Redbooks
ibm.com/redbooks

Apache Spark Implementation on IBM z/OS

Lydia Parziale

Joe Bostian

Ravi Kumar

Ulrich Seelbach

Zhong Yu Ye

Analytics

z Systems

IBM

Redbooks

International Technical Support Organization

Apache Spark Implementation on IBM z/OS

August 2016

SG24-8325-00

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

First Edition (August 2016)

This edition applies to Version 2, Release 2 of IBM z/OS (product number 5650 ZOS), Apache Spark 1.5.2

Contents

Notices . . . vii
Trademarks . . . viii

IBM Redbooks promotions . . . ix

Preface . . . xi
Authors . . . xi
Now you can become a published author, too . . . xii
Comments welcome . . . xiii
Stay connected to IBM Redbooks . . . xiii

Chapter 1. Architectural overview . . . 1
1.1 Open source analytics on z/OS . . . 2
1.1.1 Benefits of Spark on z/OS . . . 2
1.1.2 Drawbacks of implementing off-platform analytics . . . 3
1.1.3 A new chapter in analytics . . . 4
1.2 Planning your environment . . . 4
1.3 Reference architecture . . . 6
1.3.1 Spark server architecture . . . 6
1.3.2 Spark environment architecture . . . 7
1.3.3 Implementation with Jupyter Notebooks . . . 9
1.3.4 Scala IDE . . . 12
1.4 Security . . . 13

Chapter 2. Components and extensions . . . 15
2.1 Apache Spark component overview . . . 16
2.1.1 Resilient Distributed Datasets and caching . . . 16
2.1.2 Components of a Spark cluster on z/OS . . . 17
2.1.3 Monitoring . . . 18
2.1.4 Spark and Hadoop . . . 21
2.2 Mainframe Data Services for IBM z/OS Platform for Apache Spark . . . 21
2.2.1 Virtual tables . . . 23
2.2.2 Virtual views . . . 23
2.2.3 SQL queries . . . 23
2.2.4 MDSS JDBC driver . . . 23
2.2.5 IBM z/OS Platform for Apache Spark Interface for CICS/TS . . . 24
2.3 Spark SQL . . . 25
2.3.1 Reading from z/OS data source into a DataFrame . . . 26
2.3.2 Writing DataFrame to a DB2 for z/OS table using saveTable method . . . 26
2.4 Streaming . . . 27
2.5 GraphX . . . 28
2.5.1 System G . . . 28
2.6 MLlib . . . 28
2.7 Spark R . . . 29

Chapter 3. Installation and configuration 31
3.1 Installing IBM z/OS Platform for Apache Spark 32
3.2 The Mainframe Data Service for Apache Spark 32
3.2.1 Installing the MDSS started task 32
3.2.2 Configuring access to DB2 37
3.2.3 Configuring access to IMS databases. 38
3.2.4 The ISPF Panels 39
3.2.5 Installing and configuring Bash 45
3.2.6 Check for /usr/bin/env 46
3.3 Installing workstation components 46
3.3.1 Installing Data Service Studio 46
3.3.2 Installing the JDBC driver on the workstation 48
3.4 Configuring Apache Spark for z/OS 49
3.4.1 Create log and worker directories 49
3.4.2 Apache Spark directory structure 50
3.4.3 Create directories and local configuration. 50
3.4.4 Installing the Data Server JDBC driver 53
3.4.5 Modifying the log4j configuration 54
3.4.6 Adding the Spark binaries to your PATH 55
3.5 Verifying the installation 55
3.6 Starting the Spark daemons 57

Chapter 4. Spark application development on z/OS 61
4.1 Setting up the development environment 62
4.1.1 Installing Scala IDE. 62
4.1.2 Installing Data Server Studio plugins into Scala IDE 62
4.1.3 Installing and using sbt 63
4.2 Accessing VSAM data as an RDD 65
4.2.1 Defining the data mapping 65
4.2.2 Building and running the application 70
4.3 Accessing sequential files and PDS members 71
4.4 Accessing IBM DB2 data as a DataFrame 71
4.5 Joining DB2 data with VSAM 72
4.6 IBM IMS data to DataFrames 73
4.7 System log 74
4.8 SMF data. 76
4.9 JavaScript Object Notation 79
4.10 Extensible Markup Language 82
4.11 Submit Spark jobs from z/OS applications 84

Chapter 5. Production integration 87
5.1 Production deployment 88
5.2 Running Spark applications from z/OS batch 88
5.3 Starting Spark master and workers from JCL 89
5.4 System level tuning 89
5.4.1 Tuning the MDSS server. 90
5.4.2 Tuning z/OS UNIX settings 90

Chapter 6. IBM z/OS Platform for Apache Spark and the ecosystem. 91
6.1 Tidy data repository. 93
6.2 Jupyter notebooks. 94
6.2.1 The Jupyter notebook overview 94
6.2.2 Docker and the platforms that support it 95
6.2.3 The dockeradmin userid 96

6.2.4 The Role of SSH 96
6.2.5 Creating the Docker container 97
6.2.6 A note about network configuration 98
6.2.7 Building the Jupyter scala workbench 98

Chapter 7. Use case patterns 103
7.1 Banking and finance 104
7.1.1 Churn prediction 104
7.1.2 Fraud prevention 106
7.1.3 Upsell opportunity detection 111
7.2 Insurance industry 112
7.2.1 Claims payment analytics 112
7.3 Retail industry 113
7.3.1 Product recommendations 113
7.4 Other use case patterns for IBM z/OS Platform for Apache Spark 114
7.4.1 Analytics across OLTP and warehouse information 114
7.4.2 Analytics combining business-owned data and external / social data 114
7.4.3 Analytics of real-time transactions through streaming, combining with OLTP and social 114
7.5 Operations analysis 114
7.5.1 SMF data 115
7.5.2 Syslog data 115

Appendix A. Sample code to run on Apache Spark cluster on z/OS 117

Appendix B. FAQ: Frequently asked questions, and answers 121
General 122
Support 122
Technical 122

Related publications 123
IBM Redbooks 123
Other publications 123
Online resources 123
Help from IBM 123

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

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, MD-NC119, Armonk, NY 10504-1785, US

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 jurisdictions 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 provide in any way it believes appropriate without incurring any obligation to you.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

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.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

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 actual people or business enterprises 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. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Trademarks

Find and read thousands of IBM Redbooks publications

- Search, bookmark, save and organize favorites
- Get personalized notifications of new content
- Link to the latest Redbooks blogs and videos

Get the latest version of the Redbooks Mobile App

iOS **Download Now** Android

Promote your business in an IBM Redbooks publication

Place a Sponsorship Promotion in an IBM® Redbooks® publication, featuring your business or solution with a link to your web site.

Qualified IBM Business Partners may place a full page promotion in the most popular Redbooks publications. Imagine the power of being seen by users who download millions of Redbooks publications each year!

ibm.com/Redbooks
About Redbooks → Business Partner Programs

THIS PAGE INTENTIONALLY LEFT BLANK

Preface

The term *big data* refers to extremely large sets of data that are analyzed to reveal insights, such as patterns, trends, and associations. The algorithms that analyze this data to provide these insights must extract value from a wide range of data sources, including business data and live, streaming, social media data.

However, the real value of these insights comes from their timeliness. Rapid delivery of insights enables anyone (not only data scientists) to make effective decisions, applying deep intelligence to every enterprise application.

Apache Spark is an integrated analytics framework and runtime to accelerate and simplify algorithm development, depoyment, and realization of business insight from analytics. Apache Spark on IBM® z/OS® puts the open source engine, augmented with unique differentiated features, built specifically for data science, where big data resides.

This IBM Redbooks® publication describes the installation and configuration of IBM z/OS Platform for Apache Spark for field teams and clients. Additionally, it includes examples of business analytics scenarios.

Authors

This book was produced by a team of specialists from around the world, working at the International Technical Support Organization (ITSO), Poughkeepsie Center.

Lydia Parziale is a Project Leader for the ITSO team in Poughkeepsie, New York, with United States and international experience in technology management, including software development, project leadership, and strategic planning. Her areas of expertise include business development and database management technologies. Lydia is a certified Project Management Professional (PMP) and an IBM Certified information technology (IT) Specialist with a Master of Business Administration (MBA) in Technology Management. She has been employed by IBM for over 25 years in various technology areas.

Joe Bostian is a Senior Software Engineer in Poughkeepsie, NY. He has 31years of experience in the field of Software design and development. He holds a Masters degree from Rensselaer Polytechnic Institute, and a Bachelors degree from Purdue university, both in computer science. His area of expertise is in the development of operating systems componentry and middleware. He has previously contributed to Redbooks publications about Extensible Markup Language (XML) processing on z/OS, and IBM Lotus® Notes for IBM S/390® products.

Ravi Kumar is a Senior Managing Consultant at IBM (Analytics Platform, North American Lab Services). Ravi is a Distinguished IT Specialist (Open Group certified) with more than 23 years of IT experience. He has an MBA from University of Nebraska, Lincoln. He contributed to seven other Redbooks publications in the areas of Database, Analytics Accelerator, and Information Management tools.

Ulrich Seelbach is an IT Architect at IBM Systems in Frankfurt, Germany. He joined IBM in 1995, and has more than 15 years of experience with Java technology on z/OS and its major subsystems, including IBM WebSphere® for z/OS, IBM DB2®, and IBM CICS® Transaction Server. He previously co-authored several other IBM Redbooks publications, including *DB2 for z/OS and OS/390: Ready for Java*, SG24-6435; *ARCHIVED: Pooled JVM in CICS Transaction Server V3*, SG24-5275; and *Enabling z/OS Applications for SOA*, SG24-7669. As a member of the z Software Services team, he supports numerous European customers, mainly in the banking and insurance industries, in all topics related to Java and XML workload on z/OS. He holds a degree in Computer Science from the University of Erlangen, Germany.

Zhong Yu Ye is an Advisory IT Specialist at IBM Client Innovation Center in Shenzhen, China. He joined IBM in 2008 and has over 10 years of experience in z/OS and related subsystems. He currently works for the IBM Remote Lab Platform (IRLP) providing system support/development for education services across the globe.

Thanks to the following people for their contributions to this project:

Robert Haimowitz
ITSO, Poughkeepsie Center

Denis Gaebler
IBM Germany, IBM IMS™ Worldwide Advocates Team

David Rice, Richard Ko, James Perlik, John Goodyear, Michael Casile, Dan Gisolfi, Mythili Venkatakrishnan
IBM US

Stephane Faure
IBM France

Gregg Willhoit, Patrycja Grzesznik
Rocket Software

Special thanks to the additional team who took the time to perform a rigorous technical review for us:

AnnMarie Vosburgh, Erin Farr, Kieron Hinds, Jessie Yu, Christian Rund
IBM US

Andy Seuffert
Rocket Software

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 2 - 6 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:

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:

  ibm.com/redbooks

- Send your comments in an email:

  redbooks@us.ibm.com

- Mail your comments:

  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

1

Architectural overview

The Apache Spark architecture is highly flexible, allowing it to be deployed in various heterogeneous environments. It allows the inherent strengths of the IBM z/OS platform to become apparent within a carefully planned and configured enterprise. With the configurations discussed here, you can create a highly efficient analytics deployment that avoids latency, costly processing inefficiencies, and security concerns associated with data movement.

In addition, you can integrate Apache Spark into an optimized hybrid analytics framework within your organization. The IBM z/OS Platform for Apache Spark enables you to create a layered/tiered analytics infrastructure that leverages data-in-place analytics to maximize value while minimizing data movement. We do not suggest that all analytics will be on z/OS, but rather a structure that allows for flexible placement of analytics.

This chapter introduces the following topics:

- 1.1, “Open source analytics on z/OS” on page 2
- 1.2, “Planning your environment” on page 4
- 1.3, “Reference architecture” on page 6
- 1.4, “Security” on page 13

1.1 Open source analytics on z/OS

Analytics use cases for IBM Platform for Apache Spark depend on the nature of the data to be analyzed: The volume, value, whether it is mission critical, sensitivity, and rate of change. But, Spark is Spark. There is no “Spark on IBM z™ Systems” paradigm from an applications perspective. Nevertheless, you can also benefit from the strong synergy between Apache Spark and z Systems.

1.1.1 Benefits of Spark on z/OS

With z/OS system characteristics, such as collocation of transactions and data, the following are the key benefits of IBM z/OS Platform for Apache Spark:

- Real-time, fast, efficient access to current transactional data and to historical data.
- Integrated, optimized, parallel access to almost all z/OS data environments, and to distributed data sources.
- All Spark memory structures that contain sensitive data are governed with z/OS security capabilities.
- Analyzing data in place means that you can include real-time operational data and warehouse data.
- No need to have all data on z/OS, because Spark on z/OS can access various sources, including those outside of IBM z Systems™.
- Sysplex-enabled Spark clusters for world class availability. Spark can be clustered across more than one Java virtual machine (JVM), and these Spark environments can be dispersed across an IBM Parallel Sysplex®.
- Leverages z/OS superior capabilities in memory management, compression, and Remote Direct Memory Access (RDMA) communications to provide a high-performance scale up and scale out architecture.
- Uses unique features of z Systems, such as large pages, incorporating dynamic random access memory (DRAM) with large amounts of Flash as an attractive means to provide scalable elastic memory.
- Provides a best fit analytic capability for the investments made in System Management Facilities (SMF) in-memory analytics.
- Participant in hybrid structure: Spark z/OS can be a participant in a hybrid environment that involves Spark clusters on z as well as distributed, tiered/layered approach.
- Benefits from IBM z Systems compression technology, particularly when compressing internal data for caching and shuffling.
- Simultaneously Multithreading (SMT-2) for added thread performance.
- Single-instruction, multiple-data (SIMD) for better performance on select operations.
- IBM System z® Integrated Information Processor (zIIP)-eligible for affordability.
- Leverage common skill: Spark & Jupyter Notebook skill that is consistent across platforms running Spark.
- Integration with WLM: With Spark z/OS today, integration at the Optimized Data Integration layer. There might be plans to enhance this further in the Spark stack in the near term.
- Intra-Structured Query Language (SQL) and intra-partition parallelism for optimal, implemented transparently to the Spark application, data access.

- Support of SQL standards is much richer than “vanilla” SparkSQL. It can support SQL92 and SQL99 whereas standard SparkSQL is a subset of HiveQL.
- Access to many different data sources not generally supported by SparkSQL (IBM Virtual Storage Access Method (VSAM), advanced data base system (ADABAS), physical sequential (PS), partitioned data set extended (PDSE), IBM System Management Facilities (SMF), SYSLOG, abd Logstream) in addition to IBM DB2 and IBM Information Management System (IBM IMS).
- Very granular integration with System Authorization Facility (SAF) interfaces for security configuration to suit needs.

In addition to the architectural benefits of Apache Spark (such as resilient in-memory architecture, massively parallel processing compute engine, easy to use application programming interfaces (APIs) and so on), there are four major themes to remember about IBM z/OS Platform for Apache Spark. These themes are described in the following sections.

Data security increased by keeping the data where it belongs

Avoid costs that are associated with data extract, transform, and load (ETL) and data breach situations by eliminating unreliable and less secure environments.

Single unified platform to implement complex analytics

Spark provides real-time analytics with mission-critical operational data and machine learning on IBM z Systems.

Simplified development and deployment

Analytics code developed on any platform can be used on z Systems. Applications developed on other platforms can also develop on z Systems, but now they have access to data that they didn’t have access to before. Users now have access to data on z/OS using all of the languages they already know and use, such as Java and Scala.You do not need to learn new skills.

Ask not what Spark on z Systems can do for you

Ask what you can do for Spark on z Systems. You can also contribute to this open source project.

1.1.2 Drawbacks of implementing off-platform analytics

Moving the data around targets a specific set of use cases, none of which can support real-time use cases. When the data is taken off the system of record where the transactional data originates, different groups see snapshots of data that are not fresh and there is no guarantee of consistency in data synchronization. Multiple copies of the data must also be supported, which can lead to data security concerns.

Considering all of these issues, excessive costs (including personnel costs) and inefficiencies are the likely end result. This is a typical challenge that most clients are dealing with today as they reevaluate their analytics strategy to support analytics that are now highly critical to the success of the business.

When your operational data originates on the z Systems platform, you realize the following benefits to your mission critical analytic:

- Total cost of operation (TCO)
- Latency
- Availability

Using IBM z/OS Platform for Apache Spark does not mean that you must replace existing investments, but you can supplement them.

1.1.3 A new chapter in analytics

The IBM z Systems platform provides a truly modern, cost-competitive analytics alternative that is primed to embrace the market shift related to big data, mobile, customer interaction, and cloud initiatives. With the z Systems analytics, organizations can apply the same qualities of service to their business-critical analytics as they do to their transactional systems. An estimated 80% of corporate data is stored or originates on the platform.

Instead of moving the data to the analytics, the z Systems platform is a hybrid transactional and analytics environment, enabling analytics to be collocated with the data, therefore alleviating many of the complexities highlighted previously. Organizations can start with their most pressing analytics issues, quickly realize immediate business value, and then position their analytics strategy to grow and evolve along with business and market demands, all without the need to rearchitect.

With the IBM z Systems platform, organizations can perform prescriptive, predictive, investigative, and cognitive Capacity Management Analytics. This can be accomplished all on one platform. At the same time, you can integrate with many other remote sources of data and applications on other platforms. IBM has ported its IBM Cognos®, IBM SPSS®, and IBM BigInsights® solutions to the platform with Apache Spark, which is the latest analytics platform available on Linux for z Systems and IBM z/OS operating systems.

While Apache Hadoop and Map Reduce provide a parallel computing cluster for the analysis of large and complex data sets, ease of use and timely execution of queries have sometimes been barriers to adoption, falling short in terms of user expectations and experience.

In some cases, Hadoop implementations remain as a data lake or reservoir to store data for future analysis. 2.1, “Apache Spark component overview” on page 16 describes Spark and how it aims to address complexity, speed, ease of use, and why it is key to the future success of analytics and the impact it is having and will continue to have on organizations, the business, their analytics strategies, and data scientists and developers.

1.2 Planning your environment

Before making any decisions about the attributes of your Spark environment, it is important to answer a couple of questions about the nature of your enterprise and your key technical people:

- Do your people have mainframe skills, or are they “don’t-know-don’t-care” when it comes to understanding the platform where Spark is running? Platform-aware users value certain attributes of the environment differently than users who care only about the availability of the data and tools to do their analysis. You need both kinds of people to fill different roles, but it is important to understand who has the biggest influence over the environment.
- Are you integrating your z/OS enterprise transaction processing backend with new Spark infrastructure, or are you redeploying an existing Spark environment on z/OS to get it closer to your enterprise data sources? Each environment has a unique set of potential issues to be aware of. In either case, one population of users must adjust their operational environment more than the other. Knowing which group needs to do the heavy lifting can make anticipating issues easier.

Key personas in the environment

People in a Spark environment perform different, but related roles or personas. It is useful to describe each of these personas so that the structure of the environment can be correlated with a particular persona. Note that the operations and actions needed to fulfill these roles can vary somewhat between a z/OS-based Spark deployment and Spark environments on other platforms, but the personas themselves are the same.

Table 1-1 on page 5 describes the personas that are used in this IBM Redbooks publication, along with their typical responsibilities and how Apache Spark can help.

Table 1-1 Personas

Persona	What they do	How Spark can help
Data Scientist	► Identify patterns, trends, risks, and opportunities in data ► Discover new actionable insights ► Furthest from the platform, generally touches only off-platform assets	► Supports the entire data science workflow, from data access and integration, to machine learning, to visualization ► Provides a growing library of machine-learning algorithms
Information Management Specialist and Application Developer	► Build applications that use advanced analytics in partnership with the data scientist and data engineer ► Follow agile design methodologies ► Optimize performance and meet SLAs	► Supports the top analytics programming languages such as R, SQL, Java, Python, and Scala ► Eliminates programming complexity with libraries such as MLlib and simplifies DevOps ► Makes it easy to embed advanced analytics into applications
Operations Data Wrangler—Data Engineer	► Bridge between the Data Scientist and the Application Developer ► Implement machine-learning algorithms at scale ► Put the right data system to work for the current job ► Generally closest to the platform, less need for off-platform infrastructure	► Abstract data access complexity (Spark doesn't care what your data store is) ► Enables solutions at web scale

These are typical personas commonly found in all Spark environments. Your particular environment might have people in roles that contain more or less responsibility in different areas. For instance, you might have application developers who perform tasks that are separate from the Information Management specialist. The point is that these roles are mutable, and should be tailored to the need of your environment. In all cases though, close collaboration between these personas is an important key to success.

1.3 Reference architecture

The architecture of the Spark environment on a z/OS platform is highly flexible. It can accommodate the unique needs of many different enterprise configurations. The architecture consists of two parts:

- The server side. This is where z/OS is installed and the Spark server is deployed with all of the associated subsystems and components.
- The off-platform environment. This part of the environment provides the user interfaces (UIs) that are primarily used by data scientists, information managers, and application developers.

Several optional parts exist for both sides of the Spark environment. In fact, the entire environment can be thought of as optional. However, avoid the mistake of thinking that when the server is installed and configured, the Spark environment is complete. The applications that drive the Spark server are the “magic” that generates valuable, actionable insights from the raw data.

1.3.1 Spark server architecture

The server part of the reference architecture looks similar to Figure 1-1.

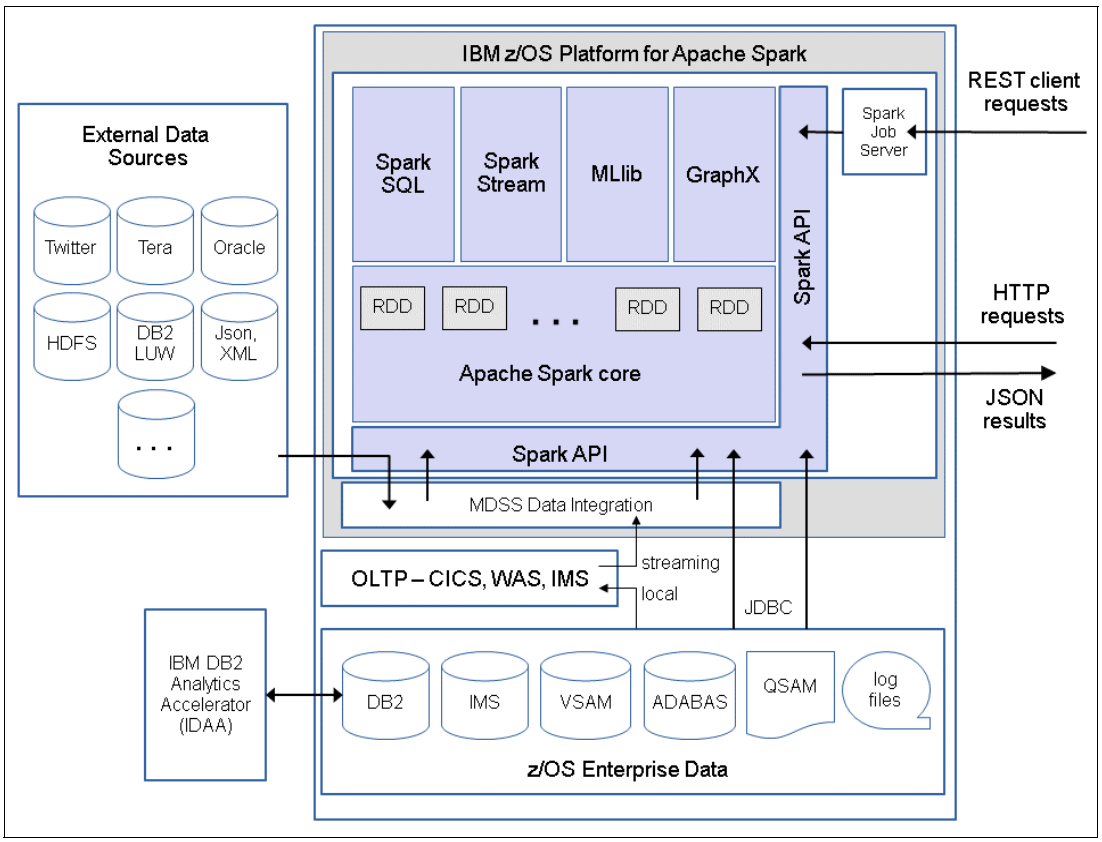


Figure 1-1 Architecture of the Spark server environment on z/OS

The Apache Spark server is made up of several components, including a core set of services, an API, and a collection of libraries that enable key functionality. This is the Spark framework. There is also an optional Spark jobserver component that allows access to Spark through a REST API.

The Mainframe Data Services for Apache Spark for z/OS (MDSS) integration component is included with the IBM z/OS Platform for Apache Spark, and provides the integration facilities for both z and non-z data sources. MDSS is an optimized data integration layer that is unique to Spark on z/OS and provides significant value and differentiation for Spark implementations on z/OS when compared with vanilla Apache Spark capabilities. It also allows for the integration of off-platform external data sources to enhance the capabilities of the Spark environment on z/OS.

Although it is possible to configure and use this environment without MDSS, doing so limits the data sources to those accessible natively by vanilla Apache Spark, typically ava Database Connectivity (JDBC) and Open Database Connectivity (ODBC) enabled environments, such as IBM DB2 and IMS. In addition, even when leveraging IBM DB2 and IBM Information Management System (IBM IMS), MDSS provides a significant set of advantages to both the application and the runtime. The data source configurations described in this book all use the MDSS component.

Online transaction processing (OLTP) can be performed within this environment using IBM Customer Information Control System (IBM CICS), IBM WebSphere Application Server, or IBM Information Management System (IBM IMS). It's important to understand though that while data can be streamed in real time to the Spark server, this environment was never intended to support low-latency, real-time transactions.

Big data configurations have always been about generating business value through insights into very large pools of data, not transaction processing. If you are going to use Spark insights through OLTP, remember that generating these insights can't take place at transaction speed.

1.3.2 Spark environment architecture

A key difference between the Spark environment and other types of client/server architectures is the way users interact with the host. With a traditional architecture, a developer can create an application that runs on the host and responds to requests that a user generates over Hypertext Transfer Protocol (HTTP) by their actions at a web browser.

The application is created using an integrated development environment (IDE) at development time, is deployed to the host system, and ultimately put into production. The user drives the application through their actions.

In a Spark environment, the developer can be thought of as the user. In this case, the developer role is split between the *Information Manager* and *Data Scientist* personas. They are both the consumers of the results from the Spark server and the creators of the applications that produce the results. They generate knowledge from information, and deliver insights from that knowledge to their business colleagues to act upon. In some sense, the ultimate user (the business) never actually uses the Spark environment directly.

The Spark ecosystem can be thought of as a complex heterogeneous, distributed, and interactive development environment. As with the server side, it can be tailored to meet a wide range of user requirements. The characteristics of this environment are determined by all three personas described in the section entitled "Key personas in the environment" on page 5, because the choice of tools to use depends on the preferences of the people involved.

It is possible to use a simple text editor to create applications for the Spark environment and run them in batch. In this instance, no part of the environment described in Figure 1-2 is necessary. Using a text editor disregards much of the Spark functionality that brings value (for example, true interactive application development and graphical visualization of results). Such an environment would not likely be embraced by key people who need to use it.

In practice, it's likely that some or all of the tools and components that are shown in Figure 1-2 are a part of the typical Spark environment.

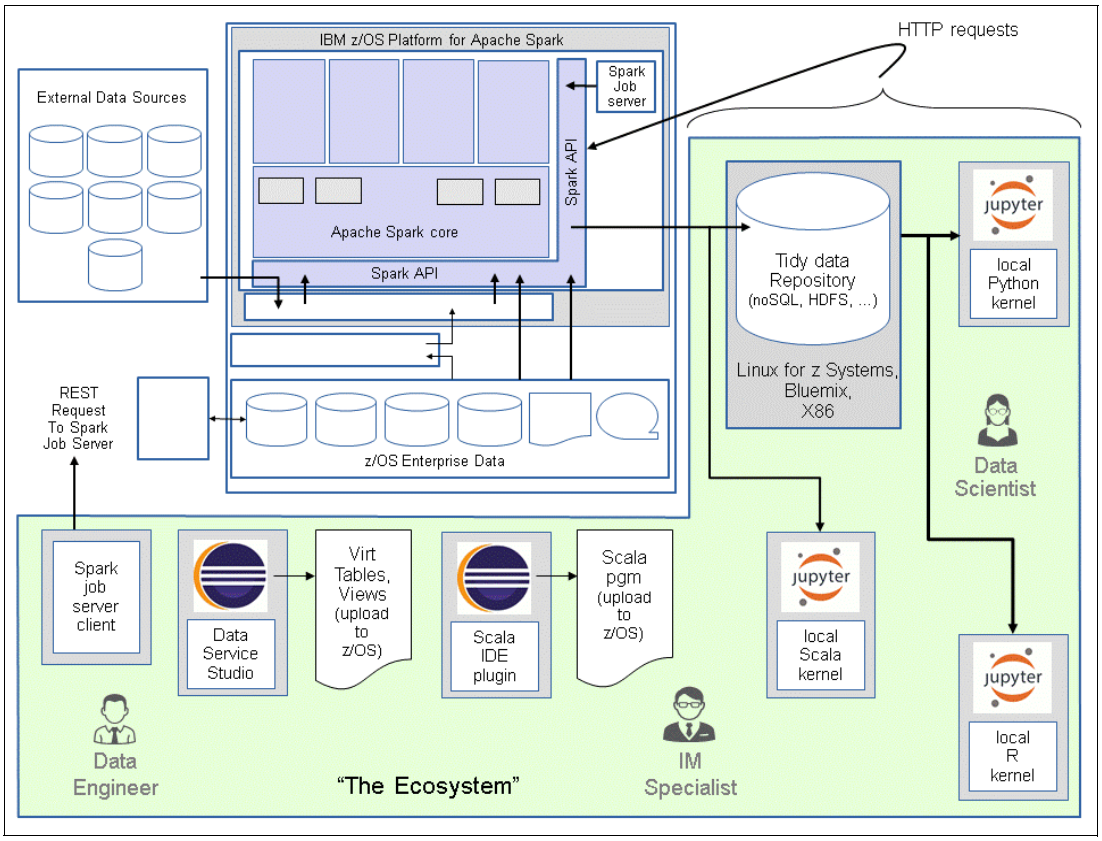


Figure 1-2 The Spark ecosystem of associated tools and components

In this section, we attempt to demonstrate a simple Apache Spark implementation on z/OS without any of the "bells and whistles".

> The reference architecture that demonstrates how the IBM z/OS Platform for Apache Spark can be combined with various other easy-to-use open source tools is discussed in section 1.3.3, "Implementation with Jupyter Notebooks" on page 9.

The rudimentary implementation that is shown in Figure 1-2 uses only the components that are part of the free download from Shopz. These components include the IBM z/OS Platform for Apache Spark, which is an optimized Apache Spark offering that incorporates a Spark Cluster with specialized data virtualization technology.

This implementation is a good starting point for data exploration and deep analytics on z/OS for data engineers who are familiar with the z/OS platform, and data scientists who are comfortable working with Spark shell on UNIX System Services (spark-shell).

In this scenario, not many visualization options are available, but you can perform data exploration by using the interactive Spark-Read-Evaluate-Print-Loop (Spark-REPL) shell. With Spark-REPL, you can also test the outcome of each line of code without needing to complete and run the entire job.

Even with this simple rudimentary architecture, you can easily develop and deploy your analytics solution in to your production environment as demonstrated in Chapter 4, “Spark application development on z/OS” on page 61.

1.3.3 Implementation with Jupyter Notebooks

The reference architecture in this section demonstrates how the IBM z/OS Platform for Apache Spark can be combined with various open source tools. Of specific interest is the combination of Project Jupyter and Apache Spark to provide a flexible and extendible data processing and analytics solution.

The reference architecture in Figure 1-3 on page 10 attempts to serve as a discussion starter and a guide if you are considering the use of Apache Spark. For example, it can be used to accomplish the following tasks:

- Optimize cost within existing transaction processing workloads.
- Optimize existing analytics pipelines with real-time inline ingestion of transactional events (for instance with CICS data).

At the center of the solution is a Tidy Data Repository that is kept fresh with data sets from various data processing end-points. It serves as the central data source for analytical activity. This can be implemented in several ways:

- NoSQL database, storing JavaScript Object Notation (JSON) data.
- Hadoop Distributed File System (HDFS) or IBM General Parallel File System (GPFS™), storing JSON files

Scala Workbench

Use Spark z/OS for data blending, cleansing, transforming, and so on with data-in-place, and then store the results in a Tidy Data Repository or into a DB2 for z/OS table when it makes sense. After successful analysis, you can refresh the data as needed.

Figure 1-3 shows a possible reference architecture.

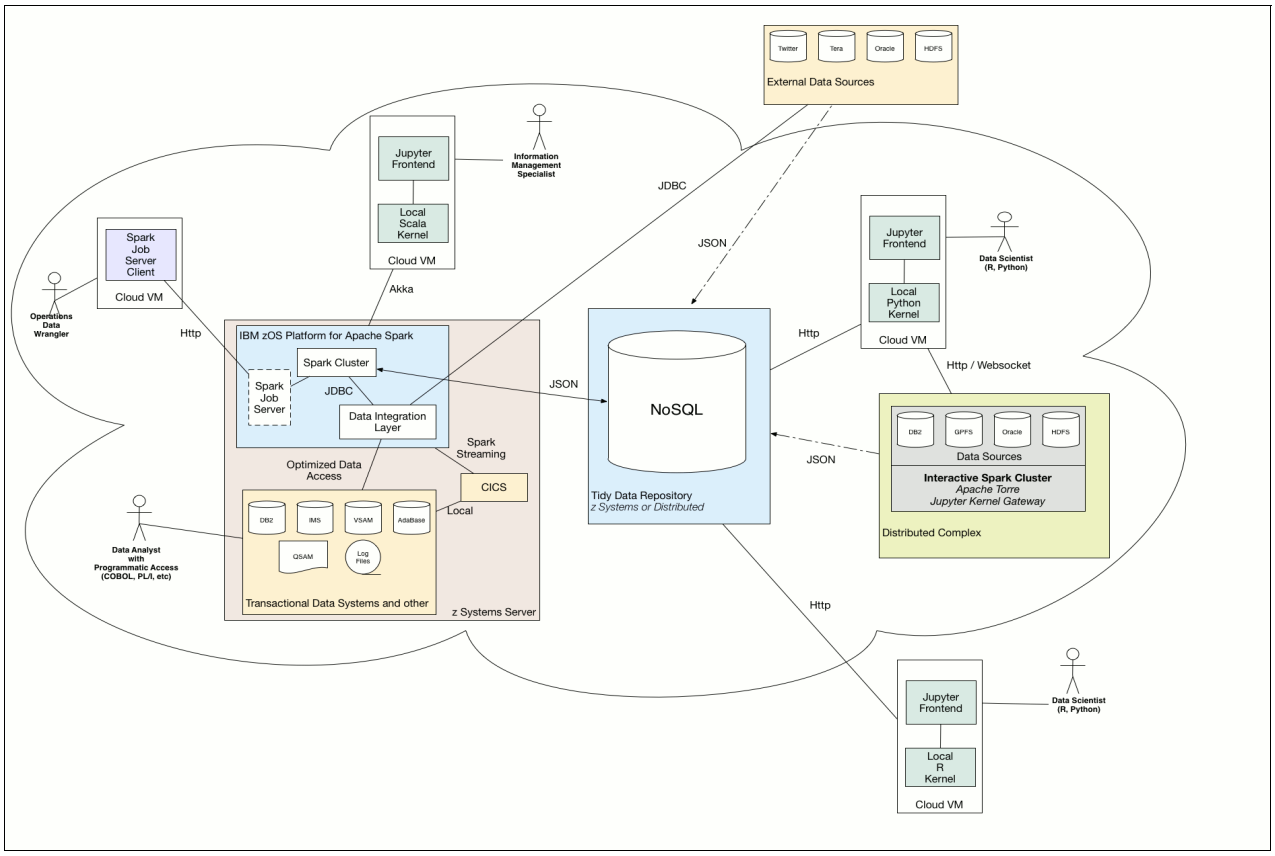


Figure 1-3 Spark for z/OS reference architecture

Flexibility

Various use cases can be addressed by this solution. JSON data sets can be created using Spark jobs that handle complex data processing tasks. Alternatively, Data Scientists can perform ETL activities within notebooks across multiple data sets in the NoSQL database. Depending on available skills and the requirements of a particular analytical task, users can determine when and where to preform ETL activities.

Extensibility

New data sets can be injected into the NoSQL database from various external data sources and data processing environments. For example, a separate instance of Apache Spark can be installed for use with a set of databases on distributed systems. Furthermore, users might want to run jobs that aggregate data from multiple data processing environments.

Interoperability

An ecosystem of analytical tools are appropriately positioned to interoperate with the various solution components. For instance, the Scala Workbench is tightly coupled with the installation of IBM z/OS Platform for Apache Spark.

Integration

This solution should be viewed as a complimentary approach to the data processing challenges of your enterprise. Nothing in this solution is intended to alter the procedures, policies, or existing programmatic approaches that are currently deployed.

Other considerations

Managed Data Processing Environment: Spark provides distributed task dispatching and scheduling. It can be used to allow data processing tasks to be submitted as batch jobs on some predefined frequency. Few users need to interact with such a managed environment for data processing jobs. Only Data Wranglers need a deep understanding of Spark and the tools necessary to integrate with it.

Tidy Data Repository: As data processing tasks are completed, the results of those jobs can be stored in a central location that is more easily accessible by a broader user community. New data sets that are produced by the Spark jobs can be refreshed or purged as wanted by the system administrators or user community. Two obvious deployment topologies stand out:

- NoSQL database deployed in a Linux on Z partition
- NoSQL database deployed on a distributed server

In either case, the NoSQL database chosen should sport a robust set of Python or R libraries for manipulating data.

Content Format: Given the various programming languages used by Data Scientists, the Tidy Data Repository should embrace a data storage format, such as JSON, that is commonly supported across programming languages and data stores.

Table 1-2 describes the personas used in this architecture, along with their typical responsibilities and how Apache Spark can help.

Table 1-2 Personas and tools used in the reference architecture

Persona	Tools used	Description
Data Scientist	► Interactive Insights Workbench	► Open source tool for Python and R users to access data sets for analysis and insight generation.
Information Management Specialist: Application Developer	► Scala Workbench	► Tightly-coupled Jupyter+Spark workbench that allows Scala users with direct access to transactional systems to query, analyze, and visualize enterprise data.
Operations Data Wrangler: Data Engineer	► Spark Job Manager	► Combine Spark Job Server and Spark Job Server Client to provide a robust toolset for managing the lifecycle.

The Spark Job Server and the associated client shown in Figure 1-3 on page 10 were not used by the authors for the purpose of this book. The Spark Job Server provides a REST API for submitting, running, and monitoring Spark Jobs. It also enables the results of jobs to be converted to JSON format. Spark Job Server Client provides Java Client API for development of client GUI tools that enable developers to easily manage Spark jobs.

See Chapter 6, “IBM z/OS Platform for Apache Spark and the ecosystem” on page 91 for information about Docker containers and easy provisioning.

1.3.4 Scala IDE

Figure 1-4 uses the Scala IDE in the rudimentary architecture, enabling ease of application development and production deployment.

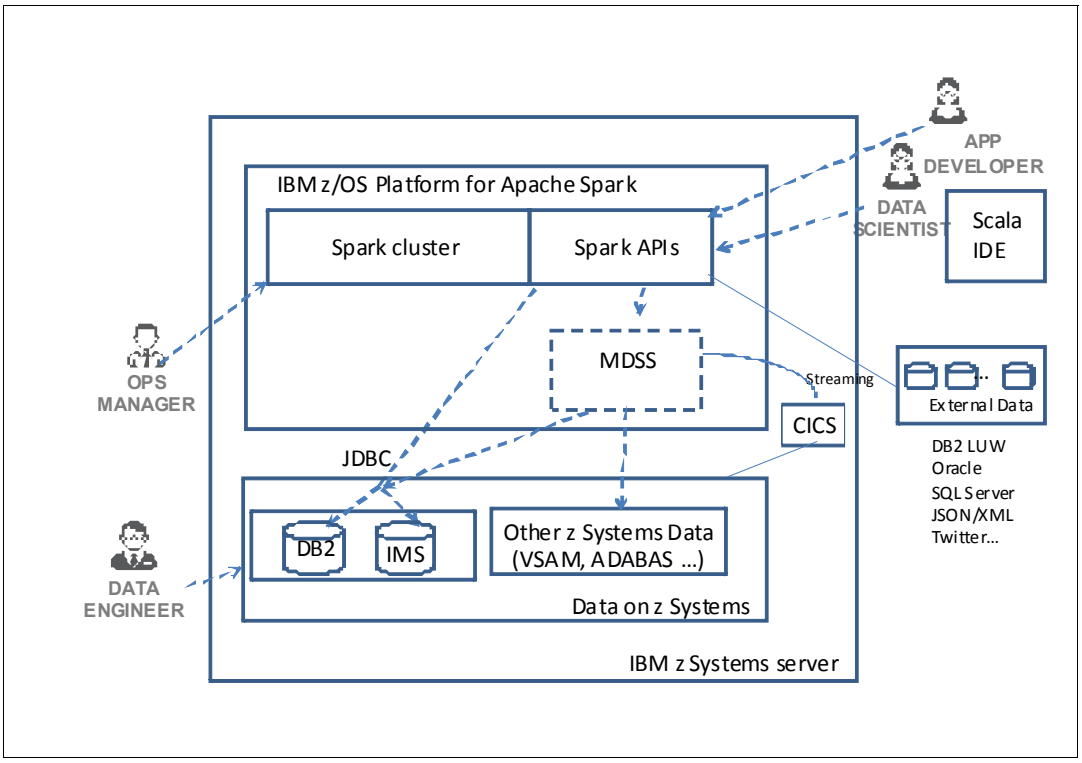


Figure 1-4 Rudimentary architecture with Scala IDE

4.1, “Setting up the development environment” on page 62 describes the Scala tooling in much more detail.

1.4 Security

You don't have to add any new security for the data sources for use with Spark on z/OS. Only authorized users are able to read from the z/OS data sources.

Advanced security is available for SQL, NoSQL, Events, and Services solutions. System programmers typically configure advanced security during the Data Service server installation. IBM z/OS Platform for Apache Spark provides the following security features:

- Security Optimization Management (SOM) is a unique feature within the mainframe integration suite that manages and optimizes mainframe security authentication for any process that requires authentication, such as a web service or SQL call.
- Secure Sockets Layer (SSL) for the Data Service server is transparently supported by the Application Transparent Transport Layer Security (AT-TLS), an IBM Transmission Control Protocol/Internet Protocol (TCP/IP) facility.
- Enterprise auditing supports the new and unique security requirements of Internet applications, while operating in the traditional enterprise computing environment. With enterprise auditing, web applications that access IBM z/OS operating system data and transactions can be used by people who do not have mainframe user IDs.
- Data Service server (AZK) provides protection for its resources using IBM Resource Access Control Facility (IBM RACF®) classes, Top Secret classes, and ACF2 generalized resource rules. You can run multiple instances of Data Service servers and either share the authorization rules or keep them separate.

2

Components and extensions

In addition to the standard components of Spark (including Spark core, Spark Structured Query Language (Spark SQL), Spark Streaming, Spark Graphx, and Spark MLlib), Spark on IBM z/OS comes with a data virtualization component called *Mainframe Data Services for Apache Spark*.

This chapter briefly describes all the components:

- 2.1, “Apache Spark component overview” on page 16
- 2.2, “Mainframe Data Services for IBM z/OS Platform for Apache Spark” on page 21
- 2.3, “Spark SQL” on page 25
- 2.4, “Streaming” on page 27
- 2.5, “GraphX” on page 28
- 2.6, “MLlib” on page 28
- 2.7, “Spark R” on page 29

2.1 Apache Spark component overview

As you can see from Figure 2-1, the component overview of IBM z/OS Platform for Apache Spark is no different from any other Spark distribution.

The Spark Stream enables processing of live streams of data. Examples of data streams include log files that are generated by production web servers, or queues of messages that contain status updates posted by users of a web service.

GraphX is a graph processing library with application programming interfaces (APIs) to manipulate graphs and perform graph-parallel computations.

Spark SQL allows developers to intermix SQL with Spark's programming language, supported by Python, Scala, and Java.

MLlib is the machine learning library that provides multiple types of machine learning algorithms. All of these algorithms are designed to scale out across the cluster as well.

Compression is optimized when running IBM z/OS Platform for Apache Spark and using IBM zEnterprise® Data Compression (IBM zEDC). RDD cache is described in more detail in section 2.1.1, "Resilient Distributed Datasets and caching" on page 16.

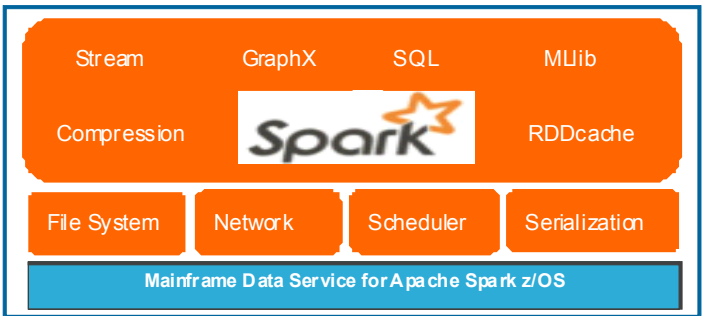


Figure 2-1 Apache Spark component overview

2.1.1 Resilient Distributed Datasets and caching

An RDD is the fundamental data structure of Spark. It's an unchanging distributed collection of objects, and each data set is divided into logical partitions (LPARs) that can be computed on different nodes of a cluster.

The following list provides a quick overview of an RDD:

- It is Spark's basic unit of data.
- It's an immutable, fault-tolerant collection of elements that can be operated on in parallel across a cluster.
- If data in memory is lost, it is re-created from lineage.
- It uses caching, persistence (memory, spilling, and disk), and check-pointing.
- Many database or file types can be supported.
- An RDD is physically distributed across the cluster, but manipulated as one logical entity. Spark distributes any required processing to all partitions where the RDD exists, and also performs necessary redistributions and aggregations.

RDD caching is good for iterative algorithms and fast interactive use. The **`cache()`** method is shorthand for using the default storage level, which is **`StorageLevel.MEMORY_ONLY`** (store deserialized objects in memory). Table 2-1 shows the definition of the available caching levels for RDDs. You can monitor memory usage through the web user interface (UI) that is shown in Figure 2-8 on page 21.

Table 2-1 RDD caching levels

Storage level	Uses disk	Uses memory	Off-heap	Deserialized	Number of replicas
`NONE`	No	No	No	No	-
`DISK_ONLY`	Yes	No	No	No	1
`DISK_ONLY_2`	Yes	No	No	No	2
`MEMORY_ONLY`	No	Yes	No	Yes	1
`MEMORY_ONLY_2`	No	Yes	No	Yes	2
`MEMORY_ONLY_SER`	No	Yes	No	No	1
`MEMORY_ONLY_SER_2`	No	Yes	No	No	2
`MEMORY_AND_DISK`	Yes	Yes	No	Yes	1
`MEMORY_AND_DISK_2`	Yes	Yes	No	Yes	2
`MEMORY_AND_DISK_SER`	Yes	Yes	No	No	1
`MEMORY_AND_DISK_SER_2`	Yes	Yes	No	No	2
`OFF_HEAP`	No	No	Yes	No	1

2.1.2 Components of a Spark cluster on z/OS

The component overview diagram from `http://spark.apache.org` is shown in Figure 2-2 to make it easier to understand the components that are involved, and how they relate to the z/OS platform. The following sections describe these components in more detail.

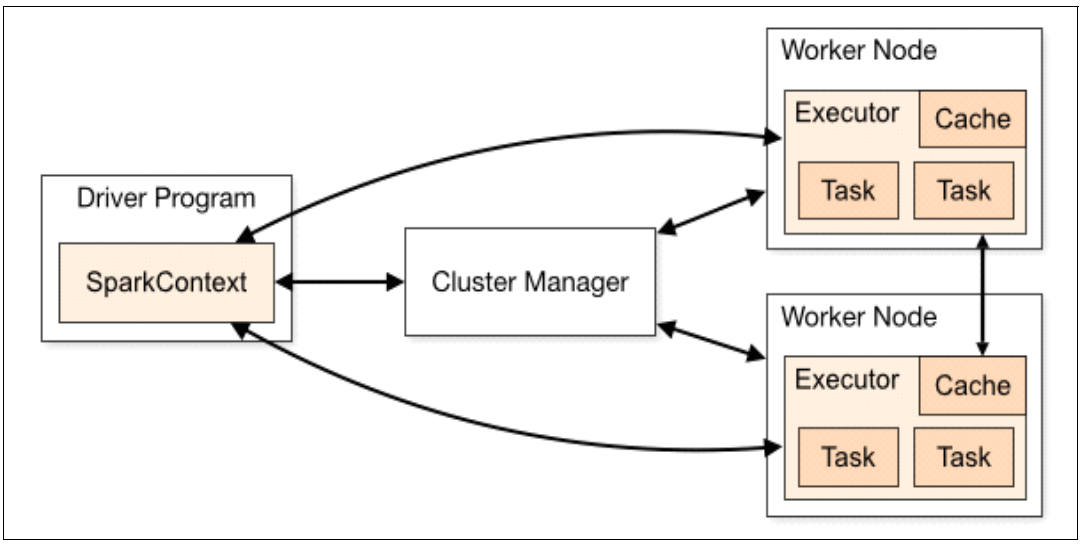


Figure 2-2 Components of a Spark cluster

Driver program

The Spark Driver is the program that declares the transformations and actions on RDDs of data, and submits requests to the master. It runs the **`main()`** function of your application that prepares the SparkContext. The SparkContext object, in your main program, orchestrates all of the activities within a Spark cluster. Each driver schedules its own tasks.

In Spark architecture, the Driver is a potential single point of failure. However, in the z/OS environment, the potential failures are mitigated by virtue of the high availability (HA) of the z/OS platform.

Cluster manager

Cluster manager is an external service for acquiring resources on the cluster. The Spark Context object connects to the cluster manager, which allocates resources across applications. The Spark applications run as independent sets of processes on the Spark cluster.

The cluster managers that Spark runs on provide facilities for scheduling across applications.

Worker node and Executor

Any node that can run application code in the cluster is a worker node. An Executor is a process that is launched for an application on a worker node, which runs tasks and keeps data in memory or direct access storage device (DASD) storage across them.

Each application gets its own executor processes, which stay active during the whole run of the application and run tasks in multiple threads. Tasks from different applications run in different Java virtual machines (JVMs). This has the benefit of isolating applications from each other on both the driver side and the executor side. However, the data cannot be shared across different Spark applications without writing the transformed data to an external persistent storage system.

2.1.3 Monitoring

Each driver program has a web UI, typically on port 4040, that displays information about running tasks, executors and storage usage. Simply point your web browser to `http://<driver-node>:4040` to access this UI. You can also use other monitoring options that you are familiar with in the z/OS environment.

Figure 2-3 shows a sample screen layout of this UI for Spark jobs.

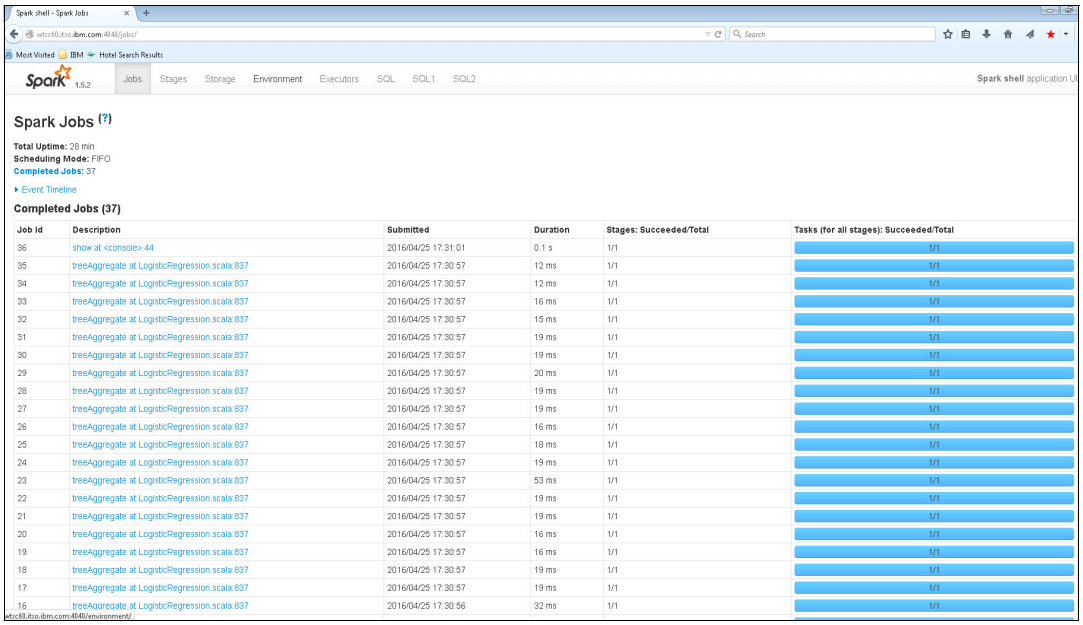


Figure 2-3 Monitoring Spark jobs from the web user interface

When you double-click **Description** and expand the directed acyclic graph (DAG) visualization, you see a screen layout similar to the one shown in Figure 2-4.

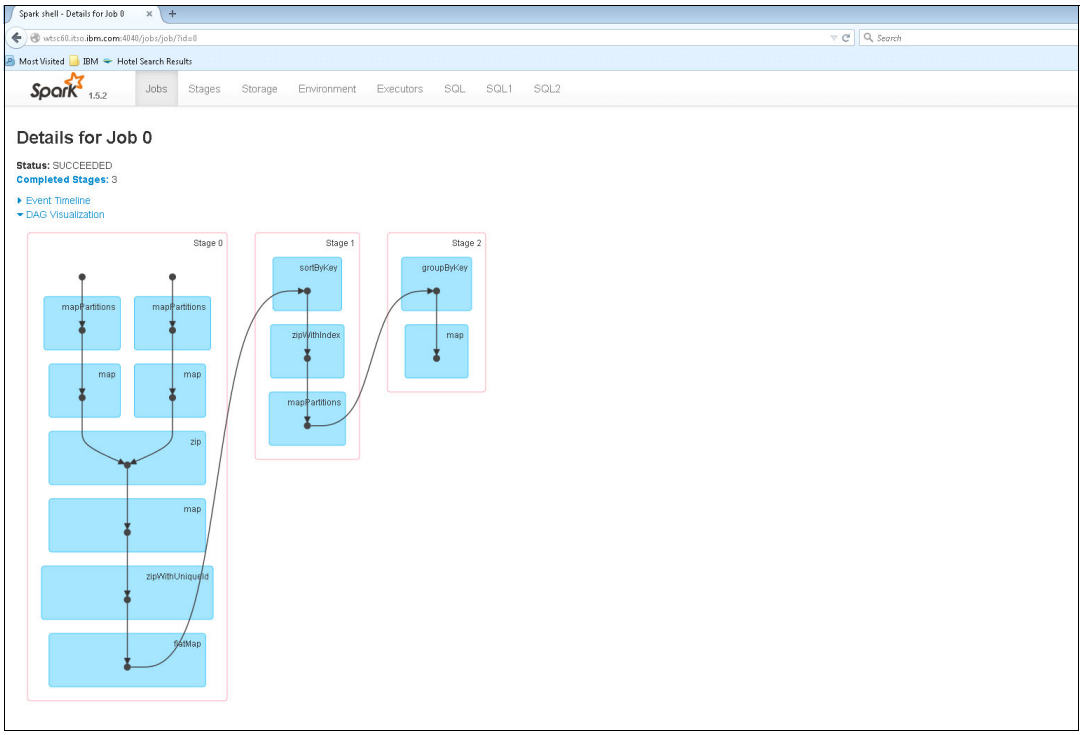


Figure 2-4 Monitoring user interface with DAG visualization

You can expand the **Event timeline** to see the screen layout shown in Figure 2-5.

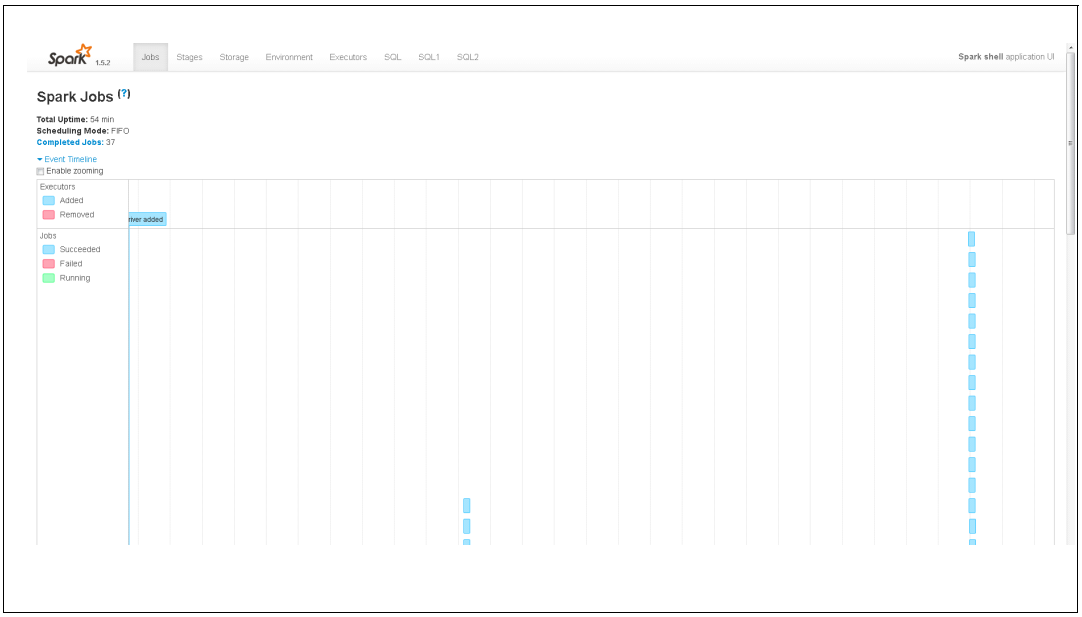
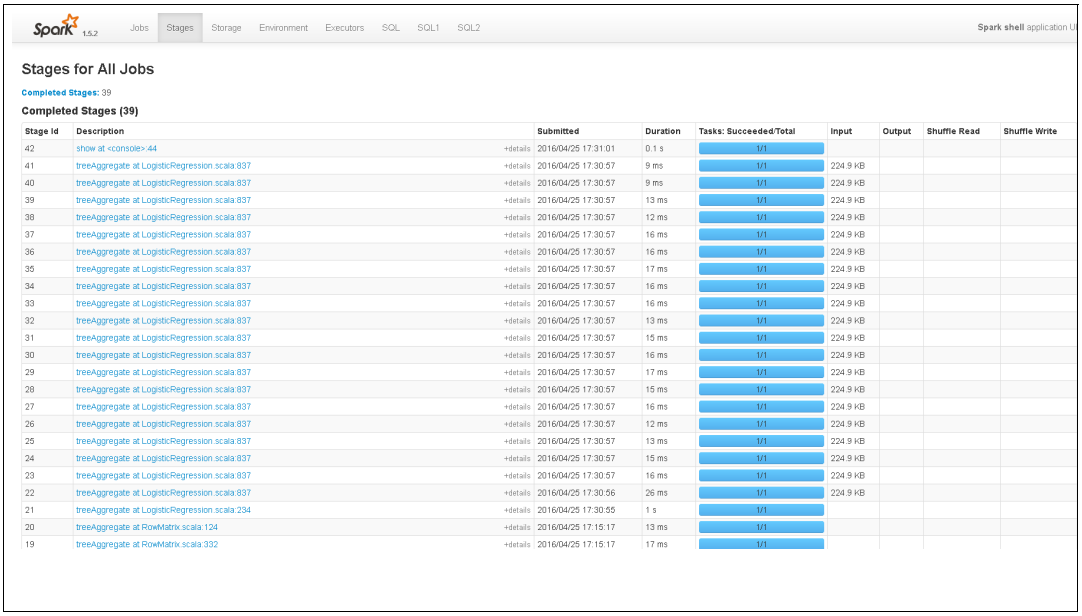


Figure 2-5 Monitoring user interface with Event Timeline

On the top portion of the web user interface, there are different tabs that take you to other monitoring screens. For instance, if you click **Stages**, you see a screen layout similar to the one depicted in Figure 2-6. This screen also enables you to drill down further.



Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
42	show at <console>:44	+details	2016/04/25 17:31:01	0.1 s	1/1				
41	treeAggregate at LogisticRegression.scala:837	+details	2016/04/25 17:30:57	9 ms	1/1	224.9 KB			
40	treeAggregate at LogisticRegression.scala:837	+details	2016/04/25 17:30:57	9 ms	1/1	224.9 KB			
39	treeAggregate at LogisticRegression.scala:837	+details	2016/04/25 17:30:57	13 ms	1/1	224.9 KB			
38	treeAggregate at LogisticRegression.scala:837	+details	2016/04/25 17:30:57	12 ms	1/1	224.9 KB			
37	treeAggregate at LogisticRegression.scala:837	+details	2016/04/25 17:30:57	16 ms	1/1	224.9 KB			
36	treeAggregate at LogisticRegression.scala:837	+details	2016/04/25 17:30:57	16 ms	1/1	224.9 KB			
35	treeAggregate at LogisticRegression.scala:837	+details	2016/04/25 17:30:57	17 ms	1/1	224.9 KB			
34	treeAggregate at LogisticRegression.scala:837	+details	2016/04/25 17:30:57	16 ms	1/1	224.9 KB			
33	treeAggregate at LogisticRegression.scala:837	+details	2016/04/25 17:30:57	16 ms	1/1	224.9 KB			
32	treeAggregate at LogisticRegression.scala:837	+details	2016/04/25 17:30:57	13 ms	1/1	224.9 KB			
31	treeAggregate at LogisticRegression.scala:837	+details	2016/04/25 17:30:57	15 ms	1/1	224.9 KB			
30	treeAggregate at LogisticRegression.scala:837	+details	2016/04/25 17:30:57	16 ms	1/1	224.9 KB			
29	treeAggregate at LogisticRegression.scala:837	+details	2016/04/25 17:30:57	17 ms	1/1	224.9 KB			
28	treeAggregate at LogisticRegression.scala:837	+details	2016/04/25 17:30:57	15 ms	1/1	224.9 KB			
27	treeAggregate at LogisticRegression.scala:837	+details	2016/04/25 17:30:57	16 ms	1/1	224.9 KB			
26	treeAggregate at LogisticRegression.scala:837	+details	2016/04/25 17:30:57	12 ms	1/1	224.9 KB			
25	treeAggregate at LogisticRegression.scala:837	+details	2016/04/25 17:30:57	13 ms	1/1	224.9 KB			
24	treeAggregate at LogisticRegression.scala:837	+details	2016/04/25 17:30:57	15 ms	1/1	224.9 KB			
23	treeAggregate at LogisticRegression.scala:837	+details	2016/04/25 17:30:57	16 ms	1/1	224.9 KB			
22	treeAggregate at LogisticRegression.scala:837	+details	2016/04/25 17:30:56	26 ms	1/1	224.9 KB			
21	treeAggregate at LogisticRegression.scala:234	+details	2016/04/25 17:30:55	1 s	1/1				
20	treeAggregate at RowMatrix.scala:124	+details	2016/04/25 17:15:17	13 ms	1/1				
19	treeAggregate at RowMatrix.scala:332	+details	2016/04/25 17:15:17	17 ms	1/1				

Figure 2-6 Monitoring web user interface to display Stages for all Jobs

Similarly, the tab that is labeled **Executors** resembles the screen layout shown in Figure 2-7.

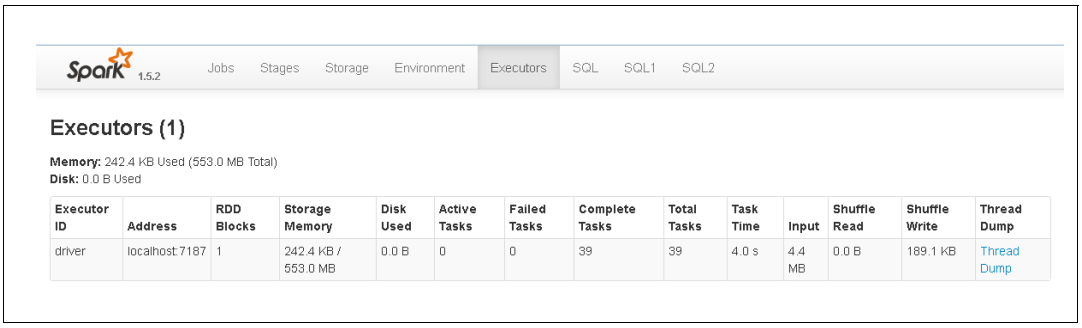


Figure 2-7 Monitoring web user interface to display Executors

The tab that is labeled **Environment** lists all of the environmental information that pertains to your Spark on z/OS installation.

The **Storage** tab is used to monitor the storage consumption of your Spark application while it is running. See Figure 2-8 for a sample display of memory usage.

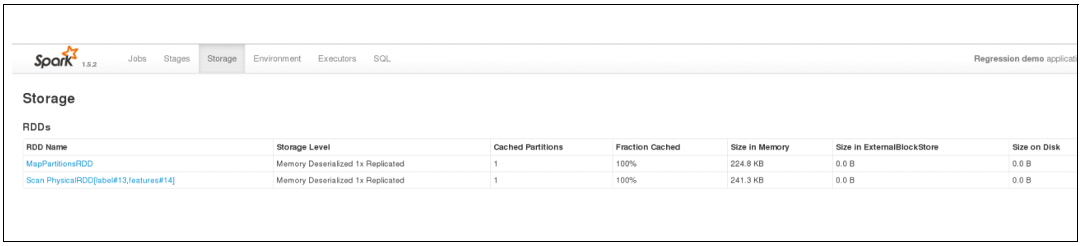


Figure 2-8 Monitoring web user interface to display storage usage

2.1.4 Spark and Hadoop

In general, Hadoop's MapReduce processing is disk-dependent, where Spark relies on in-memory capabilities, which can result in near real-time capabilities for Spark. Spark comes with a set of modules for streaming, SQL, graph, and machine learning, whereas Hadoop has an ecosystem of other tools to support these (for example, Storm, Hive, Giraph, and Mahout), but which have to be integrated separately.

Because Hadoop is not supported on z/OS, there's no Hadoop native library built for z/OS but, Spark has Hadoop dependencies, and it will try to load libhadoop - so it switches to use Java class implementation of the native methods.

2.2 Mainframe Data Services for IBM z/OS Platform for Apache Spark

Mainframe Data Services for IBM z/OS Platform for Apache Spar (MDSS) enables data from multiple, disconnected sources on z/OS to be virtually integrated into a single, logical data source, which can then be imported into a Spark DataFrame for programming complex analytics.

With MDSS, there are several major advantages:

- Support of very complex SQL
- High performance through IBM System z Integrated Information Processor (zIIP) usage and parallelism
- Low Total Cost of Ownership (TCO)
- Ease of use: Consume non-relational data as if it were relational
- Parallelism of Spark data access, within SQL and across partitions (transparent to the application)
- Support of SQL standards much richer than “vanilla” SparkSQL (can support SQL92 and SQL99 whereas standard SparkSQL is a subset of HiveQL)
- Access to many different data sources not generally supported by SparkSQL (VSAM, ADABAS, PS, PDSE, SMF, SYSLOG, and Logstream) in addition to DB2 and IMS
- Very granular integration with System Authorization Facility (SAF) interfaces for security configuration to suit needs
- Full response time or throughput goals for resource management with Workload Manager (WLM)
- Leverage of z/OS and IBM z13™ specific features:
  - IBM zEnterprise Data Compression (zEDC), large page support, SMT2, and so on
  - The total cost of ownership (TCO) and high zIIP and ease of use

MDSS is one of the core components of Apache Spark for z/OS, and highlights its capabilities:

- Data Mapping

  MDSS maps data from the native representation (for example, packed decimal) to relational format, thus making data sources uniformly accessible as virtual tables.

  The transformation between the native representation and the relational layout is described by a map that is unique to each data source. The map is generated, via Data Server Studio (DSS) tooling, from the copybook that describes the native representation.

- Caching

  In much the same way as a video streaming application, MDSS performs in-memory caching of data that is being read from the data sources until the application is ready to use it. The size of the in-memory buffer can be controlled by the user, using a parameter on the connection Uniform Resource Locator (URL).

- Map/Reduce Query Optimization

  Based on knowledge of the physical organization of the data source, and of the partitioning keys in a query, MDSS can spread the work across multiple threads and even multiple instances.

- Parallel input/output (I/O)

  Internally, MDSS performs I/O operations in parallel, by starting multiple service request blocks (SRBs) for data access and communications.

2.2.1 Virtual tables

A virtual table in MDSS represents the data that you want to access. It represents the data in relational, or tabular, format, much like a table in a database system, such as IBM DB2.

MDSS supports the following data sources:

- DB2 for z/OS tables and views
- IBM Information Management System database (IBM IMS DB)
- Virtual Storage Access Method (VSAM) key-sequenced data set (KSDS) and entry-sequenced data set (ESDS)
- Physical sequential (PS) data sets
- Distributed Resource Data Access (DRDA) data sources, including DB2 for Linux, UNIX, and Windows (DB2 LUW), and Oracle
- Adabas

Virtual tables are not required for relational data sources, unless there is a need to join your relational data with non-relational or other relational data sources from different platforms.

2.2.2 Virtual views

You may create virtual views to complete the following tasks:

- Refine data using SQL functions or casts
- Subset data using where clause
- Join two or more virtual tables

Much like a view in a relational database system, a virtual view in MDSS is essentially an alias for a query. A view is useful to present data from an existing table (in this case, an existing virtual table) in a different layout, or with additional columns. Also, and more importantly, it can be used to join two or more other tables. The beauty of using virtual views is that you can join relational data (for example, from DB2 for z/OS) with non-relational data, such as records in a VSAM file.

2.2.3 SQL queries

MDSS supports most of the SQL 92 standard, meaning that you can access your VSAM, sequential and IMS data with familiar SQL syntax. This includes support for many column functions (for example, DAYOFWEEK, DAYOFYEAR, IFNULL, JULIANDAY, LTRIM, LOWER, NULLIF, RTRIM, SUBSTR, and TRIM) and aggregate functions (for example, AVG, MAX, MEDIAN, MIN, and SUM). Also, you can use GROUP BY and ORDER BY clauses.

2.2.4 MDSS JDBC driver

Applications running on the JVM communicate with the MDSS server through the MDSS Java Database Connectivity (JDBC) driver. In JDBC terms, this driver is a Type 4 driver, that is, it is written in pure Java and does not need any native code. It connects directly to the MDSS server, and communicates with it using its own network protocol.

The MDSS JDBC driver supports connection properties that control the driver's operation.

The JDBC URL for a connection using the MDSS driver has the following format:

```
jdbc:rs:dv//<hostname>:<port>;<prop1=val1>;<prop2=val2>
```

In this example, the following values apply:

hostname The host name of the z/OS LPAR where the MDSS started task runs
port The port number on which the MDSS started task is listening
propn=valn A list of semicolon-separated property settings

Each available property has a full name and an alias (short name). The full set of connection properties is documented in *IBM z/OS Platform for Apache Spark Solutions Guide*, SC27-8452. In the following table (Table 2-2 on page 24) we give a brief overview of the most important properties and when you may want to use them.

Tip: When typing URLs from a command line prompt (for example, as an argument to a **spark-submit** command, be aware that some characters in the URL (especially the semicolon) may be interpreted and consumed by the shell. To avoid this, put the URL into single or double quotes.

Table 2-2 Important MDSS driver properties

Property name	Description	Recommendations
CompressionLevel	Sets the compression level for communication between the driver and the MDSS started task. On z/OS, MDSS can use zEDC. Allowed values: -1 (no compress)	We recommend to omit this parameter, resulting in the default setting of -1 (no compression) as long as driver and MDSS server are co-located.
LogConfiguration	Sets the log4j2 log configuration file.	Useful for logging and tracing. See the product documentation for details.
MapReduce	Enables distribution of queries over multiple servers.	
MapReduceConnectionCount (MRCC)	Sets the number of parallel connections for query distribution.	As a starting point, set MRCC to the number of zIIP engines configured on the LPAR.
MaximumBufferSize (MXBU)	Sets the communication buffer size in bytes, kilobytes, or megabytes.	
SubSystem (SUBSYS)	Sets the name of the DB2 subsystem, for connections to DB2 through the MDSS driver.	

2.2.5 IBM z/OS Platform for Apache Spark Interface for CICS/TS

IBM z/OS Platform for Apache Spark Interface for IBM Customer Information Control System Transaction Server (IBM CICS/TS), is a licensed add-on component of the IBM z/OS Platform for Apache Spark product suite. This interface offers quick and easy access to CICS/TS, making CICS data directly available to non-mainframe users.

The Data Virtualization Interface for CICS/TS provides a tool for making quick and easy CICS/TS queries. IBM z/OS Platform for Apache Spark, in conjunction with this interface, allows existing and new CICS programs to be rapidly integrated into client/server applications with minimum modification to the code.

The Data Service server component of IBM z/OS Platform for Apache Spark connects to CICS by using the IBM External CICS Interface (EXCI). In IBM z/OS Platform for Apache Spark, a CICS program behaves like a stored procedure. Applications can call the procedure by using industry-standard mechanisms, passing in the input data as part of the call. The CICS program returns data by using the communication area (COMMAREA) to the Data Service server. The Data Service server, through the Data Mapping Facility, formats the display of the returned results.

2.3 Spark SQL

Spark SQL is Spark's module for working with structured data, using an abstraction called DataFrame. A DataFrame is basically an RDD with additional schema information. Much like RDDs, DataFrames can be manipulated using a domain-specific language for filtering, aggregating, mapping, and other transformations. Also, Spark SQL adds SQL language support, thus making it possible to query non-SQL data sources in SQL syntax.

Figure 2-9 highlights the fact that data from any z/OS data store can be manipulated using DataFrame API for Spark SQL.

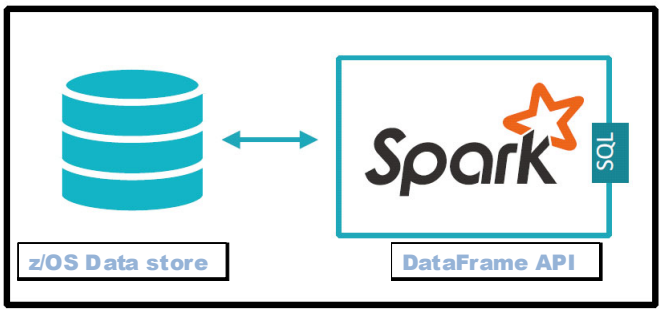


Figure 2-9 Spark SQL using DataFrame API abstraction on z/OS data

Spark SQL allows you to complete the following tasks:

- Integrate data from any z/OS data sources (DB2, VSAM, IMS, Adabas, and so on) with non-z/OS data. Example illustrates data read from VSAM, IMS, and DB2 Data sources. See Chapter 4, "Spark application development on z/OS" on page 61 for more details.
- Run SQL queries over integrated data.
- Intermix SQL with complex analytics algorithms to use the SystemML, SystemG, MLlib and GraphX functionality.
- Easily write RDDs or DataFrames to DB2 for z/OS or to any other JDBC-supported databases, for subsequent use in online transaction processing (OLTP) transactions.

Figure 2-10 shows the Spark SQL Programming Interface.

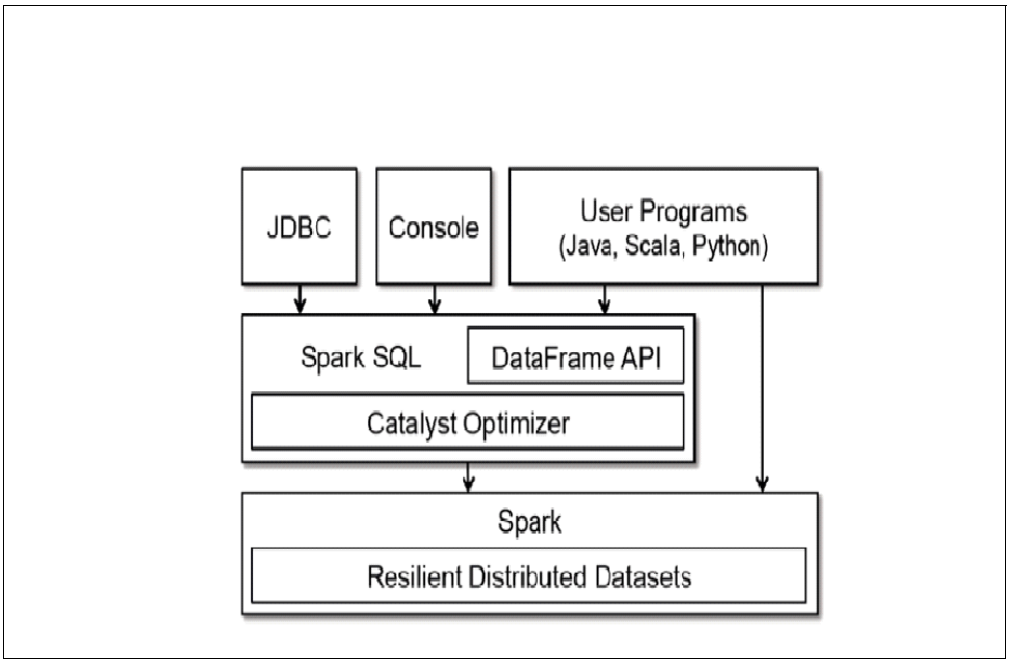


Figure 2-10 Spark SQL Programming Interface

2.3.1 Reading from z/OS data source into a DataFrame

You may use either the standard JDBC or the MDS driver, to read data from a z/OS data source into a DataFrame. You may also leverage IBM Analytics Accelerator to get operational data. You may find numerous examples on this topic in Chapter 4, “Spark application development on z/OS” on page 61.

Spark SQL (data abstraction using DataFrame API) accesses data from such a federated relational structure with ease but, at the same time the performance is much better on performing complex analytics.

2.3.2 Writing DataFrame to a DB2 for z/OS table using saveTable method

Example 2-1 illustrates how to code a simple write from DataFrame to a DB2 for z/OS table. If the schema associated with the input DataFrame matches the table definition, you can use the **`saveTable`** method to append data to that table.

Example 2-1 DataFrame to DB2 for z/OS table

```
def saveTable(df: org.apache.spark.sql.DataFrame, url: String, table: String,
properties: java.util.Properties): Unit

val u2 = "jdbc:db2://wtsc60.itso.ibm.com:38050/DB11"

org.apache.spark.sql.execution.datasources.jdbc.JdbcUtils.saveTable(custDF, u2,
"LPRES2.BANK_CUST", prop)
```

Spark SQL allows you to write relational queries that are expressed in either SQL, HiveQL, or Scala to be executed using Spark. The DataFrame consists of row objects and a schema that describes the type of data in each column of the row. You can think of this as a table in a traditional relational database.

You create DataFrames from existing RDDs, a Parquet file, a JavaScript Object Notation (JSON) dataset, or using HiveQL to query against any z/OS data source.

Both RDDs and DataFrames share the same memory space in z/OS.

Spark z/OS has unique functionality to access data across wide variety of environments, and supports the SQL 92 and SQL 99 standards with high performance and flexibility.

2.4 Streaming

Spark Streaming is a Spark extension that enables processing of streaming data (data that is not of a relatively static nature but that arrives continuously), as illustrated by Figure 2-11 on page 28. Spark Streaming supports for immediate use several kinds of data sources:

- HDFS and other file-based streaming data sources

  This enables monitoring a directory where new files accumulate. A typical example would be first-failure data capture (FFDC) log data from IBM WebSphere Application Server.

- Flume

  Flume is a service and API to efficiently collect, aggregate, and move large amounts of streaming event data, for example, log data or Really Simple Syndication (RSS) feeds.

- Kafka

  Apache Kafka is a publish/subscribe messaging system.

- Twitter

  The Twitter support for Spark Streaming allows applications access to Twitter's stream of Tweet data.

- ZeroMQ

  ZeroMQ is a messaging library. It supports fast, asynchronous communication over several transport protocols. From an API perspective (though not from a quality-of-service perspective), it is comparable to messaging systems such as IBM MQ.

In addition, you can define your own custom data sources. For example, you could create a custom data source that browses messages on an IBM MQ message queue, and streams the message contents, the message metainformation, or both.

Figure 2-11 shows the Spark Streaming extension.

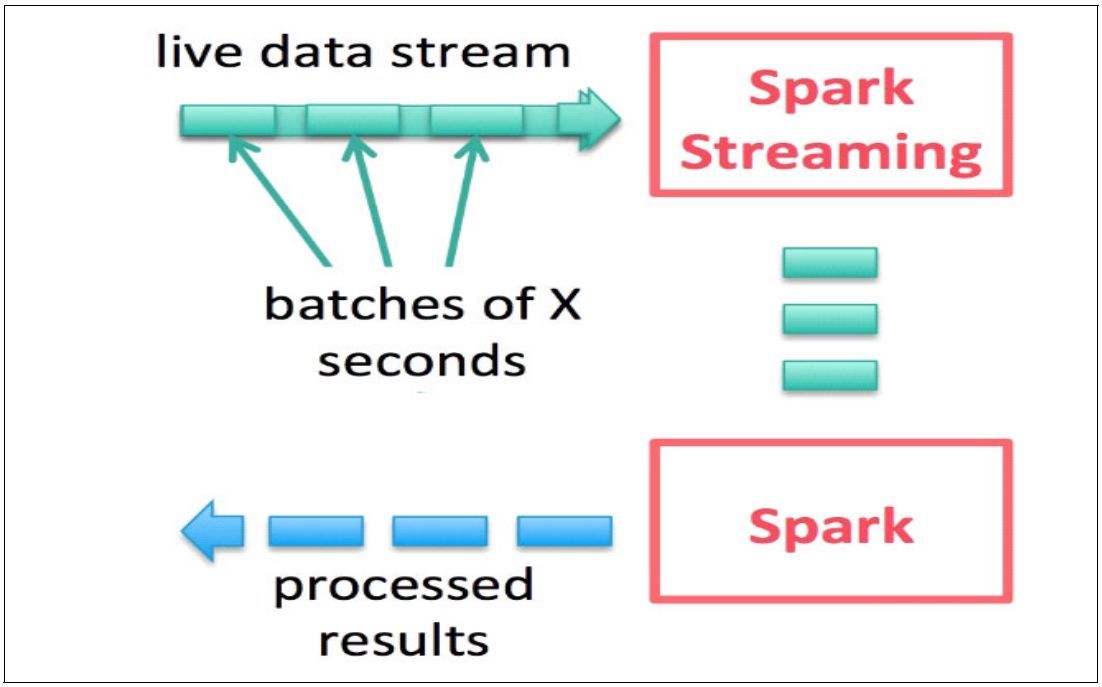


Figure 2-11 Spark Streaming

2.5 GraphX

Spark GraphX is Spark's API for performing computations on graphs. Much like the operations on RDDs, GraphX offers fundamental operations on graphs, such as joining and partitioning, and mapping. It also supports important algorithms on graphs, such as computing the PageRank graph for a given graph and computing the number of triangles passing through each vertex.

See more details on the Spark GraphX website:

http://spark.apache.org/graphx/

2.5.1 System G

IBM System G is comprehensive graph computing software for the big data portfolio. It is an integral full-stack solution for analytics, offering graph processing functions at all layers, such as property graph storage, run time, analytics, and visualization. System G is enabled to run on Linux on z. See more detail at the IBM System G website:

http://systemg.research.ibm.com/

2.6 MLlib

This machine learning framework is on top of Spark Core, and implements a number of commonly used machine learning algorithms (for example, correlations, random data generation, classifications, regressions, decision trees, clustering, and so on).

Example 2-2 demonstrates MLlib features.

Example 2-2 Simple churn prediction using a logistic regression model

```
object RegressionDemo {

  case class ContractFeatures(contID: Int, features: Vector)

  def trainModel(trainingDF: DataFrame) = {
    val lr = new LogisticRegression()
      .setRegParam(0.1)
      .setMaxIter(30)
      .setThreshold(0.55)
      .setProbabilityCol("churnProbability")

    lr.fit(trainingDF)
  }

  def main(args: Array[String]) {

    /* Setup code omitted */

    val vDF = sqlContext.read.jdbc(mdssUrl, "CLIENT_PAY_SUMM", props)
      .select("CONT_ID", "CHURN", "AGE_YEARS", "ACTIVITY_LEVEL")
      .filter($"AGE_YEARS" < 50)

    val Array(trainingData, testData) = vDF.randomSplit(Array(0.7, 0.3))

    def toVector(row: Row) =
      Vectors.dense(row.getDecimal(2).doubleValue, row.getInt(3).doubleValue)

    val trainingDF = trainingData.map { row =>
      LabeledPoint(row.getInt(1), toVector(row))
    }.toDF.cache()

    val testDF = testData.map { row =>
      ContractFeatures(row.getDecimal(0).intValue(), toVector(row))
    }.toDF.cache()

    val model = trainModel(trainingDF)
    model.transform(testDF)
      .select("contID", "churnProbability", "prediction")
      .show(100)
  }
}
```

2.7 Spark R

Spark R is an R package that provides a light-weight front-end to use Apache Spark from R. Spark R exposes the Spark API through the RDD class, and enables users to interactively run jobs from the R shell on a cluster. Rocket Software is in progress with a port of R to z/OS, and interested clients should contact either Rocket directly or IBM.

3

Installation and configuration

In this chapter, we describe the steps to install and customize the Spark server and associated components on IBM z/OS.

Specifically, this chapter describes the following topics:

- “Installing IBM z/OS Platform for Apache Spark” on page 30
- “The Mainframe Data Service for Apache Spark (MDSS)” on page 30
- “Installing workstation components” on page 44
- “Configuring Apache Spark for z/OS” on page 46
- “Verifying the installation” on page 52
- “Starting the Spark daemons” on page 54

3.1 Installing IBM z/OS Platform for Apache Spark

The IBM z/OS Platform for Apache Spark can be obtained from IBM Shopz.[1] It can be installed using the Custom-built Product Delivery Offering (CBPDO), SystemPac, or ServerPac method. The installation method used in this book is CBPDO. It uses the SMP/E for z/OS commands **RECEIVE**, **APPLY**, and **ACCEPT** to make the installation. After the general availability (GA) code is downloaded, see *Program Directory for IBM z/OS Platform for Apache Spark*, 5655-AAB for installation instructions.

The default Target Library name is `AZK.**`. To avoid making changes to SMP/E managed datasets, we copied and created a new high-level qualifier (HLQ) named `SPARK.**`.

The hardware and software requirements for IBM z/OS Platform for Apache Spark are shown in the following lists.

Hardware requirements:

- IBM z13
- z13s
- IBM zEnterprise EC12
- zEnterprise BC12

Software requirements:

- z/OS V2.1, or later
- IBM 64-bit SDK for z/OS, Java Technology Edition, Version 8 Service Refresh 2
- Bourne Again Shell (bash) version 4.2.53, or later

3.2 The Mainframe Data Service for Apache Spark

IBM z/OS Mainframe Data Service for Apache Spark (MDSS) is a started task that runs in the z/OS address space. It provides a passage or channel between Spark on z/OS and the data sources, such as IBM DB2, IBM Information Management System (IBM IMS), IBM Virtual Storage Access Method (IBM VSAM), partitioned data set (PDS), and sequential files residing in z/OS. It enables Spark applications to use data from mainframe and non-mainframe data sources while supporting simultaneous threads from Spark on IBM for z/OS, Spark on Linux on IBM z Systems, and Spark on Linux.

3.2.1 Installing the MDSS started task

This section describes the installation and configuration of MDSS and its parameter member *hlq*`.SAZKEXEC(AZKSIN00)`. You may notice that these installation steps are in a different order than those given in the *IBM z/OS Platform for Apache Spark Installation and Customization Guide,* SC27-8449. In our environment, we created the started task first in order to see what we were configuring for later.

1. Create the started task job control language (JCL):
   a. First, you need to create the JCL procedure for the MDSS started task. A sample JCL procedure is provided in *hlq*`.SAZKCNTL(AZK1PROC)`. Follow the instructions provided in the comments section to customize it to fit your environment. Our customizations are shown in Example 3-1 on page 33.

[1] https://www-304.ibm.com/software/shopzseries/ShopzSeries_public.wss

b. Make the following adjustments, as necessary:

- If DB2 for z/OS database is being used as a data source, add `DB2LIB='DB2SSID'` to the **`proc`** statement. MDSS can pull the DB2 table information from the DB2 Catalog.
- If IMS DB is being used as a data source, also add `IMSLIB='&IMSHLQ.SDFSRESL'` to the main **`proc`** statement. MDSS can pull the IMS information from the IMS RESLIB.

Example 3-1 Customized JCL procedure for the MDSS started task

```
//*********************************************
//AZKS         PROC SSID=AZKS,
//             OPT=INIT,
//             TRACE=B,
//             MSGPFX=AZK,
//             REG=8M,
//             MEM=32G,
//             HLQ='SPARK',
//             DB2LIB='DB11T',
//             IMSLIB='IMS13A.SDFSRESL'
//*********************************************
```

c. Copy the customized procedure member to your system's PROCLIB concatenation.

2. Create trace and checkpoint datasets:

a. Submit this sample job *hlq*`.SAZKCNTL(AZKDFDIV)` to create datasets for TRACE BROWSE (*hlq*`.AZKS.TRACE`) and Global Variable checkpoints (*hlq*`.AZKS.SYSCHK1`).

b. Make changes to the high-level qualifier, subsystem identifier (SSID), and add the 'VOL(VOLSER)' parameter if storage management subsystem (SMS) is not used.

3. Create datasets to store your message addressing property (MAP) definitions:

a. Submit this sample job *hlq*`.SAZKCNTL(AZKGNMP1)` to create map datasets to store the data source mapping definitions. This dataset is then concatenated to the AZKMAPP DD statement of the AZK1PROC start procedure.

b. Use the newly created dataset as the first concatenation and make sure that the server and user have both read and write authorization. The product-provided `'AZK.SAZKMAP'` should always be concatenated last and authorized as read only.

Update: Authorized program analysis report (APAR) PI63427 makes some changes to AZKGNMP1:

- The map created for application use is now called HLQ2.AZKS.SAZKMAP, where AZKS is typically the AZK subsystem name.
- The job also now creates a HLQ2.AZKS.SAZKEXEC dataset for storing the AZKxIN00. This is so that SMP/E does not update this dataset after it has been updated for the server's configuration. The AZKSIN00 is copied into this dataset as part of that job, and that member should be used later for configuring the server.

  This dataset is also now pointed to in the AZK1PROC JCL in the SYSEXEC DD allocation.
- 3) This job also now creates a complete set of Event Facility data sets, which should be done for each SPARK MDS server created, and copies the sample rules from the installation:
  - HLQ1.SAZKXATH
  - HLQ1.SAZKXCMD
  - HLQ1.SAZKXEXC
  - HLQ1.SAZKXSQL
  - HLQ1.SAZKXTOD
  - HLQ1.SAZKXVTB

SMF and SYSLOG maps are predefined and stored in AZK.SAZKSMAP. If SMF and SYSLOG need to be used as a data source, then it should also be concatenated (Example 3-2).

Give the data set plenty of space. The default value only accommodated a few map definitions, and gave an E37 (out of space) error.

Example 3-2 Data Mapping Facility (DMF) DDNAME

```
//*******************************************************************
//*      THIS DDNAME IS REQUIRED FOR DATA MAPPING FACILITY           *
//*******************************************************************
//AZKMAPP      DD   DISP=SHR,DSN=HLQ2.AZKS.SAZKMAP
//             DD   DISP=SHR,DSN=AZK.SAZKSMAP
//             DD   DISP=SHR,DSN=AZK.SAZKMAP
//*******************************************************************
```

Note: If an empty map data set is specified, the mainframe data service server refuses to start (Example 3-3).

Example 3-3 Error message from MDSS server when map data set is empty

```
AZK3134I SPARK.MAP Library directory empty (DDN=AZKMAPP)
AZK3782S MAP FILE AZKMAPP DIRECTORY READ FAILED RC=8
AZK0120I SEF INITIALIZATION FAILED DUE TO PREVIOUSLY NOTED ERRORS - SERVER WILL
SHUT DOWN
AZK3750H SEF initialization failed - SEF subtask terminating
AZK4506H SEFFULL  subtask terminating
```

4. Install Event Facility rules.

   Submit sample job *hlq.*SAZKCNTL(AZKEXSWI) to install a set of precompiled Event Facility (SEF) rules. These rules are used by Data Service Server for development and administration operations. It performs a **RECEIVE** on the *hlq.*SAZKOBJX(AZKOBJ) member, which will create the *hlq.*AZKS.SAZKXOBJ dataset that contain the rules. These rules will then be used by the *hlq.SAZKEXEC(*AZKSIN00) parameter member.

   Update: This job was slightly changed by APAR PI63427. It now uses a slightly different high-level qualifier (HLQ) for this dataset:

```
//*      1) Change "?HLQ1?" to the high level qualifier for the      *
//*         installation data sets.                                  *
//*                                                                  *
//*      2) Change "?HLQ2?" to the high level qualifier that is used  *
//*         for the runtime libraries.                               *
//*******************************************************************
//GENSWI   EXEC PGM=IKJEFT01,DYNAMNBR=20
//SAZKXOBJ DD  DISP=SHR,DSN=?HLQ1?.SAZKXOBJ(AZKOBJ)
//SYSPRINT DD  SYSOUT=*
//SYSTSPRT DD  SYSOUT=*
//SYSTSIN  DD  *
  DELETE '?HLQ2?.AZKS.SAZKXOBJ'
  RECEIVE INDDNAME(SAZKXOBJ)
          DSNAME('?HLQ2?.AZKS.SAZKXOBJ')
/*
```

   Failure to do this will lead to errors in the workstation graphical user interface (GUI) when trying to use the mapping dialogs. They will also show up in the MDSS started task's JESMSGLG (Example 3-4).

Example 3-4 Error in JESMSGLG when SEF rules are not installed

```
IKJ56228I DATA SET AZK.SAZKOBJ NOT IN CATALOG OR CATALOG CAN NOT BE ACCESSED
AZK3126E DYNAMIC ALLOCATION FAILED, RC=4, ERROR CODE=X'1708', INFO
CODE=X'0002',, DDNAME=, DSNAME=AZK.SAZKOBJ
AZK3901E PRE-CHECK XO DATASET OPEN OF AZK.SAZKOBJ FAILED, RC=X'00000010'
AZK3986E FILE SWIDATA  DEFINITION REJECTED BY SEF - DEFINITION OF RELATED XO
DATASET AZK.SAZKOBJ FAILED
AZK3983E RULESET SWIRULE  DEFINITION REJECTED BY SEF - BECAUSE SWI FACILITY
ACTIVATION HAS BEEN CANCELLED
```

In the Data Studio Client, an `HTTP Server status code: 500 - Internal Server Error` occurred when trying to create Virtual tables or create virtual source libraries (Figure 3-1).

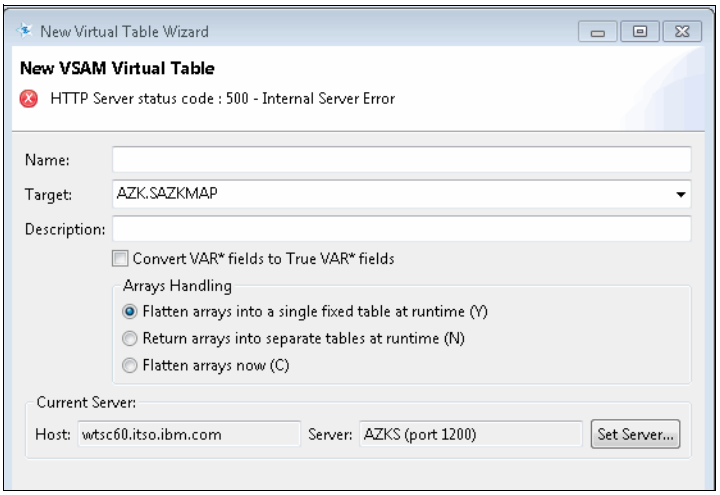


Figure 3-1 HTTP Server status code: 500 - Internal Server Error if SAZKOBJ not allocated

5. Create a user ID to run the started task.

   Submit sample job `hlq.SAZKCNTL(AZKRAVDB)` to create the user ID required to run the started task AZKS. It also defines a STARTED class resource profile and associate the user ID with the Data Service started task. Permits are also issued to various Workload Manager (WLM) facilities.

6. Authorized program facility (APF): Authorize SAZKLOAD:

   a. The load data set, `hlq.SAZKLOAD`, must be APF authorized. Issue the following command to add it dynamically:

      ```
      SETPROG APF,ADD,DSN=hlq.SAZKLOAD,VOL='volser'
      ```

   b. Create an entry in the system PARMLIB member (`PROGxx`) to make the change. permanent.

7. Configure the MDSS configuration Member `hlq.SAZKEXEC(AZKSIN00)`.

   This is the Data Service server configuration member. It is a Restructured Extended Executor (REXX) program that is used to set product parameters and define links and databases. The SYSEXEC DD statement on the MDSS started procedure points to the location of the data set. Naming of this member must be the subsystem ID followed by “IN00”, for example, `AZKSIN00`.

   Read the comments section for a full explanation on the function of this configuration member.

   This book uses data sources from DB2, IMS, VSAM, SYSLOG, and SMF. In the following sections, we explain the configuration for accessing these data sources. No additional configuration has to be done for VSAM and sequential files. SYSLOG and SMF mapping definitions can be found in hlq.SAZKSMAP. Virtual table rule events must be enabled by adding this line to the AZKSIN00 member:

   ```
   MODIFY PARM NAME(SEFVTBEVENTS) VALUE(YES)
   ```

   Update: As of APAR PI63427, the data set is now HLQ2.AZKS.SAZKEXEC. See note above in step 3 from job AZKGNMP1.

3.2.2 Configuring access to DB2

Two methods are described in *IBM z/OS Platform for Apache Spark Solutions Guide*, SC27-8452 for configuring DB2. They are either Distributed Resource Data Access (DRDA) or Resource Recovery Services Access Facility (RRSAF). Only DRDA connections are supported from MDS to DB2.

When you run DB2 on an IBM System z Integrated Information Processor (zIIP), DRDA is suggested, because it allows a higher percentage of DB2 work to run in service request block (SRB) mode and be offloaded to a zIIP specialty engine. This might bring about lower total cost. If no zIIP engines are configured so general purpose processors (CPs) are used, then RRSAF is recommended. The environment we set up for testing for this publication does use zIIP, therefore we discuss the DRDA method of configuration.

To configure DB2, complete the following steps:

1. Go to the section of AZKSIN00 member commenting on enabling DRDA. Change **`If DontDoThis`** to **`DoThis`**. Update the NAME, LOCATION, PORT, and IP addresses that match your DB2 system (Example 3-5).

   Example 3-5 AZKSIN00 DB2 configuration

```
if DoThis then do
   "MODIFY PARM NAME(TRACEOEDRDARW)       VALUE(YES)"
   "MODIFY PARM NAME(CLIENTMUSTELECTDRDA)  VALUE(NO)"
   "MODIFY PARM NAME(DRDASKIPWLMSETUP)     VALUE(NO)"
   "MODIFY PARM NAME(DRDASKIPZSERVICES)    VALUE(YES)"
   "MODIFY PARM NAME(DRDAFORLOGGINGTASK)   VALUE(NO)"
   "MODIFY PARM NAME(DRDAPACKAGEPREFIX)    VALUE(DV)"
   "MODIFY PARM NAME(DRDACOLLECTIONID)     VALUE(NULLID)"
   "DEFINE DATABASE TYPE(MEMBER)"                    ,
                  "NAME(DB11)"                       ,
                  "LOCATION(DB11)"                   ,
                  "DDFSTATUS(ENABLE)"                ,
                  "PORT(38050)"                      ,
                  "IPADDR(x.xx.x.xx)"                ,
                  "CCSID(37)"
```

2. If you are not sure about the values for your DB2 system, issue a DB2 **`DISPLAY DDF`** command (Example 3-6) or look in the DB2 master address space job log.

   Example 3-6 Output from -DIS DDF

```
DSNL080I  -DB11 DSNLTDDF DISPLAY DDF REPORT FOLLOWS: 871
DSNL081I STATUS=STARTD
DSNL082I LOCATION           LUNAME            GENERICLU
DSNL083I DB11               USIBMSC.SCPDB11   -NONE
DSNL084I TCPPORT=38050 SECPORT=38051 RESPORT=38052 IPNAME=-NONE
DSNL085I IPADDR=::x.xx.x.xx
DSNL086I SQL    DOMAIN=wtsc60.itso.ibm.com
DSNL105I CURRENT DDF OPTIONS ARE:
DSNL106I PKGREL = COMMIT
DSNL099I DSNLTDDF DISPLAY DDF REPORT COMPLETE
```

   See the *IBM z/OS Platform for Apache Spark Solutions Guide*, SC27-8452 for a list of parameters and a description of configuring support for DRDA.

3. There are two DB2 bind jobs that need to be submitted before any DB2 request is issued. These jobs bind both DRDA and RRSAF into packages within the DB2 subsystem. Even though we are only using DRDA, it is advised that both packages be bound.

   Edit sample job *hlq*.SAZKCNTL(AZKBINDD). Follow the comments to edit the variables to fit your environment. Make sure that the user ID submitting the job has proper DB2 authority. The user ID we used has DB2 SYSADM authority. The final step on GRANT can be run separately using Structure Query Language (SQL) Processor Using File Input (SPUFI).

   hlq.SAZKCNTL(AZKBINDC) is the job used to bind RRSAF packages to DB2. The following product plans are bound:

   – AZKC1010 bound using cursor stability.
   – AZKC1010 bound using repeatable read.
   – AZKC1010 bound using read stability.
   – AZKC1010 bound using uncommitted read.

   DSN3@SGN exit enables the server to use DB2 authority that was granted through secondary DB2 authorization IDs. Install this if it is not already done. It is normally placed in the DB2 SDSNEXIT library.

4. Note that *ssid*.DIST profile READ authority should be assigned to the user that is connecting to DB2 via DRDA. To enable passticket logon processing, a sample job is provided for IBM Resource Access Control Facility (IBM RACF), CA ACF2, and CA Top Secret (TSS) security server. Because our environment uses IBM RACF, we edited and then submitted *hlq*.SAZKCNTL(AZKRADB2).

3.2.3 Configuring access to IMS databases

Similar steps are taken to configure for IMS data source. Go to the section that describe IMS configuration. Alter If **`DontDoThis`** to **`DoThis.`** Update the IMSID and IMSDSNAME value to those that match your environment (Example 3-7).

Example 3-7 AZKSIN00 IMS configuration

```
/*-----------------------------------------------------------------*/
/* Enable IMS CCTL/DBCTL support                                   */
/*-----------------------------------------------------------------*/
if DoThis then
  do
   "MODIFY PARM NAME(DBCTL)               VALUE(YES)"
   "MODIFY PARM NAME(IMSID)               VALUE(IMSA)"
   "MODIFY PARM NAME(IMSDSNAME)           VALUE(IMS13A.SDFSRESL)"

   "MODIFY PARM NAME(IMSMINTHREADS)       VALUE(5)"
   "MODIFY PARM NAME(IMSMAXTHREADS)       VALUE(10)"
   "MODIFY PARM NAME(IMSNBABUFFERS)       VALUE(100)"
   "MODIFY PARM NAME(IMSFPBUFFERS)        VALUE(10)"
   "MODIFY PARM NAME(IMSFPOVERFLOW)       VALUE(10)"
   "MODIFY PARM NAME(TRACEIMSDLIEVENTS)   VALUE(NO)"
```

See the *IBM z/OS Platform for Apache Spark Solutions Guide*, SC27-8452 for the list of parameters and their descriptions.

3.2.4 The ISPF Panels

IBM z/OS Platform for Apache Spark package also includes the Interactive System Productivity Facility (ISPF) application (Example 3-8). ISPF panels are used to perform various tasks, such as mapping database definitions, managing remote users, performing server traces, event facility management, monitoring server activity, and server administrative functions. Some of these functions can also be done on the Data Service Studio. For more information, see Chapter 4, “Spark application development on z/OS” on page 61.

Example 3-8 Primary Option Menu of MDSS ISPF application

```
IBM z/OS Platform for Apache Spark - Primary Option Menu
Option  ===>


  Interface Facilities:                                    SSID    : AZKS
   1  ACI                                                 Version : 01.01.00
   2  Adabas            4  IMS                            Date    : 16/04/21
   3  DB2               5  VSAM/Sequential                Time    : 14:41

  Server Administration:
   A  Remote User       -  Manage Remote Users
   B  Server Trace      -  Server Trace Facility
   C  AZK Admin.        -  Manage Data Virtualization Server
   D  Data Mapping      -  Data Mapping Facility
   E  Rules Mgmt.       -  Event Facility Management
   F  Monitor           -  Monitor Server Activity
```

In order to access the Spark ISPF Primary Option Menu, complete the following steps:

1. First, edit the *hlq*.SAZKEXEC(AZK) member and replace the dataset name in the **`llib="hlq.SAZKLOAD"`** statement, as shown in Example 3-9.

   Example 3-9 Main procedure for the Spark ISPF Application

   ```
   /*-------------------------------------------------------------------*/
   /*                  M A I N   P R O C E D U R E                      */
   /*--+----1----+----2----+----3----+----4----+----5----+----6----+----*/
   arg parm
   address TSO

   llib = "SPARK.SAZKLOAD"

   address ISPEXEC
   "LIBDEF ISPLLIB DATASET ID('"llib"')"
   "SELECT CMD(AZK "parm")"

   address ISPEXEC
   "LIBDEF ISPLLIB"

   exit
   ```

2. Then, go to the ISPF command shell (option 6) and enter the following command:

   `EX 'hlq.SAZKEXEC(AZK)' 'SUB(SSID)'`

   SSID is the four-character subsystem name of the server instance. In our case, this is called *AZKS*.

Mapping VSAM datasets

While the preferred method of mapping would be using the Eclipse-based Studio (see “Defining the data mapping” on page 65), z Systems programmers sometimes prefer to use the ISPF panels to do this. So, to map VSAM datasets, complete the following steps:

1. Enter the Spark ISPF primary options menu. Select **5 VSAM/Sequential,** this will take you to the VSAM/Seq Data Mapping Facility panel.
2. Choose option **1 Extract VSAM**. This will take you to the screen shown in Example 3-10.

Example 3-10 Map Creation Utility

```
-------------------- DMF Map Creation Utility -------------------- SSID: AZKS
Command ===>

  Source Library Name. 'SPARK.ZSPARK.CRDEMO.DCLGEN(CLIENTB)'
  Start Field. . . . . DCL-CLIENT-INFO                N (Case Sensitive)
* End Field. . . . . .                                N (Case Sensitive)(Opt)
* Map Name . . . . . .                                 (Opt)
* Use Offset Zero. . . Y ( Y / N )
* Convert Var to True. N ( Y / N )
  Flatten Arrays . . . C ( C / Y / N ) *See HELP for more details
* Map Data Set Name. .                                                (O pt)

* These fields are always reset to their default settings when this panel is
  first entered
```

3. Enter the location of the VSAM copybook, and enter the start field of the code. Because we want to map every field of the copybook, the end field is optional. Press Enter to proceed to the next screen, shown in Example 3-11.

Example 3-11 Enter VSAM data set name

```
Please note:
- For VSAM (SQL) Read Only - the VSAM DSN name is required and the FCT
  information should be ignored.
- For Streams and VSAM (SQL) Read/Update (via CICS) - the VSAM DSN and FCT
  (File Control Table) DD information is required.

VSAM DSN: SPARK.CRDEMO.CLIENT.VSAMKSDS (no quotes)
Post Read Exit name . . . . . .           (Opt)
Pre-Write Exit name . . . . . .           (Opt) (VSAM for CICS only)

FCT entry for this VSAM cluster:         (Streams and VSAM for CICS)

Do you want to use alternate index(s) on this file: N  (Y/N)

Treat this file as an IAM file: N  (Y/N)
Suppress use of the MapReduce interface: N  (Y/N)
MapReduce thread count:
Please enter the following required information if accessing the VSAM
file via VSAM for CICS:
Connection name . . . . . .
Mirror transaction name . . . . . .
```

4. Enter the VSAM dataset name without quotes and press Enter. The map should now have been created successfully (Example 3-12).

Example 3-12 VSAM map created successfully

```
-------------------- DMF Map Creation Utility ------------- Create Successful
Command ===>

  Source Library Name. 'SPARK.ZSPARK.CRDEMO.DCLGEN(CLIENTB)'
  Start Field. . . . . DCL-CLIENT-INFO                N (Case Sensitive)
* End Field. . . . . .                                N (Case Sensitive)(Opt)
* Map Name . . . . . .                                 (Opt)
* Use Offset Zero. . . Y ( Y / N )
* Convert Var to True. N ( Y / N )
  Flatten Arrays . . . C ( C / Y / N ) *See HELP for more details
* Map Data Set Name. .                                                (O pt)

* These fields are always reset to their default settings when this panel is
  first entered
```

5. Finally, press **PF3** to go back to the Data mapping facility page and select option **5 Map Refresh.** After a successful refresh, you can go to the Data Service Studio's Explorer window to look at the virtual table created (Figure 3-2).

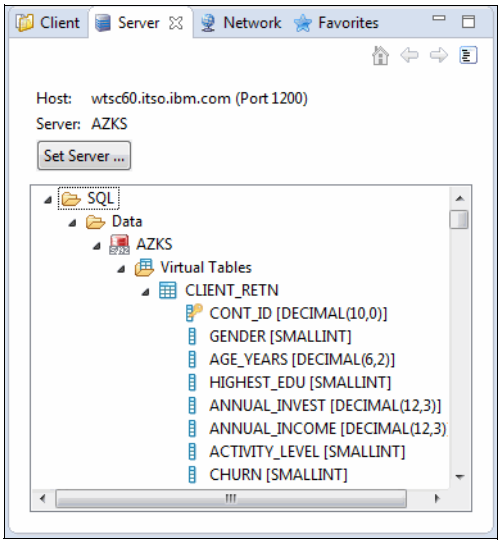


Figure 3-2 Virtual table mapped by ISPF applications

Mapping IMS

The steps to map an IMS database are very similar to mapping VSAM datasets:

1. On the Spark ISPF primary option menu, select **option 4 IMS** to enter the Server IMS control facility. Then, select **option 2 IMS Data mapping** to enter the IMS mapping options screen (Example 3-13).

   Example 3-13 IMS mapping options

   ```
   -------------------  Server IMS Mapping Options ------------------- SSID: AZKS
   Option  ===>

      0  Mapping defaults
      1  Extract COBOL Source/Listing
      2  Extract PL/I from Source
      3  Extract MFS source
      4  Map Display
      5  Map Copy
      6  Map Refresh

      SQL Access Mapping Options:
      A  Extract using DBD source
      B  Extract using PSB source
      C  Extract using COBOL/PLI Source/listings
      E  Display IMS/DB DBD Maps
      F  Display IMS/DB PSB Maps
      G  Generate a View of an IMS/DB DBD and Segment
   ```

2. We used the IMS Database Descriptor (DBD) on **option A Extract using DBD source** and Program Specification Block (PSB) on **option B Extract using PSB source** to generate the maps for IMS Database (Example 3-14).

   Example 3-14 IMS map creation

   ```
   -------------------- DMF MAP CREATION UTILITY ------------- Create Successful
   Command ===>

     Source Library Name. IMS13A.SDFSISRC(DFSIVP62)
   * Map Dataset Name . .                                              (Opt)

   * This field is always reset to it's default setting when this panel is
     first entered
   ```

3. From the DMF map creation utility, enter the source library name of the DBD or PSB and hit enter. If map creation is successful, a message will appear on the top right corner of the screen. If not successful, press **PF1** to view the error messages.

4. Again, enter **PF3** to go back to the previous panel and do a Map Refresh, then go back to the data service studio to view the virtual tables created (Figure 3-3).

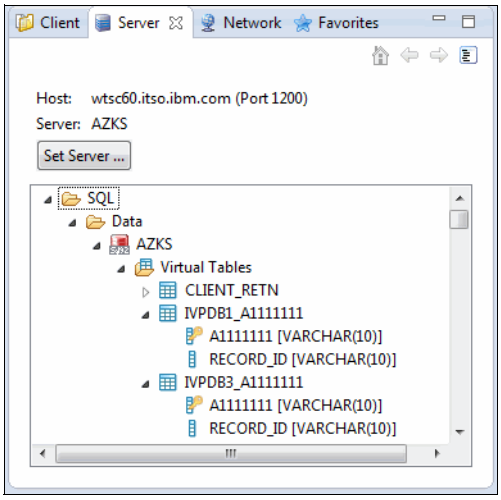


Figure 3-3 New virtual table for IMS in Data Service Studio

You can also look at the IMS DBDs and PSBs from the data service studio (Figure 3-4).

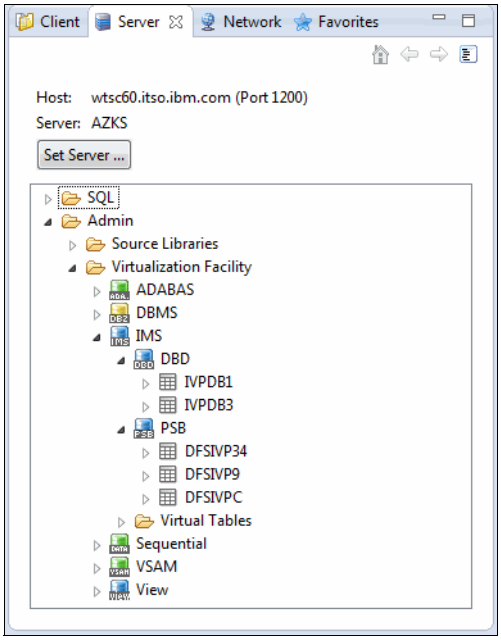


Figure 3-4 IMS DBD and PSB views in Data Service Studio

Server traces

Server traces provide much information about errors that happened when using MDS. There are two ways of looking at the server traces:

First, you can access the ISPF panel by entering **`EX 'hlq.SAZKEXEC(AZK)' 'SUB(SSID)'`** on the ISPF command shell (option 6).

Then select option **B Server Traces** (Example 3-15).

Example 3-15 Server Traces

```
CPU TIME TCBADDR  ----+----1----+----2----+----3----+----4----+----5----+----6-
.000101S C09BACF0 RAW READ EXECUTED        - SOCK 0002 - RAW READ COMPLETED
.000120S C09BACF0 ATTACH - RC 0 - OPDBTP  - NewTask
.000125S C09BACF0 BIND - JDBC - RC 0
.176996S 009BE9A8 AZK6501W DMF XML-Import WARNING: Refresh following save of ne
.010616S 009BB360 AZK6503I DMF IMPORT SAVED MAP 'SPARK.MAP(DOA149DC)' FROM ' ++
.010616S 009BB360 AZK6501W DMF XML-Import WARNING: Refresh following save of ne
.010616S 009BB360 AZK6509I DMF processing complete
.009957S 009BF378 RESMGR detected termination of remote user support task
.011021S 009BB360 RESMGR detected termination of TSO task for Userid LPRES4
.000000S C09BF378 SQL ENGINE OPEN DATABASE - RC 0
.000854S C09BF378 SQ Starting single map refresh for map DCL_CLIENT_INFO
.000854S C09BF378 SQ Single map refresh for map DCL_CLIENT_INFO done
.000854S C09BF378 SQL MAP BUILD COMPLETE
.001860S 009BB360 RESMGR detected termination of TSO task for Userid LPRES4
.000388S 009BB488 RESMGR detected termination of ACI internal service task
```

You can also look at the server traces from the Data Service Studio. Go to the output window on the lower right corner and select the Server Trace tab, then you click the **Play** button on the upper right to start receiving trace data (Figure 3-5).

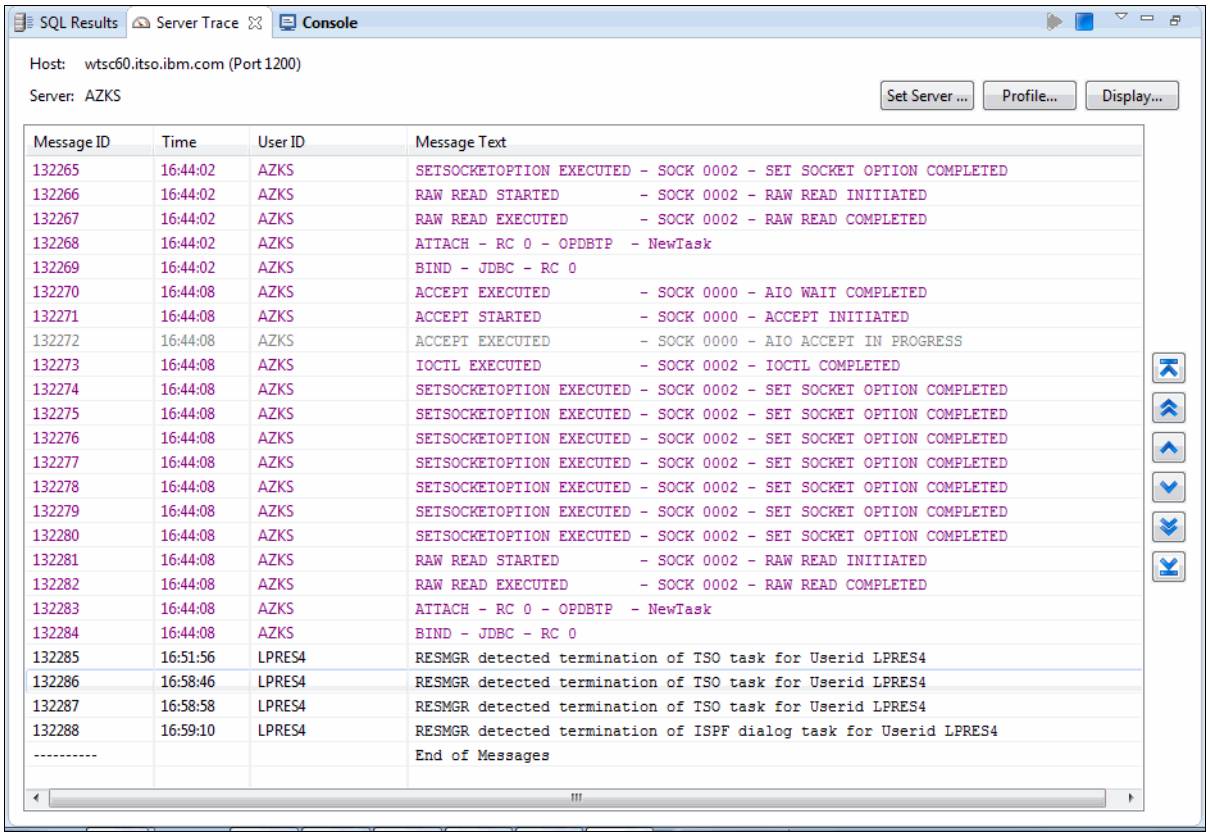


Figure 3-5 Server Trace view in Data Service Studio

See the *IBM z/OS Platform for Apache Spark Administrator's Guide*, SC27-8451 for more information about the usage of the ISPF Panels.

3.2.5 Installing and configuring Bash

Spark on z/OS, and on other platforms, is typically started and controlled by a number of shell scripts that reside in `<install_dir>/bin` and `<install_dir>/sbin`. These scripts are written for GNU Bash, the Bourne Again Shell. They will not work with the standard z/OS shell (/bin/sh) which is a plain Bourne shell without GNU extensions.

Therefore, you need a supported version of bash in order to run Spark scripts. You can obtain Bash, and many other ported tools for z/OS, from the Rocket software website:

http://www.rocketsoftware.com/ported-tools

If Bash is already installed on your system, verify that is at the minimum required level, by entering bash -version.

For detailed installation instructions for Bash (and other ported tools), see the Rocket website.

While not strictly required, we suggest that you make Bash your default login shell, because it is more user friendly than the default Korn shell (believe it or not, even the cursor keys do work). Making Bash the default login shell requires changing the initial program (that is, the PROGRAM attribute) of your user ID's OMVS segment.

Note: On many systems, `/etc/profile` does an explicit invocation of sh so that you end up with `/bin/sh` rather than Bash even if Bash was made the login shell. To fix this, you need to ask your system administrator to update `/etc/profile`. Find the following lines:

```
if [ -z "$STEPLIB" ] && tty -s;
then
    export STEPLIB=none
    exec sh -L
i
```

Have them changed to the following code:

```
if [ -z "$STEPLIB" ] && tty -s;
then
    export STEPLIB=none
    exec -a $0 $SHELL -
fi
```

If you need to install or upgrade Bash, we suggest that you also install some of the other ported tools that are available from the Rocket website. For example, you might find the various compression utilities and especially cURL to be very useful.

Note: One minor quirk with Bash is that z/OS UNIX commands that are implemented as external links (for example, **`ping`** and **`netstat`**) are not resolved from the **`PATH`** environment variable. You have to specify the full path, for example, `/bin/ping` or `/bin/netstat`.

3.2.6 Check for /usr/bin/env

The Spark z/OS shell scripts assume that `/usr/bin/env` exists, but that may not be true on your system. Your system may have that directory as simply `/bin/env` as ours did.

First, see if `/usr/bin/env` exists. If yes, then check to make sure it successfully provides a listing of the environment. In your shell, issue the command **`/usr/bin/env`**. That should result in a list of name/value pairs for the environment seen in your shell. If so, then skip to the next section on basic verification of Spark.

If `/usr/bin/env` does not exist, then look to see where the **`env`** program does reside on your system. It may be at `/bin/env` like ours. In that case, create a symbolic link so that `/usr/bin/env` resolves to the true location of **`env`**, for example the following link:

```
ln -s /bin/env /usr/bin/env
```

Verify that the symlink is good by issuing the command **`/usr/bin/env`**. That should result in a listing of the environment.

> **Important:** If `/usr/bin/env` does not exist, the Spark scripts are run in the standard z/OS Korn shell, and they fail. The reason is that Bash has many extensions to the Korn shell, and the scripts make ample use of these extensions.

3.3 Installing workstation components

This section explains how to install workstation components needed for IBM Platform for Spark on z/OS development.

3.3.1 Installing Data Service Studio

IBM Platform for Spark on z/OS includes a workstation interface called *Data Service Studio* (DSS) to create, view, and change metadata that describes the mapping from mainframe data sets to a relational view of the data on the Data Service Server.

DSS is based on Eclipse. It can either be installed separately, or can be added to an existing Eclipse shell. In the rest of this section, we explain how to install DSS separately (which explains the setup for Spark application development with Scala), and describe how to add DSS to an existing Eclipse shell as a plug-in.

To install DSS separately, perform the following steps:

1. Make sure that a supported Java runtime environment is installed on your workstation.
2. Transfer the installation binaries to your workstation.

   The installation binaries are shipped as a compressed file, in member *hlq*.SAZKBIN(AZKBIN1). Transfer the file to a temporary directory on your workstation, in binary mode, renaming it to `spark-studio-1.1.0.zip` (Example 3-16 on page 47). Also, transfer member *hlq*.SAZKBIN(AZKBIN2), which contains Java Database Connectivity (JDBC) driver software that we will describe in “Installing the JDBC driver on the workstation” on page 48.

Example 3-16 Transferring installation images for Data Service Studio and the JDBC driver

```
C:\Users\ITSOUSER>ftp wtsc60.itso.ibm.com
Connected to wtsc60.itso.ibm.com.
220-FTPD1 IBM FTP CS V2R2 at WTSC60.ITSO.IBM.COM, 13:23:41 on 2016-04-14.
220-Welcome to the SC60 system
220 Connection will not timeout.
User (wtsc60.itso.ibm.com:(none)): lpres1
331 Send password please.
Password:
230 LPRES1 is logged on.  Working directory is "LPRES1.".
ftp> bin
200 Representation type is Image
ftp> get //azk.sazkbin(azkbin1) spark-studio-1.1.0.zip
200 Port request OK.
125 Sending data set AZK.SAZKBIN(AZKBIN1)
250 Transfer completed successfully.
ftp: 265906452 bytes received in 22.73Seconds 11698.48Kbytes/sec.
ftp> get //azk.sazkbin(azkbin2) DSDriver.zip
200 Port request OK.
125 Sending data set AZK.SAZKBIN(AZKBIN2)
250 Transfer completed successfully.
ftp: 2023705 bytes received in 0.18Seconds 11242.81Kbytes/sec.
ftp> quit
221 Quit command received. Goodbye.

C:\Users\ITSOUSER>
```

3. Next, extract `spark-studio-1.1.0.zip` into another folder on your workstation. The product will be installed in this folder, so it should not be a temporary one. You should now have a directory layout similar to that shown in Figure 3-6.

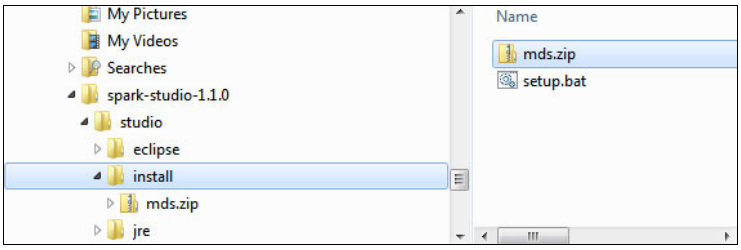


Figure 3-6 Install studio directory layout

4. Run the `SETUP.BAT` script in the install subdirectory. The script configures the DSS plugins into the Eclipse installation.

5. Now, you are ready to launch DSS via the `launch.bat` script in the studio subfolder. You should see the DSS initial window (Figure 3-7).

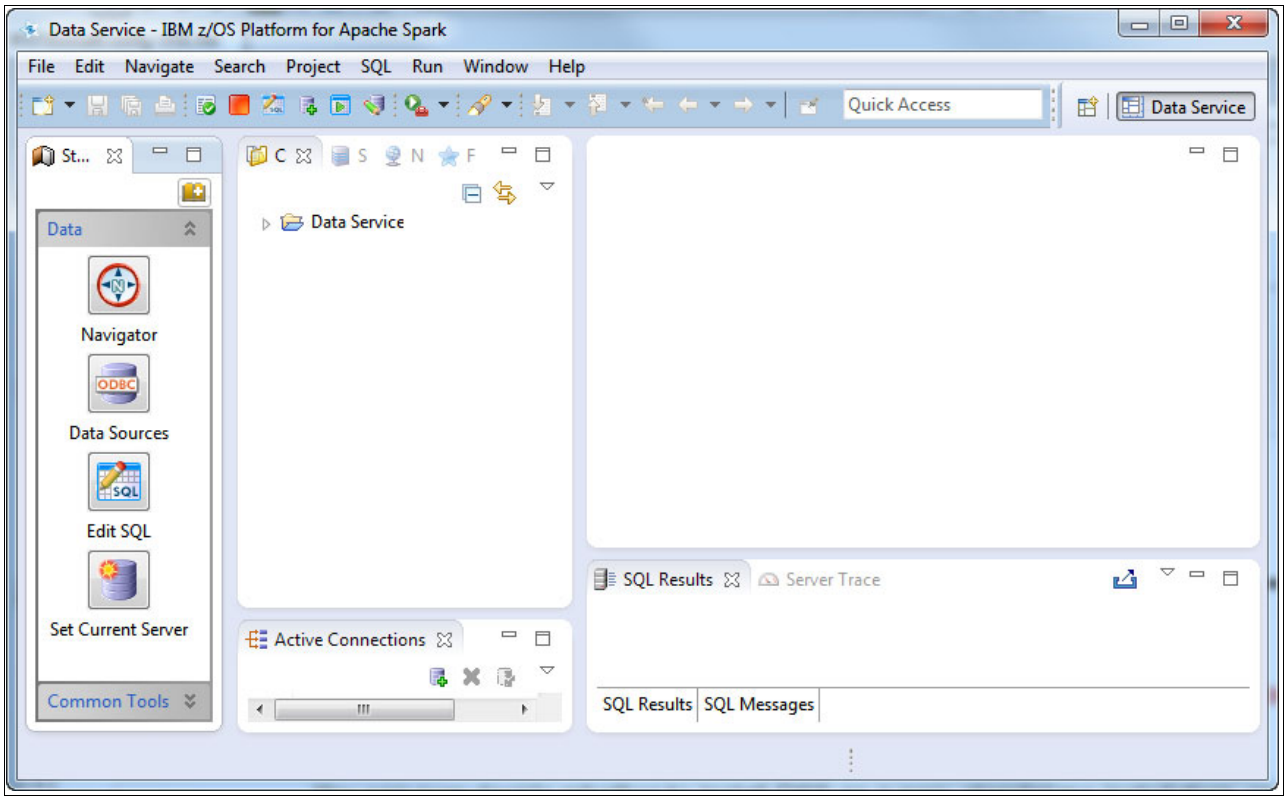


Figure 3-7 Data Service Studio initial window (Data Service perspective)

3.3.2 Installing the JDBC driver on the workstation

Java applications (or, more precisely, applications that run on the Java virtual machine (JVM)) access z/OS data by connecting to the MDSS server. In JDBC terms, this driver is a Type 4 driver. That is, it is written in pure Java, does not have any native library dependencies, and talks directly to the MDSS server using its own network protocol.

Installing the driver is straightforward. You should have already downloaded the file from the product files on z/OS, as a compressed file called `DSDriver.zip` (see Example 3-16 on page 47).

Simply extract the contents of that compressed file into a directory of your choice. The directory then contains the DSS driver (`dv-jdbc-3.1.22756.jar`), in addition to some dependencies (two `log4j` Java archive (JAR) files) and license notes (Figure 3-8).

Name	Type	Size
apidocs	File folder	
optional	File folder	
dv-jdbc-3.1.22756.jar	Executable Jar File	673 KB
helpdriver.cmd	Windows Comma...	1 KB
helpdriver.txt	Text Document	14 KB
LICENSE.txt	Text Document	8 KB
log4j2.xml	XML Document	2 KB
log4j-api-2.4.1.jar	Executable Jar File	140 KB
log4j-core-2.4.1.jar	Executable Jar File	968 KB
NOTICE.txt	Text Document	1 KB
RELEASE-NOTES-driver.txt	Text Document	2 KB
sysinfo.cmd	Windows Comma...	1 KB

Figure 3-8 Contents of the JDBC driver package

3.4 Configuring Apache Spark for z/OS

Now, we are ready to configure the IBM z/OS Platform for Apache Spark itself. The z/OS Platform for Apache Spark will install Apache Spark into a z/OS file system (zFS) or hierarchical file system (HFS) file system directory, which we'll refer to as `<spark_install_dir>/`. In our installation, this was `/usr/lpp/IBM/Spark`. Typically, Apache Spark by default runs from this installation directory, and all of its configuration, logs, and work information is stored in this installation directory structure.

On z/OS this is not ideal behavior to have the installation directory assume these tasks, so in this section, we modify the default configuration to get Apache Spark using customized directories for its configuration, logs, and temporary work.

3.4.1 Create log and worker directories

By default, Spark creates log files in `$SPARK_HOME/logs`, where `$SPARK_HOME` is the spark install directory (`<spark_install_dir>`). You could either create that directory and make it writable for each Spark user, or create a separate log directory and configure Spark to log to that directory instead.

We chose the option to create a separate directory, and created a dedicated log directory under `/var`:

```
mkdir -p /var/spark/logs
```

Similarly, Spark worker JVMs create work files in `$SPARK_HOME/work` by default. Again, we use a directory under `/var` instead:

```
mkdir -p /var/spark/work
```

See"Installing the Data Server JDBC driver" on page 53 for an explanation of how to change the Spark configuration to use these directories instead of the default.

3.4.2 Apache Spark directory structure

The first step is to identify the user that will run the Apache Spark master and worker. If you followed the steps in *IBM z/OS Platform for Apache Spark Installation and Customization Guide Version 1 Release 1*, SC27-8449, you have created a `sparkid` user that can run both the master and the worker. Understanding how this user's environment is configured allows us to focus on modifying its Spark configuration. See "Update the configuration files" on page 51 for how this is done.

Next, Spark's installation directory structure is by default as follows:

- Spark configuration files are shipped in *$SPARK_HOME*/`conf`, where *$SPARK_HOME* is the spark install directory (<*spark_install_dir*>).

  Note: Although Spark does not write to this directory for its operations, any modifications to Spark's configuration files in this directory are lost if the <*spark_install_dir*> is updated for service.

- Spark creates log files for the Master and Worker daemons in *$SPARK_HOME*/`logs`.
- Spark worker JVMs create work files in *$SPARK_HOME*/`work`.
- Spark writes executable files into `/tmp`.

You can either create a new directory as a mount point to hold Spark's directory structure completely, or create separate directories for each of Spark's subdirectories and configure Spark to use them instead.

3.4.3 Create directories and local configuration

By default, Spark creates log files in *$SPARK_HOME*/`logs`. As explained previously, we advise you to leave that directory untouched.

Complete the following steps:

1. Create a dedicated log directory and configure Spark to log to that directory, for example, in the `/var` hierarchy:

   ```
   mkdir /var/sparkinst1
   ```

2. Create a zFS (100 cylinders) and mount it on `/var/sparkinst1` (don't forget to update BPXPMRxx).

   Perform the following commands:

   ```
   mkdir /var/sparkinst1/conf
   cd /var/sparkinst1/conf
   cp /usr/lpp/IBM/Spark/conf/spark-env.sh .
   cp /usr/lpp/IBM/Spark/conf/spark-defaults.conf .
   cp /usr/lpp/IBM/Spark/conf/log4j.properties.template ./log4j.properties
   mkdir -p /var/sparkinst1/logs
   ```

3. Similarly, the Spark worker JVMs create work files in *$SPARK_HOME*/`work` by default. Again, we use a directory under `/var` instead:

   ```
   mkdir -p /var/sparkinst1/work
   mkdir -p /var/sparkinst1/tmp
   ```

4. Create a zFS that will be mounted on this `tmp` subdirectory, because the `tmp` subdirectory is used in some data transformations and you might need to increase it at some time independently of the main file system. The size of this file system depends on the volume of data that will be processed. You can use 500 cylinders for your first installation.
5. Finally, make sure that the userid that will run Apache Spark programs has read/write (R/W) access to the new file structure:

   ```
   chown -R  sparkid:sparkgroupid /var/sparkinst1
   ```

6. Because the local directory tree is now created, update the profile of the spark userid:

   ```
   export SPARK_CONF_DIR=/var/sparkinst1/conf
   ```

The next section explains how to change the Spark configuration to use these directories instead of the default.

> **Remember:** There is no need for the Spark directory to be the same for all of these local directories.

Update the configuration files

We will modify the following configuration files residing in `/var/sparkinst1/conf`:

spark-env.sh This is a shell script that is being sourced by most of the other scripts in the Spark installation. You can use it to configure environment variables that set or alter the default for several Spark settings.

spark-defaults.conf This is a configuration file that sets default values for the Spark runtime components. These default values can be overridden on the command line when you interact with Spark using shell scripts.

log4j.properties This file contains the default configuration for `log4j`, the venerable logging package that Spark uses.

First of all, we need to make two modifications to `spark-env.sh`. We set `JAVA_HOME` to point to the location of IBM Java runtime environment (JRE) Version 8 for z/OS, and we set the `_BPXK_AUTOCVT` variable to `ON ()`.

The `_BPXK_AUTOCVT` setting enables automatic code page conversion for tagged files. You can tell z/OS UNIX what code page a text file is encoded in, and it will transparently convert the text file into the code page expected by a program. For example, you can edit an American Standard Code for Information Interchange (ASCII) encoded text file with the vi editor on z/OS UNIX without first converting the file to Extended Binary Coded Decimal Interchange Code (EBCDIC), and without having to convert it back after the edit session.

We also configured the location of log files, by setting the `SPARK_LOG_DIR` environment variable to point to the directory created in. You can omit this if you choose to use the default, *$SPARK_HOME*`/logs`.

Many more environment variables can be configured in `spark-env.sh`, as the comments suggest. However, some of the settings are deprecated, and should be done in `spark-defaults.conf` instead. For more details, see the Spark documentation, which can be found on the following website:

http://spark.apache.org/docs/latest/configuration.html

A sample of our `spark-env.sh` script can be seen in Example 3-17.

Example 3-17 Sample spark-env.sh script

```
#!/usr/bin/env bash
export JAVA_HOME=/usr/lpp/java/J8.0_64
export _BPXK_AUTOCVT=ON
# Options read when launching programs locally with
# ./bin/run-example or ./bin/spark-submit
# - HADOOP_CONF_DIR, to point Spark towards Hadoop configuration files
# - SPARK_LOCAL_IP, to set the IP address Spark binds to on this node
# - SPARK_PUBLIC_DNS, to set the public dns name of the driver program
# - SPARK_CLASSPATH, default classpath entries to append
[ ... omitted ... ]
# - SPARK_LOCAL_DIRS, storage directories to use on this node for shuffle and RDD
data
SPARK_LOCAL_DIRS=/var/sparkinst1/tmp
[ ... omitted ... ]
# - SPARK_WORKER_DIR, to set the working directory of worker processes
SPARK_WORKER_DIR=/var/sparkinst1/work
[ ... omitted ... ]
# - SPARK_WORKER_DIR, to set the working directory of worker processes
SPARK_CONF_DIR=/var/sparkinst1/conf
[ ... omitted ... ]
# - SPARK_LOG_DIR       Where log files are stored.  (Default: ${SPARK_HOME}/logs)
SPARK_LOG_DIR=/var/sparkinst1/logs
```

Next, we make some modifications to `spark-defaults.conf` (see Example 3-18). We add the locations of the IBM Data Server Driver for JDBC and SQLJ (unofficially called the *JCC driver*) to the classpath of both driver and executor nodes, as well as the classpath of the DSS driver and its prerequisites.

Example 3-18 Customizations to spark-defaults.conf

```
# Default system properties included when running spark-submit.
# This is useful for setting default environmental settings.

# Example:
# spark.master                     spark://master:7077
# spark.eventLog.enabled           true
# spark.eventLog.dir               hdfs://namenode:8021/directory
# spark.serializer                 org.apache.spark.serializer.KryoSerializer
# spark.driver.memory              5g
spark.io.compression.codec         org.apache.spark.io.LZ4CompressionCodec

spark.driver.extraJavaOptions      -Dfile.encoding=ISO8859-1
spark.driver.extraClassPath        /usr/lpp/db2/db11/jdbc/classes/db2jcc4.jar:/usr
/lpp/db2/db11/jdbc/classes/db2jcc4_license_cisuz.jar:/u/sparkid/DSDriver/dv-jdbc-3
.1.22756.jar:/u/sparkid/DSDriver/log4j-api-2.4.1.jar:/u/sparkid/DSDriver/log4j-cor
e-2.4.1.jar

spark.executor.extraJavaOptions    -Dfile.encoding=ISO8859-1
spark.executor.extraClassPath      /usr/lpp/db2/db11/jdbc/classes/db2jcc4.jar:/usr
/lpp/db2/db11/jdbc/classes/db2jcc4_license_cisuz.jar:/u/sparkid/DSDriver/dv-jdbc-3
.1.22756.jar:/u/sparkid/DSDriver/log4j-api-2.4.1.jar:/u/sparkid/DSDriver/log4j-cor
e-2.4.1.jar
```

Also, we need an extra Java option, `file.encoding`, to be set to `ISO8859-1`. This latter setting switches the JVM's default character set. On z/OS, it would otherwise usually be `IBM-1047`, an EBCDIC code page.

Important: Changing the JVM's default code page to `ISO8859-1` is crucial. If this option is not set, you will see a rather strange exception stack trace when trying to start spark-shell, or any other Scala program. The underlying reason is a subtle bug in the Scala runtime library that manifests itself only when the default JVM code page is not an ASCII variant, which, alas, is the case on z/OS.

The following part of the stack trace is important:

```
java.lang.ExceptionInInitializerError at [...]
```

This exception is caused by the following error:

```
scala.reflect.internal.MissingRequirementError: error while loading package,
Scala signature package has wrong version expected: 5.0 found: 45.0 in
scala.package
```

If you see that message, it indicates that the JVM runs with the wrong default encoding.

The underlying Scala problem is scheduled to be resolved in Scala 2.11.9, so future Spark versions might not throw this exception.

One thing to remember is the size of the file system may need to be large depending on the type of work that is running, specifically the `SPARK_LOCAL_DIRS` (see `spark.local.dir` on the following website):

http://spark.apache.org/docs/1.5.2/configuration.html#application-properties

Important: When you must update the Spark driver, updating anything in `/usr/lpp/IBM/Spark/conf` is a bad idea because it will be erased with any service update.

An alternative solution is to complete the following steps:

1. Mount `/usr/lpp/IBM/Spark` in ReadOnly.
2. On some target directory, re-create the same structure (in our case the "target" was `/u/stef/spark`).
3. From `/usr/lpp/IBM/Spark`, copy everything that may be changed to the target directory.
4. For "binaries", create symbolic links inside the target pointing to the files under `/usr/lpp/IBM/Spark`.

   For example, you might enter the following commands:

   ```
   cd target/lib
   ln -s /usr/lpp/spark/V1R5M2/spark/lib/spark-assembly-1.5.2-hadoop2.6.0.jar
   spark-assembly-1.5.2-hadoop2.6.0.jar
   ```
5. Set `SPARK_HOME` to the target directory and configure accordingly.

3.4.4 Installing the Data Server JDBC driver

The Data Server JDBC driver is not pre-installed with the Spark distribution. Upload the driver manually on z/OS after you transfer the original archive file to the workstation and unpacked it there, as described in "Installing the JDBC driver on the workstation" on page 48.

Create a directory in zFS for the driver code (we used `/u/sparkid/DSDriver`) and make sure that the directory is readable and searchable (mode 755).

Using binary File Transfer Protocol (FTP), transfer the three JAR files for the driver and its dependencies to that directory (Example 3-19). The `log4j2.xml` file is only required for logging. The driver will print a warning message about a missing configuration file if this file is omitted, but that does not affect the functionality of the driver.

Example 3-19 Transferring the Data Server JDBC drivers to z/OS UNIX

```
C:\Users\ITSOUSER>ftp wtsc60.itso.ibm.com
Connected to wtsc60.itso.ibm.com.
ftp> lcd C:\Users\ITSOUSER\DSDriver\dv-jdbc-3.1.22756
Local directory now C:\Users\ITSOUSER\DSDriver\dv-jdbc-3.1.22756.
ftp> cd /u/sparkid/DSDriver
250 HFS directory /u/sparkid/DSDriver is the current working directory
ftp> bin
200 Representation type is Image
ftp> prompt
Interactive mode Off .
ftp> mput *.jar
200 Port request OK.
125 Storing data set /u/sparkid/DSDriver/dv-jdbc-3.1.22756.jar
125 Storing data set /u/sparkid/DSDriver/log4j-api-2.4.1.jar
125 Storing data set /u/sparkid/DSDriver/log4j-core-2.4.1.jar
ftp> put log4j2.xml
200 Port request OK.
125 Storing data set /u/sparkid/DSDriver/log4j2.xml
250 Transfer completed successfully.
ftp: 1629 bytes sent in 0.09Seconds 18.72Kbytes/sec.
ftp> ls
200 Port request OK.
125 List started OK
dv-jdbc-3.1.22756.jar
log4j-api-2.4.1.jar
log4j-core-2.4.1.jar
log4j2.xml
ftp> quit
221 Quit command received. Goodbye.
```

3.4.5 Modifying the log4j configuration

You may find that the default `log4j` configuration causes a lot of verbosity on your terminal when you invoke spark-shell, for example. You may want to modify the configuration to log only messages with a severity of `WARN` or higher to the console (Example 3-20 on page 55). Also, you could change the conversion pattern for the log messages, for example, if you would like a different timestamp format.

Example 3-20 Modified log4j configuration

```
# Set everything to be logged to the console
log4j.rootCategory=INFO, console

# Console config
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
log4j.appender.console.threshold=WARN
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%d{yy/MM/dd HH:mm:ss} %p %c{1}: %m%n

# Settings to quiet third party logs that are too verbose
log4j.logger.org.spark-project.jetty=WARN
log4j.logger.org.spark-project.jetty.util.component.AbstractLifeCycle=ERROR
log4j.logger.org.apache.spark.repl.SparkIMain$exprTyper=INFO
log4j.logger.org.apache.spark.repl.SparkILoop$SparkILoopInterpreter=INFO
log4j.logger.org.apache.parquet=ERROR
log4j.logger.parquet=ERROR

# SPARK-9183: Settings to avoid annoying messages when looking up nonexistent UDFs
# in SparkSQL with Hive support
log4j.logger.org.apache.hadoop.hive.metastore.RetryingHMSHandler=FATAL
log4j.logger.org.apache.hadoop.hive.ql.exec.FunctionRegistry=ERROR
```

3.4.6 Adding the Spark binaries to your PATH

Finally, you may want to add the location of the Spark binaries (executable scripts) to your search path. Add the following code to your `.profile` file (Example 3-21).

Example 3-21 Adding Spark scripts to the PATH

```
# Environment customization for Apache Spark

export SPARK_HOME=/usr/lpp/IBM/Spark
export PATH="$PATH":"$SPARK_HOME/bin"
export PATH="$PATH":"$SPARK_HOME/sbin"

. "$SPARK_HOME/conf/spark-env.sh"
```

The last line reads the customized **`spark-env.sh`** script so that `_BPX_AUTOCVT` and `JAVA_HOME` are set as well. To update your current shell with the new variables, re-execute your `.profile` (do not miss the period (.) at the beginning of the command):

```
. ./profile
```

3.5 Verifying the installation

We are now ready to verify the installation.

In an SSH or Telnet session using your ID for running Spark, issue the following command:

```
/usr/lpp/IBM/Spark/bin/spark-shell
```

You will see a series of messages, and finally the Spark prompt. From there, issue the **help** command, and the Spark shell responds with a short summary of its built-in commands (Example 3-22).

> **Note:** The warning message `Unable to load native-hadoop library for your platform` can safely be ignored. It indicates that the native Hadoop Dynamic Link Library (`libhadoop.so`) could not be found on the system.

Example 3-22 Starting the Spark shell

```
LPRES1:/u/lpres1: >/usr/lpp/IBM/Spark/bin/spark-shell
16/04/25 10:51:45 WARN NativeCodeLoader: Unable to load native-hadoop library for your
platform... using builtin-java classes where applicable
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.5.2
      /_/

Using Scala version 2.10.4 (IBM J9 VM, Java 1.8.0)
Type in expressions to have them evaluated.
Type :help for more information.
[ ... messages not shown ... ]
SQL context available as sqlContext.

scala> :help
All commands can be abbreviated, e.g. :he instead of :help.
Those marked with a * have more detailed help, e.g. :help imports.

:cp <path>                add a jar or directory to the classpath
:help [command]           print this summary or command-specific help
:history [num]            show the history (optional num is commands to show)
:h? <string>              search the history
:imports [name name ...]  show import history, identifying sources of names
:implicits [-v]           show the implicits in scope
:javap <path|class>       disassemble a file or class name
:load <path>              load and interpret a Scala file
:paste                    enter paste mode: all input up to ctrl-D compiled together
:quit                     exit the repl
:replay                   reset execution and replay all previous commands
:reset                    reset the repl to its initial state, forgetting all session
entries
:sh <command line>        run a shell command (result is implicitly => List[String])
:silent                   disable/enable automatic printing of results
:fallback
disable/enable advanced repl changes, these fix some issues but may introduce others.
This mode will be removed once these fixes stablize
:type [-v] <expr>         display the type of an expression without evaluating it
:warnings                 show the suppressed warnings from the most recent line which had
any

scala>
```

Tip: The Spark shell writes a history as `.spark_history`, and reads that history when it starts up. This is very convenient for two reasons:

- The statements that you type are remembered between sessions, so in a new Spark shell, you have access to statements from previous invocations.
- You can view the statements with an editor, and cut and paste them into, for example, a Spark Scala application.

Now, if you view the `.spark_history` file with, for example, the vi editor, it appears unintelligible because the Spark shell writes it in ISO8859-1 encoding, However, z/OS UNIX does not recognize that, and assumes the default encoding (usually IBM-1047).

To adjust for this, you can tag the file with the following command:

```
chtag -tc ISO8859-1 .spark_history
```

This tells z/OS UNIX that the file is a text file in the specified encoding. If `_BPXK_AUTOCVT` is set to `ON`, it will now be automatically converted, and can easily be viewed or edited without manual conversion.

3.6 Starting the Spark daemons

We are now almost ready to start the Spark daemons, which are the Spark Master and the workers (“slaves”).

Starting and stopping the daemons is controlled with shell scripts in `$SPARK_HOME/sbin`. Table 3-1, which can also be found on the following website, provides a brief overview:

http://spark.apache.org/docs/latest/spark-standalone.html

Table 3-1 Scripts in `$SPARK_HOME/sbin` to start and stop cluster components

Script	Description
`start-master.sh`	Starts a master instance on the machine where the script is executed.
`start-slaves.sh`	Starts a slave instance on each machine specified in the `conf/slaves` file.
`start-slave.sh`	Starts a slave instance on the machine where the script is executed.
`start-all.sh`	Starts both a master and a number of slaves as described in the previous table cells.
`stop-master.sh`	Stops the master that was started using the **`bin/start-master.sh`** script.
`stop-slaves.sh`	Stops all slave instances on the machines specified in the `conf/slaves` file.
`stop-all.sh`	Stops both the master and the slaves, as described previously.

You can choose either to start and stop the master and each slave individually, or to start master and slaves together. For starting them all together to work correctly, you need to set up the slaves file in `$SPARK_HOME/conf`. This file contains a list of host names where slaves should be started. The default is to start one slave on the local machine (localhost).

The worker startup scripts uses Secure Shell (SSH) to launch worker instances. Therefore, if you use **`start-slaves.sh`** or **`start-all.sh`**, you either have to type the password manually for each worker, or you have to set up password-less SSH.

For more information, see the Spark documentation:

http://spark.apache.org/docs/1.5.2/spark-standalone.html#cluster-launch-scripts

We chose to set up password-less SSH by completing the following steps:

1. Create a public/private Rivest-Shamir-Adleman algorithm (RSA) key pair without a passphrase, and add the public key to the list of authorized keys (Example 3-23).

   Example 3-23 Setting up password-less SSH

   ```
   SPARKID:/u/sparkid: >ssh-keygen
   Generating public/private rsa key pair.
   Enter file in which to save the key (/u/sparkid/.ssh/id_rsa):
   Enter passphrase (empty for no passphrase):
   Enter same passphrase again:
   Your identification has been saved in /u/sparkid/.ssh/id_rsa.
   Your public key has been saved in /u/sparkid/.ssh/id_rsa.pub.
   The key fingerprint is:
   d8:c7:b8:fb:79:a9:44:08:55:f5:ab:4d:53:f7:6d:c3 SPARKID@WTSC60
   The key's randomart image is:
   +--[ RSA 2048]----+
   [ … randomart image not shown … ]
   SPARKID:/u/sparkid: >cat .ssh/id_rsa.pub >> .ssh/authorized_keys
   SPARKID:/u/sparkid: >
   ```

2. You are now ready to start the Spark daemons. Run the **start-all.sh** shell script in `$SPARK_HOME/bin` (Example 3-24).

   Example 3-24 Starting Spark daemons

   ```
   SPARKID:/u/sparkid: >start-all.sh
   starting org.apache.spark.deploy.master.Master, logging to
   /var/spark/logs/spark--org.apache.spark.deploy.master.Master-1-WTSC60.out
   localhost: starting org.apache.spark.deploy.worker.Worker, logging to
   /var/spark/logs/spark-SPARKID-org.apache.spark.deploy.worker.Worker-1-WTSC60.out
   SPARKID:/u/sparkid: >
   ```

   After a successful start, the Spark master's Web UI should be available on port 8080 (Figure 3-9 on page 59).

   The port names for the Spark master URL, REST interface, and Web interface can be specified either on the spark-master command line, or can be configured in `$SPARK_HOME/conf/spark-env.sh`.

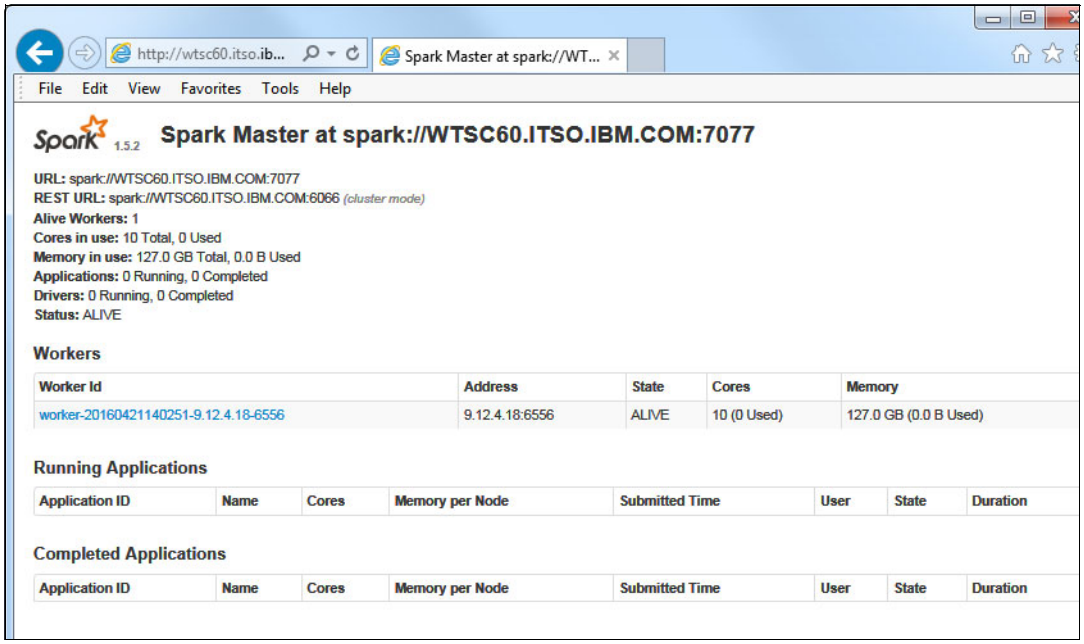


Figure 3-9 Spark Master UI

Important: The URL for our Spark master is `spark://WTSC60.ITSO.IBM.COM:7077`, with the host name in upper case, as you can see in Figure 3-9 (on our system, the z/OS UNIX **`hostname`** command returns the name in upper case as well).

When you need to specify the master URL, for example, as an argument to a **`spark-submit`** command, be very careful to spell the URL *exactly* as given, or you will probably see unexpected error messages (Example 3-25 on page 60). Also, *IP address* and *host name* are not interchangeable in this context. In this case, do *not* do what we did.

In Example 3-25 on page 60, the first command shows a spark-submit command where the master URL points to an unknown host, and the error message is clear enough. However, the second example uses the correct host name, but it is spelled in lowercase. This causes an uncaught exception with a stack trace that does not really help to identify the cause of the problem.

Example 3-25 shows the error messages caused by an incorrect host name.

Example 3-25 Error messages from spark-submit when master URL is incorrect

```
LPRES1:/u/lpres1: >spark-submit --master spark://foobar.itso.ibm.com:7077 --class
SparkHelloWorld sg24-8325-samples_2.10-1.0.jar
[ ... other messages not shown ... ]
16/04/27 10:01:54 WARN AppClient$ClientEndpoint: Failed to connect to master
foobar.itso.ibm.com:7077

LPRES1:/u/lpres1: >spark-submit --master spark://wtsc60.itso.ibm.com:7077 --class
SparkHello sg24-8325-samples_2.10-1.0.jar
16/04/27 10:04:10 ERROR SparkUncaughtExceptionHandler: Uncaught exception in
thread Thread[appclient-registration-retry-thread,5,main]
java.util.concurrent.RejectedExecutionException: Task
java.util.concurrent.FutureTask@ec634a0 rejected from
java.util.concurrent.ThreadPoolExecutor@172df515[Running, pool size = 1, active
threads = 1, queued tasks = 0, completed tasks = 0]
        at
java.util.concurrent.ThreadPoolExecutor$AbortPolicy.rejectedExecution(ThreadPoolEx
ecutor.java:2058)
        at
java.util.concurrent.ThreadPoolExecutor.reject(ThreadPoolExecutor.java:834)
[ ... ]
```

Note: In a production environment, you probably want to start the Spark daemons from a z/OS batch job. We show you how to do that in“Starting Spark master and workers from JCL” on page 89.

4

Spark application development on z/OS

We begin this chapter with a brief explanation of how to set up an Eclipse-based development environment for writing Spark applications in the Scala programming language. Then, we show several complete and working examples of how to access and analyze mainframe data sources, such as IBM Virtual Storage Access Method (VSAM) files and system log messages, from a Spark application.

Specifically, this chapter describes the following topics:

- 4.1, “Setting up the development environment” on page 62
- 4.2, “Accessing VSAM data as an RDD” on page 65
- 4.3, “Accessing sequential files and PDS members” on page 71
- 4.4, “Accessing IBM DB2 data as a DataFrame” on page 71
- 4.5, “Joining DB2 data with VSAM” on page 72
- 4.6, “IBM IMS data to DataFrames” on page 73
- 4.7, “System log” on page 74
- 4.8, “SMF data” on page 76
- 4.9, “JavaScript Object Notation” on page 79
- 4.10, “Extensible Markup Language” on page 82
- 4.11, “Submit Spark jobs from z/OS applications” on page 84

4.1 Setting up the development environment

Working from the Spark shell is great for experimenting and one-off jobs, but to do serious development work, you will want to use an integrated development environment (IDE), such as Eclipse.

If you read this as an experienced Spark developer, you probably already have your IDE and your build tools set up and ready. If that is the case, and if Eclipse is your IDE of choice, you might want to install the IBM Data Server Studio (DSS) plugin by following the instructions in 4.1.2, “Installing Data Server Studio plugins into Scala IDE”.

4.1.1 Installing Scala IDE

We downloaded Scala IDE Version 4.4.0 from `http://scala-ide.org`. Scala IDE is Eclipse with additional plugins for Scala development, created and sponsored by Typesafe.

Installation of Scala IDE is easy enough: It comes as a `.zip` archive (compressed file) that you just need to extract on your development machine. The only prerequisite is a supported version of Java to run Eclipse itself, and to run Scala code. Because we want to do Spark development, you will need to have Java 8 or later installed on your machine.

4.1.2 Installing Data Server Studio plugins into Scala IDE

After successful installation of Scala IDE, we suggest that you also add the plugins for DSS into that Eclipse. In FIXME, we describe how to install a standalone version of DSS. Follow the instructions in that section up to and including the FIXME step:

1. In Scala IDE, select **Help** → **Install New Software**.
2. In the Available Software dialog that opens, click **Add**.
3. Click **Archive**, navigate to the install folder of the DSS installation directory, select the `mds.zip` archive file, and click **OK**.
4. Now, you should see Mainframe Data Service for Apache Spark listed as available to be installed. Check it and click **Next** (Figure 4-1 on page 63). Two more screens prompt you to review what is going to be installed, and to accept the license agreement.
5. Click **Finish** to begin the installation. After a few seconds, Eclipse prompts you to confirm that you trust the certificate, then continues with installation.
6. Finally, you are prompted to restart Eclipse in order to make the changes effective. After restart, the Data Service Perspective has been added to Scala IDE. Select **Window** → **Open Perspective** → **Other** → **Data Service** to open it.

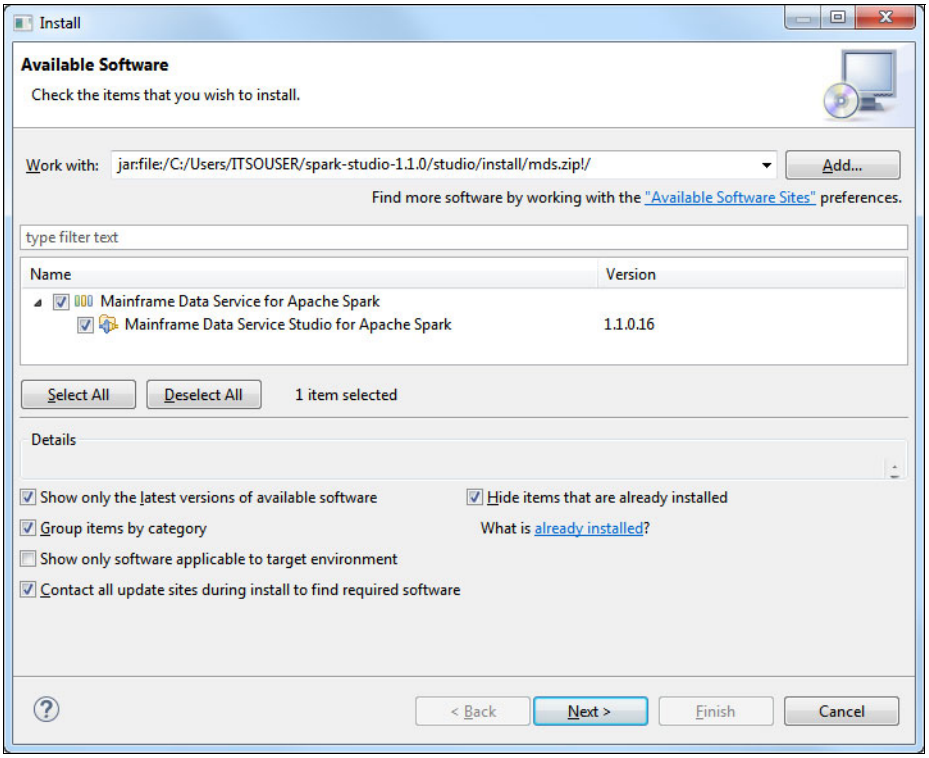


Figure 4-1 Installing the Mainframe Data Services for Apache Spark for z/OS (MDSS) plugin into Scala IDE

4.1.3 Installing and using sbt

We used sbt (“Scala Build Tool”) to build and package our Scala applications. Using sbt, we can easily resolve the project’s dependencies to external libraries, and create the necessary Eclipse artifacts.

We downloaded sbt from `http://scala-sbt.org`, and extracted it into a local folder on the development workstation.

Because we use Eclipse, we also configured the sbteclipse plugin, which can automatically create Eclipse `.project` and `.classpath` files. We chose to add sbteclipse to the global sbt plugin configuration, by creating a `.sbt/0.13/plugins/plugins.sbt` file containing the plugin descriptor for sbteclipse (Example 4-1). For more information, see the sbteclipse documentation:

`https://github.com/typesafehub/sbteclipse`

Example 4-1 Sample plugins.sbt configuration file

```
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "4.0.0")
```

You define characteristics of a Scala application, such as its name and its dependencies, in a build definition file. Our examples use the build definition file shown in Example 4-2.

Example 4-2 Sample build definition file

```
name := "SG24-8325 samples"

version := "1.0"

scalaVersion := "2.10.5"

libraryDependencies += "org.apache.spark" %% "spark-core" % "1.5.2" % "provided"
withSources() withJavadoc()
libraryDependencies += "org.apache.spark" %% "spark-sql" % "1.5.2" % "provided"
withSources() withJavadoc()
libraryDependencies += "org.apache.spark" %% "spark-graphx" % "1.5.2" % "provided"
withSources() withJavadoc()
libraryDependencies += "org.apache.spark" %% "spark-mllib" % "1.5.2" % "provided"
withSources() withJavadoc()
```

In order to easily run sbt from Eclipse, you can create an External Tools Configuration in Eclipse:

1. Select **Run** → **External Tools** → **External Tools Configurations** and create a new tool configuration by clicking the **New launch configuration** icon.
2. Enter the location where you installed sbt, and set the working directory to the current project location. In the Arguments field, enter `${string_prompt:SBT command}` (Figure 4-2).
3. In the Refresh tab, select **Refresh resources upon completion**, and select **The project containing the selected resource**.

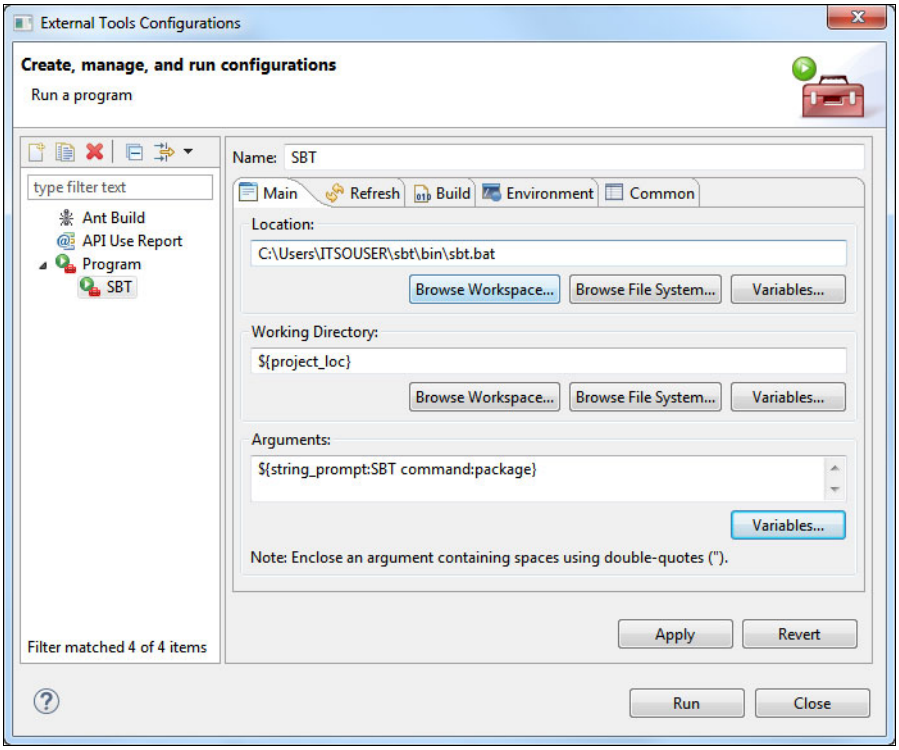


Figure 4-2 Creating an External Tools Configuration to launch sbt

4. You can now run sbt from Eclipse by selecting **Run** → **External Tools**, or by clicking the respective icon in the Eclipse toolbar. The launch configuration then prompts you for the SBT command to be run. Because we most frequently used the **package** command, we set this as the default (Figure 4-3).

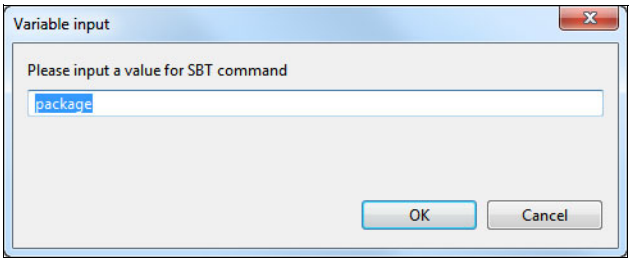


Figure 4-3 Running sbt from Eclipse

By configuring `sbt-eclipse` as an sbt plugin (Example 4-1 on page 63), there is an extra sbt command, **`eclipse`**, available. This command resolves the dependencies if required, and builds `.project` and `.classpath` files for Eclipse. Run that command whenever your dependencies changed, in order to rebuild your Eclipse class path.

In 4.2.2, “Building and running the application” on page 70, we explain how to package a Spark Scala application using sbt, and how to run it from an IBM z/OS UNIX shell. The remaining sections will then not explain this again but will assume that you follow the same steps.

4.2 Accessing VSAM data as an RDD

In this section, we show how to access VSAM data as a Spark DataFrame, and perform some simple analytical operations on it.

The VSAM data is described by a simple Common Business Oriented Language (COBOL) copybook (Example 4-3).

Example 4-3 Copybook for Client info VSAM dataset

```
01  DCL-CLIENT-INFO.
    10 CONT-ID              PIC S9(10)V USAGE COMP-3.
    10 GENDER               PIC S9(4) USAGE COMP.
    10 AGE-YEARS            PIC S9(4)V9(2) USAGE COMP-3.
    10 HIGHEST-EDU          PIC S9(4) USAGE COMP.
    10 ANNUAL-INVEST        PIC S9(9)V9(3) USAGE COMP-3.
    10 ANNUAL-INCOME        PIC S9(9)V9(3) USAGE COMP-3.
    10 ACTIVITY-LEVEL       PIC S9(4) USAGE COMP.
    10 CHURN                PIC S9(4) USAGE COMP.
```

4.2.1 Defining the data mapping

The first step is to make the VSAM file and its layout known to MDSS. To make the layout known, in turn, we first have to tell the tool where to find the copybook that describes it. In order to do this, we create a *virtual source library*.

To define the data mapping, complete the following steps:

1. In Scala IDE (or Data Server Studio), switch to the Data Server perspective and select the **Server** view.
2. Navigate to **Admin** → **Source Libraries** and double-click **Create Virtual Source Library** (Figure 4-4).

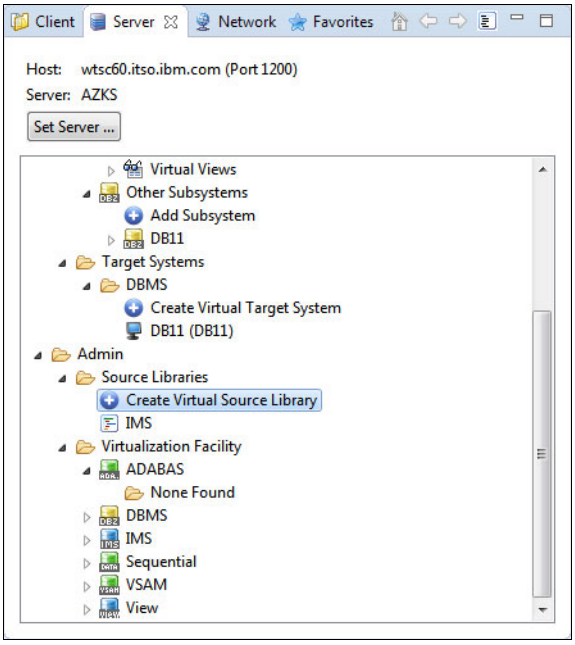


Figure 4-4 Create a new virtual source library

3. In the following dialog, enter a name and description of the library, and the data set name on z/OS (the full name of the partitioned data set (PDS) that contains the copybook describing the file layout), as shown in Figure 4-5.

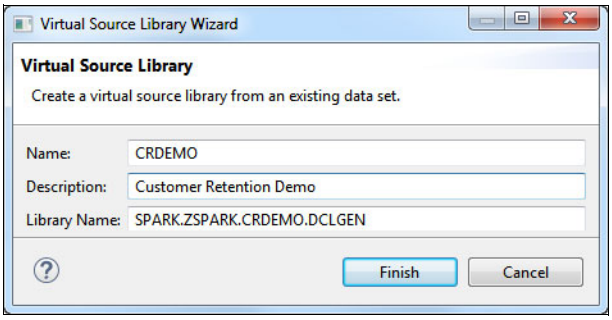


Figure 4-5 Virtual source library wizard

4. Now, we are ready to create the virtual table. While still in the Server view, navigate to **SQL** → **Data** → **AZKS** → **Virtual Tables** (your server name may be different from AZKS). Right-click and select **Create Virtual Table**. Select the **VSAM** wizard and click **Next**.
5. In the next panel, enter a name and description for the VSAM data set, where the name is not the physical VSAM file name but the name of the virtual table to be created, much like the name of a relational table in DB2 (Figure 4-6 on page 67). The **Arrays Handling** selection is relevant only for copybooks that describe repeating groups (OCCURS or OCCURS DEPENDING ON), which is not the case for our example.

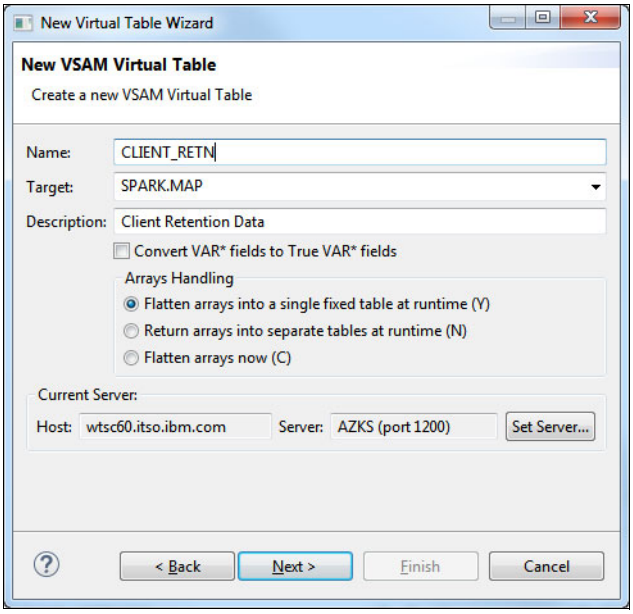


Figure 4-6 New VSAM virtual table

6. The next dialog asks us for the source library containing our copybook. Select the library that we created in the previous step. The **Source Library Members** list is populated with the available members in the library.
7. Select the relevant copybook (in our example, CLIENTB) and click **Download**. The file should appear in the **Downloaded Source Files** list. Click **Next**.
8. Now, Data Server Studio parses the copybook and asks us for the first and last field to be mapped. Normally, you will want to map every field, so select just the first field and leave the **Enable End Field selection** box unselected (Figure 4-7). Click **Next**.

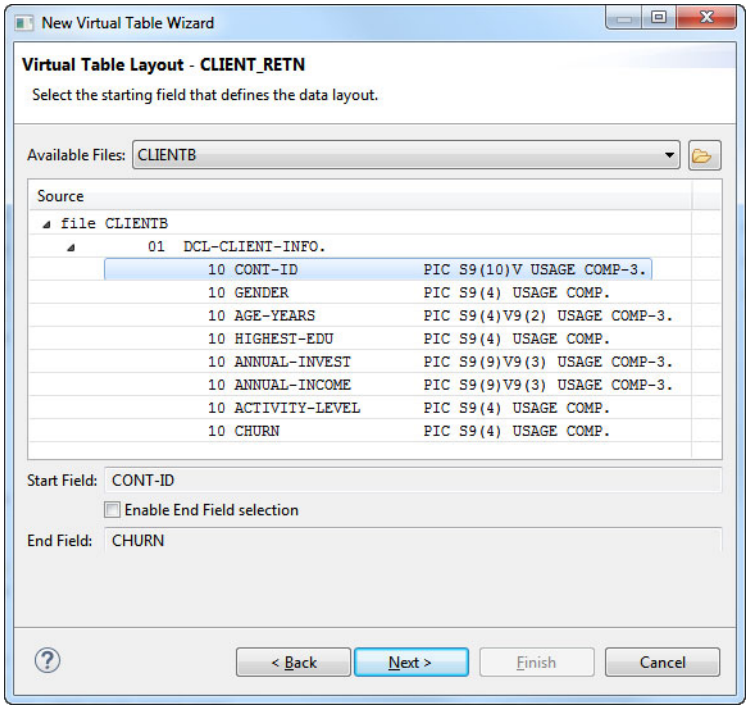


Figure 4-7 Defining the virtual table layout

9. On the next and final dialog, we specify the name of the VSAM cluster holding our data, in the Cluster Name field (in our example, `SPARK.CRDEMO.CLIENT.VSAMKSDS`).
10. The dialog prompts you to validate the name by clicking the **Validate** button. After successful validation, the **Get** button next to the Alternate indexes list becomes active. Because the VSAM cluster in our example does not have alternate indexes, the list remains empty.
11. Press **Finish**. The virtual table definition now appears in **SQL** → **Data** → **AZKS** → **Virtual Tables**. Expand it to see the mapped columns and their respective data types (Figure 4-8).

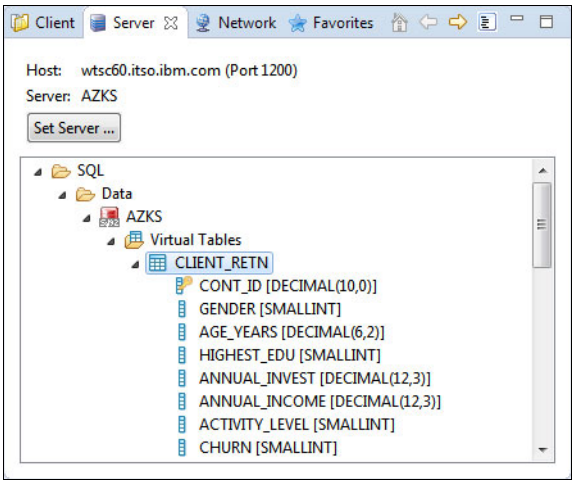


Figure 4-8 New virtual table definition with columns

12. After having created the virtual table, we can use Data Server Studio to verify the setup. Right-click the virtual table and select **Generate Query**. Data Server Studio now creates a new editor called `Generated.sql`, and runs the Structured Query Language (SQL) script after you have confirmed a warning message. For our CLIENT_RETN virtual table, the generated query looks like that shown in Example 4-4.

Example 4-4 Generated sample query for CLIENT_RETN virtual table

```
-- ----------------------------------------------------------------
-- This statement will return all rows and all columns from the
-- following table:
-- Name         : CLIENT_RETN
-- Catalog      : null
-- Schema       : AZKSQL
-- Remarks      : VSAM - SPARK.CRDEMO.CLIENT.VSAMKSDS
-- Tree Location: wtsc60.itso.ibm.com/1200/SQL/Data/AZKS/Virtual
Tables/CLIENT_RETN
-- The sql statement:
SELECT CONT_ID, GENDER, AGE_YEARS, HIGHEST_EDU, ANNUAL_INVEST, ANNUAL_INCOME,
  ACTIVITY_LEVEL, CHURN
FROM CLIENT_RETN limit 1000;
```

The result of the SQL execution opens in the SQL Results view (Figure 4-9 on page 69).

SQL Results | Server Trace | Console

	CONT_ID	GENDER	AGE_YEARS	HIGHEST_...	ANNUAL_...	ANNUAL_...	ACTIVITY...	CHURN
0	100952...	1	63.23	4	0.000	13035...	3	0
1	100952...	1	58.51	1	0.000	18267...	0	0
2	100952...	0	31.10	2	11119...	12927...	2	0
3	100952...	0	49.84	1	0.000	17867...	2	0
4	100952...	1	53.33	1	0.000	17576...	1	0
5	100952...	1	47.72	3	90419...	11156...	5	0
6	100952...	0	48.46	2	11258...	20142...	1	1
7	100952...	0	34.38	1	0.000	33727...	2	0
8	100952...	1	47.09	1	0.000	17678...	2	0
9	100952...	1	41.47	1	0.000	20747...	2	0

1000 rows | SQL Messages

Figure 4-9 SQL results view for CLIENT_RETN virtual table

Now that we verified the correct setup of the virtual table, we are ready to create a Spark application to do analysis on a VSAM file (Example 4-5).

Example 4-5 Simple analysis on a VSAM file

```
object VsamDemo {

  def printReport(avgDF: DataFrame) = {
    println("| Edu |          Age | Investments |       Income |")
    avgDF.collect.foreach { row =>
      val (edlvl, avgAge, avgAnnInv, avgAnnInc) =
        (row.getInt(0), row.getDecimal(1), row.getDecimal(2), row.getDecimal(3))
      println(f"| $edlvl%3d | $avgAge%11.2f | $avgAnnInv%11.2f | $avgAnnInc%11.2f |")
    }
  }

  def main(args: Array[String]) {
    val Array(mdssUrl, user, password) = args                                  // 1
    val props = new Properties                                                 // 2
    props.setProperty("user", user)
    props.setProperty("password", password)

    val conf = new SparkConf()                                                 // 3
      .setAppName("VsamDemo")
      .set("spark.app.id", "VsamDemo")
    val sc = new SparkContext(conf)
    val sqlContext = new SQLContext(sc)
    import sqlContext.implicits._

    val clientIncomeDF = sqlContext.read.jdbc(mdssUrl, "CLIENT_RETN", props) // 4
    val avgByEducationDF = clientIncomeDF
      .groupBy("HIGHEST_EDU")                                                  // 5
      .avg("AGE_YEARS", "ANNUAL_INVEST", "ANNUAL_INCOME")                      // 6

    printReport(avgByEducationDF)                                              // 7
  }
}
```

The numbers in Example 4-5 correspond to the following notes:

1. Initialize MDSS URL, username, and password from command line arguments. This will fail completely if the user did not supply enough arguments, but we do not supply proper command line parsing for the purpose of this sample.
2. Initialize a Properties object to hold connection properties.

3. Initialize Spark configuration, Spark context, and Spark SQL context.
4. Create a data frame to read from the CLIENT_RETN virtual table.
5. Group data by education level.
6. Compute the average values for columns AGE_YEARS, ANNUAL_INVEST, and ANNUAL_INCOME.
7. Finally, print a nicely formatted report.

4.2.2 Building and running the application

To build and package the application into a Java archive (JAR) file, we used sbt, as described in section 4.1.3, “Installing and using sbt” on page 63:

1. In the Package Explorer view in Eclipse, select your project.
2. Then, select **Run** → **External Tools** → **SBT** or click the External tools icon in the toolbar. When prompted for the sbt command to be run, enter `package` (which should have been set up as the default) and press **OK**. Now, sbt will download the necessary prerequisite packages, and will package the application into a Jar file into the target/scala-2.10 subfolder in your project.
3. Transfer the JAR file to z/OS UNIX, for example, using binary File Transfer Protocol (FTP).
4. Switch to a z/OS UNIX terminal session, and run the application using the following spark-submit command (on a single line):

   ```
   spark-submit --master spark://WTSC60.ITSO.IBM.COM:7077 --class VsamDemo
   sg24-8325-samples_2.10-1.0.jar 'jdbc:rs:dv://wtsc60.itso.ibm.com:1200;DBTY=DVS'
   lpres1 secret
   ```

   In this command, the following details apply:

   – `spark://WTSC60.ITSO.IBM.COM:7077` is the URL of the Spark master. Remember what we said in section 3.6, “Starting the Spark daemons” on page 57: The URL must be specified exactly as the master reports it, including case.

   – `VsamDemo` is the name of the main application class.

   – `sg24-8325-samples_2.10-1.0.jar` is the name of the application JAR file that we built and transferred in the previous step.

   – All remaining arguments (in this case, URL, user ID and password) are passed to the application’s main method.

For our sample data, running the application resulted in the following report (Example 4-6). Seems like good education can really make a big difference.

Example 4-6 Output from running Example 4-5 on page 69

```
| Edu |          Age | Investments |       Income |
|   1 |        48.62 |        0.00 |     20092.24 |
|   2 |        47.35 |    56591.62 |    127466.17 |
|   3 |        47.65 |    95469.12 |    130289.12 |
|   4 |        48.67 |     2705.53 |     26378.65 |
|   5 |        49.33 |    89593.93 |   1035014.64 |
```

4.3 Accessing sequential files and PDS members

MDSS also provides access to sequential files and PDS(E) members in much the same way as it does for VSAM. Therefore, we did not develop a code example specifically demonstrating sequential file access.

However, it is important to stress that behind the scenes, the MDSS server has to be considerably “smarter” to support sequential files in an efficient way. For example, reading from VSAM in a parallel fashion is much more straightforward because MDSS can figure out what indexes are defined on the VSAM dataset, and can use that information to schedule several reader threads, each reading a certain key range.

While access to VSAM files is nearly 100% IBM System z Integrated Information Processor (zIIP) offloadable, access to sequential files is not, but still in the high 90% range (typically, around 97%).

4.4 Accessing IBM DB2 data as a DataFrame

In contrast to accessing VSAM data, you have a choice when it comes to access IBM DB2 data from a Spark application. You can either use the DB2-supplied Java Database Connectivity (JDBC) driver (officially called *IBM Data Server Driver for JDBC and SQLJ*, unofficially known as *JCC driver*), or you can use the MDSS driver.

From a performance perspective, there is little difference between the two choices. You might want to use the JCC driver for “DB2 only” queries that do not select from other data sources. On the other hand, if you want to join DB2 data with VSAM data, you would have to use the MDSS driver.

Example 4-7 shows a code snippet to make a DB2 table or view available as a Spark DataFrame, using the JCC driver.

Example 4-7 Accessing a DB2 table as a DataFrame

```
val props = new Properties
props.setProperty("user", user)
props.setProperty("password", password)
val url = "jdbc:db2://wtsc60.itso.ibm.com:38050/DB11"

val spPayDF = sqlContext.read.jdbc(url, "CARDUSR.SPPAYTB", props)
spPayDF.registerTempTable("SPPAYDF")
sqlContext.sql("SELECT * FROM SPPAYDF").show()
```

The contents of a DataFrame can also be written back to a database table, using the `DataFrame.write.jdbc()` method. You might want to do this, for example, to persist the results of a complex analysis for consumption from online applications.

Unfortunately, however, this does not currently work with DB2 for z/OS. The reason is that Spark tries to find out whether the target table exists. It does so using SQL syntax that is not supported in DB2 and results in an error (“`SELECT 1 FROM $table LIMIT 1`”), which in turn causes Spark to erroneously believe the table does not exist and to try and create it, which finally results in an exception because it actually does.

A possible workaround is to use the `JdbcUtils.saveTable(df, url, table, props)` method. We demonstrate this in section 4.9, “JavaScript Object Notation” on page 79.

4.5 Joining DB2 data with VSAM

Quite often, you will want to join relational data from DB2 with non-relational data, for example, from a VSAM file.

Using MDSS and Spark RDDs or DataFrames, this is quite easy to do, as we show in the following examples.

Note that with MDSS, there are essentially two ways to perform the join:

- Read each data source (DB2 data and VSAM data) into their own DataFrame, and perform the join using Spark transformations or Spark SQL joins.
- Create a virtual view in MDSS that joins DB2 and VSAM data, and read the joined data into a single DataFrame.

We show both methods in the following examples.

The first example performs a DataFrame join using regular SQL syntax. It reads both tables to be joined into a DataFrame, registers each DataFrame as a temporary table, and creates a third DataFrame which joins the two (Example 4-8).

Example 4-8 Join DB2 with VSAM data using Spark DataFrames

```
val db2DF = sqlContext.read.jdbc(jdbcUrl, "CARDUSR.SPPAYTB", prop)                  // 1
db2DF.registerTempTable("DTABLE")

val vsamDF = sqlContext.read.jdbc(mdssUrl, "CLIENT_RETN", prop)                     // 2
vsamDF.registerTempTable("VTABLE")

println("Simple join of VSAM with DB2")
sqlContext.sql("""SELECT V.CONT_ID                                                  // 3
                       , GENDER
                       , HIGHEST_EDU
                       , ANNUAL_INCOME
                       , ACAUREQ_HDR_CREDTT
                       , ACAUREQ_AUREQ_ENV_A_ID_ID
                   FROM VTABLE V
                      , DTABLE D
                  WHERE V.CONT_ID = D.CONT_ID""").show
```

The numbers in Example 4-8 correspond to the following notes:

1. Read the first table (a DB2 table called SPPAYTB) into a DataFrame, and register the DataFrame as a temporary table.
2. Do the same for the VSAM virtual table, CLIENT_RETN. Note that the syntax is exactly the same, we just use another URL that points to the MDSS server rather than to DB2.
3. Join the two registered temporary tables into a new DataFrame.

You can also perform the join operation with an explicit transformation, and register the result of the join as a new, joined, temporary table (Example 4-9).

Example 4-9 Join into a registered temporary table

```
println("Join from a registered temp table")
val joinedDF = vsamDF
  .join(db2DF, vsamDF("CONT_ID") === db2DF("CONT_ID"))                          // 1
  .drop(vsamDF.col("CONT_ID"))                                                  // 2
joinedDF.registerTempTable("JoinedTable")                                       // 3
sqlContext.sql("""SELECT CONT_ID                                                // 4
                       , sum(ACAUREQ_AUREQ_TX_DT_TTLAMT)          as total_txn_amount
                       , count(*) / 365.0                         as avg_daily_txns
                       , count(*)                                 as total_txns
                       , sum(ACAUREQ_AUREQ_TX_DT_TTLAMT)/count(*) as avg_txn_amount
                    FROM JoinedTable
                   GROUP BY CONT_ID""").show
```

The numbers in Example 4-9 correspond to the following notes:

1. Rather than using Spark SQL, this example performs the join using the Scala application programming interface (API).
2. To avoid a duplicate CONT_ID column, we drop one of them.
3. We register the joined DataFrame as a temporary table.
4. Next, query the result in Spark SQL.

Of course, you can also use SQL JOIN syntax to perform the join, and you can use grouping and aggregation on the joined tables (Example 4-10).

Example 4-10 Join using an INNER JOIN clause, with grouping and aggregation

```
println("Join with grouping")
  sqlContext.sql("""SELECT V.HIGHEST_EDU
                        , sum(ACAUREQ_AUREQ_TX_DT_TTLAMT)           as total_txn_amount
                        , count(*) / 365.0                          as avg_daily_txns
                        , count(*)                                  as total_txns
                        , sum(ACAUREQ_AUREQ_TX_DT_TTLAMT)/count(*)  as avg_txn_amount
                     FROM DTABLE D
                    INNER JOIN VTABLE V
                       ON D.CONT_ID = V.CONT_ID
                    GROUP BY HIGHEST_EDU""").show
```

4.6 IBM IMS data to DataFrames

From an application development perspective, accessing IBM Information Management System (IBM IMS) data is similar to accessing DB2 data, and again, you have a choice. You can use either the DSS driver or the standard IMS Universal JDBC driver, which requires the IMS Open Database Manager (ODBM) infrastructure. For more information, see *IMS 11 Open Database*, SG24-7856:

http://www.redbooks.ibm.com/abstracts/sg247856.html

In the case of IMS VSAM-based databases, rather than going through IMS for data access, the driver was successfully tested doing a physical read from the underlying VSAM sets holding the IMS data.

However it is recommended to use the IMS DB access feature, which connects like an IBM Customer Information Control System (IBM CICS) server or COBOL Batch Message Processor (BMP) to the IMS Control Region, and runs its queries based on the database resource adapter (DRA) or ODBM connection with IMS as the database manager.

Just like with VSAM, you create virtual tables and, optionally, virtual views to access IMS data. See 3.2.1, “Installing the MDSS started task” on page 32 for general IMS setup, and 3.2.3, “Configuring access to IMS databases” on page 38 for detailed information about mapping IMS data.

4.7 System log

You can access the z/OS system log via MDSS as a virtual table, making it easy to convert it to an Resilient Distributed Dataset (RDD) or to a DataFrame in much the same way as a DB2 table.

If you want to use this facility, make sure that MDSS is configured for SYSLOG (see 3.2.1, “Installing the MDSS started task” on page 32).

The SYSLOG facility adds four virtual tables:

- OPERLOG_SYSLOG

  Use this virtual table if your system log is in traditional Job Entry Subsystem (JES2) hardcopy format.
- OPERLOG_MDB
- OPERLOG_MDB_MDB_CONTROL_OBJECT
- OPERLOG_MDB_MDB_TEXT_OBJECT

  Use these virtual tables if your system log resides in a z/OS logstream.

The OPERLOG_SYSLOG virtual table has the columns shown in Table 4-1.

Table 4-1 Layout of SYSLOG virtual table

Column name	Data type	Description
SYSLOG_SEQUENCE_NUMBER	VARCHAR(10)	Message sequence number
SYSLOG_RECORD_TYPE	VARCHAR(1)	Type of record. Valid types are: N Single-line message W Single-line message with reply M First line of a multi-line message L Multi-line message label line D Multi-line message data line E Multi-line message data/end line S Continuation of previous line O Log command input X Non-hardcopy or LOG command source
SYSLOG_REQUEST_TYPE	VARCHAR(1)	C Command issued by operator R Command response message I Internally issued command U Command from unknown console ID
SYSLOG_ROUTING_CODES	VARCHAR(7)	

Column name	Data type	Description
SYSLOG_SYSTEM_NAME	VARCHAR(8)	System name
SYSLOG_DATE_TIME	VARCHAR(26)	Message date and time
SYSLOG_JOBID	VARCHAR(8)	Console name, job ID, or multi-line ID
SYSLOG_REQUEST_FLAGS	VARCHAR(8)	User exit flags
SYSLOG_MSG_TEXT	VARCHAR(75)	Message text
LS_TIMESTAMP	BINARY(8)	Message date and time, in internal format

Being able to read, process, and analyze SYSLOG data in a consistent manner can prove very useful to gain operational insight. For example, you might want to consolidate SYSLOG data with log data from your applications running in IBM WebSphere Application Server, in order to identify potential security weaknesses or performance issues.

Example 4-11 demonstrates how to read SYSLOG records and group them by component identifier and the day of the week.

Example 4-11 Reading and grouping SYSLOG records

```
case class SyslogRecord(sysName: String, timestamp: LocalDate,                    // 1
                        jobID: String, message: String) {
  def msgId = message.substring(0, message.indexOf(' '))
  def component = message.substring(1, 4)
}

val syslogTSFormat = DateTimeFormatter.ofPattern("yyyy/MM/dd kk:mm:ss.SSSSSS")       // 2

def analyzeSyslog(sqlContext: SQLContext, mdssUrl: String, props: Properties) = {

  val sql = """
  (SELECT SYSLOG_SYSTEM_NAME, SYSLOG_DATE_TIME, SYSLOG_JOBID, SYSLOG_MSG_TEXT        // 3
     FROM OPERLOG_SYSLOG
    WHERE SYSLOG_RECORD_TYPE IN ('M', 'N')
      AND SYSLOG_REQUEST_TYPE = ' ')"""
  sqlContext.read.jdbc(mdssUrl, sql, props)
    .map { row =>
      SyslogRecord(row.getString(0),
        parseTS(row.getString(1), syslogTSFormat),
        row.getString(2),
        row.getString(3))
    }
    .groupBy { record => (record.component, record.timestamp.getDayOfWeek.getValue) } // 4
    .map { case ((component, dayofweek), msgs) => (component, dayofweek, msgs.size) } // 5
    .sortBy(_._3, false)                                                              // 6
}

analyzeSyslog(sqlContext, mdssUrl, Map("user" -> user, "password" -> password))
    .collect()
    .foreach(println)
```

The numbers in Example 4-11 on page 75 correspond to the following notes:

1. This case class will hold the relevant data from the log, including the timestamp and the message text. Two utility methods extract the message ID and the component that logged the message from the message text.
2. We construct a DateTimeFormatter that understands the timestamp format of SYSLOG.
3. Here, we let MDSS do the filtering, using regular SQL syntax. Note that the second argument to the JDBC **read()** method is usually a table name (and the documentation suggests that it has to be), but it is perfectly legal to pass a subselect in parentheses.
4. We group by component identifier and day of week.
5. Next, map to a triple consisting of component, day of week, and number of messages.
6. Finally, sort by the number of messages, in descending order.

When we ran the sample on our system, we saw the output shown in Example 4-12.

Example 4-12 Partial output from running the SYSLOG sample

```
(ERB,4,12070)
(ERB,3,11871)
(ERB,5,11782)
(ERB,2,11724)
(ERB,7,11667)
(ERB,1,11666)
(ERB,6,11620)
(IEF,1,11359)
(IEF,4,9992)
(IEF,3,9900)
(IEF,5,9866)
(IEF,2,7687)
(IEF,6,7101)
(IEF,7,5865)
(BPX,3,4950)
(BPX,1,4942)
[ ... ]
```

So, on this system, most of the messages being logged had the prefix ERB, which happens to be the component identifier for IBM Resource Management Facility (IBM RMF™).

Tip: A list of message prefixes and their corresponding component can be found in z/OS System messages, Volume 1, section “Message directory”, or in IBM Knowledge Center for z/OS 2.1:

https://www.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.ieam100/msgpre.htm

4.8 SMF data

MDSS can access System Management Facilities (SMF) data from an SMF dump data set. Future versions might support access to in-memory SMF data, so there would no longer be the need to unload the SMF data first.

To unload SMF data, use one of the standard SMF dump programs, IFASMFDP for SMF data sets or IFASMFDL for log streams. Example 4-13 shows how to dump SMF data from a log stream.

Example 4-13 Dumping SMF data from a log stream to a dump dataset

```
//SMFDUMP  JOB (999,POK),'DUMP SMF DATA',REGION=0M,NOTIFY=&SYSUID
/*
// SET DUMPDS=&SYSUID..SMFDUMP.D160419
/*
//DELETE   EXEC  PGM=IEFBR14
//DUMPDS   DD DSN=&DUMPDS,
//    DISP=(MOD,DELETE,DELETE),UNIT=SYSDA,SPACE=(TRK,1)
/*
//DUMP      EXEC PGM=IFASMFDL
//SYSPRINT  DD SYSOUT=*
//DUMPOUT   DD DSN=&DUMPDS,DISP=(,CATLG),
//             SPACE=(CYL,(1500,1500),RLSE)
//SYSIN     DD *,SYMBOLS=CNVTSYS
  LSNAME(IFASMF.DEFAULT,OPTIONS(DUMP))
  OUTDD(DUMPOUT,TYPE(0:255),START(1500),END(1700))
  DATE(2016097,2016097)
  SOFTINFLATE
```

With MDSS, you access SMF data via a predefined set of *virtual tables* or *virtual views*, where each type of SMF record is mapped by one or more virtual tables. In most of the cases, there will be more than one table for each record type, corresponding to the respective subtypes and sections. For example, virtual table SMF_07203_R723WMS maps SMF record type 72, subtype 3, Workload Manager control section.

Each virtual table that does not map a control section has a fixed set of columns corresponding to the standard SMF header (Table 4-2).

Table 4-2 Common SMF header fields

Column name	Data type	Description
SMF*x*LEN	INTEGER	Length of SMF record
SMF*x*SEG	INTEGER	SMF segment descriptor
SMF*x*FLG	VARCHAR(8)	Flags indicating IBM MVS™ product level
SMF*x*RTY	SMALLINT	Record type
SMF*x*TME	TIMESTAMP	Time when record was cut. Note that MDSS automatically converts from SMF format to SQL timestamp format.
SMF*x*SID	VARCHAR(4)	System name
SMF*x*SSI	VARCHAR(4)	Subsystem name
SMF*x*STY	SMALLINT	Record subtype

Each virtual table has many additional columns specific to that particular SMF record type. For a description of each column, see the documentation in the IBM Knowledge Center, section *z/OS MVS System Management Facilities (SMF) Records*:

https://www.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.ieag200/records.htm

In Table 4-3, we list all IBM z/OS Multiple Virtual Storage (MVS) SMF record types for which MDSS defines a corresponding virtual table.

Table 4-3 Supported SMF record types

SMF record type	Subtype(s)	Description
00	-	Initial program load (IPL)
06	-	JES writers, IBM Print Services Facility™ (PSF), and IP IBM PrintWay™
14	-	INPUT or RDBACK Data Set Activity
15	-	OUTPUT, UPDAT, INOUT, or OUTIN Data Set Activity
26	-	JES2/JES3 Job Purge
30	-	Common address space work
42	1, 2, 3, 5, 6, 9, 15–25	MVS data in virtual (DIV) objects and virtual lookaside facility (VLF) statistics
60	-	VSAM Volume Data Set Updated
70	-	RMF Processor Activity
71	1	RMF Paging Activity
72	3, 4, 5	Workload Activity, Storage Data, and Serialization Delay
73	1	RMF Channel Path Activity
74	1–9	RMF activity of several resources: ► Device activity ► Cross-system coupling facility (XCF) activity ► OMVS kernel activity ► Coupling facility (CF) activity ► Cache Subsystem Device activity ► Hierarchical file system (HFS) statistics ► IBM FICON® Director statistics ► Enterprise Disk System statistics ► Peripheral Component Interconnect Express (PCIe) activity
75	1	RMF Page Data Set Activity
76	1	RMF Trace Activity
77	1	RMF Enqueue Activity
78	2, 3	RMF Virtual Storage and I/O Queuing Activity
79	1–7, 9, 11, 12, 14, 15	RMF Monitor II Activity
80	-	Security Product Processing
81	-	IBM Resource Access Control Facility (IBM RACF) Initialization
83	1, 2	RACF Audit Record For Data Sets
89	1, 2	Usage Data
100	-	DB2 Statistics
101	-	DB2 Accounting

SMF record type	Subtype(s)	Description
110	-	CICS TS for z/OS Statistics
115	1, 2	IBM MQ for z/OS Statistics
116	1, 2, 10	IBM MQ for z/OS Accounting
118	1–5, 20, 21, 70–75	Transmission Control Protocol/Internet Protocol (TCP/IP) statistics
249		SMF records written by the MDSS server itself. See *IBM z/OS Platform for Apache Spark User's Guide*, SC27-8450 for a description of the record layout.

4.9 JavaScript Object Notation

While this section is not really specific to Spark on z/OS, you will probably need to combine z/OS data sources with JavaScript Object Notation (JSON) data. For example, you may have non-z/OS systems with a Representational State Transfer (REST)-based API that returns its payload in JSON format, or you receive bulk data from external agencies as JSON.

For this example, we downloaded freely available bulk geographical and weather information from openweathermap.org, namely which is a list of cities of the world with location and country code:

`http://bulk.openweathermap.org/sample/city.list.json.gz`

We also downloaded weather forecasts data for each city:

`http://bulk.openweathermap.org/sample/hourly_16.json.gz`

Processing JSON data as a Spark DataFrame is not much different from processing relational data, except for the fact that JSON is hierarchical in nature and has an array type, while DataFrames are column oriented. You can refer to nested items through dot notation, and you can "flatten" JSON arrays by using the `explode` method which, in relational terms, is much like an inner join with a dependent table.

For JSON data, it is quite useful to use the Spark shell to learn about the data schema. Spark, when reading the JSON file, tries to infer the schema from the data, and you can print it in easily readable tree form to learn about the data layout (Example 4-14).

Example 4-14 Data schema in tree format

```
scala> val h16 = sqlContext.read.json("hourly_16.json.gz")
h16: org.apache.spark.sql.DataFrame = [...]

scala> h16.printSchema
root
 |-- city: struct (nullable = true)
 |    |-- coord: struct (nullable = true)
 |    |    |-- lat: double (nullable = true)
 |    |    |-- lon: double (nullable = true)
 |    |-- country: string (nullable = true)
 |    |-- id: long (nullable = true)
 |    |-- name: string (nullable = true)
 |-- data: array (nullable = true)
 |    |-- element: struct (containsNull = true)

[ ... ]

 |    |    |-- wind: struct (nullable = true)
 |    |    |    |-- deg: double (nullable = true)
 |    |    |    |-- speed: double (nullable = true)
 |-- time: long (nullable = true)
```

Tip: As you can see in Example 4-14, Spark supports reading compressed files, so there is no need to run **gunzip** on the sample file.

Example 4-15 demonstrates how to read and process the sample weather forecast dataset.

Example 4-15 Reading and processing JSON data

```
object JsonDemo extends App {                                          // 1

  val conf = new SparkConf()
    .setAppName("JSON sample")
    .set("spark.app.id", "JSON sample")
  val sc = new SparkContext(conf)
  val sqlContext = new SQLContext(sc)
  import sqlContext.implicits._

  val hourly16 = sqlContext.read.json("hourly_16.json.gz")             // 2
    .withColumn("data", explode($"data"))                              // 3
    .select("city.country", "city.name", "data.wind.speed")            // 4
    .rollup("country", "name").avg("speed")                            // 5
    .sort("country", "name")                                           // 6

  hourly16.show(100)
}
```

The numbers in Example 4-15 on page 80 correspond to the following notes:

1. For this simple example, we did not even bother to write a main method but used the standard Scala App trait.
2. We read the JSON data into a data frame. As already mentioned, Spark is smart enough to figure out that the data is compressed.
3. We flatten the data array into a column using `explode()`.
4. We are interested only in country, city name, and the wind speed, so we select these from the DataFrame.
5. Next, we group with rollup on columns country and name, giving the average values including grand totals.
6. Finally, we sort the result by country and name.

As a final example, Example 4-16 shows how to store parsed JSON data into a DB2 table.

Example 4-16 Storing parsed JSON data into a DB2 table

```
/**
 * Reads "city list" JSON file and persists it to a DB2 table.
 *
 * Create the table using the following DDL template:
 *
 * {{{
 * CREATE TABLE CITIES( ID      INTEGER     NOT NULL                        // 1
 *                    , LAT     FLOAT       NOT NULL
 *                    , LON     FLOAT       NOT NULL
 *                    , COUNTRY CHAR(2)     NOT NULL
 *                    , NAME    VARCHAR(80) NOT NULL
 *                    )
 *        NOT LOGGED
 *        COMPRESS YES
 * }}}
 */
object JsonDemoWithDB2 extends App {
  /* ... setup code not shown ... */

  val citiesDF = sqlContext.read.json("city.list.json.gz")                  // 2
  val citiesDF2 = citiesDF.select($"_id".as("id"),                          // 3
                                  $"coord.lat",
                                  $"coord.lon",
                                  $"country",
                                  $"name")
  saveTable(citiesDF2, url, "CITIES", props)                                // 4
}
```

The numbers in Example 4-16 correspond to the following notes:

1. The column names and data types in the destination table (CITIES) must match the respective columns in the DataFrame that we are about to save. Since the save happens in a single transaction, we chose to create the table NOT LOGGED.
2. As in the previous example, we read the JSON file into a DataFrame.
3. We rename the _id column, and put the latitude and longitude information into separate columns.
4. Finally, we persist the DataFrame to DB2 using `JdbcUtils.saveTable`.

4.10 Extensible Markup Language

Finally, we show a small example of how to process Extensible Markup Language (XML) data. Again, this is not specific to z/OS, but can be a useful starting point for your own XML processing with Spark.

The application (Example 4-17 on page 83) processes Single Euro Payments Area (SEPA) payments information in PAIN.001 format. It reads an input file containing multiple PAIN.001 documents, one line per document.

> **Note:** The SEPA file formats are defined in International Organization for Standardization (ISO) Standard 20022. PAIN stands for **Pa**yment **In**formation, and one has to acknowledge the subtle humor of the person at ISO who came up with that abbreviation. The PAIN format is quite complex.

A full explanation of the PAIN.001 format is not in the scope of this book, but Figure 4-10 shows part of a sample PAIN.001 document that should suffice to show the structure.

```
<Document xmlns="urn:iso:std:iso:20022:tech:xsd:pain.001.001.03">
    <CstmrCdtTrfInitn>
        <GrpHdr>
            <MsgId>ABC/160427/CCT001</MsgId>
            <CreDtTm>2016-04-27T20:04:28.415473</CreDtTm>
            <NbOfTxs>3</NbOfTxs>
            <CtrlSum>1390255.61</CtrlSum>
            <InitgPty>..
        </GrpHdr>
        <PmtInf>
            <PmtInfId>001</PmtInfId>
            <PmtMtd>TRF</PmtMtd>
            <PmtTpInf>..
            <ReqdExctnDt>2016-04-27</ReqdExctnDt>
            <Dbtr>..
            <DbtrAcct>..
            <DbtrAgt>..
            <CdtTrfTxInf>
                <PmtId>
                    <InstrId>1</InstrId>
                    <EndToEndId>1</EndToEndId>
                </PmtId>
                <Amt>
                    <InstdAmt Ccy="EUR">577.37</InstdAmt>
                </Amt>
                <CdtrAcct>
                    <Id>
                        <Othr>
                            <Id>688834963</Id>
                            <SchmeNm>
                                <Prtry>ITSO</Prtry>
                            </SchmeNm>
                        </Othr>
                    </Id>
                </CdtrAcct>
            </CdtTrfTxInf>
            <CdtTrfTxInf>
                <PmtId>
                    <InstrId>2</InstrId>
                    <EndToEndId>2</EndToEndId>
                </PmtId>
                <Amt>
                    <InstdAmt Ccy="EUR">490.79</InstdAmt>
                </Amt>
```

Figure 4-10 Sample PAIN.001 XML document

In the example, we want to process payments information, extracting the currency, amount and creditor account, as shown in Example 4-17.

Example 4-17 Reading payments information from PAIN.001 into a DataFrame

```
object XMLDemo extends App {

  case class Payment(ccy: String, amt: BigDecimal, acct: String)                    // 1

  def getPayments(xml: String): Seq[Payment] = {                                    // 2
      XML.loadString(xml) \ "CstmrCdtTrfInitn" \ "PmtInf" \ "CdtTrfTxInf" map (tx =>  // 3
        Payment(tx \ "Amt" \ "InstdAmt" \ "@Ccy" text,                              // 4
          BigDecimal(tx \ "Amt" \ "InstdAmt" text),
          tx \ "CdtrAcct" text))
  }

  val conf = new SparkConf()
    .setAppName("JSON sample")
    .set("spark.app.id", "JSON sample")
  val sc = new SparkContext(conf)
  val sqlContext = new SQLContext(sc)
  import sqlContext.implicits._

  val Array(pain001file) = args

  val paymentsDF = sc
    .textFile(pain001file)                                                          // 5
    .flatMap(getPayments)                                                           // 6
    .toDF()                                                                         // 7

  paymentsDF.show()
  paymentsDF.registerTempTable("PAYMENTS")
  sqlContext.sql("""SELECT ccy
                         , AVG(amt) AS avgAmt
                      FROM PAYMENTS
                     GROUP BY ccy""").show()
}
```

The numbers in Example 4-17 correspond to the following notes:

1. This case class holds the payments data (currency, amount, and creditor account).
2. This helper function converts a PAIN.001 document into a sequence of Payment instances.
3. It does so by parsing the XML, and navigating to the repeating group of payments (the `CdtTrfTxInf` element).
4. It then creates a Payment object for each occurrence of the repeating group.
5. The main function reads the input file into an RDD, each row from the RDD holding one PAIN.001 XML document.
6. We convert each row into a sequence of Payment objects. By using `.flatMap()`, the resulting sequences are spliced together into one single sequence.
7. We now have an RDD[Payment], and convert that into a DataFrame.

Our sample data produced the output shown in Example 4-18.

Example 4-18 Output from running Example 4-17 on page 83

```
LPRES1:/u/lpres1: >spark-submit --class XMLDemo sg24-8325-samples_2.10-1.0.jar
pain001s.xml
+---+--------------------+-------------+
|ccy|                 amt|         acct|
+---+--------------------+-------------+
|EUR|577.3700000000000...|688834963ITSO|
|EUR|490.7900000000000...|989104874ITSO|
|EUR|147.4500000000000...|650441087ITSO|
|SEK|71300.00000000000...|688834963ITSO|
|EUR|2090.000000000000...|989104874ITSO|
|EUR|47.45000000000000...|650441087ITSO|
+---+--------------------+-------------+

+---+--------------------+
|ccy|              avgAmt|
+---+--------------------+
|SEK|71300.00000000000...|
|EUR|670.6120000000000...|
+---+--------------------+
```

4.11 Submit Spark jobs from z/OS applications

Spark applications can be written in Java and Scala. Both languages run well in the z/OS Java virtual machine (JVM). It might be a requirement that Spark Jobs need to be invoked from existing applications currently running on z/OS, either batch or online.

It was successfully tested to submit Scala or Java Spark applications from IMS BMP, IMS transactions, and Java Batch on z/OS to the Spark cluster. Because this is all pure Java, it should also be possible from WebSphere z/OS, CICS, and DB2 Java Stored Procedures. See Figure 4-11 on page 85.

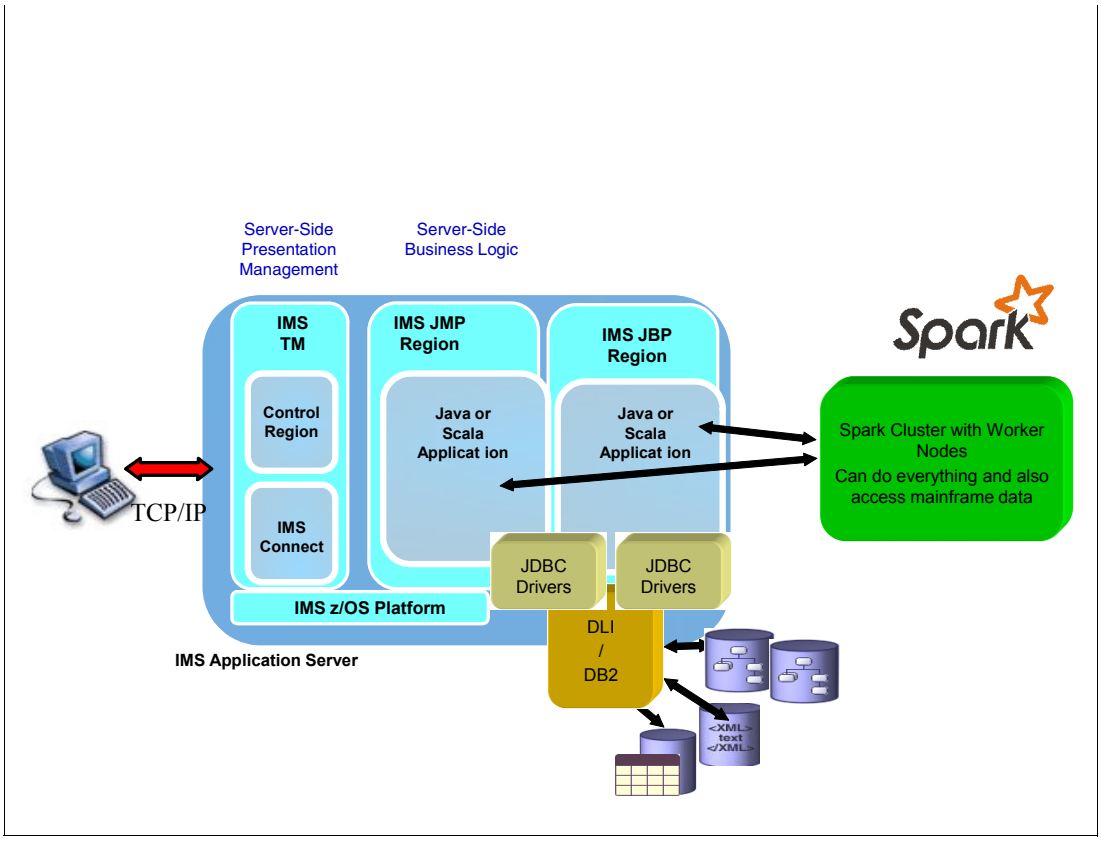


Figure 4-11 Call Spark from z/OS Java applications (using IMS as an example)

Furthermore with Java interoperability (for example, COBOL calls Java and PL/I calls Java) it was also successfully tested to invoke a Scala Spark application that submits a job to a Spark cluster (see Figure 4-12).

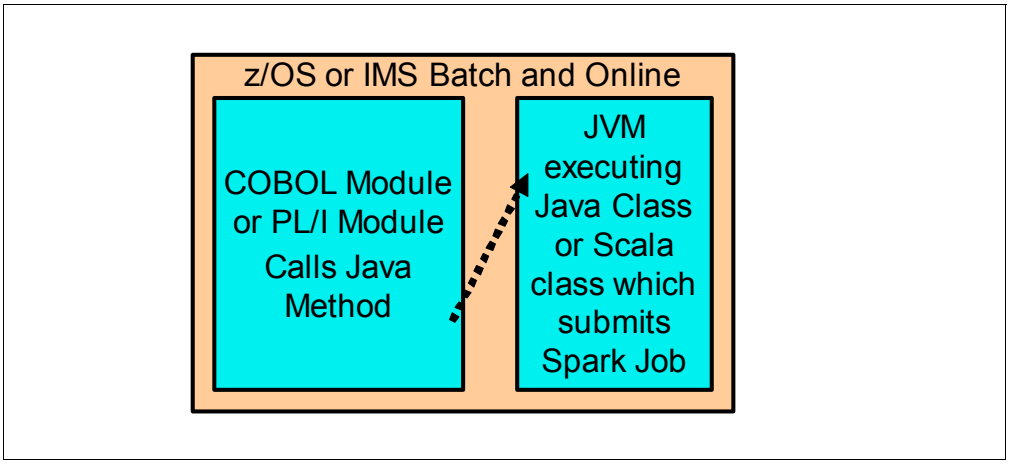


Figure 4-12 Java Interoperability on z/OS

This allows a tight integration of Spark analytics, for example into existing Job networks and existing applications. It should be noted that for transactional environments the Spark Job that was submitted to the cluster, is not part of the UOW of the caller. For example, if an IMS BMP submits a Spark job and waits for its results, the data modifications of the Spark Job running in the Spark Cluster are not part of the Unit of Work in the IMS BMP.

Due to the multithreaded nature of the Spark execution, it is not possible to run the Spark job in local mode (`.setMaster("local")`). This was tested, but is not working because most of the application environments on z/OS do not support multiple threads within one UOW or multiple DB connections from the same UOW (for example, IMS).

If there is a need to exchange data between the calling z/OS application and the Spark job, that is best done by putting the data into a database, such as IMS or DB2, and let the Spark job access and change the data in this database for returning results.

5

Production integration

This chapter gives you a starting point for production integration and performance tuning.

This chapter describes the following topics:

- 5.1, “Production deployment” on page 88
- 5.2, “Running Spark applications from z/OS batch” on page 88
- 5.3, “Starting Spark master and workers from JCL” on page 89
- 5.4, “System level tuning” on page 89

5.1 Production deployment

All of the applications in the IBM z/OS environment are considered mission-critical. Therefore, an IBM z/OS Platform for Apache Spark implementation on z/OS requires similar treatment.

Following are the major factors to remember:

- Business continuity, if something goes wrong with Spark
- Troubleshooting and fixing the problems rather quickly
- Monitoring the performance trends

You will see the steps involved in deploying Spark applications into production for a sample use case in this chapter.

5.2 Running Spark applications from z/OS batch

You may want to run your Spark applications in a scheduled fashion, for example, once per day. In environments other than z/OS, you would probably use a tool such as `cron` to schedule execution of an application.

On z/OS, however, scheduling is typically done using a job scheduling package that runs z/OS batch jobs rather than z/OS UNIX shell scripts. Therefore, you will probably want to run Spark applications from a batch job.

This can easily be done with the standard z/OS BPXBATCH utility. In Example 5-1, we show how to run the IBM Virtual Storage Access Method (VSAM) example from 4.2, “Accessing VSAM data as an RDD” on page 65 from a batch job.

Example 5-1 Submitting a Spark application from z/OS batch job control language (JCL)

```
//SPARKSUB JOB ‘Spark VSAM’,REGION=0M,NOTIFY=&SYSUID
//SUBMIT  EXEC PGM=BPXBATCH
//STDPARM DD *
sh /usr/lpp/IBM/Spark/bin/spark-submit
--master spark://WTSC60.ITSO.IBM.COM:7077
--class VsamDemo
sg24-8325-samples_2.10-1.0.jar
'jdbc:rs:dv://wtsc60.itso.ibm.com:1200;DBTY=DVS'
lpres1
secret
//STDOUT  DD SYSOUT=*
//STDERR  DD SYSOUT=*
```

The BPXBATCH utility can run either a z/OS UNIX program, or a z/OS UNIX script. The parameters to the program or script can be passed either in a PARM parameter, or they can be specified in a file allocated to the STDPARM ddname. Because the parameters for a Spark application are very likely to exceed the JCL limitation of 100 characters for the PARM parameter, we use the STDPARM ddname.

As the example shows, the parameters must begin with **`sh`** followed by the name of the script to be invoked, in our case the **`spark-submit`** script. We specify the full path to spark-submit so that the job can run successfully whether or not the submitting user has spark-submit in their PATH.

The remaining parameters are the parameters to the script itself, in much the same way as you would specify them on a z/OS UNIX command line, except that each argument is on its own line. Note that, just like in an interactive shell, parameters containing special characters, such as the semicolon, must be surrounded by quotation marks sothat the special characters are not being interpreted by the shell.

> **Tip:** If you use the IBM Interactive System Productivity Facility (ISPF) editor to create the JCL, make sure to set sequence numbering off by typing `NUMBER OFF` on the ISPF command line before you begin typing the data. If sequence numbers already exist, type `UNNUM` to remove them and then type `NUMBER OFF`.
>
> Having sequence numbers in JCL that runs z/OS UNIX commands is a frequent source of confusion, especially because you typically only see them in your ISPF edit session when you scroll to the right.

For a more detailed description of BPXBATCH, and a detailed description of how to define STDPARM, refer to the IBM Knowledge Center for BPXBATCH utility:

http://www.ibm.com/support/knowledgecenter/SSLTBW_2.2.0/com.ibm.zos.v2r2.cbcux01/bpxbatr.htm

5.3 Starting Spark master and workers from JCL

In a similar fashion to submitting Spark applications from JCL, you can also start the Spark master and worker processes from a job, as we show in Example 5-2.

Example 5-2 Starting Spark master and workers from JCL

```
//SPARKMST JOB 'SPARK MASTER',REGION=0M,USER=SPARKID,NOTIFY=&SYSUID
//************************************************************
//* Start Spark Master and Workers
//************************************************************
//MASTER  EXEC PGM=BPXBATCH,
// PARM='sh /usr/lpp/IBM/Spark/sbin/start-all.sh'
//STDOUT DD SYSOUT=*
//STDERR DD SYSOUT=*
```

Note that we run the job with a dedicated user ID (SPARKID). In order to successfully submit the job, the submitter needs to have surrogate authority for the SPARKID user.

Although we did not test this, it should be fairly straightforward to convert this to a Started Task or Started Job JCL, in order to be able to run the Spark processes as a started task.

5.4 System level tuning

System level tuning involves three areas:

- Tuning the Mainframe Data Services for Apache Spark for z/OS (MDSS) server
- Tuning z/OS UNIX settings
- Tuning the Spark master and worker settings

5.4.1 Tuning the MDSS server

The product documentation (*IBM z/OS Platform for Apache Spark Installation and Customization Guide*, SC27-8449 and *IBM z/OS Platform for Apache Spark Administrator's Guide*, SC27-8451) contains extensive details about how to configure MDSS for optimum performance. Specifically, it discusses how to set up z/OS Workload Manager (WLM) and how to configure workload classification rules.

You might also want to run multiple instances of MDSS, for load balancing and high availability (HA) purposes. In an IBM Parallel Sysplex environment, this would enable you to spread a request to multiple z/OS images, and take advantage of other Sysplex users, for example, DB2 in data sharing mode.

For high availability, you can form a group of multiple MDSS servers, and you can configure the client application with the port numbers of more than one member of a group. If a particular group member is not available, the MDSS driver transparently selects another member of the group.

5.4.2 Tuning z/OS UNIX settings

When sending Resilient Distributed Datasets (RDDs) to disk, Spark may have to open a large number of files concurrently. If you receive the following error message, you should increase the maximum number of file descriptors for the Spark user:

```
org.apache.spark.SparkException: Job aborted due to stage failure: Task 8 in stage
0.0 failed 1 times, most recent failure: Lost task 8.0 in stage 0.0 (TID 8,
localhost): java.io.IOException: EDC5124I Too many open files.
```

The maximum number setting can be changed on a per-user level in the user's OMVS segment (FILEPROCMAX), or on a system level by changing MAXFILEPROC in parmlib member BPXPRMxx. Of course, the first option is usually the preferable one.

You can verify the setting with the **`ulimit -a`** or the **`getconf -a`** commands.

6

IBM z/OS Platform for Apache Spark and the ecosystem

This chapter describes the ecosystem of users, components, and tools that are often used as a front-end to the Apache Spark server environment. Strictly speaking, the whole ecosystem outlined in this chapter is optional. It is quite possible to install and configure an Apache Spark server on an IBM z/OS platform with all of its associated data sources, write Scala or Python applications using a text editor to perform the required analysis, and output the results in text form. In this case, the only interface needed to the Apache Spark server is a Secure Shell (SSH) session into the z/OS UNIX shell.

This ecosystem is intended for the audience of users who have no interest in the platform supporting the Spark server, and have little or no experience with IBM z Systems. The only thing that matters to this population is having efficient access to a huge data set, and the infrastructure needed to perform an analysis of that data in a reasonable amount of time. These are the data scientists who are charged with delivering insights and business value to their customers.

Although data scientists create the ultimate deliverables from a set of big data, they usually collaborate with others who perform several important tasks to acquire the data and facilitate access to it. We describe the roles of others who work with the data scientist here, but this should not be taken as a prescriptive outline of responsibilities:

The Data Engineer The closest person to the z/OS platform, the data engineer assembles the raw data from different sources and builds a unified virtual view for processing on the Apache Spark cluster. This person collaborates with the information management specialist and data scientist to create a useful data scheme for the virtual view of the data.

The Information Management Specialist

Collaborates with the data engineer to generate applications that “wrangle” the data. This may involve cleaning the data (removing outliers, filling in missing data, normalizing if needed, and so on). Works with the data scientist to create applications that interact with the Apache Spark cluster on z/OS. This person might also be considered an application developer.

The Data Scientist Develops the model to explore the data and create insights that have business value. Works with the information management specialist and data engineer to create the schema for the processed data that the model will act upon.

They might also perform some application development to implement the model that generates deliverable insights. The results of the analysis can optionally be stored in the tidy data repository.

Figure 6-1 shows several possible client instances.

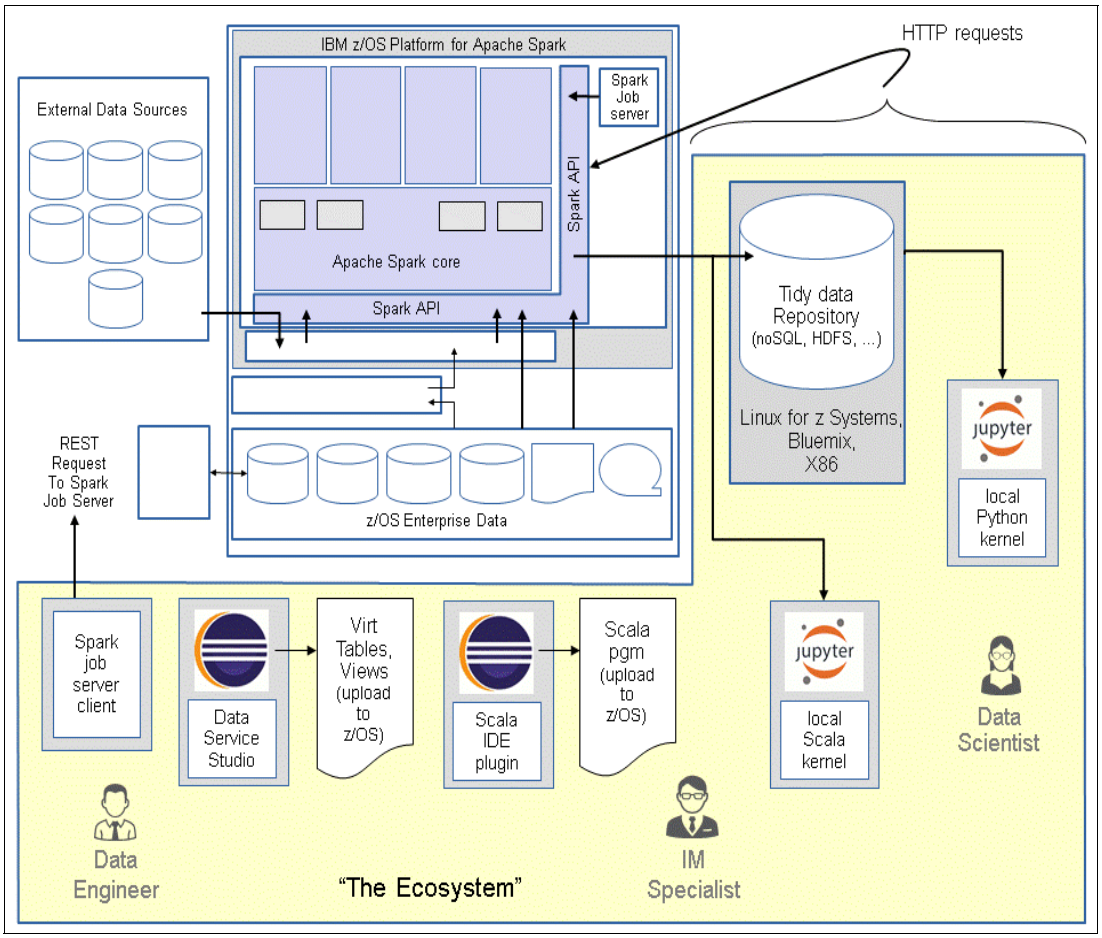


Figure 6-1 The IBM for z/OS Apache Spark ecosystem

The client portion of this environment is complex, and is tailored to the specific needs of the individual user. In Figure 6-1, there are several client instances, each tailored to a specific task:

- The Data Service Studio is an Eclipse-based interface to define virtual views of data present on the host for the Spark cluster.
- The Scala integrated development environment (IDE) plug-in is an Eclipse package that can be used to develop Scala applications.
- The Jupyter notebook is a web-based interface to a notebook driver that communicates with the Spark cluster, and allows the data scientist to interactively develop applications.

Unlike conventional client/server architectures, the user generally assembles their own client interface from a collection of components. These components are largely open source, with certain packages provided by IBM for libraries and interfaces to enable access to the IBM z/OS platform for Apache Spark.

While the Data Server Studio (DSS) and Scala plug-ins are conventional Eclipse environments, there are several different ways that the Jupyter notebook driver can be configured. In the following sections in this chapter we discuss some of these configurations, and the advantages and disadvantages of each.

6.1 Tidy data repository

The meaning of *tidy data repository* that we use when describing the ecosystem of the IBM z/OS Platform for Apache Spark differs somewhat from the conventional definition of the term.[1] In many environments, the tidy data repository is the location where the data engineer and the information management specialist assemble the cleaned data that is appropriately conditioned for the data scientist to act upon.

The data might have been transformed in some simple way to improve the fidelity of the analysis that the data scientist intends to perform. In this case, the tidy data repository holds this clean data.

One of the key values of the IBM z/OS Platform for Apache Spark is that compute resources are brought to the data. There is no need for the expensive and tedious extract, transform, and load (ETL) process. The data cleaning process that commonly results in tidy data often implies a copy to the repository, which is effectively the same thing as an ETL.

Because data movement on a large scale is something to avoid, there are two options:

- Update the data at its original sources
- Perform data cleaning programmatically

Often, original source updates are impossible, impractical, or not allowed. For example, accessing a stream of Twitter feeds through a view of the data presented by MDSS means that there is no way to update the source information to fill in some incomplete data with a default value. It is possible to create a cached version of the Twitter data on z/OS for cleaning, but this would require more infrastructure to manage the copy of the data.

An alternative to altering data at the source during cleaning is to do it programmatically. In this case, the automated cleaning process could be thought of as simply an early stage of the data analysis pipeline. There are scenarios, though, where the cleaning process requires actions that are too complex to manage programmatically, so some host-based caching of the information is needed.

With MDSS providing a single unified view of the data, and a companion cleaning strategy that is compatible with keeping the original source data in place, the tidy database repository can be used to locate the results of the processing performed on the Spark cluster. The advantage here is that the results generated during transformation and analysis of the data are orders of magnitude smaller than the original source data. This makes storing in the tidy data repository practical.

[1] See https://www.ibm.com/developerworks/community/blogs/jfp/entry/Tidy_Data_In_Python?lang=en for more about tidy data.

The choice of database to use for the repository, and the platform for the deployment, depends on the characteristics of the particular environment and the needs of the users. There are really no wrong choices when configuring this repository, but there are a few rules worth noting:

- NoSQL databases are popular and useful for this purpose. Examples of these databases include IBM Cloudant®, MongoDB, Apache Cassandra, and Redis.
- NoSQL databases often require locating the repository on a platform other than z/OS. Linux on z Systems is a good choice that keeps the data close to the Spark cluster, but this isn't a requirement. Even a cloud-based repository is a viable option, if it is secure and accessible to both the Spark cluster and any remote servers that the data scientist and other consumers of the results might use.

One other consideration for the tidy data repository is the frequency with which the data is updated. There should be a ready audience for the insights generated by the data scientist, and those insights usually have a very short shelf life. The IBM z/OS Platform for Apache Spark is meant to enable big data analysis in real, or near-real, time.

If the results written to the tidy data repository are not updated regularly and frequently, consumers of the analysis will notice. How often the analytical results are made available to users is determined entirely by the needs of the particular enterprise. However, it is probably accurate that updates once a day by a batch job that runs overnight won't be often enough.

6.2 Jupyter notebooks

The configurations outlined in the following sections make up a complete Jupyter notebook client environment. This environment can be hosted on Linux, running on an X86 laptop/workstation, or in a z Systems Linux partition.

While the following descriptions outline a few different configurations, this is not meant to be a complete list. It is possible to build a Jupyter notebook driver on other platforms, such as Microsoft Windows or Apple MacOS. However, there are runtime considerations for those environments that are beyond the scope of this chapter.

All configurations here use Docker to contain and deploy a Jupyter notebook driver called the Scala workbench. This driver is located on a host machine that communicates directly with the Spark cluster hosted on z/OS. Users interact with the Spark cluster through a web interface to the workbench.

6.2.1 The Jupyter notebook overview

The official Jupiter notebook infrastructure is kept in the `zos-spark/scala-workbench` github repository:

https://github.com/zos-spark/scala-workbench

This repository is updated with the latest code and instructions on a regular basis, so the information there will always be the most current.

Figure 6-2 on page 95 provides an overview of the environment from a build and deployment standpoint.

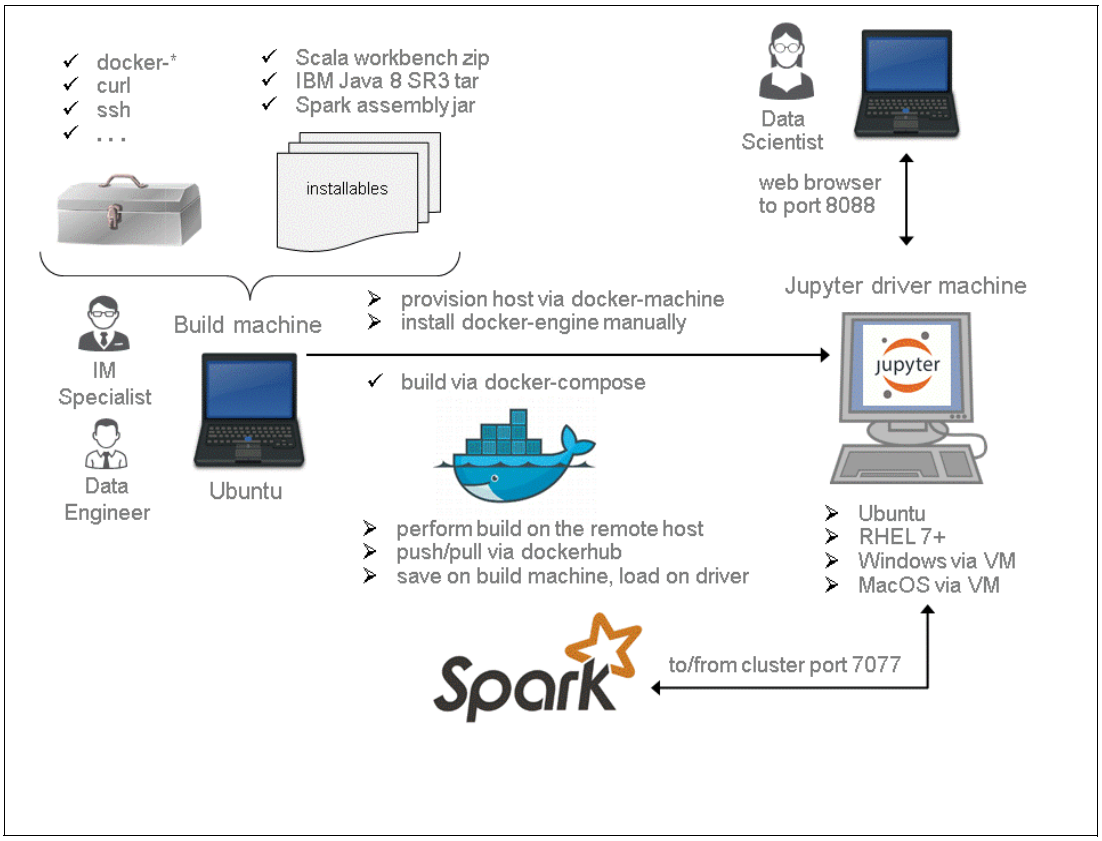


Figure 6-2 Jupyter notebook overview

6.2.2 Docker and the platforms that support it

It's a good idea to be familiar with Docker containers before proceeding.[2] There is more to it than just the runtime environment provided by docker-engine that often first comes to mind. It also includes knowledge of docker-compose, which is used to build and run containers on the host driver machine, and possibly docker-machine, depending on how you want to provision Docker host environments. Knowledge of DockerHub[3] can also be useful.

Docker was first created on the Ubuntu distribution of Linux, and Ubuntu remains the platform of choice for many Docker users. Docker is also officially supported on Red Hat Enterprise Linux (RHEL) starting with release 7. Docker requires a Linux kernel, which all Linux distributions provide, but the associated infrastructure of the platform (primarily the way software packages are managed), is what makes the environment work better on Ubuntu than other distributions.

In addition, applications packaged for Docker often contain Ubuntu-specific dependencies of various kinds. The key point is that the portability of a "dockerized" application depends on the application as much as it does on Docker.

Non-Linux platforms, such as MacOS and Windows, do not currently support Docker. There is work ongoing to change this, but in the meantime, Docker has facilities to generate container hosting capabilities in a VirtualBox image. Through docker-machine, it's possible to provision a Docker host in a VirtualBox virtual machine (VM) that can be run using the VirtualBox hypervisor on Windows or Mac machines.

[2] See `https://docs.docker.com` for an overview of all the relevant docker products.
[3] See `https://hub.docker.com` for more information on DockerHub.

Although Docker environments can be hosted on Windows of Mac, the VirtualBox approach is not covered here. Network configuration and bidirectional communication requirements with the Spark Master make this topic difficult to cover in detail. The focus of this chapter is building and deploying a Jupyter driver in a Docker container on a Linux machine.

6.2.3 The dockeradmin userid

One of the things not mentioned very prominently in the documentation for Docker is the use of the dockeradmin id. It's very easy to overlook this, and then get errors during installation or at run time because the user performing a Docker operation is not authorized. This needs to be done on any machine where Docker containers are hosted (any place where docker-engine runs).

To ensure proper authorization, complete the following steps:

1. First, create a dockeradmin userid:

   ```
   > sudo adduser dockeradmin
   > sudo passwd dockeradmin
   ```

2. Edit `/etc/sudoers` and add the following section (near the entry for the root user is a good place):

   ```
   > sudo visudo
   ```

3. Add the following line:

   ```
   dockeradmin ALL=(ALL) NOPASSWD: ALL
   ```

Be sure to give dockeradmin a good password. When the `/etc/sudoers` changes have been made, this user will have the same authority as the root user.

6.2.4 The Role of SSH

Docker hosting environments managed through docker-machine require the use Secure Shell (SSH). SSH uses public/private key pairs to provide secure communication and authentication between one machine and another. Even if you don't use docker-machine, setting up and using a key pair and using SSH is a good idea.

Here are the steps to manage keys between the build machine and the Jupyter driver machine in the overview diagram shown in Figure 6-2 on page 95.[4] Run the following commands from the build machine:

```
$ mkdir ~/.ssh
$ chmod 700 ~/.ssh
$ ssh-keygen -t rsa
```

Make sure not to use any password for the generated public key. If you do, then every time you try to use SSH with the public key generated here, the system will prompt for that password. This hangs Docker remote operations, because there is no means to enter a password during the host provisioning process performed by docker-machine.

```
$ ssh-copy-id dockeradmin@<remote-host>
```

The *<remote-host>* here should be replaced with the Internet Protocol (IP) address or name of the target machine that will host Docker containers.

[4] See https://help.ubuntu.com/community/SSH/OpenSSH/Keys for more information about SSH and key pairs.

6.2.5 Creating the Docker container

A primary difference in the configurations outlined here is how the containers are provisioned and deployed. Those familiar with managing a Docker environment using docker-machine to provision remote hosts and build containers will likely prefer to approach things differently than those with little Docker experience.

In the overview diagram shown in Figure 6-2 on page 95, the Jupyter Driver image is built on an Ubuntu platform. This can be done either locally (the resulting driver container will be hosted on the build machine itself) or remotely (using docker-machine).

Docker-machine managed environments

If you intend to use the Ubuntu build machine to manage one or more Docker hosts through docker-machine, then it's probably a good idea to use Ubuntu on all of the target machines as well. We have not successfully provisioned a Docker host on an RHEL 7+ machine using docker-machine. If you intend to run your Jupyter driver on an RHEL 7+ machine, please see the next section, "Standalone docker-engine environments".

From the build machine, run docker-machine to provision the Docker host on the target machine:

```
> docker-machine create --driver generic \
--generic-ip-address <remote_ip_address> \
--generic-ssh-key <path-to-local-ssh-private_key> \
--generic-ssh-user dockeradmin \
<jupyter-driver-machine-name>
```

This will provision a docker host environment on the target remote machine, which means that docker-engine will be installed. After it is installed, the docker engine daemon will be started, and await requests. This is why the dockeradmin userid needs to have root authority.

Standalone docker-engine environments

Standalone environments require you to install docker-engine manually on the target machine that will be hosting Docker containers. Docker-engine installs cleanly and easily in both Ubuntu and RHEL 7+ environments. As with the docker-machine remote install described previously, use the dockeradmin id that you created on the hosting machine to perform the installation and configuration.

For more details about the installation process, see the following website:

https://docs.docker.com/engine/installation/linux/

After docker-engine has been installed and the Docker daemon is running, the target machine is in a state equivalent to one provisioned remotely with docker-machine.

6.2.6 A note about network configuration

Note the requirement listed on https://github.com/zos-spark/scala-workbench indicating that the Jupyter driver must be network-addressable to all nodes in the Spark cluster. This statement necessitates the following requirements:

- The physical driver machine must be in the same network-addressable environment as the Spark cluster on z/OS.
- The driver machine needs bidirectional communication capabilities to the Spark cluster. It is not enough that the driver machine can ping the IP address of the Spark master node. The cluster needs to be able to pass data back to the driver, and then on to the web browser of the Data Scientist. Therefore, the driver has to have an externally addressable IP address. For virtual-machine-based drivers, the default addresses associated with the VM are private (for example, `10.*`, `172.*`, and `192.*`) and visible only to network entities that are local to the VM.

  The VM usually communicates externally through network address translation (NAT) to the hypervisor. This does not provide bidirectional communication with an external network endpoint, such as a Spark cluster. Setting the Jupyter driver up for a Spark cluster using NAT will result in the driver sitting quietly and appearing to hang. This is why running the Jupyter driver from a VM is a more complex environment, and is not covered here.

6.2.7 Building the Jupyter scala workbench

The Docker container built for this configuration encapsulates all of the resources needed for the Jupyter notebook to work with the Apache Spark server on z/OS. These resources include all of the libraries and interfaces that are unique to the z/OS environment. The steps outlined in the following sections all take place on the driver build machine.

Build machine prerequisites

The following software must be on the build machine:

- Ubuntu (any recent version):

  https://docs.docker.com/v1.7/installation/ubuntulinux/
- The following tools and packages (these might already be present):
  - docker-engine 1.10.0 or later
  - docker machine 0.6.0 or later (if doing remote provisioning)
  - docker-compose 1.6.0 or later
  - cURL
  - SSH

Installable parts

The z/OS-Spark github repository for the scala workbench lists the required parts, and the locations where they can be found. Although the names and versions listed in this repository will change over time, the following key parts are needed to build the Jupyter driver:

- The scala workbench master `.zip` file. This is the workbench code.
- IBM Java V8R3.
- The Spark z/OS `.jar` file. Parts from this are used to facilitate communication between the driver and the Spark cluster.

Extract the scala workbench master `.zip` file. This creates a directory where the build will take place. Copy or move the other installable parts into the scala workbench directory to prepare for the build process.

Prepare Docker for the build

To prepare Docker for the build, complete the following steps:

1. Make sure that Docker is not only installed, but that the Docker daemon is running. You can verify this with the following command:

   ```
   > sudo service --status-all | grep docker
   ```

2. If there is no output showing that Docker is running, start it:

   ```
   > sudo service docker start
   ```

   This will start the Docker daemon now, and at boot time from now on. If you are not using docker-machine to perform a remote provision and build of the Jupyter driver, you can proceed to the next section to start the build.

3. If you are using docker-machine to manage remote Docker hosts, make sure that the remote system you want to build on is the current one:

   ```
   > docker-machine ls

   NAME              ACTIVE   DRIVER       STATE     URL ...
   softlayer         *        generic      Running   tcp://10.0.0.10:2376 ...
   ```

   If there is an asterisk in the Active column, then the associated Docker host is where Docker commands will be run. In this case, the build process, which uses docker-compose, will run on this host.

4. There may be multiple hosts in this list. If you want to target a different host, perform the following command:

   ```
   > eval "$(docker-machine env <docker-host-name>)"
   ```

Set up the configuration

Now it is time to set the configuration parameters for communicating with the Spark cluster on z/OS:

1. Make the Scala work master directory that was created when the workbench was extracted your current directory. The contents of this directory should look similar to the following file listing:

   ```
   > ls
   build.log                        installer.properties
   build.sh                         LICENSE
   config                           README.md
   demos                            spark-assembly-1.5.2-hadoop2.6.0.jar
   docker-compose.yml               start.sh
   Dockerfile                       stop.sh
   ibm-java-x86_64-sdk-8.0-3.0.bin template
   ```

   Note that this directory should include the IBM Java binary and the Spark assembly Java archive (JAR) installable files that were downloaded earlier.

2. Edit the config file:

   ```
   #!/bin/bash
   SPARK_HOST="JUPYTER.DRIVER.YOUR.HOST"  # Spark Master hostname/ip
   SPARK_PORT="7077"                 # default spark master port
   WORKBOOK_NAME="workbook"          # default workbook name
   WORKBOOK_PORT="8888"              # default workbook port
   ```

3. The only line that needs to be changed is the one for the Spark master host. Put either the host name or IP address in here. The ports and workbook settings can remain with their default values.

Build the driver

Perform the following command to initiate the build:

```
> sh build.sh
```

This script invokes docker-compose, which builds the Jupyter driver on the active docker host. If you are using docker-machine, this host is the remote machine that you have designated previously. If you are not using docker-machine, the build takes place on the local machine.

The build process generates a lot of output, and takes about half an hour. It results in a Docker container, or image, which can be displayed with the **`docker images`** command:

```
> docker images
REPOSITORY                 TAG                 IMAGE ID            CREATED             SIZE
zos-spark/scala-notebook   latest              eeedfed19592        48 minutes ago      5.254
hello-world                latest              94df4f0ce8a4         8 weeks ago      967
jupyter/minimal-notebook   2d878db5cbff        c8e205a72f2c         9 weeks ago        1.84
```

The name of the image by default will be `zos-spark/scala-notebook`.

Move the container to the target machine (as needed)

If you intend to run the Jupyter driver Docker container on a different target machine, and you are using a standalone docker-engine configuration, you need to move the container from this local build machine to the target. Note that this is the only option available for environments where the target machine is an RHEL 7+ platform.

To move the container, complete the following steps:

1. Perform the following command on the local build machine to save the container to a `.tar` file:

   ```
   > docker save -o <driver_tar_file.tar> <repository-name>
   ```

   This will take some time, because the Docker container with the driver in it is over 5 gigabytes (GB) in size.

2. After the save completes, compress the `.tar` file, and transfer it to the target machine. Extract it on the target machine, and load it into Docker:

   ```
   > docker load -i <path to image tar file>
   ```

Alternatively, you can set up a repository on Dockerhub (`https://hub.docker.com/`) and perform a commit/push/pull to get the image to the target machine. If you have a Dockerhub ID and are familiar with this interface, this is probably the preferred way to transport the container.

Prepare the target machine (as needed)

As with the previous section, there is some additional setup required on the target machine for standalone docker-engine environments.

The Jupyter notebook driver is started using the **`start.sh`** script, which does some setup before calling docker-compose. For this reason, you need to install docker-compose on the target machine where the driver will be running.

You will also need the start and stop scripts, as well as the config file that you used when you built the driver container. The easiest way to accomplish this is to transfer the scala workbench master `.zip` file to the target machine and extract it there. Make sure that the config you have on the target machine matches the one from the build machine.

Run the Jupyter driver

To run the Jupyter driver, complete the following steps:

1. From the scala workbench master directory, run the following command:

   ```
   > sh start.sh
   ```

   This will read your config file, and invoke docker-compose to start the docker container with the Jupyter driver in it. When this completes, you can point a web browser to it at the port identified by WORKBOOK_PORT (defaults to 8888) in your config file. You should see something like the tab shown in Figure 6-3.

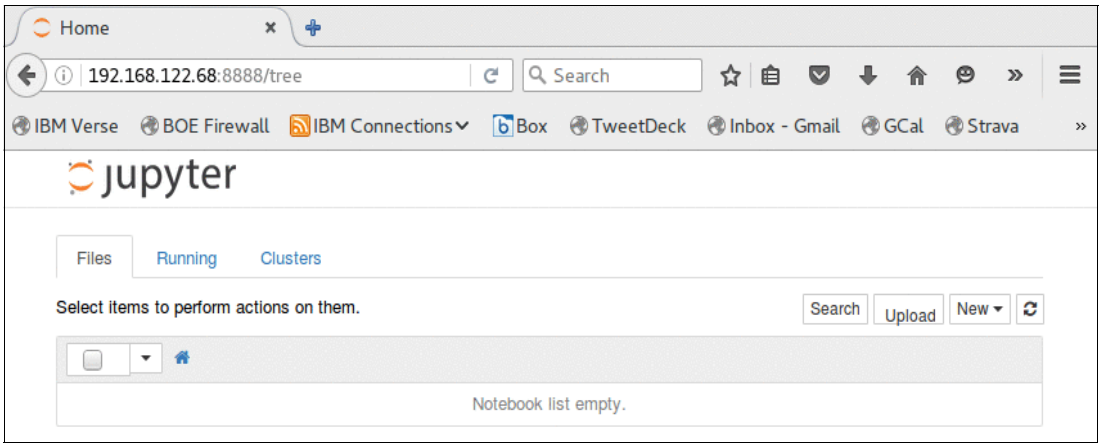


Figure 6-3 Jupyter start screen

   The Jupyter driver is now ready. If you have the network configured to allow bidirectional communication between the driver and the cluster, the user should be able to use their Jupyter notebook to drive work from the web interface through to the Spark cluster on z/OS.

2. When it's time to shut the Jupyter driver down, simply run the stop script:

   ```
   > sh stop.sh
   ```

7

Use case patterns

This chapter contains some scenarios that may be used to model your own solution to meet your specific business needs.

If the majority of your data used for analytics is on IBM z/OS, you should seriously consider using IBM z/OS Platform for Apache Spark.

Many of the use cases discussed in this chapter are intended for illustration purposes only. You may be able to design your own implementation by mixing and matching the example use cases described in this chapter.

The following use case patterns are covered in this chapter:

- Churn prediction
- Fraud prevention
- Upsell opportunity detection
- Payment analytics
- Product recommendations

Specifically, this chapter describes the following topics:

- 7.1, “Banking and finance” on page 104
- 7.2, “Insurance industry” on page 112
- 7.3, “Retail industry” on page 113
- 7.4, “Other use case patterns for IBM z/OS Platform for Apache Spark” on page 114
- 7.5, “Operations analysis” on page 114

7.1 Banking and finance

The technical information in the following banking example may be extrapolated and applied to other industries as applicable.

7.1.1 Churn prediction

In this example, we train a model to determine the probability that each client will leave (also called a *churn rate*). The predicted churn probability value can be used to project the business loss for each client using the current revenue from that particular client through extrapolation. Because all of the required information is usually available in the z/OS platform, you can accomplish this using IBM z/OS Platform for Apache Spark without moving this sensitive data to a different platform.

Data sources

Data sources include IBM Virtual Storage Access Method (VSAM) and IBM DB2 for z/OS.

Sample implementation

Churn prediction using the abundant customer data on the z/OS platform can be accomplished on-platform with the help of Spark on z/OS. Example 7-1 on page 105 contains sample code that you can use to model your Scala application on Spark.

Spark ML adopts the DataFrame from Spark SQL in order to support various data types under a *unified dataset* concept. DataFrame supports many basic and structured types. See the Spark SQL datatype reference for a list of supported types. In addition to the types listed in the Spark SQL guide, DataFrame can use ML Vector types.

Transformers

A Transformer is an abstraction that includes feature transformers and learned models. Technically, a Transformer implements a **`transform()`** method that converts one DataFrame into another, generally by appending one or more columns. In our example, the learning model takes a DataFrame as input, reads the column containing feature vectors, predicts the label for each feature vector, appends the labels as a new column, and outputs the updated data set.

Estimators

An Estimator abstracts the concept of a learning algorithm that fits or trains on data. Technically, an Estimator implements a **`fit()`** method that accepts a DataFrame and produces a Transformer. For example, a learning algorithm such as LogisticRegression is an Estimator, and calling **`fit()`** trains a LogisticRegressionModel, which is a Transformer.

Parameters

Spark ML Estimators and Transformers use a uniform application programming interface (API) for specifying parameters. A Param is a named parameter with self-contained documentation. A ParamMap is a set of (parameter, value) pairs.

There are two main ways to pass parameters to an algorithm:

- Set parameters for an instance. For instance, if lr is an instance of LogisticRegression, one could call **`lr.setMaxIter(10)`** to make **`lr.fit()`** use at most 10 iterations.
- Pass a ParamMap to **`fit()`** or **`transform()`**. Any parameters in the ParamMap will override parameters previously specified using setter methods.

Example 7-1 Scala code for predicting churn with z/OS data using Spark MLlib

```
object RegressionDemo {

  case class ContractFeatures(contID: Int, features: Vector)

  def trainModel(trainingDF: DataFrame) = {
    val lr = new LogisticRegression()
      .setRegParam(0.1)
      .setMaxIter(30)
      .setThreshold(0.55)
      .setProbabilityCol("churnProbability")

    lr.fit(trainingDF)
  }

  def main(args: Array[String]) {

    val conf = new SparkConf()
      .setAppName("Regression demo")
      .set("spark.app.id", "Regression demo")
    val sc = new SparkContext(conf)
    val sqlContext = new SQLContext(sc)
    import sqlContext.implicits._

    val Array(mdssUrl, user, password) = args
    val props = new Properties
    props.setProperty("user", user)
    props.setProperty("password", password)

    val vDF = sqlContext.read.jdbc(mdssUrl, "CLIENT_PAY_SUMM", props)
      .select("CONT_ID", "CHURN", "AGE_YEARS", "ACTIVITY_LEVEL")
      .filter($"AGE_YEARS" < 50)

    val Array(trainingData, testData) = vDF.randomSplit(Array(0.7, 0.3))

    def toVector(row: Row) =
      Vectors.dense(row.getDecimal(2).doubleValue, row.getInt(3).doubleValue)

    val trainingDF = trainingData.map { row =>
      LabeledPoint(row.getInt(1), toVector(row))
    }.toDF.cache()

    val testDF = testData.map { row =>
      ContractFeatures(row.getDecimal(0).intValue(), toVector(row))
    }.toDF.cache()

    val model = trainModel(trainingDF)
    model.transform(testDF)
      .select("contID", "churnProbability", "prediction")
      .show(100)
  }
}
```

We use LabeledPoint, which is a case class. Spark SQL can convert Resilient Distributed Datasets (RDDs) of case classes into DataFrames, where it uses the case class metadata to infer the schema.

The new `LogisticRegression()` instance created in our example is an Estimator.

You may make predictions on test data using the **`Transformer.transform()`** method. `LogisticRegression.transform` only uses the `features` column.

Note that in our example, the output `ChurnProbability` column instead of the usual `probability` is used as a column, because we renamed the probability column parameter in this sample using **`setProbabilityCol("churnProbability")`**.

In Figure 7-1, a single data set was split into training and test data sets. The training data is to train the model, and the test data is to test the accuracy of the prediction. If the predictions are satisfactory, you should be able to do projections about lost revenue to the bank.

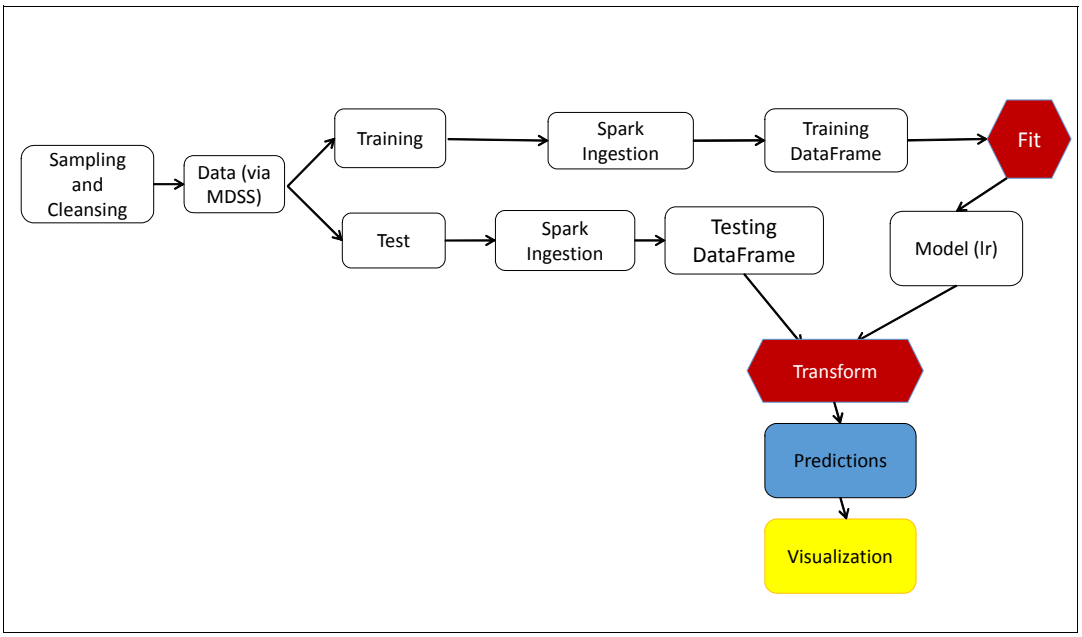


Figure 7-1 Churn prediction

7.1.2 Fraud prevention

The business goals of this use case are to reduce loss due to card fraud, reduce card deactivation, grow revenue associated with card purchases, reduce call center costs, and improve service yielding preferred card usage. Figure 7-2 on page 107 illustrates IBM's Fraud management point of view.

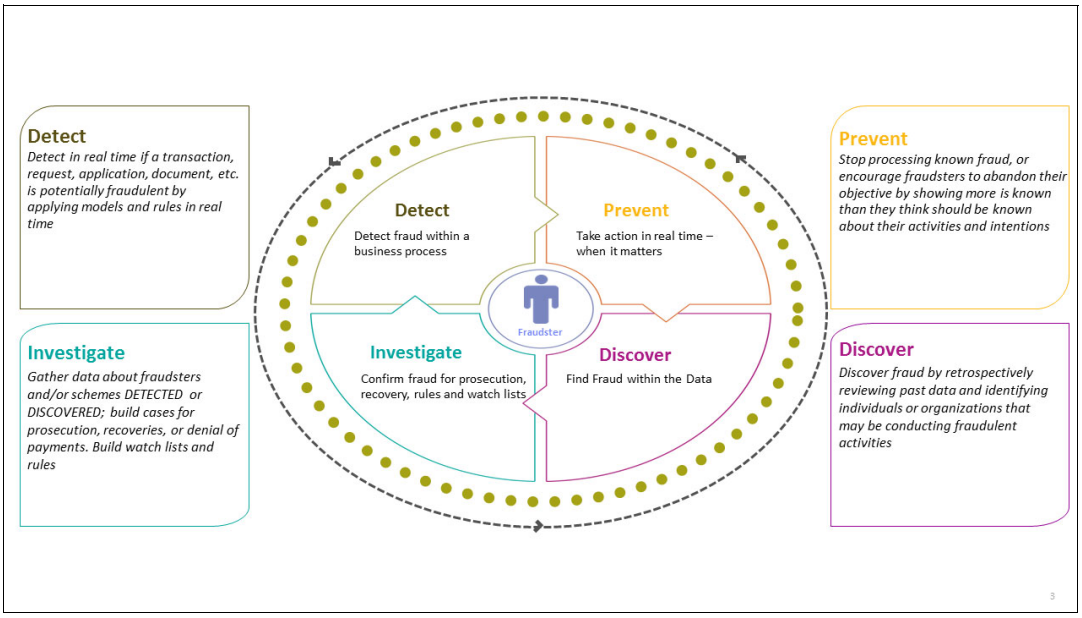


Figure 7-2 Fraud management point of view

The approach is to use Spark on z/OS to incorporate, integrate, and aggregate data from geographic location, merchant, issuer, and card history into the existing card authorization business flow to reduce fraud while preserving transactional service level agreements (SLAs).

Because all of your analytical processing can now happen on the IBM z Systems platform (where your transactional data is) this approach would completely eliminate the time and cost associated with extract, transform, and load (ETL) to a different platform. In addition, this approach drastically reduces the probability of data breaches, because your operational data never leaves the system of record.

Data Sources include VSAM, IBM Information Management System (IBM IMS), and DB2 for z/OS with or without Analytics Accelerator and optional social media data (in `.csv` format), which can be easily brought into IBM z/OS Platform for Apache Spark as DataFrames.

A business-view diagram for this use case is shown in Figure 7-3.

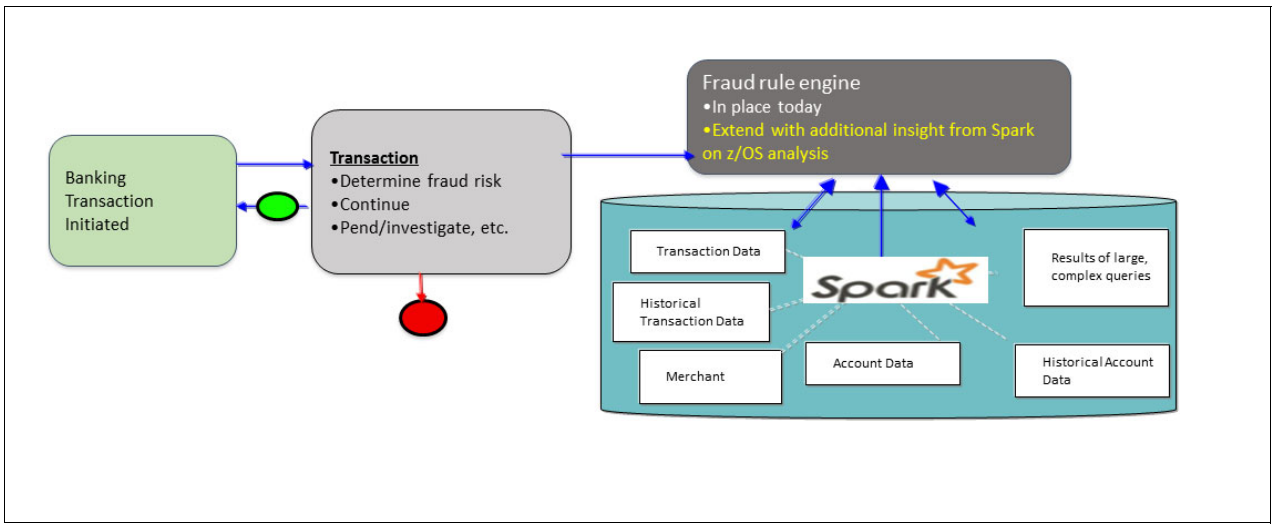


Figure 7-3 Financial crimes and Fraud prevention

With this approach, you may enable data aggregation on operational data to happen several times a day, and use this aggregated data as part of your real-time fraud detection process. Aggregation can be achieved using high performance views (either using virtual views with the mainframe data service for IBM z/OS Platform for Apache Spark, or IBM Analytics Accelerator for z/OS, or a combination of both) before the data is brought into Spark on z/OS.

The following four subsystems need to work hand in hand to detect anomalies in streams of events:

- Online transaction processing (OLTP) system leveraging the existing z/OS transactional environments (IMS, DB2, and IBM Customer Information Control System (IBM CICS))

  The goal of this system is to receive events and reply as fast as possible. Architectural design focuses on achieving very high throughput with very low latency, using well optimized z/OS transactional systems with the help of the Predictive Model Markup Language (PMML) models generated from Spark on z/OS (see the sample code in Example 7-2) to perform in-transaction scoring.

- Apache Spark Stream processing system on z/OS

  This system can process each event within a few minutes of its arrival. The goal of this system is to adjust parameters of the fraud-detection models in near real-time, using data aggregated across all user activity (for example, flagging merchants in geographic regions that are currently more suspicious). This would greatly reduce call center costs. Combining real-time transaction data streaming into Spark on z/OS via CICS with OLTP data and social media data will improve the agility.

- Optional near real-time processing system using IBM Analytics Accelerator for z/OS

  This system can also run with low latency (with the help of Change Data Capture technology) and at the same time focus on improving the predictive models themselves. This process includes training the models on newly aggregated data, exploring new features in the data, and developing new predictive models.

  Predictive scoring can be integrated with fraud detection transaction for even more preventive capabilities (for instance, to avoid credit card deactivation). The main advantage of this system is that you can accelerate data ingestion into Spark on z/OS without consuming much mainframe millions of instructions per second (MIPS).

- IBM z/OS Platform for Apache Spark offline processing system

  This system can run with “days” latency. This system should be focusing on improving the predictive models as frequently as possible. This process could include training the models on new transactional data, exploring new features in the data through aggregation of old and new data, and developing new predictive models to compare against the old ones (sample job control language (JCL) is provided in Example 7-2 on page 109 to schedule and automate this process).

  You may also perform batch scoring and take preventive actions based on your findings. It will also be necessary to include the human data analysts to explore the data using the IBM Cognos Business Intelligence (BI) tool. This system can also be designed to further improve service yielding credit card usage.

Example 7-2 contains sample code that you can use to model your Scala application on Apache Spark for a similar fraud prevention scenario.

Example 7-2 Scala code for credit card fraud prevention with z/OS data using Spark MLlib

```
import org.apache.spark.SparkConf

import org.apache.spark.SparkContext
import org.apache.spark.sql.SQLContext
import java.util.Properties

import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.linalg.{ Vector, Vectors }
import org.apache.spark.ml.param.ParamMap
import org.apache.spark.sql.Row
import org.apache.spark.sql.types.StructType
import org.apache.spark.sql.DataFrame
import org.apache.spark.mllib.classification.LogisticRegressionWithLBFGS
import org.apache.spark.rdd.RDD
import org.apache.spark.mllib.classification.LogisticRegressionModel

object RegressionFraud {

def trainModel(trainingDF: RDD[LabeledPoint]): LogisticRegressionModel =
    new LogisticRegressionWithLBFGS().setNumClasses(2).run(trainingDF)

def main(args: Array[String]) {

    val conf = new SparkConf()
      .setAppName("Regression Fraud demo")
      .set("spark.app.id", "Regression Fraud demo")
    val sc = new SparkContext(conf)
    val sqlContext = new SQLContext(sc)
    import sqlContext.implicits._

    val Array(url, user, password) = args
    val props = new Properties
    props.setProperty("user", user)
    props.setProperty("password", password)

    val vDF = sqlContext.read.jdbc("jdbc:db2://wtsc60.itso.ibm.com:38050/DB11",
"CARDUSR.SPPAYTB_FRD_V", props)

    val Array(trainingData, testData) = vDF.randomSplit(Array(0.7, 0.3))

    val trainingRDD = trainingData.map { row =>
      LabeledPoint(row.getInt(7),
                   Vectors.dense(row.getDecimal(1).doubleValue,
                                 row.getDecimal(2).doubleValue,
                                 row.getDecimal(3).doubleValue,
                                 row.getInt(4).doubleValue,
                                 row.getDecimal(5).doubleValue,
                                 row.getDecimal(6).doubleValue))
    }.cache()

    val model = trainModel(trainingRDD)
```

```
    model.toPMML("fraudPMML.xml")

    val testRDD = testData.map { row =>
      Vectors.dense(row.getDecimal(1).doubleValue,
                    row.getDecimal(2).doubleValue,
                    row.getDecimal(3).doubleValue,
                    row.getInt(4).doubleValue,
                    row.getDecimal(5).doubleValue,
                    row.getDecimal(6).doubleValue)
    }.toJavaRDD.cache()

    val out = model.predict(testRDD)

    println(out.take(20))
  }
}
```

The `fraudPMML.xml` file output is listed in the Example 7-3. This PMML output can be used in your OLTP applications for near real-time scoring, or you can also use it in batch for batch scoring.

Example 7-3 Sample PMML output

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<PMML xmlns="http://www.dmg.org/PMML-4_2">
    <Header description="logistic regression">
        <Application name="Apache Spark MLlib" version="1.5.2"/>
        <Timestamp>2016-05-09T03:17:36</Timestamp>
    </Header>
    <DataDictionary numberOfFields="7">
        <DataField name="field_0" optype="continuous" dataType="double"/>
        <DataField name="field_1" optype="continuous" dataType="double"/>
        <DataField name="field_2" optype="continuous" dataType="double"/>
        <DataField name="field_3" optype="continuous" dataType="double"/>
        <DataField name="field_4" optype="continuous" dataType="double"/>
        <DataField name="field_5" optype="continuous" dataType="double"/>
        <DataField name="target" optype="categorical" dataType="string"/>
    </DataDictionary>
    <RegressionModel modelName="logistic regression" functionName="classification"
normalizationMethod="logit">
        <MiningSchema>
            <MiningField name="field_0" usageType="active"/>
            <MiningField name="field_1" usageType="active"/>
            <MiningField name="field_2" usageType="active"/>
            <MiningField name="field_3" usageType="active"/>
            <MiningField name="field_4" usageType="active"/>
            <MiningField name="field_5" usageType="active"/>
            <MiningField name="target" usageType="target"/>
        </MiningSchema>
        <RegressionTable intercept="0.0" targetCategory="1">
            <NumericPredictor name="field_0" coefficient="-3.098264570214077E-5"/>
            <NumericPredictor name="field_1" coefficient="4.801587748962146E-4"/>
            <NumericPredictor name="field_2" coefficient="-141.39802990913407"/>
            <NumericPredictor name="field_3" coefficient="-0.44066065331726434"/>
            <NumericPredictor name="field_4" coefficient="-9.89298558449761E-4"/>
```

```
            <NumericPredictor name="field_5"
coefficient="-0.0044838606085833115"/>
        </RegressionTable>
        <RegressionTable intercept="-0.0" targetCategory="0"/>
    </RegressionModel>
</PMML>
```

You can also perform the training and testing of the predictive model with an easy-to-use Jupyter Notebooks interface.

7.1.3 Upsell opportunity detection

The business goals of this use case are banks that want to integrate information across all of their product lines in order to make real-time, targeted decisions.

Data sources include VSAM, IMS, and DB2 for z/OS with Analytics Accelerator.

Real-time decisions are necessary because banks can risk losing customer business or loyalty for other products. Therefore, there is a need to incorporate high value, predictive advanced analytics as part of transactional systems. For example, all of the data is aggregated to determine a score that can be used to reduce a bank's risk in approving a loan application.

Real-time predictive customer intelligence leads to smart business decisions at the point of impact. Benefits of this solution include building long-term customer relationships, driving one decision one interaction at a time, and maximizing customer life-time value.

Figure 7-4 shows a business-view diagram for this use case.

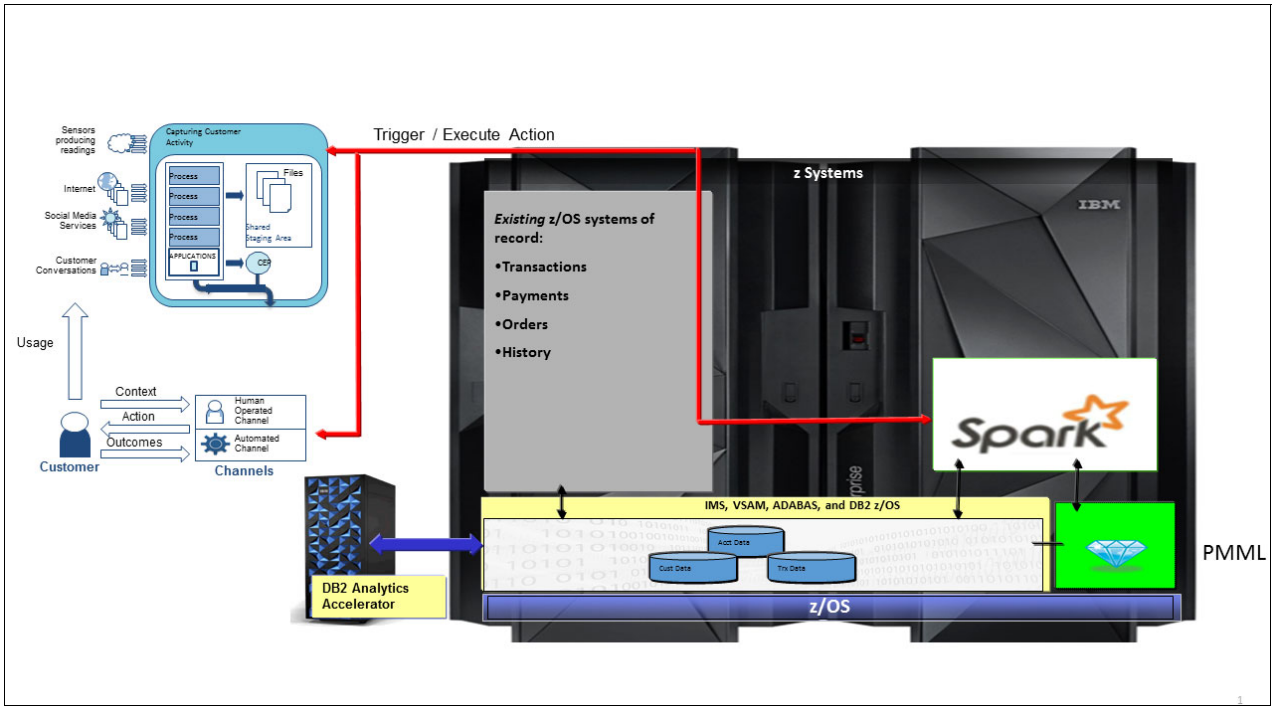


Figure 7-4 Predictive customer intelligence for upsell

7.2 Insurance industry

The following use cases may be implemented in z/OS with reference to a fictitious insurance company.

7.2.1 Claims payment analytics

The business goals for this use case are to quickly, efficiently tag each claim with additional business insight. By extending in-process claims scoring across domains, predictive analytics can be transformed from a specialized function to one that best leverages transactional systems at a repeatable enterprise scale.

One challenge that this use case addresses is that complex reports for overpaid claims are not completing on time, resulting in monetary losses.

Data sources include VSAM, IMS, and DB2 for z/OS.

The solution is to integrate optimized analytics of the IBM DB2 Analytics Accelerator for IBM z/OS (IDAA) with overpayment reporting of transactions. Benefits of this solution include huge improvements in speed of overpayment reports, line-of-business users are enabled to respond with more agility to overpayment trends, and informed decisions can be made at the right time.

Another challenge this use case addresses is to stop improper payments prior to actual payment, avoid pay and chase, and meet service level agreements (SLAs). The solution is to integrate predictive analytics into claims adjudication process.

Benefits of this solution include very efficient scale for analytics, scale requirements are only achievable with analytics as part of the transaction flow, and the expected results of efficient in-transaction analytics can be multi-million dollars per year. A business-view diagram for this use case is shown in Figure 7-5.

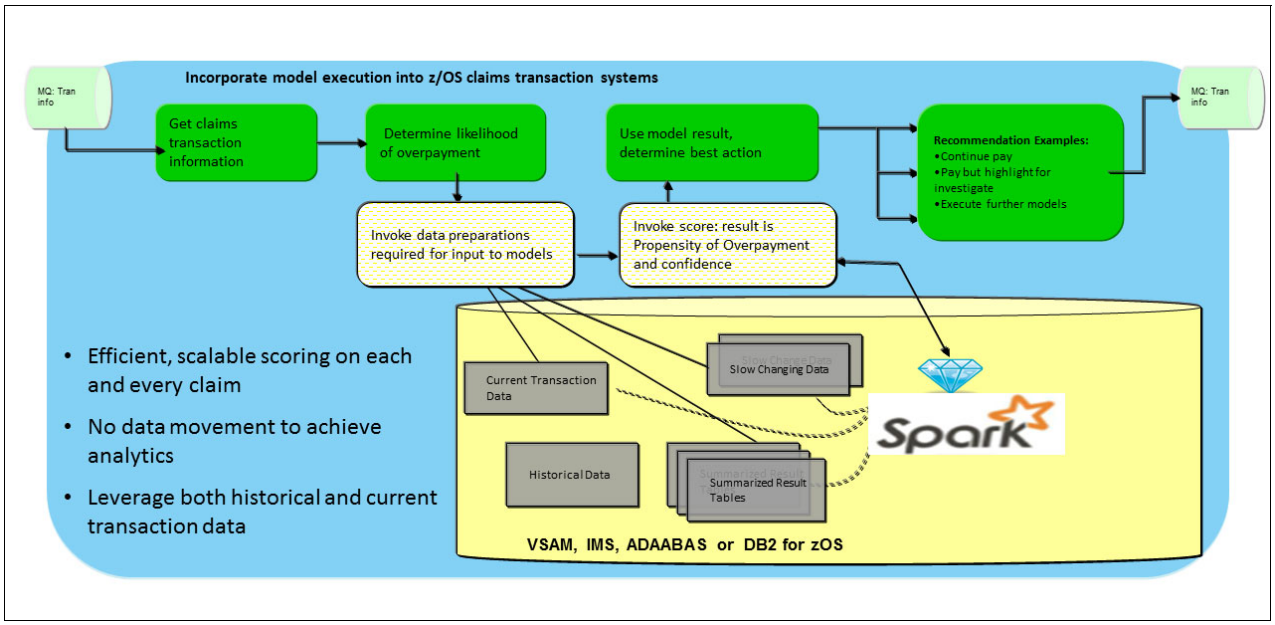


Figure 7-5 Claims processing payment analytics with Spark on z/OS

7.3 Retail industry

The retail industry can perform analytics on customer data.

7.3.1 Product recommendations

Spark enables you to compute product recommendation results much faster due to in-memory caching of live operational data on z Systems. Alternate Least Square algorithm can be used to recommend products.

Data sources include VSAM, IMS, and DB2 for z/OS.

Collaborative filtering is commonly used for recommender systems. These techniques aim to fill in the missing entries of a user-item association matrix, in our case, the user-product-rating matrix. Apache Spark MLlib supports model-based collaborative filtering, in which users and products are described by a small set of latent factors that can be used to predict missing entries. In particular, you may implement the alternating least squares (ALS) algorithm to learn these latent factors.

Figure 7-6 shows conceptually how product recommendations are made using the ALS algorithm. It uses the following logic to arrive at new product recommendations:

1. Start with random factors.
2. Hold the Products factor constant and find the best value for the Customers factor (Value that most closely approximates the original matrix).
3. Hold the Customers factor constant and find the best value for the Products factor.
4. Repeat steps 2-3 until convergence.

Figure 7-6 shows how product recommendations are made using the ALS algorithm.

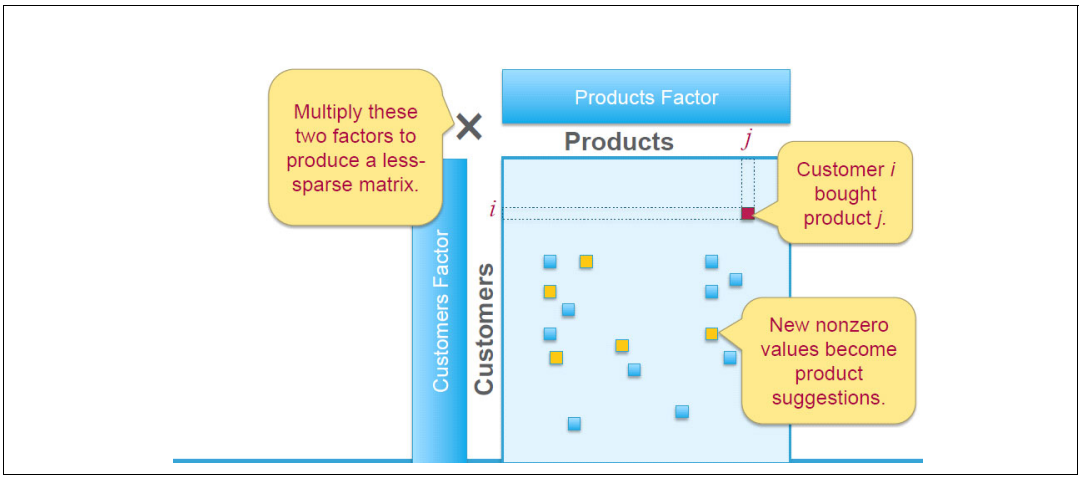


Figure 7-6 ALS algorithm for recommending new products to customers

You may also use SystemML (in SystemML's subset of R) to compile and run the ALS algorithm at scale.

Sample implementation code can be found on the following website:

http://spark.apache.org

7.4 Other use case patterns for IBM z/OS Platform for Apache Spark

The following additional patterns can be deployed programmatically by business processes, or enabled for data scientist use.

7.4.1 Analytics across OLTP and warehouse information

The data Sources on z/OS include DB2, VSAM, IMS, partitioned data set extended (PDSE), advanced database system (ADABAS), and so on.

Clients have OLTP on z/OS warehouses on distributed platforms.

Analytics across these environments can be challenging and inconsistent.

7.4.2 Analytics combining business-owned data and external / social data

Clients have OLTP on z/OS.

Clients have external (public) or social data on distributed servers.

External data delivers more value when combined with analytics from business data.

7.4.3 Analytics of real-time transactions through streaming, combining with OLTP and social

Analytics combines real-time transaction data streaming into Spark through CICS with high performance with OLTP data and social media data.

Using weather data

The data source is in JSON format.

7.5 Operations analysis

Traditionally, one of the most popular scenarios for customers to get started with Apache Spark is Operations Analysis. Operations Analysis focuses on analyzing machine data, which can include anything from information technology (IT) machines to sensors, meters, and global positioning system (GPS) devices.

It's growing at exponential rates and comes in large volumes and a variety of formats, including in-motion, or streaming, data. Leveraging machine data requires complex analysis and correlation across different types of data sets. By using big data for operations analysis, organizations can gain real-time visibility into operations, customer experience, transactions, and behavior.

Through Operations Analysis, organizations can accomplish the following goals:

- Consolidate log and metrics data from different sources and in different formats
- Reduce time required for root cause analysis by searching, filtering, and visualizing log and metrics information

- Improve overall service availability and maintainability for your infrastructure, applications, and networks
- Gain real-time visibility into operations, customer experience, and behavior
- Analyze massive volumes of machine data with sub-second latency to identify events of interest as they occur
- Apply predictive models and rules to identify potential anomalies or opportunities
- Optimize service levels in real-time by combining operational and enterprise data

7.5.1 SMF data

IBM System Management Facilities (SMF) is one of the key differentiators of z/OS in terms of system management. SMF gathers and records information about events and resource usage in your system. This information is saved in timestamped records and grouped by subsystem, providing you with unparalleled insight into the activities of your z/OS system.

SMF information is commonly used to generate reports for performance management, storage management, security violations, database performance, resource utilization, system event reports, and much more.

In section 4.8, “SMF data” on page 76, we demonstrate how to access SMF data from a Spark for z/OS application.

For more information about SMF, see the z/OS documentation and *SMF Logstream Mode: Optimizing the New Paradigm*, SG24-7919.

7.5.2 Syslog data

What is the first place that a z/OS system programmer or an operator looks into when a problem has been reported? Usually, it is the z/OS syslog. Therefore, many if not most z/OS installations have automation in place to monitor syslog messages and take corrective action whenever a problem has been detected.

From an operations insight perspective, it can therefore prove extremely valuable to analyze historic syslog data and correlate unusual events with other log data in your environment. For example, you might want to correlate security messages in the syslog to messages logged by an application running in IBM WebSphere Application Server, in order to identify potential security breaches or fraud attempts.

Again, we show an example of how to access the syslog in 4.7, “System log” on page 74.

See *Systems Programmer's Guide to: z/OS System Logger*, SG24-6898, for an in-depth discussion of the z/OS System Logger.

A

Sample code to run on Apache Spark cluster on z/OS

The following sample code can be used in your Spark on z/OS environment after modifying the following arguments:

- Table name
- URL
- Download `jcommander.jar`

Example A-1 shows the sample code.

Example A-1 Sample code to run on Apache Spark cluster on z/OS

```
import com.beust.jcommander.{ JCommander, Parameter }
import java.util.Calendar
import scala.math.random
import org.apache.spark.{ SparkConf, SparkContext }
import org.apache.spark.sql.SQLContext
import org.apache.spark.sql.functions._

// Generic Spark SQL test. Invoke it like this:
// ./bin/spark-submit --class "SparkTest" --master local[*] --jars
/rsusr/spark/spark151/targetQA/dv-jdbc-3.1.201603110348.jar
/rsusr/spark/spark151/targetQA/scala-2.10/scala-sample_2.10-1.0.jar --sql
"AZK.TABLENAME" --url
"jdbc:rs:dv://host:port;DBTY=DVS;PWD=xxx;UID=xxx;CompressionType=UNCOMPRESSED;MXBU
=4MB" --numPartitions 8 --partitionColumn t_id --lowerBound "200000000000000"
--upperBound "200000400000000"

// Note there's just a few args (--url, --sql) and we use the jcommander style
command line with named args.
// So rather than lots of individual command line args, it's mostly in the connect
string url and the sql.
// Note that the --sql param can be (usually is) just a table name, passed to
Spark as "dbtable" option.
```

```
object SparkTest {

  object Args {

    @Parameter(names = Array("--driver", "-d"), required = false, description =
"The driver class name.")
    var driverClassName = "com.rs.jdbc.dv.DvDriver"

    @Parameter(names = Array("--sql", "-s"), required = true, description = "The
SQL statement to execute.")
    var sql = ""

    @Parameter(names = Array("--url", "-u"), required = true, description = "The
connection string URL.")
    var url = ""

    @Parameter(names = Array("--numPartitions", "-n"), required = false,
description = "Number of Spark partitions.")
    var numPartitions = 0

    @Parameter(names = Array("--partitionColumn", "-p"), required = false,
description = "The table column to use for Spark partitioning.")
    var partitionColumn = ""

    @Parameter(names = Array("--lowerBound", "-l"), required = false, description
= "Spark partition lower bound.")
    var lowerBound = "0"

    @Parameter(names = Array("--upperBound", "-up"), required = false, description
= "Spark partition upper bound.")
    var upperBound = "0"

    @Parameter(names = Array("--saveFile", "-f"), required = false, description =
"Save RDD.")
    var saveFile = ""

  }

  def main(args: Array[String]): Unit = {
    println("args: " + args.mkString(","))
    new JCommander(Args, args.toArray: _*)
    println(Args.driverClassName);
    println(Args.sql);
    println(Args.url);

    val conf = new SparkConf().setAppName("SparkTest")
    val spark = new SparkContext(conf)
    val sqlContext = new SQLContext(spark)

    // Importing the SQL context gives access to all the SQL functions and
implicit conversions.
    import sqlContext.implicits._

    val before_con = Calendar.getInstance().getTimeInMillis()
```

```
    println("starting connection " + before_con)
    val dfReader = sqlContext.read
    dfReader
      .format("jdbc")
      .option("driver", Args.driverClassName)
      .option("url", Args.url)
      .option("dbtable", Args.sql)
    if (Args.numPartitions > 0) {
      dfReader
        .option("partitionColumn", Args.partitionColumn)
        .option("lowerBound", Args.lowerBound)
        .option("upperBound", Args.upperBound)
        .option("numPartitions", Args.numPartitions.toString)
    }
    val jdbcDF = dfReader.load()

    if (!Args.saveFile.isEmpty) {
          val symComm4 = jdbcDF.map(x => ((x(9)).toString,
((x(6).toString.toDouble * x(7).toString.toDouble) + x(11).toString.toDouble +
x(12).toString.toDouble)))
          val brokerSumm = symComm4.reduceByKey(_ + _)
          val homefolder=sys.env("HOME")
          brokerSumm.saveAsTextFile(homefolder+"/"+Args.saveFile)
     }
    else {
    jdbcDF.foreach(row => for (c <- 0 until row.size) row.get(c))
    }
    val after_con = Calendar.getInstance().getTimeInMillis()
    println("data retrieved " + after_con)
    val total = (after_con - before_con)
     println("time: " + total)
    spark.stop()
  }

}
```

B

FAQ: Frequently asked questions, and answers

This appendix provides high-level answers to some frequently asked questions about IBM Spark on z/OS implementation.

General

1. Won't this be too expensive on IBM z/OS due to memory?

   Though the standard pricing for IBM System z Integrated Information Processors (zIIPs) and memory might not be competitive, as part of the IBM z/OS Platform for Apache Spark offering IBM is offering specially priced zIIPs and memory. In addition to special prices on zIIP and memory specific to the Spark z/OS offering, clients are able to convert banked millions of instructions per second (MIPS) for this purpose.

2. What about currency of releases?

   At present, the plan is to refresh twice a year. However, one of the key focal points for the z/OS Spark development team is currency of refresh, so it will be maintained.

3. How will the ecosystem respond?

   We are integrated with the Spark Technology Center for IBM contributions, and any changes that are desirable in the Spark base to grow more enterprise capabilities within Spark. However, we will not create a "z/OS Spark variant' that is fundamentally different than what Spark can deliver. In addition, as seen from the announcement, we will continue to work with external ecosystem partners and projects to enhance the overall value.

Support

4. Does IBM provide proof of concept (POC) support for their distribution of Apache Spark?

   Yes.

5. Does IBM provide ongoing support for their distribution of Apache Spark?

   If you received Apache Spark as part of another IBM product, it does come with "full problem and upgrade support package" through that product's regular service channels.

   So, for example, if you have IBM BigInsights and are using its Spark capabilities, then you will get full support. You may also purchase additional services.

Technical

6. What is the difference between Spark parallelism and multiple-domain support (MDS) parallelism?

   Spark uses the *Spark core engine*. Every application programming interface (API) runs on the top of that engine. It follows a directed acyclic graph (DAG) execution engine for execution.

   The parallelism of the Spark processing cluster is determined by the total number of cores configured for the job minus the number of consumers:

   ```
   Spark Parallelism = Total # of core configured for the job - No of consumers
   ```

   There are many documents and resources to understand and learn more about Spark:

   http://ampcamp.berkeley.edu/wp-content/uploads/2012/06/matei-zaharia-part-1-amp-camp-2012-spark-intro.pdf

7. How to tune Spark on z/OS? See *Tuning on z/OS*:

   http://spark.apache.org/docs/latest/tuning.html

Related publications

The publications listed in this section are considered particularly suitable for a more detailed description 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:

- *Apache Spark for the Enterprise: Setting the Business Free, REDP-5336*
- *SMF Logstream Mode: Optimizing the New Paradigm*, SG24-7919

You can search for, view, download, or order these documents and other Redbooks, Redpapers, Web Docs, drafts, and additional materials, at the following website:

ibm.com/redbooks

Other publications

These publications are also relevant as further information sources:

- *IBM z/OS Platform for Apache Spark Installation and Customization Guide,* SC27-8449
- *IBM z/OS Platform for Apache Spark Administrator's Guide,* SC27-8451
- *IBM z/OS Platform for Apache Spark Solutions Guide,* SC27-8452
- *IBM z/OS Platform for Apache Spark User's Guide,* SC27-8450
- *Program Directory for IBM z/OS Platform for Apache Spark,* 5655-AAB

Online resources

These websites are also relevant as further information sources:

- The *IBM z/OS Platform for Apache Spark* IBM Knowledge Center:

  http://www.ibm.com/support/knowledgecenter/SSLTBW_2.2.0/com.ibm.zos.v2r2.azk/azk.htm?lang=en

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Redbooks

Apache Spark Implementation on IBM z/OS

(0.2”spine)
0.17”<->0.473”
90<->249 pages

SG24-8325-00

ISBN 0738414961

Printed in U.S.A.