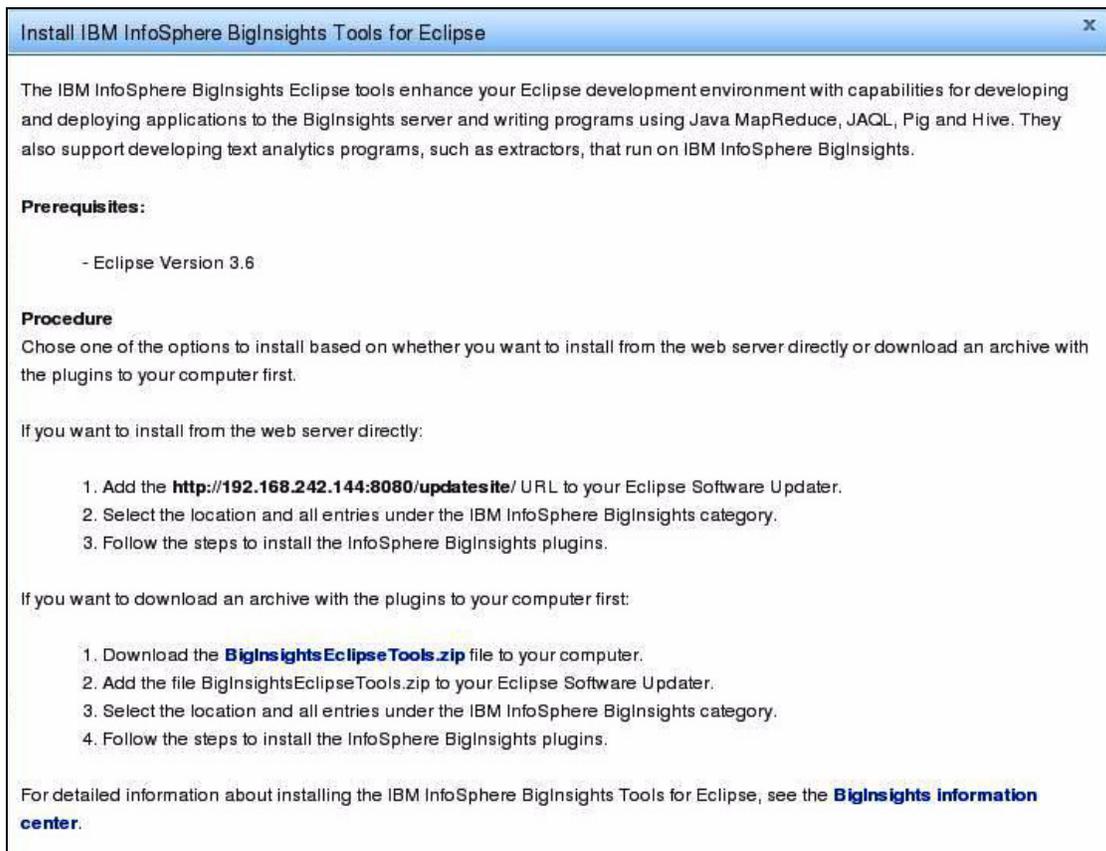


Figure 13-12 Saving the BigInsightsEclipseTools.zip file

3. From the menu, select **Help** → **Install New Software**.
If you get the No Updates Found message displayed in Figure 13-4 on page 387, click **Yes**. (This message is issued only if you have not previously configured an update site; the message is not an error.)
4. In the dialog box that opens, browse to the directory where you saved the BigInsightsEclipseTools.zip file. See Figure 13-5 on page 388. Because we want to install from a ZIP file, we click **Archive** here, and not click **Local**.

Local versus Archive: In 13.2.1, “Installing text analytics tools from Streams Studio installation” on page 386, we installed from a directory, so we clicked **Local**. Here we are installing from a ZIP file, so we click **Archive**.

5. Click through to accept the licensing terms and so on. After you click **Finish**, you might receive further prompts; again, click through to complete the installation.

13.2.3 Configuring your Streams project, add a BigInsights project

To perform our text analytics programming, we use two concurrent projects inside Streams Studio:

- ▶ A Streams project to run our AQL integrated with Streams
- ▶ A BigInsights project to develop and test our AQL

For the Streams project, complete the following steps:

1. In Streams Studio, in Streams Explorer view, right-click **Add Toolkit Location**, and browse to the toolkits directory on the hard disk, under the Streams installation directory. The default is `/opt/ibm/InfoSphereStreams/toolkits`.

Click to completion (click **OK** twice). The next window opens (Figure 13-13 on page 397).

This step must be completed once for a given Streams Studio installation.

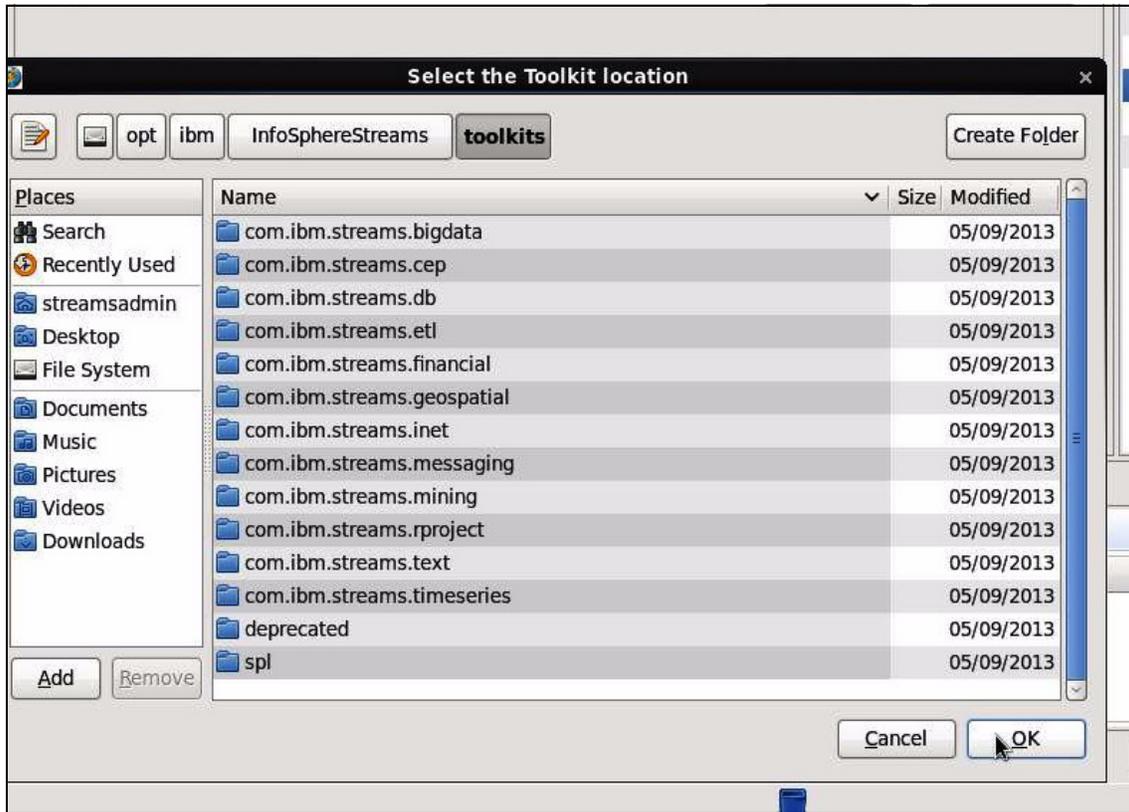


Figure 13-13 Adding a toolkit to a Streams project

2. In the Project Explorer view, right-click the Streams project and click **Add the BigInsights Nature** → **Finish**.

This step must be completed once for every Streams project for which you want to run text analytics, AQL.

3. In Project Explorer view, right-click the Streams project and select **Edit Toolkit Information** → **Dependencies** [tab] → **Add** → **Browse** → (**highlight the com.ibm.streams.text entry**) → **OK** → **OK** → **OK**. You will see an example as displayed in Figure 13-14 on page 398.

This step must be completed once for every Streams project for which you want to run text analytics, AQL.

Note: This step is how we add any Streams toolkit to a Streams project. In this specific case, we are adding the Text Analytics toolkit.

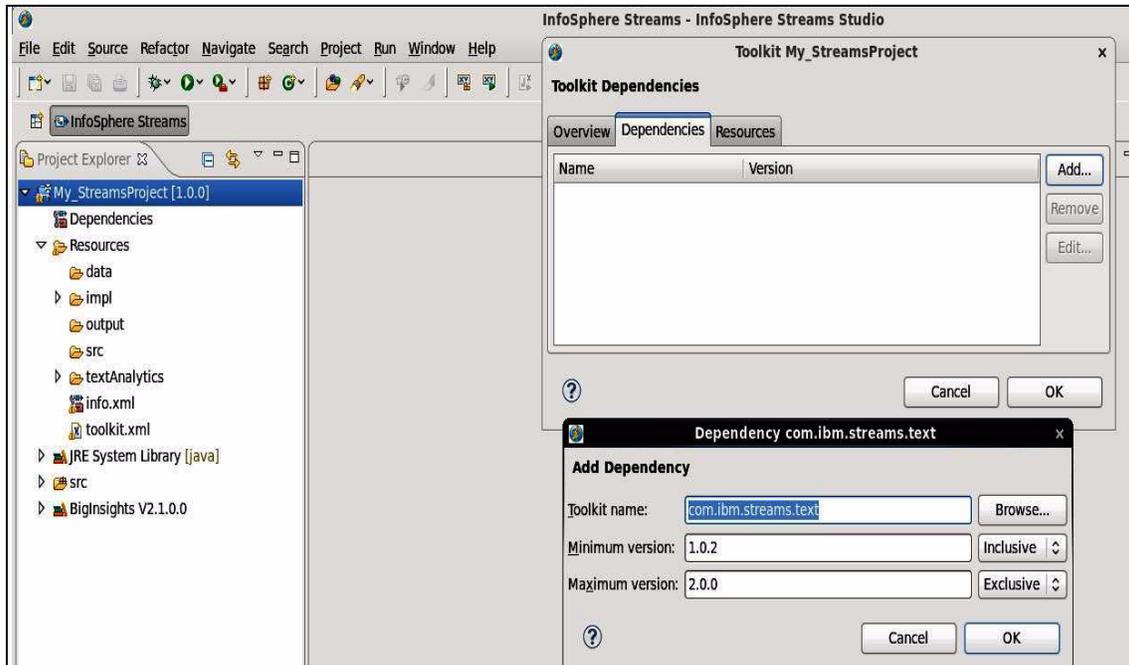


Figure 13-14 Adding a toolkit to a project

At this point, we have a Streams project that meets all prerequisites to run text analytics, to run an AQL script as part of a Streams application. The operative phrase here is *run*. While we run our AQL script inside a Streams project, we *develop* this AQL script within a BigInsights project, because this is where the text analytics tools reside.

For the BigInsights project, complete the following steps:

1. From the Streams Studio menu bar select **File** → **New** → **Project** → **BigInsights** → **BigInsights Project** → **Next**. Provide a name for the project and then click **Finish**.

If you receive a prompt to change the Eclipse perspective, click **Yes**.

Note: You can change perspectives, between BigInsights and Streams, from the Eclipse tool bar, or from the menu bar by selecting **Window** → **Open Perspective** → **Other**.

2. In the Project Explorer view, under your newly created BigInsights project, locate the textAnalytics/src folder. Right-click this folder and select **New** → **Other** → **BigInsights** → **AQL Module** → **Next**. See Figure 13-15 on page 399.

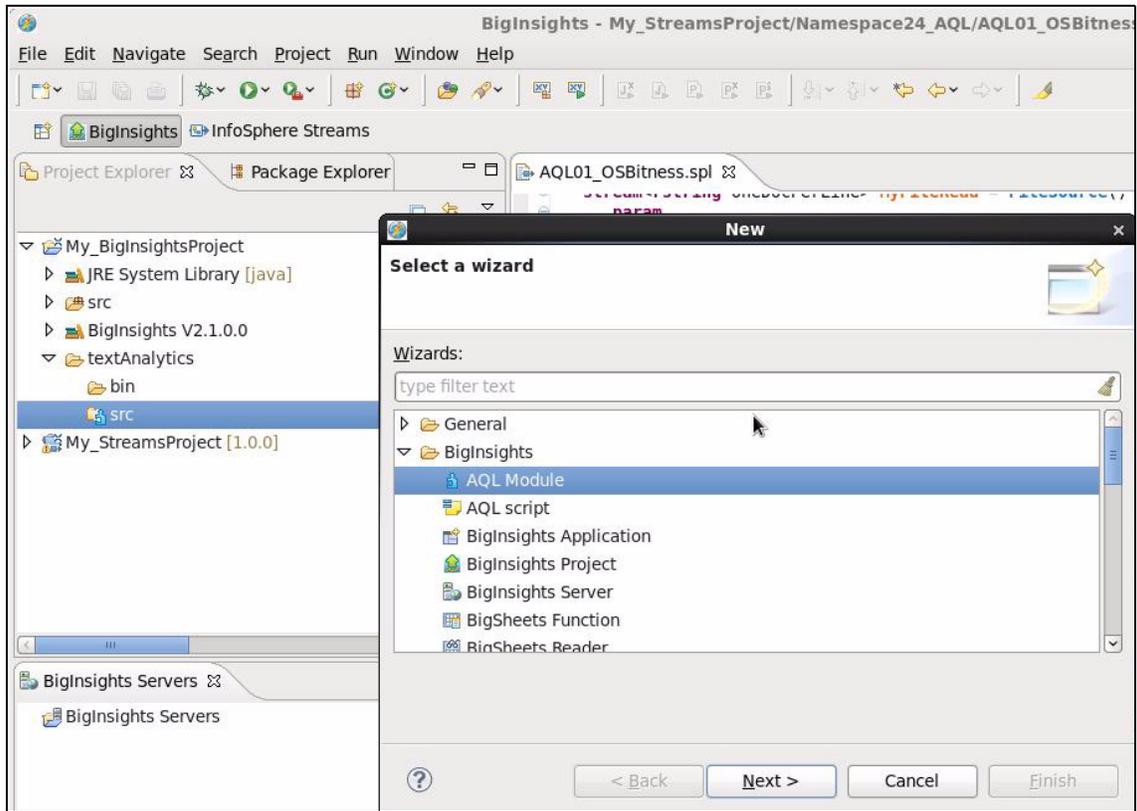


Figure 13-15 Making an AQL module, step 1 of 2

Provide a name for the module (ours is `My_AQLModule01`), and then click **Finish**. See Figure 13-16.

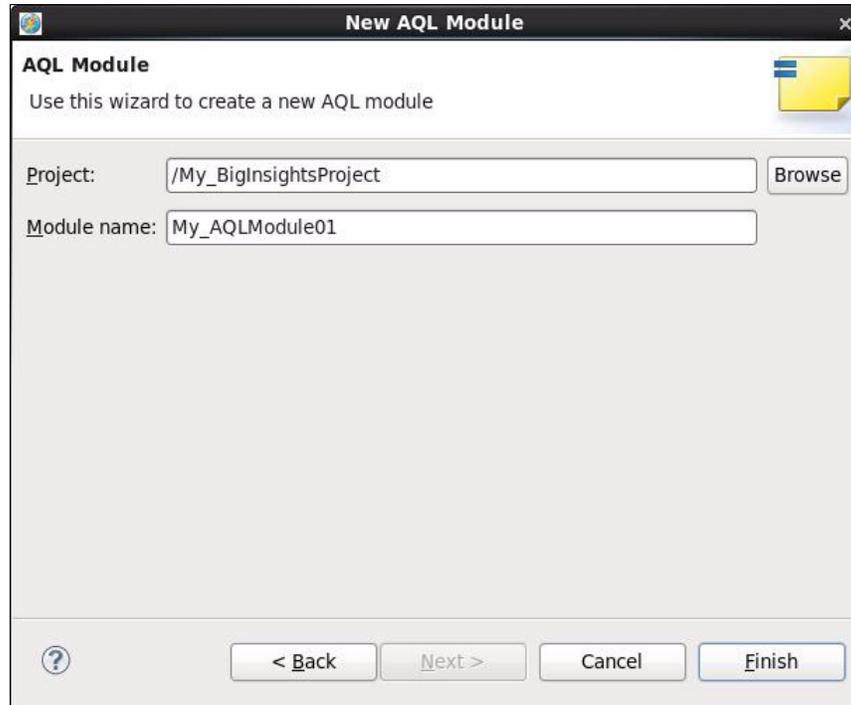


Figure 13-16 Making an AQL module, step 2 of 2

Note: AQL modules are like Java packages, or Streams namespaces, they are just folders or directories. AQL modules offer the ability to group AQL source code, and then later import or export, and reuse previously created AQL assets.

3. Right-click your newly created AQL Module (`My_AQLModule01`) and then click **New** → **Other** → **AQL Script** → **Next**. Provide a name for this script (ours is `My_AQLScript.aql`), as shown in Figure 13-17 on page 401, and then, click **Finish**.

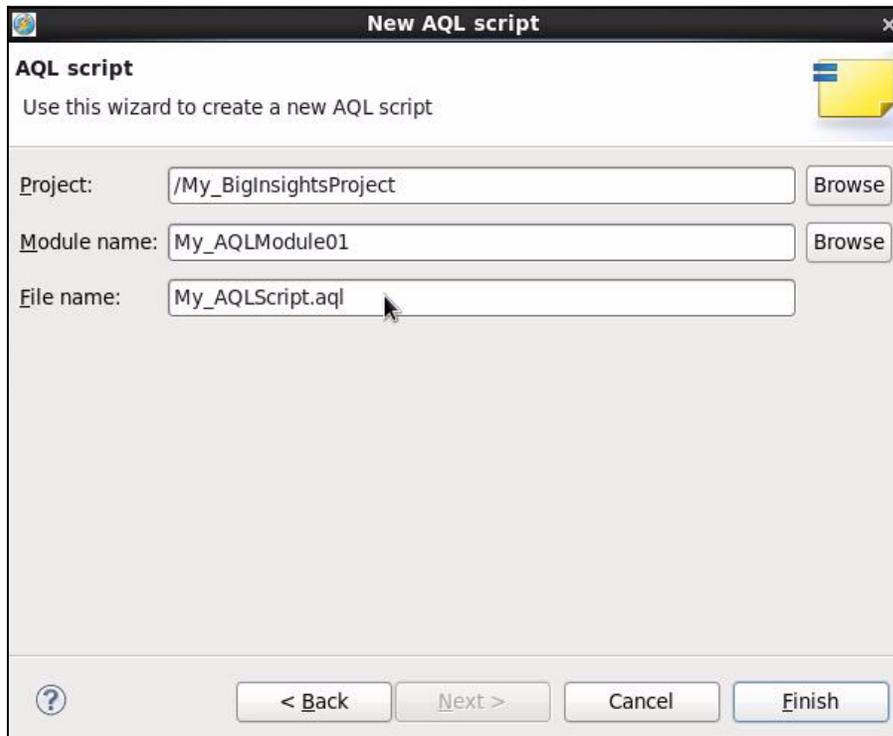


Figure 13-17 Making an AQL script object

File naming: AQL scripts contain the actual AQL language source code, and are where you perform your AQL programming. AQL scripts are ASCII text files and have `.aql` as a file name extension. When a single AQL file or set of AQL files are compiled, they are output in a single file with a `.tam` file extension. TAM is short for text annotation module

In our example, the actual compiled file name would be `My_AQLModule01.tam`, containing each of the AQL scripts located in the AQL module, and being named for the AQL module. (Our sample AQL module was named, `My_AQLModule01`. If the AQL module had been named `Gary`, and it contained two AQL scripts named `Dog.aql` and `Cat.aql`, the TAM file that is output would be named `Gary.tam`.)

In prior versions of AQL, the TAM file had `.aog` as the file extension. AOG is short for annotation operator graph. We mention this so that you can more easily identify older AQL related documentation or articles. If you see reference to AOG, you are viewing older documentation.

At this point you have a BigInsights project, an AQL module (which is a requirement), and an AQL script (an AQL source file). If you continue editing the AQL source file, next you need to know how to run or test it.

Add AQL commands to the AQL script: The steps that follow instruct you how to run an AQL script (an AQL command file) inside your BigInsights project. Thus far we have covered creating an AQL file, which still sits empty.

You will need to put AQL commands in the file, then save it by pressing Ctrl+S, and then proceed with the steps that follow.

In the BigInsights project, complete the following steps:

1. In the AQL editor view, right-click and select **Run As** → **Run Configurations** → **Text Analytics**. Click the **new** button, which has a plus sign icon on top.

The result is shown in Figure 13-18 on page 403.

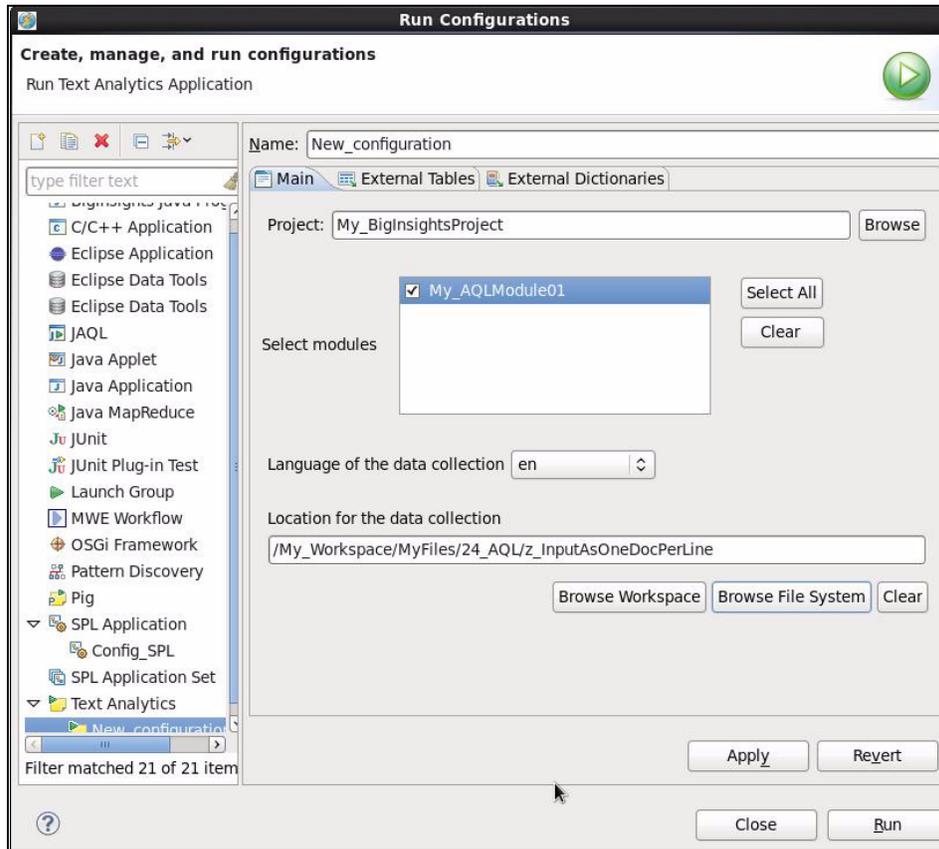


Figure 13-18 Creating a text analytics configuration run file

2. Complete the fields in the Main page of the window (Figure 13-18); the External Dictionaries and External Tables pages will be completed later.

The **Location for the data collection** field can be a directory or a named file. Consider the following information about the value to enter in this field:

- As you will see, AQL operates with the concept of a *document*. Document is the central object to AQL, and provides a reference to that text that you are processing with AQL.
- In our first sample use case, we are processing log file entries from an Apache HTTP server. Most commonly we would build (test) our Streams application reading first from a single file of sample data; each record in the sample data file is separated from the next by a new line character. If we first build a Streams application with a FileSource, and a FileSink, and some functor in the middle, Streams would send each line from the input

file to the functor, where the functor would process each line received, one by one.

- AQL expects a single document or set of documents, each of which can contain many new line characters.
- In the Main tab, if we specify the single sample input data file, then this document will be sent in its entirety to AQL, in the manner that a Streams functor receives one line. For development only, we should probably split our sample input data file into a collection of data files, each containing only one line. Then, in the Location to the data collection field, we should specify the directory containing this collection of files.

Again, this is for development only.

- Further, these input data files should have a `.txt` file name extension. To split and also name these files, we use the Linux text processing utility named, `split(C)`.

3. After completing the Main page (Figure 13-18 on page 403), click, **Apply**, and then click **Run**.

The Annotation Explorer view opens (Figure 13-19 on page 405).

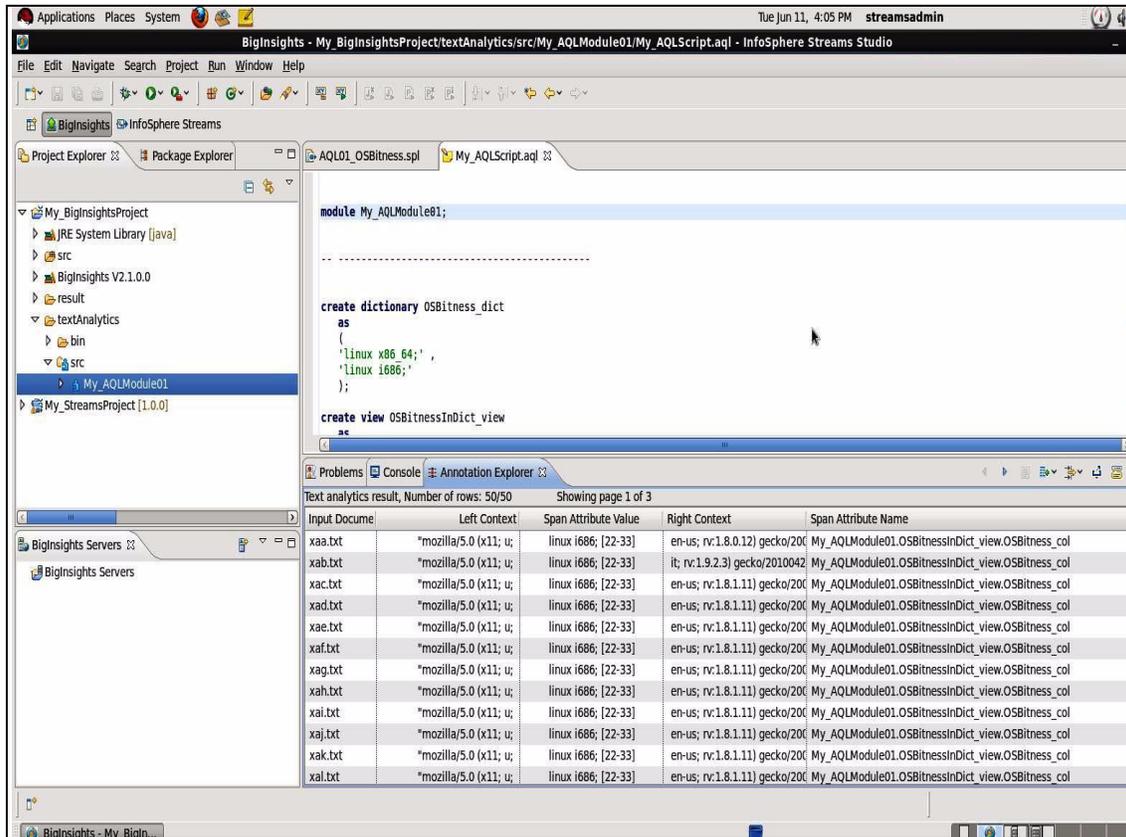


Figure 13-19 Annotation Explorer view, result of AQL script run

Note: The results from an AQL script run do not always display in the Annotation Explorer view, and the Annotation Explorer view might be empty. Whether the Annotation Explorer view contains data is the direct result of the AQL command you have run.

Some AQL commands output in a view that is available from the toolbar of the Annotation Explorer (see Figure 13-20). The red arrow in the figure highlights where the drop-down list is that has further views containing AQL run results.

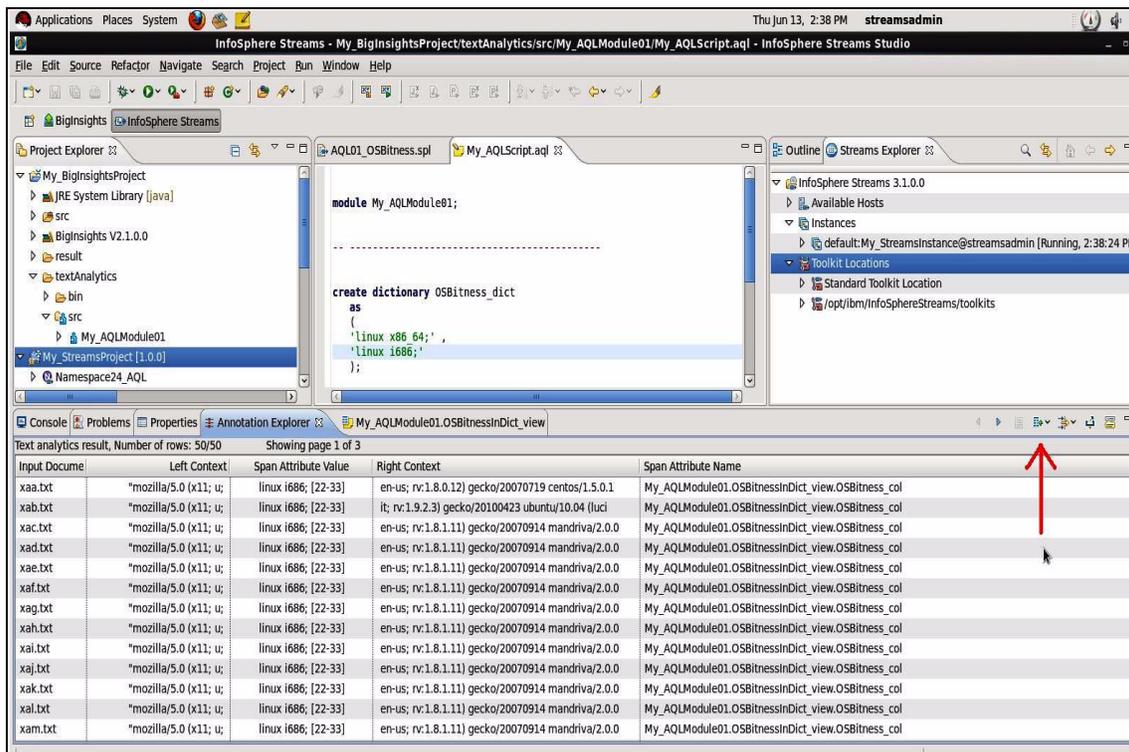


Figure 13-20 Red arrow showing location in tool bar that we have interest in

Figure 13-21 displays data output from an AQL run that might not display in Annotation Explorer. This view is from the toolbar in Annotation Explorer.

The screenshot shows the InfoSphere Streams Studio interface. The main editor displays the following AQL script:

```

module My_AQLModule01;

.....

create dictionary OSBitness_dict
as
{
  'linux x86 64;' ,
  'linux i686;'
};
  
```

The Annotation Explorer at the bottom shows the output of the AQL run. It displays a table with 50 rows (page 1 of 3) showing the results of the dictionary creation. The table has three columns: OSBitness_col (SPAN), Input Document, and a column with an 'Explain' link.

OSBitness_col (SPAN)	Input Document	Explain
linux i686; [22-33]	xaa.txt	Explain
linux i686; [22-33]	xab.txt	Explain
linux i686; [22-33]	xac.txt	Explain
linux i686; [22-33]	xad.txt	Explain
linux i686; [22-33]	xae.txt	Explain
linux i686; [22-33]	xaf.txt	Explain
linux i686; [22-33]	xag.txt	Explain
linux i686; [22-33]	xah.txt	Explain
linux i686; [22-33]	xai.txt	Explain
linux i686; [22-33]	xaj.txt	Explain
linux i686; [22-33]	xak.txt	Explain
linux i686; [22-33]	xal.txt	Explain
linux i686; [22-33]	xam.txt	Explain

Figure 13-21 Viewing data from AQL run that was not available in Annotation Explorer

13.3 First AQL example, Apache HTTP log file

In this section, we build an AQL script to extract specific values from an Apache HTTP log file. After our AQL script is working, we describe how to create a Streams application that integrates with, and runs this same AQL.

This section assumes you have no AQL knowledge, and no knowledge of Streams integration with AQL. However, this section does assume you completed the previous steps to install the text analytics tools, configured the Streams project for text analytics, created a BigInsights project, and created a text analytics runner, all described previously in this chapter.

Example 13-1 on page 381 the input data we want to process. For development we prefer to have two copies of this data:

- ▶ For AQL, we want a directory of files, with one record per file, of the sample data.
- ▶ For Streams, the easiest approach is to have one file, with all of the sample data records, one record per line, multiple lines reflecting multiple records.

Complete the following steps (see Figure 13-22 on page 409):

1. In Streams Studio, go to `My_BigInsightsProject` in the Project Explorer view. Under `textAnalytics/src`, create an AQL module named `My_AQLModule01`. Within the module create an AQL script named `My_AQLScript.aql`.
2. Provide the AQL source code as displayed in the AQL Editor view.
3. Press `Ctrl+S` to save the AQL source file, and complete the steps to run this AQL script. The are displayed in the Annotation Explorer view (also shown in Figure 13-22 on page 409).

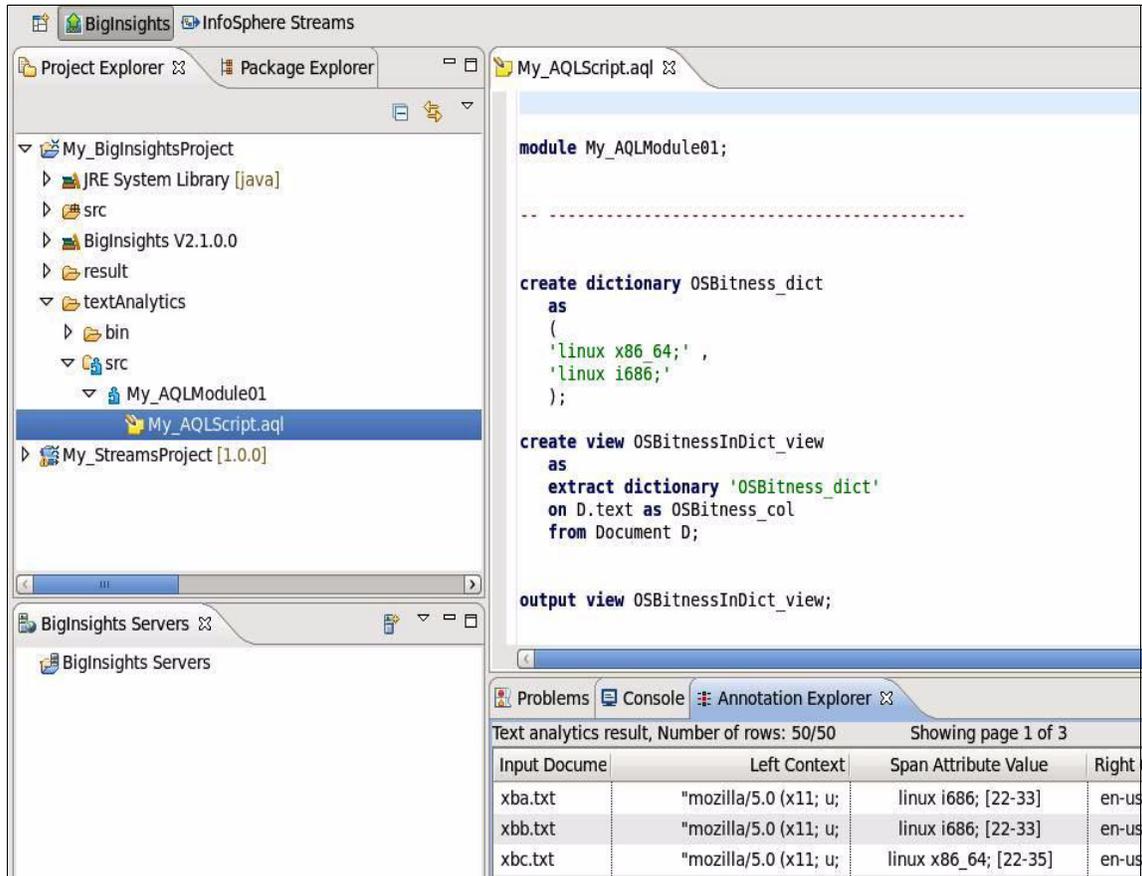


Figure 13-22 First AQL example, including output

The AQL script (Figure 13-22) has four AQL commands:

- ▶ module

This is required and it names the AQL module, of which this AQL script is a member.

- ▶ create dictionary

We use AQL dictionaries when we have a well known list of words of manageable size that we are searching for. This example lists keywords we use to identify the “bitness” of the user agent making the HTTP request: 32-bit (linux x86_64) or 64-bit (linux i686).

Note: Now we are concerned only with testing for Linux entries; Figure 13-22 on page 409 shows that you can have white space in a dictionary entry (a multi-word dictionary entry).

Later, if we discover that Windows and MAC also have entries similar to “MAC x86_64” or “Windows i686”, we may split this value into two dictionaries; one for OS_Major (Linux, MAC, Windows) and one for bitness (i686, x86_64).

If MAC and Windows did offer this type of data, using two dictionaries is considered the best practice, offering a more robust solution that is less brittle.

Our analysis of the sample data revealed that Linux entries contained these two keywords to indicate bitness, with high certainty. A small percentage of sample data did not indicate bitness, which we address later.

This dictionary is hard-coded inside our AQL script, and thus evaluated at compilation time. In a later example, we use an external dictionary, which will allow the dictionary to be included (compiled) at run time.

► `create view`

This view reads from Document, which is an AQL standard object supplying reference to the input sent to AQL. Document is always in scope so we need not do anything to define it.

This `create view` statement calls to look for matches of entries in the named dictionary, in the text found in Document (the input document).

We give the name `0SBitness_col` to the single column that is output.

Cardinality: For every document examined by this AQL script, the `create view` statement can find zero, one, or more matches. Even as our Streams application sends one row (still a Document to AQL), the `create view` can find zero, one, or more matches.

If our test case had 100 input documents, 45 matching on `linux i686` and 50 matching on `linux x86_64`, we would receive 95 results in the Annotation Explorer view, not 100. If a document has both `linux i686` and `linux x86_64` (our research indicates that this should not happen), this document would output both matches.

This is a condition we detail throughout the remainder of this example; we create AQL code specifically to manage “not-found” type records.

► `output view`

This causes the output of this view to be available to consumers outside this AQL script. If this view existed only to be read by other AQL views, we would not need this `output view` statement.

At this point, the first version of our AQL script is functioning. Now we can incorporate our AQL with a Streams application.

Complete the following steps:

1. Create a Streams namespace named `Namespace24_AQL`.
2. Inside this namespace, create a Streams application named `AQL01_Bitness`.
3. Copy the AQL script (created in the previous steps) to the following directory:
`/My_Workspace/MyFiles/24_AQL/MyAQL_Module01`

Note: Although creating a redundant copy of the AQL script is not a requirement, it is our preference.

We can make an absolute or relative reference to the AQL script as it exists in the BigInsights project. There is also a means to share compiled or uncompiled assets between the BigInsights project and the Streams project. This topic is not expanded upon further here.

4. Enter the SPL program code that is shown in Figure 13-23.



```
namespace Namespace24_AQL;

use com.ibm.streams.text.analytics::TextExtract ;

composite AQL01_OSBitness {
  graph

  stream<rstring oneDocPerLine> MyFileRead = FileSource() {
    param
    file : "/My_Workspace/MyFiles/24_AQL/AQL01.Input.txt";
    format : line;
  }

  stream<rstring OSBitness_col> MyTextExtract = TextExtract(MyFileRead) {
    param
    inputDoc : "oneDocPerLine" ;
    outputMode : "multiPort" ;
    //
    uncompiledModules : "/My_Workspace/MyFiles/24_AQL/MyAQL_Module01";
    moduleOutputDir : "/My_Workspace/MyFiles/24_AQL/MyAQL_Module01";
  }

  () as MySink = FileSink(MyTextExtract) {
    param
    file : "/My_Workspace/MyFiles/24_AQL/AQL01.Output.txt";
    format : csv;
  }
}
```

Figure 13-23 Listing for AQL01_Bitness.spl

Consider the following information about the Streams application in Figure 13-23:

- ▶ The namespace line is standard.
- ▶ The use line provides reference to the Streams Text Analytics toolkit.
- ▶ The FileSource and FileSink operators are standard. The result of the FileSource with a line parameter is that each line in the input source will be passed to the TextExtract operator as a single document, the standard AQL reference object.

- ▶ Regarding the TextExtract operator, consider this information:
 - In this context, the `inputDoc` parameter is optional.

This parameter is used when multiple input columns are sent to the TextExtract operator. This parameter is used to indicate which input column should serve as the Document to the AQL script.
 - The `outputMode` parameter operates in conjunction with the `passThrough` parameter. The `passThrough` parameter is optional, and not currently displayed in the example. In effect, these two parameters together can allow multiple AQL output view statements to be resident in a single or set of AQL scripts, and consumed by one Streams TextExtract operator.

As configured, the `outputMode : multiPort` supports a single AQL output view statement.
 - The `uncompiledModules` parameter states the absolute or relative directory path name to the source AQL scripts. You specify a directory name here. You do not specify an AQL script file name, because all AQL scripts in a given AQL module are compiled together and output as one TAM file.

Note: As we configured this example, we are supplying the (source) AQL scripts, which will be compiled for us. It is possible to supply a (precompiled) TAM file. This topic is not expanded upon further here.
 - The `moduleOutputDir` parameter indicates the directory where the resultant AQL TAM file is output.

Running the Streams application will output to the named resultant data file, as the Streams application is constructed. Sample output is shown in Figure 13-24 on page 414.

Unknown bitness: Recall that our example is currently (functionally) incomplete.

In its current state, our example returns user agents that are only 64-bit or 32-bit. In its current state, our example does not return user agents with an unknown bitness. This is a condition we address as the example develops.

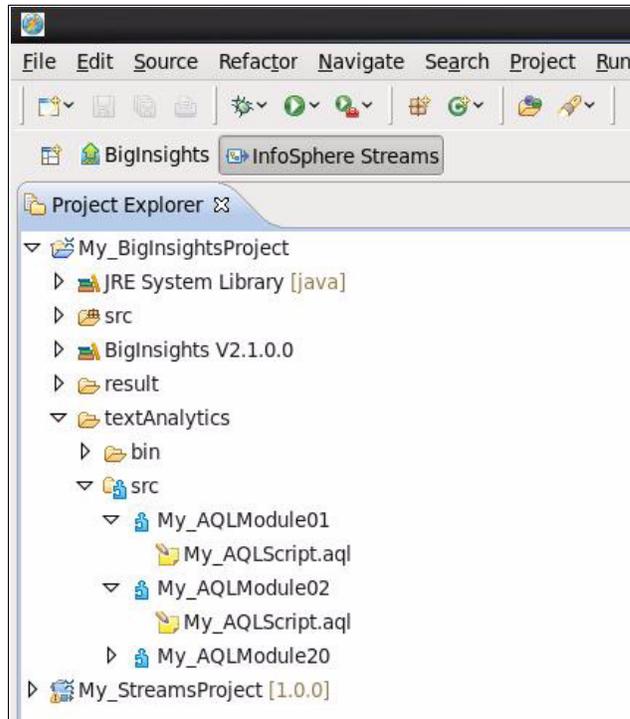


Figure 13-25 Creating a new AQL module to avoid naming conflicts

2. Create a new AQL script (we named ours `My_AQLScript.aql`) in the new AQL module. Edit this AQL script similar to the script that is displayed in the next five figures (Figure 13-26 on page 416 through Figure 13-30 on page 425; this script is shown in five parts).

A AQL script code review follows each part of the display.

```
My_AQLScript.aql 23
-- "mozilla/5.0 (x11; u; linux i686; it; rv:1.9.2.3)
--   gecko/20100423 ubuntu/10.04 (lucid) firefox/3.6.3"

module My_AQLModule02;

-----

create dictionary OSBitness_dict
as
(
  'linux x86 64;' ,
  'linux i686;'
);

create view OSBitnessInDict_view
as
extract dictionary 'OSBitness_dict'
on D.text as OSBitness_col
from Document D;

create view OSBitnessInDictCased_view
as
select
  case
  when Equals(GetText(V.OSBitness_col), 'linux i686;') then '32'
  when Equals(GetText(V.OSBitness_col), 'linux x86_64;') then '64'
  else
  as OSBitness_col
from OSBitnessInDict_view V;
```

Figure 13-26 Second version of our AQL script, part 1 of 5

Consider the following information about AQL script, part 1, in Figure 13-26:

- ▶ Remember, this one AQL script extends from Figure 13-26 through Figure 13-30 on page 425.
- ▶ The first two lines are comments, and offer a sample of the data (the input document) we expect to receive.
- ▶ The module line has changed since the first example (in Figure 13-22 on page 409).
- ▶ The first create dictionary and first create view statements have not changed since the first example.

- ▶ Regarding the `create view OSBitnessInDictCased_view` line, note the following information:
 - The first view (`OSBitnessInDict_view`) extracted dictionary matches from the input document. This second view (`OSBitnessInDictCased_view`) reads from the first view, informing us that AQL views can read from other AQL views.
 - The design intent of this second view is to change the matching dictionary entry to a value we prefer. We do this purely for style.

In this AQL view, we see details related to the AQL case statement. (Entirely similar in syntax and purpose to the SQL CASE statement.)

AQL data types: The first AQL view returned the input document match from our dictionary using the AQL data type of SPAN. An AQL SPAN data type is similar to what other environments call a record or structure (a multi-value, multi-variable data type.)

In this context, the SPAN contains the matching text, and numeric variables to indicate the beginning and ending index in the input document, where this matching text was found.

As an example, perhaps we match the word `Hey` from the input document, `123Hey789`. The resultant SPAN variable will contain the word `Hey`, because the match was found, then the numeric value `4`, because this is where the matching word begins, and then the numeric value `7`, because this is where this matching word ends.

Why do you care? Because our case statement wants to match on the text that was found, and we must extract this word from the SPAN data type using the `GetText()` built-in function to AQL.

- What is the value `-2` (minus 2) doing in the case statement?

This is a value of our own design; later you will see we choose to use the value `-1` (minus 1) for another purpose.

The `-2` value in the case statement handles the use case where our dictionary returns a value, that our case statement did not expect. That is, if a new value is added to our dictionary later, and this new value is not reflected (coded) in this case statement.
- ▶ Notice that views do not need an accompanying `output view` statement. Output view is not needed in the context of an AQL script. Output view is used to allow the output from a view to be referenced by a consumer of our AQL script.

Optimizer and view: There is an optimizer to AQL, just as SQL has its query optimizer. The AQL optimizer forms an AQL execution plan at the time the AQL script is compiled. The AQL optimizer determines how to best execute an AQL script at run time: memory utilization (paging), reduction in CPU consumption, and others.

- ▶ What about commenting out or removing AQL output view statements that are used for development, but then not consumed?

Ultimately the AQL optimizer cannot know if, for example, the consumer of an AQL script does not use the data output by an output view statement. Thus, if you do not need an AQL 'output view' statement, comment it out, or delete it.

- ▶ How does the AQL optimizer plan to reduce CPU consumption?

AQL views are logical; they are not instantiated until time of execution.

In an example where view V3 reads from view V2, which reads from view V1, and only view V3 is output, the AQL optimizer will combine and collapse the view definitions from V3, V2, and V1. Among other responsibilities, the AQL optimizer will remove redundant predicates (that is, each view calling to apply the same filter).

Much like a SQL optimizer, the AQL optimizer applies the most efficient and most restrictive filters first, thus reducing the amount of data from the input document that must be examined.

- ▶ Why do we mention this?

It is considered best practice to create the simplest, most compact simple views, then combine views on views to achieve the result you want.

The reason is because it makes your AQL code easier to read and support. Complex AQL might have enough parenthesis, (not indentions) and so on, that only a more experienced AQL programmer can make sense of it. And, with the AQL optimizer, there is no runtime penalty.

```
My_AQLScript.aql ⌵

create view OSBitnessNotInDict_view
as
select
  '-1' as OSBitness_col
from Document D
where Not(ContainsDict('OSBitness_dict',D.text));

create view OSBitness_view
as
(select * from OSBitnessInDictCased_view)
union all
(select * from OSBitnessNotInDict_view);

-- output view OSBitness_view;

-----

create dictionary LinuxDistro_dict
as
(
  'Fedora/'      ,
  'Ubuntu/'     ,
  'Mandriva/'   ,
  'CentOS/'     ,
  'Linux Mint/' ,
  'AppleWebKit/' ,
  'Spinn3r/'
);
```

Figure 13-27 Second version of our AQL script, part 2 of 5

Consider the following information about AQL script, part 2, in Figure 13-27 on page 419:

- ▶ The first two `create view` statements complete extraction of the `OSBitness_col` column.
 - The first `create view` handles the case where an input document did not offer any match from our dictionary of known OS bitness keywords.

If we send 100 input documents to our AQL script, and 45 are 32-bit, and 50 are 64-bit, only 95 matches are found. Because 5 matches are not found and do not produce output through any of the earlier views.

The first `create view`, named `OSBitnessNotInDict_view`, produces output for our 5 missing matches. And, we choose to output a value of `'-1'` to indicate a missing match. (For style, `'-1'` was simply a value we chose. We are using negative numbers because no computer program can currently have a negative value for its bitness.)
 - The `Not ContainsDict()` predicate is one means to accomplish our design goal. We can also use AQL set operands (comparing all documents to that list of documents that did have a match to this dictionary). In this case, the set operand we rely on is a *minus* clause, a topic that is expanded upon 13.6.1, “The kitchen sink AQL script and Streams application” on page 464.
 - The `create view` named `OSBitness_view` combines our found and not-found dictionary lists, using a `union all` clause, just like SQL. We will combine this `OSBitness_view` with other views and output below. This view, by its design, must output at least once for every input document sent to our AQL script.
- ▶ We create a second dictionary named `LinuxDistro_dict`. This dictionary is intended to match our input document for values related to the distribution of Linux that was observed. This dictionary is used in Figure 13-28 on page 421.

```
My_AQLScript.aql ⌵

create view LinuxDistroWithSlash_view
as
  extract dictionary 'LinuxDistro_dict'
  on D.text as LinuxDistro_col
  from Document D;

create view LinuxDistro_view
as
  extract regex /\w+/
  on v1.LinuxDistro_col as LinuxDistro_col
  from LinuxDistroWithSlash_view v1;

-- output view LinuxDistro_view;

-----
```

Figure 13-28 Second version of our AQL script, part 3 of 5

Consider the following information about AQL script, part 3, in Figure 13-28:

- ▶ The first view, named `LinuxDistroWithSlash_view`, is not syntactically different from the first view we created that performed a dictionary match.

With our first dictionary match, we used a case statement to change the values we received (`i686`, `x86_64`), to values we preferred (`32`, `64`). In this example, we use a regex expression to change values.

Note: Recall that our sample data comes in the form, “ubuntu/9.10”. Ultimately we want the “ubuntu”, and the “9.10”, but not the slash.

- ▶ The view named `LinuxDistro_view` exists for one purpose, to remove the slash from our earlier view.

As a regex expression, the `\w` indicates to return all letters, numbers and underscores (`\w` is short for word characters, such as letters, numbers, and underscores). The plus sign (+) indicates to return one or more of the characters. Because the forward slash (/) is not a word character, it is left behind.

At this point, we have the following two examples

```
create view .. extract dictionary
create view .. extract regex
```

These two AQL `create view` statements are the primary means with which we identify words we have interest in, from our input document. With these two statements, we seek to find all possible words we want to extract, including those that might be false positives.

With an additional AQL statement, `create view .. extract pattern` (see Figure 13-29 on page 423), we can look for patterns. For example, we can look for `ubuntu/9.10`, a dictionary match followed by a precisely formatted (version) number, which we can find with a `regex` statement.

We use patterns to remove false positives. We can use patterns on two words, or any number of words. We can use patterns on adjacent, or even nearby words.

We talk about matches, and false positives, but what about false negatives? *False negatives* indicate that your `create view .. extract dictionary` and `create view .. extract regex` need work; these views, or additional views, need to grab more words.

Precision and recall: In a more formal text analytics discussion, these topics are referred to as precision and recall. We strive for 100% precision and 100% recall, but that is not always possible, or is not always possible given time and labor constraints.

Semi-structured data is easier to analyze than unstructured data (text written by people, for people).

The higher you drive recall, the more likely you are to drive false positives. The higher you drive precision, the more likely you are to drive false negatives.

```
My_AQLScript.aql 83

create view AnyWordStartsWithDigit_view
as
extract regex /[0-9]+\S*/
on D.text as AnyWordStartsWithDigit_col
from Document D;

-- create view LinuxDistro_And_Version_view
-- as
-- extract pattern
-- <v1.LinuxDistro_col> <v2.AnyWordStartsWithDigit_col>
-- return group 0 as LinuxDistro_And_Version_col
-- from LinuxDistroWithSlash_view v1, AnyWordStartsWithDigit_view v2;

-- create view LinuxVersion_view
-- as
-- extract regex /[0-9]+\S*/
-- on v1.LinuxDistro_And_Version_col as LinuxVersion_col
-- from LinuxDistro_And_Version_view v1;

create view LinuxVersion_view
as
extract pattern
(<v1.LinuxDistro_col>) (<v2.AnyWordStartsWithDigit_col>)
return
group 2 as LinuxVersion_col
from LinuxDistroWithSlash_view v1, AnyWordStartsWithDigit_view v2;

-- output view LinuxVersion_view;
```

Figure 13-29 Second version of our AQL script, part 4 of 5

Consider the following information about AQL script, part 4, in Figure 13-29:

- ▶ This segment of this AQL script completes our extractions, the last of identifying words we want to return to our consumer.
- ▶ The AQL `create view`, named `AnyWordStartsWithDigit_view`, is a regex match. This regex expression returns any word that starts with a number, and then contains any non-whitespace character.

This regex matches values similar to 9.10, 3.03.14.UC4, 2_24.0, and many more. (From our use case, we are seeking to match on software version numbers.)

Note: This is the second regular expression (regex) we have used in this example. See 13.5, “Regular expressions (regex)” on page 436, and then also tooling (in 13.4, “Guided team-based AQL and using AQL tools” on page 432) to help you write and decipher regular expressions.

- ▶ Our input document can contain zero, one, or dozens of matches of this type (any word that starts with a number), producing many false positives.

The second view, named `LinuxVersion_view`, calls to match on a specific pattern, so that we are certain to use a number match, but only when preceded by a reference to the Linux distribution.

Return groups: The two views that are commented out (--) in Figure 13-29 serve to replace the final view, named `LinuxVersion_view`. The two views that are commented out offer different AQL syntax to accomplish the same goal.

Return groups are most commonly seen in extract pattern statements, although they are also available in extract regex statements.

In the view named `LinuxDistro_And_Version_view`, we match pattern on `LinuxDistro_col` followed immediately by `AnyWordStartsWithDigit_col`. The `return group 0` statement says to return the whole set of matching text.

In the view named `LinuxVersion_view`, we match on the same matching group, but then call to `return group 2`, which means return only the `AnyWordStartsWithDigit_col`.

Return group 0 returns the whole match, 1 returns the first word in the match, 2 returns the second, and so on.

We do not need to return the `LinuxDistro_col`, because we get that from the view named `LinuxDistro_view`.

Extracting on patterns (removing false positives) and then return groups allows you to match on distinct patterns, and then only return the specific elements of a large pattern that you care about.

Create view and output view are shown in Figure 13-30.

```
-----  
  
create view AllColumns_view  
as  
select  
    v1.OSBitness_col    ,  
    v2.LinuxDistro_col  ,  
    v3.LinuxVersion_col  
from  
    OSBitness_view    v1,  
    LinuxDistro_view  v2,  
    LinuxVersion_view v3;  
  
output view AllColumns_view;
```

Figure 13-30 Second version of our AQL script, part 5 of 5

Consider the following information about AQL script, part 5, in Figure 13-30:

- ▶ This `create view` statement assembles all of the distinct columns we produced.
- ▶ The `output view` statement makes this product available to any consumers of this AQL script.

Outer cartesian product: In SQL, a three-table (or view) `select` statement with no join criteria (no `where` clause to specify the relationship between the tables), produces an outer cartesian product, which is a full intersection of all of the rows from all of the tables. Generally in SQL, this is bad.

In AQL this is largely not an issue. The scope in AQL is all matching text from the one input document. Our example so far generally returns one of each target word, versus multiples.

Creating numerous (simple) views that accurately identify and output what we want, then assembling these views in a final output view is the standard design pattern when using AQL.

Because we added a second AQL module to our BigInsights project, we face an additional visual control we must adjust when running our AQL script. See the example in Figure 13-31.

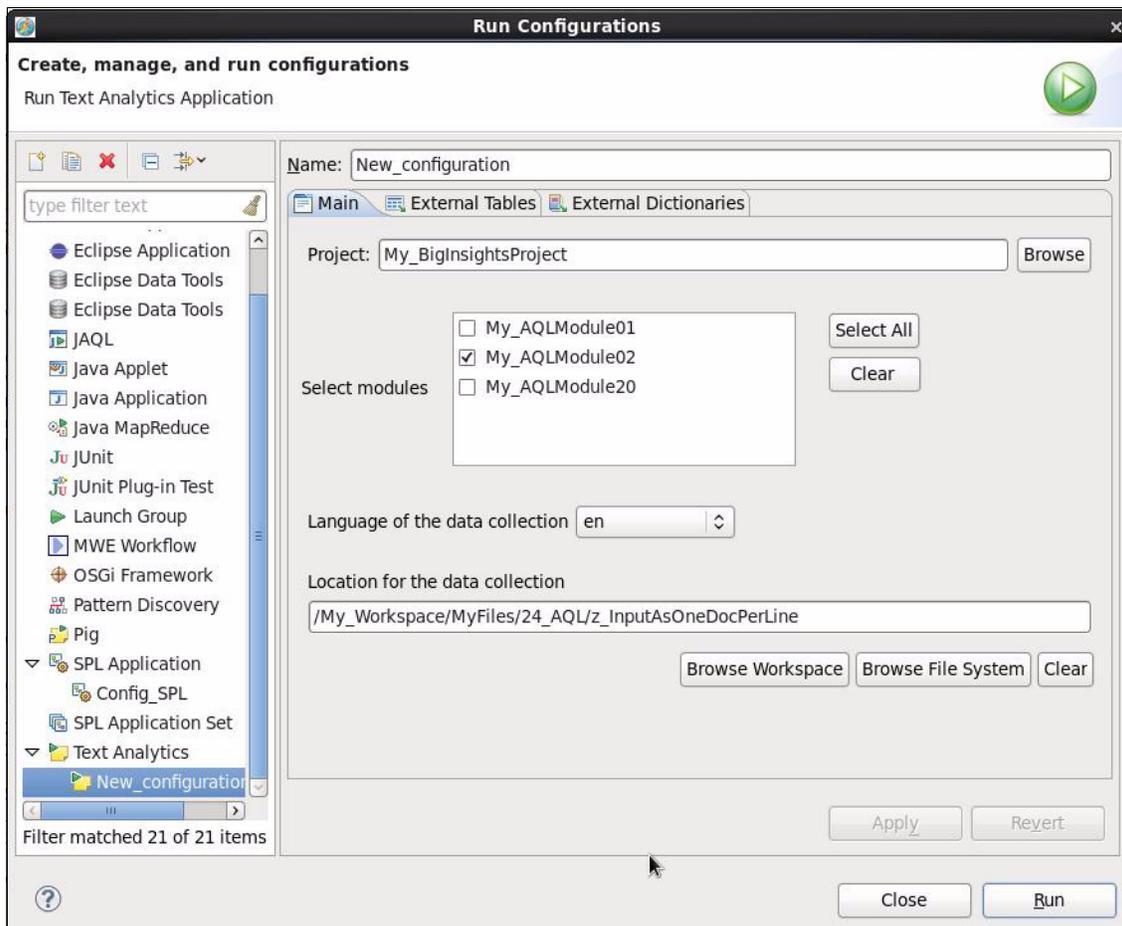
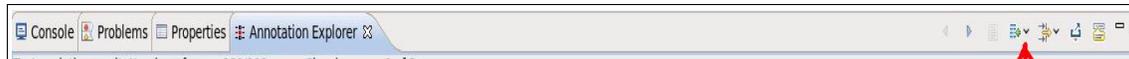


Figure 13-31 Running an AQL script when multiple AQL modules are present

Run the new AQL script by right-clicking in the AQL Editor view. Results are displayed in the Annotation Explorer view (Figure 13-32 on page 427).

Figure 13-32 has an arrow to indicate where in the tool bar to click for further examination of the AQL output view.



Text analytics result, Number of rows: 100/100 Showing page 1 of 3

Input Document	Left Context	Span Attribute Value	Right Context	Span Attribute Name
xaa.txt	x i686; en-us; rv:1.8.0.12) gecko/20070719 centos/	1.5.0.12-3.el5.centos [76-97]	firefox/1.5.0.12"	My_AQLModule02.AllColumns_
xaa.txt	u; linux i686; en-us; rv:1.8.0.12) gecko/20070719	centos [69-75]	/1.5.0.12-3.el5.centos firefox/1.5.0.12"	My_AQLModule02.AllColumns_
xab.txt	linux i686; it; rv:1.9.2.3) gecko/20100423 ubuntu/	10.04 [72-77]	(linux) firefox/3.6.3"	My_AQLModule02.AllColumns_
xab.txt	11; u; linux i686; it; rv:1.9.2.3) gecko/20100423	ubuntu [65-71]	/10.04 (linux) firefox/3.6.3"	My_AQLModule02.AllColumns_
xac.txt	i686; en-us; rv:1.8.1.11) gecko/20070914 mandriva/	2.0.0.11-1.1mdv2007.0 [78-99]	(2007.0) firefox/2.0.0.11"	My_AQLModule02.AllColumns_
xac.txt	u; linux i686; en-us; rv:1.8.1.11) gecko/20070914	mandriva [69-77]	/2.0.0.11-1.1mdv2007.0 (2007.0) firefox/2.0.0.11"	My_AQLModule02.AllColumns_
xad.txt	i686; en-us; rv:1.8.1.11) gecko/20070914 mandriva/	2.0.0.11-1.1mdv2007.0 [78-99]	(2007.0) firefox/2.0.0.11"	My_AQLModule02.AllColumns_
xad.txt	u; linux i686; en-us; rv:1.8.1.11) gecko/20070914	mandriva [69-77]	/2.0.0.11-1.1mdv2007.0 (2007.0) firefox/2.0.0.11"	My_AQLModule02.AllColumns_
xae.txt	i686; en-us; rv:1.8.1.11) gecko/20070914 mandriva/	2.0.0.11-1.1mdv2007.0 [78-99]	(2007.0) firefox/2.0.0.11"	My_AQLModule02.AllColumns_
xae.txt	u; linux i686; en-us; rv:1.8.1.11) gecko/20070914	mandriva [69-77]	/2.0.0.11-1.1mdv2007.0 (2007.0) firefox/2.0.0.11"	My_AQLModule02.AllColumns_
xaf.txt	i686; en-us; rv:1.8.1.11) gecko/20070914 mandriva/	2.0.0.11-1.1mdv2007.0 [78-99]	(2007.0) firefox/2.0.0.11"	My_AQLModule02.AllColumns_
xaf.txt	u; linux i686; en-us; rv:1.8.1.11) gecko/20070914	mandriva [69-77]	/2.0.0.11-1.1mdv2007.0 (2007.0) firefox/2.0.0.11"	My_AQLModule02.AllColumns_
xag.txt	i686; en-us; rv:1.8.1.11) gecko/20070914 mandriva/	2.0.0.11-1.1mdv2007.0 [78-99]	(2007.0) firefox/2.0.0.11"	My_AQLModule02.AllColumns_

Figure 13-32 Results in Annotation Explorer view

Open the My_AQLModule02.AllColumns_ view from the tool bar in the Annotation Explorer view. Sample output as displayed in Figure 13-33 on page 428.

AQL output view name precedes AQL module name: The reason for this is because using the AQL directives named import and export, you can share (reuse) AQL objects between AQL modules. In this manner, AQL object names are only distinct when prefaced with the AQL module name.

Figure 13-33 shows our three output columns: OSBitness_col, LinuxDistro_col, and LinuxVersion_col.

OSBitness_col (TEXT)	LinuxDistro_col (SPAN)	LinuxVersion_col (SPAN)	Input Document	Double-click this column to explain a tuple
32	fedora [67-73]	1.0.6-1.1.fc3 [74-87]	xaw.txt	Explain
32	fedora [67-73]	1.0.6-1.1.fc3 [74-87]	xax.txt	Explain
32	fedora [67-73]	1.0.6-1.1.fc3 [74-87]	xay.txt	Explain
32	fedora [67-73]	1.0.6-1.1.fc3 [74-87]	xaz.txt	Explain
32	mandriva [69-77]	2.0.0.11-1.1mdv2007.0 [78-99]	xba.txt	Explain
32	mandriva [69-77]	2.0.0.11-1.1mdv2007.0 [78-99]	xbb.txt	Explain
64	applewebkit [43-54]	533.4 [55-60]	xbc.txt	Explain
64	applewebkit [43-54]	533.4 [55-60]	xbd.txt	Explain
64	applewebkit [43-54]	533.4 [55-60]	xbe.txt	Explain
64	applewebkit [43-54]	533.4 [55-60]	xbf.txt	Explain
-1	applewebkit [36-47]	532.4 [48-53]	xbg.txt	Explain
-1	applewebkit [36-47]	532.4 [48-53]	xbh.txt	Explain

Figure 13-33 Further displays of results from menu bar in Annotation Explorer

There are other columns we want to extract (user language, for example), but none require any further AQL techniques or statements than we have detailed.

The Explain column in Figure 13-33 on page 428 is handy. See also Figure 13-34.

Notes about AQL explain:

- ▶ The Explain column details the exact manner in which a given column is qualified to be output, the result of your single or set of AQL statements.
- ▶ AQL explain is similar to SQL SET EXPLAIN, with an additional benefit that AQL explain offers a row-by-row (document-by-document) level of detail.
- ▶ AQL explain is handy for development and for debugging.
- ▶ AQL Annotation Explorer, and AQL explain are just part of the development tools we installed.

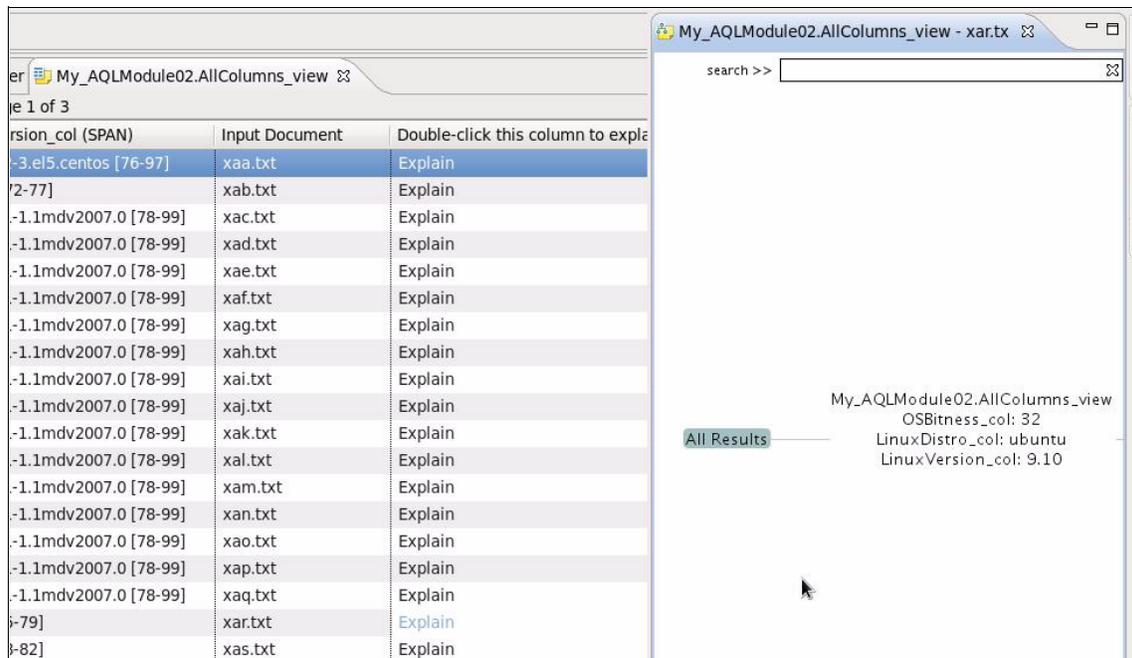


Figure 13-34 Set Explain view, how and why did a given match get produced

Because we now output three columns from our AQL script, our Streams application must also change, if only slightly. An example is in Figure 13-35.

```

AQL02_FullBrowser.spl
namespace Namespace24_AQL;

use com.ibm.streams.text.analytics::TextExtract ;

composite AQL02_FullBrowser {
graph
stream<rstring oneDocPerLine> MyFileRead = FileSource() {
param
file : "/My_Workspace/MyFiles/24_AQL/AQL01.Input.txt";
format : line;
}

stream<rstring OSBitness_col, rstring LinuxDistro_col,|
rstring LinuxVersion_col> MyTextExtract =
TextExtract(MyFileRead) {
param
inputDoc : "oneDocPerLine" ;
outputMode : "multiPort" ;
//
uncompiledModules : "/My_Workspace/MyFiles/24_AQL/MyAQL_Module02";
moduleOutputDir : "/My_Workspace/MyFiles/24_AQL/MyAQL_Module02";
}

() as MySink = FileSink(MyTextExtract) {
param
file : "/My_Workspace/MyFiles/24_AQL/AQL02.Output.txt";
format : csv;
}
}

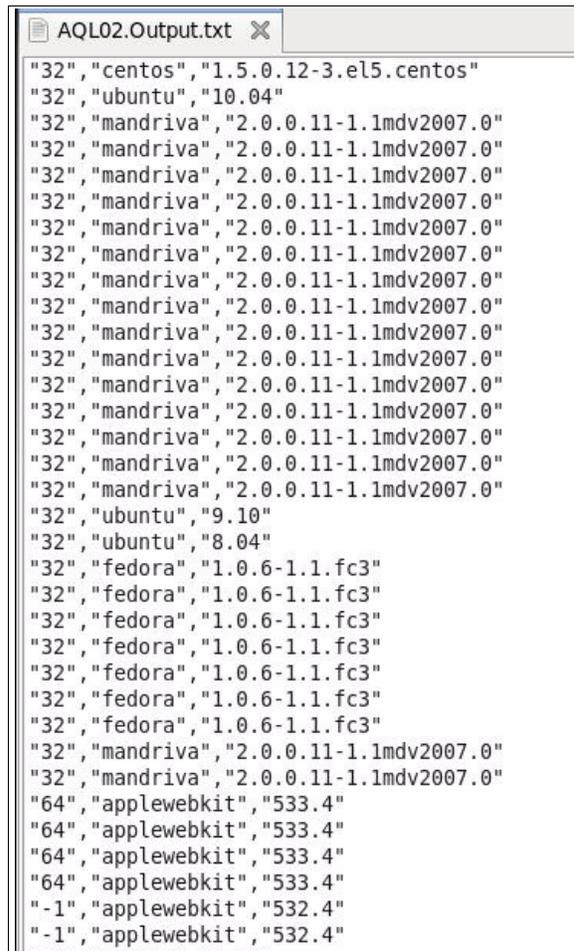
```

Figure 13-35 Streams application to support our AQL script

Consider the following information about Figure 13-35 on page 430:

- ▶ The two differences between Figure 13-35 on page 430 and Figure 13-23 on page 412 are these:
 - We now receive 3 output columns from the AQL view, as highlighted by the red arrow in the figure.
 - We changed the source and target directory for the AQL module: uncompiledModules and moduleOutputDir.
- ▶ Everything else shown in Figure 13-35 on page 430 is unchanged.

Sample output from this Streams application is displayed in Figure 13-36.



```
AQL02.Output.txt X
"32","centos","1.5.0.12-3.el5.centos"
"32","ubuntu","10.04"
"32","mandriva","2.0.0.11-1.lmdv2007.0"
"32","ubuntu","9.10"
"32","ubuntu","8.04"
"32","fedora","1.0.6-1.1.fc3"
"32","fedora","1.0.6-1.1.fc3"
"32","fedora","1.0.6-1.1.fc3"
"32","fedora","1.0.6-1.1.fc3"
"32","fedora","1.0.6-1.1.fc3"
"32","fedora","1.0.6-1.1.fc3"
"32","fedora","1.0.6-1.1.fc3"
"32","mandriva","2.0.0.11-1.lmdv2007.0"
"32","mandriva","2.0.0.11-1.lmdv2007.0"
"64","applewebkit","533.4"
"64","applewebkit","533.4"
"64","applewebkit","533.4"
"64","applewebkit","533.4"
"-1","applewebkit","532.4"
"-1","applewebkit","532.4"
"32","ubuntu","8.10"
```

Figure 13-36 Sample output from completed application, three columns

Consider the following information about Figure 13-36 on page 431:

- ▶ Near the bottom of Figure 13-36 on page 431, we see our “-1” records, referring to those bitness types that were not found.
- ▶ Regardless of what we found, 100 input documents in should create 100 records out, as produced by our union all view.

The first example is now complete. With these AQL objects and techniques, you can text-process a large percentage of the documents you encounter in the real world. The remainder of this chapter details additional AQL tools, and conventions.

13.4 Guided team-based AQL and using AQL tools

So far we created an AQL script to extract semi-structured text from an Apache HTTP log file. We used the AQL Editor view, Annotation Explorer view, the AQL Explain view, and more. More views are included with the AQL tooling and that we have not described, including the most powerful AQL-related view, the (AQL) Extraction Tasks view.

The Extraction Tasks view includes guided steps, sample data, and more, to help you complete even more capable AQL than before. The Extraction Tasks view supports a team-based approach to developing AQL; also available is versioning, notes-taking, step-by-step guidance, and AQL templates.

Although you can continue to develop AQL as we have so far, many users believe that the Extraction Tasks view is a necessary tool for completing all but the most trivial document extractions.

The IBM Big Data Channel on YouTube.com is helpful. To access the information, search for IBM Big Data Channel on YouTube.com website.

The IBM Big Data Channel has several videos related to the history of AQL, and several series of videos about AQL use, tutorials about AQL, and more. Currently the most recent AQL tutorial video series are at the following locations:

- ▶ <http://www.youtube.com/watch?v=8RwunzmPu4Q>
- ▶ <http://www.youtube.com/watch?v=BpddYCEz15o>
- ▶ http://www.youtube.com/watch?v=0-7WtwfxLJ8&list=PL7FnN5oi7Ez_KjX7zYhBc8GiK-HoNmqrJ&index=8

These videos form a three-part series about using AQL and using the Extraction Tasks view. Although the videos instructions, including steps to download the sample data used in the example, we also give the first few steps to get started, at least until the point where your specific trials vary.

As an example, complete the following steps:

1. From the Streams Studio menu bar, select **Help** → **Task Launcher for Big Data** → **Develop** [tab] → **Tasks** → **Create a text extractor**.

The New BigInsights Project dialog opens (Figure 13-37).

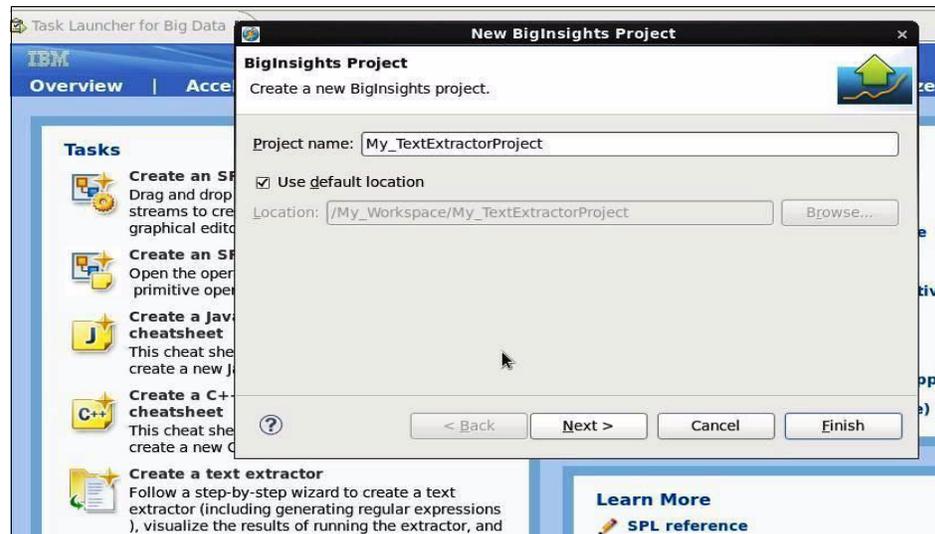


Figure 13-37 Creating a new BigInsights project, part of creating a text extractor

2. Provide a name for the project, and then click **Finish**.

The perspectives (and views) are displayed (Figure 13-38). Six steps are listed in the Extraction Tasks view panel.

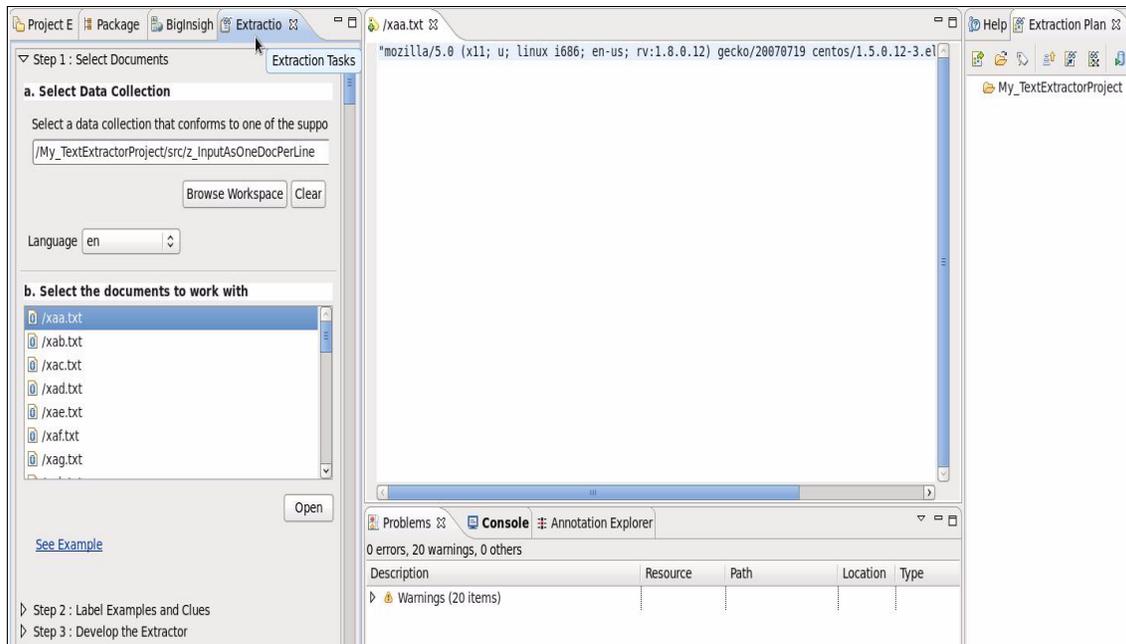


Figure 13-38 Extraction Tasks view, showing Step 1 through Step 3 of six steps

Consider the following information about Figure 13-38:

- ▶ The Extraction Tasks view is open on the left of the display. We dragged the Extraction Plan view to the right side of the display.
- ▶ The Extraction Tasks view offers a guided set of 6 steps to create advanced AQL. The Extraction Plan view offers (or will offer, as you complete steps) an object hierarchy of the AQL objects you create: the objects themselves, their properties, and more.
- ▶ Figure 13-38 shows that we already completed several tasks:
 - We set the language to English (en). Other languages are, of course, available. The language setting is in effect with dictionaries and natural language processing.
 - (This task is not displayed.) We copied our sample input documents into the Eclipse workspace. The Extraction Tasks view tooling requires that the sample documents you work with be somewhere in your Eclipse workspace. We used the Streams Project Explorer view to accomplish this task.

- We pointed to the directory containing our sample documents in Step 1a, and we selected one of these documents in Step 1b. (You can test with multiple documents. We chose to develop with just one.)

Figure 13-39 shows that Step 1 is completed (identifying our input documents), and that we can begin Step 2 and Step 3.

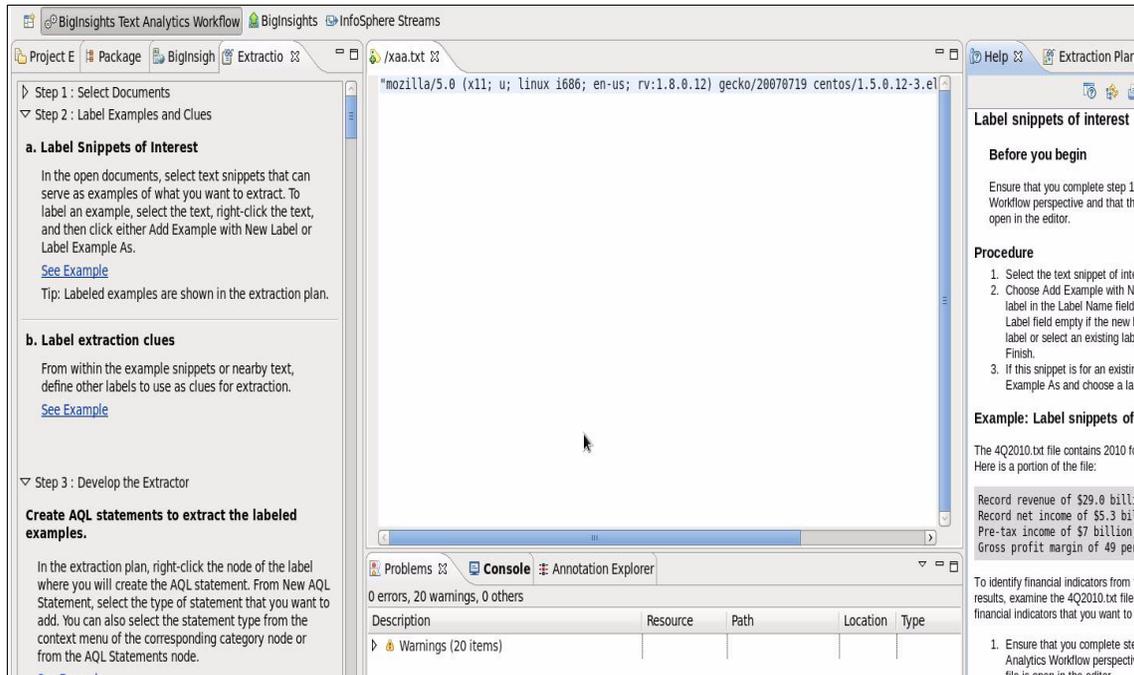


Figure 13-39 Steps 2 and 3 in the Extraction Tasks view

In Step 2, you highlight pieces of text of interest in your input document, and associate this text with the AQL objects you will create. All of these steps and notations begin to appear in the Extraction Plan view.

Step 3 prompts you to create AQL objects, in response to those items identified in Step 2.

Now, you can complete the steps in Extraction Tasks view by referring the AQL tutorial video series you located on YouTube.com, and the tutorial you loaded through Streams Studio Help.

13.5 Regular expressions (regex)

Although this document mentions regular expression (regex), we have not yet described what these regular expressions mean, or how we derive them. For a history and overview of regex, we suggest you read the Wikipedia.com entry about regex. While in Wikipedia, we also suggest you read the entry for metacharacter, which is also used in this chapter. Go to the following address:

<http://en.wikipedia.org/wiki/Metacharacter>

In this chapter, we briefly describe regular expression metacharacters (syntax), the views inside the text analytics tools used to develop and support regex, and specific regex functions available within the Streams processing language.

Figure 13-40 on page 437 lists some of the most commonly used regex expressions.

Regex Special characters	
<code>\d</code>	Any digit
<code>\D</code>	Any non-digit
<code>\w</code>	Any word character (number, letter, underscore)
<code>\W</code>	Any non-word character
<code>\b</code>	Any word boundary character
<code>\s</code>	Any whitespace character
<code>\S</code>	Any non-whitespace character
<code>.</code>	Any single character

Regex Position	
<code>^</code>	Start of line
<code>\$</code>	End of line
<code>\A</code>	Start of string
<code>\z</code>	End of string

Regex Cardinality	
<code>a?</code>	Zero or one of a
<code>a*</code>	Zero or more of a
<code>a+</code>	One or more of a
<code>a{4}</code>	Exactly 4 or a
<code>a{5,}</code>	Five or more of a
<code>a{2,7}</code>	Between 2 and 7 of a

Regex Sets	
<code>(ab)</code>	a or b
<code>[abc]</code>	Any single character, a, b or c
<code>[^a,b,c]</code>	Any single character except a, b or c
<code>[a-z]</code>	Any single character in the range, a thru z
<code>[a-zA-Z]</code>	Above, but a-z or A-Z

Figure 13-40 Guide to regex metacharacters that cover most cases

Consider the following information about Figure 13-40 (regex metacharacter expressions):

- ▶ The first regex expression we use in this chapter is shown in Figure 13-28 on page 421. The expression is `/\w+/` on a word similar to `Fedora/` or `Ubuntu/`.
 - Figure 13-40 shows that `\w` is a regex metacharacter to represent any single number, letter, or underscore.
 - The plus sign (+) indicates one or more of something, in this case, a `\w` metacharacter. Thus, from the word `Fedora/` we extract `Fedora`, and from the word `Ubuntu/` we extract `Ubuntu`.

- Why do we focus on the leading part of the word, and not the trailing forward slash (/) character? No reason; both approaches, and probably other approaches, are all valid.
- ▶ The second regex expression we use in this chapter is displayed in Figure 13-29 on page 423. The expression is `/[0-9]+\S*/` on a word similar to `2.24.UC4` or `3.13.17`, basically a software version number.
 - The `[0-9]` leads that word and represents any digit, any number.
 - The plus sign `(+)` indicates one or more.
 - The `\S` is any non-whitespace character; so hyphens, underscores, upper and lowercase letters, and periods, are all valid.

13.5.1 AQL tools to aid in regex development and debug

Two additional views are in the Text Analytics toolkit to aid in development and debugging using regular expressions. These are available from the Streams Studio toolbar, by default, when the text analytics tools are installed. An example is in Figure 13-41.

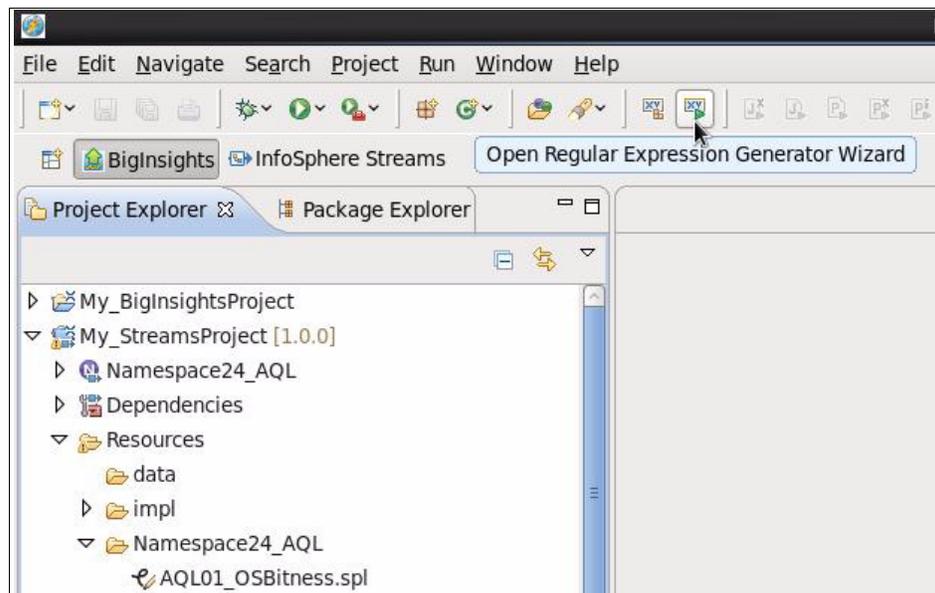


Figure 13-41 Streams Studio tool bar, Regular Expression Generator Wizard

Figure 13-41 shows the icon for the Text Analytics toolkit Regular Expression Generator Wizard. The Regular Expression Builder Wizard is to its left.

If these toolbar buttons are not visible, you can show them by clicking **Window** → **Customize Perspective** from the Streams Studio menu bar, as shown in Figure 13-42.

As wizards, these are not views that you can keep open, or add to any perspective or menu bar.

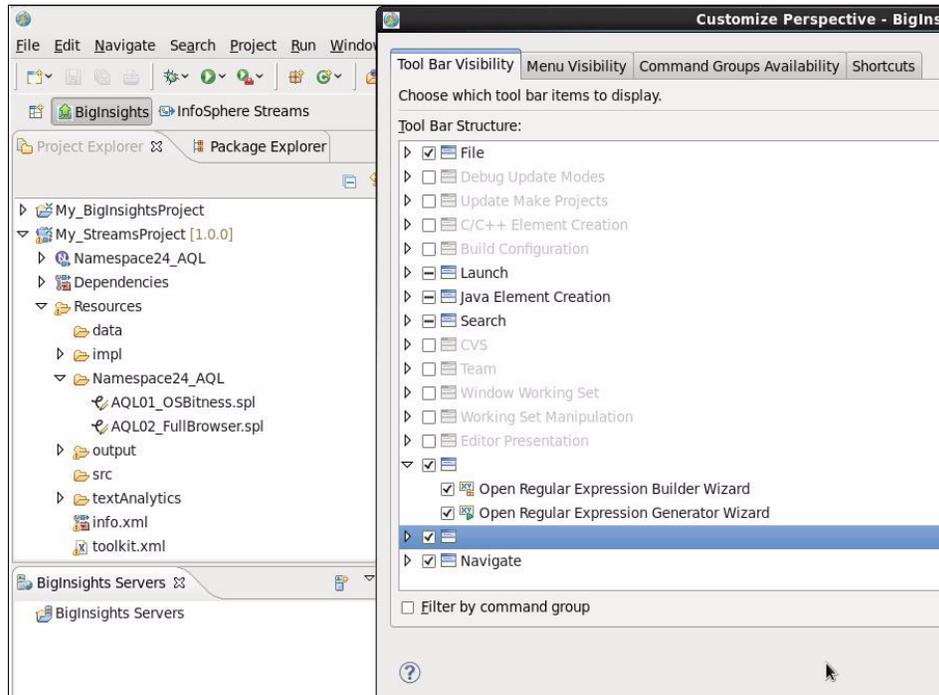


Figure 13-42 Streams Studio menu bar

If you are developing a regular expression, you only need to use one of these two wizards; both accomplish the same task, but through different methods.

The Regular Expression Builder Wizard is displayed in Figure 13-43 on page 440.

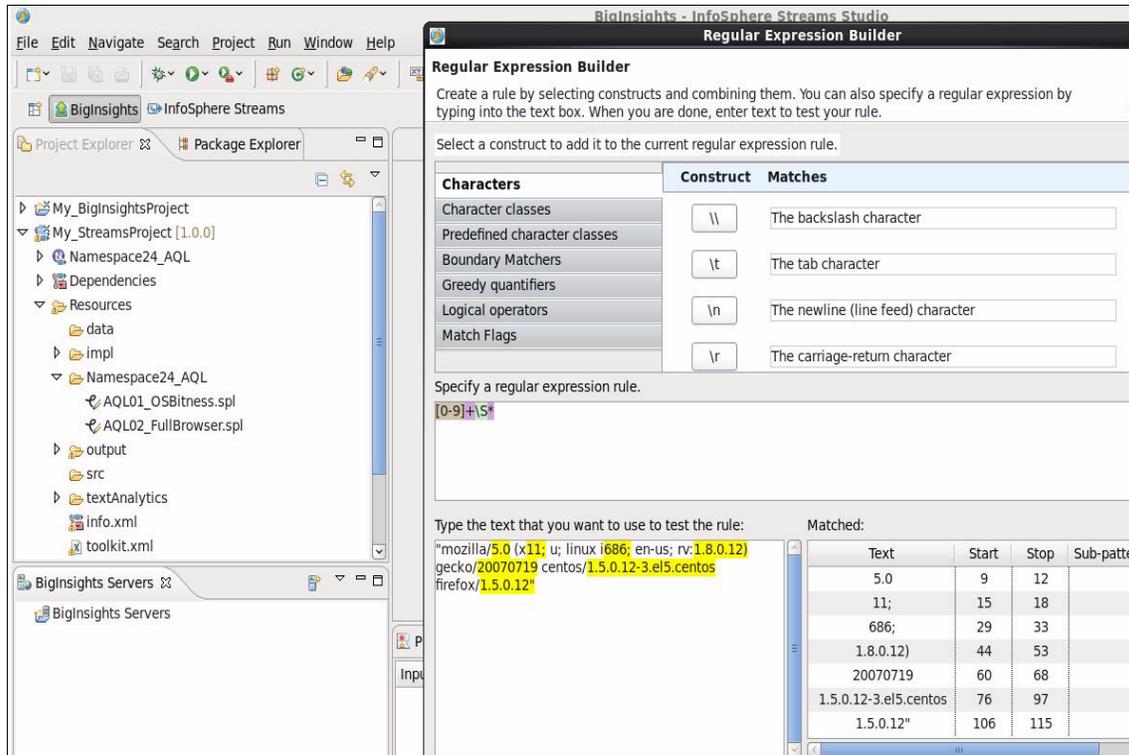


Figure 13-43 Regular Expression Builder Wizard

Consider the following information about Figure 13-43:

- ▶ In the area labeled “Type the text that you want to use to test the rule,” we pasted text from one of our sample input files.
- ▶ Then, from the list of regex metacharacters (Figure 13-40 on page 437), we iteratively tried various new metacharacter sets. We typed our trial metacharacters into the area labeled “Specify a regular expression rule.”
- ▶ The results are listed in the area labeled “Matched.”
- ▶ In Figure 13-43, you see the exact (final) regular expression we use in Figure 13-29 on page 423.

Figure 13-44 on page 441 displays part 1 of 2 of the Regular Expression Generator Wizard.

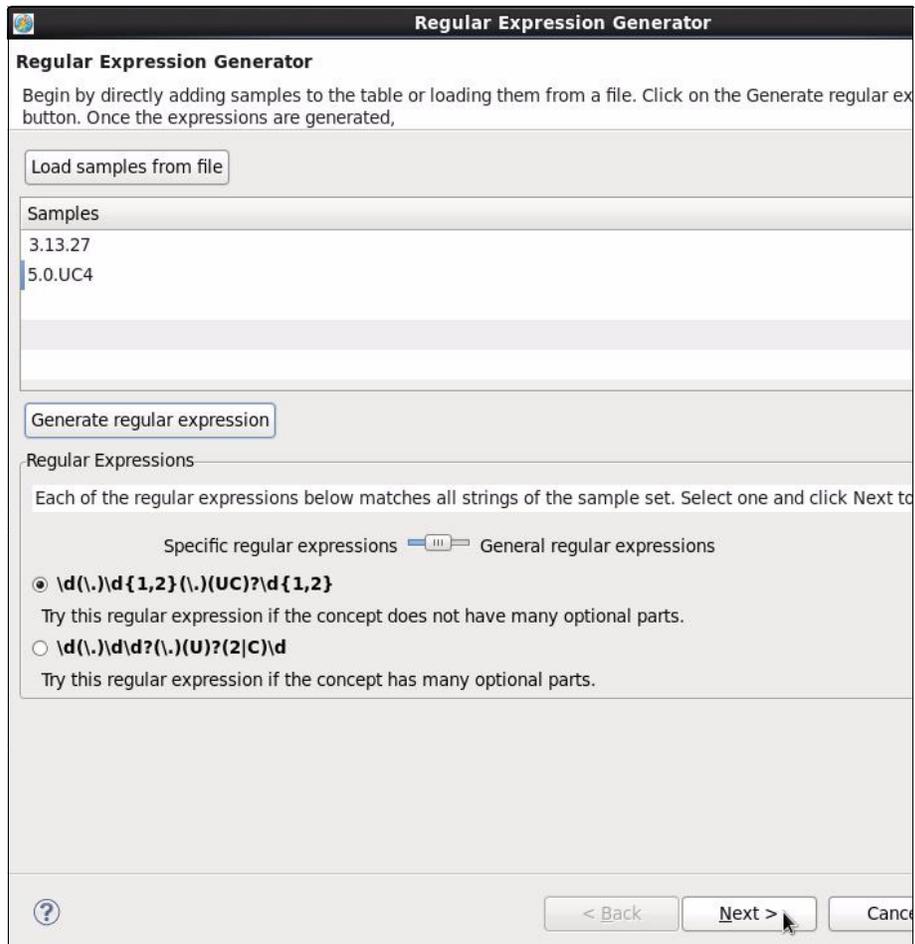


Figure 13-44 Regular Expression Generator Wizard, part 1 of 2

Consider the following information about Figure 13-44:

- ▶ As displayed you can load sample data from a file, or paste it directly.
 - Unless you want to generate a regular expression for your entire input data document (possible, although probably not useful), you copy and paste only the text that you want to give focus to.
- ▶ The slide bar instructs this wizard how precise to be, and how much to abstract on digits, versus characters, and so on.
- ▶ Click **Next** when you have a workable regular expression. This action produces the display in Figure 13-45 on page 442.

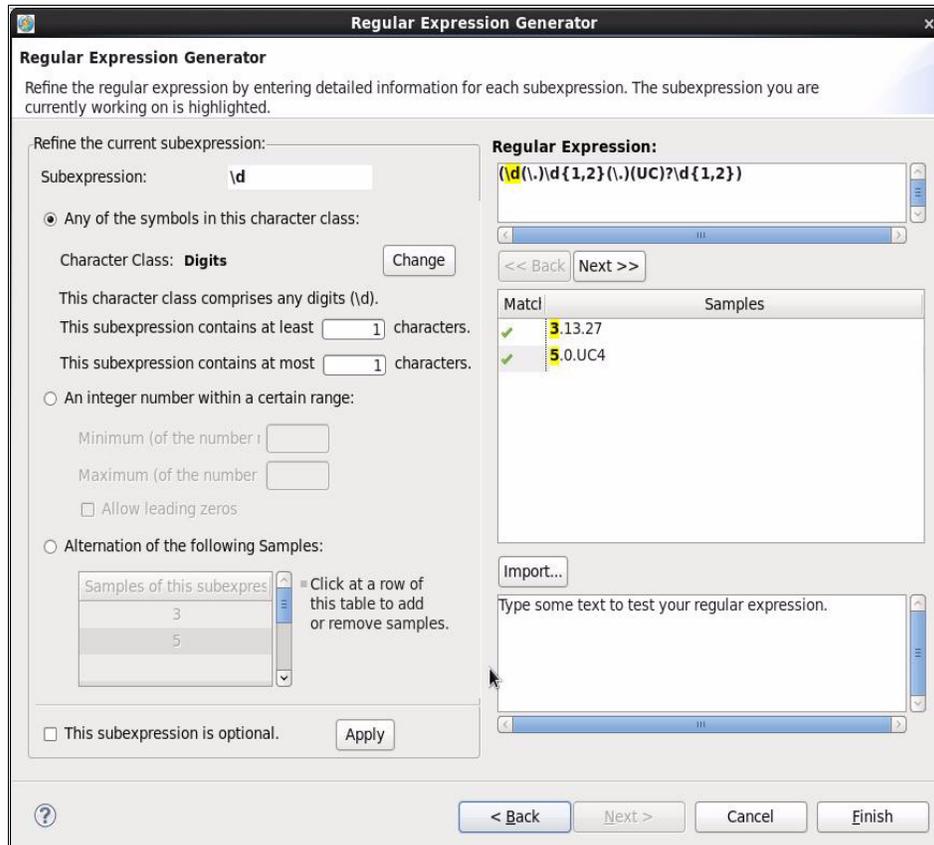


Figure 13-45 Regular Expression Generator Wizard, part 2 of 2

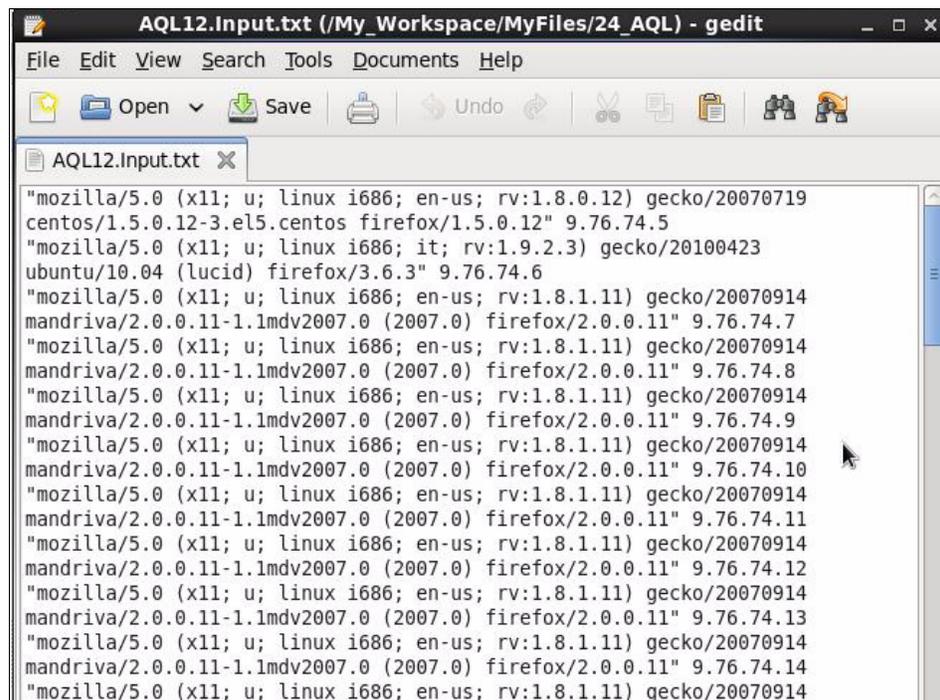
Consider the following information about Figure 13-45:

- ▶ You can manually adjust the regular expression, add ranges, and more.
- ▶ When you click **Finish**, the regular expression you created is copied to the clip board.

13.5.2 Regex directly inside Streams; no AQL required

The primary purpose of this section is to make you aware that you can use of regular expressions in a Streams application without writing AQL, without using the Streams TextExtract operator. The Streams standard toolkit includes several regular expression functions, including but not limited to `regexReplace()` and `regexMatch()`.

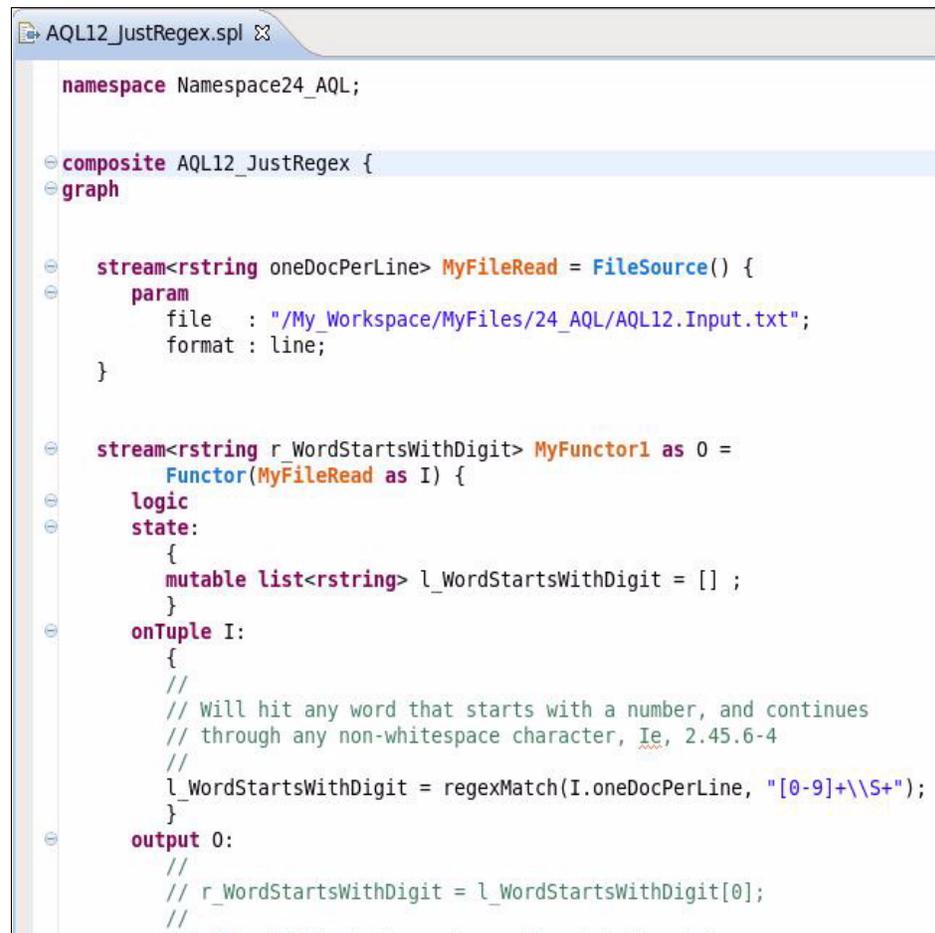
The sample data we process is displayed in Figure 13-46, and is nearly the same data we use throughout this chapter: Apache HTTP server log file data.

A screenshot of a gedit text editor window titled "AQL12.Input.txt (/My_Workspace/MyFiles/24_AQL) - gedit". The window contains a list of Apache HTTP server log entries. Each entry is a single line of text representing a log record. The entries are as follows:

```
"mozilla/5.0 (x11; u; linux i686; en-us; rv:1.8.0.12) gecko/20070719
centos/1.5.0.12-3.el5.centos firefox/1.5.0.12" 9.76.74.5
"mozilla/5.0 (x11; u; linux i686; it; rv:1.9.2.3) gecko/20100423
ubuntu/10.04 (lucid) firefox/3.6.3" 9.76.74.6
"mozilla/5.0 (x11; u; linux i686; en-us; rv:1.8.1.11) gecko/20070914
mandriva/2.0.0.11-1.1mdv2007.0 (2007.0) firefox/2.0.0.11" 9.76.74.7
"mozilla/5.0 (x11; u; linux i686; en-us; rv:1.8.1.11) gecko/20070914
mandriva/2.0.0.11-1.1mdv2007.0 (2007.0) firefox/2.0.0.11" 9.76.74.8
"mozilla/5.0 (x11; u; linux i686; en-us; rv:1.8.1.11) gecko/20070914
mandriva/2.0.0.11-1.1mdv2007.0 (2007.0) firefox/2.0.0.11" 9.76.74.9
"mozilla/5.0 (x11; u; linux i686; en-us; rv:1.8.1.11) gecko/20070914
mandriva/2.0.0.11-1.1mdv2007.0 (2007.0) firefox/2.0.0.11" 9.76.74.10
"mozilla/5.0 (x11; u; linux i686; en-us; rv:1.8.1.11) gecko/20070914
mandriva/2.0.0.11-1.1mdv2007.0 (2007.0) firefox/2.0.0.11" 9.76.74.11
"mozilla/5.0 (x11; u; linux i686; en-us; rv:1.8.1.11) gecko/20070914
mandriva/2.0.0.11-1.1mdv2007.0 (2007.0) firefox/2.0.0.11" 9.76.74.12
"mozilla/5.0 (x11; u; linux i686; en-us; rv:1.8.1.11) gecko/20070914
mandriva/2.0.0.11-1.1mdv2007.0 (2007.0) firefox/2.0.0.11" 9.76.74.13
"mozilla/5.0 (x11; u; linux i686; en-us; rv:1.8.1.11) gecko/20070914
mandriva/2.0.0.11-1.1mdv2007.0 (2007.0) firefox/2.0.0.11" 9.76.74.14
"mozilla/5.0 (x11; u; linux i686; en-us; rv:1.8.1.11) gecko/20070914
```

Figure 13-46 Our sample input file to process

The Streams application we use to demonstrate regular expressions, more specifically the `regexMatch()` function inside the Streams standard toolkit, is displayed in Figure 13-47 through Figure 13-49 on page 448.



```
namespace Namespace24_AQL;

composite AQL12_JustRegex {
graph

stream<rstring oneDocPerLine> MyFileRead = FileSource() {
param
file : "/My_Workspace/MyFiles/24_AQL/AQL12.Input.txt";
format : line;
}

stream<rstring r_WordStartsWithDigit> MyFunc1 as 0 =
Func1(MyFileRead as I) {
logic
state:
{
mutable list<rstring> l_WordStartsWithDigit = [];
}
onTuple I:
{
//
// Will hit any word that starts with a number, and continues
// through any non-whitespace character, I.e., 2.45.6-4
//
l_WordStartsWithDigit = regexMatch(I.oneDocPerLine, "[0-9]+\\S+");
}
}
output 0:
//
// r_WordStartsWithDigit = l_WordStartsWithDigit[0];
//
```

Figure 13-47 Our Streams application, part 1 of 3

Consider the following information about Figure 13-47:

- ▶ Unlike the Streams application in Figure 13-35 on page 430, this Streams application has no use statement. We do not need to import the text analytics toolkit, because `regexMatch()` is part of the Streams standard toolkit.
- ▶ `MyFileRead` is a standard `FileSource` operator, and serves as the only input device to this Streams application.

- ▶ Regarding MyFunctor1, note the following information:
 - A list of rstrings is defined. The regexMatch() function can return zero, one, or more values, which is the list is necessary.

Lists and types: You should be aware at this point that Streams has three variable types similar to what other environments might call arrays. Streams calls these data types *collections*, and list is just one type of collection.

Lists are homogenous, containing a list of one data type, for example, a list of just rstrings, or a list of just integer 32-bit.

You can create a list of types. Types are similar to what other environments called *records* (or what Java calls a type or class).

Further, regexMatch() can return data types other than rstring, for example, int32, but we prefer to receive rstrings lest we possibly create an casting error.

- We place the regexMatch() function invocation in the onTuple clause.
- Notice that this regex differs from all of our previous regex examples. All of the backslash characters (\) are “escaped;” instead of having \S, as in Figure 13-43 on page 440, here we offer a \\S characters.

Double backslash: We blame the double backslash (\\), versus single backslash (\), issue on C++ versus Java.

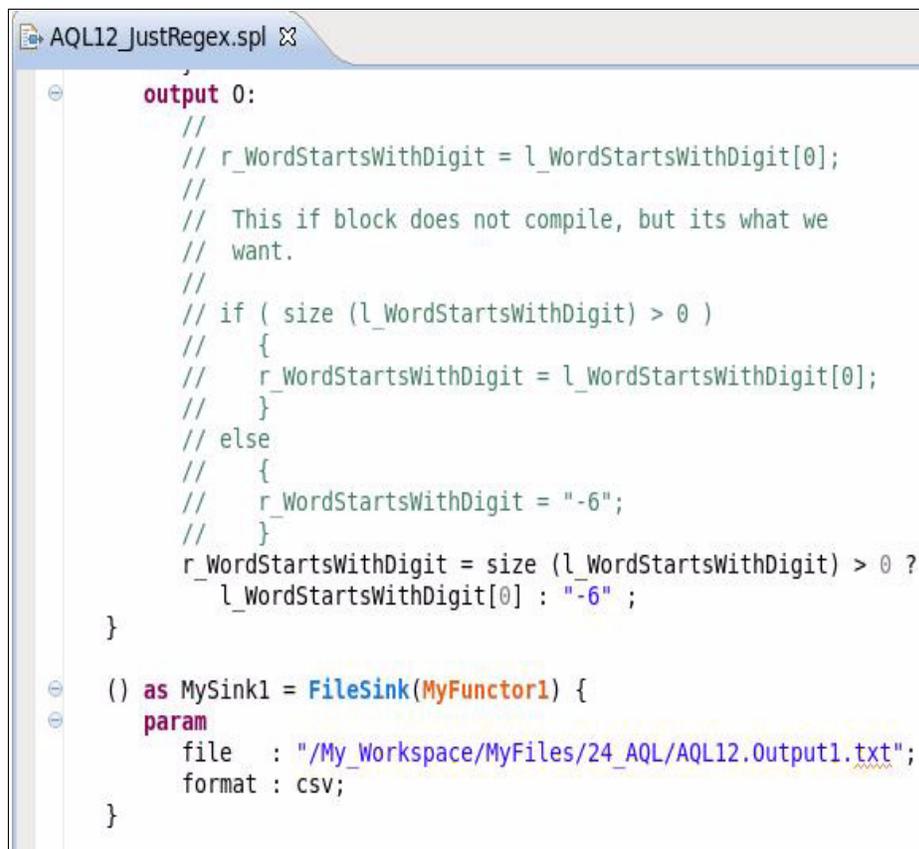
Generally, the core of Streams operates in a C++ environment, whereas the text analytics tools and run time operate in Java.

Figure 13-48 continues our Streams application source code listing.

Note: Although we have used this regex before in this chapter, this is our regex to return all words that start with a number. Each line in our sample data has many words that start with a number.

But, *as our regex is currently constructed*, our `regexMatch()` will only ever return the first match. The reason is because that is how `regexMatch()` works. Our regex calls for one match, not multiple matches, not a global match.

If you want the second, third, or further number, you must write a different (better) regular expression. Our next example details this use case.



```
AQL12_JustRegex.spl
output 0:
    //
    // r_WordStartsWithDigit = l_WordStartsWithDigit[0];
    //
    // This if block does not compile, but its what we
    // want.
    //
    // if ( size (l_WordStartsWithDigit) > 0 )
    // {
    //     r_WordStartsWithDigit = l_WordStartsWithDigit[0];
    // }
    // else
    // {
    //     r_WordStartsWithDigit = "-6";
    // }
    r_WordStartsWithDigit = size (l_WordStartsWithDigit) > 0 ?
        l_WordStartsWithDigit[0] : "-6" ;
}

() as MySink1 = FileSink(MyFunc1) {
    param
        file : "/My_Workspace/MyFiles/24_AQL/AQL12.Output1.txt";
        format : csv;
}
```

Figure 13-48 Our Streams application, part 2 of 3

Consider the following information about Figure 13-48 on page 446:

- ▶ The output clause to this functor offers syntax that may not be familiar.
 - The `regexMatch()` function can return zero, one, or more values. Thus, our list containing the results of this function might be empty, and contain no values.
 - The `size()` function measures the size of our list variable. If the size is zero, we choose to return a constant value equal to `-6`. The reason for this value is that all of our error conditions in this chapter are distinct negative values, which is purely a style choice.
 - If `size()` is greater than zero (if our list has any values), we call to return the first value in our list. Because lists begin with an index of zero, `list[0]` refers to the first element in a list.
 - All code that is commented out displays the logic, the intent, of our return statement. You cannot place an `if` statement in an output clause, so we choose to use the statement we have in place.
- ▶ We have the *first* of two `FileSink` operators.

Figure 13-49 on page 448 displays the remainder of our sample Streams application.

```

AQL12_JustRegex.spl
stream<rstring r_WordStartsWithDigit> MyFunc2 as 0 =
  Functor(MyFileRead as I) {
  logic
  state:
  {
    mutable list<rstring> l_WordStartsWithDigit = [] ;
  }
  onTuple I:
  {
    //
    // Will hit an IP address, I.e, 123.456.789.012
    //
    l_WordStartsWithDigit = regexMatch(I.oneDocPerLine,
      "[0-9]{1,3})\\.([0-9]{1,3})\\.([0-9]{1,3})\\.([0-9]{1,3})$");
  }
  output 0:
  //
  // l_WordStartsWithDigit[0] is whole string
  //
  // [1] is first sub group, ...
  //
  r_WordStartsWithDigit = size (l_WordStartsWithDigit) > 1 ?
    l_WordStartsWithDigit[1] + "|" +
    l_WordStartsWithDigit[2] : "-8" ;
  }

  () as MySink2 = FileSink(MyFunc2) {
  param
  file : "/My_Workspace/MyFiles/24_AQL/AQL12.Output2.txt";
  format : csv;
  }
}

```

Figure 13-49 Our Streams application, part 3 of 3

Consider the following information about Figure 13-49; the first new part to this display is the new and much larger regular expression:

- ▶ This regular expression looks for IP addresses type data, for example 192.168.1.12.
- ▶ The regular expression reads as, [0-9]{1,3} period [0-9]{1,3} and so on. [0-9] means number, and {1,3} means exactly one to three occurrences of this number.

This pattern repeats. Put these together, with periods, and you have an IP address.

- ▶ The dollar sign means end-of-line, which is where the IP address we want is located.
- ▶ The output clause is similar to our first example; checking to see if zero values were return from `regexMatch()`, and in that case returning a negative number status code (here we use a "-8").
- ▶ If we do have data to return, we concatenate it with a field delimiter equal to the vertical bar, ASCII character 127.

We return the second and third octets, which are represented by `list[1]` and `list[2]`. (Recall, lists index starting from zero.)

Parentheses are important: Recall that we said the `regexMatch()` function can return zero, one or more values. This is why we need to return into a list.

Zero is obvious: you had no matches.

Without a regular expression containing parenthesis, `regexMatch()` can only return the entire match. With parenthesis, `regexMatch()` can return the entire match, and subsets of the match.

With parenthesis you can return a phone number, with the area code and exchange as separate values, which is handy.

Here we use the parenthesis to give us the separate elements of the IP address (these are called octets, by the way). With these separate elements of the IP address, we might then perform math to determine if this is a class-A network, class-B, and so on. The point is, `regexMatch()` did this separation for us.

Without parenthesis in the regular expression, `regexMatch()` can only return zero or one values. With parenthesis in the regular expression, `regexMatch()` can return many values.

Regardless, you must still receive the output of `regexMatch()` in a list.

Figure 13-50 displays the output from our first functor.

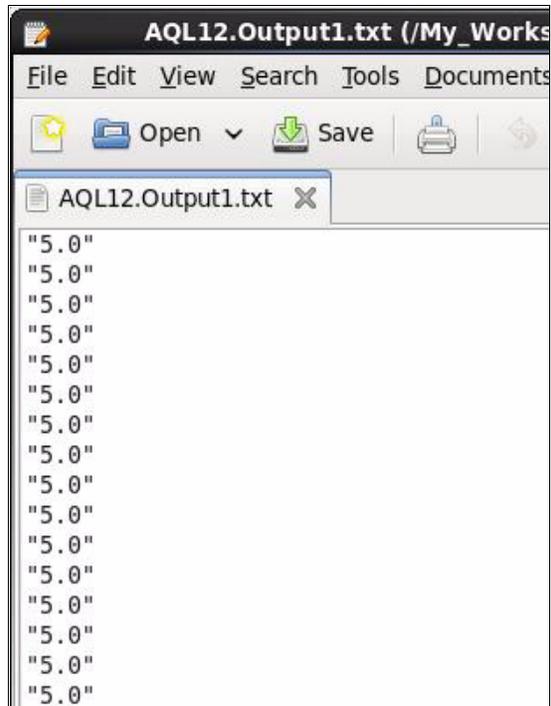


Figure 13-50 Results from the first FileSink

Note: Because of how the first regular expression is written, you will only ever receive the first word to start with a number in this output.

If you need the second or later word, you need to supply a different regular expression.

Figure 13-51 displays the output of our second functor. We have output the second and third octets from our IP address.

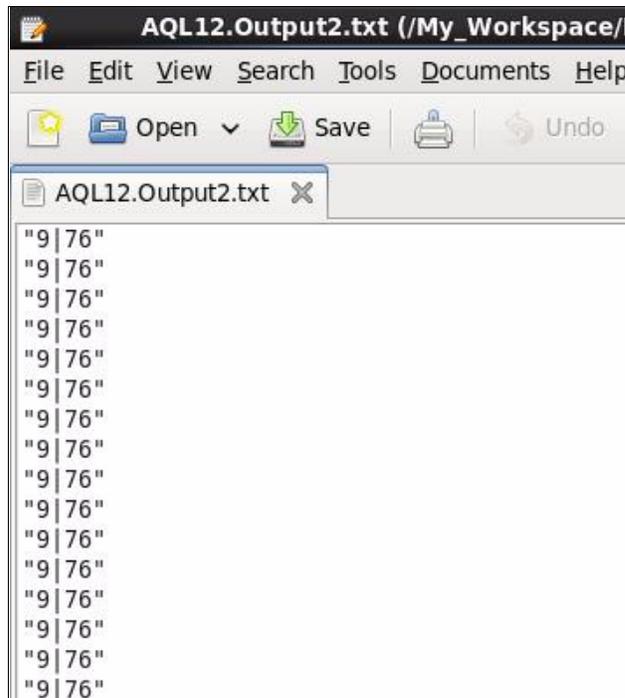


Figure 13-51 Results from the second FileSink

13.6 Additional AQL objects and techniques

In this section, we add even more AQL objects and capabilities to those previously discussed. The first AQL items we add are external dictionaries, AQL tables, AQL external tables, and a true where clause that joins a view and a table.

For this example, we continue to use our standard set of sample input files: Apache HTTP server log files.

Figure 13-52 on page 452 through Figure 13-54 on page 456 display our new AQL script.

```
My_AQLScript.aql  AQL20_ExternalDictionaryTable.spl
module My_AQLModule20;

-----

create external dictionary OSBitness_dict
  allow_empty false
  with case insensitive;

-- Regarding the above ..
-- . 'insensitive is the default,
-- . 'exact' is the other choice
--
-- . 'allow_empty false' means dict
--   must be readable, other choice
--   is 'true'
--
-- . This dict pathname must be specified
--   in the SPL, or the text analytics
--   runner

-- export dictionary OSBitness_dict;

-----

create view OSBitnessInDict_view
  as
  extract dictionary 'OSBitness_dict'
  on D.text as OSBitness_col
  from Document D;
```

Figure 13-52 AQL script source code listing, part 1 of 3

Consider the following information about the AQL script in Figure 13-52:

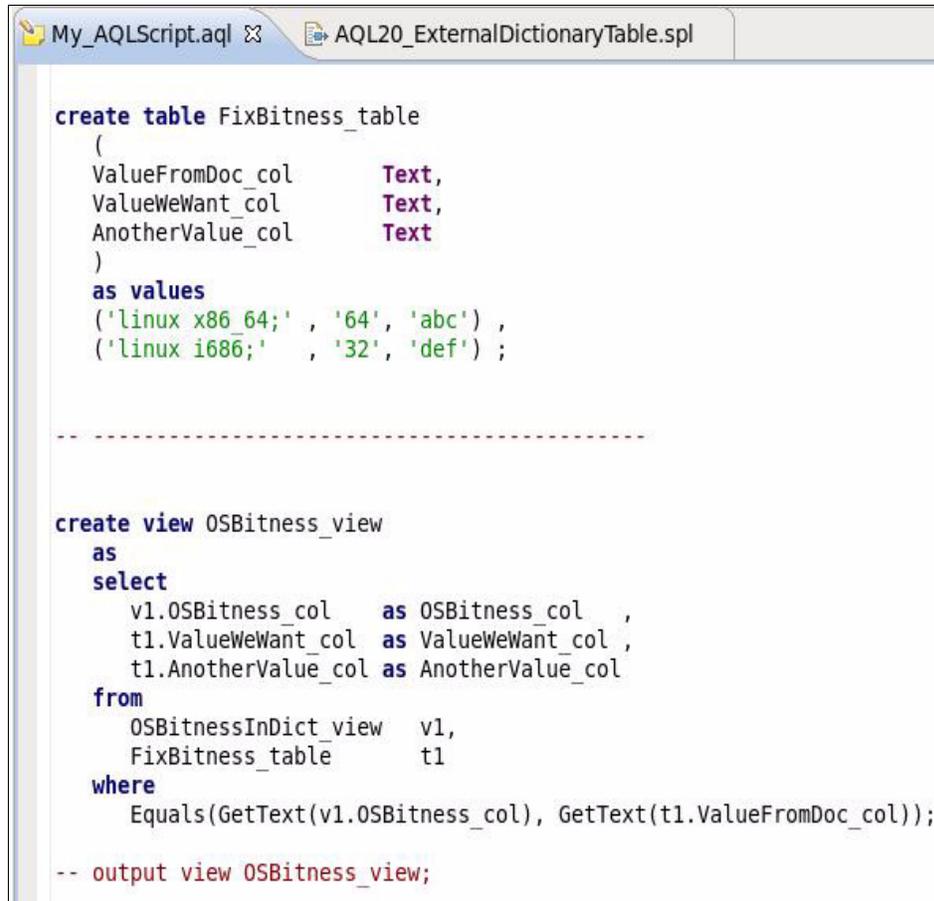
- ▶ This is the first time we have displayed the `create external dictionary` statement. As commented, the `allow_empty` switch determines whether the dictionary must be readable by any consumers. By default, an external dictionary does not require an exact match on case.
- ▶ There is no mention of where the dictionary resides, no path name to a file, or other. This value is supplied externally to the AQL script.

Note: Actually, there are three ways to create dictionaries: in-line, in a file, and external. Our first dictionary (Figure 13-22) is in-line; it is easy to use and easy to test.

This dictionary is external, and is preferred over in-line, because you do not have to open the AQL to make changes.

- ▶ If this dictionary were to be referenced by other AQL modules, then we would need the `export dictionary` statement. Because this is not our case, we comment out the `export dictionary` statement.
- ▶ The `create view` statement offers no new techniques to those previously discussed.

Figure 13-53 on page 454 displays the next portion of our AQL script.



```
My_AQLScript.aql  AQL20_ExternalDictionaryTable.spl

create table FixBitness_table
(
  ValueFromDoc_col      Text,
  ValueWeWant_col       Text,
  AnotherValue_col      Text
)
as values
('linux x86_64;' , '64', 'abc') ,
('linux i686;'   , '32', 'def') ;

-----

create view OSBitness_view
as
select
  v1.OSBitness_col      as OSBitness_col ,
  t1.ValueWeWant_col    as ValueWeWant_col ,
  t1.AnotherValue_col   as AnotherValue_col
from
  OSBitnessInDict_view v1,
  FixBitness_table     t1
where
  Equals(GetText(v1.OSBitness_col), GetText(t1.ValueFromDoc_col));

-- output view OSBitness_view;
```

Figure 13-53 AQL script source code listing, part 2 of 3

Consider the following information about the AQL script in Figure 13-53:

- ▶ This is the first time we show an AQL `create table` statement. AQL tables may be embedded entirely within an AQL script, or its data may be external, similar to an external dictionary. (An AQL external table is offered in Figure 13-54 on page 456)
- ▶ Generally, the use for an AQL table is *enrichment*, meaning you have an identifying or key column, and want to add more descriptive columns to your output.

Note: We did an alternate form of enrichment in Figure 13-26. There we used the case statement to return an alternate value.

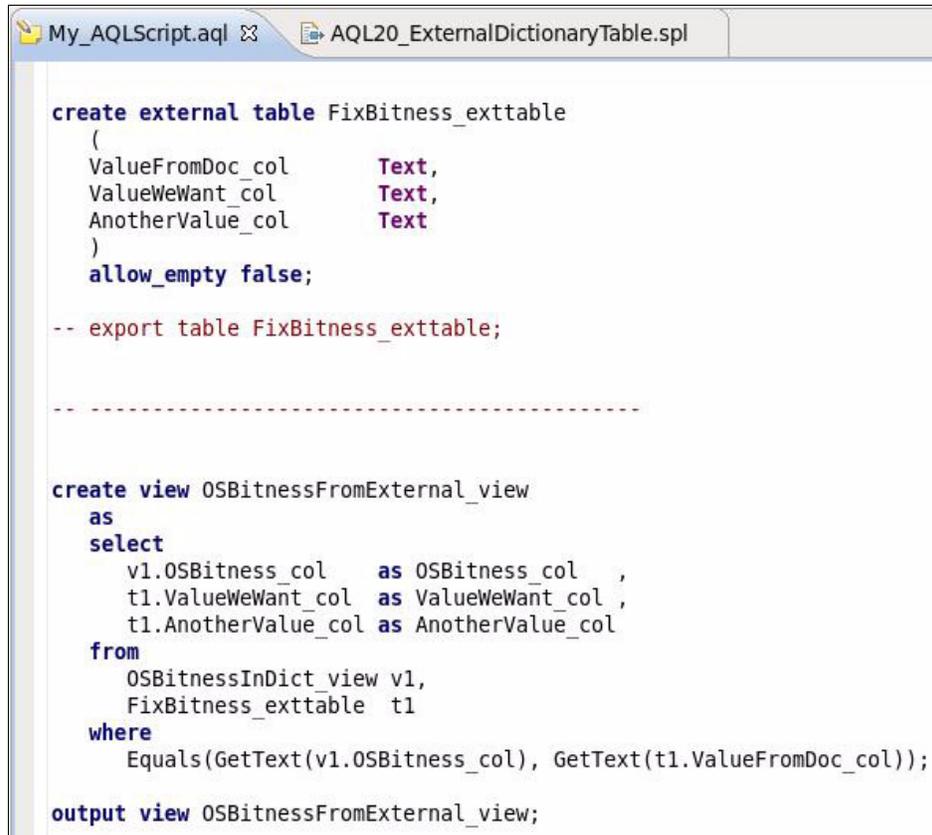
We could have joined to an AQL table to get the value 64, from a dictionary entry for 64-bit. This might even be advantageous; while the dictionary and table are in sync, then you do not need the `else` clause in the case statement, providing the minus two (-2) value for unknowns.

Consider this example: “Wisconsin” and “The Badger State.” Through your text analytics, you extracted Wisconsin, but had no other means to produce Wisconsin’s nickname, as required by your application.

- ▶ The create view, named `OSBitness_view`, contains a join between a view and a table; you can join views to views, tables to tables, and views to tables.

Note that the join criteria is entered in a manner that differs slightly from SQL. AQL uses the `Equals()` predicate function. In our specific case we are joining matching text, of type `SPAN`, thus, we must again use the `GetText()` function to return the text value of each match.

Figure 13-54 on page 456 displays the final portion of this AQL script.



```
My_AQLScript.aql  AQL20_ExternalDictionaryTable.spl

create external table FixBitness_exttable
(
  ValueFromDoc_col      Text,
  ValueWeWant_col       Text,
  AnotherValue_col      Text
)
allow_empty false;

-- export table FixBitness_exttable;

-----

create view OSBitnessFromExternal_view
as
select
  v1.OSBitness_col      as OSBitness_col ,
  t1.ValueWeWant_col    as ValueWeWant_col ,
  t1.AnotherValue_col   as AnotherValue_col
from
  OSBitnessInDict_view v1,
  FixBitness_exttable t1
where
  Equals(GetText(v1.OSBitness_col), GetText(t1.ValueFromDoc_col));

output view OSBitnessFromExternal_view;
```

Figure 13-54 AQL script source code listing, part 3 of 3

Consider the following information about Figure 13-54:

- ▶ This is the first time you see an external table. Similar to an external dictionary, notice that the location of the actual file containing this table's data is not specified inside the AQL script. This value is supplied externally to the AQL script.
- ▶ Also similar to an external dictionary, the export statement to this external table is required only if this external table is referenced by another AQL module.
- ▶ The final view offers no new techniques; it serves to compile all of our output.

Note: Actually this is not true. We built this example iteratively.

With a sharp eye you will notice we never use the output from `FixBitness_table`, having replaced it with `FixBitness_exttable`. And we never use the output from the `OSBitness_view`, replacing it with the view named, `OSBitnessFromExternal_view`.

We have two external dependencies: the location of our external dictionary and the location of our external table. Figure 13-55 displays the file listing for both the dictionary and the table.

For style, we have always named our dictionaries `<Something>.dict`. External tables must be in comma-separated value (CSV) ASCII text-file format, and must have a `.csv` file name extension.

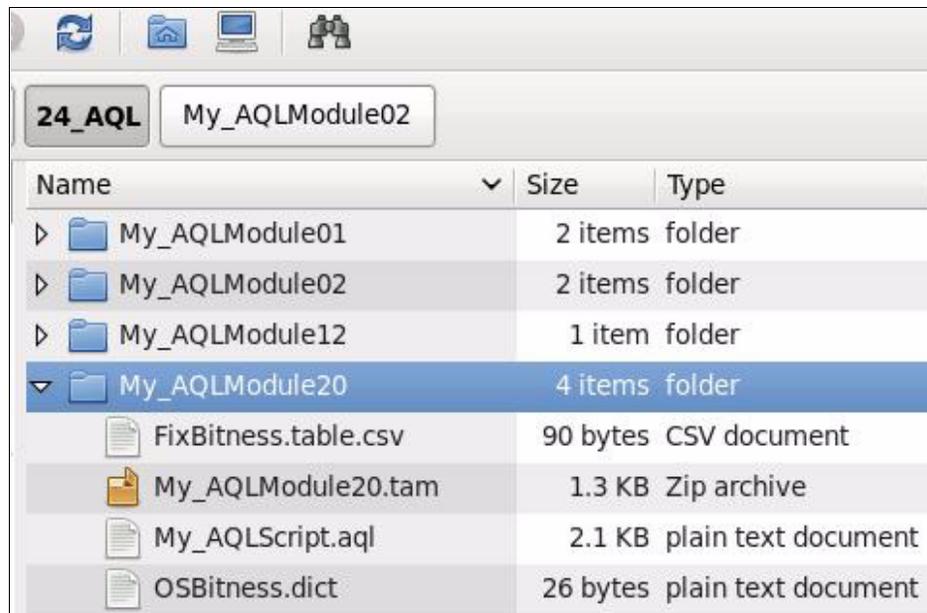


Figure 13-55 File listing, external dictionary and table

Figure 13-56 on page 458 displays the contents of our external dictionary.

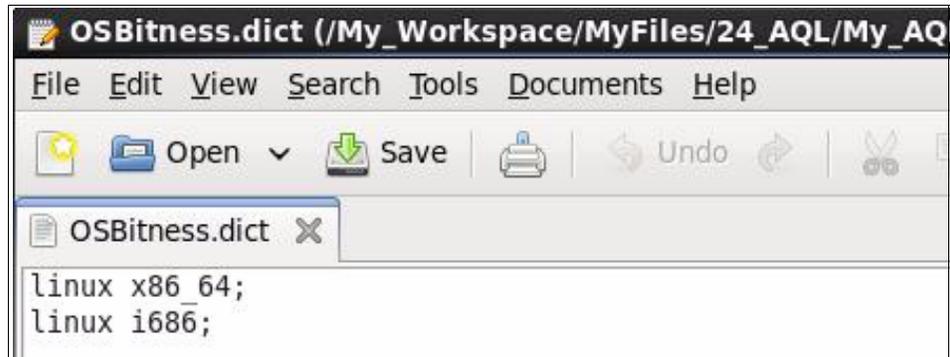


Figure 13-56 Contents of external dictionary

Figure 13-57 displays the contents of our external table. Notice the required column header line, as the first line of this file. These column names must match the external table definition from the accompanying AQL script, although the column orders might differ between actual file and AQL definition.

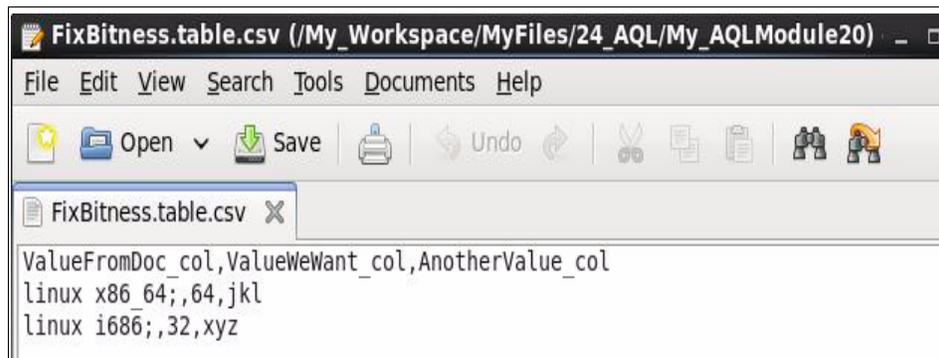


Figure 13-57 Contents of external table

Because we added runtime dependencies to our AQL script, the Text Analytics runner must now be supplied with additional information. The example is displayed in Figure 13-58 on page 459 through Figure 13-60 on page 460.

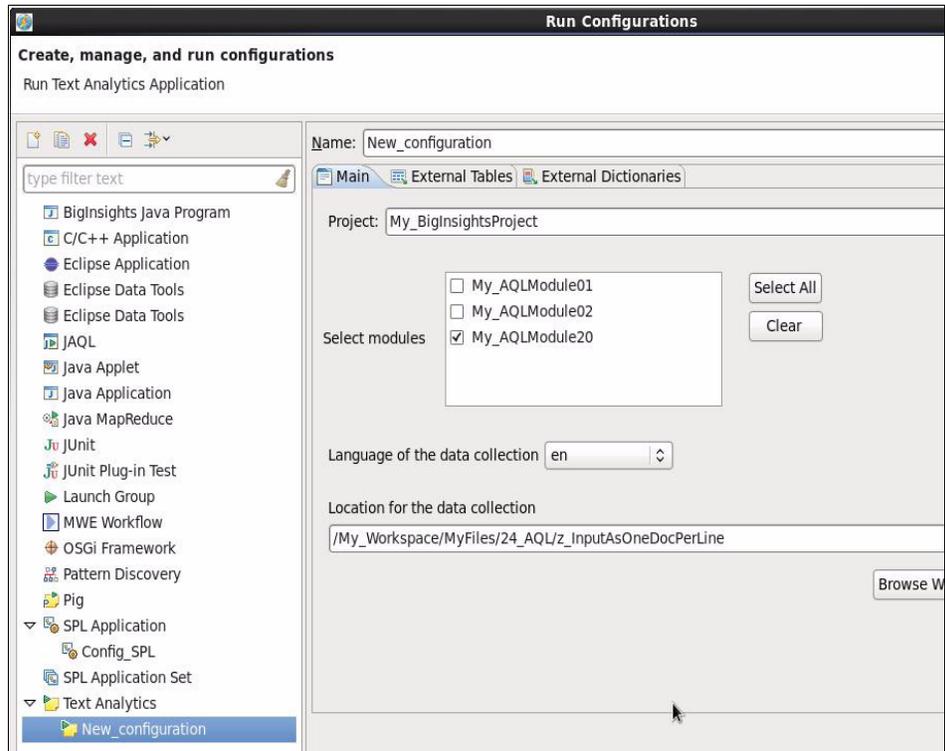


Figure 13-58 Text Analytics runner, Main tab, part 1 of 3

Consider the following information about Figure 13-58:

- ▶ You have seen this Main tab to the Text Analytics runner before.
- ▶ Nothing is new here, other than we must select the correct AQL module (this is always true) and we must, for the first time, access the two additional tabs to this dialog box.

Figure 13-59 shows the External Tables tab.

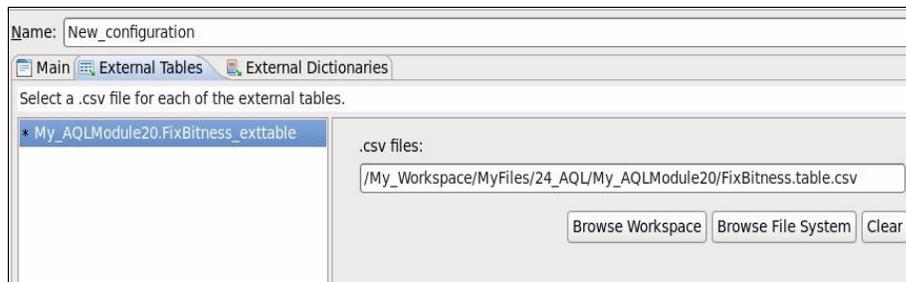


Figure 13-59 Text Analytics runner, External Tables tab, part 2 of 3

Consider the following information about Figure 13-59 on page 459:

- ▶ Figure 13-57 on page 458 shows file naming and file format requirements for AQL external tables.
- ▶ Here we specify the location of this external table, this file. If there is an error with the file, a red “X” is displayed on the tab.
- ▶ The left side of this display highlights the specific external table for which you are entering a value, if you have multiples from which to choose.

Figure 13-60 displays the External Dictionaries tab, the final tab we address.

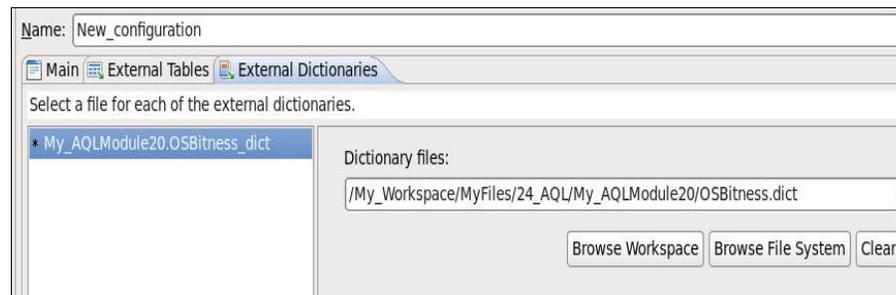
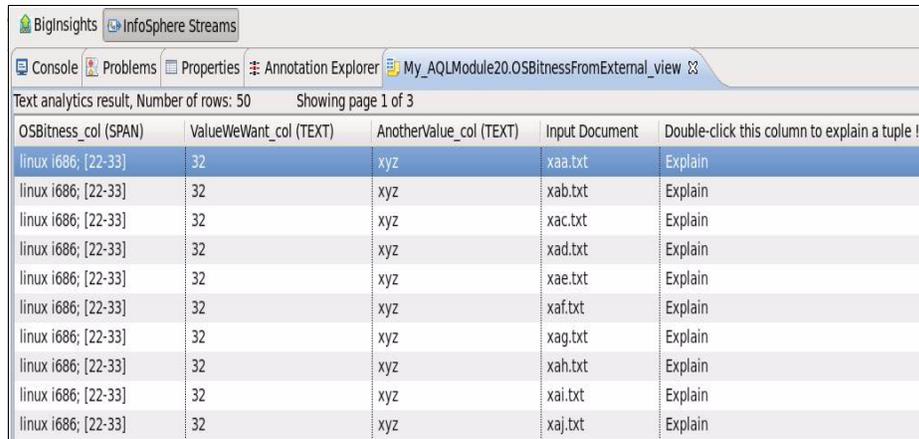


Figure 13-60 Text Analytics runner, External Dictionaries tab, part 3 of 3

Consider the following information about Figure 13-60:

- ▶ Not much is new here, we are merely specifying the path to the external dictionary displayed in Figure 13-56 on page 458.
- ▶ As before, the left side of the display highlights the specific external dictionary for which you are entering a value, if you have multiples.

Figure 13-61 displays the output from a successful run of this AQL script.

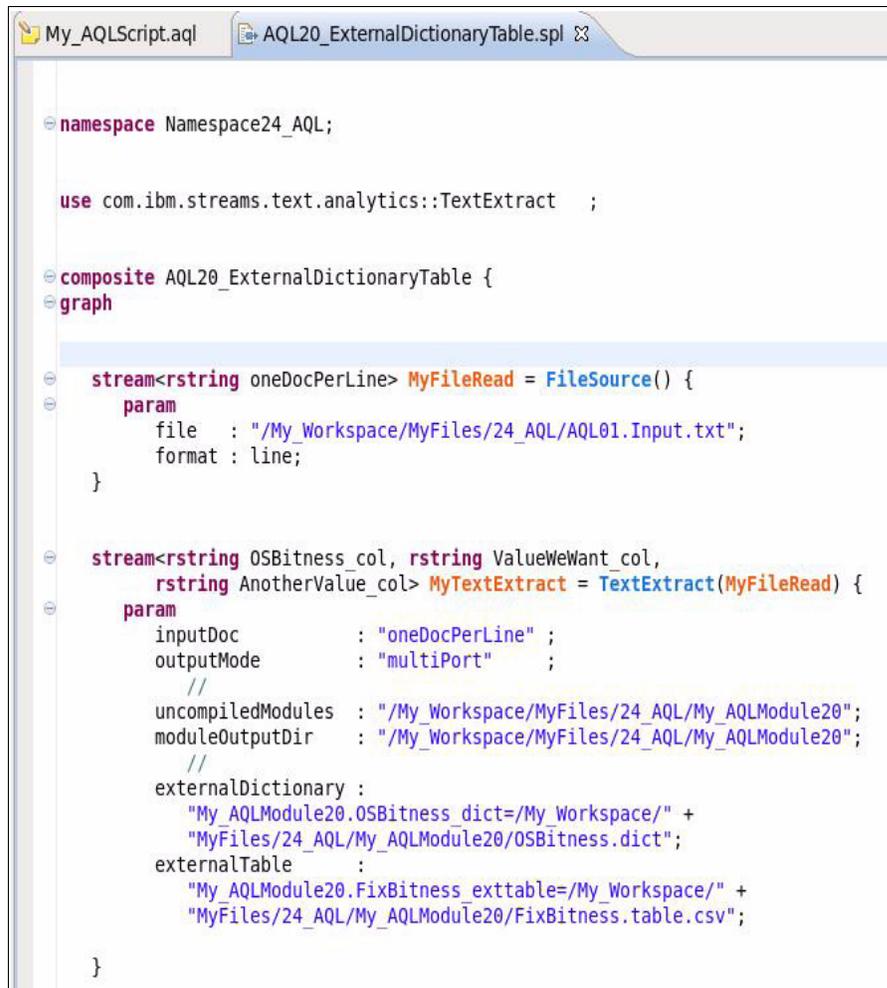


Text analytics result, Number of rows: 50 Showing page 1 of 3

OSBitness_col (SPAN)	ValueWeWant_col (TEXT)	AnotherValue_col (TEXT)	Input Document	Double-click this column to explain a tuple !
linux i686; [22-33]	32	xyz	xaa.txt	Explain
linux i686; [22-33]	32	xyz	xab.txt	Explain
linux i686; [22-33]	32	xyz	xac.txt	Explain
linux i686; [22-33]	32	xyz	xad.txt	Explain
linux i686; [22-33]	32	xyz	xae.txt	Explain
linux i686; [22-33]	32	xyz	xaf.txt	Explain
linux i686; [22-33]	32	xyz	xag.txt	Explain
linux i686; [22-33]	32	xyz	xah.txt	Explain
linux i686; [22-33]	32	xyz	xai.txt	Explain
linux i686; [22-33]	32	xyz	xaj.txt	Explain

Figure 13-61 Output from AQL script, view accessed from Annotation Explorer

Figure 13-62 and Figure 13-63 on page 463 display the accompanying Streams application that runs the AQL script.



```
namespace Namespace24_AQL;

use com.ibm.streams.text.analytics::TextExtract ;

composite AQL20_ExternalDictionaryTable {
graph

stream<rstring oneDocPerLine> MyFileRead = FileSource() {
param
file : "/My_Workspace/MyFiles/24_AQL/AQL01.Input.txt";
format : line;
}

stream<rstring OSBitness_col, rstring ValueWeWant_col,
rstring AnotherValue_col> MyTextExtract = TextExtract(MyFileRead) {
param
inputDoc : "oneDocPerLine" ;
outputMode : "multiPort" ;
//
uncompiledModules : "/My_Workspace/MyFiles/24_AQL/My_AQLModule20";
moduleOutputDir : "/My_Workspace/MyFiles/24_AQL/My_AQLModule20";
//
externalDictionary :
"My_AQLModule20.OSBitness_dict=/My_Workspace/" +
"MyFiles/24_AQL/My_AQLModule20/OSBitness.dict";
externalTable :
"My_AQLModule20.FixBitness_exttable=/My_Workspace/" +
"MyFiles/24_AQL/My_AQLModule20/FixBitness.table.csv";
}
}
```

Figure 13-62 Streams application, part 1 of 2

Consider the following information about Streams application in Figure 13-62:

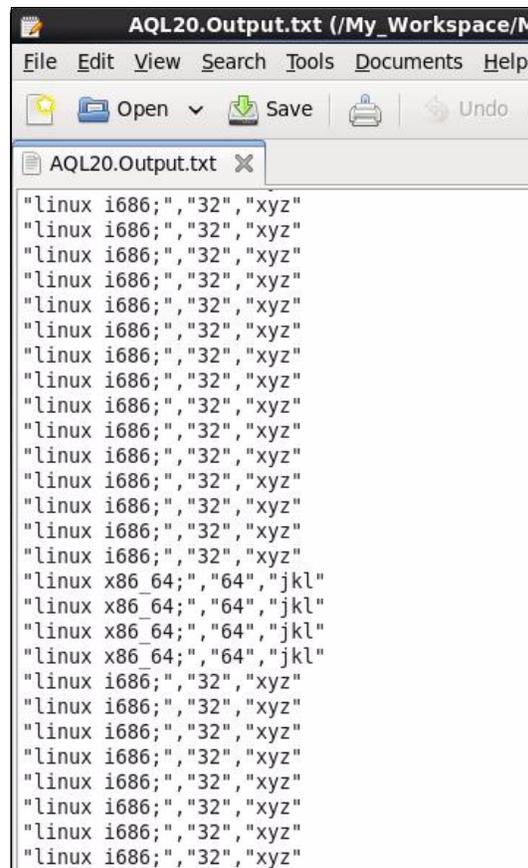
- ▶ The only new aspects to this Streams application (the only new topics we have not covered previously) are the two parameters: externalDictionary and externalTable.
- ▶ Both parameters have the following format:
AQL_ModuleName.AQL_ObjectName = Asolute_Or_Relative_Pathname_To_File

Figure 13-63 completes this Streams application source code listing, and offers no new techniques.

```
( ) as MySink = FileSink(MyTextExtract) {  
  param  
    file : "/My_Workspace/MyFiles/24_AQL/AQL20.Output.txt";  
    format : csv;  
}  
  
}
```

Figure 13-63 Streams application, part 2 of 2

Figure 13-64 displays sample output from this Streams application.



```
"linux i686;", "32", "xyz"
"linux x86_64;", "64", "jkl"
"linux x86_64;", "64", "jkl"
"linux x86_64;", "64", "jkl"
"linux x86_64;", "64", "jkl"
"linux i686;", "32", "xyz"
```

Figure 13-64 Sample output from Streams application

13.6.1 The kitchen sink AQL script and Streams application

This is the last AQL script in this section, and is used to demonstrate several AQL and Streams application topics:

- ▶ Having multiple output views in the AQL script
- ▶ Using the AQL minus clause
- ▶ Outputting the Document from the AQL view, including SPAN attributes

Figure 13-65 and Figure 13-66 on page 467 display our new AQL script.



```
My_AQLScript.aql  AQL21_TwoOutputViews.spl

module My_AQLModule21;

-----

create dictionary OSBitness_dict
as
(
  'linux x86_64;' ,
  'linux i686;'
);

create view OSBitnessInDict_view
as
extract dictionary 'OSBitness_dict'
on D.text as OSBitness_col
from Document D;

create view Filter64_view
as
select
  V.OSBitness_col
  GetBegin(V.OSBitness_col) as BeginOffset_col ,
  GetEnd(V.OSBitness_col)  as EndOffset_col   ,
  D.text                    as InputDocument_col
from
  OSBitnessInDict_view V,
  Document              D
where Equals(GetText(V.OSBitness_col), 'linux x86_64;');

output view Filter64_view;
```

Figure 13-65 AQL script, part 1 of 2

Consider the following information about Figure 13-65 on page 465:

- ▶ We use our same sample input data file from Example 13-1 on page 381.
- ▶ The first three AQL statements in Figure 13-65 on page 465 (module, create dictionary, and create view), offer no new techniques.

- ▶ The create view Filter64_view statement has several new concepts:
 - We output D.text from the AQL standard object, the Document view.

Note: We have not documented this set of topics before, and will only briefly mention it here.

AQL has several directives that can be applied to the input document itself. The most common use is when you are processing HTML or XML text. AQL can be instructed to automatically remove the HTML and XML tags.

If you sent the input document to AQL, why do you need to output it? Because you can use AQL to remove the tags; this, and other capabilities.

- We also use the GetBegin() and GetEnd() functions, on the SPAN column, which was produced from our extract dictionary view.

Note: We discussed SPAN data types after the example in Figure 13-26. A SPAN contains your matching text and also the offsets (beginning and ending) into the input document, where the match was found.

Previously we extracted the matching text from SPAN, using GetText(). With GetBegin() and GetEnd(), now you can also get the indices into the input Document.

Why would you want these values? There is not an obvious use for this. All of your text processing should be done inside AQL; this is what AQL does really well.

If you are sending more than the text from a SPAN back to the consumer, odds are the consumers are performing work that should have been done in AQL.

- The where clause comparing a column to a literal value is somewhat new; previously, where clauses were only joining tables and views.
- This view outputs four columns.
- This is the first of multiple output view statements that are present in this AQL script.

Figure 13-66 displays the remainder of this AQL script.

```

output view Filter64_view;

-----

create view NotInFilter64_view
as
(select V1.OSBitness_col from OSBitnessInDict_view V1)
minus
(select V2.OSBitness_col from Filter64_view      V2);

output view NotInFilter64_view;

```

Figure 13-66 AQL script, part 2 of 2

Consider the following information about Figure 13-66:

- ▶ This view displays use of the minus construct.

Figure 13-27 on page 419 displays a union all construct to concatenate the output from multiple views. Our specific case was to output matches not found in our dictionary; we generated *matches*, then we generated *not matches*, “unioned,” and output both lists as one.

We use a minus construct to intersect two or more views, to remove matches found in one or more lists (one or more views). The use for this technique is another means to remove false positives, or to perform scoring (identify matches as belonging to specific sub-groups).
- ▶ One view contains four columns, and the other view contains one column.
- ▶ This is the second AQL view we output in this AQL script.

Figure 13-67 on page 468 and Figure 13-68 on page 469 display the Streams application to run this AQL script.

```

namespace Namespace24_AQL;

use com.ibm.streams.text.analytics::TextExtract ;

composite AQL21_TwoOutputViews {
graph
    stream<rstring oneDocPerLine> MyFileRead = FileSource() {
    param
        file : "/My_Workspace/MyFiles/24_AQL/AQL01.Input.txt";
        format : line;
    }

    (
    stream<rstring OSBitness_col, int32 BeginOffset_col,
        int32 EndOffset_col, rstring InputDocument_col>
        MyTextExtract1 as Out1;
    stream<rstring OSBitness_col> MyTextExtract2 as Out2
    )
    = TextExtract(MyFileRead) {
    param
        inputDoc : "oneDocPerLine" ;
        //
        outputMode : "multiPort" ;
        passThrough : false ;
        //
        uncompiledModules : "/My_Workspace/MyFiles/24_AQL/My_AQLModule21";
        moduleOutputDir : "/My_Workspace/MyFiles/24_AQL/My_AQLModule21";
    }
}

```

Figure 13-67 Streams application, part 1 of 2

Consider the following information about Figure 13-67:

- ▶ The first real line offering new techniques is the Streams TextExtract operator.
 - This operator has one input port (one input stream), as all TextExtract operators do.
 - This operator has two output ports: MyTextExtract1 (also known as Out1), and MyTextExtract2 (also known as Out2).

The first port outputs four columns; the second port outputs one column. The columns that are related to the SPAN GetBegin() and GetEnd() must be cast as type integer 32-bit (int32).

- The outputMode parameter is still set to multiPort, and we added a new parameter named, passThrough, which is set to false.

These two parameters work together and, as configured, support this text operator receiving output from two output views in the associated AQL script, and passing them as two output streams.

The one input port is our input document.

If the associated AQL script had three or more output view statements, this operator would need three or more (an equal number) of output ports.

Note: We have not exhausted this topic. The Streams TextExtract operator can be variably configured to have additional ports to output just the input document, and to output (tuples) with given cardinality.

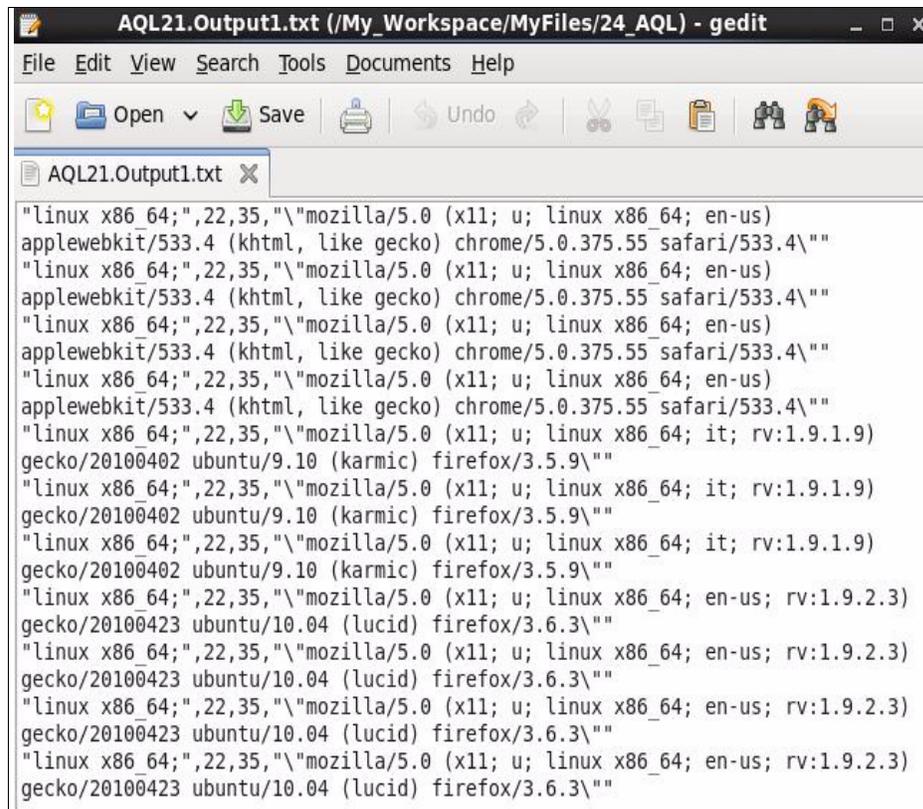
Although the TextExtract operator has more capabilities than are demonstrated here, *functionally* we believe we have demonstrated most or all of what you will want to do in the real world.

Figure 13-68 displays the remainder of this Streams application, with no new functionality.

```
() as MySink1 = FileSink(MyTextExtract1) {
  param
  file : "/My_Workspace/MyFiles/24_AQL/AQL21.Output1.txt";
  format : csv;
}
() as MySink2 = FileSink(MyTextExtract2) {
  param
  file : "/My_Workspace/MyFiles/24_AQL/AQL21.Output2.txt";
  format : csv;
}
}
```

Figure 13-68 Streams application, part 2 of 2

Figure 13-69 displays the sample output file of our four-column AQL view.

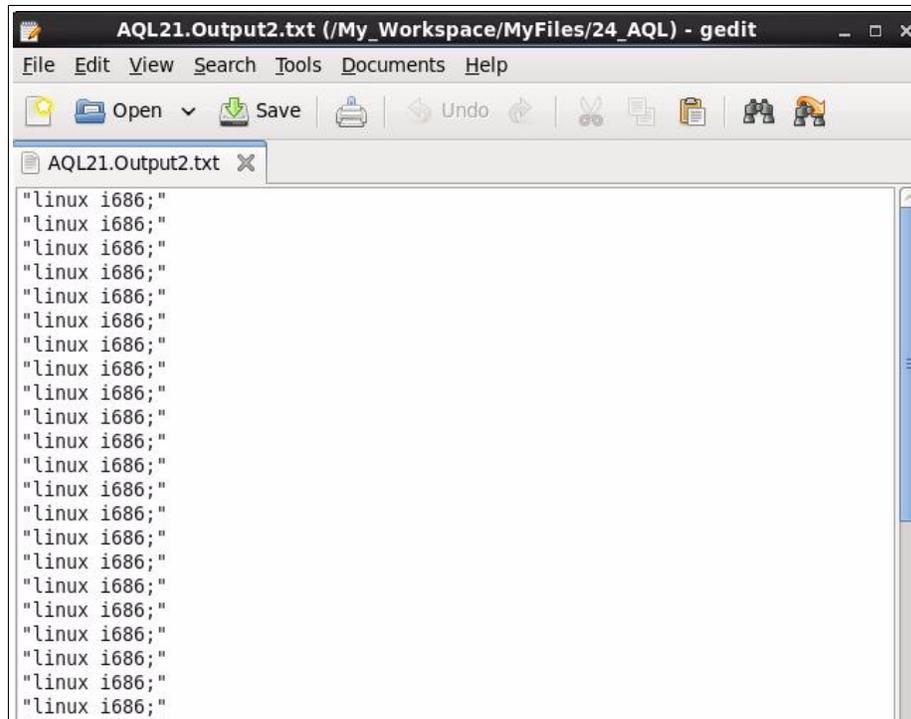


```
AQL21.Output1.txt (/My_Workspace/MyFiles/24_AQL) - gedit
File Edit View Search Tools Documents Help
Open Save Undo
AQL21.Output1.txt X
"linux x86_64;",22,35,"mozilla/5.0 (x11; u; linux x86_64; en-us)
applewebkit/533.4 (KHTML, like Gecko) Chrome/5.0.375.55 Safari/533.4\"
"linux x86_64;",22,35,"mozilla/5.0 (x11; u; linux x86_64; en-us)
applewebkit/533.4 (KHTML, like Gecko) Chrome/5.0.375.55 Safari/533.4\"
"linux x86_64;",22,35,"mozilla/5.0 (x11; u; linux x86_64; en-us)
applewebkit/533.4 (KHTML, like Gecko) Chrome/5.0.375.55 Safari/533.4\"
"linux x86_64;",22,35,"mozilla/5.0 (x11; u; linux x86_64; en-us)
applewebkit/533.4 (KHTML, like Gecko) Chrome/5.0.375.55 Safari/533.4\"
"linux x86_64;",22,35,"mozilla/5.0 (x11; u; linux x86_64; it; rv:1.9.1.9)
Gecko/20100402 Ubuntu/9.10 (Karmic) Firefox/3.5.9\"
"linux x86_64;",22,35,"mozilla/5.0 (x11; u; linux x86_64; it; rv:1.9.1.9)
Gecko/20100402 Ubuntu/9.10 (Karmic) Firefox/3.5.9\"
"linux x86_64;",22,35,"mozilla/5.0 (x11; u; linux x86_64; it; rv:1.9.1.9)
Gecko/20100402 Ubuntu/9.10 (Karmic) Firefox/3.5.9\"
"linux x86_64;",22,35,"mozilla/5.0 (x11; u; linux x86_64; en-us; rv:1.9.2.3)
Gecko/20100423 Ubuntu/10.04 (Lucid) Firefox/3.6.3\"
"linux x86_64;",22,35,"mozilla/5.0 (x11; u; linux x86_64; en-us; rv:1.9.2.3)
Gecko/20100423 Ubuntu/10.04 (Lucid) Firefox/3.6.3\"
"linux x86_64;",22,35,"mozilla/5.0 (x11; u; linux x86_64; en-us; rv:1.9.2.3)
Gecko/20100423 Ubuntu/10.04 (Lucid) Firefox/3.6.3\"
"linux x86_64;",22,35,"mozilla/5.0 (x11; u; linux x86_64; en-us; rv:1.9.2.3)
Gecko/20100423 Ubuntu/10.04 (Lucid) Firefox/3.6.3\"
```

Figure 13-69 First sample output file

The values "22,35" in Figure 13-69 reflect the beginning and ending offsets where the given text extract match was found in the input document. The trailing portion of each line contains the input document; the AQL object named D. text.

Figure 13-70 shows the output of our second AQL view.



```
"linux i686;"
```

Figure 13-70 Second sample output file

13.7 Topics not covered

As much as we detail AQL and Streams integration with AQL in this chapter, there is much more information and even major topics that we did not include.

As examples, we did not include these topics:

- ▶ The AQL Profiler view
The AQL Profiler view details resource utilization of your AQL script, and provides data that might be useful for tuning and debugging.
- ▶ Demonstration of AND, OR, and NOT in AQL predicates
- ▶ Demonstration of predicates named Follows(), FollowsTok(), MatchesRegex() (similar to MatchesDict, but for Regex), Overlaps(), or NotNull()
- ▶ External views, modularizing your AQL code (code re-use), or using AQL code from multiple AQL modules

- ▶ AQL user-defined functions.

The Streams online information center is available for more information:

<http://pic.dhe.ibm.com/infocenter/streams/v3r0/index.jsp>

AQL user-defined functions are available in the information center:

<http://ibm.co/111YKJz>

You can also search for “AQL user defined functions” in the first link.

- ▶ Streams supplied utilities to generate Streams applications from your AQL scripts.
- ▶ AQL parts of speech.

AQL has a huge area of capability wherein it can detect nouns, predicates, tense (past, future, current time reference) to speech, and more.

Parts of speech is important for determining sentiment. For example, a customer writes this regarding your product:

The X-60 is outstanding, but the new model .. not so much.

Note: Customer sentiment, and parts of speech are easily the pinnacle of text processing. It is the most difficult, perhaps the most elusive, and perhaps the most valuable to mine results from.

Fortunately, AQL has robust capabilities here, having more than 20 built-in speech identifiers: pronouns, possessive pronouns, on and on.

To get started, search for “parts of speech” in the information center.

If you need to calculate sentiment from text similar to that offered here, where do you even begin? The text contains past and current references; it uses jargon.

You begin with AQL parts of speech. You could add your company’s product names to a dictionary, and let AQL discern tense (the time-based reference), predicate reference, and so on.



IBM Accelerator for Telecommunications Event Data Analytics V1.2

In this chapter, we introduce the Accelerator for Telecommunications Event Data Analytics (TEDA) Version 1.2. We show the main features of TEDA and provide simple exercises, teaching you how to implement your own use cases.

14.1 Overview of TEDA

The IBM Accelerator for Telecommunications Event Data Analytics provides a framework to implement telecommunications applications. It is derived from a solution developed for a large scale telecommunications provider using InfoSphere Streams version 1.2. As other telecommunication providers have similar use cases, this solution was generalized and released as sample application with InfoSphere Streams version 3.0.

Although TEDA is released with Streams, it is an optional component that is delivered in its own installation package. To find the appropriate package, see the information in 14.2, “Installing TEDA” on page 475. If you install TEDA onto your system, you have a sample application showing real-time mediation and analytics on a large volume of mock-up call detail records (CDR). Figure 14-1 shows you the structure of TEDA.

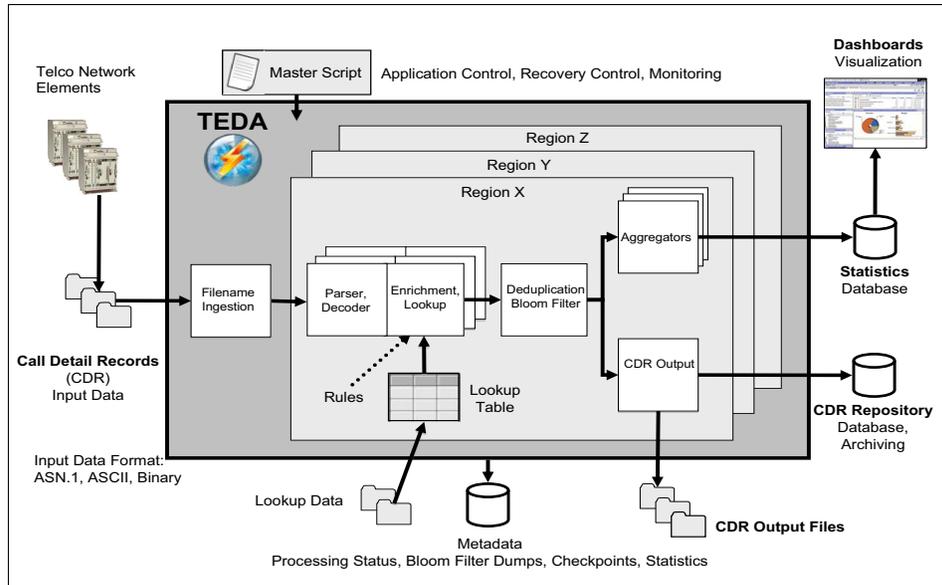


Figure 14-1 TEDA overview

The TEDA functionality can be generally summarized as follows: Network elements produce data record files, which are copied into TEDA’s input data directories. In the Filename Ingestion part, the input directories are scanned periodically for new input files and the file names are reported to the appropriate processing stage, depending on the origin of the file. This stage parses the files and produces one data tuple per call detail record. TEDA then feeds these tuples into the business logic where the data is enriched following the given rules and

by looking up the given business data. Then, the Bloom Filter processes the tuples and detects duplicate records, after which the data reaches the output stages of the application. Good, erroneous, and duplicate event records are written to respective tables in the CDR Repository Database or to appropriate files in the output directory. Additionally, TEDA feeds good data tuples to the aggregation operators, processes them to gain new business insights, and writes the resulting data into the Statistics Database.

Figure 14-1 on page 474 shows several boxes that are stacked. In the case of the Aggregators, this symbolizes that there are many aggregation operators. In the case of the region and parser/enrichment boxes the stack of boxes symbolizes the scalability of TEDA. By modifying a few values in the configuration files, you create an application processing data of several regions and even parallelize the processes within each of the regions. Thus, you can easily achieve the high processing throughput you need and expect from the InfoSphere Streams platform.

Furthermore, TEDA writes internal status information to a metadata database to keep track of its current state. The application commits data to the databases based on configurable parameters like the number of processed files and timeouts. On startup, the Master Script reads this state data and determines if it needs to recover from fault conditions, for example, hardware or database failures. These checkpoints enhance the reliability of the application.

Changing configuration values allows you to align an existing application to your data streams and reliability restrictions. But the real challenge lies in adapting the TEDA application template to your use case. We demonstrate this with a simple example in 14.4, “Customizing TEDA” on page 483.

14.2 Installing TEDA

The primary source for information is the InfoSphere Streams Information Center:

<http://pic.dhe.ibm.com/infocenter/streams/v3r0/index.jsp>

A search for TEDA in the information center will produce results leading to product overviews, detailed descriptions, troubleshooting guides, and installation instructions. In the “Installing IBM Accelerator for Telecommunications Event Data Analytics” section of the information center has guides for obtaining the software, installing the software stack, and installing TEDA. After you install the prerequisites, follow the instructions in the topic “Installing IBM Accelerator for Telecommunications Event Data Analytics interactively with the wizard or console.”

That topic describes three ways to install TEDA on your machine:

- ▶ Setup Wizard
- ▶ Installation Wizard
- ▶ Installation through the console

The suggestion is to use the Setup Wizard provided with the TEDA archive, especially when you are installing TEDA for the first time or for the first time on a particular machine. It leads you interactively through the installation process and checks the validity of your environment. If you extracted the TEDA delivery archive into the `~/install` path, run the Setup Wizard with the following steps:

1. Change to the TEDA-Setup-Wizard directory:

```
cd ~/install/TEDA-Setup-Wizard
```

2. Run the TEDA Setup Wizard Perl script:

```
./tedaSetupWizard.pl
```

Follow the interactive process on the console by reading the displayed instructions for every step and deciding on the presented choices. The wizard finally offers to run the TEDA sample application for a final test of your installation. However, you might prefer to do this manually by following the steps in the “Running the IBM Accelerator for Telecommunications Event Data Analytics demo application” topic in the information center. If the sample application runs successfully, you know that your installation is fine. You will copy the sample’s directory to start your own customizing exercises.

Table 14-1 shows the main directories of the TEDA project and what data is stored in them.

Table 14-1 Main directories of the TEDA project

Path	Directory contents
demo	TEDA sample application.
demo/application	All the parts of the application that are not organized in the toolkit, such as the operators that are put into this directory.
demo/application/apps/main	SPL code of the application, for example the FileNameIngestion, RulesEngine, and DeDuplication composites.
demo/application/common	Common C++ and Java code, for example the parser libraries.
demo/application/config	Configuration files of the application

Path	Directory contents
demo/application/ddl	Data Definition Files. These files are used to automatically create the needed tables in the databases.
demo/application/rules	Rule definition files that are the input for the Rules Compiler.
demo/application/scripts	Scripts used to run the TEDA application.
demo/toolkits	Streams primitive operators that belong to TEDA, for example the FileReader or BloomFilter operators.
rules-compiler	Rules Compiler Java archive. This tool reads the rule definition files and generates the SPL code, which is integrated into the Rules Engine composite.
testdata	Data directories used to store input, output, processed, and status files of the application.

You can find the latest information regarding TEDA in the “Product Overview / Release Notes” section in the information center. Also the “Troubleshooting IBM Accelerator for Telecommunications Event Data Analytics” topic might help.

14.3 Understanding concepts and terms

In this section, we explain more of the inner workings of the Accelerator for Telecommunications Event Data Analytics.

14.3.1 Application components

The parser component of the sample application is the ASN.1 Parser, which is used by the FileReader operator. It reads the input files and decodes the binary data. After a CDR is decoded, the attributes are sent as tuples to the downstream operators. The parser is implemented in the FileReader operator. This operator loads C/C++ libraries that contain the code to parse the files. These libraries are implemented using the Lionet ASN.1 compiler. To adapt the application to ASN.1 grammar, new libraries must be developed. The topic “Implementing a parser” provides the instructions to develop a parser on your own.

You can find the description in the information center for InfoSphere Streams version 3.1 at the following address:

<http://pic.dhe.ibm.com/infocenter/streams/v3r1/index.jsp>

If the application needs to apply business logic on the decoded CDRs, this can be implemented by using the Rules Compiler. The Rules Compiler is a Java tool, which takes a rule definitions file as input and translates the specified rules into the SPL language. Its output is an SPL composite that is invoked by the main application. This Rules Engine composite applies the logic to the CDR tuples emitted by the parser. The rule definition language is proprietary, and allows for configuring in-memory lookup tables, which are loaded into the application on startup. If you do not want to use the Rules Compiler to develop your business logic, you can implement a Streams composite operator by using the SPL language. The information center provides descriptions for how to create Composite operators.

The deduplication uses the Bloom Filter operator, whose purpose is to detect duplicate tuples in the input data stream. In that sense, it is similar to the DeDuplicate operator in the standard SPL toolkit, but the internal implementation is completely different. The Bloom Filter operator stores only a few bits per tuple in memory, and therefore, is memory-efficient. The drawback is that the detection is not accurate, because there is a small probability of detecting a tuple as duplicate although it is unique (false positive). The Bloom Filter operator provides a parameter to define the acceptable error probability. The lower you set this value, the more memory will be allocated by the operator to achieve the required behavior. A value of 0.000001 means that one false duplicate per one million records is acceptable. Thus, there is a trade-off between memory usage and required accuracy.

The Bloom Filter needs a bit field for each tuple as input. This bit field must be computed from the input tuple and is then passed to the Bloom Filter as additional tuple attribute. If the operator detects that the tuple is a duplicate based on this input bit field, it will set the tuple attribute `isDuplicate` to true. So the downstream operators can decide what to do with this tuple.

In the TEDA sample application, the input bit field is created by concatenating several CDR attributes that are likely to be unique for each CDR into an rstring. Then, a SHA2 hash on this string is computed and passed to the Bloom Filter as an input bit field.

A single Bloom Filter operator in TEDA is responsible for storing the deduplication information of one day. If you have the requirement to deduplicate CDRs that are older than one day, you can configure additional Bloom Filter operators (say, one bloom filter per day) deployed by TEDA. In this case, a timestamp attribute from the CDR will be used to calculate the corresponding day and route the CDR to the Bloom Filter holding the data for that day.

Aggregation operators are Streams primitive operators or SPL composites you can implement to further process the CDR tuples after they pass the Bloom Filter. In the TEDA sample application and the example in 14.4.6, “Exercise 3: Adding an aggregation operator” on page 504, the aggregations on tuple attributes are calculated, for example, the call duration on a per-minute basis. When a punctuation is received, signaling that a file has been completely read, the aggregated values are written to the external database tables. From here the results can be read and used by reporting systems, such as IBM Cognos®, to display dashboards. The usage is not limited to aggregation. The operators are just a mechanism to plug in additional functionality.

14.3.2 Application infrastructure

In TEDA you configure multiple *groups* to parallelize the processing of files. Groups are numbered from 1 to n. Each group contains the complete set of functional blocks (Parser, Rules, Bloom Filter, Output), thus parallelizing the processing. For each input file, the group number is derived from the file name, based on a configurable name pattern. This is done in the `FilenameIngestion` composite. The file is then routed to the corresponding group. Users may choose to configure different groups for files from different geographical regions. Synonymous terms for this concept are *circles* or *regions*.

Within a group, you can configure *intra-circle parallelization* for file processing. If the Parallelization Degree value is higher than one, the group has an appropriate number of parallel branches containing the parser and the rules processor components. Data files for the circle are distributed in round-robin fashion to the branches. Note that the number of Bloom Filter and Aggregation operators within the group are not affected, thus, the data of all branches will be routed through the same Bloom Filter operators.

After the CDR tuples go through the business logic and the Bloom Filters, they are forwarded to the `CDRSp1t1t1er` operator. This operator routes the data tuples to different *output splits* consisting of aggregation operators and file or database output sinks. It has a function that may be adapted to examine tuple attributes and send the tuple to different output ports, based on that evaluation.

There are always three output splits per default:

- ▶ The first one is for CDR tuples that were processed without problems
- ▶ The second one is for CDR tuples where problems were detected. Each CDR tuple has an error flag attribute named **errorInd** that will be evaluated to determine if the tuple should go to the error output. This attribute is set by the business logic to indicate that some data in the CDR is incorrect.
- ▶ The third output split is the path of the duplicates. CDR tuples with the **isDuplicate** tuple attribute set to true by the Bloom Filter will be directed to this path.

The *Master Script* `startup.pl` resides in the `TEDA/demo/application/scripts` directory and controls the operation of TEDA. On invocation, it performs several tasks, such as creating a Streams instance, starting the instance, compiling the TEDA application, and submitting it as a Streams job. You can specify several parameters on the command line to change the behavior of the script, for example, you can instruct it to use an existing Streams instance instead of creating the instance. After the Master Script starts TEDA, it continues to run in the background and periodically monitors the health of TEDA by means of the **streamtool** command. If it detects problems, it tries to recover from the situation by measures described in 14.3.4, “Fault tolerance” on page 481.

14.3.3 Configuration

The configuration for TEDA is done through two files residing in the `TEDA/demo/application/config` directory:

- ▶ The `config.cfg` file contains the simpler configuration parameters as name value pairs.
- ▶ The `tedaConfig.xml` file contains configuration data in XML format. The schema for this XML file is specified in the file `tedaConfig.xsd` file residing in the same directory. The configurations in this file are more complex than the ones in `config.cfg`, for example, it contains the list of the tuple attributes emitted by the parser. This XML file is used as input for the `generateTeda.pl` script residing in `TEDA/demo/application/scripts`. This Perl script reads the XML configuration file and creates several artifacts, for example SPL and C++ code files. After, the modified application may be compiled. You can edit the `tedaConfig.xml` file with any XML or text editor.

You will get to know this configuration mechanism when you work through the examples in 14.4, “Customizing TEDA” on page 483.

14.3.4 Fault tolerance

To allow recovery from failures TEDA uses the concept of *checkpoints*. Checkpoint information is written at certain configurable intervals into a database and the data in the Bloom Filters is stored into files. This metadata information is used to determine which files need reprocessing when TEDA attempts to recover from failures.

TEDA uses a local DB2 database to store the internal state metadata of the file processing in a set of tables. This database must be setup before TEDA can be used. The name of the database can be changed in the configuration file. Its default name is ISS. The tables maintain the status information for each file, each Bloom checkpoint, and each commit to the database. Checkpoints are configurable and happen on file boundaries. The following tables are used:

- ▶ The File Status Table (FST) table is used to track the current processing state of every single file that is processed by TEDA. A row is created after the file has been detected and directed to the group's processing stages. Updates of the row are performed by following processing stages, for example the parser updates the FST with the time when processing starts for a file.
- ▶ The Bloom Checkpoint Table (BCT) table collects the status data of every Bloom checkpoint. An entry is identified by the circle, the Bloom Filter operator, and the checkpoint ID it belongs to. Whenever a checkpoint is written, the Bloom Filter operator also updates the data in the BCT. The data in this table is used to restore the last known good state of the Bloom Filter operators.
- ▶ The Commit Table (CT) table collects the status data for the applications commits. The DB Status Table (DB_STATUS_TABLE) table tracks of the database being up and running or down.
- ▶ The Recovery Checkpoint Table (RCT) table is used during Recovery Mode to organize the list of files per circle and checkpoint that need to be recovered.
- ▶ The Loader Checkpoint Table (LCT) table is used during Loader Mode to organize the list of output files per circle and checkpoint that need to be loaded to the database.

The TEDA application knows three *operation modes* that are controlled by the Master Script, namely Main, Recovery, and Loader Mode. In the Main Mode, which is the normal operation mode, call detail records are parsed, processed by the business logic, checked for duplicates, analyzed, and written to the database or file storage. When the production database becomes unavailable the output is automatically redirected from the database into output files. These files are used as input to load the tuple data into the database when it becomes available again (Loader Mode).

The Master Script periodically monitors the running Streams application and thus is able to detect problems, for example, crashed processing elements, failed jobs, or an unavailable database. Some counter measures require that the application is restarted. When this happens, the Master Script checks if the application needs special handling.

The FST, BCT, and CT tables are first queried to determine if data files need reprocessing. In this case, the script fills the RCT table with appropriate data and starts the TEDA application in *Recovery Mode*. The application first initializes the Bloom Filter operators with the data of the last checkpoint. Then, it reprocesses all input files that have been processed and committed to the database since the last checkpoint. When the Master Script detects that the recovery process finished successfully, it stops the TEDA application and cleans the system from files it no longer needs.

Next, the Master Script queries the FST, BCT, and CT tables to determine if there are output files that have data to be stored in the database. The script then populates the LCT table and starts the TEDA application in *Loader Mode*. When the Master Script detects that the loader process finished successfully, it stops the TEDA application and cleans the system. Then, the Master Script restarts the TEDA application in Main Mode and resumes the normal file processing.

Most activities related to checkpoint and recovery are implemented in the *Checkpoint Controller* operator. In Main Mode, the Checkpoint Controller signals all the Bloom Filter operators to create a checkpoint according to configurable parameters. When all Bloom Filters acknowledge the checkpoint, the Checkpoint Controller triggers a commit to the database. The status tables are updated accordingly.

14.4 Customizing TEDA

You can adapt the IBM Accelerator for the Telecommunications Event Data Analytics sample application to your use case.

14.4.1 Workflow overview

To customize TEDA, you do the following steps:

1. Install TEDA and setup your environment.
2. Create a copy of the TEDA sample application to work with.
3. Define your use case.
4. Define the data model.
5. Adapt the `tedaConfig.xml` file to your needs.
6. Adapt the business logic rules file to add enrichments and lookups.
7. Create analytics/aggregation operators using SPL or C++ if needed.
8. Compile the application.
9. Test the application.

14.4.2 Preparation

The most important step before you customize TEDA is to set up a working environment. If you follow the installation instructions in InfoSphere Streams 3.1 Information Center, be sure to give the sample application a test run. If the sample application finishes successfully, your environment has been set up correctly. Use the following steps:

1. The first step is the installation of the necessary software stack (such as InfoSphere Streams, DB2, and Perl) and of TEDA. Detailed information about the installation process is in the Information Center:

<http://pic.dhe.ibm.com/infocenter/streams/v3r1/index.jsp>

To simplify the following exercises, install the DB2 database, InfoSphere Streams, and TEDA onto the same host. The installation of the Cognos parts is not necessary unless you want to develop a Cognos dashboard for the data produced by an aggregation operator like the one we implement in 14.4.6, “Exercise 3: Adding an aggregation operator” on page 504. For the exercises we assume that you install TEDA in your home directory (`~/TEDA`).

2. After you install TEDA and verify that it is working, create a copy of the complete TEDA directory to work with, as shown with these commands:

```
cd TEDA
cp -r demo demo2
```

3. In the new copy, verify that the application compiles without errors by using the following commands:

```
cd demo2/application
make all
make clean
```

4. You need an additional database, named TTEST, to store the CDRs. To create this database, enter the following commands as the DB2 instance owner (this can take several minutes):

```
db2start
db2 create database TTEST
db2 terminate
```

5. To be able, to use the new database, add the following entry to your unixODBC configuration file (which is in the etc subdirectory in the unixODBC installation path, for example, /usr/local/unixODBC/etc/odbc.ini):

```
[TTEST]
Driver=DB2
```

6. To check whether your ODBC configuration works correctly, use the following command to connect to the database:

```
isql -v TTEST <db2InstUser> <password>
```

14.4.3 Defining the sample use case

Suppose you want to read files containing CDRs and write these CDRs to a DB2 database or to output files. In addition to that, the CDRs will be deduplicated and a small aggregation based on the call durations will be added. The call duration is calculated by the Rules Engine. For sake of simplicity, we will not use ASN.1 CDR files but plain CSV files as input. We also assume that we do not need any lookups for this case.

Note: Using CSV files as input can also serve as a quick way to prototype your application. An ASN.1 or binary data parser for your input files can be developed in parallel and exchanged with the CSV parser when it is done. The important point is, that the ASN.1 parser emits the same attributes as your CSV prototype. More information about implementing ASN.1 parsers is available in the information center.

Requirement list

A requirement list for this sample use case might be similar to the following list:

1. CSV files are processed from a default input directory.
2. The sample CDRs will contain fields such as recordId, called and calling IMSIs (International Mobile Subscriber Identity) and MSISDNs (Mobile Subscriber Integrated Services Digital Network-Number), call start and end times, and a release cause (see Table 14-2 on page 486).
3. We will enrich the CDRs with an additional field callDuration, which is calculated from the call start and call end times, using the rules file (see Table 14-3 on page 487).
4. No lookups are needed.
5. To allow for de-duplication, we will also enrich the CDRs with two attributes: CallStartDateTime, which contains a normalized timestamp for the call start time and CDRIdKey, which contains a hash code of the CDR (see Table 14-3 on page 487).
6. CDRs which contain datetime fields with incorrect string length, will be marked as erroneous CDRs and written to the error file/table.
7. The CDRs will be written to files in the first exercise and the TTEST database in the following exercises.
8. A new aggregation operator will be added, which calculates average call durations based on the callDuration of each CDR. The aggregation will be done over calls from the same minute, thereby reducing the amount of entries we need to write to the DB.
9. The aggregation results will be written to the ISS database into a new table.
10. Only one circle is used, and no intra-circle parallelization is configured.

Overview of implementation steps

The sample use case is implemented in the following three steps:

1. Perform the basic setup of the application and data formats using the **generateTeda.pl** tool. Also, create a sample input file, and run the application to process the file. In the first exercise, the CDRs are written to files.
2. Adapt the configuration files to write CDRs to the database instead. A simple rule is added to verify the length of the call start and end attributes and to calculate the call duration.
3. Add an aggregation operator written in SPL and write its results to a new database table.

Start by defining the attributes used in the sample use case.

Defining the data model

Basically, you define three sets of attributes:

- ▶ For the data fields that are emitted by the parser
- ▶ For the enrichment attributes added when deploying the Rules Engine
- ▶ For each aggregation operator that will be added.

The attributes will be added to the `tedaConfig.xml` file. After they are in the XML file, the **generateTeda.pl** tool will create the necessary code for these attributes in TEDA.

Input data

These attributes are extracted for each CDR by the parser. The parser reads the input files and sends out one tuple for each CDR. The tuple will contain the attributes listed in Table 14-2; the Field name column gives the name of the Tuple attribute. SPL data types `int32` for `neld`, `recordId`, `callType`, and `releaseCause` will be used. All others will be of type `rstring`. Because we are using the CSV parser, we also specify the column of each field within the CSV record.

The data format for the fictive CDRs is shown in Table 14-2.

Table 14-2 Example CDR data format

CSV field number	Field name	Description
1	<code>neld</code>	The network element ID that the CDR was processed by
2	<code>recordId</code>	The record ID of the CDR
3	<code>callType</code>	1=voice, 2=SMS
4	<code>callingImsi</code>	A numbers
5	<code>callingMsisdn</code>	A numbers
6	<code>calledImsi</code>	B numbers
7	<code>calledMsisdn</code>	B numbers
8	<code>callStartTime</code>	Timestamp for call start
9	<code>callEndTime</code>	Timestamp for call end
10	<code>releaseCause</code>	Call termination code (success, failure, and so on)

Enrichment data

These attributes are calculated by the Rules Engine, based on the attributes sent from the parser. In the output stream of the Rules Engine, they are added as new tuple attributes. Attributes from the parser can be passed through the Rules Engine or suppressed, so they will not appear in the output.

In this sample, we calculate the callDuration attribute from the callStartTime and callEndTime attributes from the CDR. The other two attributes, CallStartDateTime and CDRIDKey, are necessary for the Bloom Filter to work correctly. The CallStartDateTime attribute is used to determine if the CDR is either too old, or from the future. In both cases the CDR will not be processed by the Bloom Filter. The CDRIDKey is a hash string that uniquely identifies a CDR and that is stored in the Bloom Filter.

CallStartDateTime and CDRIDKey: These two attributes must be included in the output stream, even in case the Bloom Filter is disabled by using the BYPASS_BLOOMFILTER configuration parameter. However, they can be set to default values in this case.

Table 14-3 shows the enriched fields that the business logic will add.

Table 14-3 *Enrichment fields*

SPL type	Field name	Description
Int32	callDuration	The call duration
rstring	CallStartDateTime	The normalized timestamp for deduplication
rstring	CDRIDKey	The CDR hash code for de-duplication

In addition to these attributes, the business rules output stream will contain a standard attribute named errorInd. It is set to true by the business logic to mark a CDR as erroneous. When set, the CDR will be written to the error table/file. The errorInd attribute is included automatically by TEDA, so there is no need to configure it in the tedaConfig.xml file.

Aggregation data

The aggregation operator in this sample case receives a tuple with two attributes for each CDR: the normalized CallStartDateTime and the callDuration. Both of have been calculated by the business logic earlier. The aggregation operator will accumulate all call durations with respect to the minute the call was started. On receipt of punctuation, it calculates the average call duration for each time slot and sends out one tuple for each time slot (minute) containing the time stamp

and the averageDuration for storage in the database. The averageDuration attribute is of type int64. The aggregation operator fields are shown in Table 14-4.

Table 14-4 Aggregation operator fields

In/Out	Field Name	Description
in	callDuration	The call duration calculated by the rules
in/out	CallStartDateTime	The normalized timestamp calculated by the rules
out	averageDuration	The average call duration calculated by the aggregator

Description of the datetime input fields

In CDRs, the time stamps are often encoded in non-intuitive formats. For the sample use case, the time stamps are stored in strings of length 14, with the digit format, as shown in Example 14-1.

Example 14-1 Datetime input fields

00101601011320
SSMMHHDDmmYYCC

Where:

SS means second

MM means minute

HH means hour

DD means day

mm means month

YY means year

CC means century

So the above example string 00101601011320 expresses the datetime
1.1.2013 16:10:00

14.4.4 Exercise 1: Basic setup

In this exercise, we describe the basic setup, generate and compile the application, and run it with some manually created input files.

Note: Make sure you worked through 14.4.2, “Preparation” on page 483, so that your environment is set up correctly, and your copy of TEDA in the demo2 directory is compiling properly.

Before you start to customize TEDA, create a sample input file for the use case and copy it into the input directory of TEDA. After installation, there is a `testdata` directory in the TEDA directory. By default, TEDA looks for input files in a directory, such as the following location:

```
~/TEDA/testdata/input
```

Change to that directory and create the following input file with the CSV content shown in Example 14-2

```
CDR_RGNO_20110530000000.csv
```

Example 14-2 Sample data for CDR_RGNO_20110530000000.csv

```
21,100,0,123456,67890123,923456,97890123,00101601011320,00111601011320,0
21,101,0,123456,67890123,923456,97890123,00101601011320,28101601011320,0
21,102,0,123456,67890123,923456,97890123,00111601011320,30111601011320,0
28,100,0,123456,67890123,923456,97890123,00111601011320,30111601011320,0
28,200,0,123456,67890123,923456,97890123,00121601011320,30121601011320,0
28,300,0,123456,67890123,923456,97890123,00121601011320,30121601011320,0
28,300,0,123456,67890123,923456,97890123,00131601011320,30131601011320,0
28,300,0,123456,67890123,923456,97890123,00131601011320,30131601011320,0
28,300,0,123456,67890123,923456,97890123,00131601011320,30131601011320,0
```

The values in the columns reflect the input data defined in Table 14-2 on page 486. It contains nine CDRs coming from two network elements (IDs 21 and 28) with incremented record IDs (100-102 and 100-300). The time stamps for the CDRs vary.

Note: The directory for the input files can be changed in the `config.cfg` file by setting the parameter `WATCHEDDIR`. The file name pattern can be changed in the `FilenameIngestion.splmm` file.

Setting up the `tedaConfig.xml` file

The `tedaConfig.xml` file is the main configuration file for customizing TEDA. It is located in the `demo2/application/config` directory and used as input for the `generateTeda.pl` tool. Based on this file, the `generateTeda.pl` tool modifies several parts of the SPL code to adapt them to the configuration. Because it is an XML file and the corresponding schema definition is also located in the `config` directory (`tedaConfig.xsd` file), you can use any XML editor to work with this file. For this example, you can use a simple text editor is for modifications.

The `tedaConfig.xml` file contains four top-level elements with general configurations (groups, databases, aggregatorToOperator, and outputSchemas) and one top-level element named *attribute*. For each attribute defined in the data

model (see Table 14-2 on page 486 through Table 14-4 on page 488) we add an attribute element in `tedaConfig.xml`. But, start with the four general elements.

Open the existing `tedaConfig.xml` in a text editor and adapt the first elements to look like the contents of Example 14-3.

Example 14-3 Configuration File `tedaConfig.xml` (part)

```
...
<groups>
  <group id="0" parallelizationDegree="1"/>
</groups>

<databases>
  <database id="STAGING" name="TTEST" schema="CDRADM" user="db2inst1"
    password="ibm2blue" partitions="0" partitionMappingFile="TTEST"
    splitPartitionMappingFile="TTEST"/>
</databases>

<aggregatorToOperator>
</aggregatorToOperator>

<outputSchemas>
  <outputSchema name="CDRGoodStreamSchema" type="good"
    splitName="CDR_GOOD" database="STAGING" dbName="CDR_GOOD"
    mode="FILE" perGroup="true"/>
  <outputSchema name="CDRDupStreamSchema" type="duplicate"
    splitName="CDR_DUP" database="STAGING" dbName="CDR_DUP"
    mode="FILE"/>
  <outputSchema name="CDRErrorStreamSchema" type="error"
    splitName="CDR_ERR" database="STAGING" dbName="CDR_ERR"
    mode="FILE"/>
</outputSchemas>
...
```

The following configuration is the simplest. You can go through the list of element sections, step by step:

- ▶ **Groups:** Here you can configure the number of processing groups (also referred to as *circles* or *chains*), and the number of parallel parsers in each group. In our sample use case, we have one group only and no parallelization within this group.
- ▶ **Databases:** Here you must configure the databases that will be used by TEDA to write CDRs or aggregated data to. For each database, configure an ID, the

database name (the DSN in unixODBC), a database schema in which the tables will be created, and user and password information.

Note: Currently the partition parameters are not supported. Leave the partitions parameter at 0, and set the other two parameters (partitionMappingFile and splitPartitionMappingFile) to the same value as the name parameter.

- ▶ **AggregatorToOperator:** In this section, you can configure additional output paths for the application. The outputs are called *aggregators* because they are primarily used to perform aggregations on the data tuples provided by the parser and the Rules Engine. Aggregation operators can be implemented as SPL composites or primitive operators. Each aggregator can be configured to have different input and output tuple types. For this basic setup, leave the aggregatorToOperator element empty.
- ▶ **OutputSchemas:** Here you define characteristics of the output streams used in the application. Each output schema has a name, which is basically the name of an SPL tuple type definition. The type attribute is relevant only for internal processing. When adding new output schemas working on correct CDRs, set the type to "good". The splitName attribute is used internally to differentiate between the configured output streams. The splitName can be any string, but must be unique for each output schema. With the remaining attributes, you can specify where the output for this split is written to. You can reference a database ID, which must be contained in the <databases> element, and a table name in the database, where the records will be stored. With the mode parameter, you can decide if you want the output to go into a database table (value "SERIAL") or into plain files (value "FILE"). Because of some restrictions in the XSD file you need to reference a database, even if you use files only. This will be addressed in the next versions.

Note: TEDA always uses the three output schemas shown in Example 14-3 on page 490. Therefore, these three entries are always present in the <outputSchemas> element.

- ▶ The CDRGoodStreamSchema defines where the parsed and enriched CDRs are written to.
- ▶ The CDRDupStreamSchema defines where duplicate CDRs detected by the Bloom Filter are written to
- ▶ The CDRErrorStreamSchema defines where CDRs are written to, for which the error indicator flag is set to true.

Now we add the attributes defined in the data model to `tedaConfig.xml`. The entry for the first attribute (`neId`) will appear, as shown in Example 14-4.

Example 14-4 Adding attribute `neId` to `tedaConfig.xml` file

```
...
<attribute name="neId" splType="int32" connectionType="Integer"
sqlType="INTEGER" column="NE_ID" csvFieldNumber="1">
  <belongsToTuple name="ParserOut"/>
  <belongsToTuple name="BusinessRulesOut"/>
  <belongsToTuple name="CDRGoodStreamSchema"/>
  <belongsToTuple name="CDRDupStreamSchema"/>
  <belongsToTuple name="CDRErrorStreamSchema"/>
</attribute>
...
```

There are several properties you can define for each attribute. First, you name the attribute. Then, you define the attribute's data types in the SPL / SQL and connection domain using `splType`, `sqlType`, and `connectionType`. Although the first property is required, the other two are used only for attributes that are output by the application. You define the column of the database table the attribute is written to, by assigning the name to the column attribute.

When you are working with CSV files that are read in through the Java Parser, you identify the attribute's field in the list of comma-separated values through `csvFieldNumber`. We define the optional `description` property to store some information concerning the attribute in the XML configuration file. It does not trigger any functions. With `sqlTypeAddOn`, you can refine the SQL type information. The text you add here is appended to the SQL type and length information in the generated code.

Finally, there is the `altSource` parameter. In nearly all of the cases the Splitter distributing tuples to the aggregation operators will connect input attributes to output attributes with the same name. Rarely a different source is needed and with this parameter you may choose a different input attribute and connect it to the current attribute.

Note: The CSV field number is optional. If case the attribute is not emitted by the parser, or the CSV parser is not used in the application, it can be omitted.

Now, specify into which SPL streams each attribute should be included. The following three predefined internal streams are in TEDA:

- ▶ **ParserOut**
Include this stream name if the attribute is part of the tuple emitted by the parser.
- ▶ **BusinessRulesEnrichment**
Include this name if the attribute is created in the Rules Engine instead of being emitted by the parser.
- ▶ **BusinessRulesOut**
This name is valid only for attributes that are emitted by the parser. It indicates that the attribute is passed through the Rules Engine. If this name is not included, the attribute will be available for evaluation in the Rules Engine, but will not be included in the output. So it is effectively removed from the data stream and will not be available for output or aggregations.

As a guideline, use `ParserOut` and `BusinessRulesOut` for attributes from the parser and `BusinessRulesEnrichment` for attributes created in the Rules Engine.

In addition to the three internal data streams, there are also the three output streams that have been defined in the `outputSchemas` section of the XML file:

- ▶ `CDRGoodStreamSchema`
- ▶ `CDRErrorStreamSchema`
- ▶ `CDRDupStreamSchema`

Add these names to the attribute if you want to have the attribute in the output file or database table. For example, you might configure to have only the `recordId` in the error output, instead of the whole CDR.

For the attributes defined in this use case, the attribute entries in the XML file can be similar to those in Example 14-5.

Example 14-5 Attribute definitions in `tedaConfig.xml` file

```
...
<attribute name="neId" splType="int32" connectionType="Integer"
sqlType="INTEGER" column="NE_ID" csvFieldNumber="1">
  <belongsToTuple name="ParserOut"/>
  <belongsToTuple name="BusinessRulesOut"/>
  <belongsToTuple name="CDRGoodStreamSchema"/>
  <belongsToTuple name="CDRDupStreamSchema"/>
  <belongsToTuple name="CDRErrorStreamSchema"/>
</attribute>
```

```

<attribute name="recordId" splType="int32" connectionType="Integer"
sqlType="INTEGER" column="REC_ID" csvFieldNumber="2">
  <belongsToTuple name="ParserOut"/>
  <belongsToTuple name="BusinessRulesOut"/>
  <belongsToTuple name="CDRGoodStreamSchema"/>
  <belongsToTuple name="CDRDupStreamSchema"/>
  <belongsToTuple name="CDRErrorStreamSchema"/>
</attribute>

<attribute name="callType" splType="int32" connectionType="Integer"
sqlType="INTEGER" column="CALL_TYPE" csvFieldNumber="3">
  <belongsToTuple name="ParserOut"/>
  <belongsToTuple name="BusinessRulesOut"/>
  <belongsToTuple name="CDRGoodStreamSchema"/>
  <belongsToTuple name="CDRDupStreamSchema"/>
</attribute>

<attribute name="callingImsi" splType="rstring" connectionType="String"
csvFieldNumber="4" sqlType="VARCHAR" column="CLG_IMSI">
  <restriction name="length" value="15"/>
  <belongsToTuple name="ParserOut"/>
  <belongsToTuple name="BusinessRulesOut"/>
  <belongsToTuple name="CDRGoodStreamSchema"/>
  <belongsToTuple name="CDRDupStreamSchema"/>
</attribute>

<attribute name="callingMsisdn" splType="rstring" connectionType="String"
sqlType="VARCHAR" column="CLG_MSISDN" csvFieldNumber="5">
  <restriction name="length" value="15"/>
  <belongsToTuple name="ParserOut"/>
  <belongsToTuple name="BusinessRulesOut"/>
  <belongsToTuple name="CDRGoodStreamSchema"/>
  <belongsToTuple name="CDRDupStreamSchema"/>
</attribute>

<attribute name="calledImsi" splType="rstring" connectionType="String"
sqlType="VARCHAR" column="CLD_IMSI" csvFieldNumber="6">
  <restriction name="length" value="15"/>
  <belongsToTuple name="ParserOut"/>
  <belongsToTuple name="BusinessRulesOut"/>
  <belongsToTuple name="CDRGoodStreamSchema"/>
  <belongsToTuple name="CDRDupStreamSchema"/>
</attribute>

<attribute name="calledMsisdn" splType="rstring" connectionType="String"
sqlType="VARCHAR" column="CLD_MSISDN" csvFieldNumber="7">
  <restriction name="length" value="15"/>
  <belongsToTuple name="ParserOut"/>
  <belongsToTuple name="BusinessRulesOut"/>

```

```

        <belongsToTuple name="CDRGoodStreamSchema"/>
        <belongsToTuple name="CDRDupStreamSchema"/>
    </attribute>

    <attribute name="callStartTime" splType="rstring" connectionType="String"
    sqlType="VARCHAR" column="CALLSTART" csvFieldNumber="8">
        <restriction name="length" value="32"/>
        <belongsToTuple name="ParserOut"/>
        <belongsToTuple name="BusinessRulesOut"/>
        <belongsToTuple name="CDRGoodStreamSchema"/>
        <belongsToTuple name="CDRDupStreamSchema"/>
    </attribute>

    <attribute name="callEndTime" splType="rstring" connectionType="String"
    sqlType="VARCHAR" column="CALLEND" csvFieldNumber="9">
        <restriction name="length" value="32"/>
        <belongsToTuple name="ParserOut"/>
        <belongsToTuple name="BusinessRulesOut"/>
        <belongsToTuple name="CDRGoodStreamSchema"/>
        <belongsToTuple name="CDRDupStreamSchema"/>
    </attribute>

    <attribute name="releaseCause" splType="int32" connectionType="Integer"
    sqlType="INTEGER" column="TERM_CODE" csvFieldNumber="10">
        <belongsToTuple name="ParserOut"/>
        <belongsToTuple name="BusinessRulesOut"/>
        <belongsToTuple name="CDRGoodStreamSchema"/>
        <belongsToTuple name="CDRDupStreamSchema"/>
    </attribute>

    <!-- enriched attributes -->

    <attribute name="callDuration" splType="int32" connectionType="Integer"
    sqlType="INTEGER" column="CALLDURATION">
        <belongsToTuple name="BusinessRulesEnrichments"/>
        <belongsToTuple name="CDRGoodStreamSchema"/>
        <belongsToTuple name="CDRDupStreamSchema"/>
    </attribute>

    <!-- required for bloomfilter -->
    <attribute name="CallStartDateTime" splType="rstring" connectionType="String"
    sqlType="TIMESTAMP" column="CALL_REFERENCE_TIME">
        <restriction name="length" value="19"/>
        <belongsToTuple name="BusinessRulesEnrichments"/>
        <belongsToTuple name="CDRGoodStreamSchema"/>
        <belongsToTuple name="CDRDupStreamSchema"/>
    </attribute>

    <!-- required for bloomfilter -->

```

```
<attribute name="CDRIdKey" splType="rstring" connectionType="String"
sqlType="VARCHAR" sqlTypeAddOn="FOR BIT DATA" column="CDR_ID_KEY">
  <restriction name="length" value="28"/>
  <belongsToTuple name="BusinessRulesEnrichments"/>
  <belongsToTuple name="CDRGoodStreamSchema"/>
  <belongsToTuple name="CDRDupStreamSchema"/>
</attribute>
```

...

After saving your changes to the `tedaConfig.xml` file, run the code generator tool to modify the TEDA source code. Enter the following commands to clean the application and invoke the generator:

```
cd ~/TEDA/demo2/application
make clean
cd scripts
./generateTeda.pl
```

If the script detects errors in the XML configuration file, it prints error messages that can help you in resolving the issue. Next, we need to adapt the rules file to create the three attributes: `callDuration`, `callStartDateTime`, and `CDRIdKey`.

Adapting the rules file

The rules file (`rules.br`) resides in the `TEDA/demo2/application/rules` directory. You may edit it with any text editor. Regarding the general structure of the rules file, the file starts with this line:

```
RULESET RulesEngine;
```

This specifies the name of the SPL composite operator, which is created by the Rules Compiler for the `rules.br` file. The resulting SPL file (`RulesEngine.spl`) is generated in the `application/apps/main` directory.

Note: If you want to implement the business logic directly in SPL, you can remove the `rules.br` file. After that, the Rules Compiler will not generate the output file so you must supply the SPL file for the implementation.

The next section defines macros, which are used in the rules later. Basically you can define any valid SPL expression or function call in a macro, as shown in Example 14-6.

Example 14-6 Functions section in the rules file

```
FUNCTIONS {  
...  
}
```

The next two sections, INPUT and OUTPUT determine the tuple attributes entering and leaving the Rules Engine. Do not modify these sections because they are replaced when you invoke the **generateTeda.pl** tool.

After the OUTPUT section, the rule definitions start. There is one default rule, which should be used to set up default values for attributes that are created in the Rules Engine, for example attributes that are marked as belonging to the BusinessRulesEnrichments tuple in the tedaConfig.xml file. For our sample case, locate the DEFAULTS rule and adapt it as shown in Example 14-7.

Example 14-7 DEFAULTS rule in the rules file

```
RULE DEFAULTS {  
=>  
[CallStartTime] = "2011-05-21-12.00.00.000";  
[callDuration] = 12;  
[errorInd] = 0;  
[CDRIdKey] = "";  
}
```

This defines default values for the attributes calculated by the rules. The attribute errorInd is automatically generated by TEDA, so you do not need to configure it in tedaConfig.xml. It is used to indicate that a tuple (CDR) is erroneous. In that case, the tuple will be directed to the output split named CDR_ERR. Now, add two rules to calculate the time stamp and the hash attributes for the Bloom Filter. Remove all existing rules after the default rule, then add the two rules, TR0 and TR1, as shown in Example 14-8 on page 498.

Note: You do not need to add these two rules, if you do not use the Bloom Filter by setting BYPASS_BLOOMFILTER=1 in file config.cfg.

Example 14-8 Rules to calculate timestamp and hash values

```
RULE TR0 {
VAR A = toString([recordId])
VAR B = toString([neId])
VAR C = concat(A, B)
VAR D = sha2hash(C)
=>
[CDRIdKey] = D;
}

RULE TR1 {
VAR X = converttime([callStartTime])
=>
[CallStartDateTime] = X;
}
```

The first rule concatenates the recordId and neId attributes, and then calculates a SHA2 hash value from the resulting string. This hash is used later in the Bloom Filter to uniquely identify a given CDR. You may use any of the input attributes that might seem fit to calculate the hash value; for example the calling and called numbers can be included in the calculation too.

The second rule calculates a time stamp from the given input attribute. You probably need to adapt this rule, based on the format of your actual input attribute.

Finally, after saving the rules file, make small modifications in the main configuration file.

Adapting the configuration file

The configuration file (config.cfg) resides in the TEDA/demo2/application/config directory. Most parts of this configuration file are automatically modified by the **generateTeda.pl** tool based on the content of the tedaConfig.xml file. However, some of the entries still need manual modification. The parts that are automatically generated are enclosed in comments, advising you not to modify the corresponding part, because it will be overwritten the next time you invoke **generateTeda.pl**.

For our sample case, alter the following settings. Locate them in the config.cfg file and change the lines accordingly:

```
USE_JAVA_DECODER=1
STREAMS_IID=demo2
COGNOS_DBSHEMA=
BLOOM_PROBABILITY=0.001
```

The first parameter configures the CSV decoder for parsing the input files. The `STREAMS_IID` entry lets you specify the name of the Streams instance, which is created when starting up the application. Because we do not use any aggregation operators in this exercise, and thus do not use a database for aggregation results, we set the `COGNOS_DBSCHEMA` to be empty. Finally, we set the probability for false duplicates detected by the Bloom Filter to a lower value than the default value.

Building the sample application

Before you compile the application, the changes made until now are summarized in the following list:

- ▶ We configured output streams and attributes in `tedaConfig.xml` and invoked the `generateTeda.pl` tool to adapt the code automatically.
- ▶ Then, we adapted the `rules.br` file to set defaults for attributes created in the Rules Engine and implemented the two rules for the attributes we need in the Bloom Filter.
- ▶ Finally, we modified a few configuration parameters in `config.cfg`, mainly to inform TEDA to use the Java CSV parser.

To build the application, use the following commands:

```
cd ~/TEDA/demo2/application
make clean
make all
```

The `makefile` in the application directory automatically invokes the Rules Compiler to generate the `RulesEngine.sp1` file from the `rules.br` file. However, it will not invoke the `generateTeda.pl` tool to adapt source code based on the `tedaConfig.xml` file. You must do that manually after changing `tedaConfig.xml`.

Tip: After changing either the `rules.br` or the `config.cfg` file, it is sufficient to recompile TEDA using the `makefile` in `~/TEDA/demo/application`. But after changing `tedaConfig.xml`, you need to regenerate and recompile, using the following commands:

```
cd ~/TEDA/demo2/application
make clean
cd scripts
./generateTeda.pl
cd ..
make all
```

Running the sample application

If the sample application compiles successfully, you can start TEDA to process the input file you created. Use the following commands to clean the status tables and start the application:

```
cd TEDA/demo2/application/scripts
./cleanMetaDB -y
./startup.pl --retry=0 --rollset=1 --verbose=3
```

Attention: The `cleanMetaDB` command removes any information about the application status and the files already processed from the meta database tables. For testing purposes, run this command before each restart of TEDA to start from a clean state and have a well defined environment for your test. Do not use this script in a production environment unless you really want to get rid of your status data.

To monitor the progress of the startup and the runtime health checking, enter this command:

```
./startup.pl --tail
```

Because we gave the `--verbose=3` option, you see much information in the console now. When the startup process finishes, you will see a line similar to the following message in the traces:

```
waiting for job completion
```

This means the application successfully started. The Master Script periodically monitors the health of TEDA, by invoking the `streamtool` command to check the PE status.

After some time (60 - 120 seconds, the default commit interval), you see the processed CDR records in TEDA's output directory. The default output directory is `~/TEDA/testdata/output`. Check the content of that directory now. You will find several files in it. Each defined output split will write its own set of files. The file names start with the number of the output split, as in the following examples:

- ▶ `0_*` for good CDRs
- ▶ `1_*` for duplicate CDRs
- ▶ `2_*` for error CDRs

After the split identifier and the underscore, the next digits contain the UNIX time stamp for the creation of this file. Finally, the group number (also called *circle* or *region*) is given after the dash. Because we have configured only the three default output splits and only one group, there should be six files similar to what is shown in Example 14-9 on page 501. The time stamp part of the file name will be different on your system, of course.

Example 14-9 Sample output files

```
0_1369158441-0.dat
0_1369158525-0.dat
1_1369158441-0.dat
1_1369158525-0.dat
2_1369158441-0.dat
2_1369158525-0.dat
```

The three files with the more recent timestamps are empty because they are created for the CDRs that will arrive next. The other three contain the CDRs from our input file. When you check the contents of the files, you will find six CDRs in the 0_* file (the good CDRs) and three CDRs in the 1_* file. The latter are the duplicate CDRs (remember, we used the `neld` and the `recordId` attributes only to calculate the Bloom Filter hash, and the input file contained three CDRs with identical combinations of `neld` and `recordId`). The 2_* file is empty, because we did not mark any CDR as faulty, using the `errInd` attribute in the Rules Engine.

Finally, stop TEDA with the following commands, then continue with the next exercises where you add more rules, database access and an aggregation operator.

```
cd ~/TEDA/demo2/application/scripts
./startup.pl -stop
```

14.4.5 Exercise 2: Writing to the database

In this exercise, you make simple changes to the XML configuration file to redirect the output into a database.

Changing the XML configuration file

Change directory to `~/TEDA/demo2/application/config` and open the file `tedaConfig.xml` for modification. Find the `outputSchemas` section and change the `mode` attribute of every `outputSchema` from `FILE` to `SERIAL` as shown in Example 14-10.

Example 14-10 Switching output to database through `tedaConfig.xml`

```
...
<outputSchemas>
  <outputSchema name="CDRGoodStreamSchema" type="good"
    splitName="CDR_GOOD" database="STAGING" dbTableName="CDR_GOOD"
    mode="SERIAL" perGroup="true"/>
  <outputSchema name="CDRDupStreamSchema" type="duplicate"
    splitName="CDR_DUP" database="STAGING" dbTableName="CDR_DUP"
```

```
        mode="SERIAL"/>
    <outputSchema name="CDRErrorStreamSchema" type="error"
        splitName="CDR_ERR" database="STAGING" dbTableName="CDR_ERR"
        mode="SERIAL"/>
</outputSchemas>
...
```

Do not modify the rules file because there is no need for more business logic yet.

Building the sample application

After saving your changes to the `tedaConfig.xml` file, you clean the application, invoke the generator and build the application by using the following commands:

```
cd ~/TEDA/demo2/application
make clean
cd scripts
./generateTeda.pl
cd ..
make all
```

Create database tables

The `generateTeda.pl` tool uses the data in `tedaConfig.xml` file to create Data Definition Language (DDL) files that define the table content for the different output schemas. Use the `cdrTables.ddl` file to create the CDR tables necessary for the application by using the following commands:

Note: You create the tables using this procedure only once after you add or remove at least one attribute to any output schema.

```
cd ~/TEDA/demo/application/ddl
cp cdrTables.ddl /tmp
chmod 777 /tmp/*ddl
su - db2inst1
db2start
db2 connect to TTEST user db2inst1 using ibm2blue
db2 -td\; -vf /tmp/cdrTables.ddl
db2 terminate
```

Attention: The DDL files have a "drop table" line before each "create table" so stored data will be lost if you use the files unmodified. If that is not what you want, modify the DDL files.

Running the sample application

You use the same input data file you created for 14.4.4, “Exercise 1: Basic setup” on page 488.

1. To start consecutive test runs from a well defined starting point, remove records from the CDR tables by using the following commands:

```
db2 connect to TTEST user db2inst1 using ibm2blue
db2 'delete from CDRADM.CDR_GOOD_RGN_0_1'
db2 'delete from CDRADM.CDR_DUP_1'
db2 'delete from CDRADM.CDR_ERR_1'
```

2. For the same reason, remove existing files from the output directory and clean the status database by using the following commands:

```
cd ~/TEDA/demo2/application/testdata/output
rm *
cd ../../scripts
./cleanMetaDB.sh -y
```

3. Start the application by using the startup.pl Master Script in the scripts directory:

```
./startup.pl --retry=0 --rollset=1 --verbose=3
```

4. Monitor the progress of the startup:

```
./startup.pl --tail
```

5. Wait for about two minutes after the message “waiting for job completion” appears in the traces, then check the number of records in the different CDR tables with the following commands:

```
db2 'select count(*) from CDRADM.CDR_GOOD_RGN_0_1'
db2 'select count(*) from CDRADM.CDR_DUP_1'
db2 'select count(*) from CDRADM.CDR_ERR_1'
```

There are six entries in the CDR_GOOD table, three entries in the CDR_DUP table, and no entries in the CDR_ERR table.

6. Finally, stop TEDA with the following commands:

```
cd ~/TEDA/demo2/application/scripts
./startup.pl --stop
```

14.4.6 Exercise 3: Adding an aggregation operator

In this exercise, you add an aggregation operator to your TEDA application. A rule is added that generates a random call duration. This call duration is accumulated in the newly added aggregation operator.

Changing the XML configuration file

Complete the following steps:

1. Open the `tedaConfig.xml` file (in `~/TEDA/demo2/application/config`) and find the databases section.
2. Create a new database element with the ID `COGNOS`, as shown in Example 14-11.

Example 14-11 New database entry

```
...
<databases>
  <database id="STAGING" name="TTEST" schema="CDRADM" user="db2inst1"
    password="ibm2blue" partitions="0" partitionMappingFile="TTEST"
    splitPartitionMappingFile="TTEST"/>
  <database id="COGNOS" name="ISS" schema="COGNOS_DEMO"
    user="db2inst1" password="ibm2blue" partitions="0"
    partitionMappingFile="ISS" splitPartitionMappingFile="ISS"/>
</databases>
...
```

3. Add the first aggregation operator to the operator mapping section and assign it the `CallDurationAggregator` operator, as shown in Example 14-12.

Example 14-12 Aggregator to operator mapping

```
...
<aggregatorToOperator>
  <aggregator name="Aggregator001" operator="CallDurationAggregator"/>
</aggregatorToOperator>
...
```

4. Add the Aggregator001Out output schema to the outputSchemas section as Example 14-13 shows. It will consume good CDRs and write into the CALL_DURATIONS table of the COGNOS database.

Example 14-13 Aggregator001Out output schema added

```
...
<outputSchemas>
  <outputSchema name="CDRGoodStreamSchema" type="good"
    splitName="CDR_GOOD" database="STAGING" dbTableName="CDR_GOOD"
    mode="SERIAL" perGroup="true"/>
  <outputSchema name="CDRDupStreamSchema" type="duplicate"
    splitName="CDR_DUP" database="STAGING" dbTableName="CDR_DUP"
    mode="SERIAL"/>
  <outputSchema name="CDRErrorStreamSchema" type="error"
    splitName="CDR_ERR" database="STAGING" dbTableName="CDR_ERR"
    mode="SERIAL"/>
  <outputSchema name="Aggregator001Out" type="good"
    splitName="CALL_DURA" database="COGNOS"
    dbTableName="CALL_DURATIONS" mode="SERIAL"/>
</outputSchemas>
...
```

5. Add the attributes to the different tuple streams. The callDuration attribute is produced by the business logic in the Rules Engine and is put out for good and duplicate CDRs. Also, it becomes an input of the new aggregation operator. Make the CallStartDateTime attribute part of the aggregation operator's input and output schemas, and create a new averageDuration attribute for the output of the aggregation operator. All modifications are shown in Example 14-14.

Example 14-14 New and modified attribute elements

```
...
<attribute name="callDuration" splType="int32"
  connectionType="Integer" sqlType="INTEGER" column="CALLDURATION">
  <belongsToTuple name="CDRGoodStreamSchema"/>
  <belongsToTuple name="CDRDupStreamSchema"/>
  <belongsToTuple name="BusinessRulesEnrichments"/>
  <belongsToTuple name="Aggregator001"/>
</attribute>

...

<attribute name="averageDuration" splType="int64"
  connectionType="Integer" sqlType="INTEGER" column="AVGCCALLDURATION">
  <belongsToTuple name="Aggregator001Out"/>
</attribute>
```

```

...
<!-- required for bloomfilter -->
<attribute name="CallStartDateTime" splType="rstring"
  connectionType="String" sqlType="TIMESTAMP"
  column="CALL_REFERENCE_TIME">
  <restriction name="length" value="19"/>
  <belongsToTuple name="BusinessRulesEnrichments"/>
  <belongsToTuple name="CDRGoodStreamSchema"/>
  <belongsToTuple name="CDRDupStreamSchema"/>
  <belongsToTuple name="Aggregator001"/>
  <belongsToTuple name="Aggregator001Out"/>
</attribute>
...

```

Adapting the rules file

Open the `~/TEDA/demo2/application/rules/rules.br` file and add a rule named TR2 that generates random numbers for the `callDuration` attribute. The code for the rule is shown in Example 14-15.

Example 14-15 Random call duration rule

```

...
RULE TR2 {
VAR A = getRandom(10,600)
=>
[callDuration] = A;
}
...

```

Adapting the configuration file

Open the `~/TEDA/demo2/application/config/config.cfg` file and remove the following line that we added in 14.4.4, “Exercise 1: Basic setup” on page 488.

```
COGNOS_DBSHEMA=
```

Adding the operator code

Create the `CallDurationAggregator.spl` file in the following directory:

`~/TEDA/demo2/application/apps/main`

The code for the operator composite is in Example 14-16.

Example 14-16 CallDurationAggregator.spl code

```
use com.ibm.streams.tma::*;

public composite CallDurationAggregator (input inStream; output
outStream)
{
    param
        expression<rstring> $callDurationAttribute;
        expression<rstring> $callStartDateTimeAttribute;

    type
        OutputTupleType = Aggregator001Out;

        CallCount = tuple<
            uint64 sumSeconds,
            uint64 numCalls
        >;

    graph

    stream<OutputTupleType> outStream as O = Custom(inStream as I)
    {
        logic
        state :
        {
            mutable tuple<O> o;
            mutable map<rstring, CallCount> callData;
            mutable rstring timeMinutes;
        }
        onTuple I :
        {
            // truncate seconds from the timestamp
            timeMinutes = substring(I.CallStartDateTime, 0,
length(I.CallStartDateTime) - 6) + "00.000000";

            // check if we need to insert the timeslot into the map
            if (!(timeMinutes in callData))
            {
```


Create database tables

The `generateTeda.pl` tool creates a DDL file for the database tables needed by the aggregation operator too. Use this file to create the aggregation results table by using the following commands:

```
cd ~/TEDA/demo2/application/ddl
db2 connect to ISS user db2inst1 using ibm2blue
db2 -td\; -vf aggregatorTables.ddl
db2 terminate
```

Running the sample application

Remove existing output files, clean the status and production database tables, and the application by using the following commands:

```
cd ~/TEDA/demo2/application/testdata/output
rm *
cd ../../scripts
./cleanMetaDB.sh -y
./cleanProdDB -y
cd ..
make clean
```

Start the TEDA application using the Master Script `startup.pl` in the scripts directory:

```
./startup.pl --retry=0 --rollset=1 --verbose=3
```

Check the aggregation operator table for results by using these commands:

```
db2 connect to ISS user db2inst1 using ibm2blue
db2 'select * from COGNOS_DEMO.CALL_DURATIONS'
```

Depending on the test data, the result is similar to the table shown in Example 14-17. The values for the average call duration can vary because the random call duration numbers and row numbers will also vary.

Example 14-17 Aggregation operator database content

AVGCCALLDURATION	CALL_REFERENCE_TIME	ROW_NUMBER
449	2013-01-01-16.13.00.000000	21
281	2013-01-01-16.12.00.000000	22
170	2013-01-01-16.11.00.000000	23
243	2013-01-01-16.10.00.000000	24

14.5 Conclusion

In this chapter, we introduce the Accelerator for Telecommunications Event Data Analytics (TEDA) Version 1.2. We direct you to the topics in the information center that guide you through the accelerator's installation procedure. You know how the deduplication and recovery mechanisms in TEDA work, and that you simply need to configure, not implement, them in your own use cases. If you work through the exercises, you learn to adapt TEDA to various use cases by changing the processed data streams, defining your business logic in the rules format, and creating aggregation operators to gather more insight into your data.

You are now able to rapidly create your own prototypes for CSV data formats. If you need to process binary input data defined in ASN.1 format, you know to follow the instructions in the "Implementing a parser" topic in the information center.

Altogether, TEDA provides you a head start for implementing your own telecommunications applications for data in motion.



SPSS Toolkit

In this chapter, we describe how to integrate IBM SPSS Modeler predictive analytics into InfoSphere Streams applications. We briefly cover predictive modeling topics from training the predictive models, through predictive scoring branch design, to publishing and refreshing the models. This introduction provides sufficient background in the activities of the data analyst to understand the important requirements, design, and integration coordination topics to be discussed with the Streams application development team.

The primary audience for this chapter is the Streams application developers based on their interactions with the SPSS Modeler data analyst. The main focus is on the prepared model as configured for use in the SPSS Analytics Toolkit for InfoSphere Streams, and also how the predictive model can be refreshed without interrupting the flow of the deployed Streams application.

A prerequisite for this chapter is Streams Programming Language (SPL) skills. To work with the examples in this chapter, you should have a general familiarity with defining and deploying an InfoSphere Streams application.

To run the examples, you need a Red Hat Enterprise Linux system with InfoSphere Streams V2.0 or later, and IBM SPSS Modeler Solution Publisher 15.0 Fix Pack 1 or later installed on it. The IBM Modeler Solution Publisher installation contains the IBM SPSS Analytics Toolkit for InfoSphere Streams package.

15.1 An overview of InfoSphere Streams and SPSS

In this section, we cover some of the basics of InfoSphere Streams and SPSS to better enable you to work with them together.

15.1.1 Integrating InfoSphere Streams and SPSS

IBM SPSS Modeler provides a state-of-the-art environment for understanding data and producing predictive models. InfoSphere Streams provides a scalable high-performance environment for real-time analysis of data in motion, including traditional structured or semi-structured data, to unstructured data types. Some applications have a need for deep analytics derived from historic information to be used to score streaming data in low-latency, high-volume, and real time, and to leverage those analytics. The SPSS Analytics Toolkit for InfoSphere Streams lets you integrate the predictive models designed and trained in IBM SPSS Modeler with your IBM InfoSphere Streams applications.

15.1.2 Roles and terminology

First, we describe a few roles and their responsibilities, and present some of the terminology used throughout the chapter.

Roles

The primary roles of interest are as follows:

- ▶ **Data analyst:** A modeling expert who knows how to use the IBM SPSS Modeler tools to build and evaluate predictive models and to design and publish scoring branches for InfoSphere Streams integration.
- ▶ **Streams application developer:** An InfoSphere Streams developer responsible for building applications and configuring the operators in the SPSS toolkit.

Terminology

The following terminology is used throughout this chapter:

- ▶ **Predictive model:** We use this term or the term model as a reference to the prepared scoring branch of an SPSS Modeler analytics design. The scoring branch itself may contain predictive model nuggets (trained instances of a specific predictive model algorithm) and also other processing required to generate the desired analytics.
- ▶ **Streams Processing Language (SPL):** This is the language used to write InfoSphere Streams applications.

- ▶ **Operators:** The basic building-blocks of InfoSphere Streams applications. The standard operator set is included in the Streams product. There are many toolkits that can be installed and clients can write their own custom operators. In this chapter, the focus is on the IBM SPSS Analytics Toolkit for InfoSphere Streams.

Note: There is potential for confusion with overloading of the term streams. The InfoSphere Streams product refers to streams of data and streams applications built using SPL. The SPSS Modeler product creates a workflow of connected modeler components (process nodes), documented in the product literature as a stream describing the data flowing from the source nodes through the process nodes to the terminal nodes. For the purpose of this chapter, the SPSS Modeler *streams* are referred to as predictive models or models as noted previously (focus on scoring); and the term *stream* will mean an InfoSphere Streams data stream.

15.1.3 Example development process

In this section, we describe an overall application development flow that starts with a focus on the predictive model development process and ends with the Streams application development process:

1. A data analyst determines what input attributes will be required for the predictive analytics that have been defined to be of interest in a Streams application.
2. A Streams application developer and the data analyst work together to determine the data quality and latency requirements for the predictive analytics data flow in the proposed Streams application.
3. A Streams application developer builds the application that obtains the attributes, calls the scoring operator, and takes action based on the resulting scores.

In practice, this typically is an iterative process, starting with discussions of what attributes are needed from all of those available in the planned Streams flow and leading to questions about what predictive analytics can be generated and how they might be used by the application.

For an existing Streams application, the available inputs are known but the data quality and other requirements of the predictive analytics might require some changes to its design. For example, the required analytics might have a higher confidence or be able to use a more efficient model algorithm if the data flows contained certain additional attributes.

In the following sections, we work through a sample scenario to illustrate this process for a new Streams application with a predictive analytics focus.

15.2 Coordinating Data Analyst and Streams developer efforts

In this section, we describe the coordination of the efforts of the data analyst and the Streams application developer. This coordination addresses the following information:

- ▶ Input and output data models for the scoring operator
- ▶ Assets required to enable scoring in a Streams application
- ▶ Latency requirements for the scoring operator
- ▶ Predictive model refresh plan for how the assets will be *refreshed* for use in the Streams application

In 15.3, “Building the predictive models” on page 515, we describe the Input and Output data model contract and also the asset generation required to enable scoring in a Streams application.

Latency requirements

Expectations on latency should be stated before the predictive modeling begins, because the latency requirements have a major influence on predictive modeling design choices. The starting point is the overall processing time for a tuple flowing through the Streams application. Any general latency for Streams applications of a pattern can be used to determine the latency window available for the generation of the predictive analytics. This window of time will probably require processing related to data quality requirements and any additional data required to be sourced for the one or more prediction, and so will have to wait until the initial scoring plan has been designed.

Any scoring plan designed for a given Streams application should be tested for latency. To this end, the data analyst should provide test-run data on a sufficiently large sample data set and note the batch scores-per-second performance on these runs. As the designs for the generation of the required predictive analytics finalize, the data models, data quality, and latency requirements will also finalize.

Predictive model refresh plan

Training predictive models by using data mining techniques can provide accurate, high confidence predictive analytics when applied by a data analyst. An important note is that the historic data pertinent to training these predictive models is always changing and causing a slow drift in the accuracy and confidence of the predictions. The scoring branches designed for the Streams applications will be evaluated periodically by the data analyst for accuracy. The plan for refreshing the scoring branch used by the Streams application is part of the application design. This plan include performance tests of the refreshed branch and also the honoring of the input and output data contract of the configured SPSSScoring operator. We describe this in more detail in 15.4, “Configuring the SPSSScoring operator” on page 521.

15.3 Building the predictive models

Executing a model *build* process causes the model algorithm to apply data mining techniques to train a predictive model represented by the generated yellow “model nugget” as shown in Figure 15-1 on page 516. Every model algorithm has the ability to be adjusted for the current problem and this tuning involves evaluation of the trained model against data reserved for this purpose from the historic data set. We discuss the data discovery and also the predictive model build and evaluate activities in this section.

Determining what predictors are required, what model algorithms are appropriate and also training and evaluating the predictive models and designing the scoring branch are all activities typically done by a data analyst.

We use an example Streams application to illustrate this process. In this specific example, the requirement is to predict whether a customer visiting the corporate website will “churn” or not (change to a different wireless provider). We briefly introduce the process of training and evaluating the predictive model and using it in a simple scoring branch before publishing the design to permit its use in the InfoSphere Streams application of this example.

To start this activity, the data analyst will review the historical data available for customers who did and did not churn, over the last six months.

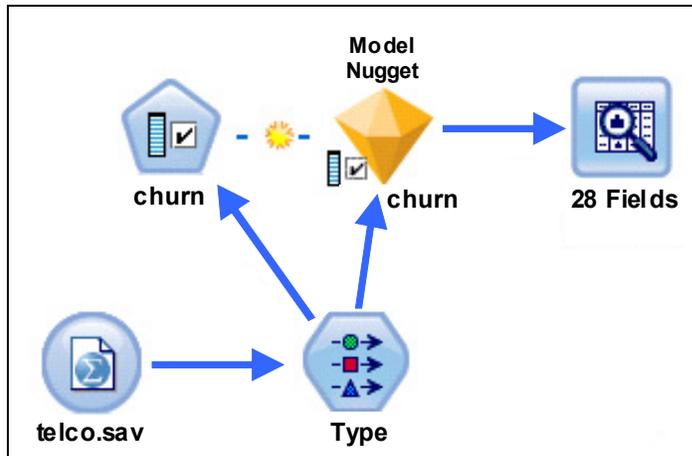


Figure 15-1 An example model-1

This example SPSS Modeler process investigates the pertinent history data using a Feature Selection model algorithm (labeled “churn” in Figure 15-1) that trains a predictive model instance, which is then used by a Data Audit process (labeled "28 Fields" in Figure 15-1) to analyze the historical data. That data audit detail is depicted in Figure 15-2.

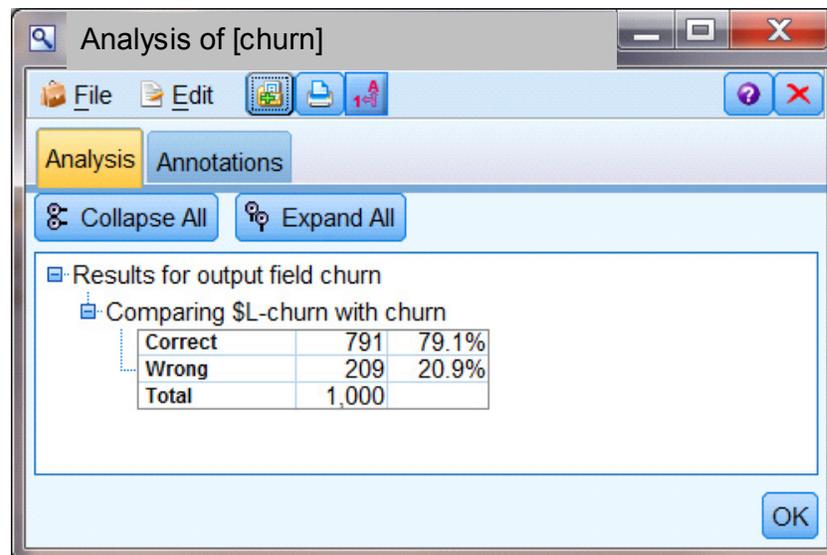


Figure 15-2 Data audit

At this point, the Streams application designer and the data analyst can begin their discussions about what data will be required to provide the predictors that are required for this application, as shown in Figure 15-3. Only predictors that can be sourced with the required degree of quality by the Streams application will be used for modeling.

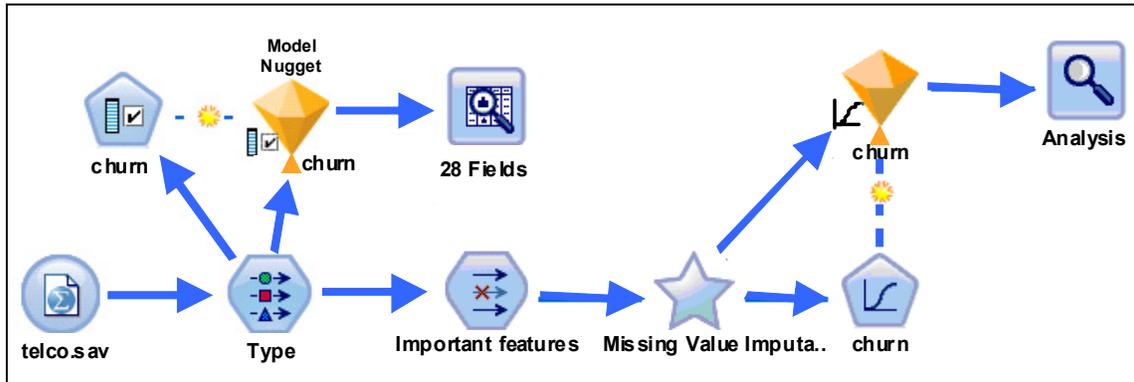


Figure 15-3 An example model-2

After a series of predictive model “builds,” the data analyst defines a plan to train a logistic model, labeled “churn” on the right of Figure 15-3. That evaluates to be of the required level of predictive confidence for use in the Streams application, as depicted in Figure 15-4.

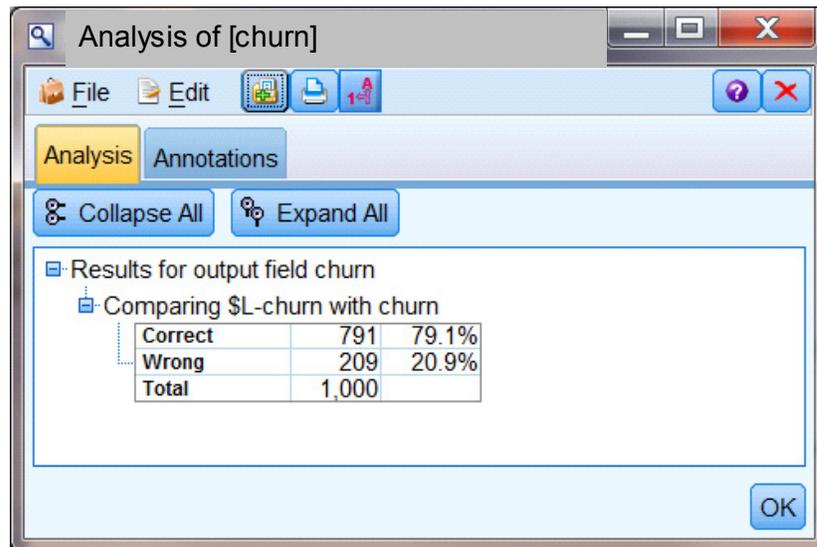


Figure 15-4 Analysis of churn

An important consideration is that the accuracy of the predictive model will change over time, requiring a “refresh” from time to time, shown in Figure 15-5. We describe the concept of refreshing the scoring branch later in this chapter.

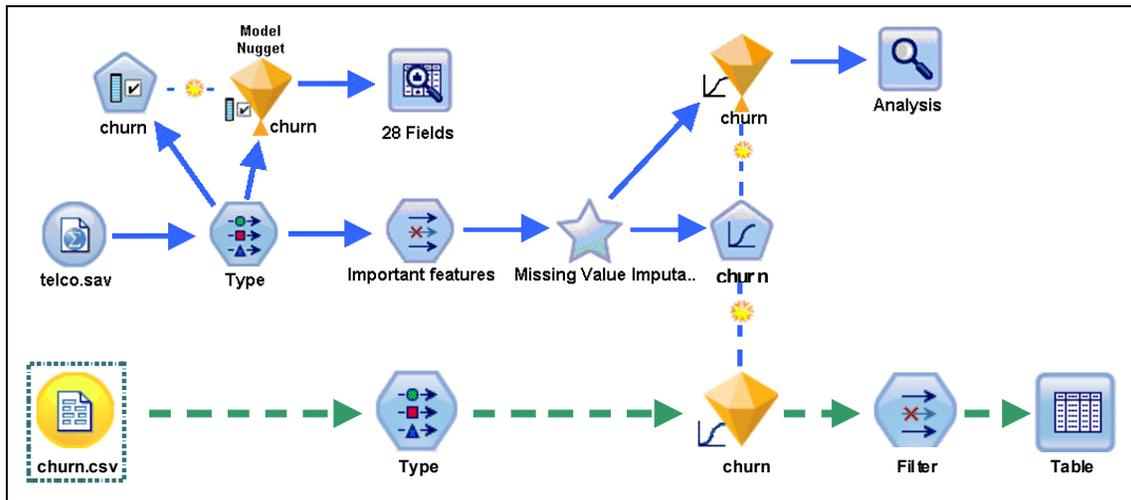


Figure 15-5 Analysis over time

The final step in the initial design of the predictive model for the example Streams application is to define the scoring branch to be configured for use by an instance of the SPSS Scoring operator in the SPSS Analytics Toolkit for InfoSphere Streams. We see a statistical sampling of input data that conforms to the required input data model being used in the tests of the scoring branch in the design. This is an important validation of the scoring branch design and a good way to measure the performance in batch processing with SPSS Modeler Client.

For a Streams application, the scoring branch usually has little “data prep” processing. Instead, this is all done by the Streams developer as part of the data quality requirements defined for the application. What remains in the scoring branch is the process flow unique to the predictive model (or models) being used to produce the required analytics. Notice that a filter node is used immediately before the terminal node in this scoring branch to apply an alias to the required predictive analytics generated. This practice avoids relying on the default attribute naming of a given model algorithm.

The act of “publishing” this scoring branch generates the executable image file (.pim extension), the initial parameters (.par extension), and, by selecting the metadata option, the XML file used to configure and verify the operator configuration.

The listings in Figure 15-6 illustrate the data models for the source and terminal nodes of the example scoring branch, as recorded in the published XML file. After configured in an instance of the SPSSScoring operator, these data models define the “contract” between the data analyst and the Streams application developer. Any modifications required during a predictive-model refresh (such as changing to a different model algorithm) can be implemented by the data analyst while this data contract is honored.

The data preparation processing is implemented in the Streams application, which means the SPSSScoring operator needs to accept only one data source in its scoring branch. The published XML file for this scoring branch lists the fields that must be sourced by the Streams application to provide the predictors required to score with this design:

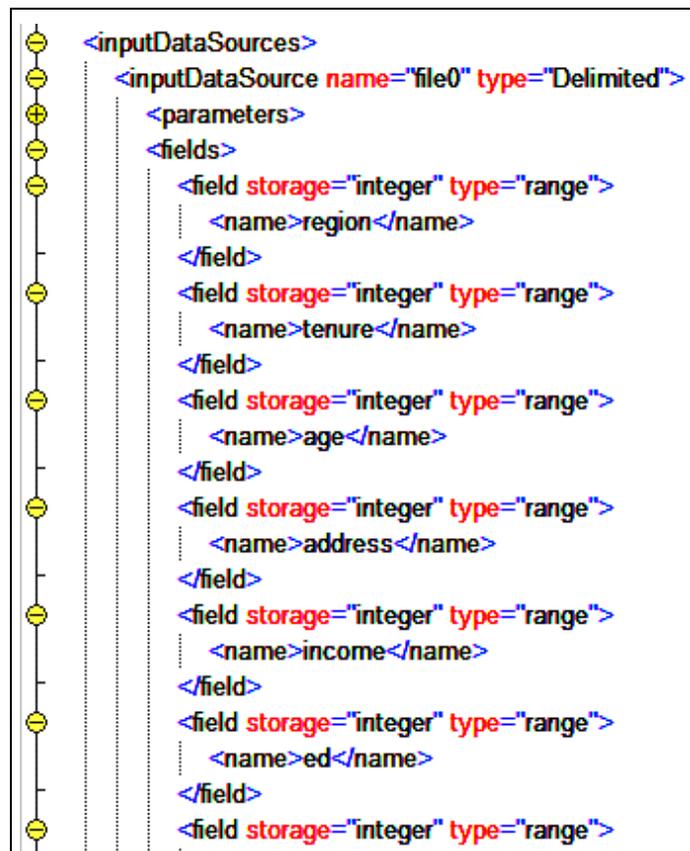


Figure 15-6 Data models

The outputs from the scoring branch can be filtered, but the normal practice is to return all inputs, and also the generated predictive analytics, in the output data model, as depicted in Figure 15-7.

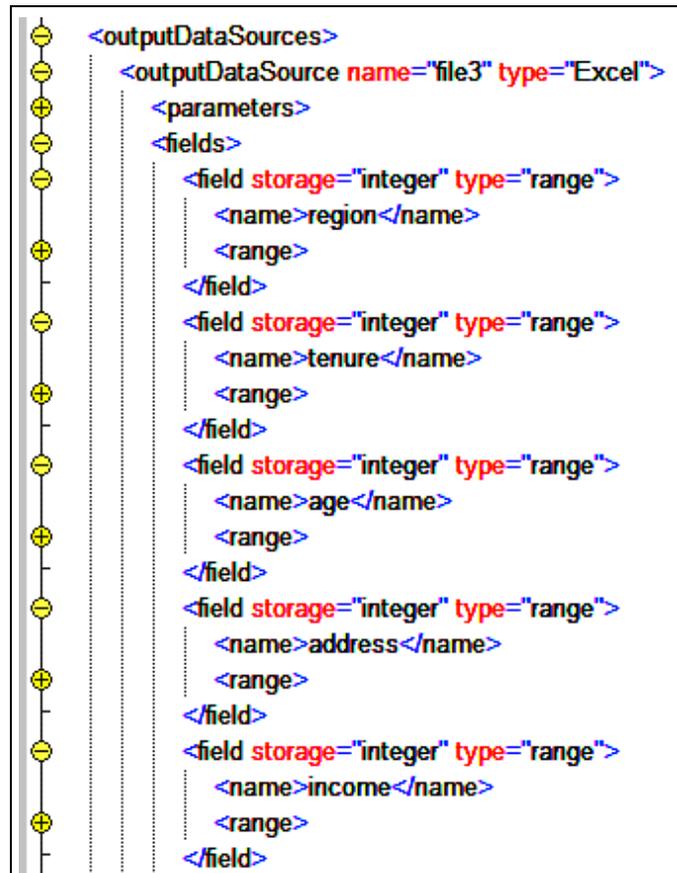


Figure 15-7 Output data model

Notice that we selected the **Use as Scoring Branch** option on the output table terminal node added to test our scoring plan, causing the SPSS Modeler Client interface to highlight the flow through the nodes involved with green dashed arrows, as shown in Figure 15-5 on page 518. This practice avoids having to note the terminal node's ID when configuring the scoring operator.

15.4 Configuring the SPSSScoring operator

Integrating SPSS predictive analytics into a Streams application is typically done by a Streams application developer. In this section, we continue to use the example Streams application and predictive model we previously used.

The example predictive model was based on several details about the customer and the customer's telecommunications service usage. We describe this input data model in our SPL composite with a simple type definition that matches the input data source signature of our published scoring branch, translating the SPSS Modeler storage data types to the following list of equivalent Streams types:

```
type
  static DataSchema =
    int64 region,    int64 tenure,    int64 age,    int64 address,
    int64 income,   int64 ed,    int64 employ, int64 equip,
    int64 callcard, int64 wireless, float64 longmon, float64 tollmon,
    float64 equipmon, float64 cardmon, float64 wiremon, float64 longten,
    float64 tollten, float64 cardten, int64 voice,    int64 pager,
    int64 internet, int64 callwait, int64 confer,    int64 ebill,
    float64 loglong, float64 logtoll, float64 lninc,    int64 custcat;
```

Next, we add the predictive analytics generated by scoring with this predictive model using the information in the XML file that describes the output data model of the published scoring branch:

```
static DataSchemaPlus =
  DataSchema, tuple<int64 predictedChurn>;
```

A good practice is to have an InfoSphere Streams application you can use to test the latency of the predictive models; therefore, we create a composite that can be used to implement these tests now. We power these tests with the same data file used in the score-per-second tests on the SPSS Modeler Client:

```
stream<DataSchema> data = FileSource() {
  param file: "../data/churn.csv";
}
```

Configuring the SPSSScoring operator requires telling it where the file assets are that represent the scoring branch image to be executed by setting the pimfile, parfile, and xmlfile parameters. These file paths have to be resolvable in any Streams node on which this configured scoring operator is deployed.

We also specify the mapping from the Streams application's attributes to the fields of the input data source in the scoring branch by listing the input fields in the modelFields parameter and specifying the match attribute expressions in the

streamAttributes parameter. Note the streamAttribute expressions do not have to match by name or be simple attribute reference expressions.

We defined what attributes from the input tuple we want to replace with the value resulting from the execution of the scoring branch, if any, and also the generated predictive analytics we want to add to the output tuple using the fromModel functions of this operator, shown in Example 15-1.

Example 15-1 fromModel

```
stream<DataSchemaPlus> scorer = com.ibm.spss.streams.analytics::SPSSScoring(data) {
  param
    pimfile: "../data/churn.pim";
    parfile: "../data/churn.par";
    xmlfile: "../data/churn.xml";
  modelFields:
    region, "tenure", "age", "address", "income", "ed", "employ",
    equip, "callcard", "wireless", "longmon", "tollmon", "equipment", "cardmon",
    wiremon, "longten", "tollten", "cardten", "voice", "pager", "internet",
    callwait, "confer", "ebill", "loglong", "logtoll", "lninc", "custcat";
  streamAttributes:
    region, tenure, age, address, income, ed, employ,
    equip, callcard, wireless, longmon, tollmon, equipment, cardmon,
    wiremon, longten, tollten, cardten, voice, pager, internet,
    callwait, confer, ebill, loglong, logtoll, lninc, custcat;
  output
    scorer:
      predictedChurn = fromModel("PredictedChurn");
}
```

The goal of this SPL composite is a focused test of the scoring branch on representative hardware in an InfoSphere Streams instance, so we add a simple Custom operator to measure performance in a stand-alone execution pattern for this chapter. The operator initializes some state variables, counts the tuples as they go by, and when the application terminates, outputs a little summary, as shown in Example 15-2.

Example 15-2 Output summary

```
stream<DataSchemaPlus> scores = Custom(scorer) {
  logic
    state: {
      mutable int64 nTuples = 0;
      mutable float64 startTS = getTimestampInSecs();
      mutable float64 endTS = 0.0;
    }
}
```

```

    onTuple scorer : {
        ++nTuples;
        submit(scorer, scores);
    }

    onPunct scorer : {
        if (currentPunct() == Sys.FinalMarker) {
            endTS = getTimestampInSecs();
            printStringLn("*** START Execution Summary ***");
            printString(" Execution time in seconds: ");
println(endTS - startTS);
            printString(" Total number of tuples : ");
println(nTuples);
            printString(" Microseconds per tuple : ");
println(((endTS - startTS) / (float64)nTuples) * 1000000.0);
            printStringLn("*** END Execution Summary ***");
        }
    }
}

```

We write the results to a file for comparison with the SPSS Modeler output, if that is what the data analyst wants, as follows:

```

() as Writer = FileSink(scores) {
    param file: "../..data/churn_scores.csv";
}

```

In a real Streams application, the input data might come from one or more continuously streaming sources of data. The predictive analytics generated by our scoring branch would be processed by further downstream application segments, written to external systems or saved in historical data stores.

A real Streams application should also enable model refresh. Figure 15-8 on page 524 shows the SPSSPublish operator being used to prepare the new execution image representing the refreshed predictive model, but also the SPSSRepository operator that can be used to listen for changes in the SPSS Collaboration and Deployment Services repository to fully automate the model refresh flow.

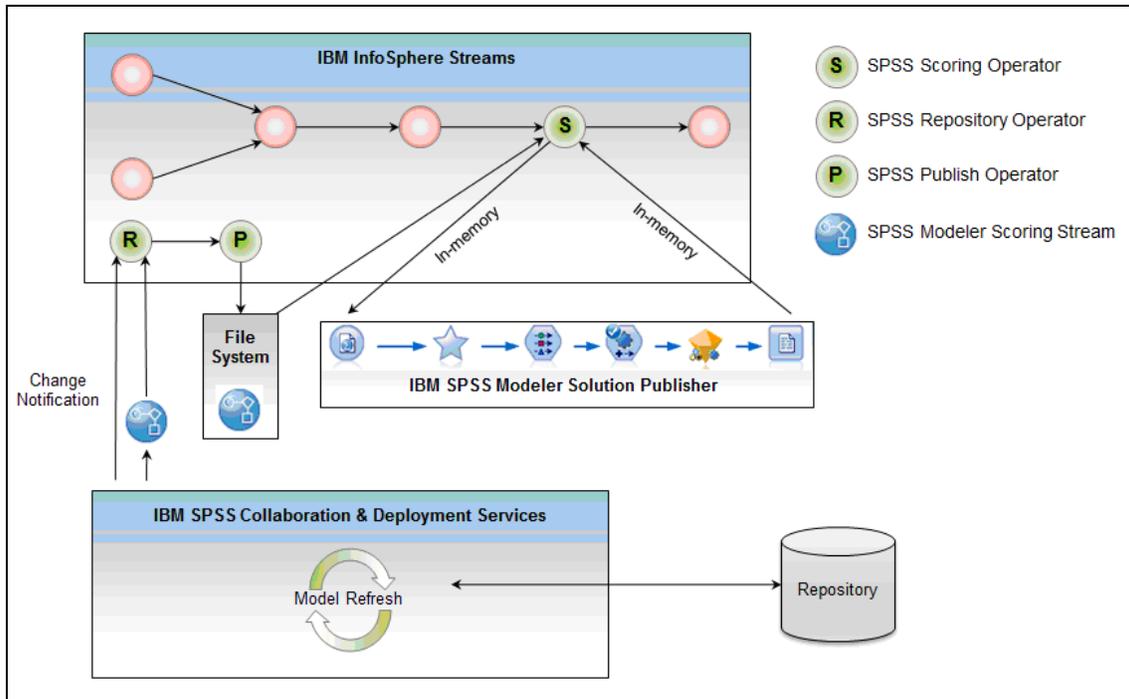


Figure 15-8 InfoSphere Streams and SPSS product integration architecture

Figure 15-8 shows the notification of a refreshed model being outside the primary data stream. The SPSS Scoring operator also performs the preparation of the published image in a worker thread to avoid any blocking of the primary data stream. When the refreshed model is successfully prepared and ready for scoring, it is swapped into the scoring flow between tuple scoring events and the previously prepared image is released.

In our example, we use the DirectoryScan operator from the standard toolkit for InfoSphere Streams to detect an updated SPSS Modeler file and notify the SPSS Publish operator of the need to refresh scoring, as shown in Example 15-3.

Example 15-3 Notify need to refresh scoring

```

stream<rstring strFilePath> strFile = DirectoryScan(){
    param
        directory : "./home/streamsadmin/";
        pattern : "churn.str";
}

```

The SPSSPublish operator is configured to look for notifications, as depicted in Example 15-4, on the desired SPSS Modeler source file, and in this case relies on the scoring branch being marked in the file when it was saved.

Example 15-4 Configuring the SPSSPublish operator

```
stream<outputStream> notifier =
com.ibm.spss.streams.analytics::SPSSPublish(strFile){
    param
        sourceFile: "/home/streamsadmin/churn.str";
    }
}
```

The only change needed in the SPSSScoring operator configuration is to wire the “notifier” into the optional port of the SPSSScoring operator designed to react to these notifications, as shown in Example 15-5.

Example 15-5 Wiring the notifier

```
stream<DataSchemaPlus> scorer = com.ibm.spss.streams.analytics::SPSSScoring(data; notifier) {
    param
        pimfile: "../data/churn.pim";
        parfile: "../data/churn.par";
        xmlfile: "../data/churn.xml";
        modelFields:
            "region", "tenure", "age", "address", "income", "ed", "employ",
            "equip", "callcard", "wireless", "longmon", "tollmon", "equipment", "cardmon",
            "wiremon", "longten", "tollten", "cardten", "voice", "pager", "internet",
            "callwait", "confer", "ebill", "loglong", "logtoll", "lninc", "custcat";
        streamAttributes:
            region, tenure, age, address, income, ed, employ,
            equip, callcard, wireless, longmon, tollmon, equipment, cardmon,
            wiremon, longten, tollten, cardten, voice, pager, internet,
            callwait, confer, ebill, loglong, logtoll, lninc, custcat;

    output
        scorer:
            predictedChurn = fromModel("PredictedChurn");
    }
}
```

At this point, the Streams application developer can add these configurations of the operators from the SPSS Analytic Toolkit for InfoSphere Streams into the production application.

15.5 Summary

In this chapter, we described how the Streams application developer and the data analyst accomplish an integration of SPSS predictive analytics into an InfoSphere Streams application.

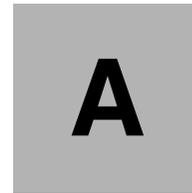
This integration includes the following activities performed by the analyst and developer:

- ▶ Data exploration
- ▶ Identification of significant predictors
- ▶ Training and evaluation of the predictive model
- ▶ Design of the scoring branch to be used in the Streams application
- ▶ Configuration of the SPSS Scoring operator in the Streams application
- ▶ Performance testing
- ▶ Predictive model refresh planning

This chapter is only an introduction to the activities related to the development of a predictive model and the design of a scoring branch for Streams integration. It illustrates how the application developer and the data analyst can work together to formalize their requirements, designs, and implementation plans. The goal of these integration planning efforts is to integrate SPSS predictive analytics in a high-throughput and low-latency Streams application, and also to plan for the refreshing of the scoring plan to keep the generated predictive analytics as accurate as possible.

The example presented in this chapter assumes the Streams application was designed around the requirements of the predictive analytics. The opposite (descriptive) is common and in this approach the first challenge for the data analyst is to determine if an acceptably high confidence prediction can be made using the attributes of the application's current data stream. As in the first approach of predictive analytics, there are usually discussions and negotiations in data quality and content required to get a high-value integration of predictive analytics in the Streams application.

As in all InfoSphere Streams applications, the latency requirements must be honored in the predictive model refresh planning. We have used a simple example of measuring the per-score performance on the reference hardware of the data analyst and their SPSS Modeler Client software and on the reference hardware of the InfoSphere Streams instance to create a "delta" to use in the refresh planning and actual activities that promote a refresh of the predictive model into the live Streams application.



Additional material

This book refers to additional material that can be downloaded from the Internet as described in the following sections.

Locating the web material

The web material associated with this book is available in softcopy on the Internet from the IBM Redbooks web server. Point your web browser at:

<ftp://www.redbooks.ibm.com/redbooks/SG248139>

Alternatively, you can go to the IBM Redbooks website at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the IBM Redbooks form number, SG24-8139.

Using the web material

Additional web material that accompanies this book includes the following files:

<i>File name</i>	<i>Description</i>
Geospatial.zip	This file includes all code listed in Chapter 10, “Geospatial Toolkit” on page 241. You can download file as a project, extract it, and then import into Streams Studio.
SampleApp.zip	This file includes the sample Streams application that is detailed in Chapter 4, “Analytics entirely with SPL” on page 85. You might want to experiment with this sample Streams application to better understand how to work with Streams applications and data.

Create a subdirectory (folder) on your workstation, and extract the contents of the web material .zip file into that folder.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

The following IBM Redbooks publications provide additional information about the topic in this document. Note that some publications referenced in this list might be available in softcopy only.

- ▶ *Addressing Data Volume, Velocity, and Variety with IBM InfoSphere Streams V3.0*, SG24-8108
- ▶ *IBM InfoSphere Streams: Assembling Continuous Insight in the Information Revolution*, SG24-7970
- ▶ *IBM InfoSphere Streams Harnessing Data in Motion*, SG24-7865
- ▶ *InfoSphere DataStage Parallel Framework Standard Practices*, SG24-7830

You can search for, view, download or order these documents and other Redbooks, Redpapers, Web Docs, draft and additional materials, at the following website:

ibm.com/redbooks

Online resources

These websites are also relevant as further information sources:

- ▶ Cayuga: Stateful Publish/Subscribe for Event Monitoring, from Cornell University Database Systems.
<http://www.cs.cornell.edu/bigreddata/cayuga/>
- ▶ IBM InfoSphere Streams Version 3.0 information center
<http://pic.dhe.ibm.com/infocenter/streams/v3r0/topic/com.ibm.swg.im.infosphere.streams.cep-toolkit.doc/doc/cep-overview.html>
- ▶ IBM InfoSphere Streams Version 3.1 information center:
<http://pic.dhe.ibm.com/infocenter/streams/v3r1/index.jsp?topic=%2Fcom.ibm.swg.im.infosphere.streams.homepage.doc%2Fdoc%2Fic-homepage.html>

- ▶ Adding toolkit locations:
<http://pic.dhe.ibm.com/infocenter/streams/v3r0/topic/com.ibm.swg.im.infosphere.streams.studio.doc/tasks/tusing-working-with-toolkits-adding-toolkit-locations.html>
- ▶ Streams Exchange:
<https://www.ibm.com/developerworks/mydeveloperworks/groups/service/html/communityview?communityUuid=d4e7dc8d-0efb-44ff-9a82-897202a3021e>
- ▶ Download the WebSphere MQ Client:
<http://www.ibm.com/software/integration/wmq/clients/>
- ▶ Download WebSphere MQ Server:
<http://www.ibm.com/software/integration/wmq/>
- ▶ Great-circle, haversine calculations, and original Vincenty paper:
http://en.wikipedia.org/wiki/Great-circle_distance
- ▶ Developing streams applications using Streams Studio:
http://pic.dhe.ibm.com/infocenter/streams/v3r0/nav/5_0
- ▶ SPL Java Operator API documentation:
<http://pic.dhe.ibm.com/infocenter/streams/v2r0/index.jsp?topic=%2Fcom.ibm.swg.im.infosphere.streams.javadoc.api.doc%2Fdoc%2Findex.html>
- ▶ AQL tutorial video series:
<http://www.youtube.com/watch?v=8RwunzmPu4Q>
<http://www.youtube.com/watch?v=BpddYCeZl5o>
http://www.youtube.com/watch?v=0-7WtfxLJ8&list=PL7FnN5oi7Ez_KjX7zYhBc8GiK-HoNmqrJ&index=8
- ▶ AQL user-defined functions:
<http://ibm.co/111YKJz>

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services



Redbooks

IBM InfoSphere Streams: Accelerating Deployments with Analytic Accelerators

(1.0" spine)
0.875" <-> 1.498"
460 <-> 788 pages



IBM InfoSphere Streams Accelerating Deployments with Analytic Accelerators



Develop real-time analytic applications with toolkits and accelerators

Build prototypes rapidly with visual development

Assemble continuous insight from data in motion

This IBM Redbooks publication describes visual development, visualization, adapters, analytics, and accelerators for IBM InfoSphere Streams (V3), a key component of the IBM Big Data platform. Streams was designed to analyze data in motion, and can perform analysis on incredibly high volumes with high velocity, using a wide variety of analytic functions and data types.

The Visual Development environment extends Streams Studio with drag-and-drop development, provides round tripping with existing text editors, and is ideal for rapid prototyping. Adapters facilitate getting data in and out of Streams, and V3 supports WebSphere MQ, Apache Hadoop Distributed File System, and IBM InfoSphere DataStage. Significant analytics include the native Streams Processing Language, SPSS Modeler analytics, Complex Event Processing, TimeSeries Toolkit for machine learning and predictive analytics, Geospatial Toolkit for location-based applications, and Annotation Query Language for natural language processing applications. Accelerators for Social Media Analysis and Telecommunications Event Data Analysis sample programs can be modified to build production level applications.

Want to learn how to analyze high volumes of streaming data or implement systems requiring high performance across nodes in a cluster? Then this book is for you.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks