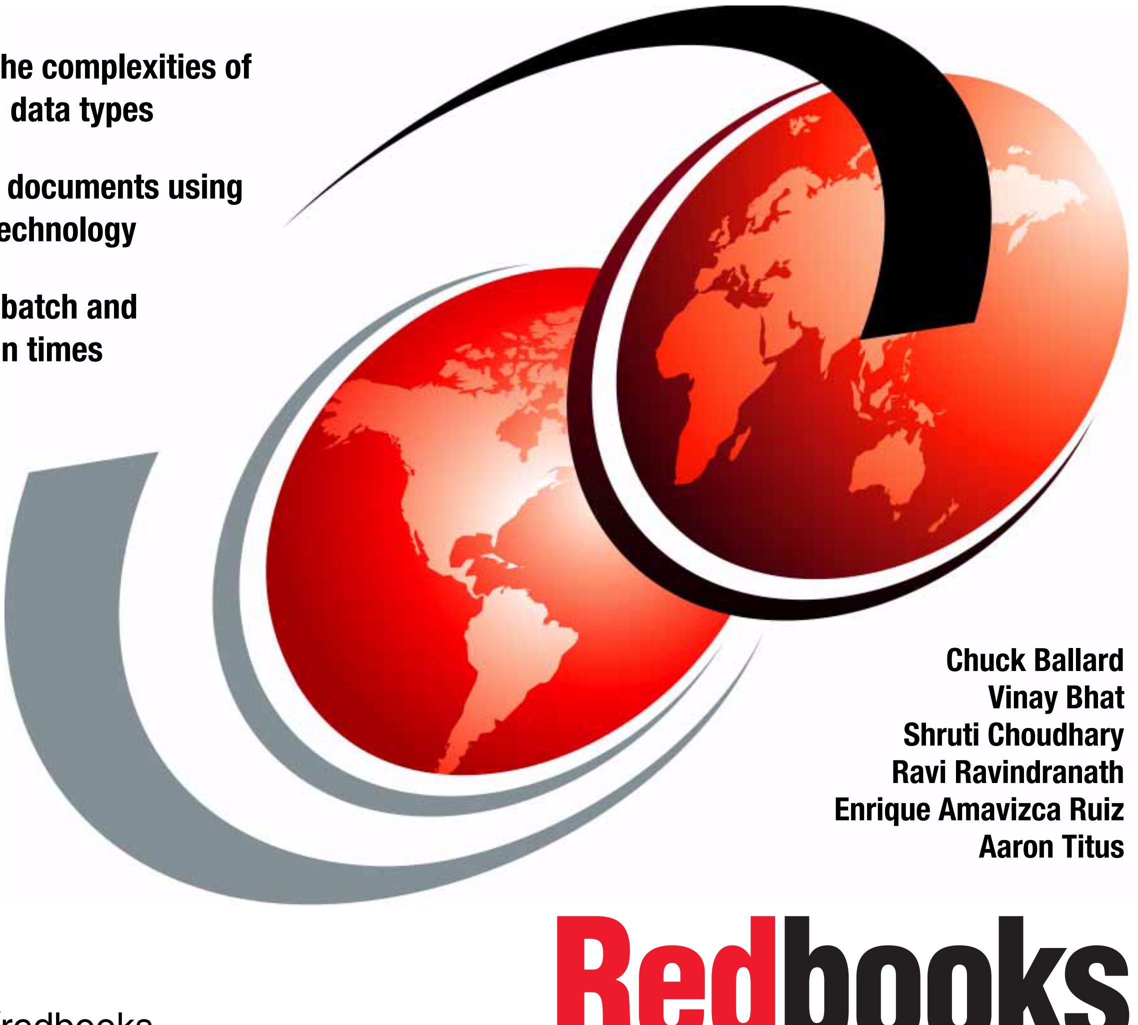
IBM

InfoSphere DataStage for Enterprise XML Data Integration

Addresses the complexities of hierarchical data types

Reads huge documents using streaming technology

Spans both batch and real-time run times



Chuck Ballard
Vinay Bhat
Shruti Choudhary
Ravi Ravindranath
Enrique Amavizca Ruiz
Aaron Titus

Redbooks

ibm.com/redbooks

International Technical Support Organization

InfoSphere DataStage for Enterprise XML Data Integration

May 2012

SG24-7987-00

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

First Edition (May 2012)

This edition applies to Version 8.7.1 of InfoSphere DataStage.

Contents

Notices

Trademarks

Preface

XML is one of the most common standards for the exchange of information. However, organizations find challenges in how to address the complexities of dealing with hierarchical data types, particularly as they scale to gigabytes and beyond. In this IBM® Redbooks® publication, we discuss and describe the new capabilities in IBM InfoSphere® DataStage® 8.5. These capabilities enable developers to more easily manage the design and processing requirements presented by the most challenging XML sources. Developers can use these capabilities to create powerful hierarchical transformations and to parse and compose XML data with high performance and scalability. Spanning both batch and real-time run times, these capabilities can be used to solve a broad range of business requirements.

As part of the IBM InfoSphere Information Server 8.5 release, InfoSphere DataStage was enhanced with new hierarchical transformation capabilities called *XML Stage*. XML Stage provides native XML schema support and powerful XML transformation functionality. These capabilities are based on a unique state-of-the-art technology that allows you to parse and compose any complex XML structure from and to a relational form, as well as to a separate hierarchical form.

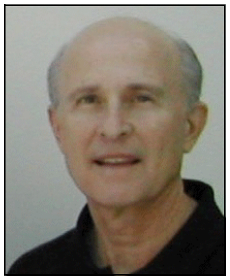
Also, XML Stage has the capability to read single huge documents using a new streaming methodology that avoids the need to load the document into memory. XML Stage has support for any type of XSD, or a collection of XSDs, to define your XML metadata. The most important capability is a whole new hierarchical editing mode called an *assembly*, which provides support for the creation of complex multi-node hierarchical structures. Much more function exists, such as the explicit control of XML validation, a built-in test facility to ease transformation development, and support for both enterprise edition (EE) and server jobs.

This book is targeted at an audience of systems designers and developers who focus on implementing XML integration support in their environments. It includes information about the following subjects:

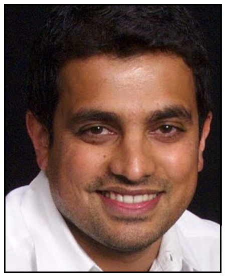
- Overview of IBM DataStage and XML technology, features, and benefits
- Integration of DataStage and XML and how they support the various platforms and environments
- Parsing, transforming, and composing XML documents
- Parallel processing and performance tuning of the environment
- Developing XML solutions, including migration and integration scenarios

The team who wrote this book

This book was produced by a team of specialists from around the world working with the International Technical Support Organization, in San Jose, California. We list the team members, along with a short biographical sketch of each team member.



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



Vinay Bhat is a Technical Sales Specialist, and a member of the sales team for IBM Information Management software products. He has over eight years of experience in architecture, design, implementation, and management of Data Integration solutions using the IBM Information Server and information management products. His experience includes extensive development of IBM Information Server DataStage, enterprise data warehouse solutions, information governance, and MDM solutions. Vinay has helped develop proven practices and deliver workshops to facilitate successful client deployments using IBM Information Server.

Shruti Choudhary is a Senior Quality Assurance (QA) Engineer at the IBM Software labs in India, with over three years of experience. She has extensive experience with Information Server having deep-rooted expertise on DataStage XML Stage. Shruti has written several IBM developerWorks® articles and has given several demos and presentations to clients on XML Stage. She has a Bachelors degree in Electronic and Communication Engineering from Visvesvaraya Technological University.

Ravi Ravindranath is a Senior Solutions Architect at the IBM Information Management Technology Ecosystem in Dallas, Texas, specializing in InfoSphere information integration middleware products. He has over 30 years of experience in lead positions in the areas of software development, product management, and pre-sales. Ravi has implemented several turn-key projects for major airlines, telecommunications, financial institutions, federal and state governments, and has contributed to the five-year IT strategy plans of the Government of Indonesia and the Government of India. He advises IBM Business Partners, mentors field sales, and presents at conferences and seminars worldwide. Ravi has a Bachelors degree in Engineering from the University of Madras and an MBA from the University of Dallas in Texas.

Enrique Amavizca Ruiz is an Information Server Consultant at the Information Management Technology Ecosystem (IMTE) in Mexico D.F. He has over 16 years of experience in positions in the areas of development, technical support, systems administration, management, and consulting. His expertise is in the areas of database, data integration, and software development. He has worked extensively on these subjects, and taught Information Server courses in North and South America. Enrique holds a Bachelors degree in Computer Systems Engineering from Universidad de las Americas-Puebla.

Aaron Titus is an Advanced Technical Support Engineer for the IBM InfoSphere Platform products, located in Littleton, Massachusetts. He has provided technical support for Information Server, DataStage, and other data integration products since 1999. Aaron has taught classes to clients and IBM Business Partners, and presented workshops and demos at user conferences. His expertise is with data integration technology, specializing in parallel processing, clustering and GRID deployments, real time, and service-oriented architectures (SOA).

Other contributors

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

From IBM locations worldwide

Ernie Ostic - Client Technical Specialist, IBM Software Group, Worldwide Sales Enablement, Piscataway, NJ
Tony Curcio - InfoSphere Product Management, Software Product Manager, Charlotte, NC

From the International Technical Support Organization

Mary Comianos - Publications Management
Ann Lund - Residency Administration
Emma Jacobs - Graphics Support

Now you can become a published author, too!

Comments welcome

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`

Heterogeneous data integration

Most key business initiatives require the integration of data from heterogeneous data sources. These initiatives can be accomplished by developing a custom solution for each data source, as depicted in Figure 1-1 on page 2. In such an architecture, as the number of data sources increases, the complexity required to integrate them increases. Costs for managing the infrastructure skyrocket. Using the eXtensible Markup Language (XML) to standardize the exchange of information between disparate sources is becoming a modern, industry standard solution. This book addresses the XML technology and the use of IBM InfoSphere DataStage to successfully delivery enterprise-wide heterogeneous data integration.

In this book, we discuss in detail the creation of Information Server DataStage jobs that integrate disparate data using XML technology. You can understand the topics in this book better if you have a basic working knowledge of InfoSphere DataStage, and general understanding of XML concepts. You can obtain more information about how to develop DataStage jobs, and reference information about XML technologies in “Related publications” on page 375.

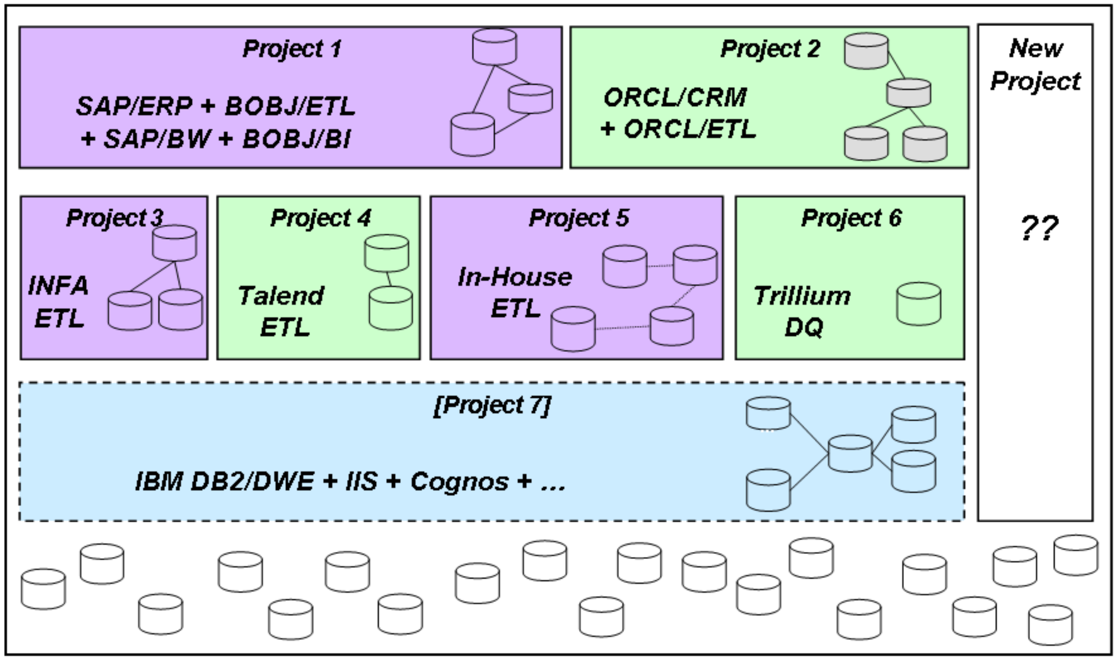


Figure 1-1 Heterogeneous data integration

Enterprise data integration initiatives present many challenges. You might deploy enterprise services using a service-oriented architecture (SOA), implementing a cloud initiative, or building the infrastructure for Dynamic Data Warehousing. For many of these tasks, XML can be an important building block for your integration architecture.

Implementing innovative solutions using XML technology is an integral part of every enterprise. The XML Transformation capabilities of InfoSphere DataStage can handle large, complex XML documents and process thousands of real-time transactions with Web Services or message bus implementations. XML-based solutions offer high flexibility and ease of adaptation, making them useful in a wide spectrum of industries and business solutions. In this book, we highlight several of the most common XML-based solutions, and how to use IBM InfoSphere DataStage to successfully implement these solutions in your enterprise.

1.1 Importance of XML in the industry

XML technology is pervasive in virtually all industries and sectors, because of its versatility and neutrality for exchanging data among diverse devices, applications, and systems from various vendors, as depicted in Figure 1-2. These qualities of XML and its easy to understand self-describing nature, ability to handle structured, semi-structured, and unstructured data, and support for Unicode make XML a universal standard for data interchange.

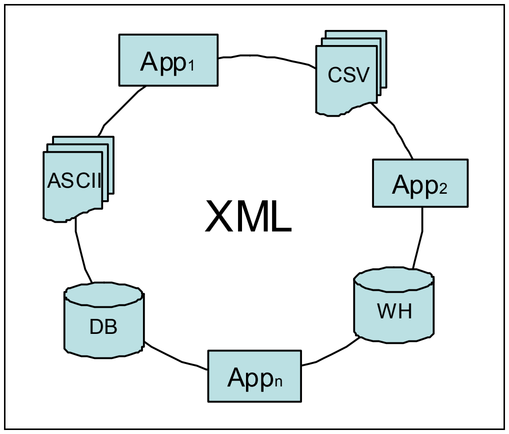


Figure 1-2 Data exchange with XML

1.1.1 Growth of XML

Nearly every company today comes across XML in some form. The amount of XML data with which organizations work is growing at a rapid rate. In fact, it is estimated that the volume of XML data is growing twice as fast as that of the traditional data that typically resides in relational databases. The following factors fuel the growth of XML data:

- XML-based industry and data standards
- Service-oriented architectures (SOAs) and Web Services
- Web 2.0 technologies, such as XML feeds and syndication services

XML-based industry standards

Almost every industry has multiple standards based on XML, and there are numerous cross-industry XML standards, as well.

The following list includes a few examples of XML-based industry standards:

- ACORD - XML for the Insurance Industry:

  `http://www.acord.org/`
- FPML - Financial Product:

  `http://www.fpml.org/`
- HL7 - Health Care:

  `http://www.hl7.org/`
- IFX - Interactive Financial Exchange:

  `http://www.ifxforum.org/`
- IXRetail - Standard for Retail operation:

  `http://www.nrf-arts.org/`
- XBRL - Business Reporting/Accounting:

  `http://www.xbrl.org/`
- NewsML - News/Publication:

  `http://www.newsml.org/`

These standards facilitate purposes, such as the exchange of information between the various players within these industries and their value chain members, data definitions for ongoing operations, and document specifications. More and more companies adopt XML standards or are compelled to adopt them to stay competitive, improve efficiencies, communicate with their trading partners or suppliers, or perform everyday tasks.

1.2 XML business benefits

The benefits of a common base upon which the technical world can build layers upon layers of technical innovation are enormous. These benefits are only possible if this standard is agreed to by all.

The following list shows the primary XML applications:

- Sharing information: The main problem with integrating data between any two business organizations is the interface between them. If they can at least agree upon the standard of their common meeting point and its usage, they can build upon this standard to start building their applications. If there is already an existing interface or infrastructure provided by industry or government standard or infrastructure, the business cost of developing it is extinguished.

- Storage, transmission, and interpretation of data: If the storage of information is adaptable to many mediums, its cost is driven to the lowest cost of all mediums. XML is based on text, which obviously is accepted by all. It can be read. Its storage is cheap, compared to the storage requirements of graphics. And, because the size of this text-based object is small, its transmission, not withstanding cost, is inexpensive as well. And, it is commonly accepted, because it adheres to worldwide standards and it is easily interpreted.
- Security: With the explosion of confidential information transmitted across the Internet, sophisticated levels of security are now necessary. Companies need to protect their documents and email, banks need to allow their depositors to download their accounts, and merchants need to be available for their customers to enter their credit card details without compromising their security and privacy. The more secure the transmission medium, the more confidentiality it provides, and the more it can be used to advertise a competitive advantage. With the evolution of XML digital signatures, the ability to protect or hide parts of a document, while sitting on a PC, server or mainframe, now covers up a security patch.
- Speed and the amount of content delivery: With the rapid evolution of network technologies, the speed and delivery of any content continue to grow in importance as a product evaluation criteria. Network companies now advertise download times of movies and CDs. Again, the first companies discovering the abilities of delivering something at an increasing speed without compromising their content gains the competitive advantage.

1.2.1 Information sharing

XML has been readily accepted by the technical world because of its simplicity. The benefits of having a common format to share information between any two organizations are obvious. Technologies and standards have been built upon XML. Consortiums and business organizations have developed industry-wide XML formats and standardization. Examples are inter-bank payments between banks, insurance companies, and trading agencies; supply chains between manufacturers, distributors, and retailers; and battlefield information between soldiers, satellites, and defense analysts.

In December 2000, the United Nations Centre for Trade Facilitation and Electronic Business (UN/CEFACT) and the Organizations for the Advancements of Standard Information Standards (OASIS) came together to initiate a project to standardize XML specifications for business. This initiative, which is called the Electronic Business XML (ebXML), developed a technical framework that enabled XML to be utilized for all exchange of all electronic business data. The main aim of ebXML was to lower the cost and difficulties, focusing on small and medium-sized business and developing nations, in order to facilitate international trade.

Another example, BizTalk, is an initiative supported by Microsoft, SAP, and Boeing, among other major vendors. However, BizTalk is not a standard. BizTalk is more of a community of users. It aims to enable the consistent adoption of XML to ease use and, therefore, easily adopt electronic commerce and application integration.

1.2.2 XML within an organization

With the emergence of Customer Relationship Management (CRM) and Enterprise Architecture Integration (EAI), client-oriented organizations re-engineered their systems to provide a whole new experience for the consumer.

CRM and EAI involve bringing together a multitude of applications and systems from multiple vendors to behave as a coordinated whole. The common thread among all these applications is XML. Imagine trying to integrate information from billing histories, customer personal information and credit histories, distribution systems, and workforce management systems; and then displaying them using browsers, databases, processes, and workflow systems on a variety of technical platforms, such as mainframes, PCs, and medium-sized servers. All this information can be shared by using XML, because it is a versatile, powerful, and flexible language. It has the ability to describe complex data. It is also extensible, allowing applications to grow and develop without architectural re-engineering.

IBM uses XML in all its new tools. The IBM applications WebSphere® Studio, DB2®, and WebSphere Application Server are based on XML with extensibility as a major advantage.

1.2.3 XML in new innovations

The innovations based on XML are too numerous to list. So many new ideas are now based in XML that entrepreneurial companies will find it difficult to avoid the use of XML and its standards.

Voice XML

In October, 2001, the World Wide Web Consortium (W3C) announced the first release of a working draft for the Voice Extensible Markup Language (VoiceXML). VoiceXML has these purposes among others:

- Hides designers and developers from low-level platform-specific details.
- Promotes portability across implementation platforms.
- Offers a common language for platform providers, development tool providers, and content providers.

- Provides features to support complex dialogs, yet it is easy to use for simple interactions.
- Caters to audio dialogs that feature digitized audio, speech recognition, telephony, and synthesized speech. Its goal is to deliver the advantages of web-based development and context delivery for interactive voice response applications

Scalable Vector Graphics

Scalable Vector Graphics (SVG) is an XML-based language for describing two-dimensional graphics. Its main use is in Geographical Information Systems (GIS) where travel maps, council boundaries, forest fires, and natural disasters can be processed and displayed independently of technical platforms. It is able to integrate with non-GIS software, and also allow graphic elements to be non-graphic.

The benefits of this new XML derivative are not difficult to comprehend. Its uses are many. For example, insurance companies can estimate and forecast natural disaster claims. Scientists can study environmental impacts. And, local and federal governments can perform city and town planning.

1.3 Technical benefits of XML

The basics of XML do not change. What changed is the extent that XML is used today and how it is incorporated into technologies for the future. XML offers many benefits, several of which are listed.

Acceptability of use for data transfer

XML is not a programming language. It is a standard way of putting information in a format that can be processed and exchanged across hardware devices, operating systems, software applications, and the web. It is such a common medium of data that it enables the transmission and retrieval, and storage of information over the Internet across company boundaries, making it a natural choice for data management for e-business transactions.

Uniformity and conformity

The inability of two computer systems or applications to talk to each other is always a challenge. When two applications are integrated, the business and technical experts must decide either to integrate the two systems, or to re-architect the applications. If data from both applications conforms to a format and is easily transformed from one to another, development costs can be reduced. If this common format can be further developed and is accepted industry-wide, interfacing the applications to other applications is less costly.

Simplicity and openness

Information that is coded in XML is visually read and accepted. Because it can be processed by computers easily, XML is widely accepted by major vendors in the computing world. Microsoft has indicated that it will use XML as the exchange format for its Microsoft Office software suite. Both Microsoft and Netscape web browsers support XML.

XML garners interest, because it is simple to understand, implement, and use. It follows the Pareto principle, an 80/20 solution, meaning that it supplies about 80% of the functionality of competing technologies with perhaps 20% of the effort that is required to build enterprise-level solutions.

XML is not a total solution for every problem in e-business, but it is making significant inroads in communications between old computer programs. Therefore, these old programs last longer, saving money and time, which are important when both are so precious to the bottom line.

Separation of data and display

Separation of data and its display is not a new paradigm in the computing world. In the last few years, application designers have raised the concept of the Model-View-Controller (MVC) approach to building applications for many reasons. Without separation of data, re-use of that data in multiple user interfaces is difficult. Websites evolved radically over the last eight years. Every year, websites must be upgraded to compete for consumer attention. Websites need better attention-getting displays and response times. If the data must be rehashed every time that an upgrade to the website is required, the development costs increase. However, if you can reuse or build upon existing data, you can save on redevelopment costs.

Imagine a navigation system that is used by consumers to move from one place to another. This system has street maps, address information, local attractions, and other information. If this information must be displayed on a web browser, personal device assistant (PDA), or a mobile phone, it is a major development cost if we must develop three data access systems and three data presentation systems. However, if we develop one data access system, we also need to create three data presentation files for each system. We transform XML using the Extensible Stylesheet Language Transformation (XSLT) feature.

Extensibility

HTML has a major problem in that it is not extensible. It is enhanced by software vendors. These enhancements are not coordinated and, therefore, non-standard. It was never designed to access data from databases. To overcome this deficiency, Microsoft built Active Server Pages (ASP) and Sun produced JavaServer Pages (JSP).

XML, however, was designed initially to allow extensions.

Industry acceptance

XML is accepted widely by the information and computing industry. It is based on common concepts. A large number of XML tools will emerge from both existing software vendors and XML start-up companies. XML is readable by every operating system, because it is in ASCII text. XML can be seen by any text editor or word processor.

The tree-based structure of XML is much more powerful than fixed-length data formats. Because objects are tree structures as well, XML is ideally suited to working with object-oriented programming.

1.4 DataStage as enterprise data integrator

IBM InfoSphere Information Server is a software platform that helps organizations derive more value from the complex, heterogeneous information that is spread across their systems. It provides breakthrough collaboration, productivity, and performance for cleansing, transforming, and moving this information consistently and securely throughout the enterprise, as depicted in Figure 1-3 on page 10. It can then be accessed and used in new ways to drive innovation, increase operational efficiency, and lower risk.

As a component of IBM InfoSphere Information Server, IBM InfoSphere DataStage integrates data across multiple, high-volume data sources and target applications. It integrates data on demand with a high-performance parallel framework, extended metadata management, and enterprise connectivity. DataStage supports the collection, integration, and transformation of large volumes of data, with data structures ranging from simple to highly complex.

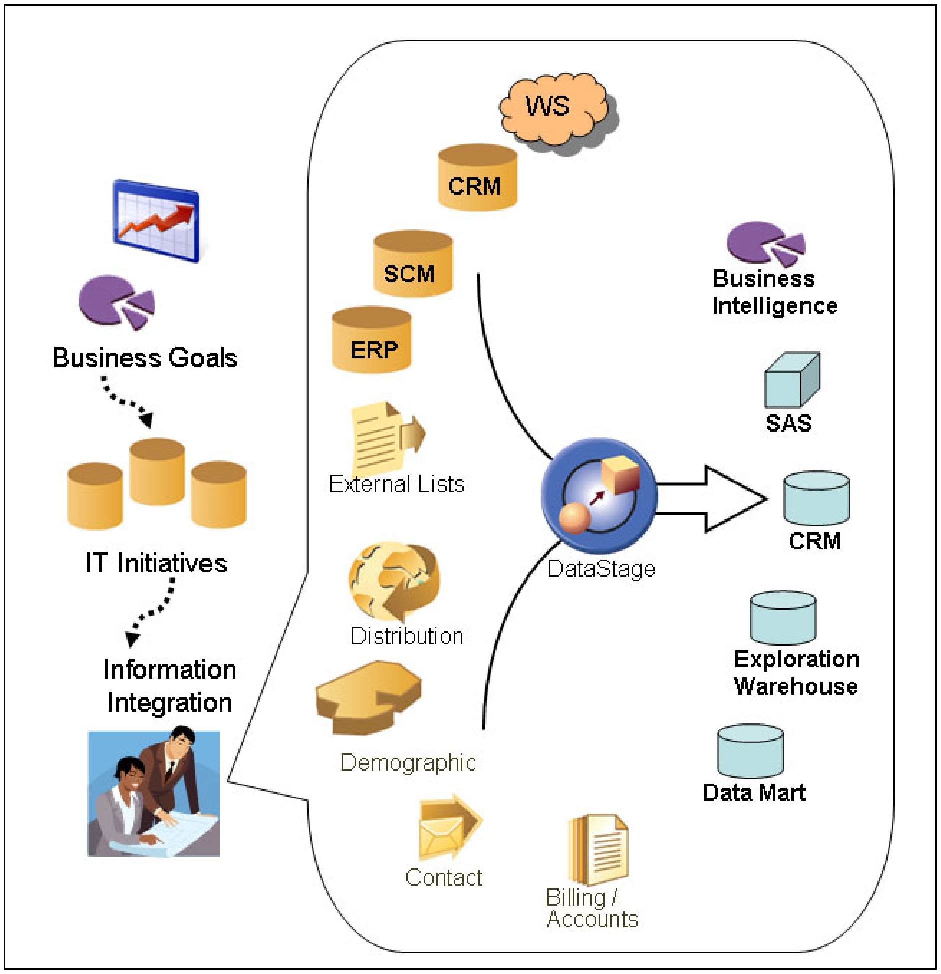


Figure 1-3 DataStage as data integrator

DataStage can manage data arriving in real time, as well as data received on a periodic or scheduled basis, which enables companies to solve large-scale business problems through high-performance processing of massive data volumes. By making use of the parallel processing capabilities of multiprocessor hardware platforms, IBM InfoSphere DataStage Enterprise Edition can scale to satisfy the demands of ever-growing data volumes, stringent real-time requirements, and ever-shrinking batch windows.

Along with these key components, establishing consistent development standards helps to improve developer productivity and reduce ongoing maintenance costs. Development standards can also make it easier to integrate external processes, such as automated auditing and reporting, and to build technical and support documentation.

1.4.1 DB2 pureXML integration

With the introduction of DB2 9 to the market, IBM pureXML® is a new feature that provides the capability of storing XML data natively in a database table. Before DB2 9, management of XML data involved these common approaches:

- Storing XML documents on file systems
- Stuffing XML data into large objects (LOBs) in relational databases
- Shredding XML data into multiple relational columns and tables
- Isolating data into XML-only database systems

Frequently, these obvious choices for managing and sharing XML data do not meet performance requirements. File systems are fine for simple tasks, but they do not scale well when you have hundreds or thousands of documents. Concurrency, recovery, security, and usability issues become unmanageable.

With the release of DB2 9, IBM is leading the way to a new era in data management. DB2 9 embodies technology that provides pure XML services. This pureXML technology is not only for data server external interfaces; rather, pureXML extends to the core of the DB2 engine. The XML and relational services in DB2 9 are tightly integrated. They offer the industry's first pureXML and relational hybrid data server. Figure 1-4 illustrates the hybrid database.

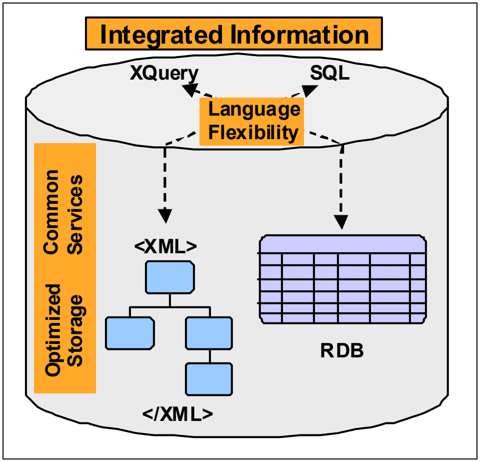


Figure 1-4 pureXML and relational hybrid database

The pureXML technology in DB2 9 includes the following capabilities:

- pureXML data type and storage techniques for the efficient management of hierarchical structures common in XML documents
- pureXML indexing technology to speed up searches of subsets of XML documents
- New query language support (for XQuery and SQL/XML) based on industry standards and new query optimization techniques
- Industry-leading support for managing, validating, and evolving XML schemes
- Comprehensive administrative capabilities, including extensions to popular database utilities
- Integration with popular application programming interfaces (APIs) and development environments
- XML shredding and publishing facilities for working with existing relational models
- Enterprise-proven reliability, availability, scalability, performance, security, and maturity that you expect from DB2

1.4.2 Real-time integration

Much of the DataStage history focuses on batch-oriented applications. Many mechanisms and techniques were devised to process huge amounts of data in the shortest amount of time, in the most efficient way, in that environment. We are aware that concerns exist relating to limiting the number of job executions, processing as much data as possible in a single job run, and optimizing the interaction with the source and target databases (by using bulk SQL techniques). All these concerns relate to bulk data processing. Batch applications tend to be bound to time constraints, such as batch execution windows, which typically require at least a few hours for daily processing. However, several applications do not process huge amounts of data at one time, but rather process several small, individual requests. The number and size of requests are not minimal. They might actually be relatively large, but not as large as high-volume batch applications. For high-volume scenarios, users typically resort to batch-oriented techniques. The types of applications in this section focus on scenarios with a large number of requests, each of varying size, spanning a long period of time.

The following list describes these types of applications:

- Real-time transactions:
  - Business-to-consumer applications: A person waits for a response, such as in a web-based or online application.
  - Business-to-business applications: The caller is another application.

- Real-time reporting/decision making:
  - Few or no updates to database tables occur.
  - Fetching and massaging large amounts of data are involved.
  - The application generates support for a specific decision, such as an insurance approval.
- Real-time data quality:
  - Other applications invoke IBM InfoSphere QualityStage® for cleansing and de-duplication, for example, the use of Master Data Management (MDM). See 1.4.3, "Master Data Management" on page 15.
- Integration with third-party business process control systems

  A business process coordinator orchestrates the processing of a series of steps, involving multiple participants. One of the participants might be DataStage, which receives requests and posts results back to the orchestrator through Java Message Services (JMS), for example. A third-party example is TIBCO.
- Near-real time:
  - Applications are message-oriented for instant access and update.
  - Data trickles from a source mainframe to populate active data warehouses and reporting systems.

Two capabilities were added to DataStage over time to address the needs of real time:

- Message Queue/Distributed Transaction Stage (MQ/DTS): MQ/DTS addresses the need for guaranteed delivery of source messages to target databases, with the once-and-only-once semantics. This type of delivery mechanism was originally made available in DataStage 7.5 in the form of the Unit-of-Work (UOW) stage. The original target in DS 7.5 was Oracle. In InfoSphere DataStage 8.x, this solution is substantially upgraded (incorporating the new database connector technology for various types of databases) and rebranded as the Distributed Transaction Stage.
- Information Services Director (ISD): ISD enables DataStage to expose DS jobs as services for service-oriented applications (SOA). ISD supports, as examples, the following types of bindings:
  - SOAP
  - Enterprise JavaBeans (EJB)
  - JMS

DTS and ISD work differently and serve separate purposes. In this section, we describe the specifics of each capability. However, common aspects relate to both capabilities:

- Job topologies: Real-time jobs of any sort need to obey certain rules so they can operate in a request/response fashion.
- Transactional support:
  - Most real-time applications require updating a target database. Batch applications can tolerate failures by restarting jobs, as long as the results at the end of processing windows are consistent.
  - Real-time applications cannot afford the restarting of jobs. For each request, the net result in the target database must be consistent.
- End-of-wave:
  - The parallel framework implements virtual datasets (memory buffers) that are excellent for batch applications. Virtual datasets are a key optimization mechanism for high-volume processing.
  - Real-time applications cannot afford records sitting in buffers waiting to be flushed.
  - The framework was adapted to support end-of-wave, which forces the flushing of memory buffers so responses are generated or transactions committed.
- Payload processing:
  - Frequently, batch applications deal with large payloads, such as big XML documents.
  - Common payload formats are COBOL and XML.
- Pipeline parallelism challenges: Fundamental for performance, the concept of *pipeline parallelism* introduces challenges in real time, but those challenges can be circumvented.

Definition of real time

The concept of real time is not consistent with applications that are typically used for embedded systems, such as avionics, traffic control, and industrial control. Similarly, DataStage is typically not used for embedded applications or for scenarios in which there are stringent timing constraints.

The term *real time* applies to the following types of scenarios:

- Request/response: For example, a caller waits for a response. The caller expects to receive a response quickly. There is no physical tragedy, such as a transportation accident, that will occur if the response takes too long. Instead, the caller might receive a time out, for example:

  - Business-to-consumer, such as web applications
  - Online data entry applications
  - Business-to-business
- The following DS solutions can be applied:
  - ISD with SOAP or EJB binding
  - ISD with JMS binding, when the caller expects the response to be placed on a response queue

The term *near-real time* is used and applies to the following types of scenarios:

- Message delivery:
  - The data is delivered and expected to be processed immediately.
  - Users can accept a short lag time (ranging from seconds to a few minutes).
  - No person waits for a response, for example:
    - Reporting systems
    - Active warehouses
  - The following DS solutions can be applied:
    - MQ → DTS/UOW
    - ISD with text over JMS binding

The high cost of starting up and shutting down jobs demanded that DataStage was enhanced with additional capabilities to support these types of scenarios. Implementing these types of scenarios with batch applications is not feasible.

1.4.3 Master Data Management

IBM InfoSphere Master Data Management (MDM) allows you to gain a unified, complete, consistent and standardized view of your data to drive critical business decisions. MDM also helps you gain control of your data, reduce information errors and eliminate duplicate data. All of which helps you to meet your growth, revenue-generation, and cost-reduction goals.

IBM InfoSphere MDM Server is a master repository that delivers a single version of an organization's data entities, such as customer, product, and supplier. Its SOA library of prepackaged business services allows organizations to define how they want users to access master data and seamlessly integrate into current architectures and business processes.

When business entities are deployed, typically data is loaded into an MDM repository, but most data changes come from existing systems. MDM implementations require a set of InfoSphere Information Server DataStage jobs (see Figure 1-5) to perform the initial and delta loads directly to the MDM database:

- An *initial load* is an original movement of data from source systems into the MDM repository when the repository is empty.
- A *delta load* is a periodic (typically daily) data update from source systems into MDM.

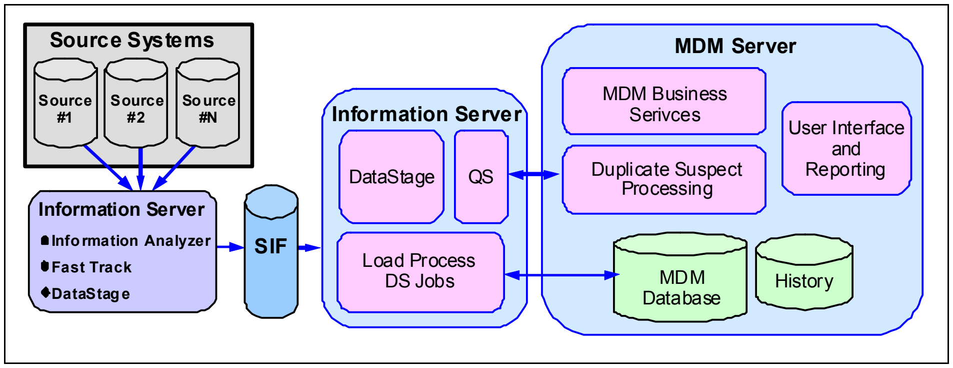


Figure 1-5 MDM server implementation

1.4.4 Web Services and XML

Web Services have gained prominence in the last few years. *Web Services* are self-contained, modular, self-explanatory applications that are published on the web. They provide functions to encapsulate objects from single functions to complex business functions. Examples of simple functions are a calculator, spreadsheet, or tutorial. Examples of complex functions are processing a tax return, receiving stock quotes, or processing credit card transactions. After this web service is deployed, anyone, either another application or web service, can locate and invoke it.

Web Services involve a few components: SOAP; Universal Description, Discovery, and Integration (UDDI); and Web Services Description Language (WSDL). These components work together, as depicted in Figure 1-6 on page 17, to interact between applications or clients, to publish themselves to be consumed by other clients and to describe themselves.

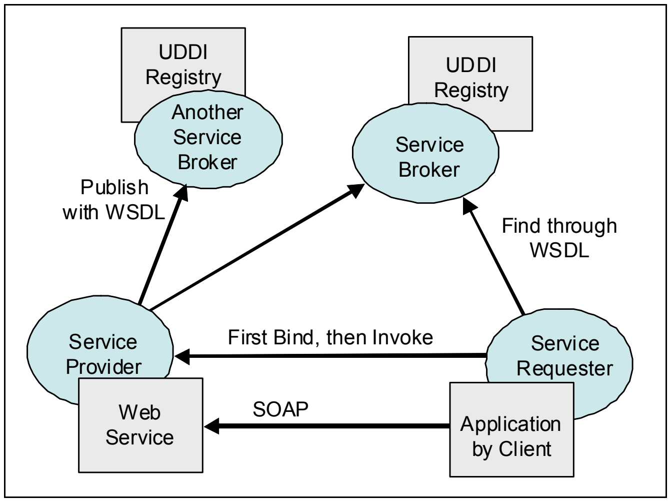


Figure 1-6 Interaction of Web Services

SOAP is an XML-based protocol that allows applications to invoke applications that are provided by service providers anywhere on the web. SOAP is supported by HTTP and, therefore, can be run on the Internet without any new requirements over an existing infrastructure. SOAP is independent of any programming language and component technology, and it is object neutral. It is also independent of operating systems.

Universal Description Discovery and Integration (UDDI) is a specification for web registries of Web Services. Web users locate and discover services on a UDDI-based registry. Registries of services are distributed over the Internet, and these registries of services are described in a common XML format or schema. With a common format, the searching and analysis of applications are easier. A UDDI registry is available from the IBM alphaWorks® website, and it supports users in various department-wide or company-wide scenarios.

Web Services Description Language (WSDL) is a language that is used to describe a service to the world. WSDL is defined in the following description from the World Wide Web Consortium (W3C)[1]:

"WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services). WSDL is extensible to allow the description of endpoints

[1] World Wide Web Consortium at http://www.w3.org/TR/wsdl

and their messages regardless of what message formats or network protocols are used to communicate. However, the only bindings described in this document describe how to use WSDL in conjunction with SOAP 1.1, HTTP GET/POST, and MIME."

1.4.5 Real-time integration using IBM InfoSphere Change Data Capture

IBM InfoSphere Change Data Capture (CDC) is an enterprise data synchronization and change capture solution. InfoSphere Change Data Capture can be used to capture changes in near-real time as data is inserted, updated, and deleted in tables in a source database. These captured changes are then transformed to meet the target database characteristics and propagated to tables in target databases, directly integrated into DataStage or trickle-fed to drive other processes via messaging, as depicted in Figure 1-7.

The source and target databases can potentially be databases from separate vendors running on separate platforms. Propagated data can involve user data that is stored both in relational format, as well as XML, and can include audit data that is hidden in the database logs.

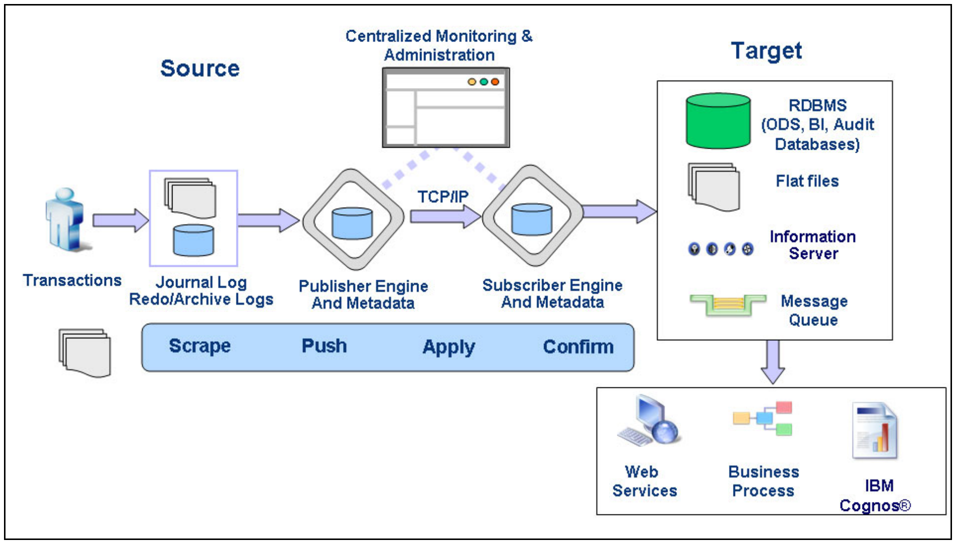

Source
Centralized Monitoring & Administration
Target
RDBMS (ODS, BI, Audit Databases)
Flat files
Information Server
Message Queue
TCP/IP
Transactions
Journal Log Redo/Archive Logs
Publisher Engine And Metadata
Subscriber Engine And Metadata
Scrape
Push
Apply
Confirm
Web Services
Business Process
IBM Cognos®

Figure 1-7 InfoSphere Change Data Capture Enterprise Architecture

The flow of data through Change Data Capture can be divided into three parts.

Capture

Changes are captured by asynchronously reading from the database recovery logs. Depending on the source database, InfoSphere Change Data Capture

might read the log files directly or it might use an application programming interface (API) to read data from the log. InfoSphere Change Data Capture has native log-based capture capabilities for the following databases:

- IBM DB2 z/OS®
- DB2 for IBM i
- DB2 Linux, UNIX, and Microsoft Windows
- Oracle
- SQL Server
- Sybase

Upon reading data from the database recovery logs, InfoSphere Change Data Capture filters data based on the table where the change occurred. Only data pertaining to tables of interest is retained for further processing. InfoSphere Change Data Capture then stages the changes in a holding area until a commit of these changes occurs in the source database. If a rollback occurs instead, InfoSphere Change Data Capture discards the associated changes.

Transformations

In many cases, the data models of the source and target are not the same. InfoSphere Change Data Capture can apply transformations to the data while it is in flight between the source and target. The following transformations are commonly used:

- Adding other information that can be obtained from the database recovery logs
- Concatenation and other string functions
- Data type conversion
- Arithmetic
- Joining to look-up data in the secondary tables
- If/then/else logic

For more complex transformations, IBM InfoSphere DataStage is used in conjunction with IBM InfoSphere Change Data Capture.

Apply

After transformations, the changes are applied to the target database. Changes are applied by executing SQL statements against the appropriate target tables.

InfoSphere Change Data Capture has native “Apply” capabilities for the following databases:

- DB2 z/OS
- DB2 on IBM i
- DB2 Linux, UNIX, and Windows
- Oracle
- SQL Server
- Sybase
- Teradata

InfoSphere Change Data Capture also provides a level of customization around the SQL statements that are executed as part of Apply processing. For example, it is possible to perform these tasks:

- Ignore deletes that occur at the source
- Convert source updates and deletes into target inserts (an audit or change history table)
- Convert source inserts into target updates (multiple source records merge to one target record)

InfoSphere Change Data Capture also supports several non-database targets:

- Flat files
- InfoSphere DataStage
- WebSphere MQ
- JMS messaging
- Netezza®

1.4.6 Real-time integration using IBM Replication Server

InfoSphere Change Data Capture also works with InfoSphere Replication Server, which is the IBM strategic data replication product for the IBM DB2 family. InfoSphere Replication Server has a large well-established client base, is a key component of the IBM Active-Active strategy, and is developed in close cooperation with DB2 development teams. InfoSphere Replication Server supports two replication architectures: SQL Replication that uses intermediate staging tables and Q Replication that uses IBM WebSphere MQ for transport, as shown in Figure 1-8 on page 21.

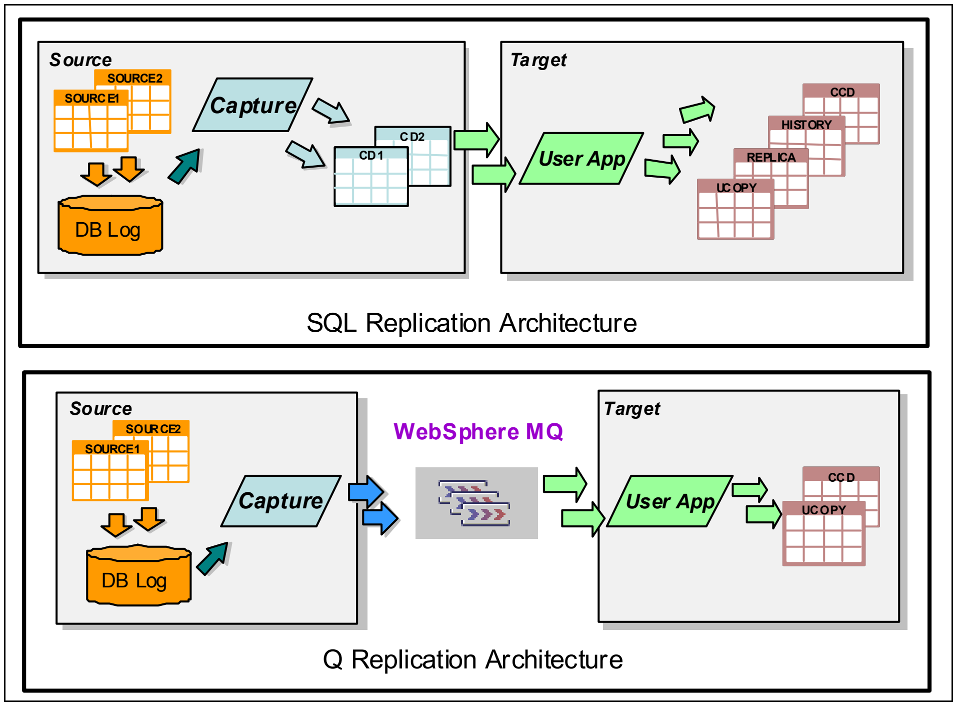


Figure 1-8 InfoSphere Replication Server Architecture

Similar to InfoSphere Change Data Capture, as described in 1.4.5, "Real-time integration using IBM InfoSphere Change Data Capture" on page 18, InfoSphere Replication Server also has Capture and Apply components, reads from DB2 logs, and performs data transformations prior to applying data to the target database tables. The difference arises in the storage and transport methods that are used, the supported data sources, and tight integration with DB2.

Propagated data can involve user data that is stored as relational data or as XML and can include audit data from the database logs. Captured changes are also transformed to meet the target database characteristics. Then, they are propagated to target DB2 tables, directly integrated into DataStage, or trickle-fed to drive other processes via messaging, as depicted in Figure 1-9 on page 22.

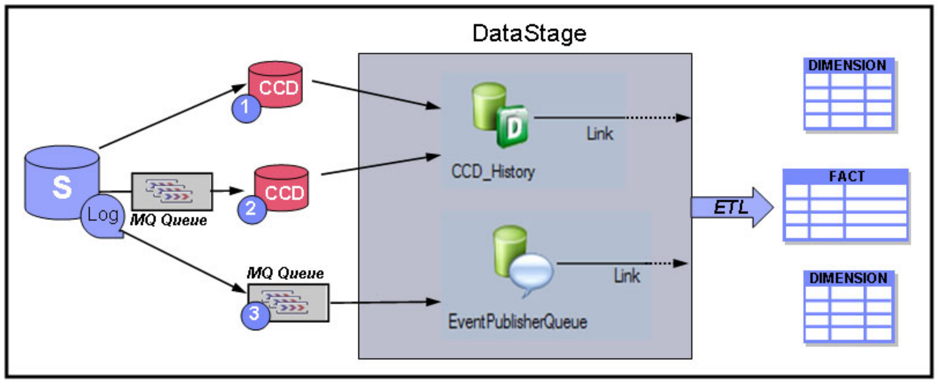


Figure 1-9 Replication Server - DataStage Integration Architecture

1.4.7 Real-time integration using Data Event Publishers

IBM InfoSphere Data Event Publisher facilitates business integration by linking changed data events in DB2 databases on Linux, UNIX, and Windows with enterprise application integration (EAI) solutions. EAI solutions include WebSphere Business Integrator, message brokers such as WebSphere Message Broker, and data transformation tools such as InfoSphere DataStage. DB2 changed-data events are captured and published as XML messages or comma-separated value (CSV) data via WebSphere MQ that can be used by other applications and tools to drive subsequent processing.

InfoSphere Data Event Publisher consists of the Capture component that is used by InfoSphere Replication Server and WebSphere MQ transport. However, it does not use the Apply component. Instead, the changed data is converted into either XML messages or CSV data and written to the WebSphere MQ queue. Consuming clients can retrieve the message at the other end of the queue and do not provide any feedback to the Capture component. This type of architecture eliminates the need for handshakes between a sending (Capture) process and a receiving (Apply) process. Therefore, this architecture provides flexible change-data solutions that enable many types of business integration scenarios, as shown in Figure 1-10 on page 23.

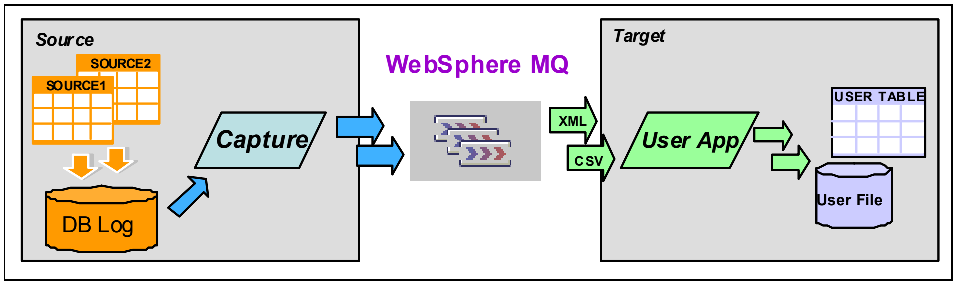


Figure 1-10 Data Event Publisher Architecture

The following methods are examples of other integration approaches:

- Application-to-application integration: Changes to operational customer data can be pushed to data integration solutions, such as InfoSphere DataStage for updating customer relationship management (CRM) application data.
- Business process initiation: Adding a customer record can initiate a WebSphere Business Integrator workflow that results in a welcome email, a credit verification, and an update to the CRM system.
- Critical data event monitoring: Changed-data events can be delivered to WebSphere MQ Broker so that a low-inventory level can be used to initiate a product restocking workflow.
- Change-only data population: Data changes in operational systems can be fed to InfoSphere DataStage or another extract, transform, and load (ETL) product. The ETL product then populates a data warehouse, data mart, or operational data store with the changes.

1.4.8 Real-time integration using Classic Data Event Publishers

Another type of Data Event Publisher is the IBM InfoSphere Classic Data Event Publisher that drives business integration and ETL processes by publishing changes made to IBM System z® data.

It captures database changes to IBM IMS™, VSAM, Computer Associates CA-IDMS, or Software AG Adabas during active log journal processing or by reading source-specific recovery logs. It therefore provides reliability and recoverability that empower near-real-time delivery. Captured data is automatically reformatted into a consistent relational format before it is published to WebSphere MQ or Java Message Queues to simplify and maximize integration possibilities.

InfoSphere Classic Data Event Publisher for z/OS V9.5 provides a new publication interface for use by InfoSphere DataStage and InfoSphere DataStage for Linux on System z. Data received from existing sources can be processed and converted into XML using the XML Transformation stage within InfoSphere DataStage.

1.4.9 Summary

In this chapter, we discuss the benefits of using XML technologies that are widely accepted by various industrial organizations throughout the world. We also describe a number of IBM solutions that facilitate data integration and how to use current and future investments. The focus of this book is the XML integration capabilities that are built into DataStage. It is imperative to have a basic understanding of XML and a working knowledge of DataStage to fully appreciate the concepts and solutions that are presented in this book.

In subsequent chapters, we provide an overview of XML and an introduction to DataStage. The following chapters provide detailed information about the XML capabilities available in DataStage, working with the XML Pack and the new XML Transformation stage, and performance considerations for working with large and complex XML data. The final chapter provides examples of integration scenarios and use case scenarios.

2

XML overview

In this chapter, we introduce you to the eXtensible Markup Language (XML). This book is not designed to teach you everything about XML, but it is important to understand the basic concepts. In the following sections, we describe what XML is (and is not), and discuss parsers, schemas, namespaces, Extensible Stylesheet Language (XSL), Web services, and other XML-related technologies.

2.1 XML introduction

The idea of universal data formats is not new. Programmers tried to find ways to exchange information between computer programs for a long time. First, Standard Generalized Markup Language (SGML) was developed. SGML can be used to mark up data, that is, to add metadata in a way that allows data to be self-describing. SGML is metalanguage.

Extensible Markup Language (XML) is a simple, flexible text format that is designed to be used in data interchange on the web and in other electronic communications.

XML was developed in 1998 and is now widely used. It is one of the most flexible ways to automate web transactions. XML is derived as a subset from Standard Generalized Markup Language (SGML) and is designed to be simple, concise, human readable, and relatively easy to use in programs on multiple platforms. For more information about the XML standard, see the following web page:

http://www.w3.org/XML

As with other markup languages, XML is built using tags. Basic XML consists of start tags, end tags, and a data value between the two. In XML, you create your own tags, with a few restrictions. Example 2-1 shows a simple XML document.

Example 2-1 Simple XML document

```
<?xml version="1.0" encoding="UTF-8" ?>
<Employee>
<ID>0000002150</ID>
<Lastname>SMITH</Lastname>
<Firstname>ROBERT</Firstname>
<Company>IBM</Company>
<Address>Bytoften 1</Address>
<Zipcode>8240</Zipcode>
<City>Risskov</City>
<Country>Denmark</Country>
</Employee>
```

The XML syntax is simple, but it is difficult to parse and transform an XML document into a form that is usable to programming languages. Therefore, it is essential to have access to efficient parsing and transformation tools.

XML contains document type and schema definitions. These document type and schema definitions are used to specify semantics (allowable grammar) for an XML document.

2.2 The value of XML data

As a result of XML industry standards becoming more prevalent, the drive toward service-oriented architecture (SOA) environments, and rapid adoption of syndication technologies, more XML data is generated every day. XML is in web feeds, purchase orders, transaction records, messages in SOA environments, financial trades, insurance applications, and other industry-specific and cross-industry data. That is, XML data and documents are becoming important business assets that contain valuable information, such as customer details, transaction data, web logs, and operational documents.

The growth and pervasiveness of XML assets present challenges and opportunities for the enterprise. When XML data is harnessed, and the value of the information it contains is unlocked, it can translate into opportunities for organizations to streamline operations, derive insight, and become agile.

Alternatively, as XML data becomes more critical to the operations of an enterprise, it presents challenges in that XML data must be secured, maintained, searched, and shared. Depending on its use, XML data might also need to be updated, audited, and integrated with traditional data. All of these tasks must be performed with the reliability, availability, and scalability afforded to traditional data assets.

Unleashing the potential of XML data requires a robust information integration platform. In the next section, we provide several scenarios that drive the use of XML technologies.

SOA and Web services

Services-based frameworks and deployments can more easily be used to integrate systems, reuse resources, and enable fast reactions to changing market conditions. In services-based architectures, consumers and service providers exchange information using messages. These messages are invariably encapsulated as XML, which can provide the foundation in SOA environments, as depicted in Figure 2-1. Thus, the drive toward information as a service and the adoption of SOA environments is also stimulating the growth of XML.

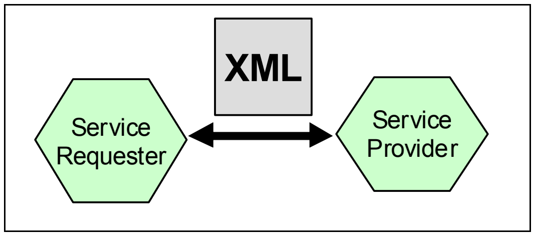


Figure 2-1 XML: The foundation for Web services

Web 2.0 technologies

Syndication is considered to be the heartbeat of Web 2.0, the next generation of the Internet. Atom and RSS feeds exist in abundance on the web, allowing you to subscribe to them and to be kept up-to-date about all kinds of web content changes, such as news stories, articles, wikis, audio files, and video files. Content for these feeds is rendered as XML files and can contain links, summaries, full articles, and even attached multimedia files, such as podcasts. Syndication and web feeds are transforming the web as we know it. New business models are emerging around these technologies. As a consequence, XML data now exists not only in companies adopting XML industry standards, or enterprises implementing SOAs, but also on virtually every web-connected desktop.

2.3 Introduction to XML documents

XML is reminiscent of HTML, because they are both derived from SGML, which was defined in 1986. But unlike HTML, XML tags identify the data, rather than specifying how to display it. For example, an HTML tag might display this data in bold font (`<b>...</b>`). An XML tag acts like a field name in your program. It puts a label on a piece of data to identify it (`<message>...</message>`).

XML documents can be well formed, or they can be well formed and valid. These important rules do not exist for HTML documents. These rules contrast with the freestyle nature of many of the concepts in XML. The rules can be defined briefly as invalid, valid, and well-formed documents.

There are three kinds of XML documents:

- *Invalid documents* do not follow the syntax rules defined by the XML specification. If a developer defined rules for the document content in a document type definition (DTD) or schema, and the document does not follow those rules, that document is invalid as well. (See 2.4, “Defining document content” on page 32, for a proper introduction to DTDs and schemas for XML documents.)
- *Valid documents* follow both the XML syntax rules and the rules defined in their DTD or schema.
- *Well-formed documents* follow the XML syntax rules but do not have a DTD or schema.

2.3.1 Well-formed XML documents

To be well-formed documents, the documents must conform to the basic rules for XML documents. In Table 2-1, we provide a list of a subset of the requirements for well-formed XML documents.

Table 2-1 Requirements for a well-formed document

Requirement	Well-formed example	Not well-formed example
The document has exactly one root element.	<element0> <elem1>value1</elem1> <elem2>value2</elem2> </element0>	<element1>value1</element1> <element2>value2</element2>
Each start tag is matched by one end tag.	<elem1> <elem2>value2</elem2> </elem1>	<elem1> <elem2>value2 </elem1>
All elements are properly nested.	<elem1> <elem2>value2</elem2> </elem1>	<elem1> <elem2>value2</elem1> </elem2>
Attribute values are quoted.	<elem1 id=”15”>value1</elem1>	<elem1 id=15>value1</elem1>
XML elements are case-sensitive.	<elem1> <elem2>value2</elem2> </elem1>	<ELEM1> <elem2>value2</ELEM2> </elem1>
Disallowed characters are not used in tags or values.	<elem1> 123&lt;456</elem1>	<elem1> 123<456</elem1>

Validation

The process of checking to see whether an XML document conforms to a schema or DTD is called *validation*. Validation is in addition to checking a document for compliance to the XML core concept of *syntactic well-formedness*. All XML documents must be well formed, but it is not required that a document is valid unless the XML parser is validating. When validating, the document is also checked for conformance with its associated schema.

Documents are only considered valid if they satisfy the requirements of the DTD or schema with which they are associated. These requirements typically include the following types of constraints:

- Elements and attributes that must or might be included, and their permitted structure.

- The structure is specified by a regular expression syntax.

InfoSphere Information Server DataStage provides robust XML parsing and validation capabilities.

XML declarations

Most XML documents start with an XML declaration that provides basic information about the document to the parser. An XML declaration is recommended, but not required. If there is an XML declaration, it must be the first item in the document.

The declaration can contain up to three name-value pairs (many people call them *attributes*, although technically they are not). The version is the version of XML used; currently this value must be 1.0. The encoding is the character set used in this document. The ISO-8859-1 character set referenced in this declaration includes all of the characters used by most Western European languages. If no encoding is specified, the XML parser assumes that the characters are in the UTF-8 set, a Unicode standard that supports virtually every character and ideograph in all languages. The following declaration is an example:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
```

Finally, `standalone`, which can be either yes or no, defines whether this document can be processed without reading any other files. For example, if the XML document does not reference any other files, you specify `standalone="yes"`. If the XML document references other files that describe what the document can contain (more about those files later), you specify `standalone="no"`. Because `standalone="no"` is the default, you rarely see `standalone` in XML declarations.

Namespaces

The power of XML comes from its flexibility. You can define your own tags to describe your data. Consider an XML document with the tag <Title>. The document includes the <title> element for a courtesy title, the <title> element for the title of a book, and the <title> element for the title to a piece of property. All the <Title> information elements mentioned in this case are reasonable choices, but all of them create elements with the same name. How do you tell if a specific <title> element refers to a person, a book, or a piece of property? These situations are handled with namespaces.

To use a namespace, define a namespace prefix and map it to a particular string. In Example 2-2, we show how you might define namespace prefixes for the three <title> elements.

Example 2-2 Define namespace prefixes

```
<?xml version="1.0"?>
```

```
<customer_summary
  xmlns:addr="http://www.xyz.com/addresses/"
  xmlns:books="http://www.zyx.com/books/"
  xmlns:mortgage="http://www.yyz.com/title/"
>
... <addr:name><title>Mrs.</title> ... </addr:name> ...
... <books:title>Lord of the Rings</books:title> ...
... <mortgage:title>NC2948-388-1983</mortgage:title> ...
```

In Example 2-2 on page 30, the three namespace prefixes are `addr`, `books`, and `mortgage`. Defining a namespace for a particular element means that all of its child elements belong to the same namespace. The first <title> element belongs to the addr namespace because its parent element, <addr:Name>, belongs to the addr namespace.

Additional components

Comments, processing instructions, and entities are additional components of XML. The following sections provide additional details about these components.

Comments

Comments can appear anywhere in the document; they can even appear before or after the root element. A comment begins with <!-- and ends with -->. A comment cannot contain a double hyphen (--) except at the end; with that exception, a comment can contain anything. Most importantly, any markup inside a comment is ignored; if you want to remove a large section of an XML document, simply wrap that section in a comment. (To restore the commented-out section, simply remove the comment tags.) This example markup contains a comment:

```
<!-- This is comment section of Customer Name XML Doc : -->
<Customer Name>
<First Name> John </First Name>
<Last Name> Smith </Last Name>
</Customer Name>
```

Processing instructions

A *processing instruction* is markup intended for a particular piece of code. In the following example, a processing instruction (PI) exists for Cocoon, which is an XML processing framework from the Apache Software Foundation. When Cocoon processes an XML document, it looks for processing instructions that begin with cocoon-process, then processes the XML document. In this example, the type="`sql`" attribute tells Cocoon that the XML document contains an SQL statement:

```
<!-- Here is a PI for Cocoon: -->
```

```
<?cocoon-process type="sql"?>
```

Entities

The following example defines an *entity* for the document. Anywhere the XML processor finds the string `&RB;`, it replaces the entity with the string ‘`RedBooks`’:

```
<!-- Here is an entity: -->
<!ENTITY RB "RedBooks">
```

The XML spec also defines five entities that can be used in place of various special characters:

- `&lt;` for the less-than sign
- `&gt;` for the greater-than sign
- `&quot;` for a double-quote
- `&apos;` for a single quote (or apostrophe)
- `&amp;` for an ampersand

2.4 Defining document content

Document type definitions and XML schema are both used to describe structured information. XML schema is gaining acceptance. Both document type definitions and schemas are building blocks for XML documents and consist of elements, tags, attributes, and entities.

2.4.1 Document type definition

A document type definition (DTD) specifies the kinds of tags that can be included in your XML document, the valid arrangements of those tags, and the structure of the XML document. The DTD defines the type of elements, attributes, and entities allowed in the documents, and it can also specify limitations to their arrangement. You can use the DTD to ensure that you do not create an invalid XML structure. The DTD defines how elements relate to one another within the document tree structure. You can also use it to define which attributes can be used to define an element and which attributes are not allowed.

A DTD defines your own language for a specific application. The DTD can be either stored in a separate file or embedded within the same XML file. If it is stored in a separate file, it can be shared with other documents. XML documents referencing a DTD contain a <!DOCTYPE> declaration. This <!DOCTYPE> declaration either contains the entire DTD declaration in an internal DTD or specifies the location of an external DTD. Example 2-3 on page 33 shows an external DTD in a file named `DTD-Agenda.dtd`.

Example 2-3   An external DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT AGENDA (PERSONA+)>
<!ELEMENT PERSONA (EMPRESA, CARGO, NOMBRE, TELEFONO1+,TELEFONO2*,
EXT?)>
<!ELEMENT EMPRESA (#PCDATA)>
<!ELEMENT CARGO (#PCDATA)>
<!ELEMENT NOMBRE (#PCDATA)>
<!ELEMENT TELEFONO1 (#PCDATA)>
<!ELEMENT TELEFONO2 (#PCDATA)>
<!ELEMENT EXT (#PCDATA)>
```

Example 2-4 is an XML document that refers to this external DTD.

Example 2-4   Reference to an external DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE AGENDA SYSTEM "DTD_Agenda.dtd">
<AGENDA>
<PERSONA>
<EMPRESA>Matutano</EMPRESA>
<CARGO>Gerente</CARGO>
<NOMBRE>Pepe Montilla</NOMBRE>
<TELEFONO1>912563652</TELEFONO1>
<TELEFONO2>658968574</TELEFONO2>
<EXT>256</EXT>
</PERSONA>
</AGENDA>
```

We can also define an internal DTD in an XML document so that both of them are in the same file, as shown in Example 2-5. In either case, internal or external DTD, the <!DOCTYPE> declaration indicates the root element.

Example 2-5   An internal DTD

```
<!DOCTYPE RAIZ [
<!ELEMENT ...... >
<!ELEMENT ...... >
<!ELEMENT ...... >
<!ELEMENT ...... >
]>
<RAIZ>
.
..
```

```
</RAIZ>
```

A DTD is not required in an XML document. However, a DTD provides parsers with clear instructions about what to check when they are determining the validity of an XML document. DTDs or other mechanisms, such as XML schemas, contribute to the goal of ensuring that the application can easily determine whether the XML document adheres to a specific set of rules, beyond the well-formedness rules that are defined in the XML standard. DTDs are also used by tools to create XML documents.

Symbols in DTDs

A few symbols are used in DTDs to indicate how often (or whether) something can appear in an XML document. We show several examples, along with their meanings:

- <!ELEMENT address (name, city, state)>

  The <address> element must contain a <name>, a <city>, and a <state> element, in that order. All of the elements are required. The comma indicates a list of items.

- <!ELEMENT name (title?, first-name, last-name)>

  This means that the <name> element contains an optional <title> element, followed by a mandatory <first-name> and a <last-name> element. The question mark indicates that an item is optional; it can appear one time or not at all.

- <!ELEMENT addressbook (address+)>

  An <addressbook> element contains one or more <address> elements. You can have as many <address> elements as you need, but there must be at least one. The plus sign (+) indicates that an item must appear at least one time, but it can appear any number of times.

- <!ELEMENT private-addresses (address*)>

  A <private-addresses> element contains zero or more <address> elements. The asterisk indicates that an item can appear any number of times, including zero.

- <!ELEMENT name (title?, first-name, (middle-initial | middle-name)?, last-name)>

  A <name> element contains an optional <title> element, followed by a <first-name> element, possibly followed by either a <middle-initial> or a <middle-name> element, followed by a <last-name> element. Both <middle-initial> and <middle-name> are optional, and you can have only one of the two elements. Vertical bars indicate a list of choices; you can choose only one item from the list. This example uses parentheses to group certain elements, and it uses a question mark against the group.

- <!ELEMENT name ((title?, first-name, last-name) | (surname, mothers-name, given-name))>

  The <name> element can contain one of two sequences: An optional <title>, followed by a <first-name> and a <last-name>; or a <surname>, a <mothers-name>, and a <given-name>.

2.4.2 XML schema

The World Wide Web Consortium (W3C) *XML Schema Definition Language* (XSDL) is an XML language for describing and constraining the content of XML documents. A *schema* is similar to a DTD in that it defines which elements an XML document can contain, how they are organized, and which attributes and attribute types elements can be assigned. The Information Server Schema Manager does not support DTDs, but the Information Server supports XML schemas. Therefore, a schema is a method to check the validity of well-formed XML documents. The following list shows the major advantages of schemas over DTDs:

- XML schemas use XML syntax.

  An XML schema is an XML document. You can process a schema just like any other document. For example, you can write an XSLT style sheet that converts an XML schema into a web form complete with automatically generated JavaScript code that validates the data as you enter it.

- XML schemas support data types.

  DTDs support data types; however, it is clear those data types were developed from a publishing perspective. XML schemas support all of the original data types from DTDs, such as IDs and ID references. They also support integers, floating point numbers, dates, times, strings, URLs, and other data types that are useful for data processing and validation.

- XML schemas are extensible.

  In addition to the data types defined in the XML schema specification, you can also create your own, and you can derive new data types based on other data types.

- XML schemas have more expressive power.

  For example, with XML schemas, you can define that the value of any <state> attribute cannot be longer than two characters, or that the value of any <postal-code> element must match the regular expression [0-9]{5}(-[0-9]{4})?. You cannot perform either of these functions with DTDs.

Sample XML schema

It is difficult to give a general outline of the elements of a schema due to the number of elements that can be used according to the W3C XML Schema Definition Language. The purpose of this language is to provide an inventory of XML markup constructs with which to write schemas. Example 2-6 is a simple document that describes the information about a book.

Example 2-6 A book description

```
<?xml version="1.0" encoding="UTF-8"?>
<book isbn="0836217462">
<title>
Don Quijote de la Mancha
</title>
<author>De Cervantes Saavedra, Miguel</author>
<character>
<name>Sancho Panza</name>
<friend-of>El Quijote</friend-of>
<since>1547-10-04</since>
<qualification> escudero </qualification>
</character>
<character>
<name>ElbaBeuno</name>
<since>1547-08-22</since>
<qualification>Amor Platonico de Don Quijote</qualification>
</character>
</book>
```

Because the XML schema is a language, several choices exist to build a possible schema that covers the XML document. Example 2-7 is a possible, simple design.

Example 2-7 XML schema

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="book">
<xs:complexType>
<xs:sequence>
<xs:element name="title" type="xs:string"/>
<xs:element name="author" type="xs:string"/>
<xs:element name="character" minOccurs="0" maxOccurs="unbounded">
<xs:complexType>
<xs:sequence>
<xs:element name="name" type="xs:string"/>
<xs:element name="friend-of" type="xs:string" minOccurs="0"
```

```
maxOccurs="unbounded"/>
<xs:element name="since" type="xs:date"/>
<xs:element name="qualification" type="xs:string"/>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
<xs:attribute name="isbn" type="xs:string"/>
</xs:complexType>
</xs:element>
</xs:schema>
```

Example 2-7 on page 36 is obviously an XML document, because it begins with the XML document declaration. The schema element opens our schema holding the definition of the target namespace. Then, we define an element named book. This element is the root element in the XML document. We decided that it is a complex type, because it has attributes and non-text children. With sequence, we begin to declare the children elements of the root element book. W3C XML schema lets us define the type of data, as well as the number of possible occurrences of an element. To learn about the possible values for these types, see the specification documents from W3C.

XML schema options

The XML schema language offers possibilities and alternatives beyond what is shown in Example 2-7 on page 36. We can develop another schema based on a flat catalog of all the elements available in the instance document and, for each of the elements, lists of child elements and attributes. Thus, we have the following two choices:

- Defining elements and attributes as they are needed
- Creating the elements first and referencing them

The first option has a real disadvantage: the schema might be difficult to read and maintain when documents are complex. W3C XML schema allows us to define data types and use these types to define our attributes and elements. It also allows the definition of groups of elements and attributes. In addition, there are several ways to arrange relationships between the elements.

2.4.3 Namespaces

Namespaces are used when you need separate elements, possibly with separate attributes, but with the same name. Depending upon the context, a tag is related to one specific element or to another specific element. Without namespaces, it is impossible to specify the context to which an element belongs. Example 2-8 shows this situation.

Example 2-8 XML document with duplicated element names

```
<?xml version="1.0" ?>
<book>
<title>XML Sample</title>
<pages>210</pages>
<isbn>1-868640-34-2</isbn>
<author>
<firstname>JuanJose</firstname>
<lastname>Hernandez</lastname>
<title>Mr</title>
</author>
</book>
```

Clearly, a problem exists with the element <title>. It appears in two separate contexts. This situation complicates the situation for processors and might cause ambiguities. We need a mechanism to distinguish between the two contexts and apply the correct semantic description to each tag. The cause of this problem is that this document uses only one common namespace. The solution to this problem is namespaces. Namespaces are a simple and straightforward way to distinguish names that are used in XML documents. By providing the related namespace when an element is validated, the problem is solved, as shown in Example 2-9.

Example 2-9 XML document using namespace prefixes

```
<?xml version="1.0" ?>
<library-entry
xmlns:authr="http://sc58ts.itso.ibm.com/Jose/2002/author.dtd"
xmlns:bk="books.dtd">
<bk:book>
<bk:title>XML Sample</bk:title>
<bk:pages>210</bk:pages>
<bk:isbn>1-868640-34-2</bk:isbn>
<authr:author>
<authr:firstname>JuanJose</authr:firstname>
<authr:lastname>Hernandez</authr:lastname>
<authr:title>Mr</authr:title>
</authr:author>
</bk:book>
```

As you can see in Example 1-8, the <title> tag is used twice, but in separate contexts: within the <author> element and within the <book> element. Note the use of the `xmlns` keyword in the namespace declaration. After a prefix is defined on an element, it can then be used by all descendants of that element. In Example 2-10 on page 41, we specified the relevant namespace prefix before each element to illustrate the relationship of each element to a specific namespace. However, if the prefix is not specified, the element is in the default namespace if a default namespace is specified or not in a namespace if there is no default namespace. A default namespace is defined on an element with the *xmlns* attribute without specifying a prefix.

2.5 Extensible Stylesheet Language

Up to this point, we described XML, its syntax, how XML is used to mark up information according to our own vocabularies, and how a program can check the validity of an XML document. We described how XML can ensure that an application running on any particular platform receives valid data. XML helps you ensure that a program can process this data. However, because XML describes document syntax only, the program does not know how to format this data without specific instructions about style.

The solution is to use XSL transformations. The Extensible Stylesheet Language (XSL) specification describes powerful tools to accomplish the required transformation of XML data:

- The XSL Transformations (XSLT) language is for transformation.
- Formatting Objects (FO) is a vocabulary for describing the layout of documents.
- XSLT uses the XML Path Language (XPath), which is a separate specification that describes a means of addressing XML documents and defining simple queries.

XSLT offers a powerful means of transforming XML documents into other forms, producing XML, HTML, and other formats. It can sort, select, and number. And, it offers many other features for transforming XML. It operates by reading a style sheet, which consists of one or more templates, and then matching the templates as it visits the nodes of the XML document. The templates can be based on names and patterns. In the context of information integration, XSLT is increasingly used to transform XML data into another form, sometimes other XML (for example, filtering out certain data, SQL statements, and plain text), or any other format. Thus, any XML document can be shown in separate formats, such as HTML, PDF, RTF, VRML, and Postscript.

2.5.1 XPath

XPath is a string syntax for building addresses to the information that is found in an XML document. We use this language to specify the locations of document structures or data found in an XML document when processing that information using XSLT. XPath allows us from any location to address any other location or content. XPath is a tool that is used in XSLT to select certain information to be formatted.

XPath patterns

Table 2-2 shows several XPath patterns. These examples show you a few objects that you can select.

Table 2-2 XPath pattern

Symbol	Meaning
/	Refer to the immediate child
//	Refer to any child in the node
.	Refer to actual context
*	Refer to all elements in the actual node
@	Refer to an attribute
@*	Refer to all attributes in the actual node

XPath models an XML document as a tree of nodes, as shown in the following list:

- Root nodes
- Element nodes
- Attribute nodes
- Text nodes
- Namespace nodes
- Processing instruction nodes
- Comment nodes

The basic syntactic construct in XPath is the expression, as shown in Example 2-10 on page 41. An object is obtained by evaluating an expression, which has one of the following four basic types:

- Node-set (an unordered collection of nodes without duplicates)
- Boolean
- Number
- String

Example 2-10 Xpath

```
<?xml version="1.0"?>
<!DOCTYPE library system "library.dtd">
<library>
   <book ID="B1.1">
      <title>xml</title>
      <copies>5</copies>
   </book>
   <book ID="B2.1">
      <title>WebSphere</title>
      <copies>10</copies>
   </book>
   <book ID="B3.2">
      <title>great novel</title>
      <copies>10</copies>
   </book>
   <book ID="B5.5">
      <title>good story</title>
      <copies>10</copies>
   </book>
</library>
```

Considering Example 2-10, we can make the following paths:

- /book/copies selects all copies of element children of book elements.
- /book//title selects all title elements in the tree, although title elements are not immediate children.

The /book/@ID path selects all ID attributes beyond book elements.

It is also possible to select elements based on other criteria. The /library/*/book[title eq "good story"] path selects all book elements beyond the library element, but only if the title element matches “good story”.

2.5.2 XSLT

XSLT helps you access and display the content in the XML file. XSLT is referred to as the stylesheet language of XML. The relationship of Cascading Style Sheets (CSS) and HTML is comparable to the relationship of XSLT and XML. However, XML and XSLT are far more sophisticated technologies than HTML and CSS.

XSLT is a high-level declarative language. It is also a transforming and formatting language. It behaves in the following way:

- The pertinent data is extracted from an XML source document and transformed into a new data structure that reflects the desired output. The XSLT markup is commonly called a *stylesheet*. A parser is used to convert the XML document into a tree structure that is composed of various types of nodes. The transformation is accomplished with XSLT by using pattern matching and templates. Patterns are matched against the source tree structure, and templates are used to create a result tree.
- Next, the new data structure is formatted, for example, in HTML or as text, and finally the data is ready for display.

Figure 2-2 shows the source tree from the XML document that was presented in Example 2-10 on page 41.

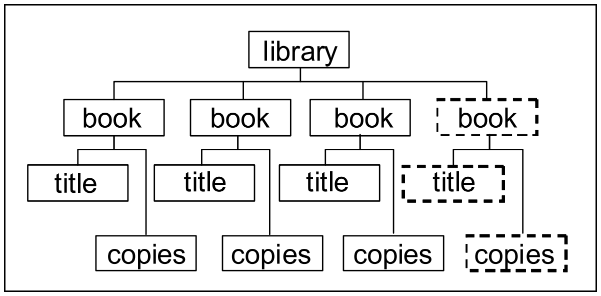


Figure 2-2 Source tree

The resultant tree after an XSL transformation might be an XHTML document, as shown in Figure 2-3.

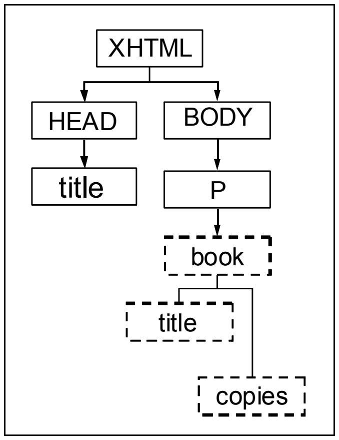


Figure 2-3 Source tree after an XML transformation

Based on how we instruct the XSLT processor to access the source of the data that is transformed, the processor incrementally builds the result by adding the filled-in templates. We write our stylesheets, or "transformation specifications," primarily with declarative constructs, although we can employ procedural techniques if and when needed. We assert the desired behavior of the XSLT processor based on conditions found in our source. XSLT manipulates the source tree only, and the original XML document is left unchanged.

The most important aspect of XSLT is that it allows you to perform complex manipulations on the selected tree nodes by affecting both content and appearance. The final output might not resemble the source document. XSLT far surpasses CSS in this capability to manipulate the nodes.

2.6 Web services

Web services offer a new dimension to information integration. Consider Web services as a gateway to service-oriented architecture (SOA) for the data integration needs of an enterprise. You can use Web services to extract and integrate business data from heterogeneous business systems. By providing interface standards, Web services can be an important component of information as a service architecture.

Web services allow applications to communicate with each other in a platform-independent and programming language-independent manner. A *web service* is a software interface that describes a collection of operations that can be accessed over the network through standardized XML messaging. A web service uses protocols based on the XML language to describe an operation to execute or data to exchange with another web service. A group of Web services interacting together in this manner defines a particular web service application in an SOA.

Web services help to bridge the gap between business users and technologists in an organization. Web services make it easier for business users to understand web operations. Business users can then describe events and activities, and technologists can associate them with appropriate services.

With universally defined interfaces and well-designed tasks, it also is easier to reuse these tasks and the applications that they represent. Reusability of application software means a better return on investment on software, because it can produce more from the same resources. Reusability of application software allows business people to consider using an existing application in a new way or offering it to a partner in a new way, thus potentially increasing the business transactions between partners.

Therefore, Web services address the issues of data and application integration and of transforming technical functions into business-oriented computing tasks. These two facets allow businesses to communicate on a process level or application level with their partners, while leaving dynamic room to adapt to new situations or work with different partners on demand.

The versatility of XML is what makes Web services different from previous generation component technologies. XML allows the separation of grammatical structure (syntax) and the grammatical meaning (semantics), and how that is processed and understood by each service and the environment in which it exists. Now, objects can be defined as services, communicating with other services in XML-defined grammar, whereby each service then translates and analyzes the message according to its local implementation and environment. Thus, a networked application can be composed of multiple entities of various makes and designs as long as they conform to the rules defined by their SOA.

Web services allow you to perform the following tasks:

- Interact between services on any platform, written in any language.
- Conceptualize application functions into tasks, leading to task-oriented development and workflows. This conceptualization allows a higher abstraction of software that can be employed by less software-technical users that work on business-level analytics.
- Allow for loose-coupling, which means that interactions between service applications might not break each time there is a change in how services are designed or implemented.
- Adapt existing applications to changing business conditions and customer needs.
- Provide existing software applications with service interfaces without changing the original applications, allowing them to fully operate in the service environment.
- Introduce other administrative or operations management functions, such as reliability, accountability, and security, independently of the original function, thus increasing its versatility and usefulness in the business computing environment.

3

Overview of IBM InfoSphere DataStage

You can take advantage of the capabilities of InfoSphere DataStage when you understand the underlying architecture. In this chapter, we describe InfoSphere Information Server (IS) architecture, InfoSphere DataStage, and the corresponding XML transformation architecture. Additionally, we explain the DataStage elements that are used in integration scenarios from an architectural perspective.

3.1 Introduction to IBM InfoSphere Information Server

Over the years, most organizations made significant investments in enterprise resource planning, customer relationship management, and supply chain management packages in addition to their home-grown applications. These investments resulted in larger amounts of data captured about their businesses. To turn all this data into consistent, timely, and accurate information for decision-making requires an effective means of integrating information.

IBM InfoSphere Information Server is the first comprehensive, unified foundation for enterprise information architectures, capable of scaling to meet any information volume requirement so that companies can deliver business results within these initiatives faster and with higher quality results. InfoSphere Information Server provides a single unified platform that enables companies to understand, cleanse, transform, and deliver trustworthy and context-rich information.

InfoSphere Information Server helps you derive more value from complex, heterogeneous information. It helps business and IT personnel collaborate to understand the meaning, structure, and content of information across a wide variety of sources. IBM InfoSphere Information Server helps you access and use information in new ways to drive innovation, increase operational efficiency, and lower risk.

Most critical business initiatives cannot succeed without the effective integration of information. Initiatives, such as a single view of the customer, business intelligence, supply chain management, and Basel II and Sarbanes-Oxley compliance, require consistent, complete, and trustworthy information.

InfoSphere Information Server supports all of these initiatives:

- Business intelligence: InfoSphere Information Server makes it easier develop a unified view of the business for better decisions. It helps you understand existing data sources, cleanse, correct, and standardize information, and load analytical views that can be reused throughout the enterprise.
- Master data management: InfoSphere Information Server simplifies the development of authoritative master data by showing where and how information is stored across source systems. It also consolidates disparate data into a single, reliable record, cleanses and standardizes information, removes duplicates, and links records across systems. This master record can be loaded into operational data stores, data warehouses, or master data applications, such as IBM WebSphere Customer Center. The record can also be assembled, completely or partially, on demand.

- Infrastructure rationalization: InfoSphere Information Server aids in reducing operating costs by showing relationships among systems and by defining migration rules to consolidate instances or move data from obsolete systems. Data cleansing and matching ensure high-quality data in the new system.
- Business transformation: InfoSphere Information Server can speed development and increase business agility by providing reusable information services that can be plugged into applications, business processes, and portals. These standards-based information services are maintained centrally by information specialists but are widely accessible throughout the enterprise.
- Risk and compliance: InfoSphere Information Server helps improve visibility and data governance by enabling complete, authoritative views of information with proof of lineage and quality. These views can be made widely available and reusable as shared services, while the rules inherent in them are maintained centrally.

3.1.1 Capabilities

InfoSphere Information Server features a unified set of separately orderable product modules, or suite components, that solve multiple types of business problems. Information validation, access, and processing rules can be reused across projects, leading to a higher degree of consistency, stronger control over data, and improved efficiency in IT projects. This single unified platform enables companies to understand, cleanse, transform, and deliver trustworthy and context-rich information, as shown in Figure 3-1 on page 48.

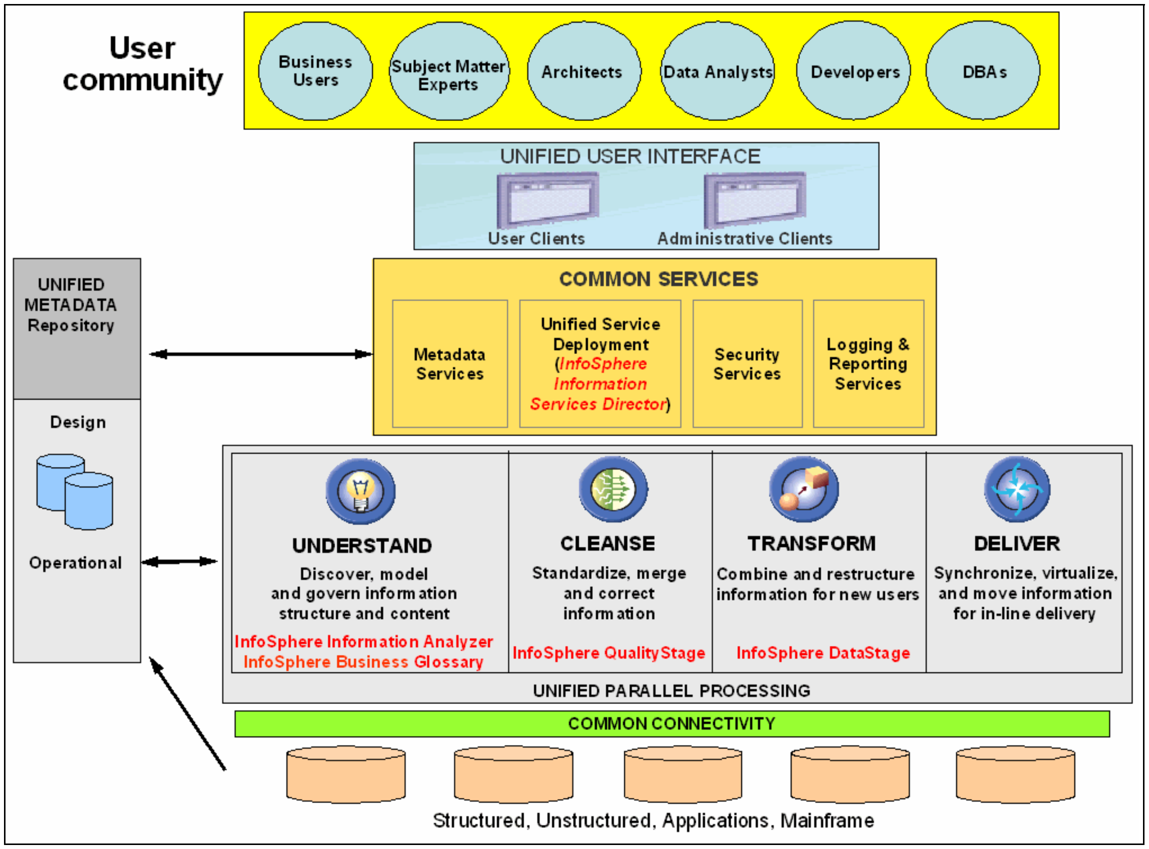

User community
Business Users
Subject Matter Experts
Architects
Data Analysts
Developers
DBAs
UNIFIED USER INTERFACE
User Clients
Administrative Clients
UNIFIED METADATA Repository
Design
Operational
COMMON SERVICES
Metadata Services
Unified Service Deployment (InfoSphere Information Services Director)
Security Services
Logging & Reporting Services
UNDERSTAND
Discover, model and govern information structure and content
InfoSphere Information Analyzer
InfoSphere Business Glossary
CLEANSE
Standardize, merge and correct information
InfoSphere QualityStage
TRANSFORM
Combine and restructure information for new users
InfoSphere DataStage
DELIVER
Synchronize, virtualize, and move information for in-line delivery
UNIFIED PARALLEL PROCESSING
COMMON CONNECTIVITY
Structured, Unstructured, Applications, Mainframe

Figure 3-1 InfoSphere Information Server Architecture

Understand the data

InfoSphere Information Server can help you automatically discover, define, and model information content and structure, and understand and analyze the meaning, relationships, and lineage of information. By automating data profiling and data-quality auditing within systems, organizations can achieve these goals:

- Understand data sources and relationships
- Help eliminate the risk of using or proliferating bad data
- Improve productivity through automation
- Use existing IT investments

InfoSphere Information Server makes it easier for businesses to collaborate across roles. Data analysts can use analysis and reporting functionality, generating integration specifications and business rules that they can monitor over time. Subject matter experts can use web-based tools to define, annotate, and report on fields of business data. A common metadata foundation enables different types of users to create and manage metadata by using tools that are optimized for their roles.

Cleanse the information

InfoSphere Information Server supports information quality and consistency by standardizing, validating, matching, and merging data. It can certify and enrich common data elements, use trusted data such as postal records for name and address information, and match records across or within data sources. InfoSphere Information Server allows a single record to survive from the best information across sources for each unique entity, helping you to create a single, comprehensive, and accurate view of information across source systems.

Transform the data into information

InfoSphere Information Server transforms and enriches information to ensure that it is in the proper context for new uses. Hundreds of prebuilt transformation functions combine, restructure, and aggregate information.

Transformation functionality is broad and flexible, to meet the requirements of varied integration scenarios. For example, InfoSphere Information Server provides inline validation and transformation of complex data types, such as the US Health Insurance Portability and Accountability Act (HIPAA), and high-speed joins and sorts of heterogeneous data. InfoSphere Information Server also provides high-volume, complex data transformation and movement functionality that can be used for stand-alone extract-transform-load (ETL) scenarios, or as a real-time data-processing engine for applications or processes.

Deliver the information

InfoSphere Information Server virtualize, synchronize, or move information to the people, processes, or applications that need it. Information can be delivered by using federation or time-based or event-based processing, moved in large bulk volumes from location to location, or accessed in place when it cannot be consolidated.

InfoSphere Information Server provides direct, native access to various mainframe and distributed information sources. It provides access to databases, files, services, and packaged applications, and to content repositories and collaboration systems. Companion products allow high-speed replication, synchronization and distribution across databases, change data capture, and event-based publishing of information.

3.2 IBM InfoSphere Information Server Architecture

IBM Information Server combines the technologies of key information integration functions within the IBM Information Platform & Solutions portfolio into a single unified platform that enables companies to understand, cleanse, transform, and deliver trustworthy and context-rich information, as shown in Figure 3-1 on page 48.

IBM InfoSphere Information Server provides a unified architecture that works with all types of information integration. Common services, unified parallel processing, and unified metadata are at the core of the server architecture.

The architecture is service-oriented, enabling IBM InfoSphere Information Server to work within evolving enterprise service-oriented architectures (SOAs). An SOA also connects the individual suite product modules of InfoSphere Information Server.

By eliminating the duplication of functions, the architecture efficiently uses hardware resources and reduces the amount of development and administrative effort that is required to deploy an integration solution.

IBM Information Server is a client-server architecture made up of client-based design, administration, and operation tools that access a set of server-based data integration capabilities through a common services layer, as shown in Figure 3-2 on page 51. Figure 3-2 on page 51 is a slightly different and more detailed view of the same information that was shown in Figure 3-1 on page 48. Additionally, in Figure 3-2 on page 51, notice the Repository and Engine tiers. In the following sections, we discuss these tiers.

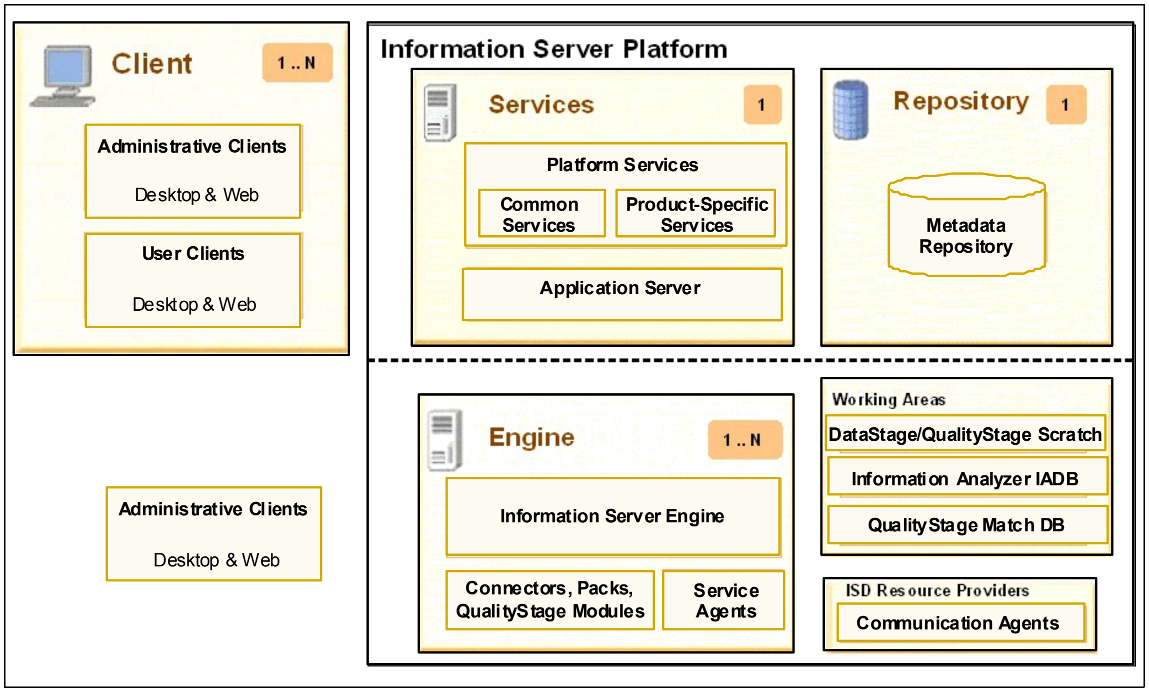


Figure 3-2   Information Server technical components

Unified parallel processing engine

Much of the work that InfoSphere Information Server performs takes place within the parallel processing engine. The engine handles data processing needs as diverse as performing the analysis of large databases for IBM InfoSphere Information Analyzer, data cleansing for IBM InfoSphere QualityStage, and complex transformations for IBM InfoSphere DataStage. This parallel processing engine is designed to deliver the following benefits:

- Parallelism and data pipelining to complete increasing volumes of work in decreasing time windows
- Scalability by adding hardware (for example, processors or nodes in a grid) with no changes to the data integration design
- Optimized database, file, and queue processing to handle large files that cannot fit in memory all at once or large numbers of small files

Common connectivity

InfoSphere Information Server connects to information sources whether they are structured, unstructured, on the mainframe, or applications. Metadata-driven connectivity is shared across the suite components, and connection objects are reusable across functions.

Connectors provide the design-time importing of metadata, data browsing and sampling, runtime dynamic metadata access, error handling, and high functionality and high performance runtime data access. Prebuilt interfaces for packaged applications called *packs* provide adapters to SAP, Siebel, Oracle, and other software, enabling integration with enterprise applications and associated reporting and analytical systems.

Unified metadata

InfoSphere Information Server is built on a unified metadata infrastructure that enables shared understanding between business and technical domains. This infrastructure reduces development time and provides a persistent record that can improve confidence in information. All functions of InfoSphere Information Server share the metamodel, making it easier for different roles and functions to collaborate.

A common metadata repository provides persistent storage for all InfoSphere Information Server suite components. All of the products depend on the repository to navigate, query, and update metadata. The repository contains two kinds of metadata:

- Dynamic metadata includes design-time information.
- Operational metadata includes performance monitoring, audit and log data, and data-profiling sample data.

Because the repository is shared by all suite components, profiling information that is created by InfoSphere Information Analyzer is instantly available to users of InfoSphere DataStage and InfoSphere QualityStage.

The repository is a Java 2 Platform, Enterprise Edition (J2EE) application that uses a standard relational database, such as IBM DB2, Oracle, or SQL Server for persistence. (DB2 is provided with InfoSphere Information Server.) These databases provide backup, administration, scalability, parallel access, transactions, and concurrent access.

Common services

InfoSphere Information Server is built entirely on a set of shared services that centralizes core tasks across the platform. These tasks include administrative tasks, such as security, user administration, logging, and reporting. Shared services allow these tasks to be managed and controlled in one place, regardless of which suite component is used. The common services also include the metadata services, which provide standard service-oriented access and analysis of metadata across the platform. In addition, the common Services tier manages how services are deployed from any of the product functions. The common Services tier allows the cleansing and transformation rules or federated queries to be published as shared services within an SOA with a consistent and easy-to-use mechanism.

InfoSphere Information Server products can access three general categories of service:

- Design services help developers create function-specific services that can also be shared. For example, InfoSphere Information Analyzer calls a column analyzer service that was created for enterprise data analysis. This column analyzer service can be integrated with other parts of InfoSphere Information Server, because it exhibits common SOA characteristics.
- Execution services include logging, scheduling, monitoring, reporting, security, and web framework.
- Metadata services enable metadata to be shared across tools so that changes made in one InfoSphere Information Server component are instantly visible across all of the suite components. Metadata services are integrated with the metadata repository. Metadata services also allow the exchange of metadata with external tools.

The common Services tier is deployed on J2EE-compliant application servers, such as IBM WebSphere Application Server, which is included with InfoSphere Information Server.

Unified user interface

The face of InfoSphere Information Server is a common graphical interface and tool framework. Shared interfaces, such as the IBM InfoSphere Information Server Console and the IBM InfoSphere Information Server Web Console, provide a common interface, visual controls, and user experience across products. Common functions, such as catalog browsing, metadata import, query, and data browsing, all expose underlying common services in a uniform way. InfoSphere Information Server provides rich client interfaces for highly detailed development work and thin clients that run in web browsers for administration.

Application programming interfaces (APIs) support various interface styles that include standard request-reply, service-oriented, event-driven, and scheduled task invocation.

3.3 Information Server component overview

IBM Information Server is a client-server architecture made up of client-based design, administration, and operation tools that access a set of server-based data integration capabilities through a common services layer, as shown in Figure 3-3. Additionally, in Figure 3-3, notice the four tiers of IBM Information Server. The Client tier provides a number of client interfaces, optimized to different user roles within an organization. The Client tier includes IBM InfoSphere DataStage and QualityStage clients (Administrator, Designer, and Director), IBM Information Server Console, and IBM Information Server Web Console.

This tier can be categorized in two broad categories of clients: Administrative clients and User clients. Information Server provides a common layer of services through the Services tier where all the metadata is stored in a common repository in the Metadata Repository tier. The Engine tier provides the high-performance parallel engine that performs analysis, cleansing, and transformation processing. In the following sections, we explain these tiers.

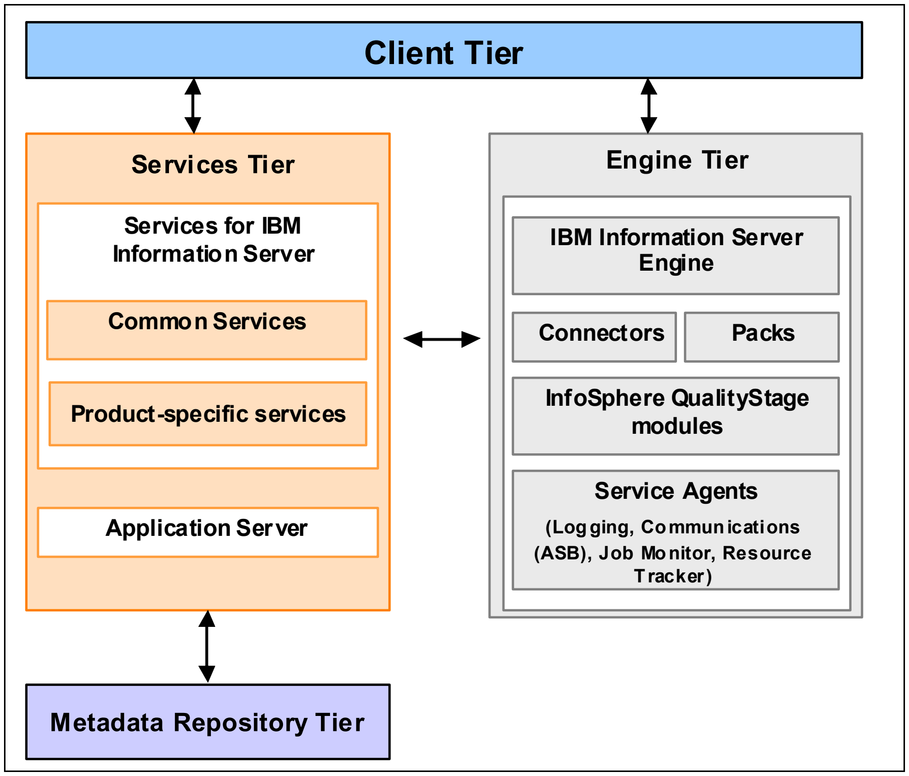

Client Tier
Services Tier
Services for IBM Information Server
Common Services
Product-specific services
Application Server
Engine Tier
IBM Information Server Engine
Connectors
Packs
InfoSphere QualityStage modules
Service Agents
(Logging, Communications (ASB), Job Monitor, Resource Tracker)
Metadata Repository Tier

Figure 3-3 Information Server tiers

3.3.1 Administrative clients

With these clients, you can manage the areas of security, licensing, logging, and scheduling.

Web Console

Administration tasks are performed in the IBM Information Server Web Console. The IBM Information Server Web Console is a browser-based interface for administrative activities, such as managing security and creating views of scheduled tasks.

IBM InfoSphere DataStage Administrator

For IBM InfoSphere DataStage and IBM InfoSphere QualityStage project administration, you use the IBM InfoSphere DataStage Administrator client. It administers IBM InfoSphere DataStage and QualityStage projects and conducts daily operations on the server. It is used to specify general server defaults, add and delete projects, and to set project properties. User and group privileges are also set with the Administrator client.

3.3.2 User clients

These clients help perform client tasks, such as creating, managing, and designing jobs, as well as profiling and analyzing data. Information Server includes the following product modules.

IBM InfoSphere DataStage

IBM InfoSphere DataStage enables organizations to design data flows that extract information from multiple source systems, transform it in ways that make it more valuable, and then deliver it to one or more target databases or applications.

IBM InfoSphere QualityStage

Designed to help organizations understand and improve the overall quality of their data assets, IBM InfoSphere QualityStage provides advanced features to help investigate, repair, consolidate, and validate heterogeneous data within an integration workflow.

IBM InfoSphere Information Services Director

IBM Information Server provides a unified mechanism for publishing and managing shared SOA services across data quality, data transformation, and federation functions. This mechanism allows information specialists to easily deploy services for any information integration task and consistently manage them. Developers can take data integration logic that was built with IBM Information Server and publish it as an “always on” service in minutes. The common services also include the metadata services, which provide standard service-oriented access and analysis of metadata across the platform.

IBM InfoSphere Information Analyzer

IBM InfoSphere Information Analyzer profiles and analyzes data so that you can deliver trusted information to your users. It can automatically scan samples of your data to determine their quality and structure. This analysis aids you in understanding the inputs to your integration process, ranging from individual fields to high-level data entities. With information analysis, you can correct problems with structure or validity before it affects your project. While analysis of source data is a critical first step in any integration project, you must continually monitor the quality of the data. With IBM InfoSphere Information Analyzer, you can treat profiling and analysis as an ongoing process and create business metrics that you can run and track over time.

IBM Information Server FastTrack

IBM Information Server FastTrack simplifies and streamlines communication between the business analyst and developer by capturing business requirements and automatically translating them into IBM InfoSphere DataStage ETL jobs.

IBM InfoSphere Business Glossary

IBM Information Server provides a web-based tool that enables business analysts and subject-matter experts to create, manage, and share a common enterprise vocabulary and classification system. IBM InfoSphere Business Glossary enables users to link business terms to more technical artifacts that are managed by the metadata repository. The metadata repository also enables sharing of the business terms by IBM InfoSphere Data Architect and IBM InfoSphere Information Analyzer, creating a common set of semantic tags for reuse by data modelers, data analysts, business analysts, and users.

IBM InfoSphere Discovery

IBM InfoSphere Discovery analyzes the data values and patterns from one or more sources, to capture these hidden correlations and bring them clearly into view. InfoSphere Discovery applies heuristics and sophisticated algorithms to perform a full range of data analysis techniques: single-source and cross-source data overlap and relationship analysis, advanced matching key discovery, and transformation logic discovery. InfoSphere Discovery accurately identifies relationships and business objects, accelerating the deployment of information-centric projects.

IBM InfoSphere Data Architect

IBM InfoSphere Data Architect is a data modeling tool to create and manage your enterprise data models. Data Architect comes with a full set of features with which you perform mappings, relationships between elements, compare and synchronize different models, enforce standards for naming, meaning, values, and relationships. Data Architect is a tool that is fully integrated with a suite of applications that covers the entire software development lifecycle.

IBM Metadata Workbench

With IBM Metadata Workbench (MWB), you can explore and analyze technical, business, and operational metadata about sources of data. IBM Metadata Workbench helps you to graphically understand these relationships from the sources of information to the places where information is used, even across different tools and technologies. IBM Metadata Workbench outlines the complete lineage of fields from applications, reports, or data warehouses back to source systems, including the types of processing that were performed.

IBM InfoSphere Blueprint Director

With IBM InfoSphere Blueprint Director, you can define and manage a blueprint of your information project landscape from initial sketches through delivery. The integration team can collaborate on actionable information blueprints that connect the business vision with the corresponding technical metadata. It directs the team to apply preferred practices that are based on reference architectures and methodology, provides consistency and connectivity to your information architecture solution by linking the solution overview and detailed design documents together, allowing all team members to understand the project as it evolves. InfoSphere Blueprint Director includes several major capabilities that enable governance teams to develop a common information blueprint, creating project blueprints so that you can link blueprint elements to related artifacts.

3.3.3 Server tiers

IBM Information Server is built entirely on a set of shared services that centralizes core tasks across the platform. Shared services allow these tasks to be managed and controlled in one place, regardless of which suite component is used. The Server tiers of the Information Server Platform include the Services, Engine, and Repository tiers.

Services tier

The Services tier includes both common and product-specific services:

- Common services are used across the Information Server suite for tasks, such as security, user administration, logging, reporting, metadata, and execution.
- Product-specific services provide tasks for specific products within the Information Server suite. For example, IBM InfoSphere Information Analyzer calls a column analyzer service (a product-specific service) that was created for enterprise data analysis. The shared service environment allows integration across IBM Information Server, because the shared services are deployed by using common SOA standards.

IBM Information Server products can access three general categories of service:

- Design: Design services help developers create function-specific services that can also be shared.
- Execution: Execution services include logging, scheduling, monitoring, reporting, security, and web framework.
- Metadata: Using metadata services, metadata is shared "live" across tools so that changes made in one IBM Information Server component are instantly visible across all of the suite components. Metadata services are tightly integrated with the common repository. You can also exchange metadata with external tools by using metadata services.

The common services layer is deployed on the J2EE-compliant application server IBM WebSphere Application Server, which is included with IBM Information Server.

Repository tier

The shared repository is used to store all IBM Information Server product module objects[1] (including IBM InfoSphere DataStage objects) and is shared with other applications in the suite. Clients can access metadata and the results of data analysis from the service layers.

Engine tier

This tier is the parallel runtime engine that executes the IBM Information Server tasks. It consists of the Information Server Engine, Service Agents, Connectors, and Packaged Application Connectivity Kits (PACKs[2]):

[1] IBM Information Server product module objects include jobs and table definitions, as well as operational metadata, such as job start and stop times. The repository is also used to store Information Server configuration settings, such as user group assignments and roles.
[2] PACKs provide an application-specific view of data and use the packaged application vendor's application programming interfaces (APIs) for connectivity and business metadata.

- The IBM Information Server Engine consists of the products that you install, such as IBM InfoSphere DataStage and IBM InfoSphere QualityStage. It runs jobs to extract, transform, load, and standardize data. The engine runs DataStage and QualityStage jobs. It also executes the parallel jobs for Information Analyzer tasks.
- Service Agents are Java processes that run in the background on each computer that hosts IBM InfoSphere DataStage. They provide the communication between the Services and Engine tiers of Information Server.
- Connectors and PACKs. IBM Information Server connects to various information sources whether they are structured, unstructured, on the mainframe, or applications. Metadata-driven connectivity is shared across the suite components, and connection objects are reusable across functions. Connectors provide design-time importing of metadata, data browsing and sampling, runtime dynamic metadata access, error handling, and high functionality and high performance runtime data access. Prebuilt interfaces for packaged applications called PACKs provide adapters to SAP, Siebel, Oracle, and other software, enabling integration with enterprise applications and associated reporting and analytical systems.
- Working areas: These working areas are temporary storage areas used by the suite components.
- Information Services Director (ISD) Resource Providers: Information service providers are the (data) sources of operations for your services. Using IBM InfoSphere Information Services Director, you can create services from five sources:
  - IBM InfoSphere DataStage and QualityStage
  - IBM DB2 for Linux, UNIX and Windows
  - IBM InfoSphere Federation Server
  - IBM InfoSphere Classic Federation Server for z/OS
  - Oracle Database Server

3.3.4 Supported topologies

IBM Information Server is built on a highly scalable parallel software architecture that delivers high levels of throughput and performance. For maximum scalability, integration software must do more than run on Symmetric Multiprocessing (SMP) and Massively Parallel Processing (MPP) computer systems. If the data integration platform does not saturate all of the nodes of the MPP box or system in the cluster or grid, scalability cannot be maximized. The IBM Information Server components fully exploit SMP, clustered, grid, and MPP environments to optimize the use of all available hardware resources.

IBM Information Server supports multiple topologies to satisfy various data integration and hardware business requirements:

- Two tier
- Three tier
- Cluster
- Grid

For all topologies, you can add clients and engines (for scalability) on additional computers.

To select a topology, you must consider your performance needs by reviewing the capacity requirements for the topology elements: the server, disk, network, data sources, targets, data volumes, processing requirements, and any service-level agreements (SLAs).

We describe each topology briefly.

Tip: We suggest that you use the same topology for your test and production environments to minimize issues when a job is deployed into production.

Microsoft Windows platform: On a Microsoft Windows platform, the clients, engine, application server, and metadata repository can be collocated on the same machine. This topology (not shown here) is only suitable for demonstrations, as an educational or proof-of-concept (POC) platform.

Two tier

The engine, application server, and metadata repository are all on the same computer system, while the clients are on a different machine, as shown in Figure 3-4 on page 61.

High availability and failover are simpler to manage with two computers, because all the servers fail over at the same time.

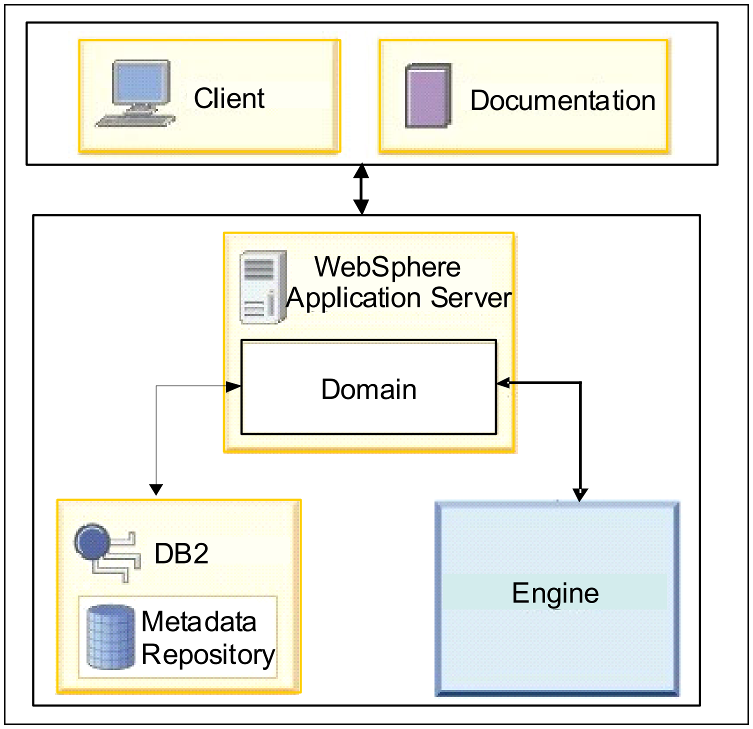


Figure 3-4 Two-tier topology

Three tier

The engine is on one machine, and the application server and metadata repository are co-located on another machine. The clients are on a third machine, as shown in Figure 3-5 on page 62.

Failover configuration is more complex, because of the increased number of failover scenarios that are required by three or more computers.

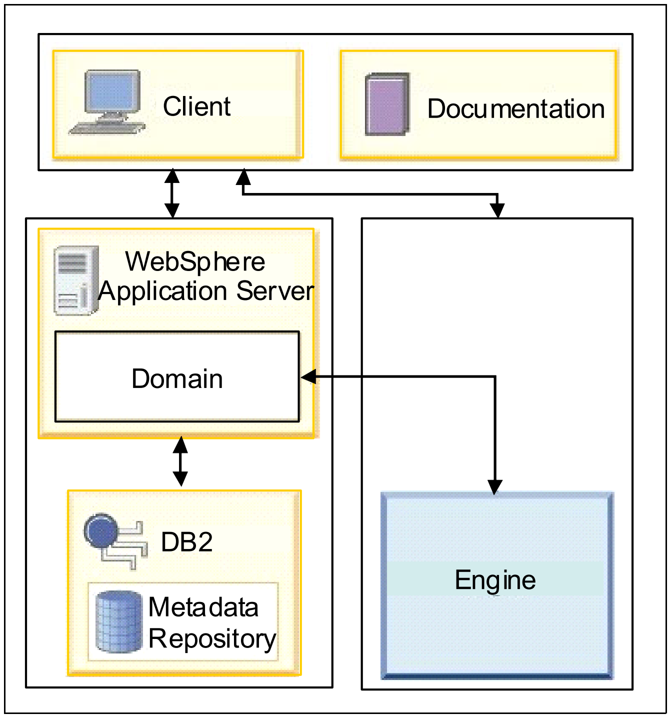


Figure 3-5 Three-tier topology

Cluster

This cluster topology is a slight variation of the three-tier topology with the engine duplicated over multiple computers, as shown in Figure 3-6 on page 63.

In a cluster environment, a single parallel job execution can span multiple computers, each with its own engine.

The processing of a job on the multiple machines is driven by a configuration file that is associated with the job. The configuration file specifies the machines to be used by the job.

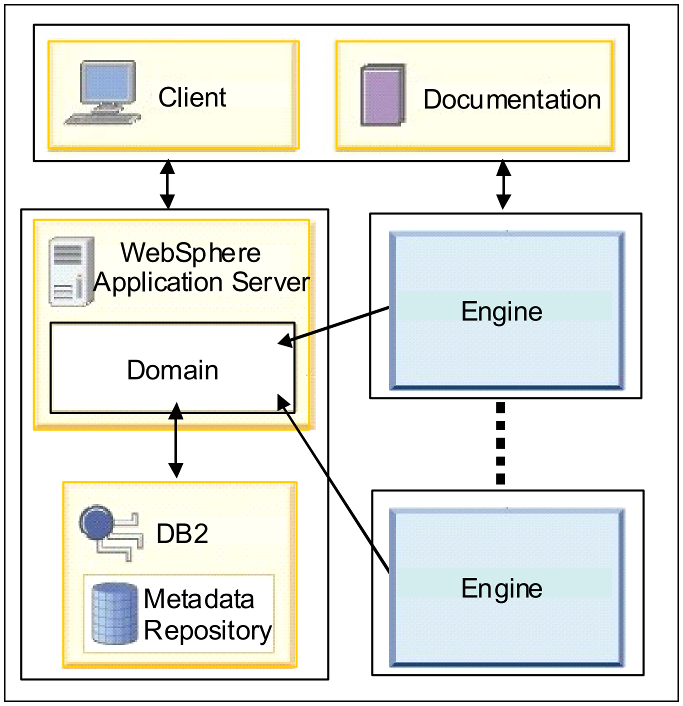


Figure 3-6 Cluster and grid

Grid

With hardware computing power a commodity, grid computing provides more processing power to a task than was previously possible. Grid computing uses all of the low-cost computing resources, processors, and memory that are available on the network to create a single system image.

Grid topology is similar to that of a cluster (Figure 3-6) with engines distributed over multiple machines. As in the case of a cluster environment, a single parallel job execution can span multiple computers, each with its own engine.

The key difference with cluster computing is that in a grid environment, the machines over which a job executes are dynamically determined (through the generation of a dynamic configuration file) with an integrated resource manager, such as IBM Tivoli® Workload Scheduler LoadLeveler®.

The parallel processing architecture of IBM Information Server uses the computing power of grid environments and greatly simplifies the development of scalable integration systems that run in parallel for grid environments.

3.4 InfoSphere Information Server integration scenarios

Information integration is a complex activity that affects every part of an organization. To address the most common integration business problems, these integration scenarios show how you can deploy and use IBM InfoSphere Information Server and the InfoSphere Foundation Tools components together in an integrated fashion. The integration scenarios focus on data quality within a data warehouse implementation.

3.4.1 Data integration challenges

Today, organizations face a wide range of information-related challenges:

- Varied and often unknown data quality problems
- Disputes over the meaning and context of information
- Management of multiple complex transformations
- Use of existing integration processes rather than a duplication of effort
- Ever-increasing quantities of data
- Shrinking processing windows
- Growing need for monitoring and security to ensure compliance with national and international law

Organizations must streamline and connect information and systems across enterprise domains with an integrated information infrastructure. Disconnected information leaves IT organizations unable to respond rapidly to new information requests from business users and executives. With few tools or resources to track the information sprawl, it is also difficult for businesses to monitor data quality and consistently apply business rules. As a result, information remains scattered across the enterprise under a multitude of disorganized categories and incompatible descriptions.

The following examples show several key data integration issues:

- Enterprise application source metadata is not easily assembled in one place to understand what is available. The mix can also include existing sources, which often do not make metadata available through a standard API, if at all.
- Master reference data, names and addresses of suppliers and customers, and part numbers and descriptions differ across applications and duplicate sources of this data.
- Hundreds of extract, transform, and load (ETL) jobs need to be written to move data from all the sources to the new target application.
- Data transformations are required before loading the data so that it fits into the new environment structures.

- The ability to handle large amounts of data that can be run through the process, and finish on time, is essential. Companies need the infrastructure to support running any of the transformation and data-matching routines on demand.

3.4.2 Building a data integration application scenario

Information integration is the process of integrating and transforming data and content to deliver authoritative, consistent, timely and complete information, and governing its quality throughout its lifecycle. The InfoSphere Information Server platform and InfoSphere DataStage (DS) are core to these activities.

IBM InfoSphere Information Server is a software platform that helps organizations derive more value from the complex, heterogeneous information that is spread across their systems. It provides breakthrough collaboration, productivity, and performance for cleansing, transforming, and moving this information consistently and securely throughout the enterprise. It can then be accessed and used in new ways to drive innovation, increase operational efficiency, and lower risk.

IBM InfoSphere DataStage integrates data across multiple and high volume data sources and target applications. It integrates data on demand with a high performance parallel framework, extended metadata management, and enterprise connectivity. DataStage supports the collection, integration, and transformation of large volumes of data, with data structures that range from simple to highly complex.

DataStage can manage data that arrives in real time, as well as data received on a periodic or scheduled basis. This capability enables companies to solve large-scale business problems through high-performance processing of massive data volumes. By using the parallel processing capabilities of multiprocessor hardware platforms, IBM InfoSphere DataStage Enterprise Edition can scale to satisfy the demands of ever-growing data volumes, stringent real-time requirements, and ever-shrinking batch windows.

Along with these key components, establishing consistent development standards helps to improve developer productivity and reduce ongoing maintenance costs. Development standards can also make it easier to integrate external processes, such as automated auditing and reporting, and to build technical and support documentation.

With these components and a great set of standard practices, you are on your way to a highly successful data integration effort.

IBM InfoSphere Information Server features a unified suite of product modules designed to streamline the process of building a data integration application. With InfoSphere Information Server, you can perform five key integration functions:

- Understand the data. InfoSphere Information Server helps you to automatically discover, model, define, and govern information content and structure, as well as understand and analyze the meaning, relationships, and lineage of information. With these capabilities, you can better understand data sources and relationships and define the business rules that help eliminate the risk of using or proliferating bad data.
- Cleanse the data. InfoSphere Information Server supports information quality and consistency by standardizing, validating, matching, and merging data. The platform can help you create a single, comprehensive, accurate view of information by matching records across or within data sources.
- Transform data into information. InfoSphere Information Server transforms and enriches information to help ensure that it is in the proper context for new uses. It also provides high-volume, complex data transformation and movement functionality that can be used for stand-alone ETL scenarios or as a real-time data processing engine for applications or processes.
- Deliver the right information at the right time. InfoSphere Information Server can virtualize, synchronize, or move information to the people, processes, or applications that need it. It also supports critical SOAs by allowing transformation rules to be deployed and reused as services across multiple enterprise applications.
- Perform unified metadata management. InfoSphere Information Server is built on a unified metadata infrastructure that enables shared understanding between the user roles involved in a data integration project, including business, operational, and technical domains. This common, managed infrastructure helps reduce development time and provides a persistent record that can improve confidence in information while helping to eliminate manual coordination efforts.

3.5 Introduction to InfoSphere DataStage

In its simplest form, IBM InfoSphere DataStage performs data transformation and movement from source systems to target systems in batch and in real time. The data sources might include indexed files, sequential files, relational databases, archives, external data sources, enterprise applications, and message queues.

DataStage manages data that arrives and data that is received on a periodic or scheduled basis. It enables companies to solve large-scale business problems with high-performance processing of massive data volumes. By using the parallel processing capabilities of multiprocessor hardware platforms, DataStage can scale to satisfy the demands of ever-growing data volumes, stringent real-time requirements, and ever-shrinking batch windows.

Using the combined suite of IBM Information Server offerings, DataStage can simplify the development of authoritative master data by showing where and how information is stored across source systems. DataStage can also consolidate disparate data into a single, reliable record, cleanse and standardize information, remove duplicates, and link records across systems. This master record can be loaded into operational data stores, data warehouses, or master data applications, such as IBM Master Data Management (MDM), by using IBM InfoSphere DataStage.

IBM InfoSphere DataStage delivers four core capabilities:

- Connectivity to a wide range of mainframe, existing, and enterprise applications, databases, file formats, and external information sources.
- Prebuilt library of more than 300 functions, including data validation rules and complex transformations.
- Maximum throughput by using a parallel, high-performance processing architecture.
- Enterprise-class capabilities for development, deployment, maintenance, and high availability. It uses metadata for analysis and maintenance. It also operates in batch, real time, or as a Web service.

IBM InfoSphere DataStage enables part of the information integration process, data transformation, as shown in Figure 3-2 on page 51.

In the following sections, we briefly describe the following aspects of IBM InfoSphere DataStage:

- Data transformation
- Jobs
- Parallel processing

3.5.1 Data transformation

Data transformation and movement are the processes by which source data is selected, converted, and mapped to the format that is required by targeted systems. The processes manipulate data to bring it into compliance with business, domain, and integrity rules and with other data in the target environment.

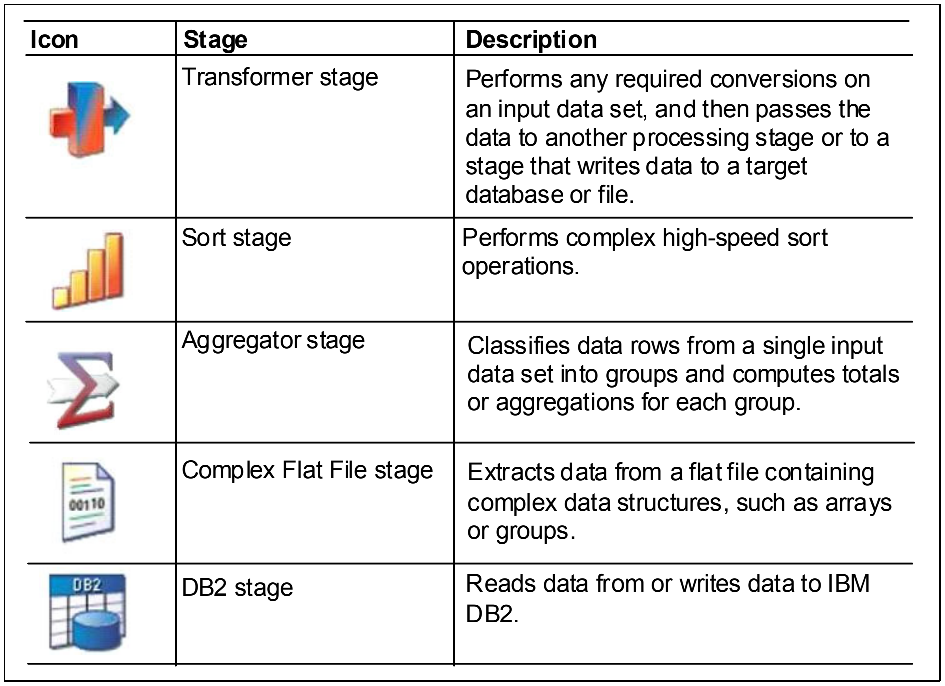
Transformation can take several of the following forms:

- Aggregation: Consolidating or summarizing data values into a single value. Collecting daily sales data to be aggregated to the weekly level is a common example of aggregation.
- Basic conversion: Ensuring that data types are correctly converted and mapped from source to target columns.
- Cleansing: Resolving inconsistencies and fixing the anomalies in source data.
- Derivation: Transforming data from multiple sources by using a complex business rule or algorithm.
- Enrichment: Combining data from internal or external sources to provide additional meaning to the data.
- Normalizing: Reducing the amount of redundant and potentially duplicated data.
- Combining: The process of combining data from multiple sources via parallel Lookup, Join, or Merge operations.
- Pivoting: Converting records in an input stream to many records in the appropriate table in the data warehouse or data mart.
- Sorting: Grouping related records and sequencing data based on data or string values.

3.5.2 Jobs

An IBM InfoSphere DataStage job consists of individual, linked stages, which describe the flow of data from a data source to a data target.

A *stage* usually has at least one data input and one data output. However, certain stages can accept more than one data input, and output to more than one stage. Each stage has a set of predefined and editable properties that tell it how to perform or process data. Properties might include the file name for the Sequential File stage, the columns to sort, the transformations to perform, and the database table name for the DB2 stage. These properties are viewed or edited by using stage editors. Stages are added to a job and linked by using the Designer. Figure 3-7 on page 69 shows several of the stages and their iconic representations.



Icon	Stage	Description
	Transformer stage	Performs any required conversions on an input data set, and then passes the data to another processing stage or to a stage that writes data to a target database or file.
	Sort stage	Performs complex high-speed sort operations.
	Aggregator stage	Classifies data rows from a single input data set into groups and computes totals or aggregations for each group.
	Complex Flat File stage	Extracts data from a flat file containing complex data structures, such as arrays or groups.
	DB2 stage	Reads data from or writes data to IBM DB2.

Figure 3-7 Stage examples

Stages and links can be grouped in a shared container. Instances of the shared container can then be reused in different parallel jobs. You can also define a local container within a job, which groups stages and links into a single unit. The local container must be used within the job in which it is defined.

Separate types of jobs have separate stage types. The stages that are available in the Designer depend on the type of job that is currently open in the Designer.

Parallel Job stages are organized into groups on the Designer palette:

- General includes stages, such as Container and Link.
- Data Quality includes stages, such as Investigate, Standardize, Reference Match, and Survive.

  Data Quality: This group applies when IBM InfoSphere QualityStage is installed.

- Database includes stages, such as Classic Federation, DB2 UDB, DB2 UDB/Enterprise, Oracle, Sybase, SQL Server, Teradata, Distributed Transaction, and Open Database Connectivity (ODBC).

- Development/Debug includes stages, such as Peek, Sample, Head, Tail, and Row Generator.
- File includes stages, such as Complex Flat File, Data Set, Lookup File Set, and Sequential File.
- Processing includes stages, such as Aggregator, Copy, FTP, Funnel, Join, Lookup, Merge, Remove Duplicates, Slowly Changing Dimension, Surrogate Key Generator, Sort, and Transformer.
- Real Time includes stages, such as Web Services Transformer, WebSphere MQ, and Web Services Client.
- Restructure includes stages, such as Column Export and Column Import.

Stages: For details about all the available stages, refer to *IBM InfoSphere DataStage and QualityStage Parallel Job Developer's Guide*, SC18-9891-03, and relevant connectivity guides for the stages that relate to connecting to external data sources and data targets.

3.5.3 Parallel processing

Figure 3-8 represents a simple job: a data source, a Transformer (conversion) stage, and the data target. The links between the stages represent the flow of data into or out of a stage.

In a parallel job, each stage normally (but not always) corresponds to a process. You can have multiple instances of each process to run on the available processors in your system.

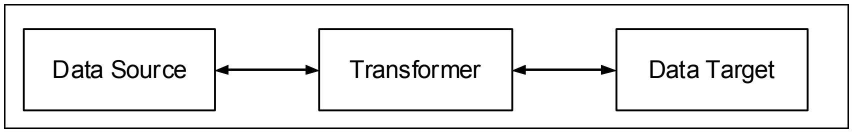


Figure 3-8 Simple IBM InfoSphere DataStage job

A parallel DataStage job incorporates two types of parallel processing: pipeline and partitioning. Both of these methods are used at run time by the Information Server Engine to execute the simple job that is shown in Figure 3-8.

To the DataStage developer, this job appears the same way on your Designer canvas, but you can optimize it through advanced properties.

Pipeline parallelism

In the Figure 3-8 example, all stages run concurrently, even in a single-node configuration. As data is read from the Oracle source, it is passed to the

Transformer stage for transformation, where it is then passed to the DB2 target. Instead of waiting for all source data to be read, as soon as the source data stream starts to produce rows, these rows are passed to the subsequent stages. This method is called *pipeline parallelism*. All three stages in our example operate simultaneously regardless of the degree of parallelism of the configuration file. The Information Server Engine always executes jobs with pipeline parallelism.

If you ran the example job on a system with multiple processors, the stage reading starts on one processor and starts filling a pipeline with the data that it read. The Transformer stage starts running as soon as data exists in the pipeline, processes it, and starts filling another pipeline. The stage that writes the transformed data to the target database similarly starts writing as soon as data is available. Thus, all three stages operate simultaneously.

Important: You do not need multiple processors to run in parallel. A single processor can run multiple concurrent processes.

Partition parallelism

When large volumes of data are involved, you can use the power of parallel processing to your best advantage by partitioning the data into a number of separate sets, with each partition handled by a separate instance of the job stages. Partition parallelism is accomplished at run time, instead of a manual process that is required by traditional systems.

The DataStage developer needs to specify only the algorithm to partition the data, not the degree of parallelism or where the job executes. By using partition parallelism, the same job effectively is run simultaneously by several processors, and each processor handles a separate subset of the total data. At the end of the job, the data partitions can be collected again and written to a single data source, as shown in Figure 3-9.

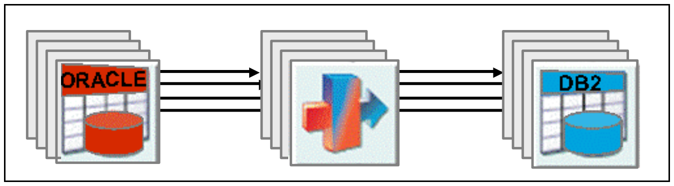


Figure 3-9 Conceptual representation of a job that uses partition parallelism

Combining pipeline and partition parallelism

The Information Server Engine combines pipeline and partition parallel processing to achieve even greater performance gains. In this scenario, stages process partitioned data and fill pipelines so that the next stage can start on that partition before the previous stage finishes, as shown in Figure 3-10.

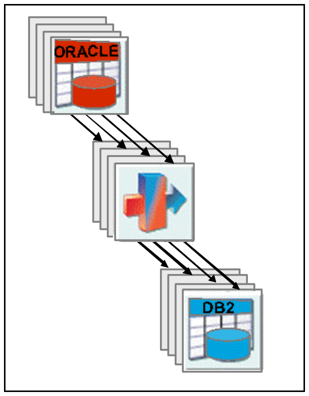


Figure 3-10 Pipeline and partition parallelism

In certain circumstances, you might want to repartition your data between stages. This repartition might happen, for example, where you want to group data differently. Suppose that you initially processed data based on customer last name, but now you want to process data grouped by zip code. You must repartition to ensure that all customers that share a zip code are in the same group. With DataStage, you can repartition between stages as and when necessary. With the Information Server Engine, repartitioning happens in memory between stages, instead of writing to disk.

3.6 DataStage architecture

IBM InfoSphere DataStage facilitates data integration in both high-volume batch and services-oriented deployment scenarios required by enterprise system architectures. As part of the integrated IBM Information Server platform, it is supported by a broad range of shared services and benefits from the reuse of several suite components.

IBM InfoSphere DataStage and IBM InfoSphere QualityStage share the infrastructure for importing and exporting data, designing, deploying, and running jobs, and reporting. The developer uses the same design canvas to specify the flow of data from preparation to transformation and delivery.

Multiple discrete services give IBM InfoSphere DataStage the flexibility to match increasingly varied customer environments and tiered architectures. Figure 3-2 on page 51 shows how IBM InfoSphere DataStage Designer (labeled “User

Clients") interacts with other elements of the IBM Information Server platform to deliver enterprise data analysis services.

In this section, we briefly describe the following topics:

- Shared components
- Runtime architecture

3.6.1 Shared components

On Figure 3-2 on page 51, the following suite components are shared between IBM InfoSphere DataStage and IBM Information Server. For the unified user interface, the following client applications make up the IBM InfoSphere DataStage user interface:

- IBM InfoSphere DataStage and QualityStage Designer: A graphical design interface is used to create InfoSphere DataStage applications, which are known as *jobs*. Because transformation is part of data quality, the InfoSphere DataStage and QualityStage Designer is the design interface for both InfoSphere DataStage and InfoSphere QualityStage. Each job specifies the data sources, required transformations, and destination of the data. Jobs are compiled to create parallel job flows and reusable components that are scheduled by the InfoSphere DataStage and QualityStage Director and run in parallel by the Information Server Engine. The Designer client manages design metadata in the repository, while compiled execution data is deployed on the Information Server Engine tier.
- IBM InfoSphere DataStage and QualityStage Director: A graphical user interface that is used to validate, schedule, run, and monitor InfoSphere DataStage job sequences. The Director client shows job runtime information, including job status and detailed job logs. This client can also be used to establish schedules for job execution.
- InfoSphere DataStage and InfoSphere QualityStage Administrator: A graphical user interface that is used for administration tasks:
  - Setting up DataStage and Information Server Engine users
  - Creating, deleting, and customizing projects
  - Setting up criteria for purging runtime log records

Common services

As part of the IBM Information Server Suite, DataStage uses the common services, as well as DataStage-specific services.

The common services provide flexible, configurable interconnections among the many parts of the architecture:

- Metadata services, such as impact analysis and search
- Execution services that support all InfoSphere DataStage functions
- Design services that support the development and maintenance of InfoSphere DataStage tasks

Common repository

The common repository holds three types of metadata that are required to support IBM InfoSphere DataStage:

- Project metadata: All the project-level metadata components, including IBM InfoSphere DataStage jobs, table definitions, built-in stages, reusable subcomponents, and routines, are organized into folders.
- Operational metadata: The repository holds metadata that describes the operational history of integration process runs, the success or failure of jobs, the parameters that were used, and the time and date of these events.
- Design metadata: The repository holds design-time metadata that is created by the IBM InfoSphere DataStage and QualityStage Designer and IBM InfoSphere Information Analyzer.

The common parallel processing engine runs executable jobs that extract, transform, and load data in a wide variety of settings. The engine uses parallelism and pipelining to handle high volumes of work more quickly and to scale a single job across the boundaries of a single server in cluster or grid topologies.

The common connectors provide connectivity to many external resources and access to the common repository from the processing engine. Any data source that is supported by IBM Information Server can be used as input to or output from an IBM InfoSphere DataStage job.

3.6.2 Runtime architecture

In this section, we briefly describe the generation of the IBM Orchestrate® SHell (OSH) script. We describe the execution flow of IBM InfoSphere DataStage using the Information Server Engine.

OSH script

The IBM InfoSphere DataStage and QualityStage Designer client creates IBM InfoSphere DataStage jobs that are compiled into parallel job flows, and reusable components that execute on the parallel Information Server Engine. With the

InfoSphere DataStage and QualityStage Designer client, you can use familiar graphical point-and-click techniques to develop job flows for extracting, cleansing, transforming, integrating, and loading data into target files, target systems, or packaged applications.

The Designer generates all the code. It generates the OSH and C++ code for any Transformer stages used.

The Designer performs the following tasks:

- Validates link requirements, mandatory stage options, and transformer logic
- Generates OSH representation of data flows and stages (representations of framework “operators”)
- Generates transform code for each Transformer stage, which is then compiled into C++ and then to corresponding native operators
- Compiles Reusable BuildOp stages by using the Designer GUI or from the command line

You need to know the following information about the OSH:

- Comment blocks introduce each operator, the order of which is determined by the order that you added stages to the canvas.
- OSH uses the familiar syntax of the UNIX shell, such as Operator name, schema, operator options (“-name value” format), input (indicated by *n*< where *n* is the input#), and output (indicated by *n*> where *n* is the output #).
- For every operator, input and output datasets are numbered sequentially starting from zero.
- Virtual datasets (in-memory, native representation of data links) are generated to connect operators.

Execution order: The execution order of operators is dictated by input/output designators, and not by their placement on the diagram. The datasets connect the OSH operators. These datasets are “*virtual datasets*,” that is, in-memory data flows. Link names are used in dataset names; therefore, ensure that you use meaningful names for the links.

Framework (Information Server Engine) terms and DataStage terms have equivalency. The GUI frequently uses terms from both paradigms. Runtime messages use framework terminology, because the framework engine is where execution occurs.

The following list shows the equivalency between framework and DataStage terms:

- Schema corresponds to table definition
- Property corresponds to format
- Type corresponds to SQL type and length
- Virtual dataset corresponds to link
- Record/field corresponds to row/column
- Operator corresponds to stage
- Step, flow, and OSH command correspond to a job
- Framework corresponds to Information Server Engine

Execution flow

When you execute a job, the generated OSH and contents of the configuration file ($APT_CONFIG_FILE) are used to compose a “score,” which is similar to an SQL query optimization plan.

At run time, IBM InfoSphere DataStage identifies the degree of parallelism and node assignments for each operator, and inserts sorts and partitioners as needed to ensure correct results. It also defines the connection topology (virtual datasets/links) between adjacent operators/stages, and inserts buffer operators to prevent deadlocks (for example, in fork-joins). It also defines the number of actual operating system (OS) processes. Multiple operators/stages are combined within a single OS process as appropriate, to improve performance and optimize resource requirements.

The job score is used to fork processes with communication interconnects for data, message, and control. Processing begins after the job score and processes are created. Job processing ends when either the last row of data is processed by the final operator, a fatal error is encountered by any operator, or the job is halted by DataStage Job Control or human intervention, such as DataStage Director STOP.

Settings: You can direct the score to a job log by setting $APT_DUMP_SCORE. To identify the Score dump, look for “`main program: This step....`”.

You can set $APT_STARTUP_STATUS to show each step of the job start-up. You can set $APT_PM_SHOW_PIDS to show process IDs in the DataStage log.

Job scores are divided into two sections: datasets (partitioning and collecting) and operators (node/operator mapping). Both sections identify sequential or parallel processing.

The execution (orchestra) manages control and message flow across processes and consists of the conductor node and one or more processing nodes, as shown in Figure 3-11. Actual data flows from player to player. The conductor and section leader are only used to control process execution through control and message channels:

- Conductor is the initial framework process. It creates the Section Leader (SL) processes (one per node), consolidates messages to the DataStage log, and manages orderly shutdown. The Conductor node has the start-up process. The Conductor also communicates with the players.
- Section Leader is a process that forks player processes (one per stage) and manages up/down communications. SLs communicate between the conductor and player processes only. For a specific parallel configuration file, one section leader is started for each logical node.
- Players are the actual processes that are associated with the stages. A player sends **`stderr`** and **`stdout`** to the SL, establishes connections to other players for data flow, and cleans up on completion. Each player must be able to communicate with every other player. Separate communication channels (pathways) exist for control, errors, messages, and data. The data channel does not go through the section leader/conductor, because this design limits scalability. Data flows directly from the upstream operator to the downstream operator.

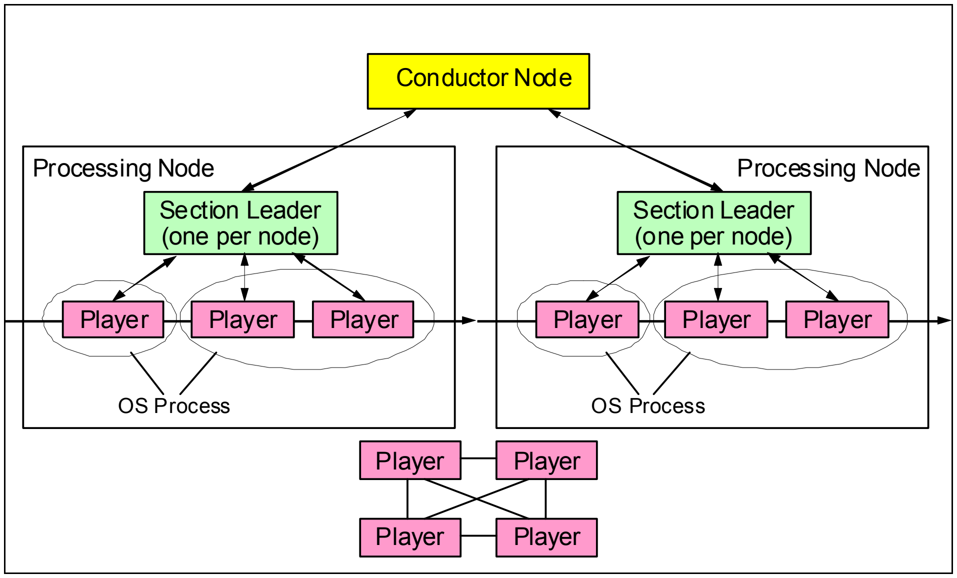


Figure 3-11 Parallel execution flow

3.6.3 XML Pack architecture

XML Pack is a DataStage connector that is designed to handle general XML data processing. XML Pack is installed as part of DataStage installation in the Client, Service, and Engine tiers. XML Pack architecture is depicted in Figure 3-12.

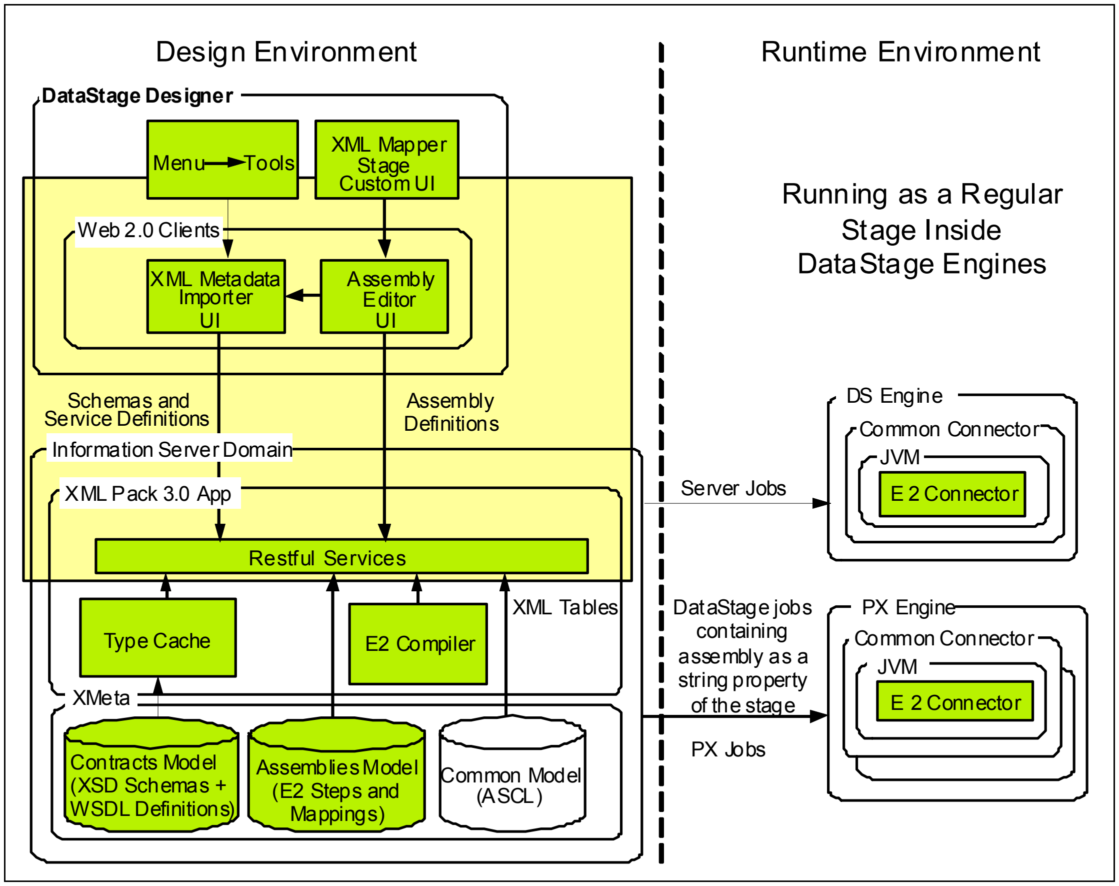


Figure 3-12 XML Pack architecture

XML Pack has four logic tiers:

- Client tier has two major components:
  - XML Metadata Importer UI, which is also called *Schema Library Manager*
  - Assembly Editor, which you use to create the XML Pack job
- Service tier provides Representational State Transfer (REST) services to wrap around the XML Pack schema type library cache and XML Pack compilation function.

- Metadata Repository tier stores schema library information and the assembly information.
- Engine tier hosts the XML Pack connector run time to run the XML Pack job.

In terms of binary components, XML Pack provides client, services, and engine components. The Client tier has one executable file and a set of dynamic link library (DLL) files.

Important: Because the Client tier is installed on top of the Windows operating system, binary components are Windows components.

The Service tier has one WAR file for client interaction and one EAR file to provide REST service. The Engine tier has one large JAR file, which is expanded as a list of JAR files in the engine installation directory.

3.7 Real-time applications

The parallel framework is extended for real-time applications. The parallel framework includes three extended aspects:

- Always-on: Real-time stage types that keep jobs always up and running.
- End-of-wave: The ability for jobs to remain always-on introduces the need to flush records across process boundaries.
- Support for transactions in target database stages.

3.7.1 Always-on source stage types

The ability for a job to stay always-on is determined by the nature of the source stages. In batch applications, source stages read data from sources, such as flat files, database queries, and FTP feeds. All those stages read their corresponding sources until exhaustion. Upon reaching the end-of-file (EOF), each source stage propagates downstream the end-of-file (EOF) to the next stage in the pipeline. The job terminates when all target stages run to completion.

In real-time applications, this model is inefficient, because the job must remain running even though there might not be any data for it to process. Under those circumstances, the stages are to remain idle, waiting for data to arrive.

A few stages keep listening for more incoming data and terminate only under certain special circumstances, depending on the type of stage:

- ISD Input

  This stage is the input stage for Information Services Director applications. It is the same source stage for a number of supported bindings, such as Enterprise JavaBeans (EJB), SOAP, and Java Message Service (JMS).

- MQConnector:

  - MQ reader stage in Information Server 8.x.
  - For pre-8.x Information Server, the corresponding stage is named MQRead.

- Custom plug-ins

  New stage types can be created for special purposes with the following approaches:

  - Custom operators
  - BuildOps
  - Java client

The termination of real-time DS jobs is started by each source stage:

- For ISD jobs, the Information Services Director application must be suspended or undeployed.
- For applications, the MQConnector must reach one of the following conditions:
  - It reads a message of a certain special type (configured on the stage properties).
  - It reaches a read timeout.
  - It reads a certain pre-determined maximum number of messages.

3.7.2 End-of-wave

The ability for jobs to remain always-on introduces the need to flush records at the end of each request.

An end-of-wave (EOW) marker is a hidden record type that forces the flushing of records. An EOW marker is generated by the source real-time stage, depending on its nature and certain conditions set as stage properties.

Figure 3-13 on page 81 shows an example of how EOW markers are propagated through a parallel job. The source stage can be an MQConnector; in this case, the target stage is a Distributed Transaction stage (DTS).

EOW markers are propagated as normal records. However, they do not carry any data. They cause the state of stages to be reset, as well as records to be flushed out of virtual datasets, so a response can be forced.

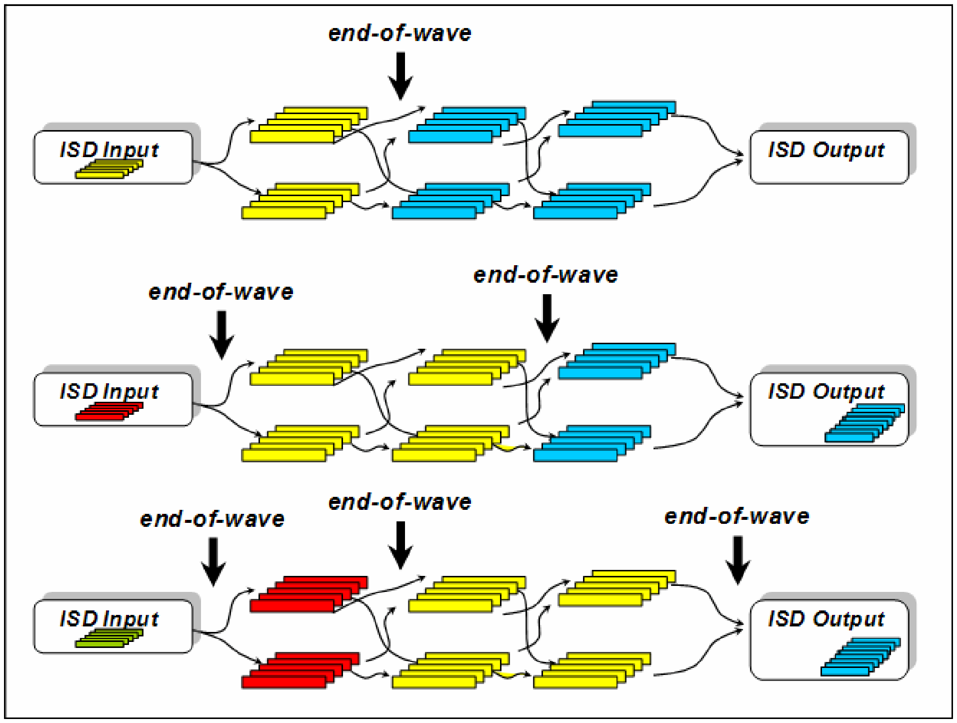


Figure 3-13 EOWs flowing through a parallel job

The ISD and MQConnector stages generate EOW markers in the following way:

- ISD Input:
  - It issues an EOW for each incoming request (SOAP, EJB, or JMS).
  - ISD converts an incoming request to one or more records. A request might consist of an array of records, or a single record, such as one large XML payload. The ISD input stage passes on downstream all records for a single incoming request. After passing on the last record for a request, it sends out an EOW marker.
  - The mapping from incoming requests to records is determined when the operation is defined by the Information Services Console.

- MQConnector:
  - It issues an EOW for one or more incoming messages. The EOW is determined by a parameter named Record Count.
  - It issues an EOW after a certain amount of time elapses, which is determined by a parameter named Time Interval.
  - A combination of the two.

EOWs modify the behavior of regular stages. Upon receiving EOW markers, the internal state of the stage must be reset, so a new execution context begins. For most stages (Parallel Transformers, Modify, or Lookups), there is no practical impact from a job design perspective.

For database sparse lookups, record flow for the Lookup stage is the same. But the stage needs to keep its connections to the database across waves, instead of reconnecting after each wave.

However, there are a few stages whose results are directly affected by EOWs:

- Sorts
- Aggregations

For these stages, the corresponding logic is restricted to the records that belong to a certain wave. Instead of consuming all records during the entire execution of the job, the stage produces a partial result, for the records that belong to a certain wave. A Sort stage, for instance, writes out sorted results that are sorted only in the wave, and not across waves. The stage continues with the records for the next wave, until a new EOW marker arrives.

3.7.3 Transaction support

Support for real-time applications is not complete without the proper handling of database transactions. Batch applications rely on standard database stage types to execute bulk loads or inserts/updates/deletes.

The transaction context for database stages prior to Information Server 8.5 always involved a single target table. Those stages support a single input link, which maps to one or a couple of SQL statements against a single target table. If there are multiple database stages on a job, each stage works on its own connection and transaction context. One stage is oblivious to what is going on in other database stages, as depicted in Figure 3-14 on page 83.

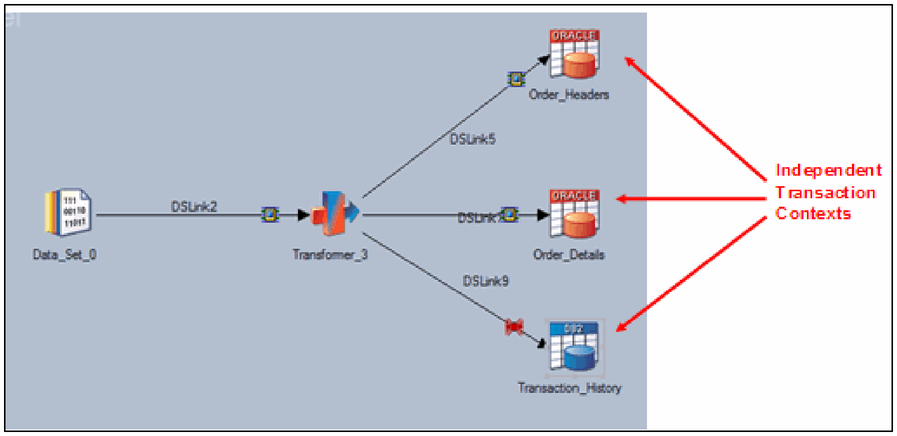


Figure 3-14 Transaction contexts with standard database stage types

The grouping of records into transactions might be specified as a stage property. However, from a database consistency perspective, ultimately the entire batch process tends to be the transactional unit. If one or more jobs fail, exceptions must be addressed and jobs must be restarted and completed so the final database state at the end of the batch window is fully consistent.

In real-time scenarios, restarting jobs is not an option. The updates from a real-time job must yield a consistent database after the processing of every wave.

Prior to Version 8.1, no other option was available to Information Services Director (ISD) and 7.X Real-Time Integration (RTI) jobs other than resorting to multiple separate database stages when applying changes to target databases as part of a single job flow. The best option was to synchronize the reject output of the database stages before sending the final response to the ISD Output stage. See 3.8.3, “Synchronizing database stages with ISD output” on page 95 for a description of this technique.

The Connector stages, introduced in Information Server 8.0.1 and further enhanced in 8.5, provide for a new uniform interface that enables enhanced transaction support when dealing with a single target database stage type. This new uniform interface enables enhanced transaction support when dealing with message-oriented and heterogeneous targets, such as the Distributed Transaction stage (DTS).

We describe the Connector and DTS stages in the following subsections.

Connector stages

Information Server 8.5 has Connector stages for the following targets:

- DB2
- Oracle
- Teradata
- ODBC
- IBM MQSeries®

Begining with IS 8.5, Connectors support multiple input links, instead of a single input link, which is the limit in versions prior to IS 8.5 for Connectors and Enterprise database stage types. With multiple input links, a Connector stage executes all SQL statements for all rows from all input links as part of single unit of work. Figure 3-15 shows this action.

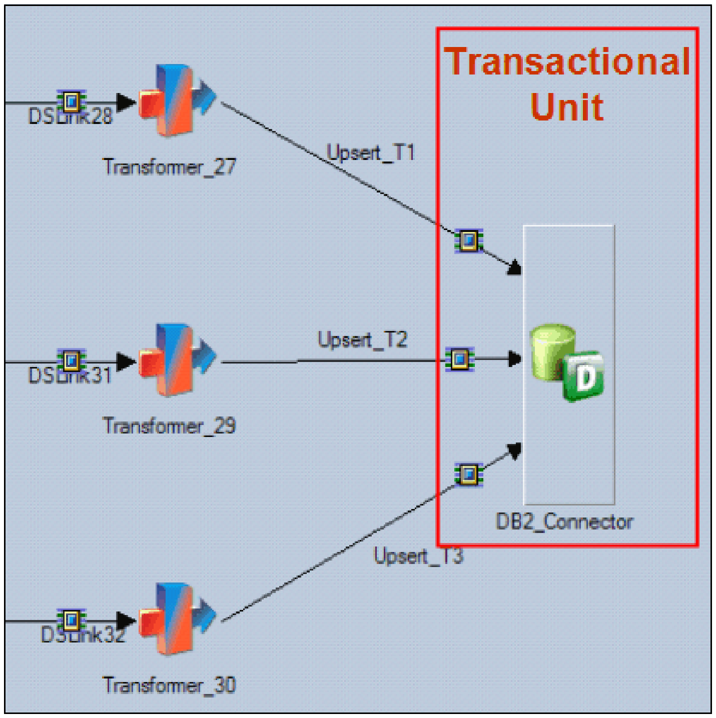


Figure 3-15 Multistatement transactional unit with a connector for a single database type

One transaction is committed per wave. It is up to a source stage (such as ISD input or MQConnector) to generate EOW markers that are propagated down to the target database Connector.

With database Connectors, ISD jobs no longer have to cope with potential database inconsistencies in the event of failure. ISD requests might still need to be re-executed (either SOAP, EJB, or JMS, depending on the binding type), but the transactional consistency across multiple tables in the target database is guaranteed as a unit of work.

For transactions spanning multiple database types and those transactions that include guaranteed delivery of MQ messages, you must use the Distributed Transaction stage, which is described next.

Distributed Transaction stage (DTS)

The stage that provides transactional consistency across multiple SQL statements for heterogeneous resource manager types in message-oriented applications is the Distributed Transaction stage (DTS).

This stage was originally introduced in DataStage 7.5, with the name *UnitOfWork*. It was re-engineered for Information Server 8, and it is now called DTS. Figure 3-16 illustrates DTS.

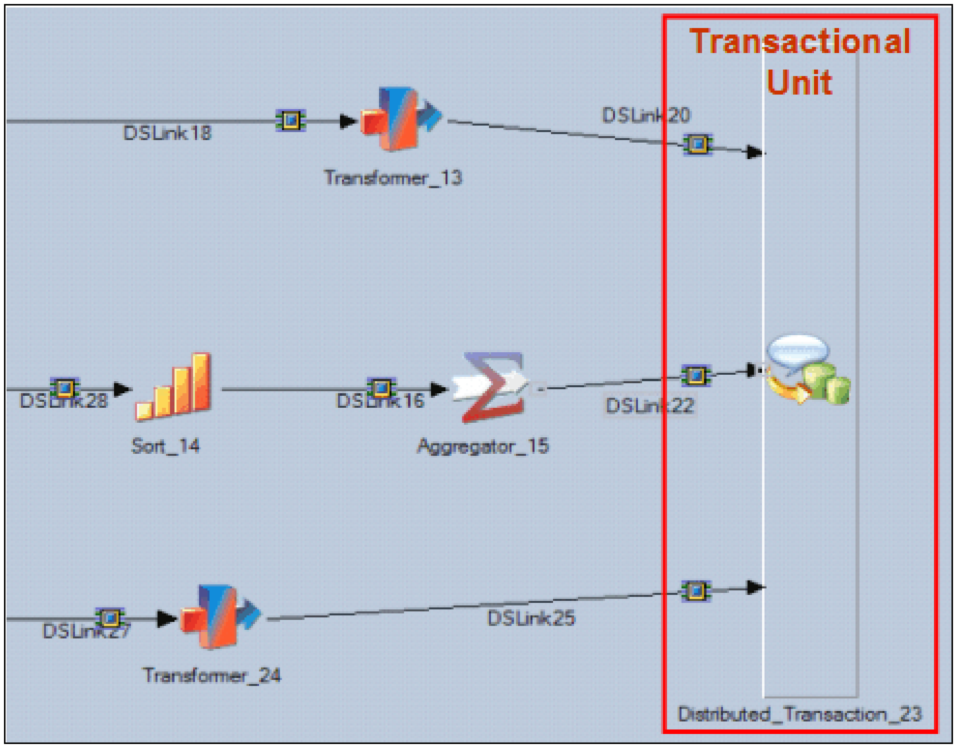


Figure 3-16 Distributed Transaction stage

The DTS applies to target databases all the SQL statements for incoming records for all input links for a specific wave, as a single unit-of-work. Each wave is independent from the other waves.

It was originally intended to work in conjunction with the MQConnector for message-oriented processing. When used in conjunction with the MQConnector, the DTS provides the following capabilities:

- Guaranteed delivery from a source queue to a target database

  No messages are discarded without making sure that the corresponding transactions are successfully committed.

- One time only semantics

  No transaction is committed more than once.

In IS 8.5, the DTS supports transactions against single or heterogeneous resource managers as part of the same transaction:

- DB2
- Oracle
- Teradata
- ODBC
- MQ

Also, you can use the DTS without a source MQConnector, such as in Information Services Director jobs. You still need to install and configure MQ, because it is required as the underlying transaction manager.

3.7.4 Transactional support in message-oriented applications

In this section, we provide an overview of transactional support in message-oriented applications.

Information Server supports two types of solutions for message-oriented applications:

- Information Services Director with JMS bindings
- MQ/DTS

The first solution is for interoperability with JMS. The second solution is for MQ.

You can use ISD with JMS to process MQSeries messages and MQ/DTS to process JMS messages by setting up a bridging between MQ and JMS with WebSphere enterprise service bus (ESB) capabilities. However, the bridging setup is relatively complexity between JMS and MQ.

We put both solutions side-by-side in a diagram in Figure 3-17. The goal of this illustration is to draw a comparison of how transactions are handled and the path that the data flows.

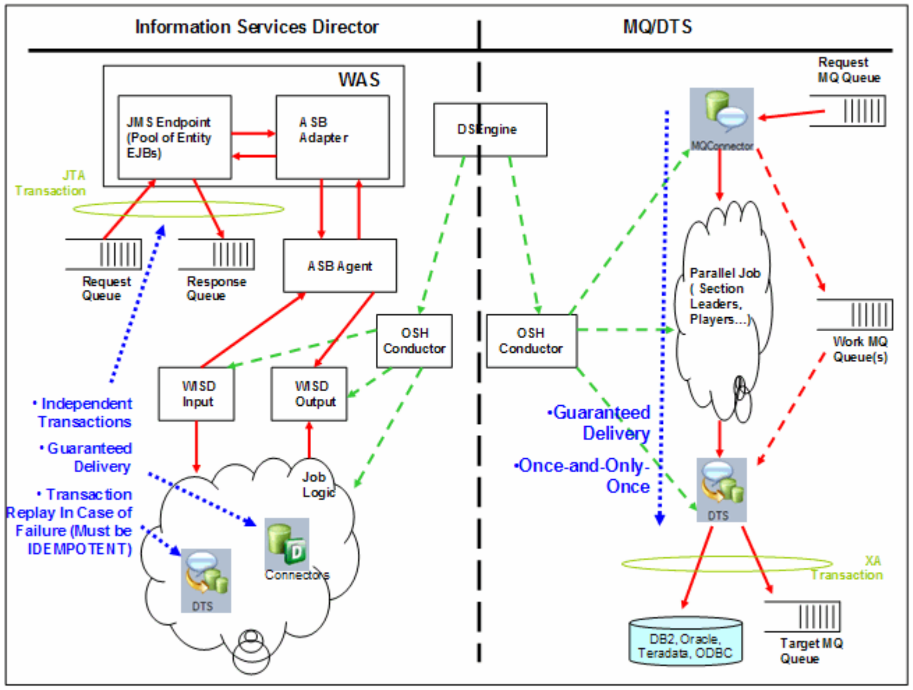


Figure 3-17 Transactional contexts in ISD and MQ/DTS jobs

Both ISD and MQ/DTS jobs are parallel jobs, which are composed of processes that implement a pipeline, possibly with multiple partitions. The parent-child relationships between OS processes are represented by the dotted green lines. The path that is followed by the actual data is represented by solid (red) lines.

ISD jobs deployed with JMS bindings have the active participation of the WebSphere Application Server and the ASB agent. In an MQ/DTS job, the data flow is restricted to the parallel framework processes.

MQ/DTS jobs provide both guaranteed delivery and once-and-only-once semantics. A specific transaction is not committed twice against the target database. And, each transaction is guaranteed to be delivered to the target database. Transaction contexts are entirely managed by DS job stages. A local transaction is managed by the MQConnector (for the transfer of messages from the source queue to the work queue), and an XA transaction involves multiple database targets and the work queue.

In an ISD job, transaction contexts are managed by DS parallel processes, depending on how many database stages are present in the flow. See 3.8.3, “Synchronizing database stages with ISD output” on page 95 through 3.8.5, “Information Services Director with connectors” on page 98.

However, one additional transaction context exists on the JMS side, which is managed by EJBs in the WebSphere Application Server J2EE container as Java Transaction API (JTA) transactions.

JTA transactions make sure that no messages are lost. If any components along the way (WebSphere Application Server, ASB Agent, or the parallel job) fail during the processing of an incoming message before a response is placed on the response queue, the message is reprocessed. The JTA transaction that selected the incoming message from the source queue is rolled back, and the message remains on the source queue. This message is then reprocessed on job restart.

Therefore, database transactions in ISD jobs that are exposed as JMSs must be idempotent. For the same input data, they must yield the same result on the target database.

The MQ/DTS and ISD/JMS solutions offer these strengths:

- MQ/DTS

  Guaranteed delivery from a source queue to a target database
- ISD/JMS

  Adequate for request/response scenarios when JMS queues are the delivery mechanism

The DTS is enhanced in Version 8.5 to support MQ queues as targets. Therefore, a request/response type of scenario can be implemented with MQ/DTS. However, this scenario implies a non-trivial effort in setting up the necessary bridging between JMS and MQ on the WebSphere Application Server instance.

Consider these aspects when taking advantage of ISD with JMS bindings:

- The retry attempt when JTA transactions are enabled largely depends on the JMS provider. In WebSphere 6, with its embedded JMS support, the default is five attempts (this number is configurable). After five attempts, the message is considered a poison message and goes into the dead letter queue.
- Subtle differences from provider to provider can cause problems. You must understand exactly how things operate when MQ is the provider or when the embedded JMS in WebSphere Application Server is the provider.

- The JMS binding also creates other issues because of the pool of EJB. Multiple queue listeners can result in confusion when messages are out of order and a flood of concurrent clients go into ISD and overwhelm the number of instances that are established for DataStage.

3.8 Publishing jobs as services

InfoSphere Information Services Director (ISD) is used to create live, real-time jobs. ISD automates the publication of jobs, maps, and federated queries as services of the following types:

- SOAP
- EJB
- JMS

ISD is notable for the way that it simplifies the exposure of DS jobs as SOA services, letting users bypass the underlying complexities of creating J2EE services for the various binding types.

ISD controls the invocation of those services, supporting request queuing and load balancing across multiple service providers (a DataStage Engine is one of the supported provider types).

A single DataStage job can be deployed as different service types, and it can retain a single data flow design. DS jobs that are exposed as ISD services are referred to as "*ISD Jobs*" throughout this section.

Figure 3-18 on page 90 is reproduced from the ISD manual and depicts its major components. The top half of the diagram shows the components that execute inside the WebSphere Application Server on which the Information Server Domain layer runs. Information Server and ISD are types of J2EE applications that can be executed on top of J2EE containers.

With Version 8.5, Information Server is tightly coupled with WebSphere Application Server, as is Information Services Director and its deployed applications.

The bottom half of Figure 3-18 on page 90 presents components that belong to the Engine Layer, which can reside either on the same host as the Domain, or on separate hosts.

Each WebSphere Information Services Director (WISD) endpoint relates to one of the possible bindings: SOAP, JMS, or EJB. Such endpoints are part of the J2EE applications that are seamlessly installed on the Domain WebSphere

Application Server when an ISD application is successfully deployed with the Information Server Console.

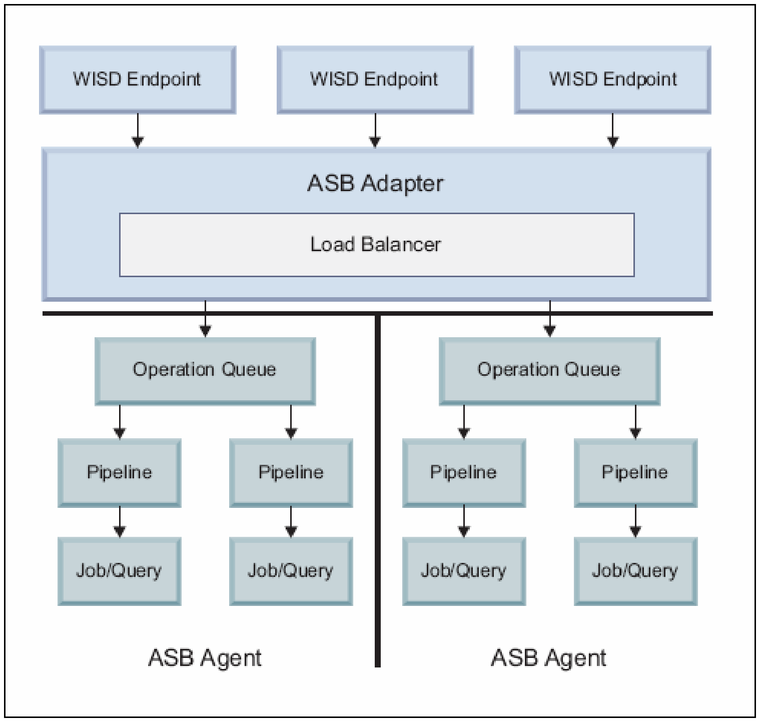


Figure 3-18 Major components of Information Services Director

The endpoints forward incoming requests to the ASB adapter, which provides load balancing and interfacing with multiple services providers. *Load balancing* is another important concept in SOA applications. In this context, it means the spreading incoming requests across multiple DataStage engines.

A single DS engine can service a considerable number of requests. However, if bottlenecks are identified, more engines might be added. In this case, the same DS jobs must be deployed on all participating engines.

The ASB agent is a Java application that intermediates the communication between the ASB adapter and the DS engine. The ASB agent is a stand-alone Java application, because it does not run inside a J2EE container. Because it is a Java application, it is multithreaded and supports servicing multiple incoming requests.

An ASB agent implements the queuing and load balancing of incoming requests. Inside a specific DS engine, multiple pre-started, always-on job instances of the same type might exist, ready to service requests. The always-on nature is important in this situation. Instead of incurring the cost of reactivating jobs whenever a new request arrives, the ASB agent controls the lifecycle of jobs, keeping them up and running for as long as the ISD application is supposed to be active.

One important concept to understand is the *pipelining* of requests. Multiple requests are forwarded to an ISD job instance, before responses are produced by the job and sent back to the ASB agent. A correlation exists between this concept and the pipeline parallelism of DS parallel jobs. Multiple concurrent processes execute the steps of a parallel job in tandem. Pipelining requests allows multiple requests to flow through a job at the same time.

For ISD applications, a direct mapping exists between a service request and a wave. For each service request, an end-of-wave marker is generated. See 3.7.2, “End-of-wave” on page 80.

In this section, we include an explanation of how the components in Figure 3-18 on page 90 map to Information Server layers and how they interact with each other from various aspects:

- Job activation
- Parent-child process relationships
- The flow of actual data

Figure 3-19 on page 92 illustrates these relationships.

After an ISD job is compiled and ready, the ISD developer creates an operation for that job by using the Information Server Console. That operation is created as part of a service, which is an element of an ISD application.

When the ISD operations, services, and application are ready, you can use the Information Server Console to deploy that application. The result is the installation of a J2EE application on the WebSphere Application Server instance. The deployment results in the activation of one or more job instances in the corresponding service provider, which are the DS engines that participate in this deployment.

This deployment is represented by the dashed (blue) arrows. The ISD Administration application forwards the request to the target DS engines through the ASB adapter and the ASB agents.

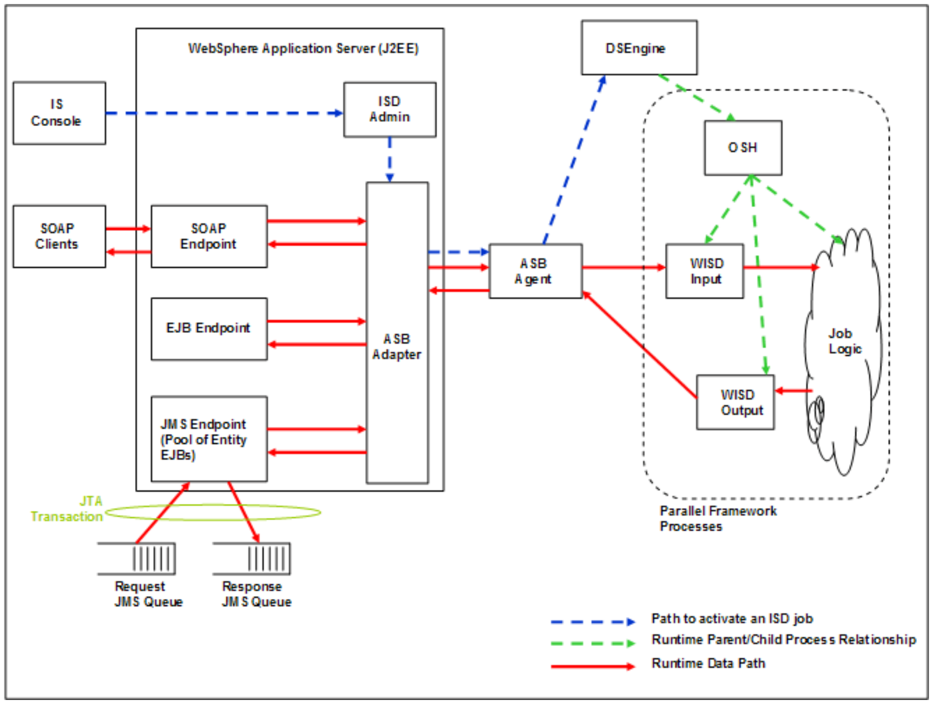


Figure 3-19 Data and control paths in ISD applications

The DS engine, in turn, creates one or more parallel job instances. A parallel job is started with an OSH process (the conductor). This process parses the OSH script that the ISD job flow was compiled into and launches the multiple section leaders and players that implement the runtime version of the job. This process is represented by the dashed (green) arrows.

Incoming requests follow the path that is depicted with the red arrows. They originate from remote or local applications, such as SOAP or EJB clients, and even messages posted onto JMS queues. One endpoint exists for each type of binding for each operation.

All endpoints forward requests to the local ASB adapter. The ASB adapter forwards a request to one of the participating engines according to a load balancing algorithm. The request reaches the remote ASB agent, which puts the request in the pipeline for the specific job instance.

The ASB agent sends the request to the WebSphere Information Services Director (WISD) Input stage for the job instance. The ISD job instance processes the request (an EOW marker is generated by the WISD Input for each request). The job response is sent back to the ASB agent through the WISD Output stage.

The response flows back to the caller, flowing through the same components from which it came originally. Notice that the WebSphere Application Server actively participates in the request processing.

3.8.1 Design topology rules for always-on ISD jobs

The following rules are the job flow design topology rules for always-on ISD jobs:

- There must be one source WISD Input stage and one target WISD Output stage.
- The following rules must be obeyed:
  - All data paths must originate from the same source stage (the WISD Input). Reference links to Normal Lookups are the exceptions. Reference links to Normal Lookups must originate from independent branches.
  - All data paths must lead to the same target WISD Output stage.
  - Dangling branches might lead to Sequential File stages, for instance, but these dangling branches are to be used only for development and debugging. They must not be relied on for production jobs.
  - Normal Lookups can only be used for static reference tables.
  - For any lookups against dynamic database tables, sparse database lookups must be used.
  - Sparse lookups might return multiple records, depending on the business requirements.

3.8.2 Scalability

ISD supports two layers of scalability:

- Multiple job instances
- Deployment across multiple service providers, that is, multiple DS engines

These layers are illustrated in Figure 3-20 on page 94.

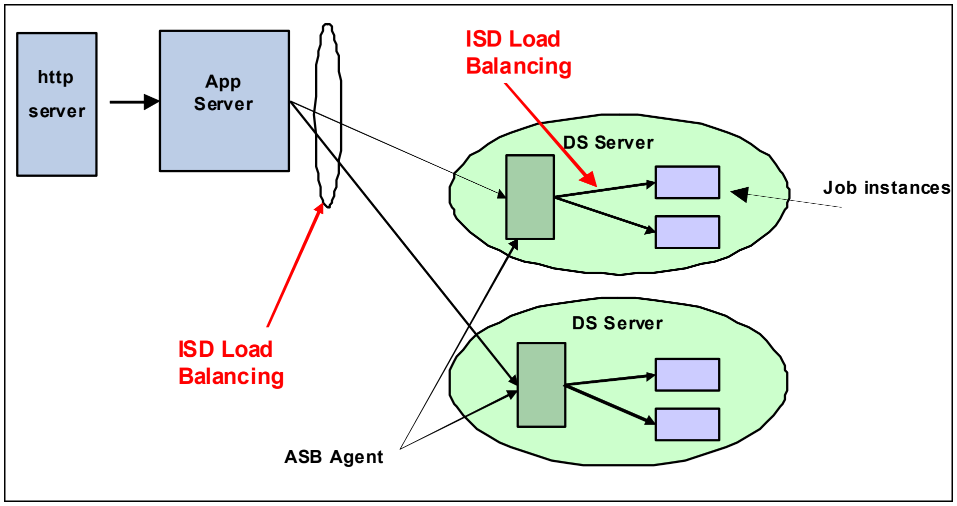


Figure 3-20 ISD load balancing

Whether to deploy multiple job instances and multiple DS engines to service requests for the same ISD application is a matter of performance monitoring and tuning. The performance of the application must be closely monitored to understand possible bottlenecks, which then drive the decisions in terms of additional job instances and engines.

To address performance requirements, we recommend making assessments in the following order:

1. Job design efficiency
2. Number of job instances
3. Number of DS engines

The first task is to ensure that the logic is efficient, which includes ensuring that database transactions, transformations, and the entire job flow are optimally designed.

After the job design is tuned, assess the maximum number of requests that a single job instance can handle. This maximum is a function of the job complexity and the number of requests (in other words, EOW markers) that can flow through the job simultaneously in a pipeline. If that number of requests is not enough to service the amount of incoming requests, more than one job instance can be instantiated. In most cases, increasing the number of job instances helps meet the service-level agreement (SLA) requirements.

However, cases might occur when you reach the maximum number of requests that a single ASB agent can handle. Therefore, the limit of a single DS engine is reached. You can verify this limit when no matter how many job instances are instantiated, the engine cannot handle more simultaneous requests. In this situation, add more DS engines either on the same host (if there is enough spare capacity) or on separate hosts.

Remember that throughout this tuning exercise, we assume that you have enough hardware resources. Increasing the number of job instances and DS engines does not help if the CPUs, disks, and network are already saturated.

3.8.3 Synchronizing database stages with ISD output

Prior to Information Server 8.5, ISD jobs relied on multiple database stages when updating more than one table in a single job flow, similar to what was discussed in 3.7.3, “Transaction support” on page 82.

Figure 3-14 on page 83 illustrates database transaction contexts in a batch application. Each separate database stage maintains a separate connection and transaction context. One database stage is oblivious to what is going on in other database stages even if they are part of the same job.

In an always-on job, the database stages must complete the SQL statements before the response is returned to the caller (that is, before the result is sent to the WISD Output stage).

Pre-8.5 jobs required a technique, which is illustrated in Figure 3-21, that involves a sequence of stages connected to the reject link of a standard database stage.

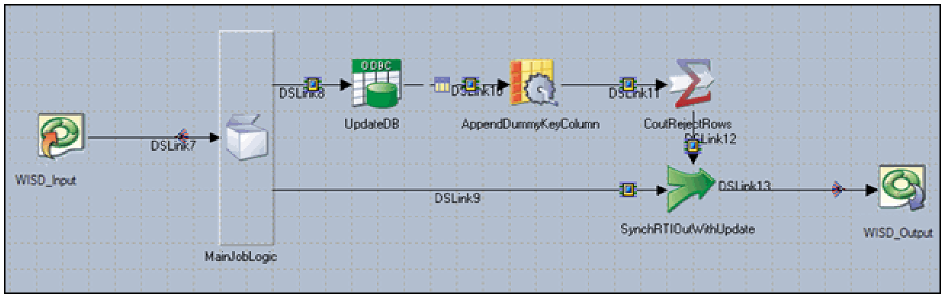


Figure 3-21 Synchronizing database and WISD output

The only exception to this rule in releases prior to 8.5 was the Teradata Connector, which in Version 8.0.1 already supported an output link similar to what is described in 3.8.5, “Information Services Director with connectors” on page 98. However, prior to 8.5, the Teradata Connector did not allow more than a single input link.

A Column Generator created a column, which was used as the aggregation key in the subsequent aggregator. The output from the aggregator and the result from the main job logic (depicted as a local container) synchronized with a Join stage.

The Join stage guaranteed that the response was sent only after the database statements for the wave complete (either successfully or with errors).

The logic that is implemented by the Column Generator, Aggregator, and Join stages is repeated for each standard database stage that is present in the flow. Synchronized results ensure that no other database activity occurs for a request that is already answered.

Again, those steps were necessary in pre-8.5 releases. In IS 8.5, the Database Connectors are substantially enhanced to support multiple input links and output links that can forward not only rejected rows, but also processed rows.

Two alternatives for always-on 8.5 jobs are available for database operations:

- DTS
- Database Connector stages

We describe these alternatives in subsequent sections.

3.8.4 ISD with DTS

The DTS stage is enhanced as part of the 8.5 release to support transactions without the presence of source and work queues. This enhancement enables a DTS stage to respond and commit transactions solely as a result of EOW markers. Using the mode that is described in this section, a DTS stage can be included as part of an always-on ISD job. The DTS stage supports units of work that meet the following requirements:

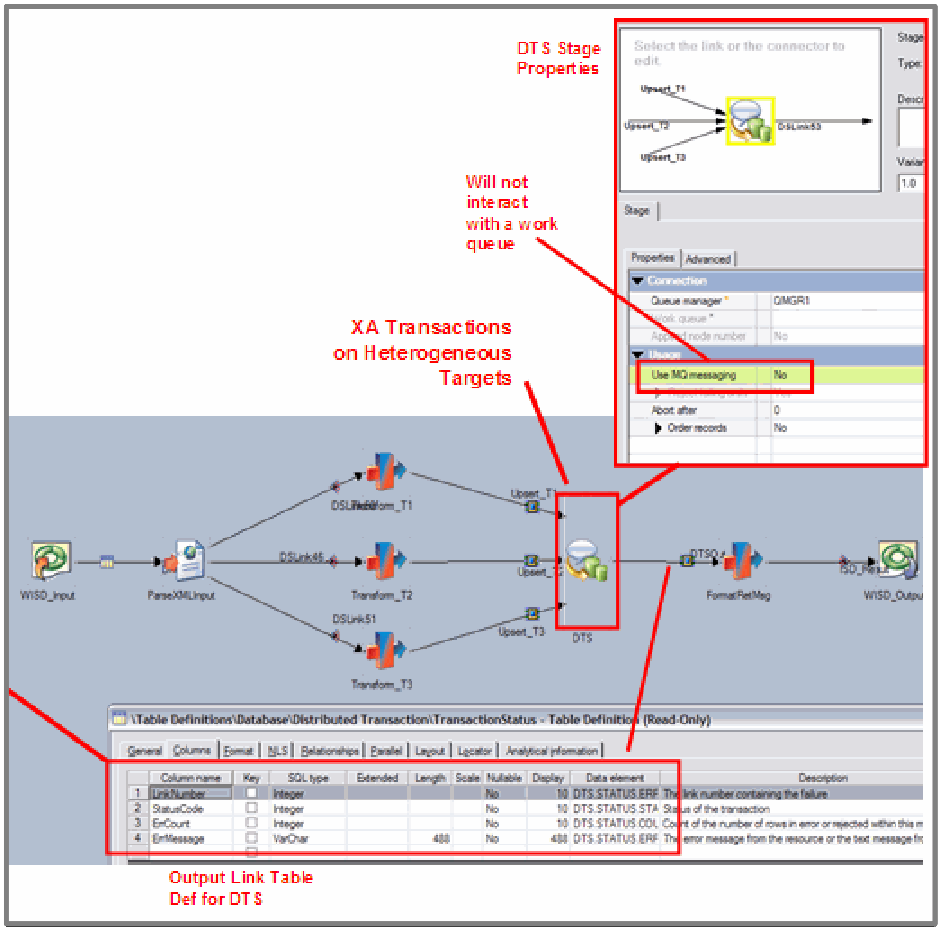
- Multiple SQL statements
- Multiple target types:
  - DB2
  - Oracle
  - Teradata
  - Oracle
  - MQSeries

Figure 3-22 on page 97 illustrates a DTS stage that is used in an ISD job. It significantly reduces the clutter that is associated with the synchronization of the standard database stages. We provide this illustration to give you an overall perspective.

The DTS stage supports an output link, whose table definition can be found in category Table Definitions/Database/Distributed Transaction in the DS repository tree.

When used in ISD jobs, the Use MQ Messaging DTS property must be set to **NO**. Although source and work queues are not present, MQSeries must still be installed and available locally, because it acts as the XA transaction manager.

DTS must be used in ISD jobs only when there are multiple target database types and multiple target database instances. If all SQL statements are to be executed on the same target database instance, a database connector must be used instead.



Queue manager *	QMGR1
Work queue *	
Append node number	No
Use MQ messaging	No
Abort after	0
Order records	No

\Table Definitions\Database\Distributed Transaction\TransactionStatus - Table Definition (Read-Only)

General | Columns | Format | NLS | Relationships | Parallel | Layout | Locator | Analytical information

	Column name	Key	SQL type	Extended	Length	Scale	Nullable	Display	Data element	Description
1	LinkNumber	☐	Integer				No	10	DTS.STATUS.ERF	The link number containing the failure
2	StatusCode	☐	Integer				No	10	DTS.STATUS.STA	Status of the transaction
3	ErrCount	☐	Integer				No	10	DTS.STATUS.COL	Count of the number of rows in error or rejected within this m
4	ErrMessage	☐	VarChar		488		No	488	DTS.STATUS.ERF	The error message from the resource or the text message fr

Figure 3-22 DTS in an Information Services Director job

3.8.5 Information Services Director with connectors

The database connectors in IS 8.5 support units of work involving multiple SQL statements against the same target database instance.

For Connector stages, an output link carries the input link data, plus an optional error code and error message.

If configured to output successful rows to the reject link, each output record represents one incoming row to the stage. Output links are already supported by the Teradata Connector in IS 8.0.1, although that connector is still restricted to a single input link.

Figure 3-23 shows an example of multiple input links to a DB2 Connector (units of work with Connector stage in an Information Services Director job). All SQL statements for all input links are executed and committed as part of a single transaction, for each wave. An EOW marker triggers the commit.

All SQL statements must be executed against the same target database instance. If more than one target database instance is involved (of the same or different types), the DTS stage must be used instead.

The example also depicts multiple input links to a DB2 Connector (units of work with the Connector stage in an Information Services Director job).

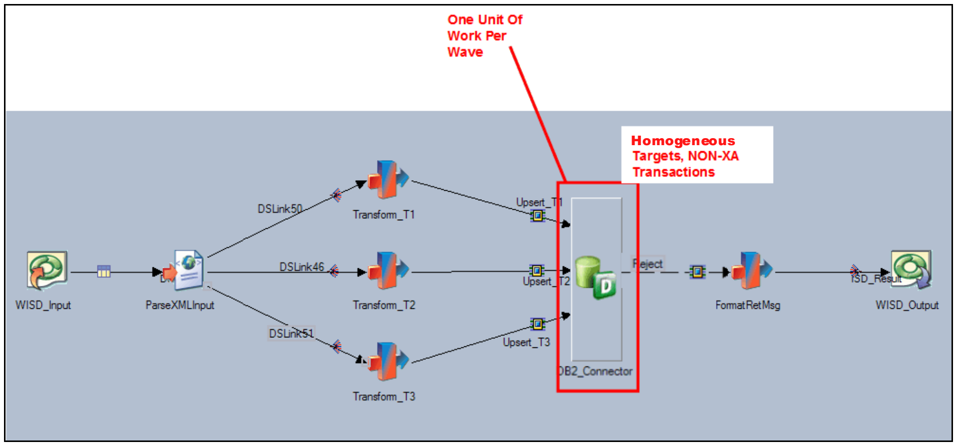


Figure 3-23 Units of work with the Connector stage

3.8.6 Partitioning in ISD jobs

In InfoSphere DataStage parallel jobs, when a multiple node configuration file is used, data can be passed through different paths and at different speeds between the stages of the job. Because of this architecture, the data records that make up an Information Services Director (ISD) service request can arrive at the ISD output stage in a different order than they were received. When this situation occurs, the response data produced by the job for one request can be directed to a different request. Or, the response for one request might contain both its own data plus the data that was meant for a different request. When a multiple node configuration file is used and multiple requests are processed simultaneously, there is no guarantee that the response provided contains the correct data.

This issue only occurs for jobs that are deployed as ISD providers and that meet all of the criteria in the following list:

- The job has an ISD Input stage.
- The configuration file that is used by the job (APT_CONFIG_FILE) contains multiple nodes.
- Multiple requests can be processed by the job at the same time.

To avoid this problem, ensure that your ISD jobs only use a single-node configuration file. You can ensure that your ISD jobs only use a single-node configuration file by adding the environment variable *$APT_CONFIG_FILE* to the job properties at the job level. You then need to set *$APT_CONFIG_FILE* to reference a configuration file that contains only a single defined node. After this change, you must disable this job within Information Services Director, recompile it in DataStage Designer, and re-enable it in the ISD Console.

Alternatively, if all jobs in the project are ISD jobs, you can change the *$APT_CONFIG_FILE* environment variable at the DataStage project level to point to a single node configuration file as the default.

Important: Even when making a project-level change, you must still disable and then re-enable the jobs in the ISD Console. However, recompilation of the job is not necessary.

When setting the variable at the project level, be careful that the individual jobs do not override the project-level default by defining a value locally.

Consider the performance implications. For complex jobs that process larger data volumes, limiting the number of nodes in the configuration file might result in reduced performance on a request-by-request basis. To ensure that overall performance remains acceptable, you can increase the number of minimum job instances that are available to service requests. See 3.8.2, “Scalability” on page 93 for more information about multiple job instance servicing requests, as determined at deployment time through the IS Console.

Proper monitoring must be in place to ensure that hardware resources are not saturated and that enough spare capacity exists to accommodate additonal job instances.

3.8.7 General considerations for ISD jobs

Consider when and how to use ISD with always-on jobs:

- Generally, the best job candidates for service publication with Information Services Director have the following characteristics:
  - Represent fine-grained data integration or data quality functions
  - Are request/response-oriented
  - Have small to medium payloads
  - Are entirely idempotent (can be run over and over again in succession without impact)
  - Do little or no writing
  - Have high watermark traffic requirements that are 300 requests per second or less
- Job sequences cannot be published as services. If you have a business requirement to be able to invoke a sequence of jobs via a service, design one of the following jobs:
  - Create a server job as a wrapper that uses job control logic only (no stages) to invoke an actual DataStage job sequence.
  - Deploy this wrapper job by using an asynchronous binding, such as JMS.

Synchronous binding: Using a synchronous binding (such as soap over HTTP) can result in a timeout if the sequence runs for a long time.

3.8.8 Selecting jobs for publication through ISD

When selecting server or enterprise edition (EE) jobs for publication through ISD, parallel jobs are the preference for the following reasons:

- Parallel jobs support pipeline and partitioning parallelism.
- Parallel jobs must be the preferred engine for any new development.
- They can scale on cluster and grid environments, and they are not restricted to run on the head node only. They reduce the strain on the head node. Therefore, it is less likely to require load balancing across multiple service providers (DS engines).

- The transaction support enhancements that are described in 3.8.4, “ISD with DTS” on page 96 and 3.8.5, “Information Services Director with connectors” on page 98 overcome the previous limitations with parallel stages that made DS server jobs necessary. The DTS and Database Connector stages in IS 8.5 provide for a similar functionality to a DS server combination of a Transformer stage directly connected to an ODBC stage, for instance.

The following scenarios justify the publication of DS server jobs as ISD services:

- Publishing existing DS server jobs
- Invoking DS sequences:
  - Sequences are in the Batch category and are outside of the scope of this chapter.
  - Sequences cannot be exposed as ISD services.

A DS server job can be created with a transformer that in turn invokes a DS sequence with the utility **`sdk`** routine. The utility **`sdk`** routine is in the Transformer category in the pull-down list of the Transformer. It is called UtilityRunJob.

4

XML integration in DataStage

XML technology is one of the most versatile methods for data interchange. DataStage provides highly flexible integration capabilities and access to a host of distributed enterprise data sources for industrial applications and shared metadata repositories. Combining DataStage with XML offers opportunities to develop powerful, industry-leading solutions that support enterprise initiatives, such as customer relationship management (CRM) and decision support systems (DSS).

DataStage was conceived as an extract, transform, and load (ETL) product in the year 1996, the same year XML was adopted as a World Wide Web Consortium (W3C) standard. Since then, it constantly evolved, adding new features and capabilities and remaining a leading ETL tool.

Today, DataStage offers exceptional capabilities, such as the ability to connect to many objects, including a wide variety of enterprise data sources (relational, semi-relational, non-relational, and existing data):

- External applications, such as SAP
- Services, such as Web services
- Message queues, such as WebSphere MQ
- Other tools, such as InfoSphere Change Data Capture

With the integration of QualityStage, which can be installed into DataStage as an add-module, DataStage also provides in-stream data cleansing functionality. This function eliminates the need for separate business processes for data standardization and cleansing activities.

In addition, DataStage can expose user jobs as service-oriented architecture (SOA) services, which allows external applications to use its services. InfoSphere Master Data Management (MDM) Server and InfoSphere Metadata Workbench (MWB) are examples of powerful applications that use the DataStage SOA capabilities to provide value to the enterprises. The connectors and adapters that are supported in DataStage can connect to external data sources and also provide real-time data integration capabilities. The ability to flow incoming and outgoing data by the use of XML Stage provides the ultimate flexibility for handling large complex data and enhances the value of DataStage as an ETL tool.

In this chapter, we review the available XML functionality in DataStage and how it evolved over time. In the next several chapters, we explain the XML Stage, its components, and how to work with it in detail. In the last two chapters, we describe performance issues when handling large, complex XML documents and designing XML-based solutions with DataStage.

4.1 DataStage components for handling XML data

DataStage relies on multiple types of components to access a wide variety of enterprise applications and data sources, including XML. These components can be divided into major groups, with the following terminology:

- Application programming interfaces (APIs)

  APIs facilitate calling a DataStage job from an external application, such as a Java program.

- Plug-Ins and Operators

  Plug-ins and Operators are DataStage stages that are optimized for their corresponding execution environments in Server and Parallel jobs.

- Stages

  The stage components are the basic building blocks of a DataStage job. Stages can be Built-in, Operators, Connectors, or Plug-Ins. Additional stages that are not shipped with the product can be installed individually, developed by the user, or included in Enterprise Packs.

- Packs and Enterprise Packs

  Packs are special purpose components that consist of one or more stages, often including specialized utilities for metadata import or other tasks. Enterprise packs offer advanced features for accessing enterprise-wide tools, such as Web services, InfoSphere Change Data Capture, and XML.

- Connectors

  Connectors implement a common framework that can be used in DataStage, Information Analyzer, and FastTrack, with a common connectivity framework that also provides the capability to integrate with future tools.

We focus on the history of the implementation of XML technology, which evolved with the standards of XML and with the connectivity component architecture that is available to DataStage.

4.2 The XML Pack

The following sections describe the various XML Packs.

4.2.1 XML Pack Version 1.0

The first generation of XML capabilities in DataStage, XML Pack 1.0, included the XML Reader and XML Writer stages, which provided read and write access to XML data. These stages were Built-in stages within Server jobs only. The primary purpose of these components was to provide basic data conversion between hierarchical XML and flat relational formats. XML Pack 1.0 relied on Document Type Definition files (DTD files) for metadata import of document structure, and the transformation capability was limited. As XML standards evolved, XML schema led the industry as the method for the encapsulation and validation of document structure. DataStage evolved with the industry, creating a second generation of XML processing components, XML Pack 2.0. The two packs coexisted for a few releases, but XML Pack 1.0 was eventually deprecated in favor of the more feature-rich and capable newer version.

4.2.2 XML Pack Version 2.0

XML Pack Version 2.0 was implemented with the Plug-in architecture and was available for use in both Server and Parallel jobs. Version 2.0 of the pack added support for XML Schema Definition (XSD) schemas, schema validation, and Extensible Stylesheet Language Transformation (XSLT) capabilities. It included the XML Input, XML Output, and XML Transformer stages. It used industry standard Xerces parser and Xalan XSLT engine technology, and the XPath standard for identifying XML structure when parsing or composing documents. XML Pack 2.0 is available in all currently supported DataStage releases that are available at the time of writing this book.

Although the schema was used for validation at run time, and for metadata import of the XPath and namespace information, the job developer needed to manually design and configure the stage to ensure that the XML documents were created to match the schema that was used. Additionally, when validation was used, the schema must be accessible at run time either on disk or online through a URL. The next generation of XML technology improves its performance and eliminates these restrictions.

4.2.3 XML Pack Version 3.0

The third and current generation of XML capabilities in DataStage, XML Pack 3.0, includes a single XML Stage, which can be used as a source, target, or transformation. Therefore, XML Pack 3.0 consolidates the functionality that previously required three separate stages. The XML Stage adds internal schema library management functionality that offers superior support for complex schemas and advanced capabilities to simplify parsing and composing complex XML documents.

In addition to the simplified single-stage design, the XML Stage is a Connector stage, which is built on the common connectivity framework that was introduced with Information Server. One of the exceptional features of the common connectivity framework is “large object pass by” reference support. This feature allows large chunks of text to be sent from one Connector stage to another with minimal resource usage along the way. The connector framework passes only a small reference value through the links. Then, the Connector stage retrieves the large object directly from the source connector at run time when it is accessed.

The XML Stage uses a unique custom GUI interface, which is called the *Assembly Editor*, that is designed to make the complex task of defining hierarchical relationships and transformations as convenient and manageable as possible. Chapter 6, “Introduction to XML Stage” on page 131 describes the Assembly Editor in great detail. The XML Stage also can accept data from multiple input links, and then establish a hierarchical relationship between the two within the Assembly. For small data volumes, this capability can simplify job design requirements and make the jobs easier to maintain. Chapter 10, “DataStage performance and XML” on page 291 describes performance tuning for the XML Stage, including how to manage large volumes of relational data and large XML document sizes.

The most important feature of the XML Pack Version 3.0 is its tight integration with XSD schema. After an XSD schema is imported by using the new Schema Library Manager, the metadata from within the schema is saved locally. This decoupling means that the schema files do not have to be accessible at run time. This feature is important when deploying into restricted production environments. The transformation and mapping functions directly link to the schema, so that the job developer can focus on the business logic. The stage ensures that the document conforms to the schema.

In the remaining chapters in this book, we fully explain the new XML Pack 3.0 functionality and performance tuning. We explain how to implement XML-based solutions and migration strategies from XML Pack 2.0.

4.3 Web Services Pack

The Web Services Pack facilitates real-time event-driven integration capabilities in DataStage. It provides three stages for integrating Web Services (with SOAP over HTTP protocol) into DataStage jobs:

- The WSClient (source) stage
- The WSClient (target) stage
- The WSTransformer (in-line transformation) stage

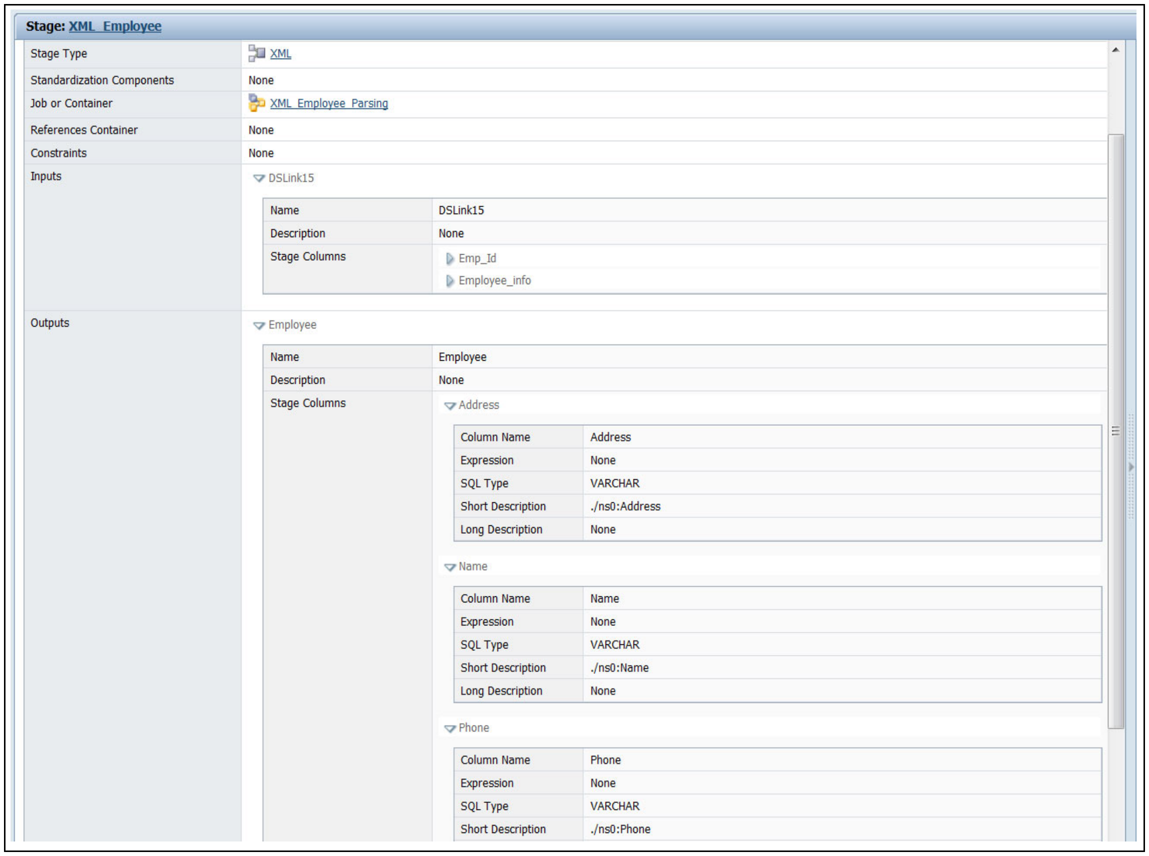
With these stages, you can supply input data to call a web service operation, and retrieve the response as rows and columns. SOAP is an XML-based standard. SOAP messages are XML documents that adhere to the SOAP schema.

The WSClient and WSTransformer stages can directly convert from relational to SOAP XML, for simple structures. For more complex structures, with repeating groups, the XML Stage is used in combination with the Web Service Pack stages to compose or parse the SOAP message body or header, as needed. See 11.1, “Integration with Web services” on page 330 for a complete solution to implement a web service scenario with a complex structure in the SOAP message.

In 3.7, “Real-time applications” on page 79, we described real-time integration with message queues and distributed transactions. DataStage contains an MQ Connector stage for use as a source or target for IBM WebSphere MQ message queues or Java Message Services (JMS) topics. The MQ connector can parse relational data into rows. It also delivers the message contents as a single payload value, which is useful for handling XML documents. See 11.3, “Event-driven processing with IBM WebSphere MQ” on page 342 for a description of using the MQ Connector in conjunction with the XML Stage to process XML documents. This description includes important configuration steps for the MQ Connector for job developers to implement this type of solution successfully.

4.4 DataStage, Information Server, and XML metadata

In Chapter 3, “Overview of IBM InfoSphere DataStage” on page 45, we introduced InfoSphere Information Server and described its unified metadata infrastructure and the Metadata Workbench component. With Metadata Workbench (MWB), you can explore and analyze technical, business, and operational metadata about sources of data. With MWB, you can view the complete lineage of data fields across jobs, applications, reports, or data warehouses back to source systems, including the types of processing that were used. Data that passes through the XML Stage in your jobs is included in this data lineage sample. In Figure 4-1, we illustrate a sample Data Lineage report that was generated by Metadata Workbench.



Stage: XML_Employee

Stage Type	XML
Standardization Components	None
Job or Container	XML_Employee_Parsing
References Container	None
Constraints	None
Inputs	DSLink15

DSLink15

Name	DSLink15
Description	None
Stage Columns	Emp_Id Employee_info

Outputs: Employee

Name	Employee
Description	None
Stage Columns	Address

Address

Column Name	Address
Expression	None
SQL Type	VARCHAR
Short Description	./ns0:Address
Long Description	None

Name

Column Name	Name
Expression	None
SQL Type	VARCHAR
Short Description	./ns0:Name
Long Description	None

Phone

Column Name	Phone
Expression	None
SQL Type	VARCHAR
Short Description	./ns0:Phone

Figure 4-1 Sample Data Lineage report with XML Stage

InfoSphere Metadata Workbench supports both Data Lineage and Job Lineage reports. *Job lineage* provides information about the flow of data through jobs and physical data sources. It is a high-level view (business view) of the lineage. *Data Lineage* provides much more granular information about the flow of data through stages at the column level and includes XPath information for each of the columns.

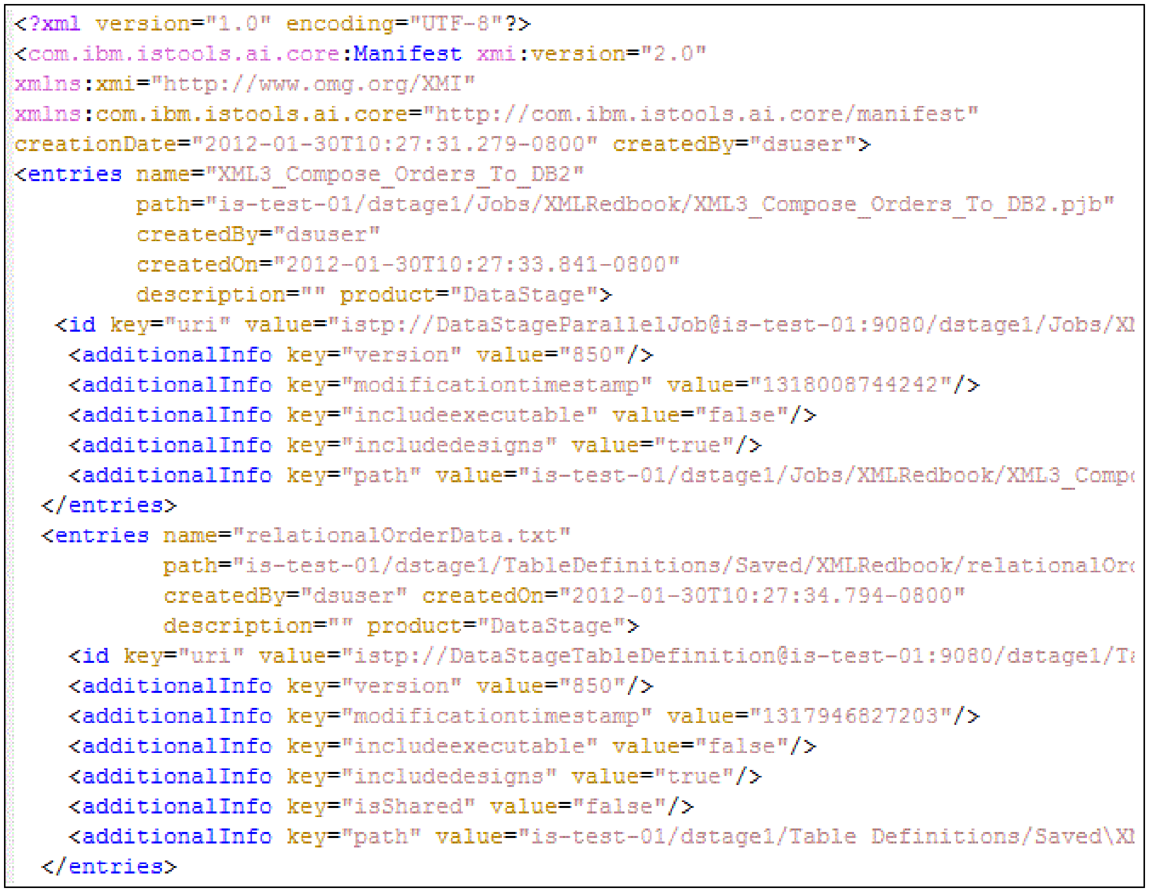
XPath represents one of the steps within the flow of data in either direction of a selected metadata asset. It passes through a series of DataStage stages, including an XML Stage and a sequence of DataStage jobs. It ends in databases or other targets, such as business intelligence (BI) reports.

4.5 Exporting and importing DataStage repository objects

DataStage includes several utilities for exporting and importing repository objects, such as jobs, table definitions, routines, and shared containers to and from physical files. Export and import are used for a number of purposes. With them, you can migrate applications from one environment to another for deployment purposes, for backing up applications, and for upgrades between software releases. In addition to the commonly used DSX file format, XML format is also available. XML is more human readable, making it a better choice if a developer wants to compare the contents of export files before importing them.

The DataStage client includes Windows command-line utilities for automating the export/import processes, when needed. Using the utilities in conjunction with the XML export format was one method of integrating with external source control systems. The built-in source control integration that is now available with Information Server significantly replaces this manual XML-based method.

The Information Server Manager tool and its command-line version **`istool`** provide the source control integration features and robust packaging options to give the developers greater control of how objects are backed up and deployed. The `.isx` file that is created by Information Server Manager is actually a compressed file/jar format. The files that are inside that represent the contents are actually XML files. Of particular interest is the manifest file, which is located within the export file at `../META-INF/IS-MANIFEST.MF`. This file is an XML file, and it contains an index of all the repository objects that are contained within the export file. Figure 4-2 on page 111 contains an excerpt from an `IS-MANIFEST.MF` file.



```
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.istools.ai.core:Manifest xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:com.ibm.istools.ai.core="http://com.ibm.istools.ai.core/manifest"
creationDate="2012-01-30T10:27:31.279-0800" createdBy="dsuser">
<entries name="XML3_Compose_Orders_To_DB2"
         path="is-test-01/dstage1/Jobs/XMLRedbook/XML3_Compose_Orders_To_DB2.pjb"
         createdBy="dsuser"
         createdOn="2012-01-30T10:27:33.841-0800"
         description="" product="DataStage">
   <id key="uri" value="istp://DataStageParallelJob@is-test-01:9080/dstage1/Jobs/Xl
    <additionalInfo key="version" value="850"/>
    <additionalInfo key="modificationtimestamp" value="1318008744242"/>
    <additionalInfo key="includeexecutable" value="false"/>
    <additionalInfo key="includedesigns" value="true"/>
    <additionalInfo key="path" value="is-test-01/dstage1/Jobs/XMLRedbook/XML3_Comp
  </entries>
  <entries name="relationalOrderData.txt"
           path="is-test-01/dstage1/TableDefinitions/Saved/XMLRedbook/relationalOr
           createdBy="dsuser" createdOn="2012-01-30T10:27:34.794-0800"
           description="" product="DataStage">
    <id key="uri" value="istp://DataStageTableDefinition@is-test-01:9080/dstage1/T
    <additionalInfo key="version" value="850"/>
    <additionalInfo key="modificationtimestamp" value="1317946827203"/>
    <additionalInfo key="includeexecutable" value="false"/>
    <additionalInfo key="includedesigns" value="true"/>
    <additionalInfo key="isShared" value="false"/>
    <additionalInfo key="path" value="is-test-01/dstage1/Table Definitions/Saved\Xl
  </entries>
```

Figure 4-2 Excerpt from IS-MANIFEST.MF file within an ISX export

Each object that is contained in the export file includes an <entries> element. The developer can extract the `IS-MANIFEST.MF` file and use this index to identify the contents of the file as needed without running the file through the tool. For information about the Information Server Manager, see the links to the online product documentation in , “Related publications” on page 375.

5

Schema Library Manager

In this chapter, we describe how to use the Schema Library Manager for importing XML schemas into the metadata repository to use in the XML Stage when designing the DataStage job. We also describe how these schemas are grouped into libraries that help in the management of many schemas inside a single project, and the benefits of XML schemas inside a common repository. The first step in configuring the XML Stage is to use the schemas in the repository. In the subsequent chapters, we describe the XML Stage in detail.

The metadata repository stores imported metadata, project configurations, reports, and results for all components of IBM InfoSphere Information Server.

You can access the information in the metadata repository from any of the suite components, which means that metadata can be shared by multiple suite components. For example, after importing metadata into the repository, an authorized user can then use the same metadata in projects in any of the suite components, such as IBM InfoSphere FastTrack, InfoSphere DataStage, or InfoSphere QualityStage.

The XML Stage can be used to read, write, and transform XML data. To read, write, and transform XML files, the XML schema is required. Without a valid schema, you cannot use the XML Stage. The XML schema defines the structure of the XML file. That is, it defines the metadata of the XML file. The XML schema is analogous to the columns that are defined in the Sequential File stage. The columns in the Sequential File stage describe the data that is read by the

Sequential File stage. Similarly, the XML schema defines the data that is present in the XML file.

You must store the XML schema in the metadata repository to use it in the XML Stage while designing the DataStage job. You use the *Schema Library Manager* to import the XML schema into the metadata repository. You access the Schema Library Manager from the DataStage Designer Client by selecting **Import** → **Schema Library Manager**, as shown in Figure 5-1.

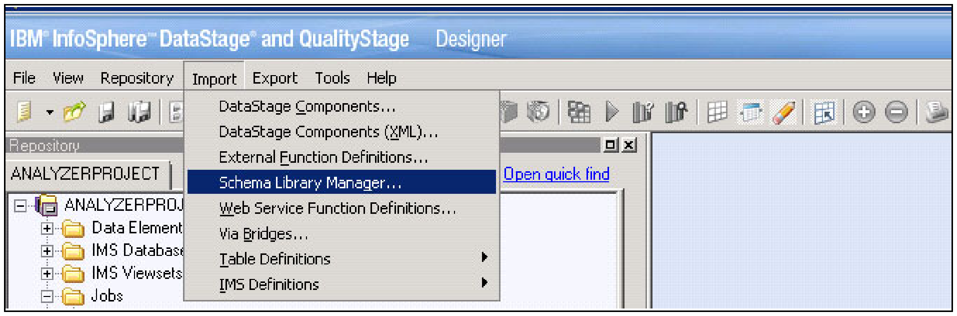

IBM InfoSphere DataStage and QualityStage Designer
File View Repository Import Export Tools Help
DataStage Components...
DataStage Components (XML)...
External Function Definitions...
Schema Library Manager...
Web Service Function Definitions...
Via Bridges...
Table Definitions
IMS Definitions
Repository
ANALYZERPROJECT
ANALYZERPROJ
Data Element
IMS Database
IMS Viewsets
Jobs
Open quick find

Figure 5-1 The Schema Library Manager in the Import menu

5.1 Creating a library

When you open the Schema Library Manager, the Schema Library Manager window opens as shown in Figure 5-2 on page 115. In the Schema Library Manager window, the schema is imported into a *library*. The library is used to organize schemas into a single group. Therefore, to import an XML schema into the Schema Library Manager, you must first create the library.

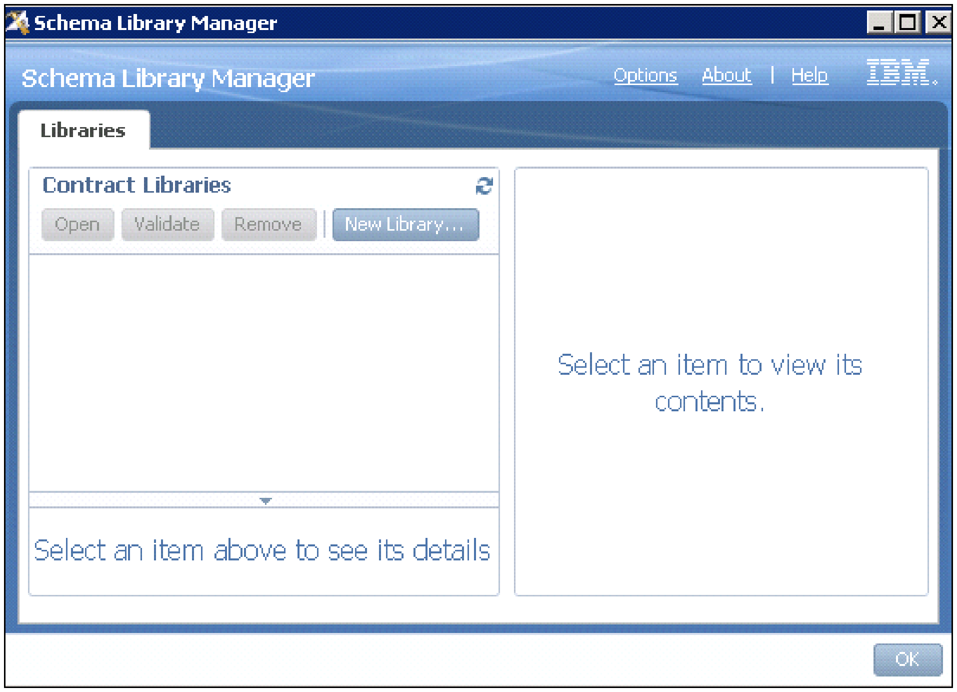


Figure 5-2 Schema Library Manager window

Use the following steps to create a library in the Schema Library Manager:

1. In the Schema Library Manager window, click **New Library**. The New Contract Library window opens, as shown in Figure 5-3.

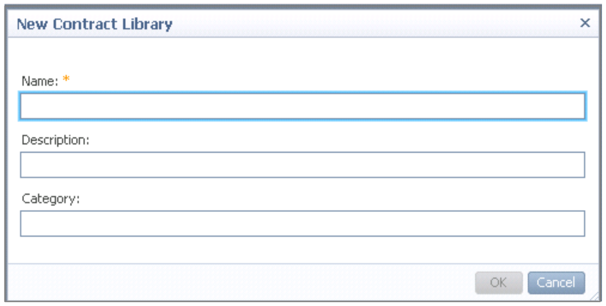


Figure 5-3 New Contract Library window

2. In the New Contract Library window, in the Name text box, enter the name of the library, which can be any valid name that begins with an underscore, letter, or colon. Enter the library name as `Test`, as shown in Figure 5-4 on page 116.

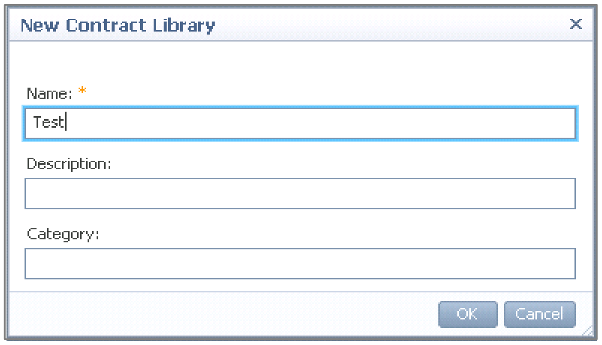


Figure 5-4 Name text box in the New Contract Library window

3. In the Description text box, enter a short optional description of the library. Enter `This is a test library` in the Description field, as shown in Figure 5-5. Click **OK**.

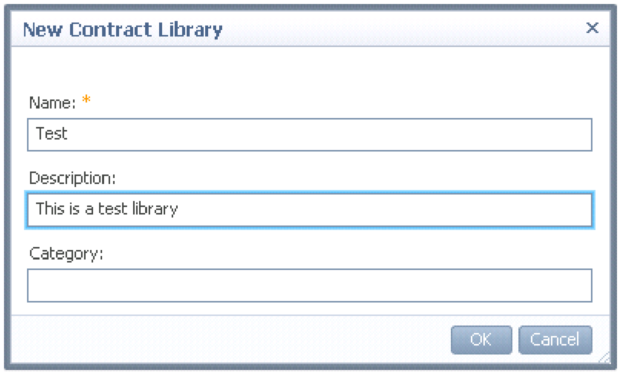


Figure 5-5 Description text box in the New Contract Library window

A library with the name Test is created, as shown in Figure 5-6 on page 117.

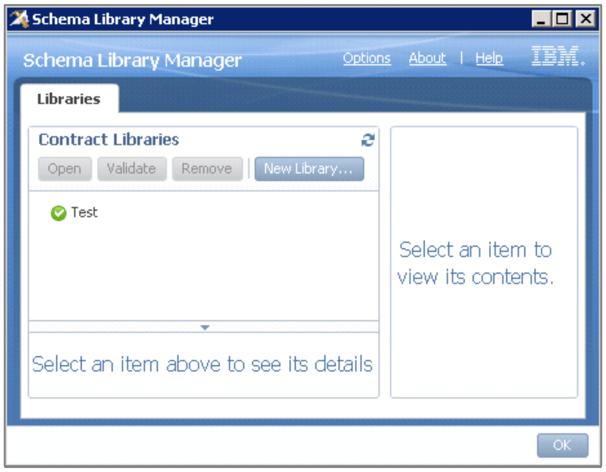


Figure 5-6 The Test library in the Schema Library Manager

Multiple libraries can be created in the Schema Library Manager, but each library name must be unique. Therefore, two libraries cannot have the same name. The names are case-sensitive; therefore, the names `Test` and `test` are unique.

A library can be accessed through all DataStage projects; therefore, our library Test can be accessed through all DataStage projects.

5.2 Importing an XML schema into a library

After creating a library, you can import an XML schema into it. Use the following steps to import the schema `employee.xsd` into the library Test:

1. Select the library **Test**; the Resource View section becomes active, as shown in Figure 5-7 on page 118.

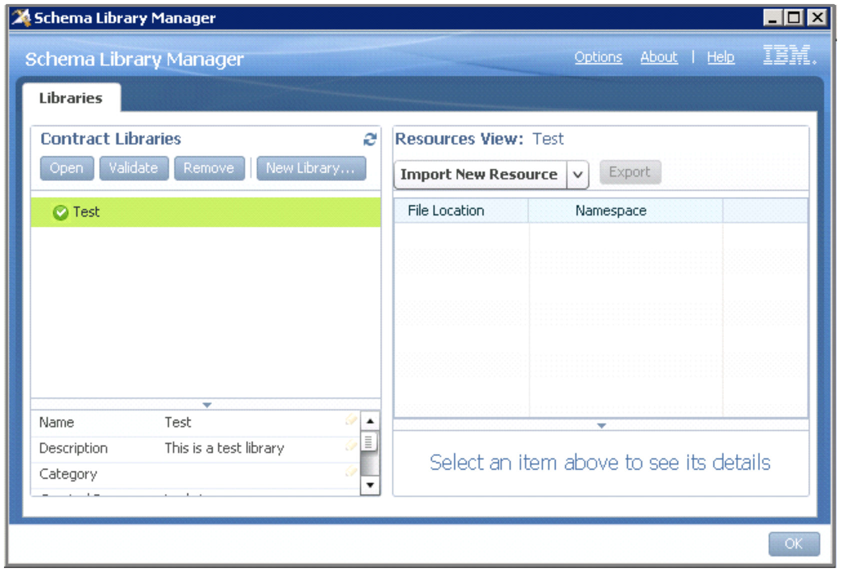
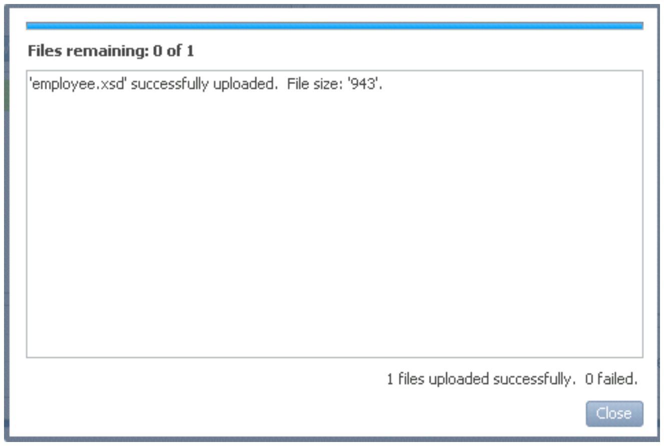


Figure 5-7 The Resource View section in the Schema Library Manager

2. Click **Import New Resource**. You can browse through the folders in the client machine. Select the schema **employee.xsd** and click **Open**. The `employee.xsd` file is imported. You are notified through the window, as shown in Figure 5-8. Click **Close**.



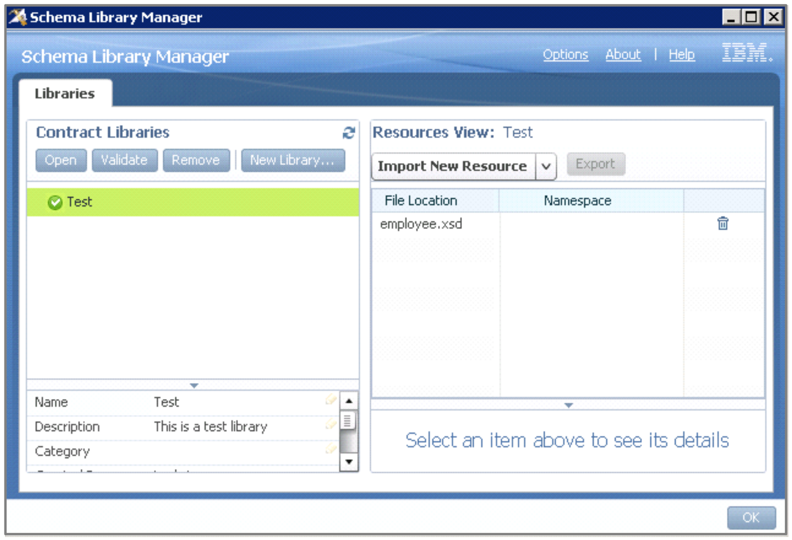
Files remaining: 0 of 1

'employee.xsd' successfully uploaded. File size: '943'.

1 files uploaded successfully. 0 failed.

Figure 5-8 The notification window while importing a schema into the library

The `employee.xsd` is imported into the library Test, as shown in Figure 5-9 on page 119.



File Location	Namespace
employee.xsd	

Name	Test
Description	This is a test library
Category	

Select an item above to see its details

Figure 5-9 Schema file is imported into the Schema Library Manager

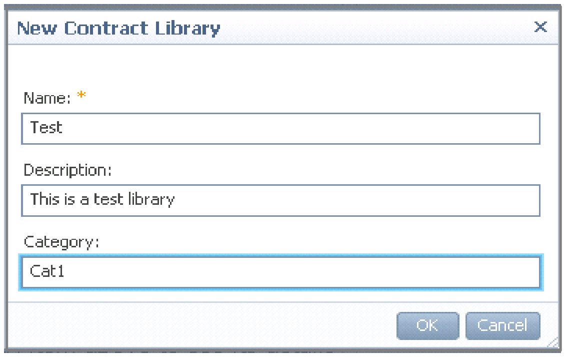
Two schemas with the same name cannot be imported into the same library. But, two libraries can hold the same schema. Therefore, you can have the same set of schemas in two separate libraries.

5.3 Categorizing the libraries

A library in the Schema Library Manager can be added to a *category*. A category is used to organize multiple libraries into a single group.

A library can be added to a category in two ways: while creating the library and after creating the library. Follow these steps:

1. When you create a library, you can enter a name in the Category text box, as shown in Figure 5-10 on page 120.


New Contract Library

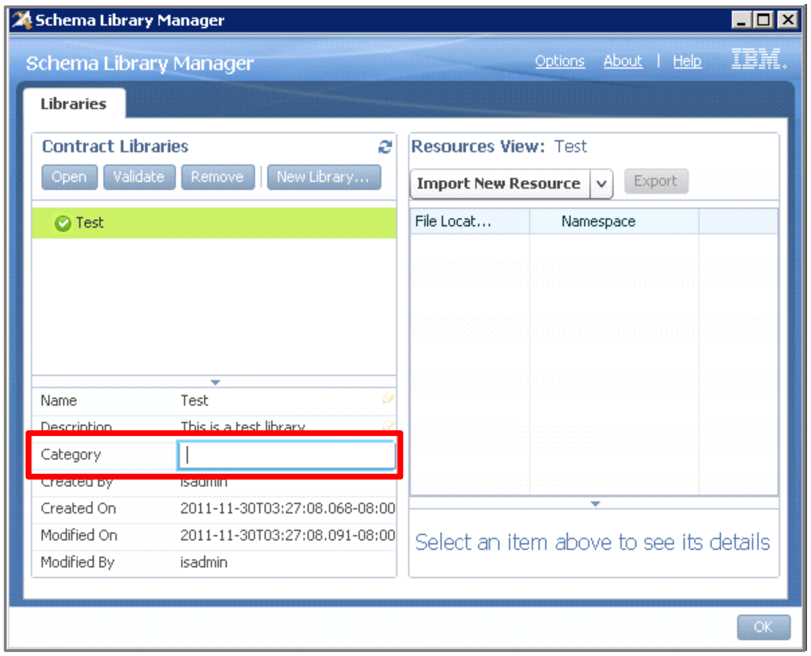
Name: *
Test

Description:
This is a test library

Category:
Cat1

Figure 5-10 The Category field in the New Contract Library window

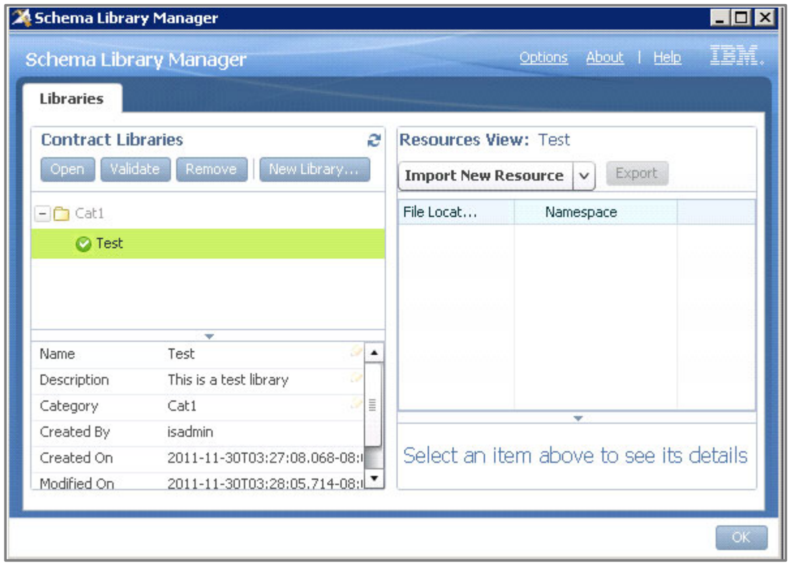
2. If the library is created without a category, the library can still be categorized. To categorize an existing library, select the library. The details of the library are listed, as shown in Figure 5-11. You can enter a category name in the Category text box.



Name	Test
Description	This is a test library
Category	
Created By	isadmin
Created On	2011-11-30T03:27:08.068-08:00
Modified On	2011-11-30T03:27:08.091-08:00
Modified By	isadmin

Figure 5-11 The Category option for an existing library

When you add the library Test to the Category Cat1, you create a folder called Cat1. The Test library is in the folder, as shown in Figure 5-12 on page 121.



Name	Test
Description	This is a test library
Category	Cat1
Created By	isadmin
Created On	2011-11-30T03:27:08.068-08:
Modified On	2011-11-30T03:28:05.714-08:

Figure 5-12   The library Test in the category Cat1

You cannot have two categories with the same name; each category must have a unique name. The names are case-sensitive; therefore, the names `Cat1` and `cat1` are unique. Two categories cannot have the same library. For example, the library Test cannot exist in two separate categories.

5.4 Viewing the XML schema structure

When an XML schema is imported into the Schema Library Manager, it is translated into a simplified model. You can see this simplified structure of the XML schema in the Schema Library Manager. For example, the schema `employee.xsd` is imported into a library Test. To see the structure of the schema, double-click the library **Test,** or select the library **Test** and click **Open**. The Types tab opens, as shown in Figure 5-13 on page 122.

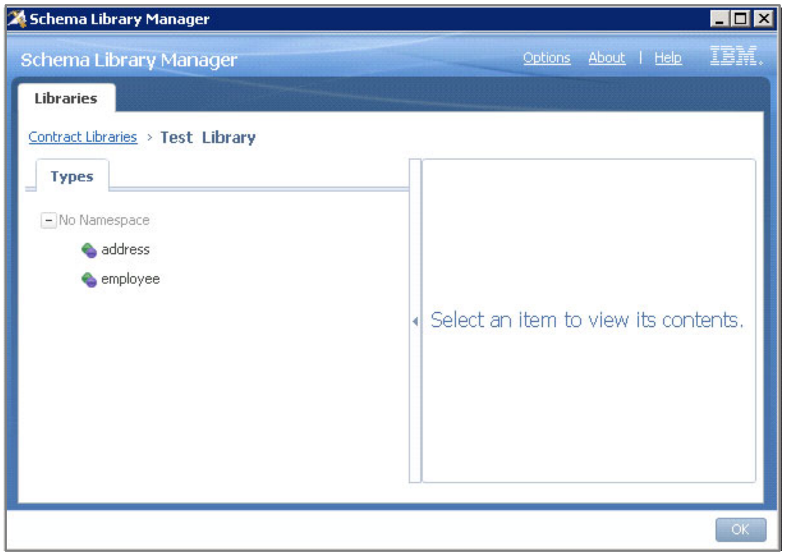


Figure 5-13 The Types tab in the Schema Library Manager

The global elements of the schema shows on the Types page. In `employee.xsd`, there are two global elements, as shown in Figure 5-13. The elements do not have a namespace; therefore, the namespace is not shown.

When you click the global element **employee**, you can see its structure on the right side of the window, as shown in Figure 5-14 on page 123.

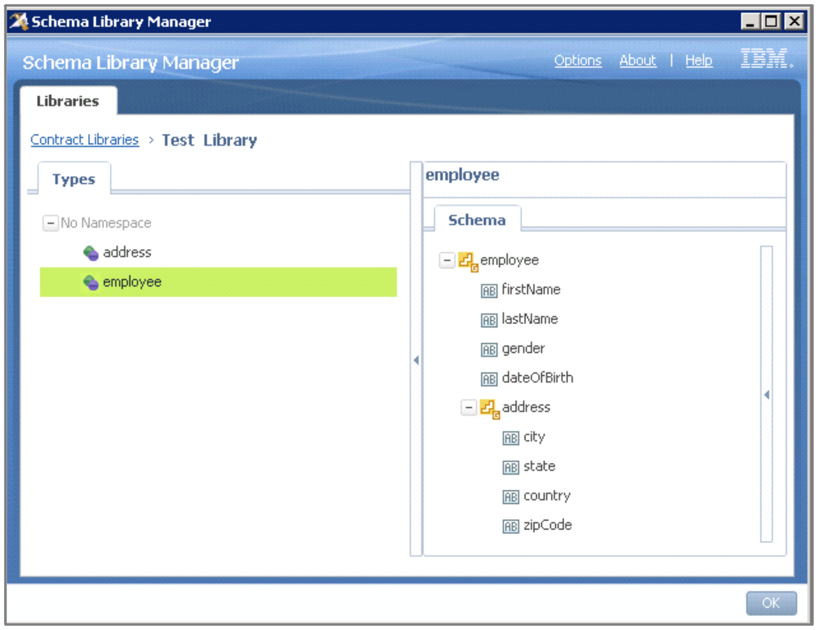


Figure 5-14 Structure of the element employee

5.5 Importing related schema files

Multiple schemas can relate to one another; therefore, one schema can reference the elements and complexTypes that are used in another schema. Schema files can relate to each other by an INCLUDE or IMPORT statement. Reference other schema files through the Schema Location (INCLUDE) or the Namespace (IMPORT) of the schema file. Such references must be resolved within a single schema library. Also, you need to include all files that are referenced by a schema file in the same library.

5.5.1 IMPORT statement

Two schemas can reference one another by using the IMPORT statement. The IMPORT statement uses the namespace as the ID of the schema file. Because both the schemas related to one another, both the schemas must be imported into the same library in the Schema Library Manager. The Schema Library Manager automatically relates both the schemas to create the simplified schema structure. No user intervention is required. If you import only one schema, the library shows an error.

We show an example of the two schemas that relate to each other by using the IMPORT statement in Example 5-1.

Example 5-1 Schemas that use the IMPORT statement

```
--<?xml version="1.0" encoding="UTF-8"?>
   <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://ibm.com/infosphere/xml/Organization"
   xmlns:prn="http://ibm.com/infosphere/xml/Employee">
   <xs:import namespace="http://ibm.com/infosphere/xml/Employee"
   schemaLocation="import_schema2.xsd"/>
    <xs:element name="departments">
      <xs:complexType>
      <xs:sequence>
          <xs:element name="department" minOccurs="1"
   maxOccurs="unbounded">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="departmentId" type="xs:string"/>
                <xs:element name="departmentName" type="xs:string"/>
                <xs:element ref="prn:employees"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
      </xs:sequence>
      </xs:complexType>
    </xs:element>
   </xs:schema>
```

```
   <?xml version="1.0" encoding="UTF-8"?>
   <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://ibm.com/infosphere/xml/Employee">
     <xs:import namespace="http://ibm.com/infosphere/xml/Organization"
   schemaLocation="import_schema1.xsd"/>
       <xs:element name="employees">
         <xs:complexType>
           <xs:sequence>
             <xs:element name="name" type="xs:string"/>
             <xs:element name="dob" type="xs:date"/>
             <xs:element name="title" type="xs:string"/>
           </xs:sequence>
         </xs:complexType>
       </xs:element>
   </xs:schema>
```

5.5.2 INCLUDE statement

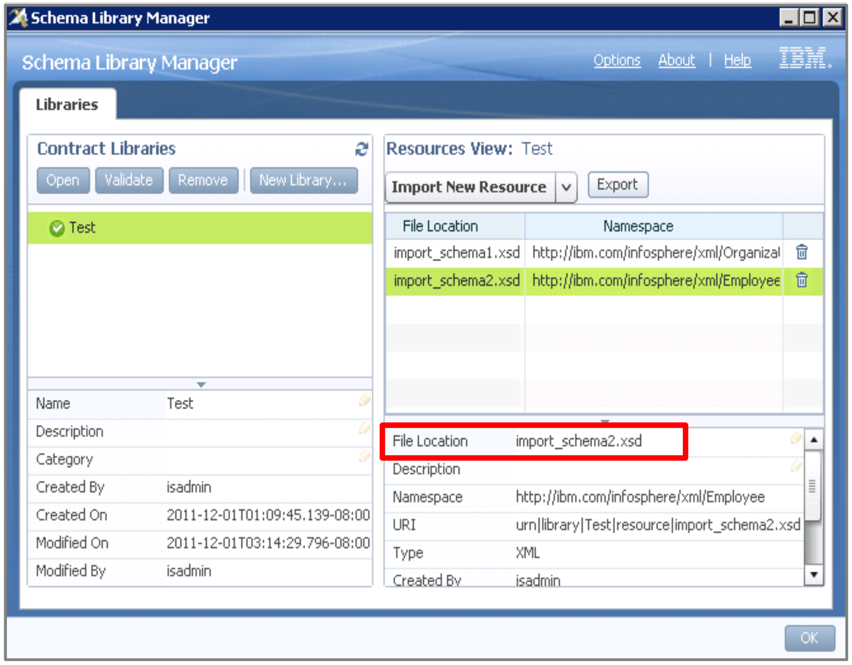
The INCLUDE statement uses the location of the file that relies on a physical location. When the schema files are imported into the Schema Library Manager, the location attribute of each file defaults to the file name. However, the files are commonly referenced by more than their file names. For example, Schema A can reference Schema B by using a relative directory structure:

```
<xs:include schemaLocation="../common/basic.xsd">
```

Or, it can even reference a URL to a web-hosted file:

```
<include schemaLocation ="http://www.example.com/schemas/address.xsd">
```

In these cases, you must modify the file location attribute, as shown in Figure 5-15, of the referenced file in the Schema Library Manager to the location used by the INCLUDE statements.



File Location	Namespace
import_schema1.xsd	http://ibm.com/infosphere/xml/Organizat
import_schema2.xsd	http://ibm.com/infosphere/xml/Employee

Name	Test
Description	
Category	
Created By	isadmin
Created On	2011-12-01T01:09:45.139-08:00
Modified On	2011-12-01T03:14:29.796-08:00
Modified By	isadmin

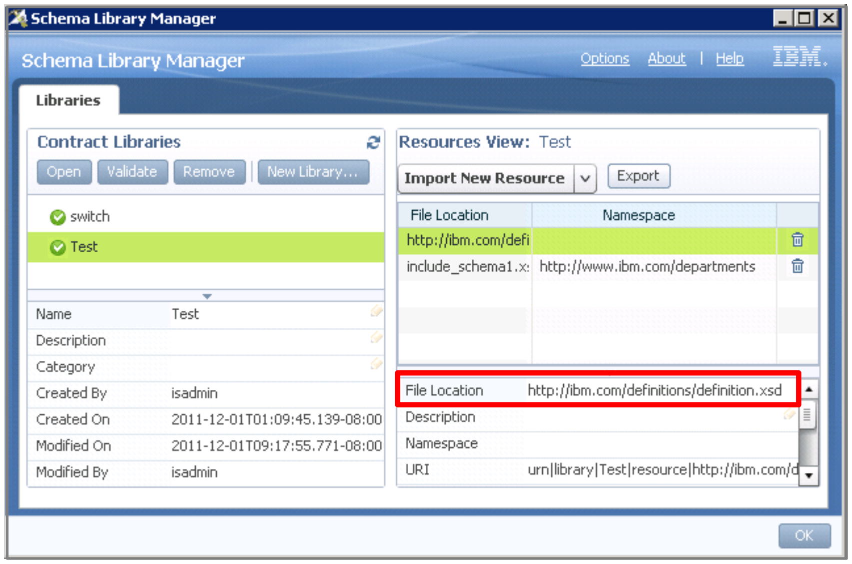
File Location	import_schema2.xsd
Description	
Namespace	http://ibm.com/infosphere/xml/Employee
URI	urn|library|Test|resource|import_schema2.xsd
Type	XML
Created By	isadmin

Figure 5-15 The File Location for an imported schema file

The following steps demonstrate how to import two schema files (`include_schema1.xsd` and `include_schema2.xsd`) with an include dependency between them:

1. Import the schema **include_schema1.xsd** into the library Test.

2. The library shows errors. The error message indicates that the type definition for some of the elements cannot be found, because the definitions are in the schema `include_schema2.xsd`. Import `include_schema2.xsd` into the same library.
3. Even after importing `include_schema2.xsd`, errors are still seen in the library. The location attribute in the `include_schema1.xsd` schema points to the actual physical location of the file, which is `http://ibm.com/definitions/definition.xsd`. However, in the Schema Library Manager, the location attribute defaults to the schema name, for example, `include_schema2.xsd`.
4. To resolve the errors, change the file location of the schema `include_schema2.xsd`, as shown in the following steps:
   a. Click the **include_schema2.xsd** schema in the Resources View window.
   b. In the File Location field, replace `include_schema2.xsd` with `http://ibm.com/definitions/definition.xsd`, as shown in Figure 5-16.



switch	
Test	
Name	Test
Description	
Category	
Created By	isadmin
Created On	2011-12-01T01:09:45.139-08:00
Modified On	2011-12-01T09:17:55.771-08:00
Modified By	isadmin

File Location	Namespace
http://ibm.com/defi	
include_schema1.x:	http://www.ibm.com/departments

File Location	http://ibm.com/definitions/definition.xsd
Description	
Namespace	
URI	urn|library|Test|resource|http://ibm.com/d

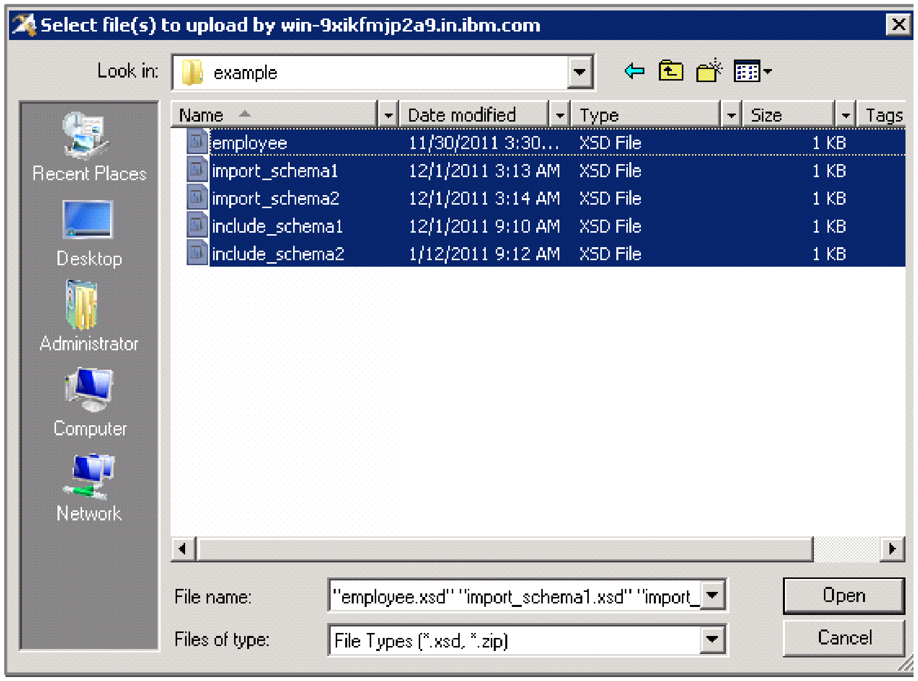
Figure 5-16 Changing file location to match the URL in the INCLUDE statement

   c. When you change the file location name, the library turns green, which indicates that the library is valid.

5.5.3 Importing multiple schemas

Multiple XML files can be imported into a single library by using two methods:

1. When browsing for an XML schema, multiple files can be selected, as shown in Figure 5-17.


Select file(s) to upload by win-9xikfmjp2a9.in.ibm.com

Look in: example

Name	Date modified	Type	Size	Tags
employee	11/30/2011 3:30...	XSD File	1 KB	
import_schema1	12/1/2011 3:13 AM	XSD File	1 KB	
import_schema2	12/1/2011 3:14 AM	XSD File	1 KB	
include_schema1	12/1/2011 9:10 AM	XSD File	1 KB	
include_schema2	1/12/2011 9:12 AM	XSD File	1 KB	

File name: "employee.xsd" "import_schema1.xsd" "import_

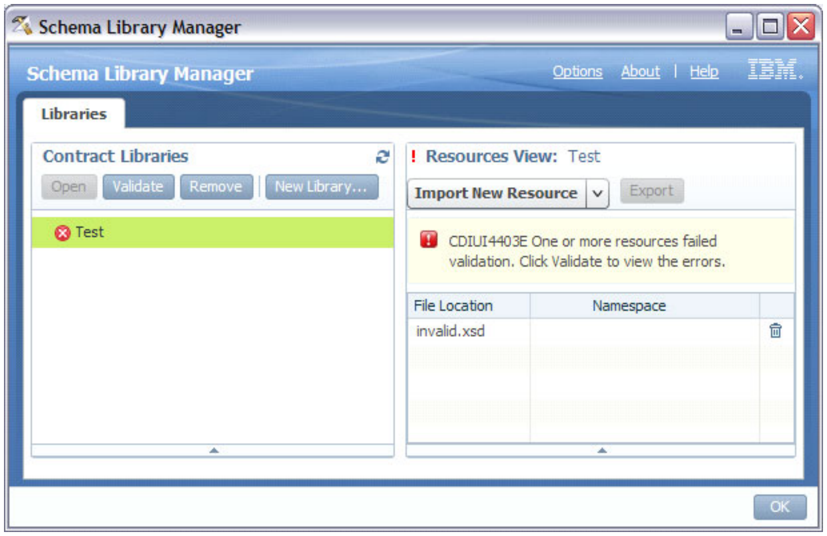
Files of type: File Types (*.xsd, *.zip)

Figure 5-17 Selecting multiple files in the browse window

2. The Schema Library Manager can import an entire compressed file that contains many XSD files. When a compressed file is imported, the Schema Library Manager imports all the files in the compressed file and sets the location attribute to the relative directory from the root of the compressed file. This feature can save the tedious work of importing all the files one at a time and updating their locations.

5.6 Errors while importing schemas

After importing a schema into a library, the library is automatically validated to determine whether the schema contains any errors. If the validation is successful, the library contains all the element and type definitions from the schemas. If the library is invalid, you are notified about the errors. When an invalid schema is imported into the Schema Library Manager, the library shows a red cross to the left of it, as shown in Figure 5-18.



CDIUI4403E One or more resources failed validation. Click Validate to view the errors.

Figure 5-18 The library shows errors

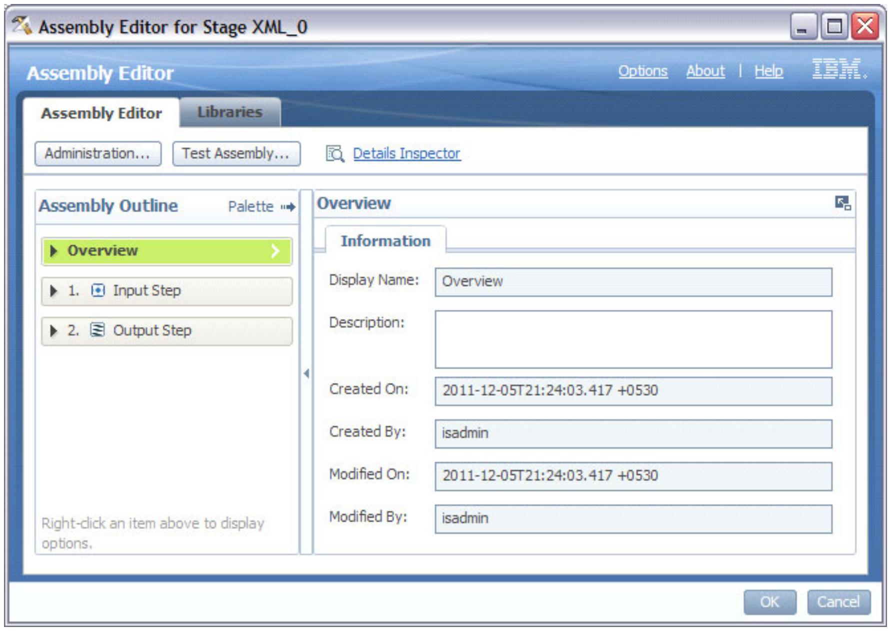
To view the list of errors, click **Validate**. Whenever you modify a schema, delete a schema from the library, or add a schema to the library, the library is automatically revalidated.

Libraries can show the following errors:

- The library shows errors if the XML schema is invalid. For example, the XML schema is missing a closing tag for one of the element definitions.
- Two schema files have the same global elements. When both the schemas are imported into a single library, the library becomes invalid and shows the error "`A schema cannot contain two global components with the same name`". Therefore, two schemas with the same elements cannot be imported into a single library.
- Consider two schemas that relate to each other with the IMPORT statement. If a single schema is imported into the library, the library becomes invalid. For the library to be valid, both schemas must be imported into a single library.
- Two schemas can relate to each other with the INCLUDE statement. When you import both schemas into the same library, the library can still be invalid. To make the library valid, the file name URL must be modified, as described in 5.5.2, "INCLUDE statement" on page 125.

5.7 Accessing the Schema Library Manager from the XML Stage

The XML Stage consists of the *Assembly Editor*. You use the Assembly Editor to design the XML data. You can add steps to read, write, and transform XML data. You access the Schema Library Manager from the Assembly Editor by using the Libraries tab, as shown in Figure 5-19. This tab allows the user to import new schemas without needing to close the Assembly Editor.



Display Name: Overview
Description:
Created On: 2011-12-05T21:24:03.417 +0530
Created By: isadmin
Modified On: 2011-12-05T21:24:03.417 +0530
Modified By: isadmin

Figure 5-19 Accessing the Schema Library Manager through the Assembly Editor

In this chapter, we described how to use the Schema Library Manager to import XML schemas into the metadata repository so that it can be used in the XML Stage while designing the DataStage job. We also described how these schemas are grouped into libraries that help in the management of many schemas inside a single project. We explained the benefits of using XML schemas inside a common repository. Think of putting the schemas in the repository as a first step in configuring the XML Stage. In the subsequent chapters, we describe the XML Stage in detail.

6

Introduction to XML Stage

The XML Stage was introduced in InfoSphere DataStage 8.5 (all previous solutions are called the *XML Pack*). The XML Stage can support hierarchical metadata and perform hierarchical transformations. It is based on a unique state-of-the-art technology to parse and compose any complex XML structure from and to a relational form or a different hierarchical form. Earlier versions of InfoSphere DataStage parsed, composed, and transformed XML data by using three stages, the XML Input, XML Output, and XML Transformer stages. But, the XML Stage can perform all these operations in a single stage.

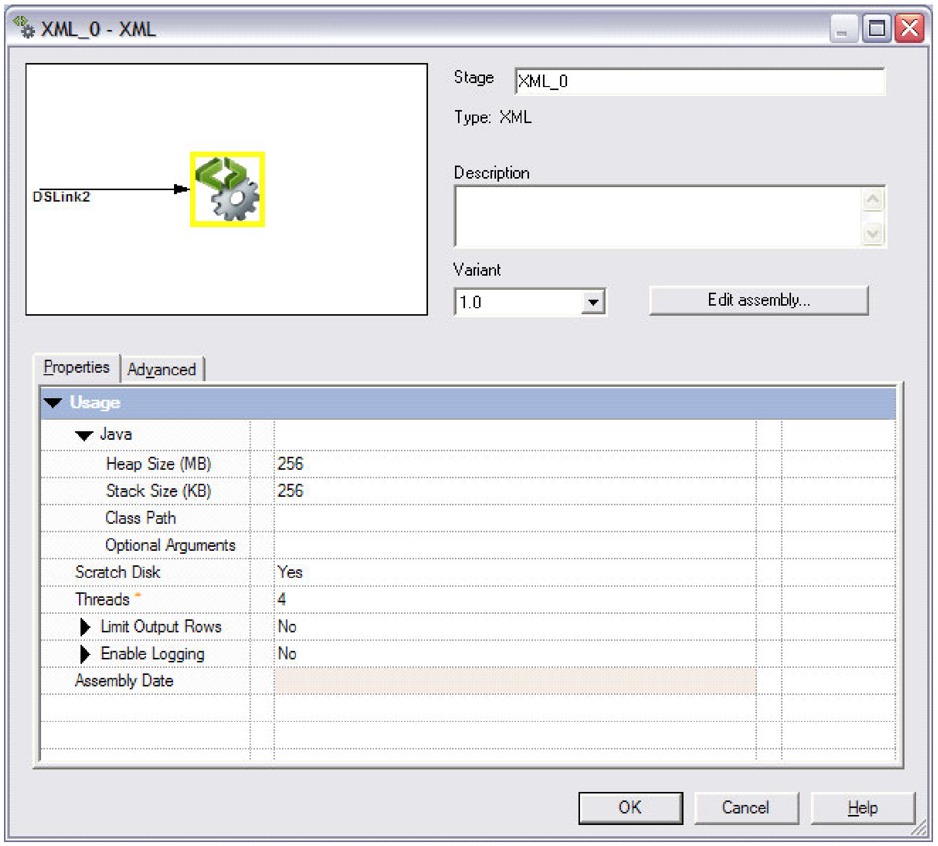
Now, InfoSphere DataStage supports two XML solutions: the XML Pack and the XML Stage. The XML Pack, which includes the XML Input, XML Output, and XML Transformer stages, is useful to perform simple transformations that do not involve a large amount of data. The XML Stage is the best choice for an XML solution, because it can perform complex transformations on a large amount of data.

The XML Stage differs from other known XML tools because of its powerful execution, which can process any file size, with limited memory and in parallel. The XML Stage has unique features, such as the ability to control and configure the level of validation that is performed on the input XML data. Language skills, such as Extensible Stylesheet Language Transformation (XSLT) or Xquery, are not required to use XML Stage.

6.1 The XML Stage

The XML Stage is in the Real Time section of the palette in the DataStage Designer and contains all the components that are required to process any XML document.

The XML Stage consists of the *Stage Editor* and the *Assembly Editor*. The Stage Editor is the first window that opens when you click the stage, as shown in Figure 6-1. The runtime properties of the stage are defined in the Stage Editor.



Stage: XML_0

Type: XML

Description:

Variant: 1.0

Usage	
Java	
Heap Size (MB)	256
Stack Size (KB)	256
Class Path	
Optional Arguments	
Scratch Disk	Yes
Threads *	4
Limit Output Rows	No
Enable Logging	No
Assembly Date	

Figure 6-1 The Stage Editor

Click **Edit Assembly** on the Stage Editor to open the Assembly Editor. You design and create an *assembly* in the Assembly Editor. An assembly consists of a series of steps that enriches and transforms the XML data, as depicted in Figure 6-2 on page 133.

The assembly is analogous to a car factory. In a car factory, there are multiple stations and in each station a new part is added to complete the car. Each station performs only one task, and the car moves along all the stations until it is complete. Similarly, each step in the assembly acts on the incoming data until all the transformations are finished and the assembly is complete.

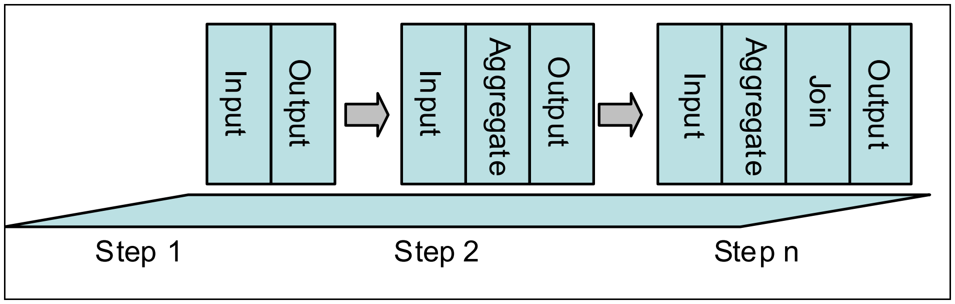
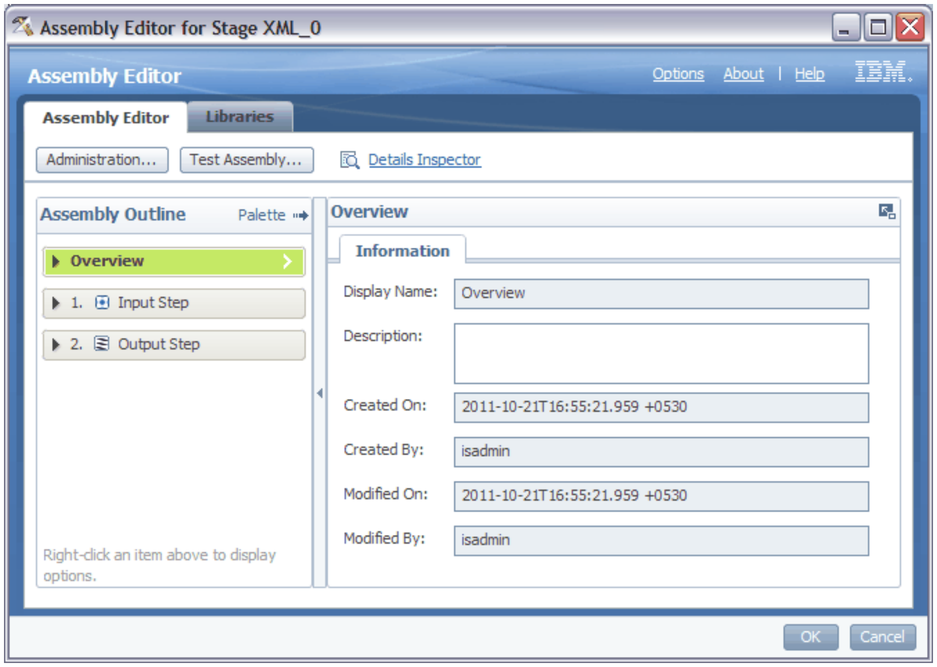


Figure 6-2 The Assembly process

When you open the Assembly Editor, as shown in Figure 6-3, by default, it includes the Input and the Output steps.



Display Name:	Overview
Description:	
Created On:	2011-10-21T16:55:21.959 +0530
Created By:	isadmin
Modified On:	2011-10-21T16:55:21.959 +0530
Modified By:	isadmin

Figure 6-3 The Assembly Editor

The XML Stage contains multiple steps, and each step has a specific functionality. These steps are used together to design a solution. The steps are in the *Palette* in the Assembly Editor. The steps are added between the Input and the Output steps, to parse, compose, and transform XML data. The XML Stage is similar to a DataStage job. In the DataStage job, multiple stages are used to process the incoming data. Similarly, in the XML Stage, multiple steps are used to transform the data.

6.2 Schema representation in the Assembly Editor

The XML Stage is a schema-driven stage. The XML schema is required to parse, compose, or transform an XML file. You cannot start designing the assembly without a valid schema. The schema, that is, the `xsd` file, needs to be imported into the Schema Library Manager so that the schema metadata can be accessed by the steps within the assembly.

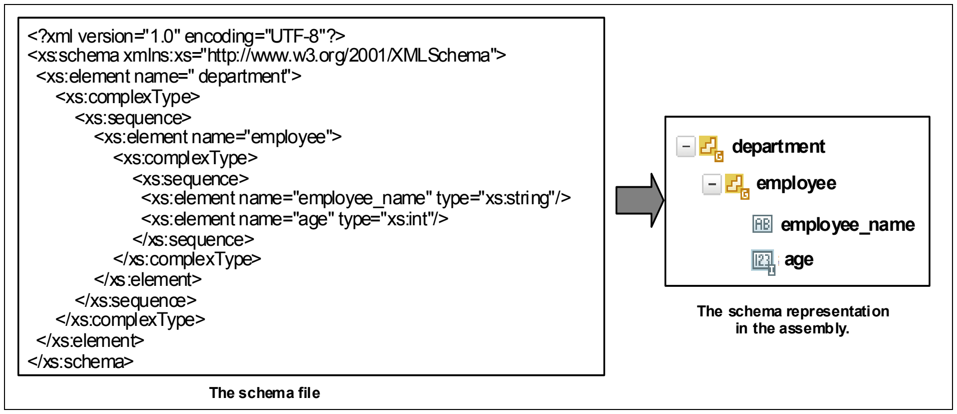
When an XML schema is imported into the Schema Library Manager, it is translated into a simplified model. This simplified model is used by all the transformation steps that are available in the assembly. In this model, the primitive data types are the same primitive types that the XML schema defines. However, when describing complex data, a few of the XML schema concepts are simplified, although all the information from the XML schema is still preserved.

It is important that you understand the schema representation in the assembly. An understanding of the schema representation helps you to understand deeper concepts that are required while designing the assembly. The schema representation is the foundation on top of which the actual building (assembly) is built.

In the following sections, we provide a detailed description of the schema representation in the assembly.

6.2.1 The schema structure

The schema is depicted with a tree structure in the assembly. In the tree structure, the child elements are placed underneath the parent elements. As shown in Figure 6-4 on page 135, `department` is the parent of `employee`. The elements `employee_name` and `age` are the children of the element `employee`. By using the plus sign (+) or the minus sign (-) next to each parent element, the child elements can be hidden or unhidden from view.



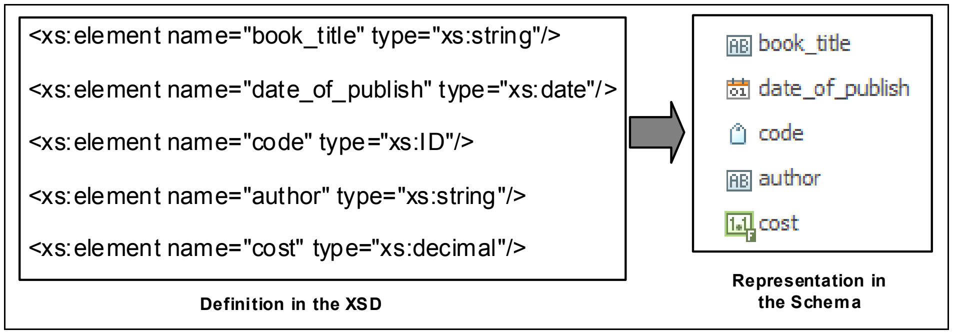
```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name=" department">
    <xs:complexType>
       <xs:sequence>
          <xs:element name="employee">
             <xs:complexType>
                <xs:sequence>
                  <xs:element name="employee_name" type="xs:string"/>
                  <xs:element name="age" type="xs:int"/>
                </xs:sequence>
             </xs:complexType>
          </xs:element>
       </xs:sequence>
    </xs:complexType>
 </xs:element>
</xs:schema>
```

Figure 6-4 Schema representation in the Assembly

This pictorial representation of the schema in the assembly helps you to see the structure of the incoming XML and clearly establish the relationship between various elements within the schema.

6.2.2 Primitive data type representation

A primitive-type element in the XML schema is called an item in the assembly. Therefore, an item in the assembly corresponds to a single primitive data type in the XML schema. In Figure 6-5, we show the representation of a few elements within the assembly. Each element has a unique symbol before it that is based on the data type of the element.

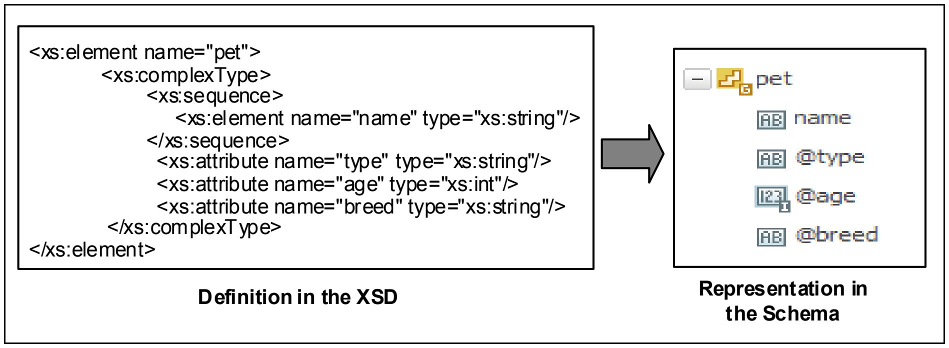


```
<xs:element name="book_title" type="xs:string"/>

<xs:element name="date_of_publish" type="xs:date"/>

<xs:element name="code" type="xs:ID"/>

<xs:element name="author" type="xs:string"/>

<xs:element name="cost"  type="xs:decimal"/>
```

Figure 6-5 Primitive types representation in the Assembly

6.2.3 Attribute representation

The attributes in the schema are represented by an item with a leading at sign (@). As shown by the representations in Figure 6-6, the attribute *type* is represented as an item with the name @type. Therefore, from the representations in the assembly, `name` is an element, and `type`, `age`, and `breed` are attributes.



```
<xs:element name="pet">
        <xs:complexType>
             <xs:sequence>
                 <xs:element name="name" type="xs:string"/>
             </xs:sequence>
               <xs:attribute name="type" type="xs:string"/>
               <xs:attribute name="age" type="xs:int"/>
               <xs:attribute name="breed" type="xs:string"/>
        </xs:complexType>
</xs:element>
```

```
pet
    name
    @type
    @age
    @breed
```

Figure 6-6 Attribute representation in the Assembly

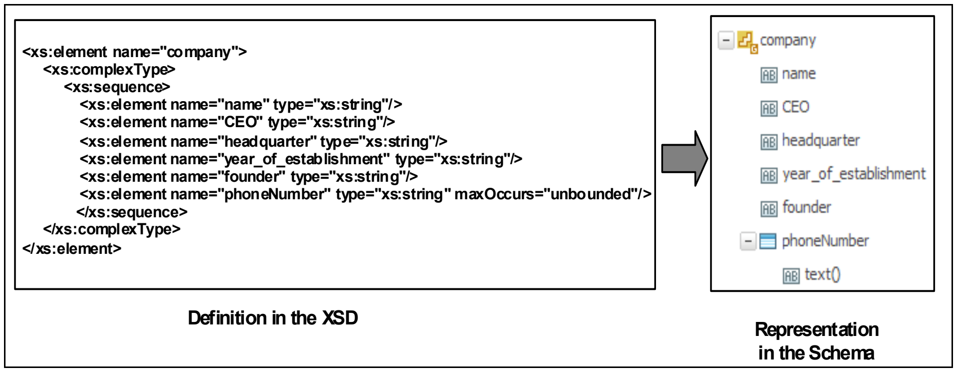
6.2.4 Representation of repeating elements (LIST)

In the XML document, elements might occur more than one time. In the XML schema, these elements show that `maxOccurs` is defined to a value greater than 1, as shown in Example 6-1.

Example 6-1 Repeating elements

```
<xs:element name="employee_name" type="xs:string" maxOccurs="3">
```

These repeating elements of the schema are represented by a *list* in the assembly. As shown in Figure 6-7 on page 137, the element "phoneNumber" can occur multiple times in the XML file. In the assembly, this element is depicted by a blue icon, which represents it to be a list. The list in the assembly holds the actual number of occurrences of the element. The item text() under the list holds the actual data for the element.

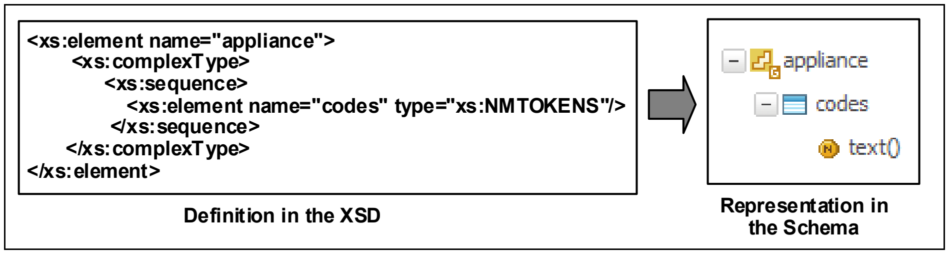


```
<xs:element name="company">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="CEO" type="xs:string"/>
            <xs:element name="headquarter" type="xs:string"/>
            <xs:element name="year_of_establishment" type="xs:string"/>
            <xs:element name="founder" type="xs:string"/>
            <xs:element name="phoneNumber" type="xs:string" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

Figure 6-7 Repeating elements representation in the Assembly

NMTOKENS-type representation

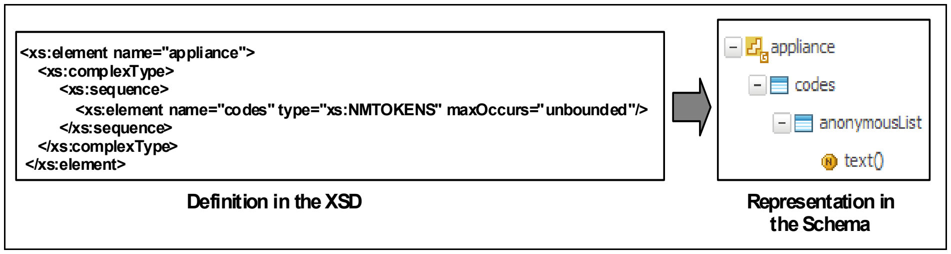
The XML schema can have elements of type NMTOKENS. In these elements, maxOccurs might not be set to a value greater than 1 in the XML schema. In the assembly, these elements are represented by a list, because the data type NMTOKENS represents an array of data separated by spaces. Because multiple data values are held by a single element, the assembly represents this element as a list. As shown in Figure 6-8, the element "codes" is represented by a list with the same name. The actual value of the element is held by the item text().



```
<xs:element name="appliance">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="codes" type="xs:NMTOKENS"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

Figure 6-8 NMTOKENS-type representation in the Assembly

In the XML schema, when an element of type NMTOKENS shows maxOccurs set to a value greater than 1, the element is represented by two successive lists. One list is represented by the name of the element, and the other list is represented by anonymousList (Figure 6-9 on page 138).

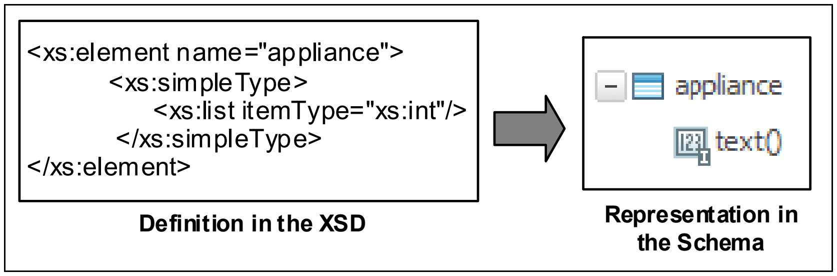


```
<xs:element name="appliance">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="codes" type="xs:NMTOKENS" maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Figure 6-9 NMTOKENS with maxOccurs greater than 1 representation in the Assembly

xs:List representation

An element of type xs:list in the schema is also represented by a list in the assembly, as shown in Figure 6-10.



```
<xs:element name="appliance">
      <xs:simpleType>
         <xs:list itemType="xs:int"/>
      </xs:simpleType>
</xs:element>
```

Figure 6-10 List type representation in the Assembly

6.2.5 DataStage links representation

DataStage input and output links are also represented as a list in the assembly, because each link can carry multiple rows of data during run time. An example of an input link is shown in Figure 6-11 on page 139. The input link DSLink1 links to the XML Stage. The corresponding representation in the Assembly Editor is a list with the following items: name, age, dob, and hireDate. These items are defined columns in the DataStage link DSLink1.

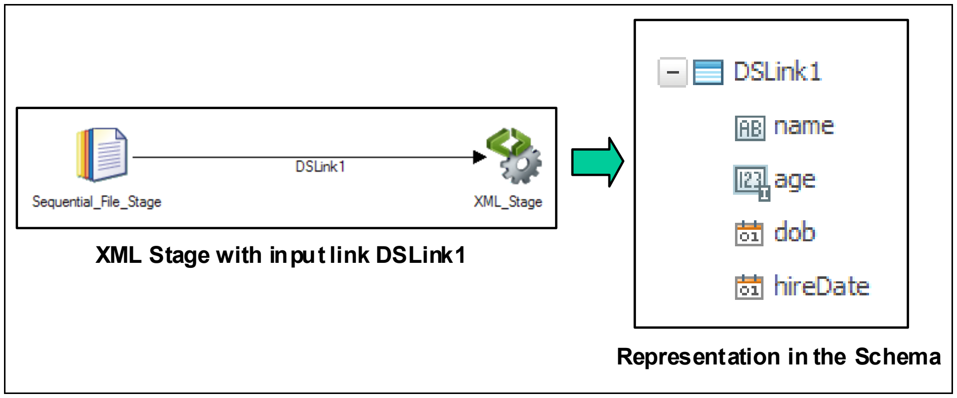
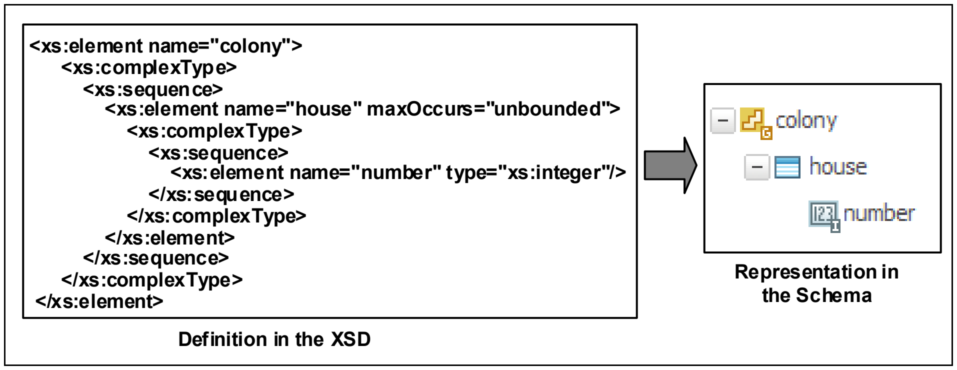


Figure 6-11 Input links representation in the Assembly

6.2.6 Complex-type representation

In the assembly, a *group* is a node in the hierarchy whose only purpose is to create a structure that categorizes or encapsulates information. It does not describe a dimension or value. A group can correspond to a complex element with the type: sequence or choice. In Figure 6-12, the element `colony` is a complex element with type sequence. Therefore, it is represented as a group within the assembly. A group is depicted by a brown icon with a subscript of G.

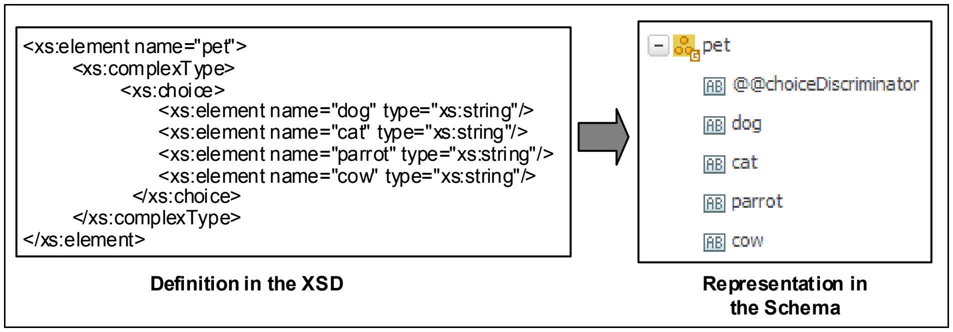


```
<xs:element name="colony">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="house" maxOccurs="unbounded">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="number" type="xs:integer"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

Figure 6-12 Complex-type representation in the Assembly

Specific schema definitions are represented in certain ways in the assembly. We explain several schema representations in the following list:

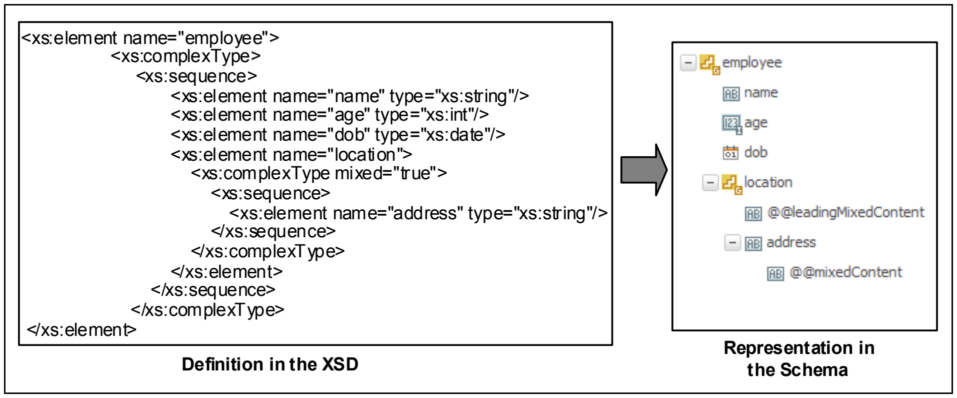
- When a complex element is defined with a choice, an item `@@choiceDiscriminator` is added to the schema representation in the assembly. This item holds the value of the branch that was selected at run time. As shown in Figure 6-13, the element `pet` is a choice. Therefore, the item `@@choiceDiscriminator` is added as a child of the group `pet`.



```
<xs:element name="pet">
        <xs:complexType>
                <xs:choice>
                        <xs:element name="dog" type="xs:string"/>
                        <xs:element name="cat" type="xs:string"/>
                        <xs:element name="parrot" type="xs:string"/>
                        <xs:element name="cow" type="xs:string"/>
                  </xs:choice>
        </xs:complexType>
</xs:element>
```

Figure 6-13 Complex choice type representation in the Assembly

- For an element with mixed content, two new items are added in the assembly: `@@mixedContent` and `@@leadingMixedContent`:
  - `@@leadingMixedContent` captures the mixed content at the start of the element.
  - `@@mixedContent` captures the data that follows the element. As shown in Figure 6-14, the complex element "location" is of mixed type.

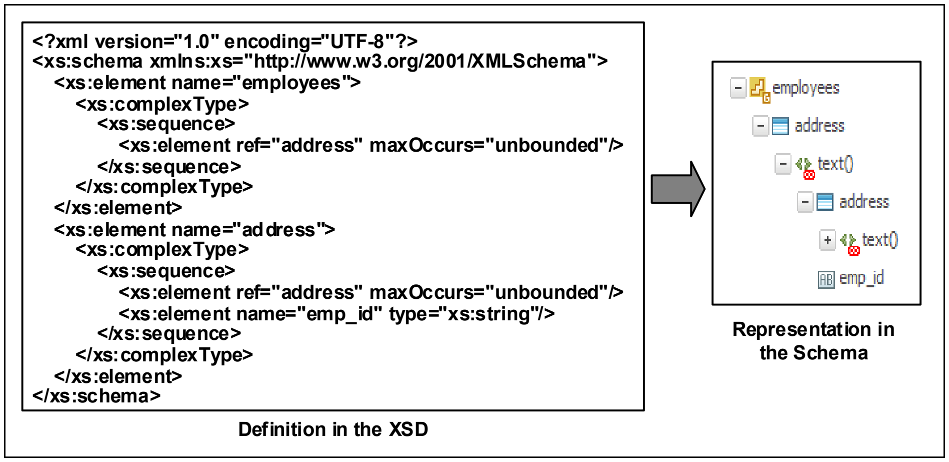


```
<xs:element name="employee">
              <xs:complexType>
                  <xs:sequence>
                        <xs:element name="name" type="xs:string"/>
                        <xs:element name="age" type="xs:int"/>
                        <xs:element name="dob" type="xs:date"/>
                        <xs:element name="location">
                            <xs:complexType mixed="true">
                                <xs:sequence>
                                    <xs:element name="address" type="xs:string"/>
                                </xs:sequence>
                            </xs:complexType>
                        </xs:element>
                    </xs:sequence>
                </xs:complexType>
</xs:element>
```

Figure 6-14 Elements with mixed content representation in the Assembly

6.2.7 Representation of recursive elements

A *recursive element* is represented with the infinity symbol as its suffix. An element is called a recursive element if it refers to itself in the XML schema. In Figure 6-15 on page 141, the element *address* is a recursive element, because its child element also references address.

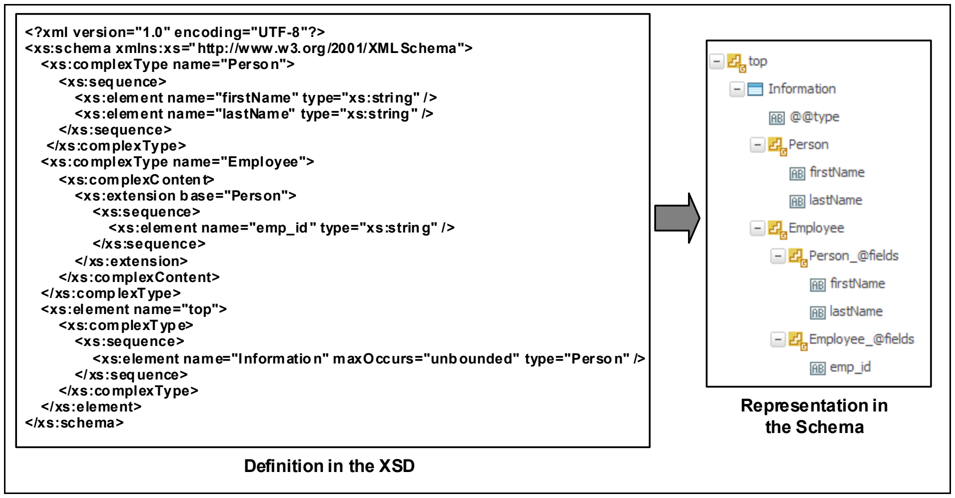


```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="employees">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="address" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="address">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="address" maxOccurs="unbounded"/>
        <xs:element name="emp_id" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 6-15   Recursive element representation in the Assembly

6.2.8 Representation of derived types

When an element is derived from another element, the item `@@type` is added to the schema representation in the assembly. As shown in Figure 6-16, the element `Information` is of type `Person`. Because `Employee` is derived from `Person`, the element `Information` can also be of type `Employee`. Therefore, the Item `@@type` is added under `Information`, so that at run time it can decide which structure to use.

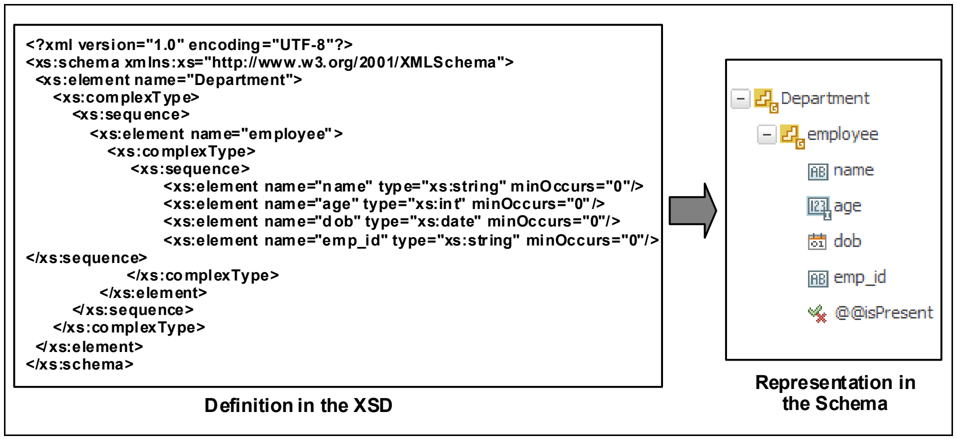


```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="Person">
    <xs:sequence>
      <xs:element name="firstName" type="xs:string" />
      <xs:element name="lastName" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Employee">
    <xs:complexContent>
      <xs:extension base="Person">
        <xs:sequence>
          <xs:element name="emp_id" type="xs:string" />
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:element name="top">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Information" maxOccurs="unbounded" type="Person" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 6-16   Element derivation representation in the Assembly

6.2.9 Representation of an element with all optional elements

An element in the XML schema can have all optional child elements. When an element is optional, minOccurs is set to "0" in the XML schema. In the assembly, the parent element with all optional child elements has an extra item called `@@isPresent` added to it. As shown in Figure 6-17, the element `employee` has all optional child elements. Therefore, the item `@@isPresent` is added as a child of the element `employee`.



```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="Department">
   <xs:complexType>
     <xs:sequence>
       <xs:element name="employee">
         <xs:complexType>
           <xs:sequence>
               <xs:element name="name" type="xs:string" minOccurs="0"/>
               <xs:element name="age" type="xs:int" minOccurs="0"/>
               <xs:element name="dob" type="xs:date" minOccurs="0"/>
               <xs:element name="emp_id" type="xs:string" minOccurs="0"/>
</xs:sequence>
           </xs:complexType>
         </xs:element>
     </xs:sequence>
   </xs:complexType>
 </xs:element>
</xs:schema>
```

Figure 6-17 Optional element representation in the Assembly

When an XML schema is imported into the library, it is translated into a simplified model. All the Transformation steps use these simplified concepts. For example, in the Aggregate step, aggregating an attribute of type `double` is the same as aggregating an element of type `double`. The only steps that are sensitive to the original XML schema concepts are the XML Parser and Composer steps. Due to the ability to preserve the information in the XML schema, these steps can recreate the XML schema concepts and adhere to them, as needed.

6.3 The assembly model

The *assembly* is used to describe the hierarchical data that flows through the assembly steps. The assembly computational model differs from the DataStage job computational model in the following ways:

- The DataStage job design is well-suited for relational data; the assembly is well-suited for hierarchical data.
- In a DataStage job, each link carries a relation. Therefore, if a stage transforms more than one relation, two input links need to be attached to the

stage, and the output behavior is similar. However, in the assembly, a single input can carry multiple lists; therefore, all steps have a single input and a single output, each of which contains multiple lists.

- The assembly steps, similar to the DataStage stages, always iterate input lists. However, unlike DataStage stages, steps iterate lists that exist in any level of the hierarchical input and produce output that is contained in the input list that they iterate. As shown in Figure 6-18, in the DataStage job, Stage3 can perform transformations only on the output of Stage2 and not on the output of Stage1. But in the assembly, Step3 can process the data output of Step1 and Step2. The output of Step1 is not fed to Step3. Therefore, each step can access the output of any other previous step.

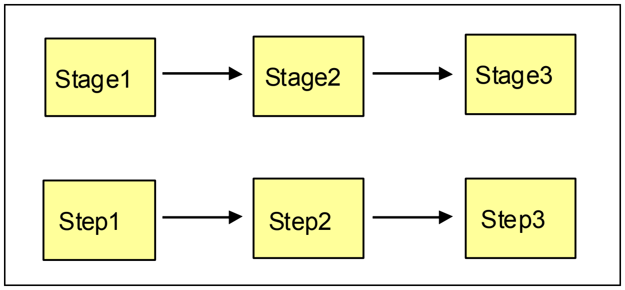


Figure 6-18 Comparison between DataStage jobs and Assembly computational model

The input and output steps of the assembly have a distinguished role. The Input step transforms relational data into hierarchical data, and the Output step transforms hierarchical data to relational data. Each input/output link to the XML Stage turns into a list in the assembly. Each column turns into an item of the list, as shown in Figure 6-11 on page 139.

In the assembly, the entire input tree is rooted in a list named `top`. Therefore, `top` is the parent of all lists and items in the assembly. Consider that the XML Stage has two input links: `DSLink1` and `DSLink2`. As shown in Figure 6-19 on page 144, these links turn into a list in the assembly, and both links are present under the list `top`.

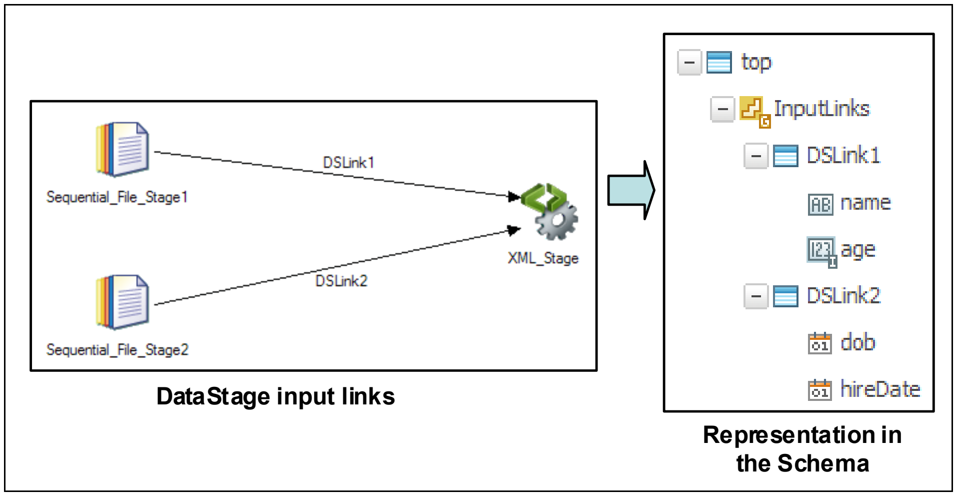
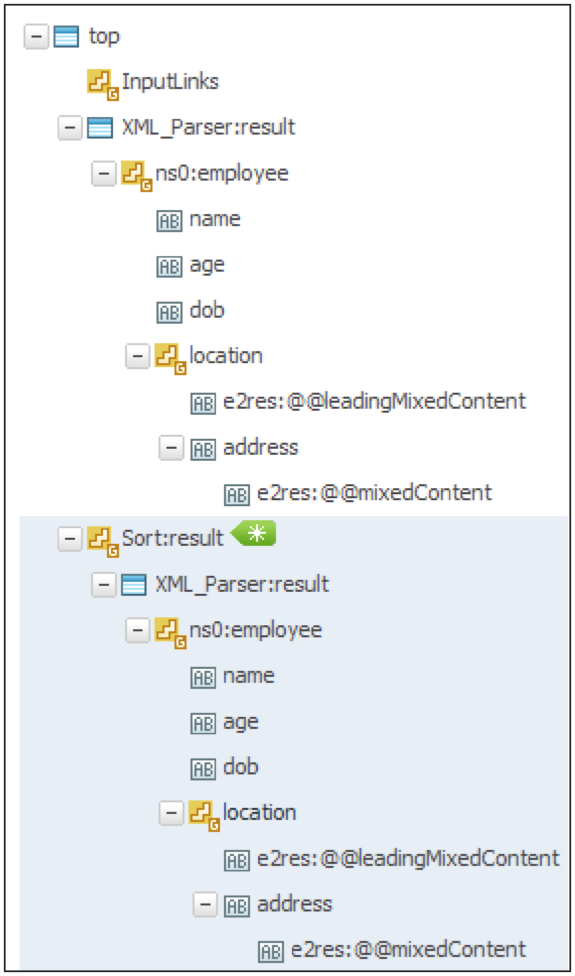


Figure 6-19 Input step representation in the Assembly

Similarly, when multiple steps are used, the input and output of each step is present under the list `top`.

When input links exist to the XML Stage, all the input lists in the assembly are categorized under the group `InputLinks`. This group is the child of the list `top`, as shown in Figure 6-19. Even when no input links exist to the XML Stage, the group `InputLinks` is still present under `top`, but it does not contain any child lists or items.

The steps in the assembly pass the entire input to the output along with an enrichment branch that contains the result of the step. The output of a step is always contained within the top-level list through which the current step iterates. The Assembly Editor presents for each step its input and output trees. The Assembly Editor highlights the enrichment branch that the current step computed, so that the enrichment branch can be distinguished from the original input of the step. As shown in Figure 6-20 on page 145, the output of the Sort step is highlighted by graying out the area.



```
top
    InputLinks
    XML_Parser:result
        ns0:employee
            name
            age
            dob
            location
                e2res:@@leadingMixedContent
                address
                    e2res:@@mixedContent
    Sort:result
        XML_Parser:result
            ns0:employee
                name
                age
                dob
                location
                    e2res:@@leadingMixedContent
                    address
                        e2res:@@mixedContent
```

Figure 6-20 Output step representation in the Assembly

This powerful assembly data model simplifies the transformation description into a series of steps rather than a direct graph.

7 Consuming XML documents

In the scenarios in this chapter, we reference a fictitious healthcare company that we call the *RedbookXML Healthcare Company*. The company, which was established in the late 1990s, receives insurance claims from residents in and around San Jose, California. At the company, a web application is used to capture the claims from their customers, and this data is stored in XML files. At the end of every month, a report is created that summarizes the data for the total number of claims in that month. The application that creates these reports requires the data to be in a database. Figure 7-1 shows this process.

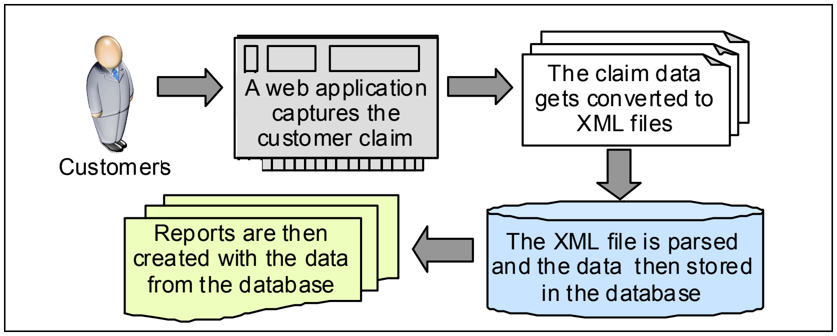

Customers
A web application captures the customer claim
The claim data gets converted to XML files
The XML file is parsed and the data then stored in the database
Reports are then created with the data from the database

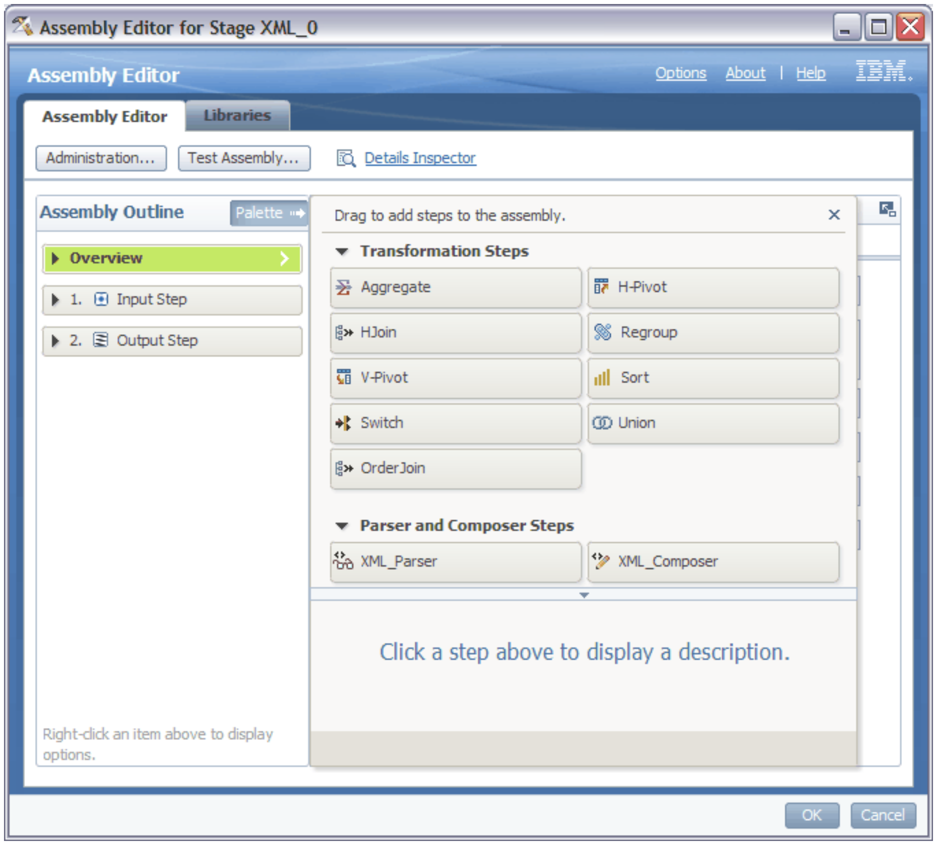
Figure 7-1 Healthcare company claims process

7.1 XML Parser step

Because the company requires data that is resident in a database to generate the monthly reports, the company needed a way to extract the data from the XML files and to store it in the database. The company already uses InfoSphere DataStage for processing and cleansing its customer information. Therefore, the company wants to use the XML stages that are available with DataStage to process the XML claim files.

First, the company parses the XML file with either the XML Input stage or the XML Stage within the DataStage Designer. The company selects the XML Stage, because it can process large XML files easily. Within the XML Stage, the Parser step is used, and it is added to the Assembly Editor. Follow these steps:

1. Open **Assembly Editor** and click **Palette** in the Assembly Outline bar. The steps are shown in the Palette, as shown in Figure 7-2.

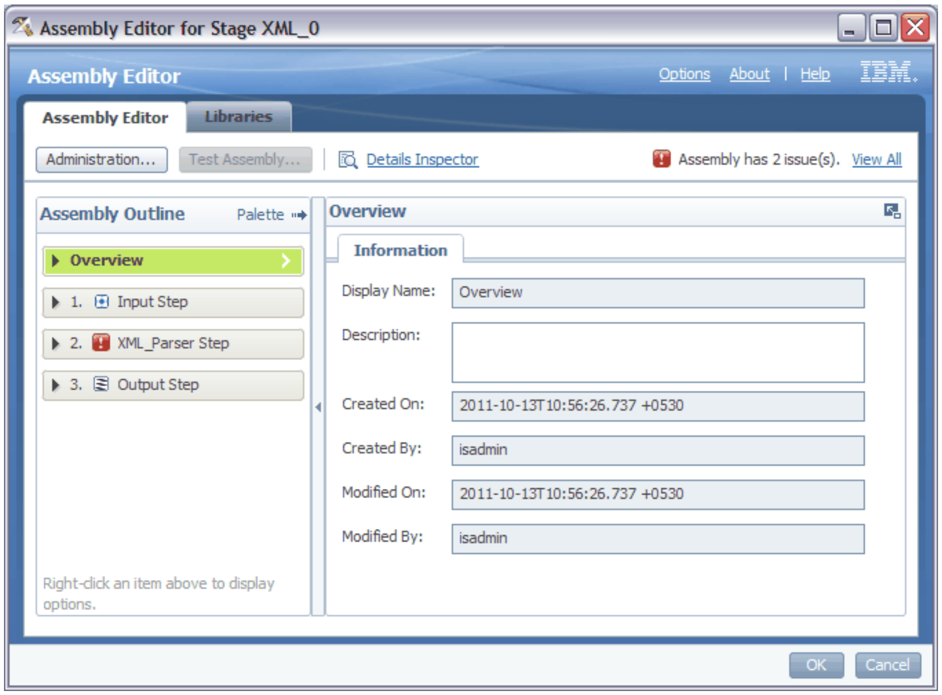


Drag to add steps to the assembly.

Click a step above to display a description.

Figure 7-2 The Palette in the Assembly Editor

2. Double-click **XML_Parser Step** in the palette. The XML_Parser Step is added between the Input Step and the Output Step, as shown in Figure 7-3.


Assembly Editor for Stage XML_0

Assembly has 2 issue(s).

Display Name: Overview

Description:

Created On: 2011-10-13T10:56:26.737 +0530

Created By: isadmin

Modified On: 2011-10-13T10:56:26.737 +0530

Modified By: isadmin

Right-click an item above to display options.

Figure 7-3 XML_Parser Step added to the Assembly Editor

The company stores all the XML files in two central repositories: the database and one of its internal servers. The Parser step must read the XML files from these locations to parse the XML data. In the XML Parser step, the method of reading the XML data must be specified in the XML Source tab. A single Parser step cannot read the XML data from multiple sources at the same time.

7.1.1 Reading XML data from a database

In the fictitious healthcare company, we use DB2 for the database storage. The XML files are stored in a single DB2 pureXML column in the database. In a pureXML column, the XML files can be stored in their native hierarchical format without needing to shred the data into relational format. Applications can access the XML data in this column. In the DataStage Designer canvas, you can use the DB2 Connector stage to read the XML files from the database. The XML files are stored in a single column, *claim_XML,* in the database. The DB2 Connector can access the appropriate database table to read this particular column. The output of the DB2 Connector stage is given to the XML Stage, as shown in Figure 7-4.

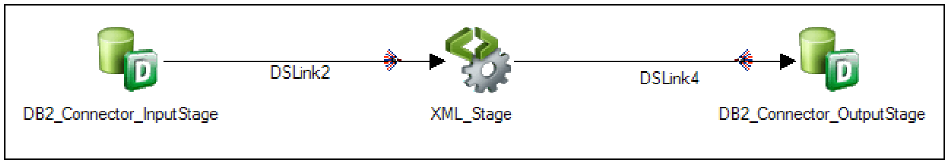
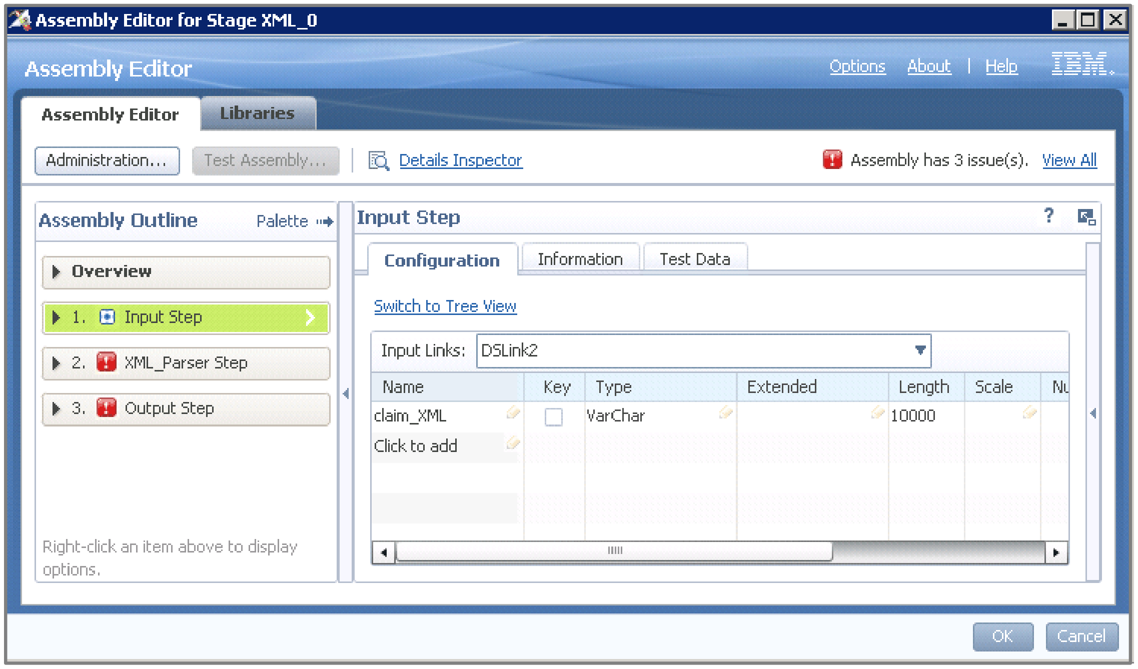


Figure 7-4 The DataStage job to read XML files from the database column

The Input step in the Assembly Editor describes the metadata of a relational structure. The step becomes active when an input link is provided to the XML Stage. You can view the columns that are added to the preceding stage of the XML Stage here. You can modify, remove, or add the columns. When the input is provided from the DB2 Connector to the XML Stage, the Input Step becomes active and the DB2 column, claim_XML, is displayed, as shown in Figure 7-5.



Name	Key	Type	Extended	Length	Scale	Nu
claim_XML	☐	VarChar		10000		
Click to add						

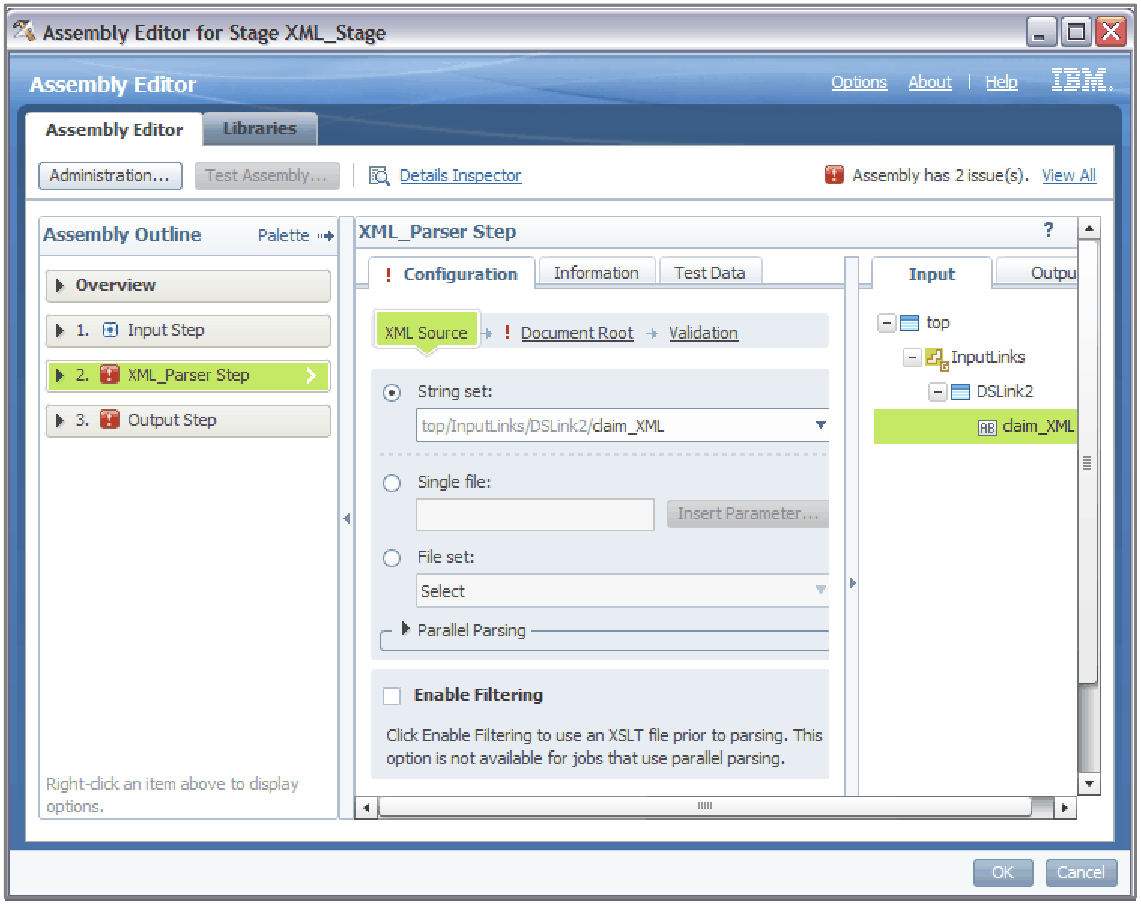
Figure 7-5 The Input Step in the Assembly Editor

To read the data from the database column, you must configure the XML Source tab of the Parser Step. You must use the *String set* option in the Parser Step to read the XML data from the database column claim_XML. Use the String set option when the entire XML file must be read from a previous stage in the designer canvas. Another stage reads the XML data and passes it on to the XML Stage.

Use the following steps to configure the String set option in the XML Parser Step:

1. Select the **String set** option in the XML Source tab. The drop-down list for the String set option becomes active.
2. Click the String set drop-down list. All input columns to the XML Stage show. First, `top/InputLinks`, followed by the name of the input link to the XML Stage, shows. If the input link is `DSLink2`, the column name `top/InputLinks/DSLink2/claim_XML` shows.
3. Select the **claim_XML** column from the drop-down list.

After following these steps, you can see that the XML Source tab is configured, as shown in Figure 7-6 on page 151.



String set: top/InputLinks/DSLink2/claim_XML

Single file:

File set: Select

Enable Filtering

Click Enable Filtering to use an XSLT file prior to parsing. This option is not available for jobs that use parallel parsing.

Figure 7-6 The XML Source tab in the XML Parser Step

7.1.2 Reading multiple XML files from a single directory or folder

The healthcare company also stores several of the XML files in a folder on one of its internal servers. The company does not store these files in the database. Therefore, these files must be read by the XML Stage directly from the folder.

You can use the *File set* option in the XML Source tab of the Parser Step to read multiple files that need to be parsed by the stage. You need to pass the absolute location of the input XML files to the File set option. The File set option reads the files from the folder based on the absolute paths of the files.

You can use one of the following approaches to feed the absolute path of the XML files to the XML Stage:

- A single flat file can be created to store the absolute path of each input XML file. This flat file can be read by using the Sequential File stage in the designer canvas. The input of the sequential file must be given to the XML Stage, as shown in Figure 7-7. With this approach, you need to make an entry in the flat file for every new XML file.

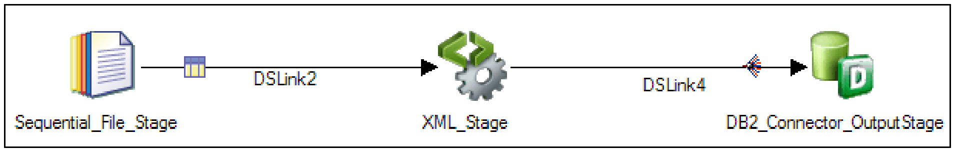


Figure 7-7 Sequential_File_Stage reads multiple XML documents job design

- The External Source stage in the parallel canvas can be used to find the absolute paths of the XML files. The output of the External Source stage must be given to the XML Stage, as shown in Figure 7-8. The advantage of using the External Source stage is that you do not need to enter the paths manually in a flat file. The stage can be configured to run a query and fetch all XML file locations at run time from the folder, for example, a query to read the XML files from the folder `C:\files` is **`ls C:\files\*.xml`** (**`ls`** is the command to list files).

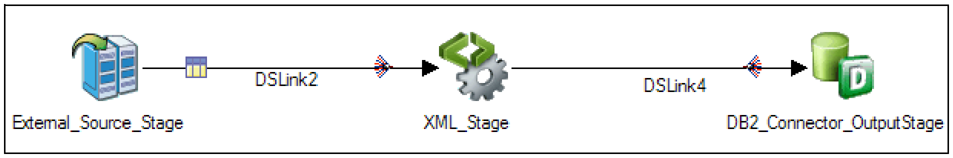
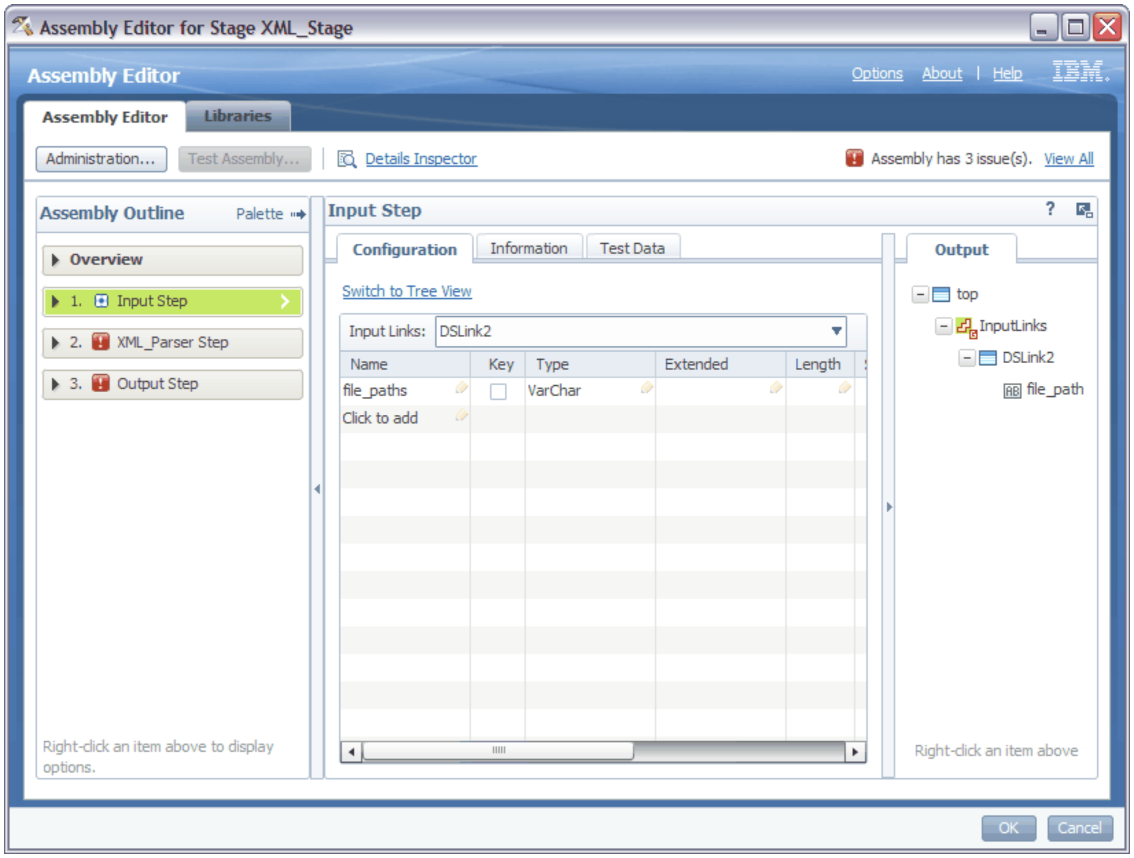


Figure 7-8 External Source stage reads multiple XML documents job design

In both cases, as input is provided to the XML Stage, the Input Step becomes active and the file_paths column is displayed, as shown in Figure 7-9 on page 153.



Input Links: DSLink2

Name	Key	Type	Extended	Length
file_paths	☐	VarChar		
Click to add				

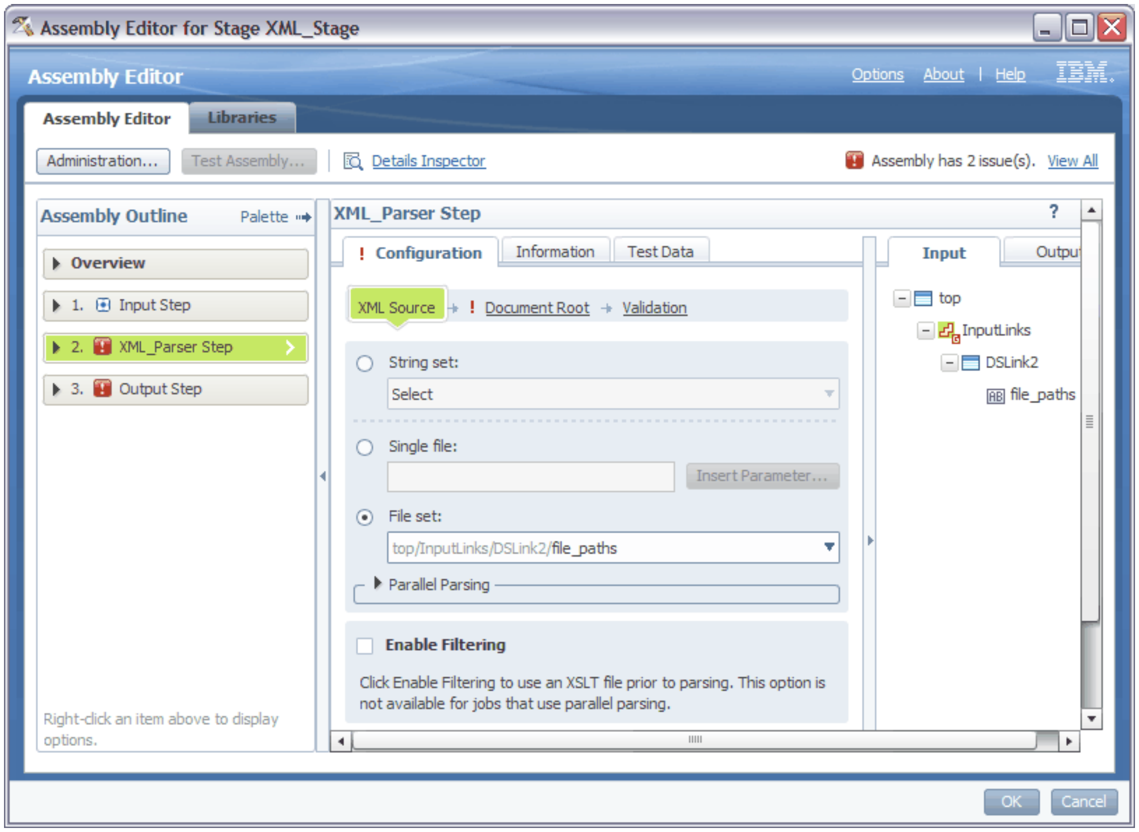
Figure 7-9 File_paths column in the Input Step in the Assembly Editor

Use the following steps to configure the File set option in the XML Parser Step:

1. Select **File set** option in the XML Source tab. The drop-down list under the File set option becomes active.
2. Click the drop-down list. The input columns to the XML Stage appear.

   All input columns appear beginning with `top/InputLinks` followed by the name of the input link to the XML Stage. Therefore, if the input link is `DSLink2` and the input column name is `file_paths`, the column name is `top/InputLinks/DSLink2/file_paths`.
3. Select the column **file_paths** from the drop-down list. The column file_paths holds the absolute location of the XML files.

After following these steps, you can see that the XML Source tab is configured, as shown in Figure 7-10 on page 154.



String set: Select

Single file: Insert Parameter...

File set: top/InputLinks/DSLink2/file_paths

Parallel Parsing

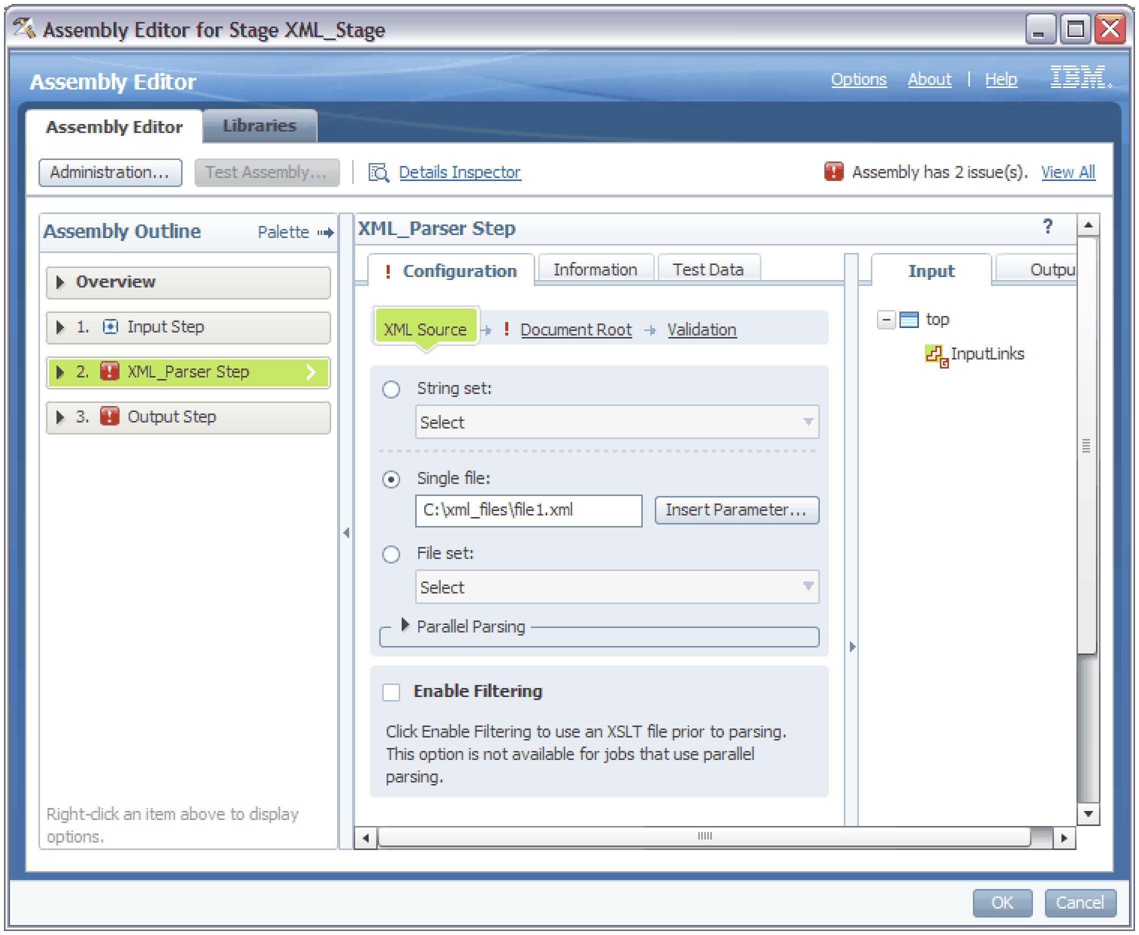
Enable Filtering

Click Enable Filtering to use an XSLT file prior to parsing. This option is not available for jobs that use parallel parsing.

Figure 7-10 File set option configuration in the XML Parser Step

7.1.3 Reading a single file directly from the XML Stage

The healthcare company can also configure the XML Stage to directly read the XML file. You can use the *Single file* option in the XML Source tab of the Parser step to read an XML file directly from the server. You need to specify the absolute location of the file and the file name in the Single file option, as shown in Figure 7-11 on page 155.


Assembly Editor for Stage XML_Stage

Assembly Editor

Options | About | Help

Assembly Editor | Libraries

Administration... | Test Assembly... | Details Inspector

Assembly has 2 issue(s). View All

Assembly Outline — Palette

- Overview
- 1. Input Step
- 2. XML_Parser Step
- 3. Output Step

Right-click an item above to display options.

XML_Parser Step

! Configuration | Information | Test Data

XML Source → ! Document Root → Validation

String set: Select

[✓ Single file:] C:\xml_files\file1.xml — Insert Parameter...

File set: Select

Parallel Parsing

Enable Filtering

Click Enable Filtering to use an XSLT file prior to parsing. This option is not available for jobs that use parallel parsing.

Input | Outpu

- top
  - InputLinks

OK | Cancel

Figure 7-11 Directly read configuration in the XML Parser Step

In this job design, the XML Stage is the source stage in the designer canvas and no input is provided to it, as shown in Figure 7-12.

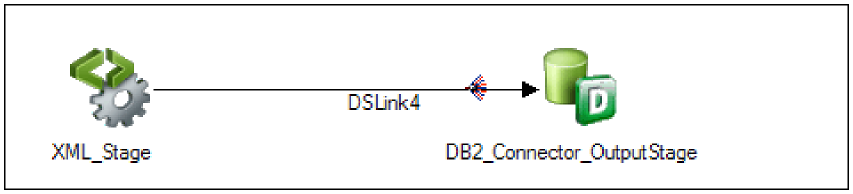
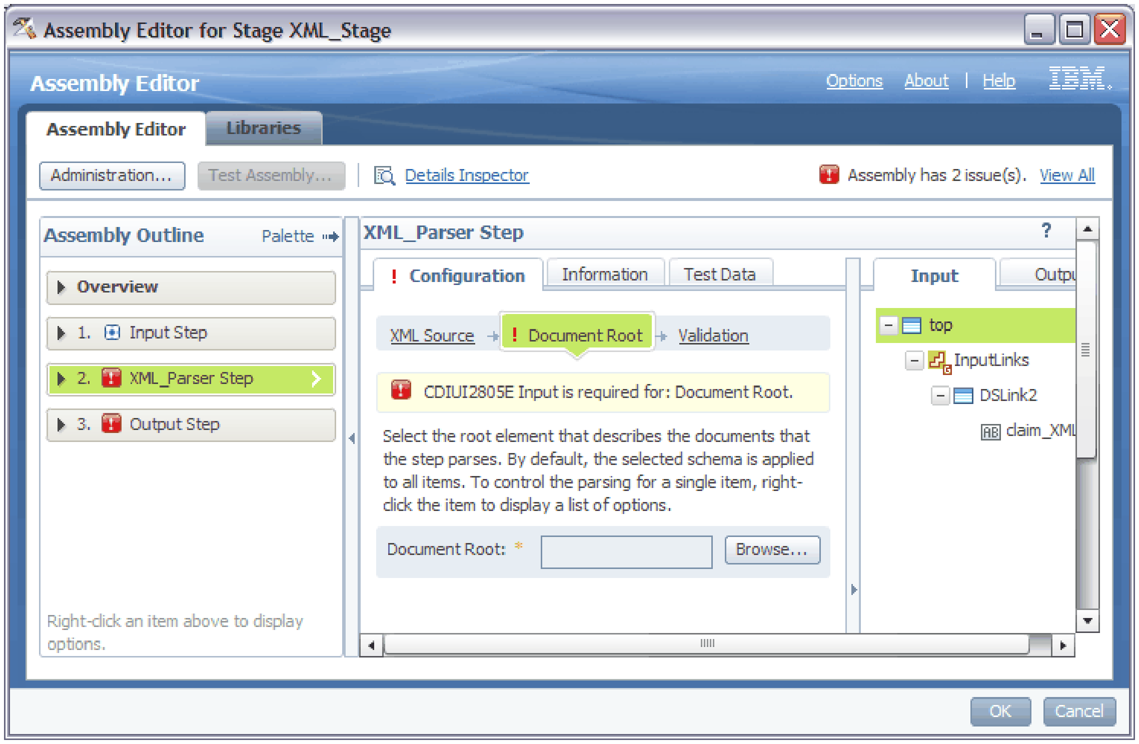


Figure 7-12 Job design when the XML file is read from the XML Stage

7.1.4 Specifying the XML schema in the Document Root tab

The XML Stage is a schema-driven stage. Without a valid schema, you cannot design the assembly. In order for the schema to be available within the assembly, the schema (`xsd` files) needs to be imported into the Schema Library Manager. When you import the schema into the Schema Library Manager, the schema is converted into a simplified model. This simplified model is then available to the steps of the assembly.

The fictitious healthcare company based its input files on an XML schema. This schema needs to be defined in the Document Root tab of the XML Parser Step, as shown in Figure 7-13. The schema defined in the Document Root tab describes the common structure of all input XML files to be parsed by the XML Parser step.



Assembly has 2 issue(s).

CDIUI2805E Input is required for: Document Root.

Select the root element that describes the documents that the step parses. By default, the selected schema is applied to all items. To control the parsing for a single item, right-click the item to display a list of options.

Document Root: *

Right-click an item above to display options.

Figure 7-13 Document Root tab in the XML Parser Step

Use the following steps to define the schema in the Document Root tab:

1. Click **Browse** in the Document Root tab of the Parser Step, and the Select Document Root window opens. This window shows all the available libraries

in the Schema Library Manager. Each library shows the elements of the schema that are imported into it.

The healthcare company created two libraries in the Schema Library Manager, Claim and Customer, and imported the `xsd` files `Claim.xsd` and `Customer.xsd` into the appropriate libraries. You can see both libraries in the Select Document Root window, as shown in Figure 7-14.

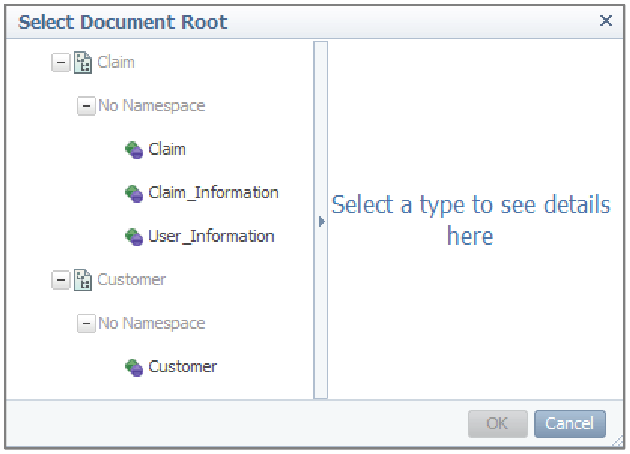


Figure 7-14 Select Document Root window

2. In the Select Document Root window, select the top-level element on which the XML file is based. The top-level element in the claim files of the company is Claim, as shown in Example 7-1. Therefore, you select the item **Claim** in the Select Document Root window.

Example 7-1 An example of the Claim XML file

```
<?xml version="1.0" encoding="UTF-8"?>
<Claim>
    <Claim_Information>
        <claim_line_number>1</claim_line_number>
        <claim_number>A100</claim_number>
        <member_id>12Z00</member_id>
        <provider_id>00X0</provider_id>
        <claim_code>I</claim_code>
    </Claim_Information>
</Claim>
```

3. After you select the element Claim in the Select Document Root window, you can see the structure of the element on the right side of the window, as shown in Figure 7-15 on page 158.

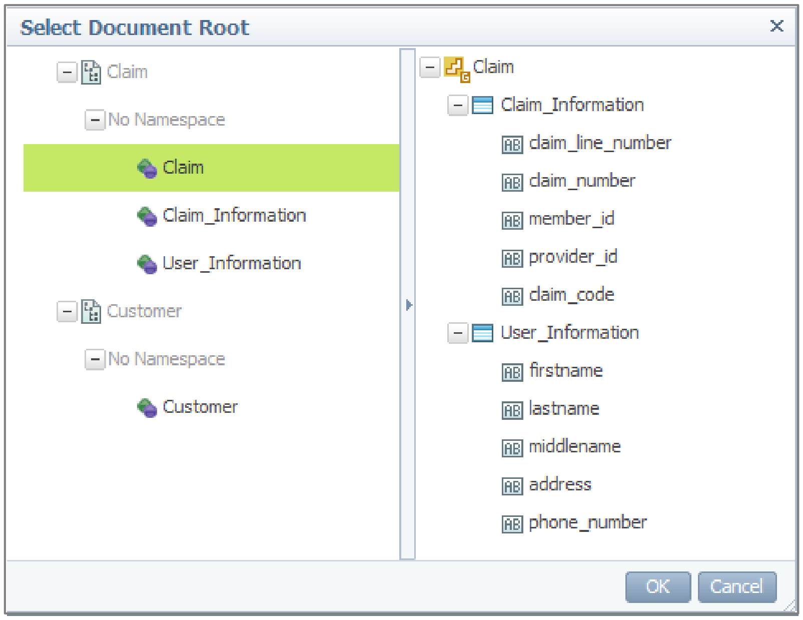
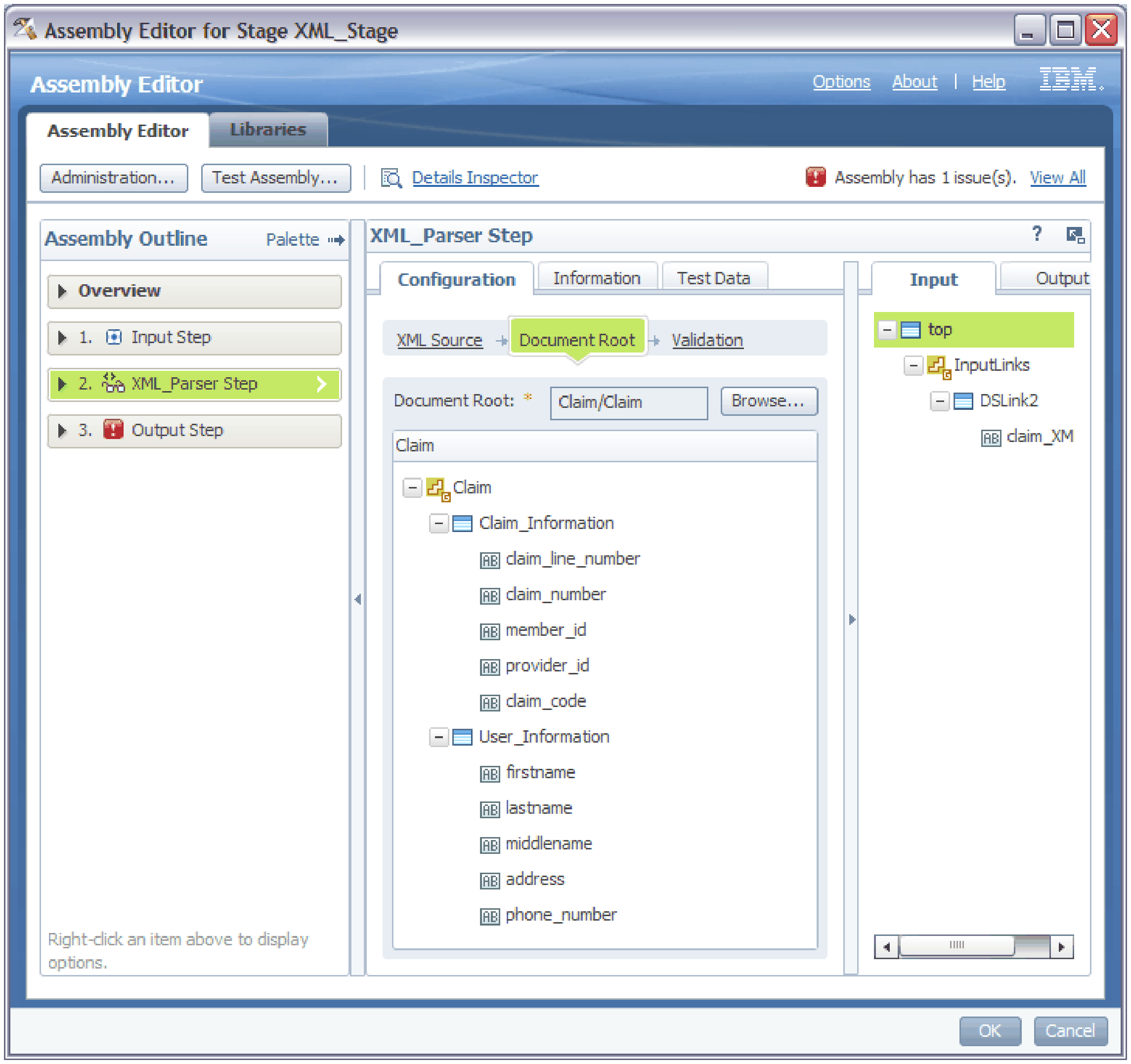


Figure 7-15 The Select Document Root window that shows the schema structure

4. Click **OK** to close the Select Document Root window. The schema structure is defined in the Document Root tab, as shown in Figure 7-16 on page 159.



Claim
- Claim
  - Claim_Information
    - claim_line_number
    - claim_number
    - member_id
    - provider_id
    - claim_code
  - User_Information
    - firstname
    - lastname
    - middlename
    - address
    - phone_number

- top
  - InputLinks
    - DSLink2
      - claim_XM

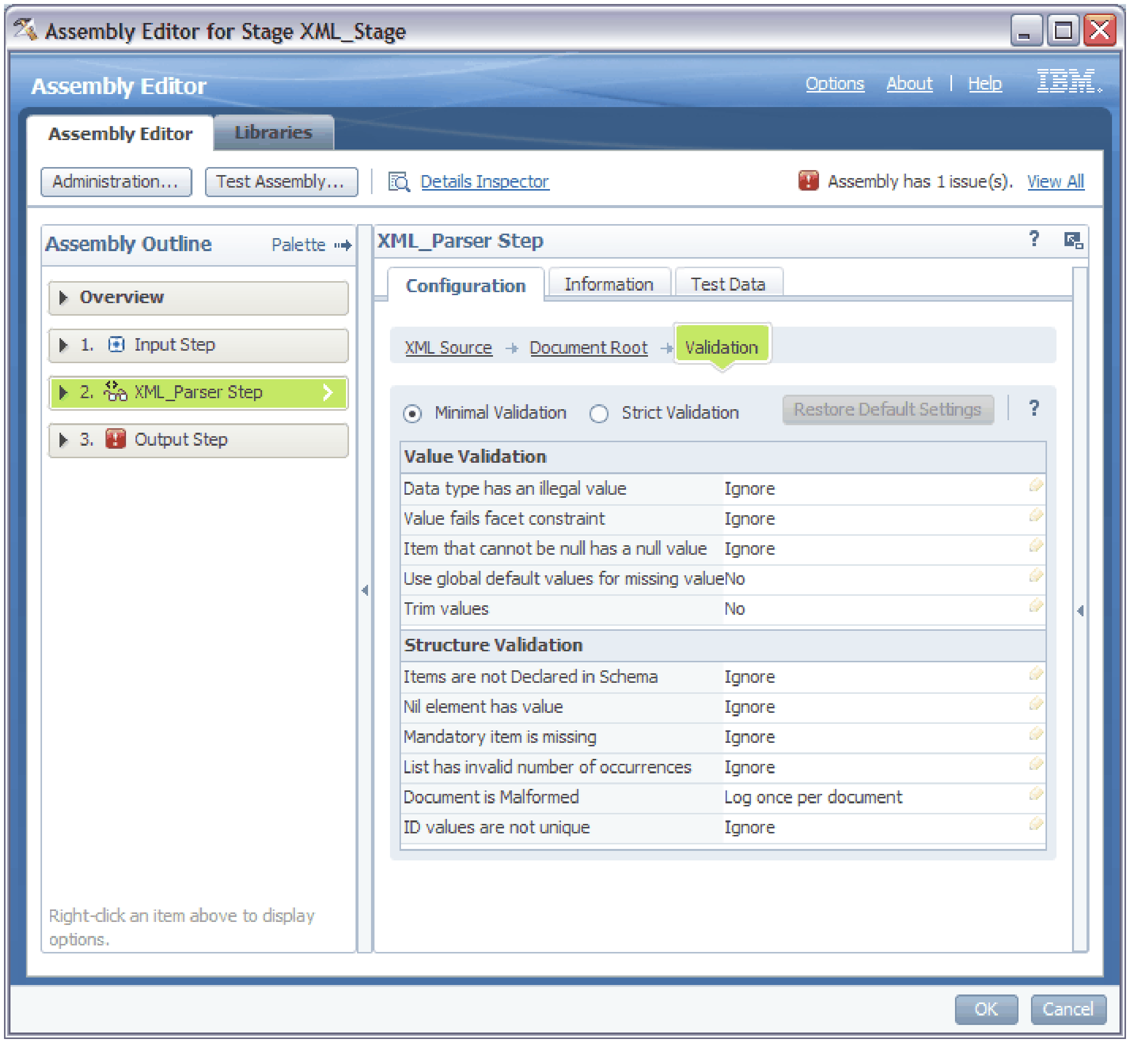
Figure 7-16 Schema defined in the Document Root tab of the Parser Step

7.1.5 Validating the XML file against the schema

To claim insurance from the healthcare company, the customers need to fill the claim form through a web application. These claim forms are converted into XML files. Human errors occur, because the forms are manually completed by the customers. Therefore, the company validates the claim XML files against the schema before the data is stored in the database.

The XML Stage has unique validation capabilities so that you can control the amount of schema validation when parsing XML documents. The spectrum of validation ranges from minimal validation to strict schema validation. You control the trade-off between performance and validation so that you can achieve the degree of complexity that you want.

The degree of validation can be selected through the Validation tab in the XML Parser Step, as shown in Figure 7-17. The Validation tab presents a set of validation rules. Each rule consists of a condition and an action. The condition is the validation check. If the check fails, the action is performed.



(●) Minimal Validation ( ) Strict Validation

Value Validation	
Data type has an illegal value	Ignore
Value fails facet constraint	Ignore
Item that cannot be null has a null value	Ignore
Use global default values for missing value	No
Trim values	No
Structure Validation	
Items are not Declared in Schema	Ignore
Nil element has value	Ignore
Mandatory item is missing	Ignore
List has invalid number of occurrences	Ignore
Document is Malformed	Log once per document
ID values are not unique	Ignore

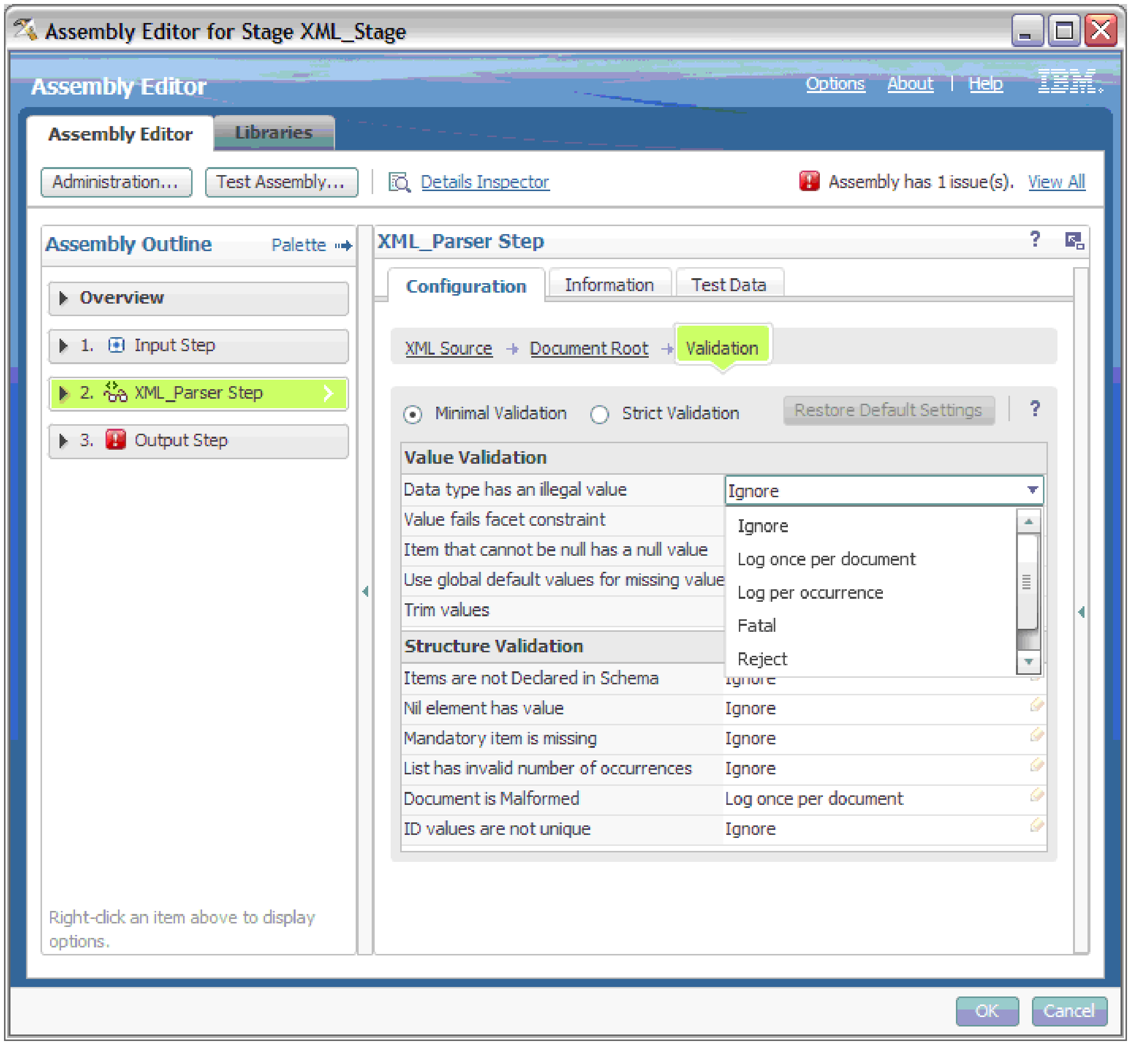
Figure 7-17 The Validation tab in the XML Parser Step

The Validation Rules are in two main categories, as shown in Figure 7-17 on page 160, which are Value Validation and Structure Validation. *Value validation rules* check the actual data values in the XML document against the defined schema types. *Structure Validation rules* check whether the XML document conforms to the schema, which is specified in the Document Root tab.

By default, Minimal Validation is selected as the validation mode in the XML Parser Step. Minimal Validation mode sets all the validation rule actions to `Ignore`. The only verifications on the XML document in this mode are that the top-level element of the XML file matches the top-level element defined in the Document Root and that the XML document is formed. No other validation is performed on the XML file.

To ensure that the XML file strictly conforms to the schema, Strict Validation mode can be selected. In Strict Validation mode, all the validation rule actions are set to `Fatal` to ensure that the job cancels on the first occurrence of invalid data.

Each of the validation rules can be configured to perform a specific action. When you click the action for a particular rule, the drop-down list shows the available actions that can be selected for that rule, as shown in Figure 7-18 on page 162.



(●) Minimal Validation ( ) Strict Validation

Value Validation	
Data type has an illegal value	Ignore
Value fails facet constraint	
Item that cannot be null has a null value	
Use global default values for missing value	
Trim values	
Structure Validation	
Items are not Declared in Schema	Ignore
Nil element has value	Ignore
Mandatory item is missing	Ignore
List has invalid number of occurrences	Ignore
Document is Malformed	Log once per document
ID values are not unique	Ignore

Figure 7-18 The actions available for the validation rules

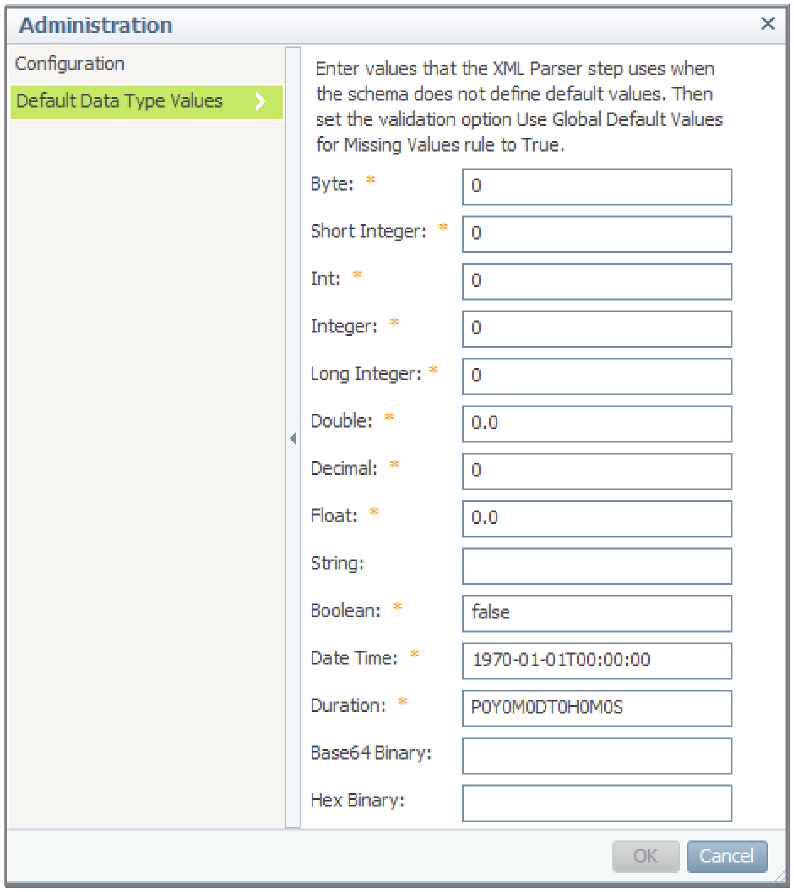
While entering the claim information in the web application, the customers of the healthcare company might make errors, such as the following errors:

- Not provide any data for the Phone Number field
- Add leading or trailing spaces in the name fields
- Provide a claim code that is invalid

The healthcare company takes the following actions for each of these errors:

- Provides a default Phone Number
- Trims all leading and trailing spaces
- Logs the information when the claim codes are invalid

The default Phone Number can be specified in the XML schema or in the Administration tab in the Assembly Editor, as shown in Figure 7-19. The Phone Number field is a string field, and therefore, the default value must be entered in the text box next to String in the Default Data Type Values tab. To add this default value to the element during parsing, you must set the action for the validation rule *Use global default values for missing values* to **Yes**.



Enter values that the XML Parser step uses when the schema does not define default values. Then set the validation option Use Global Default Values for Missing Values rule to True.

Byte: *	0
Short Integer: *	0
Int: *	0
Integer: *	0
Long Integer: *	0
Double: *	0.0
Decimal: *	0
Float: *	0.0
String:	
Boolean: *	false
Date Time: *	1970-01-01T00:00:00
Duration: *	P0Y0M0DT0H0M0S
Base64 Binary:	
Hex Binary:	

Figure 7-19 The Default Data Type Values window in the Administration tab

To trim the leading and trailing spaces in each of the fields, you must set the action for the validation rule Trim Values to **Yes**.

To log error messages, the actions *Log per Occurrence* and *Log once per Document* are available. These messages are logged in the DataStage Director Log. Log per Occurrence logs the message for each data that fails the validation rule, and Log once per Document logs only the first error message when the validation rule fails. For example, in the XML document that is shown in Example 7-2, the claim code fields have invalid data. Log per Occurrence logs messages for each of the claim codes. Therefore, two error messages are present in the Director log. Log once per Document logs the message only for the first invalid claim code, that is `AZXBIUTRE`.

Example 7-2 An example of the Invalid XML file

```
<?xml version="1.0" encoding="UTF-8"?>
<Claim>
    <Claim_Information>
        <claim_line_number>1</claim_line_number>
        <claim_number>A100</claim_number>
        <member_id>12Z00</member_id>
        <provider_id>00X0</provider_id>
        <claim_code>AZXBIUTRE</claim_code>
        <phone_number></phone_number>
        <firstname>  James  </firstname>
    </Claim_Information>

    <Claim_Information>
        <claim_line_number>2</claim_line_number>
        <claim_number>A100</claim_number>
        <member_id>12Z00</member_id>
        <provider_id>00X0</provider_id>
        <claim_code>ZZCVTBJU</claim_code>
        <phone_number></phone_number>
        <firstname>  James  </firstname>
    </Claim_Information>
</Claim>
```

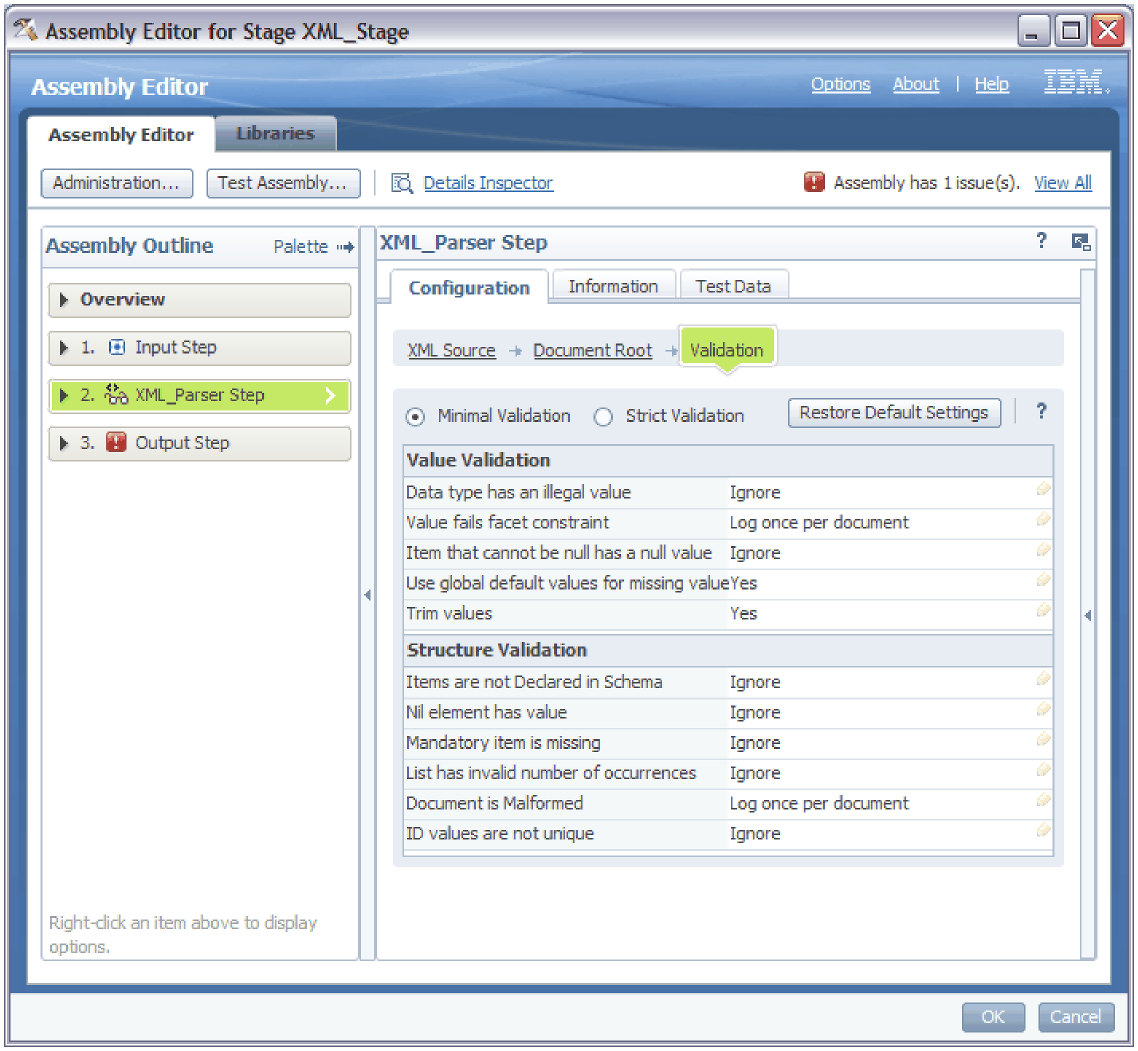
The valid claim_code values are specified in the XML schema. To ensure that the value in the XML file matches the value in the XML schema, the validation rule *Value fails facet constraint* is set to *Log once per document*. The validation rule Value fails facet constraint ensures that the incoming data value is in the range specified in the schema.

For the XML document shown in the Example 7-2, if these validation rules are applied, the following actions occur while parsing:

- For the Phone Number fields, the default value "`1-333-333-0000`" is added. This value needs to be defined in the Administration tab.

- The firstname field is trimmed to output `James`.
- An error message is logged in the DataStage Director for the claim_code field.

The final configuration of the Validation tab can be seen in Figure 7-20.



Minimal Validation [✓] Strict Validation

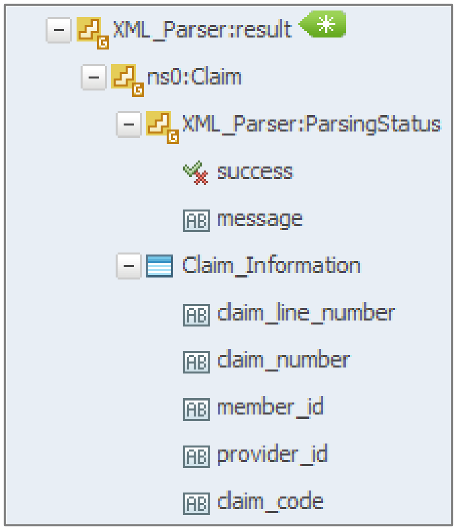
Value Validation	
Data type has an illegal value	Ignore
Value fails facet constraint	Log once per document
Item that cannot be null has a null value	Ignore
Use global default values for missing value	Yes
Trim values	Yes
Structure Validation	
Items are not Declared in Schema	Ignore
Nil element has value	Ignore
Mandatory item is missing	Ignore
List has invalid number of occurrences	Ignore
Document is Malformed	Log once per document
ID values are not unique	Ignore

Figure 7-20 Validation in the Configuration tab in the XML Parser Step

7.1.6 The Reject action

The XML Stage provides an option to reject the invalid claim XML files. To reject the XML files, the *Reject* action can be applied to any of the validation rules. The Reject action turns the status of the validation into regular data on which the downstream steps or DataStage stages can act.

When the Reject action is selected for any of the validation rules, two new items, named `success` and `message`, are added to the output schema of the XML Parser step, as shown in Figure 7-21. Success holds the value `true` if the validation passes and `false` if the validation fails. Message holds the error message if the validation fails. Based on the value of the success item, the XML documents can be divided into valid and invalid files.



```
XML_Parser:result
    ns0:Claim
        XML_Parser:ParsingStatus
            success
            message
        Claim_Information
            claim_line_number
            claim_number
            member_id
            provider_id
            claim_code
```

Figure 7-21 The output of the XML Parser step after using the Reject action

For example, assume that the rule *Data type contains an illegal value*. This rule checks whether the value specified for each of the elements/attributes in the XML file is valid and within the range for the specified data type. When the reject action is selected for this rule and if any of the elements or attributes have invalid data, the item success holds the value `false` and message holds the reason for the failure. Based on the value of success, the files can be filtered into valid and invalid files. For more detail, see Chapter 9, “Transforming XML documents” on page 265.

Even if one of the data values in the XML file is invalid, the value of the successful item is false. Therefore, the entire XML file becomes invalid.

Unlike the other DataStage stages, there might be no reject link from the XML Stage. All output links from the XML Stage are stream links.

7.1.7 Treating all XML data types as string

The validation rule *Data type has an illegal value* checks to see whether the value specified for each of the elements/attributes in the XML file is valid and within the range for the data type specified. When the action for this rule is set to *Ignore*, *Log per Occurrence*, or *Log once per Document*, all the elements are converted to string type. Therefore, the actual XML schema data types are not validated in the XML Parser step and all elements are treated as string types.

In Figure 7-22 on page 168, we show the actual schema representation in the Document Root tab of the XML Parser Step. The output of the XML Parser Step is seen in the Output tab on the rightmost section of the Assembly Editor. The output of the XML Parser Step shows all elements converted to string elements, because the action for the validation rule *Data type has an illegal value* is set to ignore.

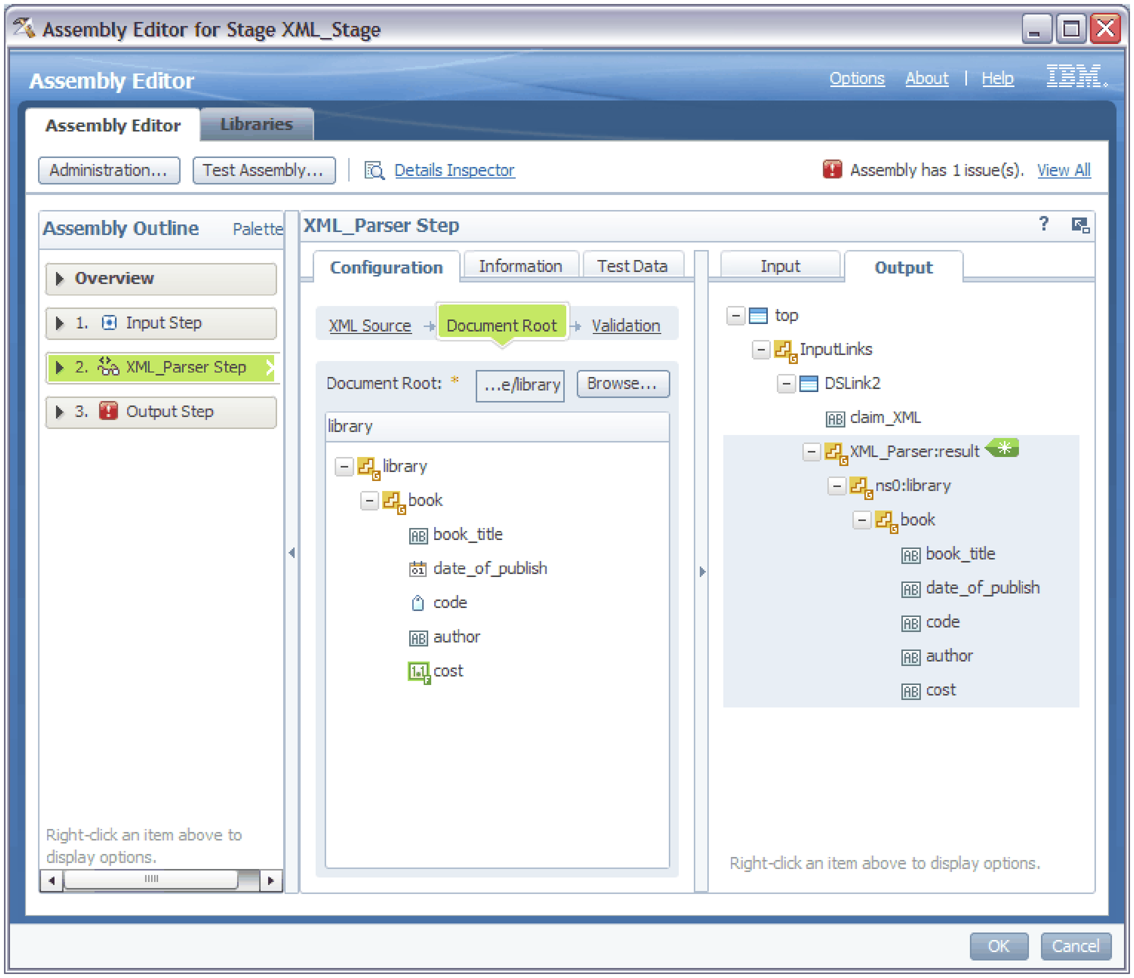


Figure 7-22 Schema representation between Document Root and Output tab

7.1.8 The output of the XML Parser step

To completely configure the XML Parser step, the healthcare company needs to configure the XML Source, Document Root, and Validation tabs. The output of the Parser step is seen in the Output tab. The output of the Parser step is contained within the group XML_Parser:result. You can see the entire XML schema beneath that group, as shown in Figure 7-23 on page 169.

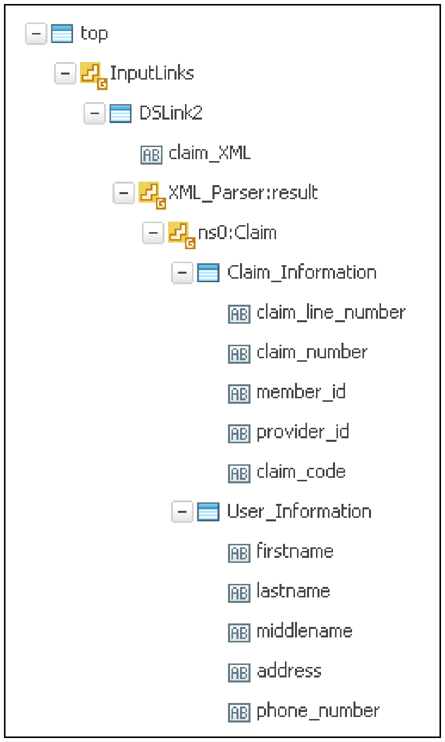


Figure 7-23 The output of the XML Parser step

When the String set or File set option is selected in the XML Source tab of the XML Parser Step, the XML_Parser:result is under the input list. As shown in Figure 7-23, the XML_Parser:result is under the list DSLink2. If the Single file option is selected, the XML_Parser:result will under the list `top`.

7.2 The Output step

With the Output step, you can map a hierarchical data structure to a relational data structure. This step becomes active when output links are present from the XML Stage. The Output step is used when the XML Stage output needs to be provided to another stage within DataStage.

The healthcare company wants to parse the XML claim files and store the data in a database. The Parser step, which we described in 6.1, "The XML Stage" on page 132, parses the claim files. To write the parsed data into the different columns of the database, the Output step is required.

The healthcare company uses a single table into which the data needs to be written. Because the company uses DB2 as its database, the DB2 Connector can be used to write the data to the database table. Figure 7-24 shows the DataStage job design when the XML files are read from the database, parsed, and written to different tables of the same database.

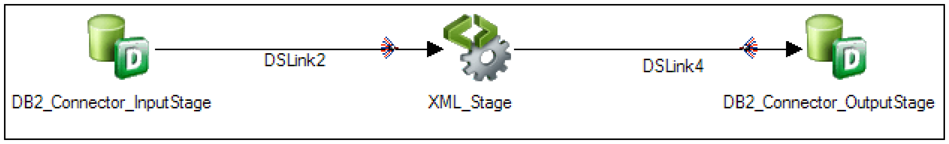
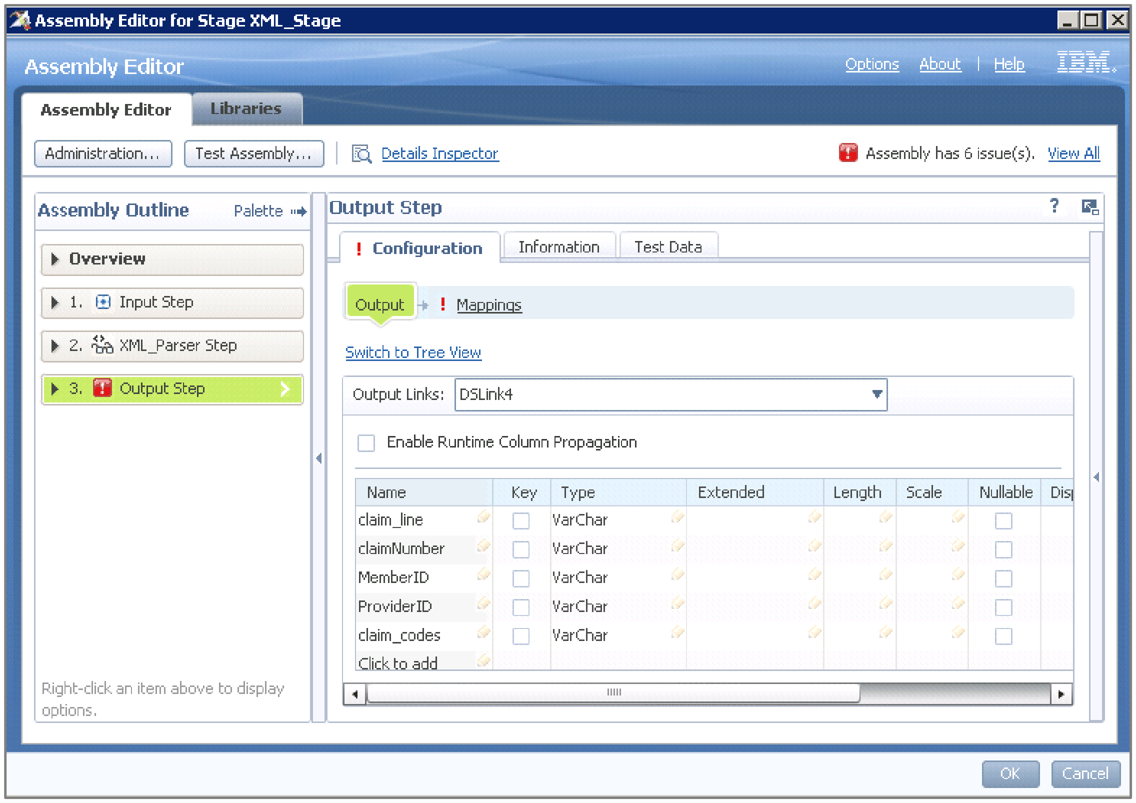


Figure 7-24 The DataStage job design

When an output link is provided from the XML Stage, the link is represented as a list within the assembly. The output columns are represented as items in the assembly. The columns for each of the output links are visible in the Output tab of the Output Step. As shown in Figure 7-25 on page 171, the output link DSLink4 has five columns in it. These columns can be modified or removed, or new columns can be added in the Output tab.



Output Links: DSLink4

☐ Enable Runtime Column Propagation

Name	Key	Type	Extended	Length	Scale	Nullable	Dis
claim_line	☐	VarChar				☐	
claimNumber	☐	VarChar				☐	
MemberID	☐	VarChar				☐	
ProviderID	☐	VarChar				☐	
claim_codes	☐	VarChar				☐	
Click to add							

Figure 7-25 The Output tab of the Output Step

To map the output of the Parser step to the columns of the DB2 Connector, the *Mappings* tab is used. The Mappings table is a common construct to map one hierarchical structure to another. The Mappings table is used as the last step in adapting the input to the output structure. You cannot perform a join, union, or any other set-operation and map to a target list. If a set-operation is needed, it needs to be performed in a Transformation step before the Output step. As shown in Figure 7-26 on page 172, the Mappings table contains three columns: Source, Result, and Target. The Target column displays the output link and the columns under it. The Source column displays the columns that need to be mapped to the target items. The Result column displays the status of the current mapping.

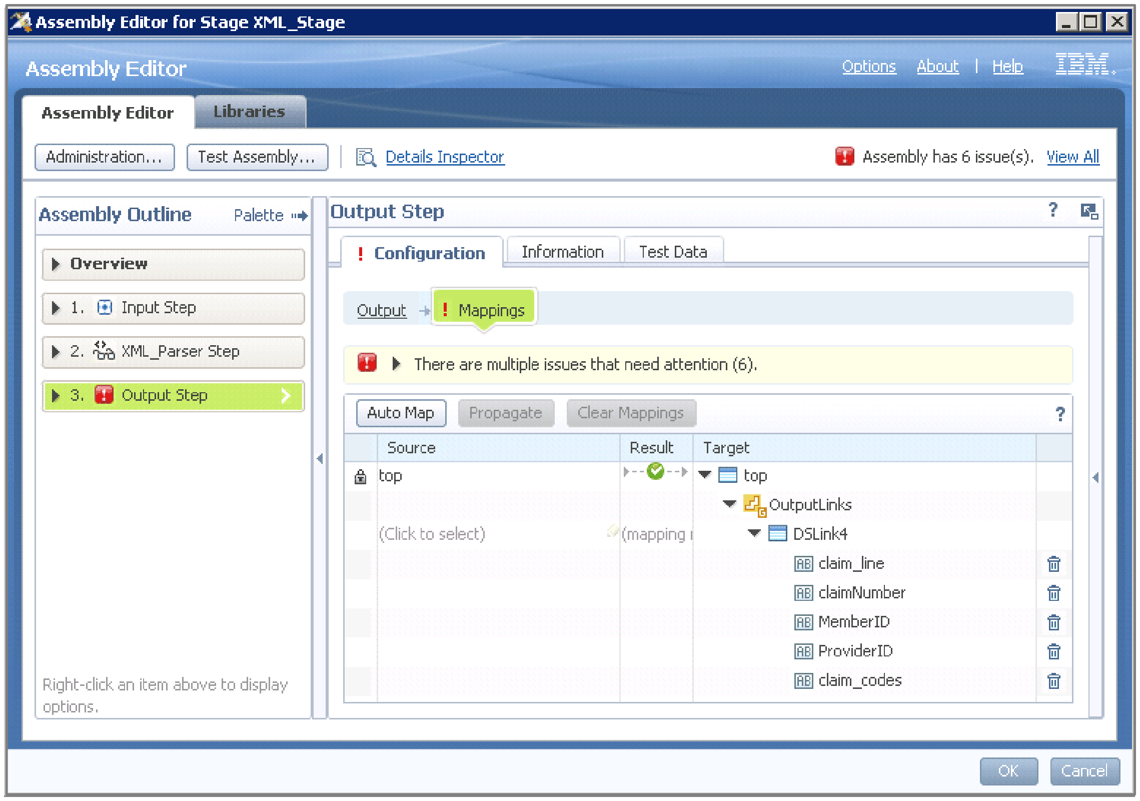


Figure 7-26 The Mappings tab of the Output step

The `top` list is always the first list in the Mappings table and is always mapped to the source item `top`. This mapping cannot be configured. `Top` contains the group OutputLinks. The entire output structure is contained in this group.

To map a hierarchical structure to a relational structure, use the following mapping rules:

- *You can only map a list in the Target column to a list in the Source column*. For example, in Figure 7-26, only a source list can be mapped to DSLink4. Because all lists are represented with a blue icon, you must map an item with a blue icon to another item with a blue icon.
- *A group is never mapped*. Groups are only structures used to create, and they do not need to be mapped. In the Output step, only a single group exists in the Target column, for example, OutputLinks. No mapping occurs in the Source column for this group.

- *To map a source item to a target column, the parent list of the item needs to be mapped first*. Therefore, in Figure 7-26 on page 172, to map a source item to the Column claim_line, the list DSLink4 needs to be mapped first.

Following these rules, the output of the Parser step must be mapped to the target columns in the Output step, as shown in Figure 7-27.

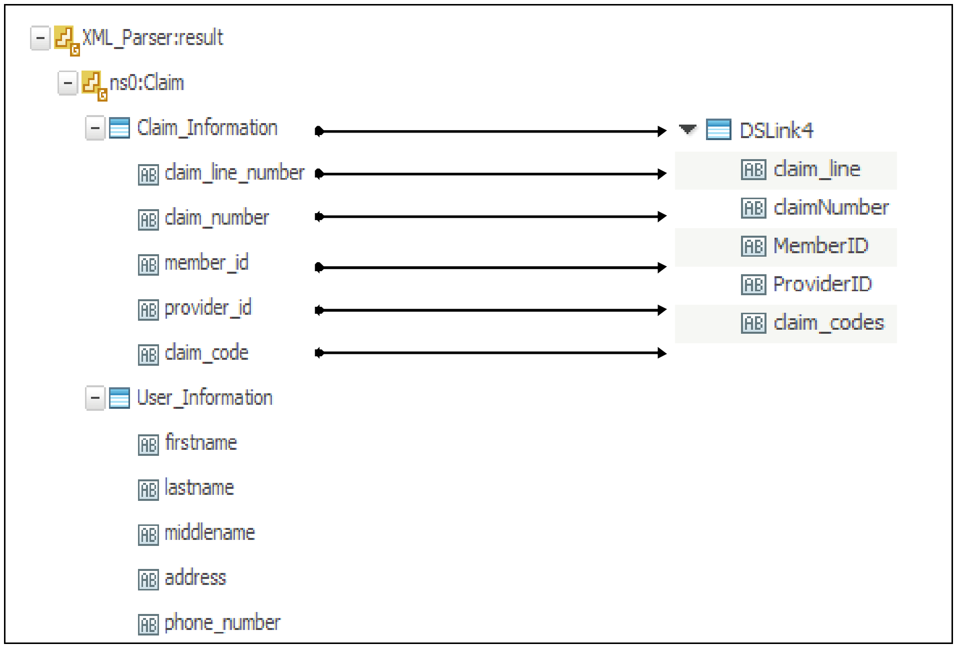


Figure 7-27 Mapping Parser step output and the target columns in the Output step

Use the following steps to complete the Mappings table:

1. Double-click the **Source** column for the target list DSLink4 to see a drop-down list. This drop-down list is called the suggestion list. The *suggestion list* shows all the available source lists that can be mapped to the target list. Both source lists, User_information and Claim_Information, are available in the suggestion list, as shown in Figure 7-28 on page 174. The absolute paths can be seen in the suggestion list, for example, `top/InputLinks/DSLink2/XML_Parser:result/Claim/User_Information`.

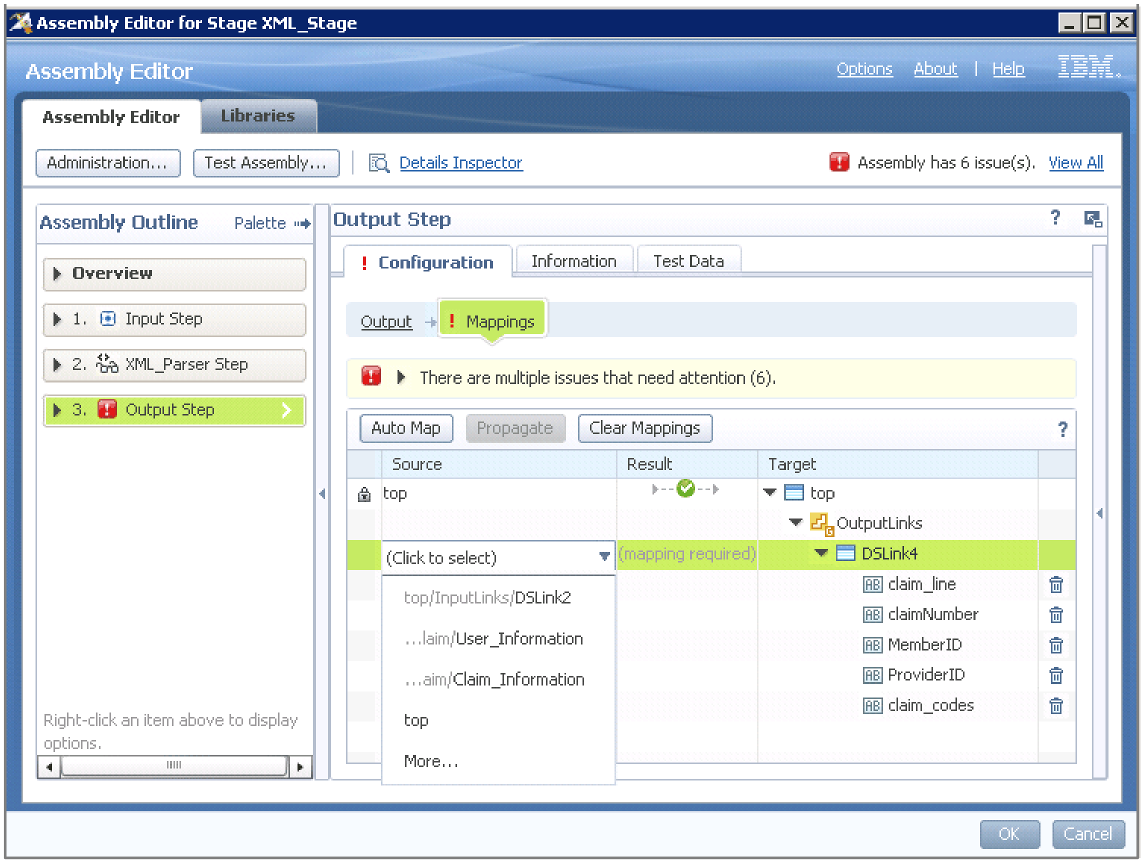
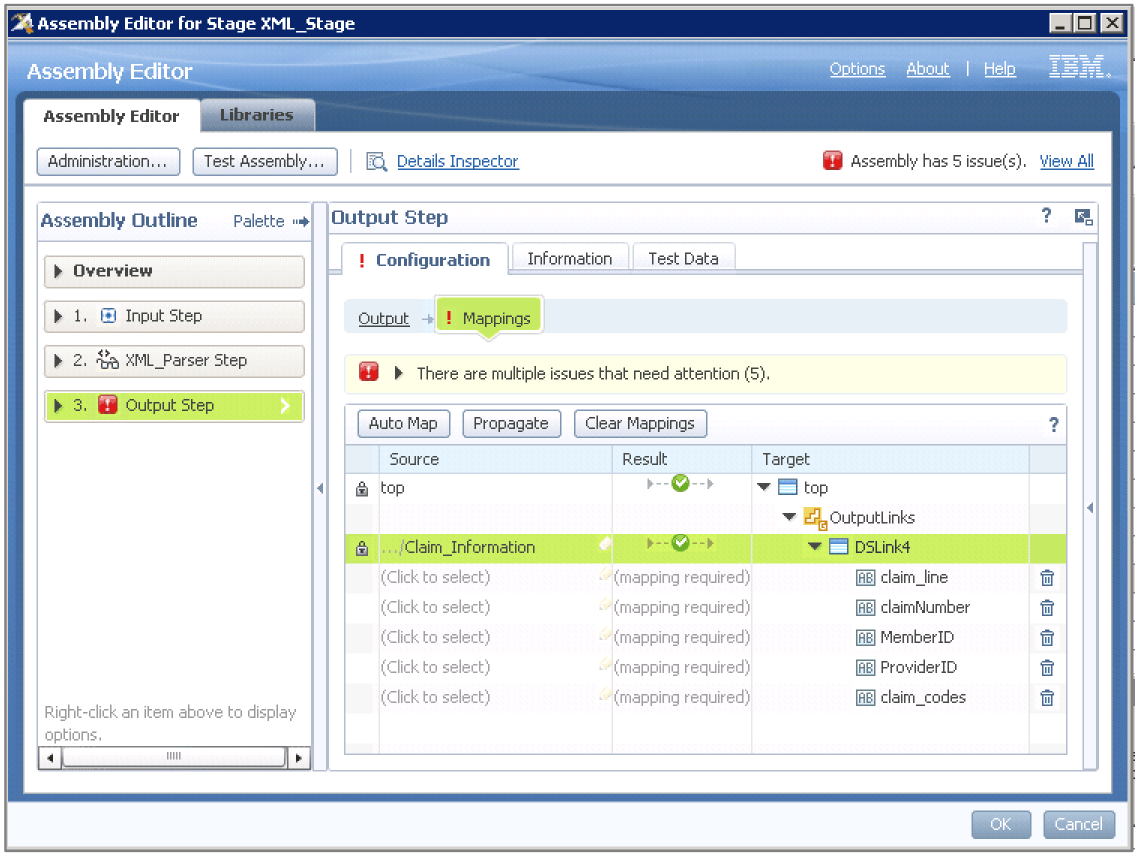

Assembly Editor for Stage XML_Stage
Assembly Editor
Options About | Help
Assembly Editor Libraries
Administration... Test Assembly... Details Inspector
Assembly has 6 issue(s). View All
Assembly Outline Palette
Overview
1. Input Step
2. XML_Parser Step
3. Output Step
Right-click an item above to display options.
Output Step
Configuration Information Test Data
Output Mappings
There are multiple issues that need attention (6).
Auto Map Propagate Clear Mappings
Source Result Target
top top
OutputLinks
(Click to select) (mapping required) DSLink4
top/InputLinks/DSLink2
...laim/User_Information
...aim/Claim_Information
top
More...
claim_line
claimNumber
MemberID
ProviderID
claim_codes
OK Cancel

Figure 7-28 The suggestion list for the output link DSLink4

2. Because the source columns that need to be mapped to the target columns are under the source list Claim_Information, this column is selected from the suggestion list. When you select Claim_Information, the mapping occurs for this row and a white check mark in a green circle shows between the source and target lists, as shown in Figure 7-29 on page 175.



Assembly has 5 issue(s). View All

There are multiple issues that need attention (5).

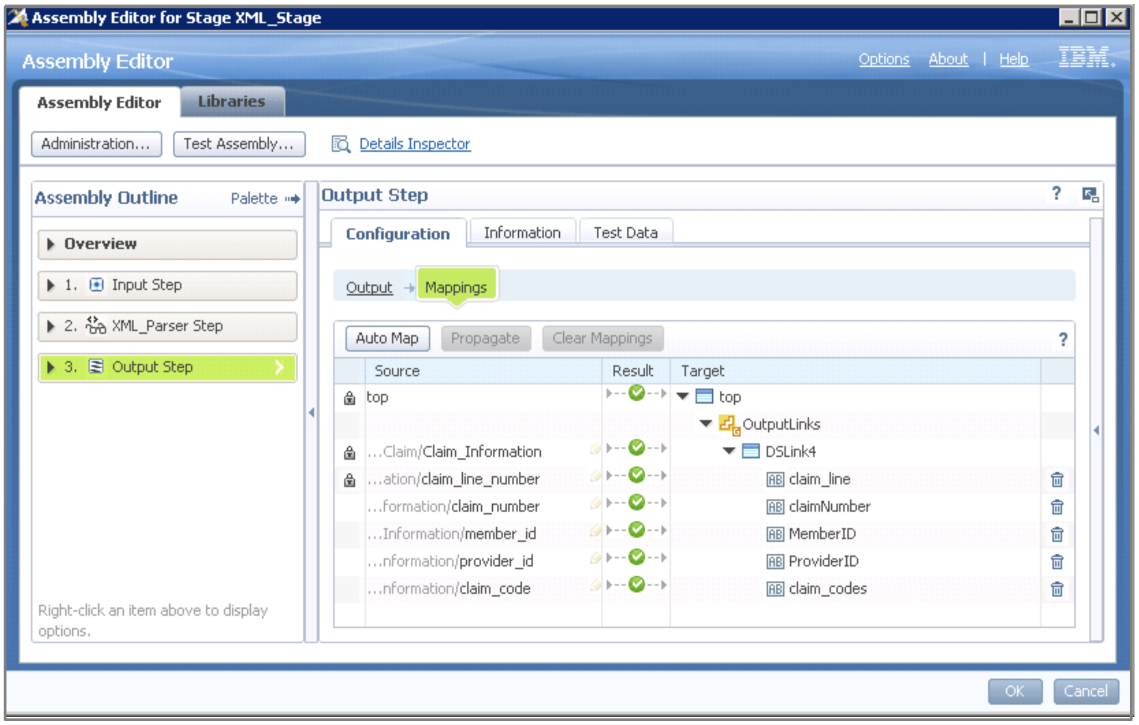
Right-click an item above to display options.

Source	Result	Target
top		top
		OutputLinks
.../Claim_Information		DSLink4
(Click to select)	(mapping required)	claim_line
(Click to select)	(mapping required)	claimNumber
(Click to select)	(mapping required)	MemberID
(Click to select)	(mapping required)	ProviderID
(Click to select)	(mapping required)	claim_codes

Figure 7-29 The Mappings table for the Source and Target lists

3. After mapping the target list DSLink4, the columns under it can be mapped. To map each of the target columns, select the corresponding row in the Source column. From the suggestion list, select the appropriate item to be mapped. The item recommendations in the suggestion list are based on the name and data type of the Target column.

After mapping all the columns, the Mappings table is complete. No errors are seen in the Output Step, as shown in Figure 7-30 on page 176.



Source	Result	Target
top	✓	top
		OutputLinks
...Claim/Claim_Information	✓	DSLink4
...ation/claim_line_number	✓	claim_line
...formation/claim_number	✓	claimNumber
...Information/member_id	✓	MemberID
...nformation/provider_id	✓	ProviderID
...nformation/claim_code	✓	claim_codes

Figure 7-30 The completed Mappings table

7.2.1 Automatically mapping the columns

One of the company output tables has over a hundred columns. It can be tedious to map each column in the target to the corresponding column in the source. The Output step provides an easy mechanism to map multiple columns simultaneously, and that is the *Auto Map* feature. The following steps describe how to use the Auto map feature:

1. Map the source list Claim_Information to the target list DSLink4.
2. Select the row corresponding to the target list DSLink4.
3. Click **Auto Map**. All the columns under the list Claim_Information are automatically mapped.

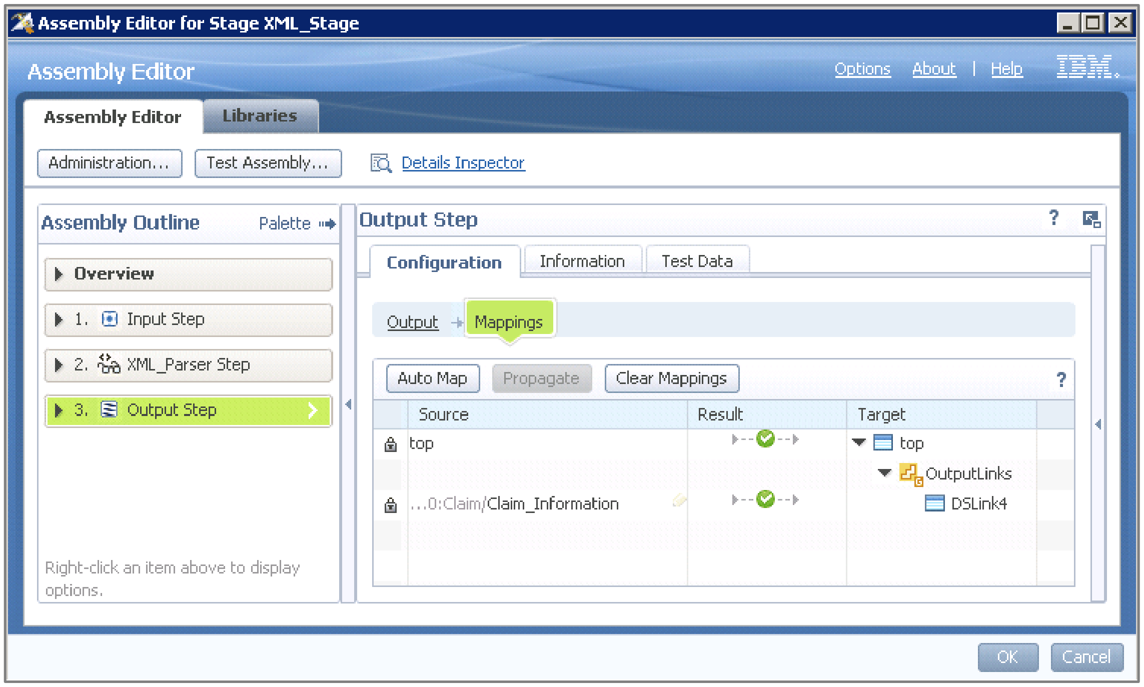
The mapping occurs based on the similarity between the name and data type of the source and target columns. The automatic mapping suggestion might not always be correct. The Auto Map feature also cannot map columns of different data types. Therefore, if the Target column is an integer and all source columns are of type string, by using the Auto Map feature, no mapping occurs for the integer column.

7.2.2 Propagating columns from the source to the target

Sometimes the output columns are not defined in the Output stage as with the database. The developers at the company need to create and map the columns. The Output step provides the *Propagate* feature to facilitate this process. The Propagate option creates the target columns and also automatically maps these columns to the appropriate columns in the source structure.

Follow these steps to use the Propagate feature:

1. Map the source list to the target list. For example, DSLink4 is mapped to Claim_Information in Figure 7-31. No target columns show under the target list.

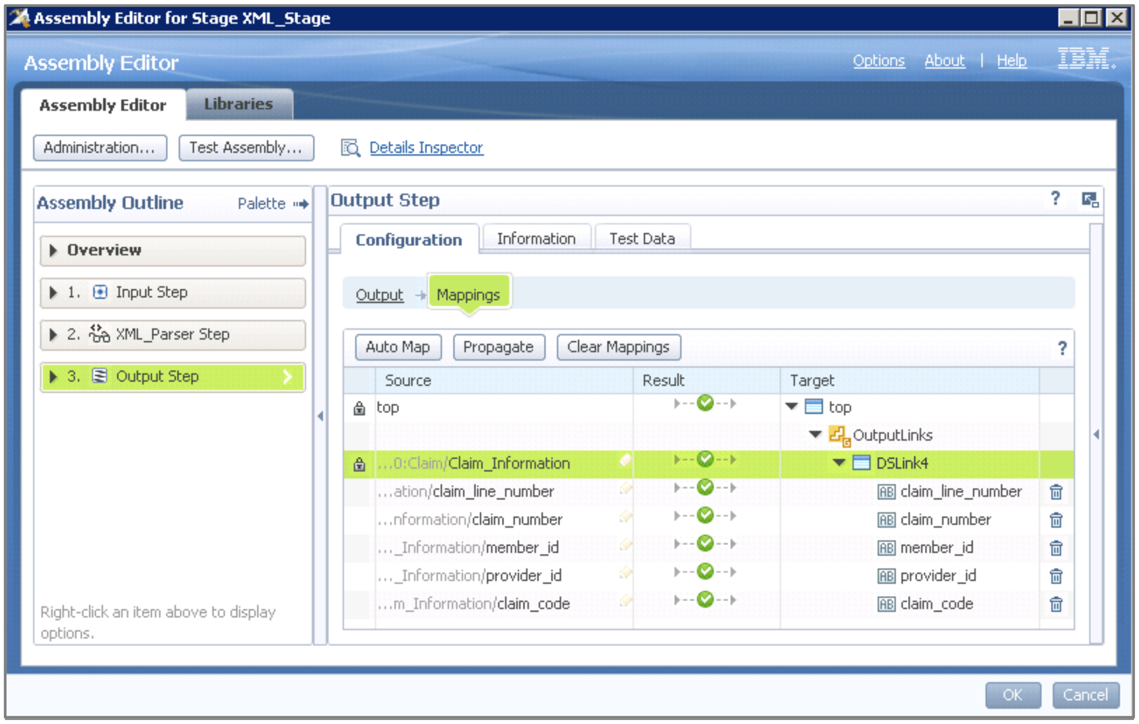


Source	Result	Target
top	✓	top
		OutputLinks
...0:Claim/Claim_Information	✓	DSLink4

Figure 7-31 The Mappings tab

2. Select the row corresponding to the target list DSLink4, and click **Propagate**.

3. All the columns under the list Claim_Information are propagated to the target, as shown in Figure 7-32. The names of the columns created in the target list are the same names of the columns in the source list.


Assembly Editor for Stage XML_Stage
Assembly Editor
Options About | Help
Assembly Editor Libraries
Administration... Test Assembly... Details Inspector
Assembly Outline Palette
Overview
1. Input Step
2. XML_Parser Step
3. Output Step
Right-click an item above to display options.
Output Step
Configuration Information Test Data
Output Mappings
Auto Map Propagate Clear Mappings
OK Cancel

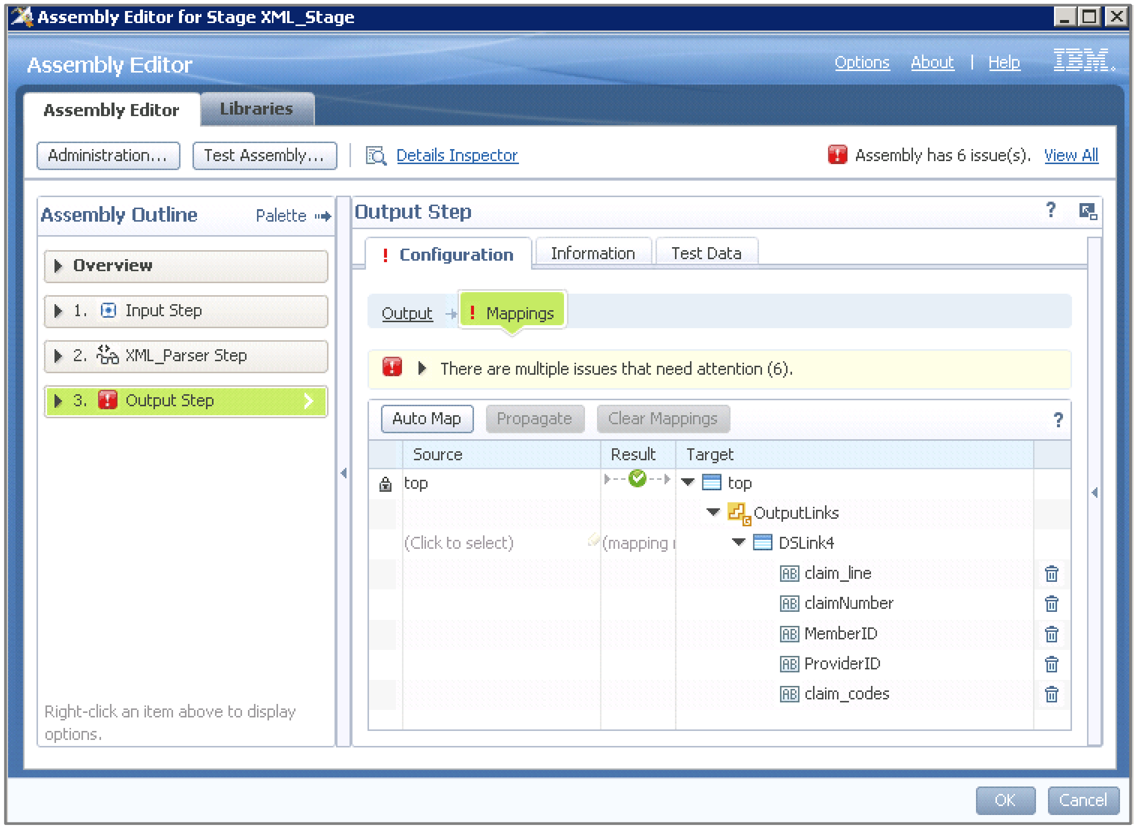
Source	Result	Target
top	✓	top
		OutputLinks
...0:Claim/Claim_Information	✓	DSLink4
...ation/claim_line_number	✓	claim_line_number
...nformation/claim_number	✓	claim_number
..._Information/member_id	✓	member_id
..._Information/provider_id	✓	provider_id
...m_Information/claim_code	✓	claim_code

Figure 7-32 Columns created with the Propagate option

7.2.3 Mapping constants to a target column

The company uses a table in which the ProviderID must always be set to '00X0' irrespective of the data that comes from the XML files. Therefore, a constant value needs to be assigned to the output column ProviderID. Use the Output Step Constant option. To map the target item ProviderID to the constant value '00X0', use the following steps:

1. Ensure that the parent list of the target item is mapped. Therefore, in Figure 7-33 on page 179, to assign a constant value to the item ProviderID, DSLink4 needs to be mapped. So, you can map Claim_Information to the list DSLink4.



Assembly has 6 issue(s).

Right-click an item above to display options.

There are multiple issues that need attention (6).

Source	Result	Target
top		top
		OutputLinks
(Click to select)	(mapping	DSLink4
		claim_line
		claimNumber
		MemberID
		ProviderID
		claim_codes

Figure 7-33 The Mappings table in the Output Step

2. Select the row in the source corresponding to the target item ProviderID. The suggestion list is shown in Figure 7-34 on page 180.

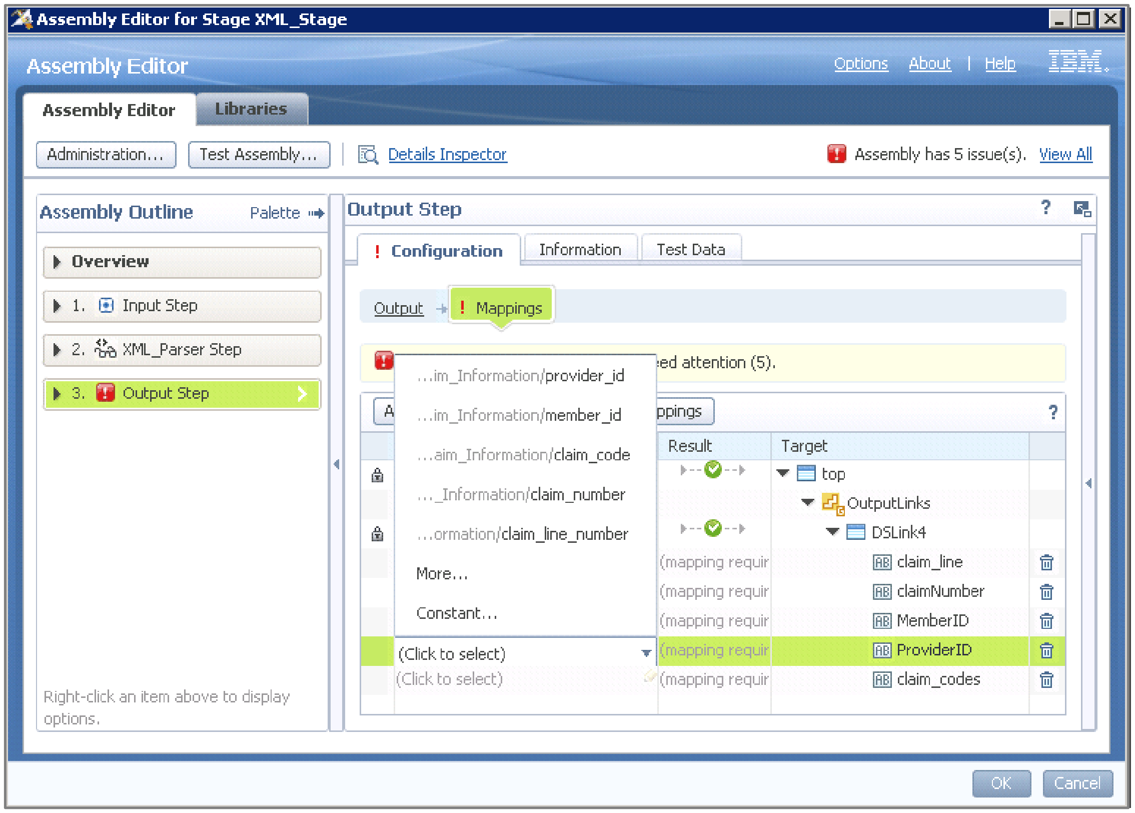


Figure 7-34 The suggestion list

3. The last option in the suggestion list is Constant. Select the Constant option to open the Constant window. In the Constant window, as shown in Figure 7-35, you can enter the value that needs to be assigned to the target item. To assign the value `'00X0'`, enter '00X0' in the Constant window and click **OK**.

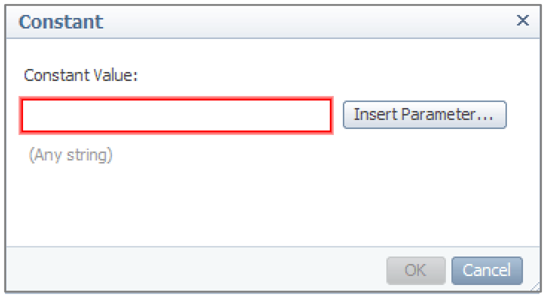
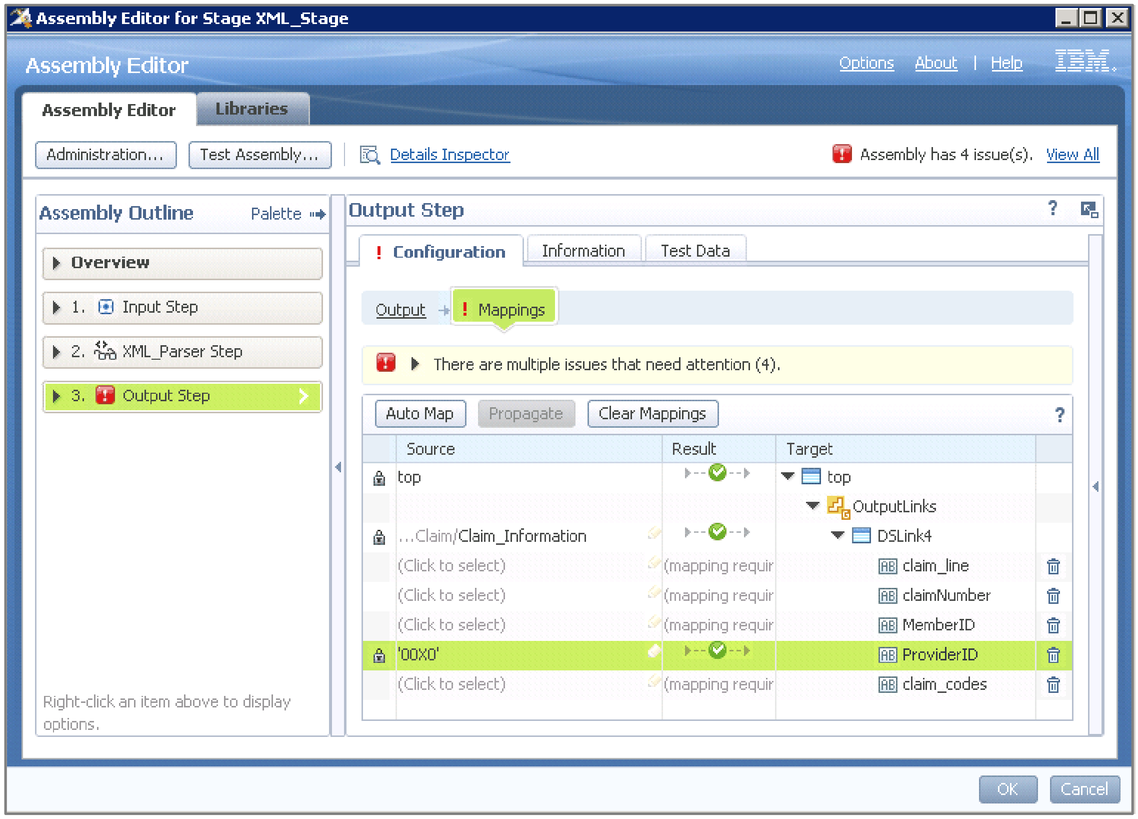


Figure 7-35 The Constant window

Figure 7-36 shows a constant value assigned to the Target column ProviderID.



Source	Result	Target
top		top
		OutputLinks
...Claim/Claim_Information		DSLink4
(Click to select)	(mapping requir	claim_line
(Click to select)	(mapping requir	claimNumber
(Click to select)	(mapping requir	MemberID
'00X0'		ProviderID
(Click to select)	(mapping requir	claim_codes

Figure 7-36 Mappings table after assigning a constant value to a column

7.2.4 Selecting mapping candidates by using the More option

When many columns are present in the output of the XML Parser Step, the suggestion list cannot list all these columns. By default, only five source items can be seen in the suggestion list. If the required column is not available in the suggestion list, use the *More* option. The More option shows the entire source tree structure from which the Source column to be mapped can be selected. Follow these steps to use the More option:

1. Select the row in the Source column that needs to be mapped to the Target column.
2. The More option is available at the end of the suggestion list, as shown in Figure 7-37 on page 182. Select **More**.

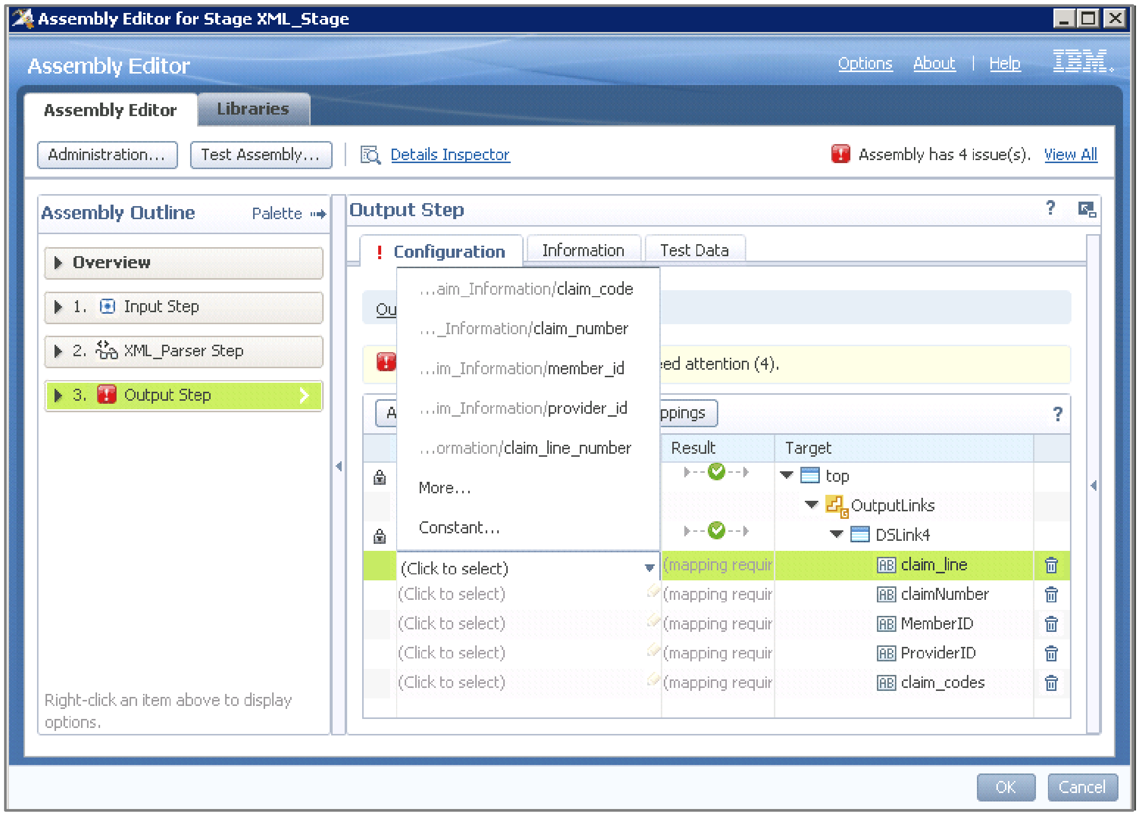


Figure 7-37 The More option in the suggestion list

3. The Target window opens, as shown in Figure 7-38 on page 183. In this window, you can see the entire schema tree and select the item to map.

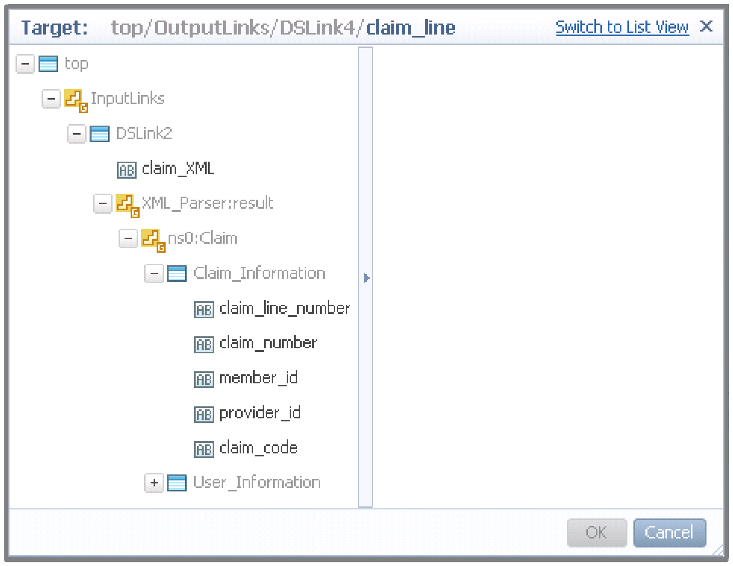


Figure 7-38 The Target window

4. Select the item to map and click **OK**.

The mapping is complete.

7.3 Mapping suggestions

In the Output step, particular mappings must be performed in certain ways. Two scenarios are commonly used conditions. We describe these conditions in the following sections.

7.3.1 Mapping child and parent items to same output list

In the Output step, to map a source item under a list, the list needs to be mapped to the output link. For example, in Figure 7-39 on page 184, to map the source item claim_line_number to the Target column claim_line, the list Claim_information needs to be mapped to the output link DSLink4. If User_Information is mapped to the output link, only the items under this link can be mapped to the target items and claim_line_number cannot be mapped to the target item.

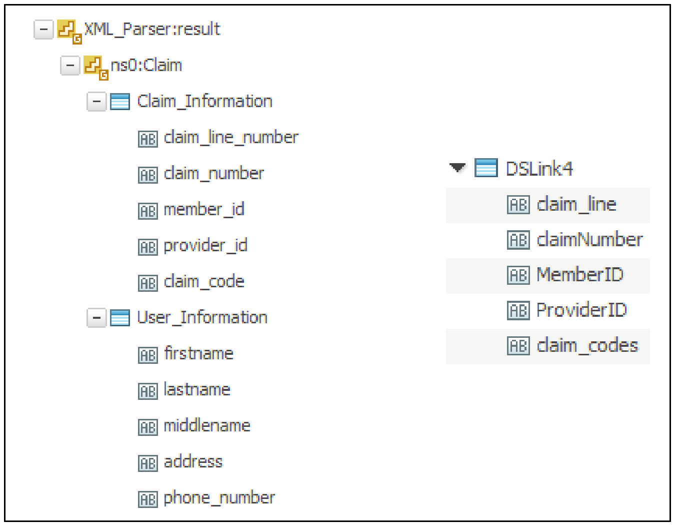


Figure 7-39 The source and target structures

In certain instances, you might want to map the parent and child fields to columns under the same output link in the Output step. This way, you relate the parent and child columns in the output file or database. For example, in Figure 7-40, the source items claim_type and claim_line_number can be mapped to the columns under the same output link. These source items are not under the same list. For example, the claim_type item is the child of Claim_details, and Claim_details is the parent of Claim_Information.

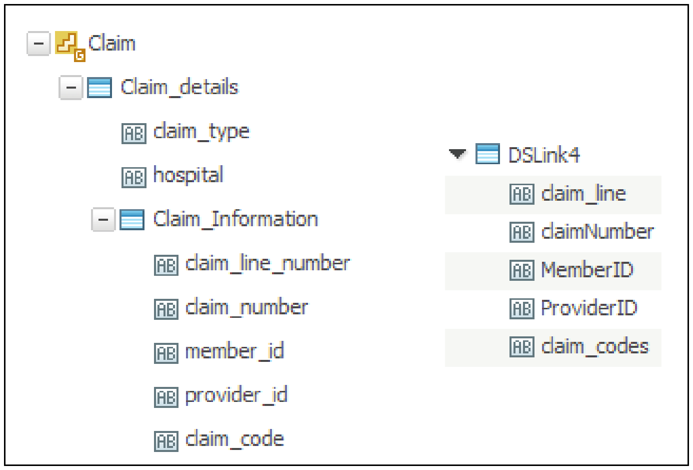


Figure 7-40 The source and target structures

To map two columns under different lists to the same output link, the innermost list is mapped to the output link. Therefore, the list Claim_Information is mapped to the output link DSLink4. Now, you can map both items, claim_line_number and claim_type, to the appropriate output columns. If the list Claim_details is mapped to the output link, only the item claim_type can be mapped to the output column. You cannot map claim_line_number.

It is important that the lists Claim_details and Claim_Information share a parent-child relationship. If these lists are in the same level, such as the lists Claim_Information and User_Information in Figure 7-39 on page 184, the columns under these lists cannot be mapped to a single output link.

7.3.2 Mapping when no lists are available in the Parser step

Certain schemas do not include lists. That is, no element occurs more than one time in the schema. For example, the schema that is shown in Figure 7-41 does not show any multiple occurrences of the same element. You want to map the items of this XML schema to the columns of a database by using the Output step.

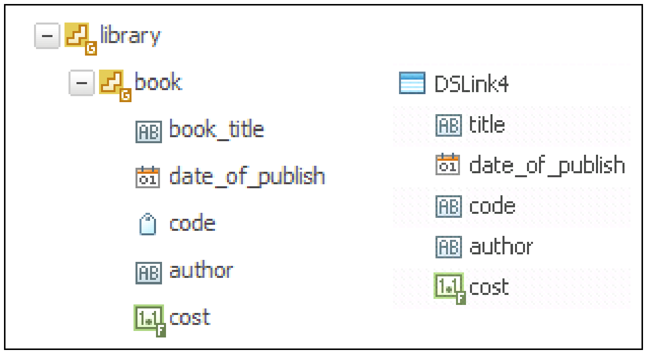


Figure 7-41 The source and target structures

To map the source columns to the target columns, first the source list and target list need to be mapped. But as you can see in Figure 7-41, no lists show in the output of the XML Parser step. To map them, use the following options.

Group to List projection

The *Group to List* projection is available in the Output tab of each step as part of the other projections, such as Drop or Rename. The Group To List projection can apply to a group node (an element with a brown icon) only. For more detail, see Chapter 5, “Schema Library Manager” on page 113. This option converts the group into a list of one.

You can use the Group to List projection to create a list in the output schema of the XML Parser Step. Follow these steps:

1. Right-click the group **XML_Parser:result** in the Output tab of the XML Parser Step. The Group to List projection is visible, as shown in Figure 7-42.

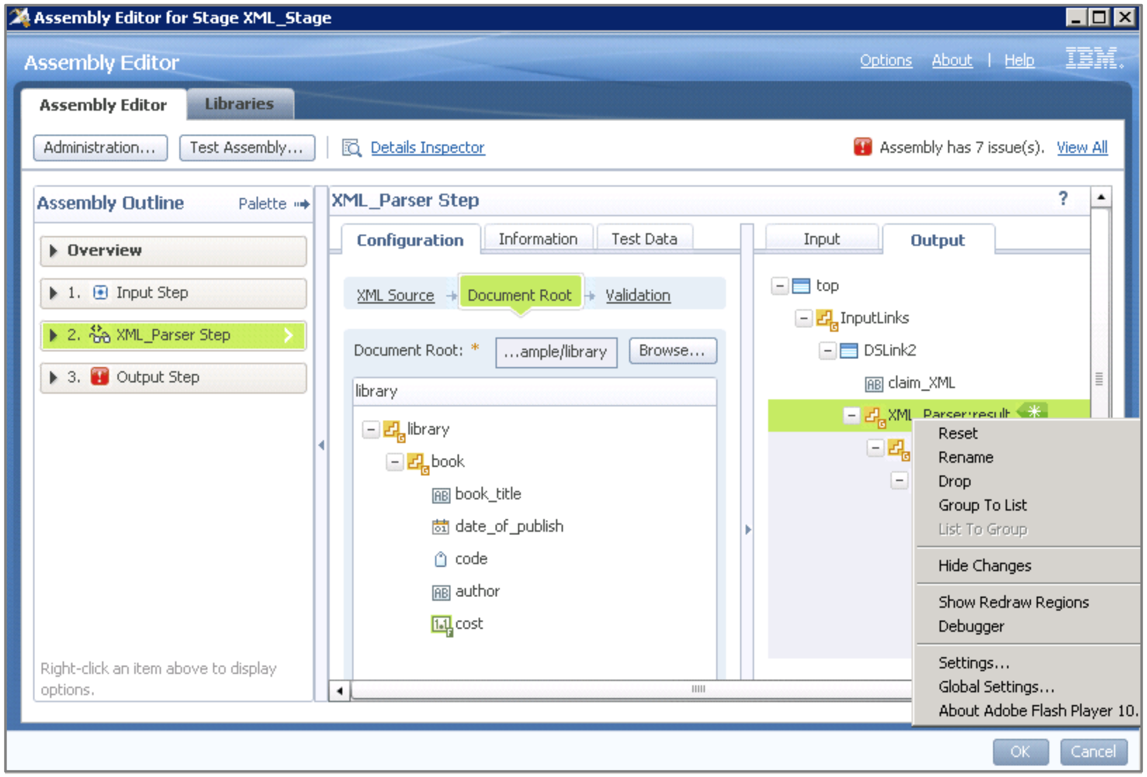


Figure 7-42 The Group to List projection

2. Select the **Group To List** projection. The group changes into a list, as shown in Figure 7-43 on page 187.

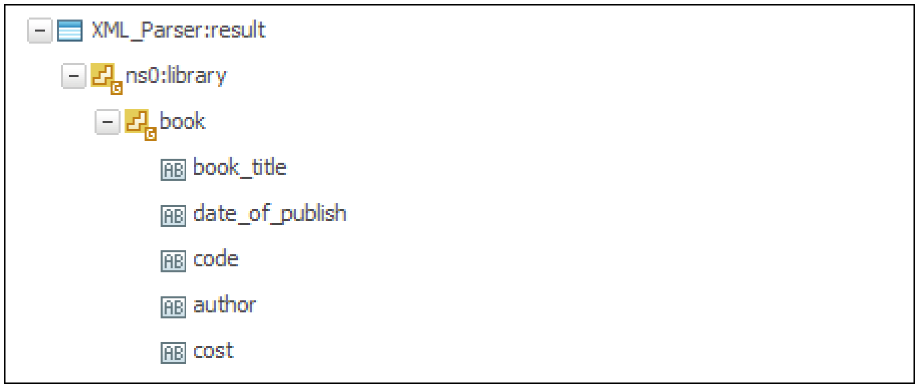
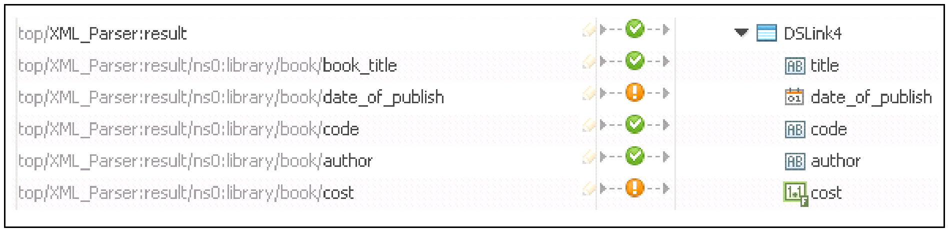


Figure 7-43 The XML_Parser:result as a list

3. In the Output step, XML_Parser:result can be mapped to the output list DSLink4, as shown in Figure 7-44.



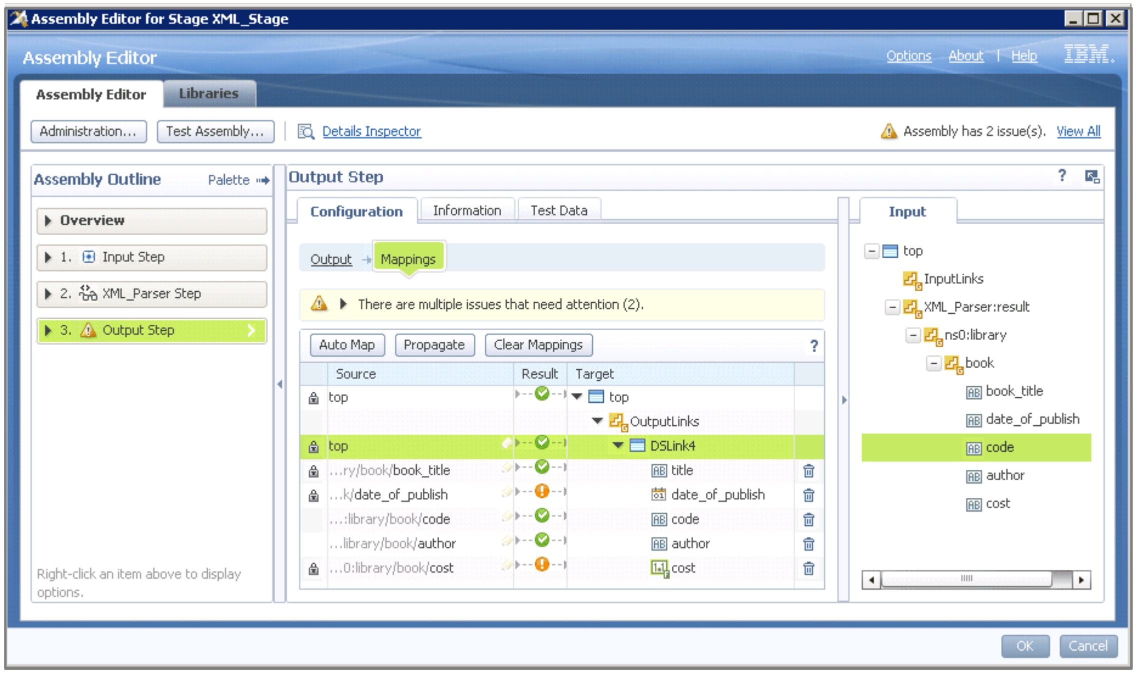
top/XML_Parser:result	DSLink4
top/XML_Parser:result/ns0:library/book/book_title	title
top/XML_Parser:result/ns0:library/book/date_of_publish	date_of_publish
top/XML_Parser:result/ns0:library/book/code	code
top/XML_Parser:result/ns0:library/book/author	author
top/XML_Parser:result/ns0:library/book/cost	cost

Figure 7-44 The complete mapping between the source and target structures

Mapping top to the output list

The entire input and output tree in the assembly is present under the list `top`. The list top is the parent list in the assembly. The result of the XML Parser step is also present under the list `top`.

When the Single File option is selected in the XML Source tab of the XML Parser Step, the output of the XML Parser Step is directly under the list `top`. Therefore, in the Output Step, the list `top` can be mapped to the output link to map the items under the XML Parser Step, as shown in Figure 7-45 on page 188. The warnings in the mapping are the result of the difference in data types between the Source column and the Target column.



Assembly has 2 issue(s).

Right-click an item above to display options.

There are multiple issues that need attention (2).

Source	Result	Target
top	✓	top
		OutputLinks
top	✓	DSLink4
...ry/book/book_title	✓	title
...k/date_of_publish	!	date_of_publish
...:library/book/code	✓	code
...library/book/author	✓	author
...0:library/book/cost	!	cost

Figure 7-45 The Mappings table

7.4 Parsing large schemas

Many schema types are available. Parsing the types of schemas works the same way as the XML Parser step. However, you must change the job design for large schemas.

The XML Stage is a schema-driven stage. To parse any XML document, the schema needs to be defined in the Document Root tab of the XML Parser Step. This XML schema is converted into an internal model by the XML Stage. When large schemas are used, a large amount of memory is required to store the schema in the internal XML Stage model, because every aspect of the schema needs to be saved. Therefore, for large schemas, the representation of the XML schema in the assembly is changed.

For large schemas, several elements are automatically *chunked*. Chunking refers to converting all child items under an element to a single item.

The leftmost diagram in Figure 7-46 on page 189 shows the actual schema representation. When this schema is part of another large schema, the element Claim_Information might be automatically chunked. As shown in Figure 7-46 on page 189, the auto-chunked element is represented with the symbol <>. All child items of the element Claim_Information are contained in this element.

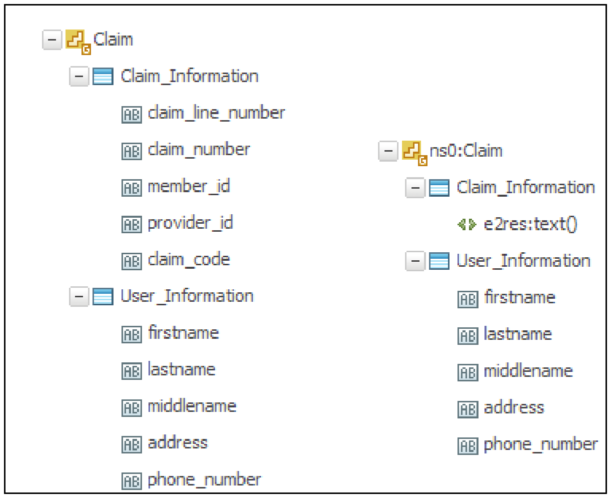


Figure 7-46   Schema structure when a node is auto-chunked

When an element is automatically chunked, its child items are not available for mapping in the Output step. For example, you cannot map the element claim_line_number to the Target column in the Output step, because the Output step sees only the single element e2res:text() under Claim_Information.

To parse the automatically chunked elements, another XML Parser step is required in the assembly. The second XML Parser step is used to parse the auto-chunked element so that the child elements can be available for mapping in the Output step.

Follow these steps to parse the auto-chunked element:

1. Add a second XML Parser step to the assembly.
2. In the second Parser step, use the String Set option to read the auto-chunked element. Select the **String Set** option in the XML Source tab. From the drop-down list, select **e2res:text()**, which is the auto-chunked node, as shown in Figure 7-46. The String Set option reads the XML data from the first XML Parser step.
3. As soon as the String Set option is configured, the Document Root is automatically set by the XML Stage. Therefore, you do not need to configure the Document Root tab for the second XML Parser step. The Document Root is the same structure as the auto-chunked node, as shown in Figure 7-47 on page 190.

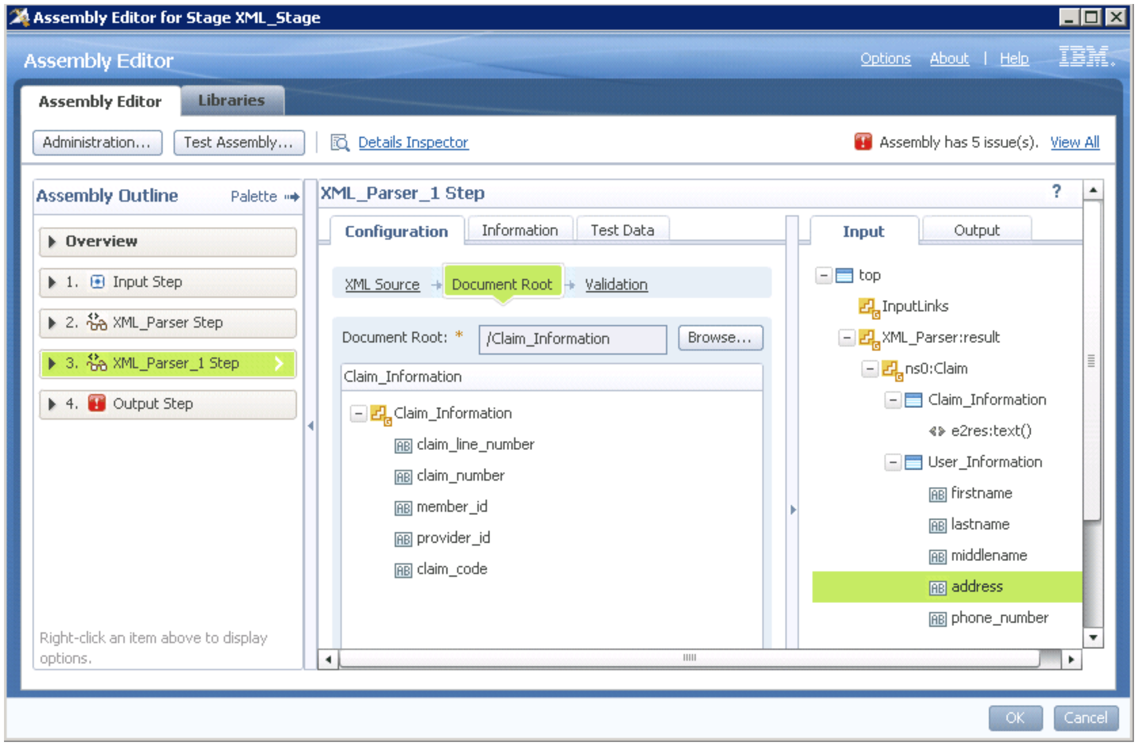


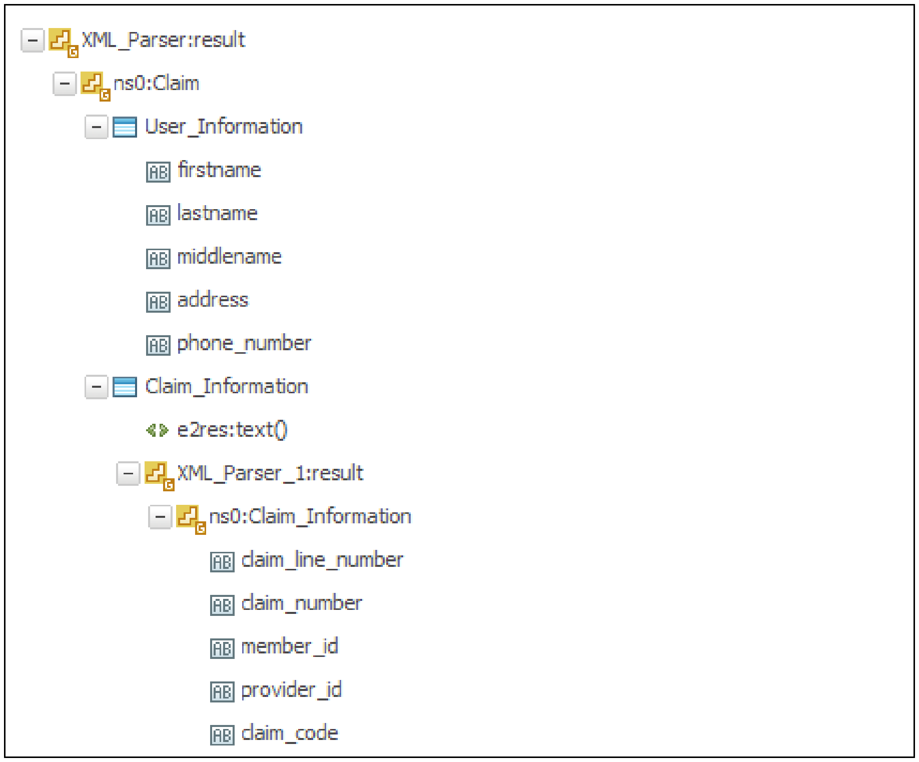
Figure 7-47 Document Root tab that shows the structure of the auto-chunked element

After configuring the second XML Parser step, the child items of the auto-chunked element are available for mapping in the Output step.

7.4.1 Output schema location of the second XML Parser step

The output of the second XML Parser step is under the parent list of the auto-chunked element, as shown in the following examples:

- Look at the auto-chunked element under the list Claim_Information in Figure 7-48 on page 191. The output of the second XML Parser step shows under the list Claim_Information, because Claim_Information is the parent list of the auto-chunked node.



```
XML_Parser:result
  ns0:Claim
    User_Information
      firstname
      lastname
      middlename
      address
      phone_number
    Claim_Information
      e2res:text()
      XML_Parser_1:result
        ns0:Claim_Information
          claim_line_number
          claim_number
          member_id
          provider_id
          claim_code
```

Figure 7-48 Output schema of the second XML Parser step

- Assume that the auto-chunked element, book, is under the group library. The parent list of the element book is `top`. Therefore, the output of the second XML Parser step is placed under the list `top`, as shown in Figure 7-49 on page 192.

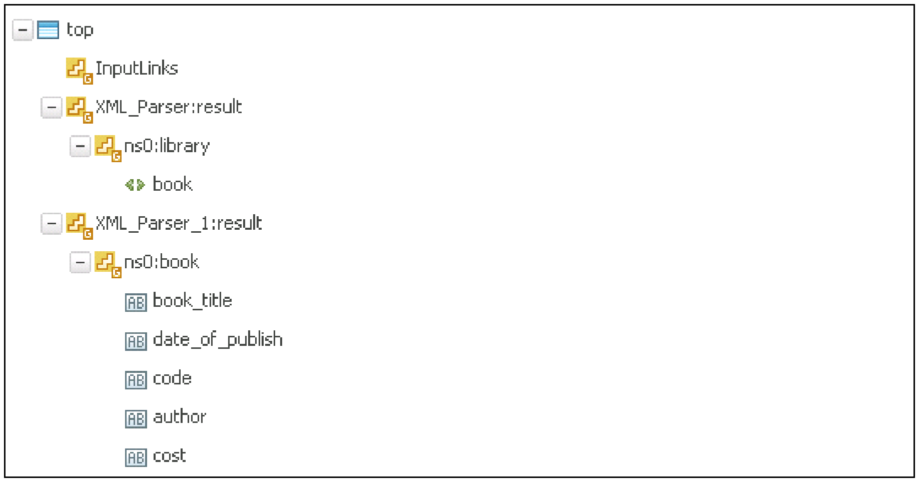

```
top
    InputLinks
  XML_Parser:result
    ns0:library
      book
  XML_Parser_1:result
    ns0:book
      book_title
      date_of_publish
      code
      author
      cost
```

Figure 7-49   Output schema of the second XML Parser step

- Assume that the auto-chunked element phone_number is under a group address. This group is under the Employee list. Because the parent list of the element phone_number is Employee, the output of the second XML Parser step is under the Employee list, as shown in Figure 7-50.

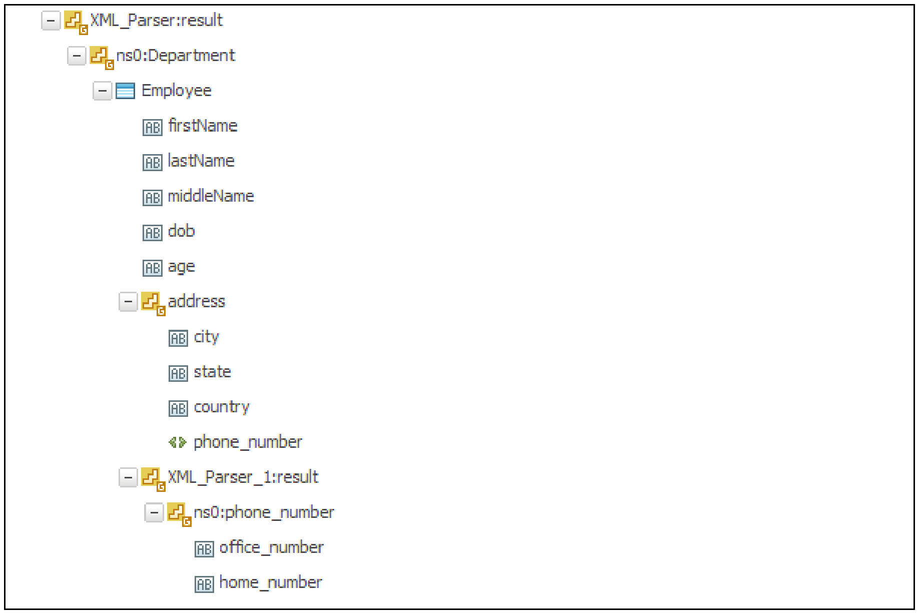
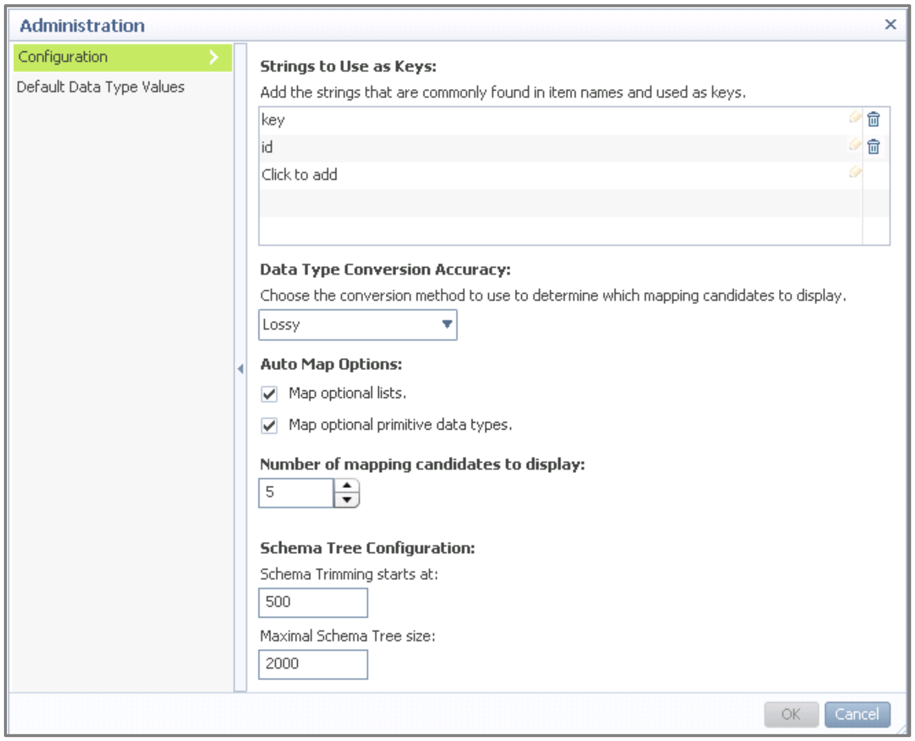

```
XML_Parser:result
  ns0:Department
    Employee
      firstName
      lastName
      middleName
      dob
      age
      address
        city
        state
        country
        phone_number
      XML_Parser_1:result
        ns0:phone_number
          office_number
          home_number
```

Figure 7-50   Output schema of the second XML Parser step

7.4.2 Configuring the auto-chunked elements in the assembly

While configuring the Document Root in the XML Parser step, select the top-level element from the Schema Library Manager. The top-level element is in the form of a schema tree, containing schema elements in a hierarchical structure. For every top-level element, the XML Stage maintains a count of the number of schema elements imported into the assembly.

Two fields are defined in the Administration tab, as shown in Figure 7-51. You can use these fields, `Schema Trimming starts at` and `Maximal Schema Tree size`, to specify thresholds on the expected size of schema trees that need to be imported. When the number of imported elements reaches these thresholds, XML Stage starts auto-chunking the schema elements. These thresholds are soft limits.



Strings to Use as Keys:
Add the strings that are commonly found in item names and used as keys.

key
id
Click to add

Data Type Conversion Accuracy:
Choose the conversion method to use to determine which mapping candidates to display.
Lossy

Auto Map Options:
✓ Map optional lists.
✓ Map optional primitive data types.

Number of mapping candidates to display:
5

Schema Tree Configuration:
Schema Trimming starts at:
500
Maximal Schema Tree size:
2000

Figure 7-51 The Administration Configuration tab

The XML Stage starts to chunk the top-level XML elements when the tree size reaches the value for the `Schema Trimming starts at`. When the tree size reaches `Maximal Schema Tree size`, the XML Stage auto-chunks all the qualified

XML elements (not only the top-level XML elements, but also nested XML elements).

If you do not use a large schema inside the assembly, there is no user behavior change. The field `Schema trimming starts at` defaults to `500`, and the field `Maximal Schema Tree size` defaults to `2000`. For schemas whose size is smaller than 500, XML Stage does not auto-chunk any schema elements. When you need to import a large schema into the assembly, configure the two fields in the Administration tab and use multiple parsers to extract the auto-chunked elements.

Because the values are defined inside the Assembly Editor, they apply to the particular assembly in use. When the schema is defined in the Document Root tab of the XML Parser step, the representation of the schema depends on the values for the Schema Tree Configuration fields at that instance. After defining the schema in the Document Root of the XML Parser step, the values can be changed. The changed values do not affect the schema representation in the XML Parser steps in which the Document Root is already defined. Therefore, in a single assembly, each XML Parser step can be configured to use different configuration values.

7.4.3 Designing the assembly

When the auto-chunked element is selected in the second XML Parser step, the Document Root is automatically defined. The Document Root is automatically defined if the first and second XML Parser steps are defined one after another without saving the assembly in between. But, in case, the Assembly Editor is closed after defining the first XML Parser step, the Document Root is not automatically defined in the second XML Parser step when it is added to the assembly on reopening the Assembly Editor.

For example, suppose that you configure the first XML Parser step, save the Assembly Editor by closing it, and then reopen it. When you add the second XML Parser step to parse the auto-chunked element, the Document Root is not automatically defined. You must go to the first parser and set the Document Root again. You do not need to delete or reconfigure the first parser. You merely need to reset the Document Root.

This step also applies to a scenario with several parsers in the assembly. The first XML Parser is used to import the whole schema. The rest of the Parsers are used to parse several auto-chunked elements from the output of the first parser. If you reopen the job to add another parser to parse another auto-chunked schema element, you need to reset the Document Root in the first parser.

7.5 Parsing only a section of the schema

Occasionally, only a part of an XML document must be written into the output, along with a particular XML element. The healthcare company stores most of the XML files in the database. However, the company wants to extract an element from the XML file and then store the XML file and the extracted element in two columns of another database table.

Use the XML Parser step to parse the XML file, extract a single element from it, and write the XML file along with the column back into the database.

Use the *chunk* operation in the XML Parser step to perform these operations. This operation is manual as opposed to the automatic chunk described in 7.4, "Parsing large schemas" on page 188.

The chunk feature is available in the Document Root tab of the XML Parser step. You can select any element in the Document Root window and right-click that element to activate the chunk option, as shown in Figure 7-52 on page 196.

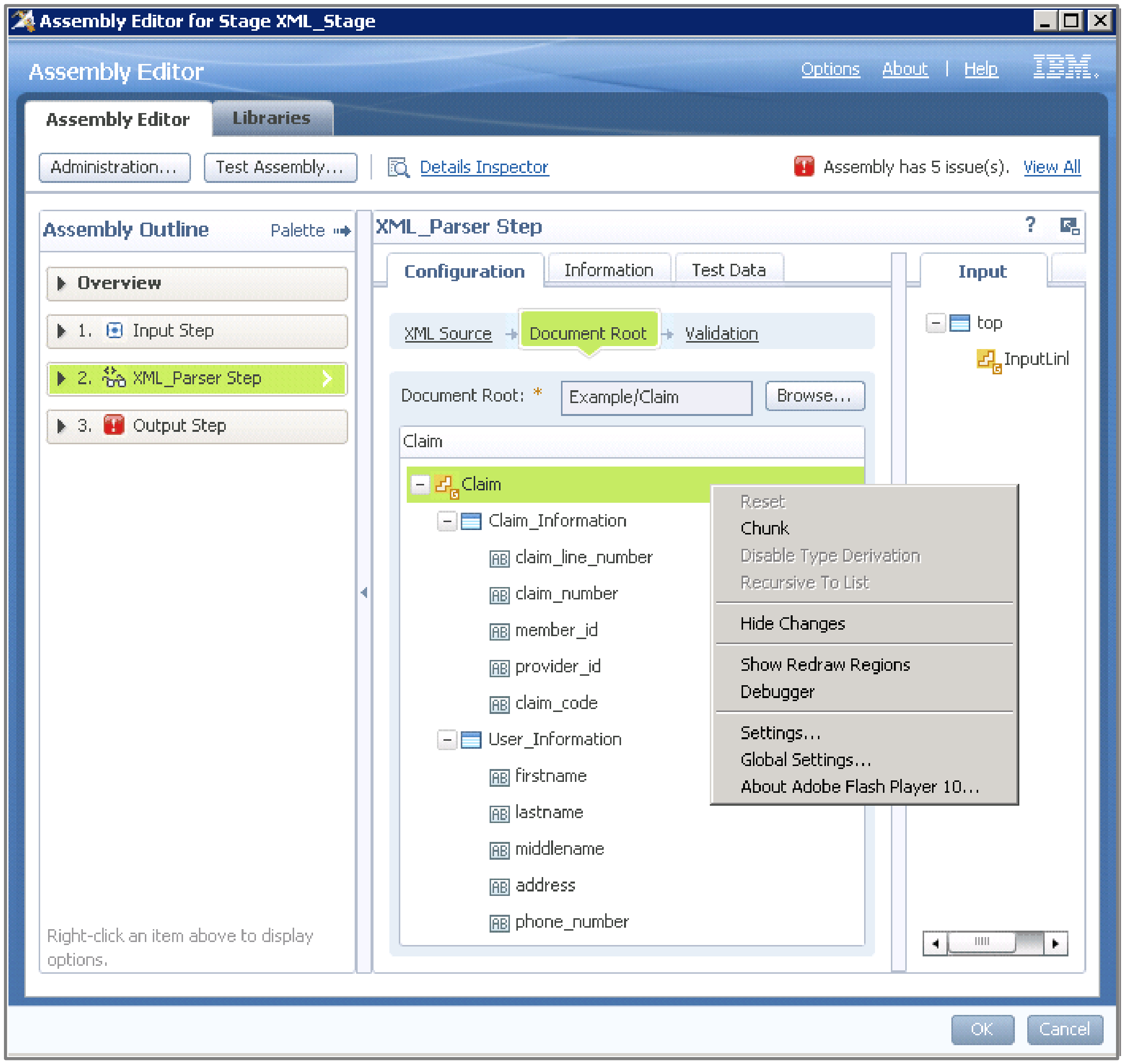


Figure 7-52 The Chunk option in the Document Root tab of the XML Parser Step

Chunking refers to converting all child items under an element to a single item. In Figure 7-53 on page 197, we show the actual schema representation. When the element Claim is manually chunked, it is represented with the symbol <>. All the child items of the element Claim are contained in this single element, as shown in Figure 7-53 on page 197.

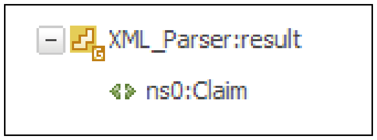

XML_Parser:result
ns0:Claim

Figure 7-53 Manual chunk representation

To extract the item claim_number, the chunk Claim needs to be passed through a second XML Parser step. Use the following steps to parse the element Claim:

1. Add a second XML Parser step to the assembly.
2. In the second XML Parser step, use the String Set option to read the chunked element. Select the **String Set** option in the XML Source tab. From the drop-down list, select **Claim**. The String Set option reads the XML data from the first XML Parser step.
3. As soon as the String Set option is configured, the Document Root is automatically set by the XML Stage. Therefore, you do not need to configure the Document Root tab for the second XML Parser step.

After configuring the second Parser step, the chunked element and its child items are available for mapping in the Output step, as shown in Figure 7-54. You can map the items Claim and claim_number to two columns in the Output step.

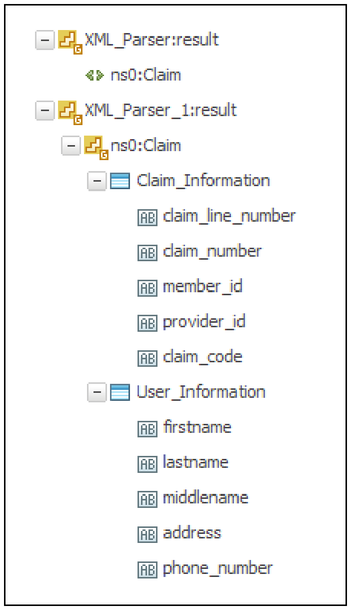

XML_Parser:result
ns0:Claim
XML_Parser_1:result
ns0:Claim
Claim_Information
claim_line_number
claim_number
member_id
provider_id
claim_code
User_Information
firstname
lastname
middlename
address
phone_number

Figure 7-54 Output available to the Output step

The output of the second XML Parser step is under the parent list of the chunked element. See 7.4.1, “Output schema location of the second XML Parser step” on page 190 for a detailed description of the location of the output schema for various types of schemas.

For auto-chunked elements, to get the Document Root automatically defined in the second XML Parser step, the first and second Parser steps need to be defined at the same time in the assembly. However, the first and second Parser steps do not need to be defined at the same time in the assembly for manually chunked elements. Even when you save and reopen the assembly, the Document Root gets automatically defined in the second XML Parser step.

7.6 Incorporating an XSLT style sheet during parsing

The Assembly Editor does not require you to create an Extensible Stylesheet Language Transformation (XSLT) style sheet, but instead, it provides a more intuitive way to describe the transformation. However, there are circumstances when you need to first employ an XSLT style sheet on the data and then parse it and map it in the assembly. We do not suggest this approach for large documents. The current XSLT technology requires the document to fit in memory. It therefore cancels the advantage of the assembly that can parse any document size.

However, if necessary, the assembly can first perform the XSLT transformation, and it reads the result of the XSLT as though it is the content of the original file. Therefore, the Document Root element reflects the result of the XSLT and not the original XML input document, as shown in Figure 7-55.

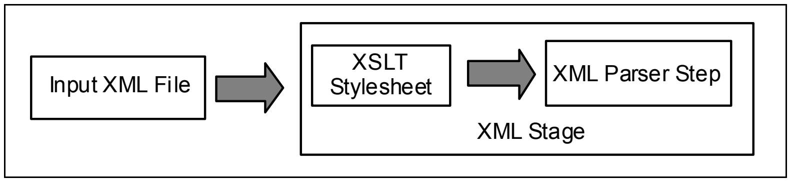
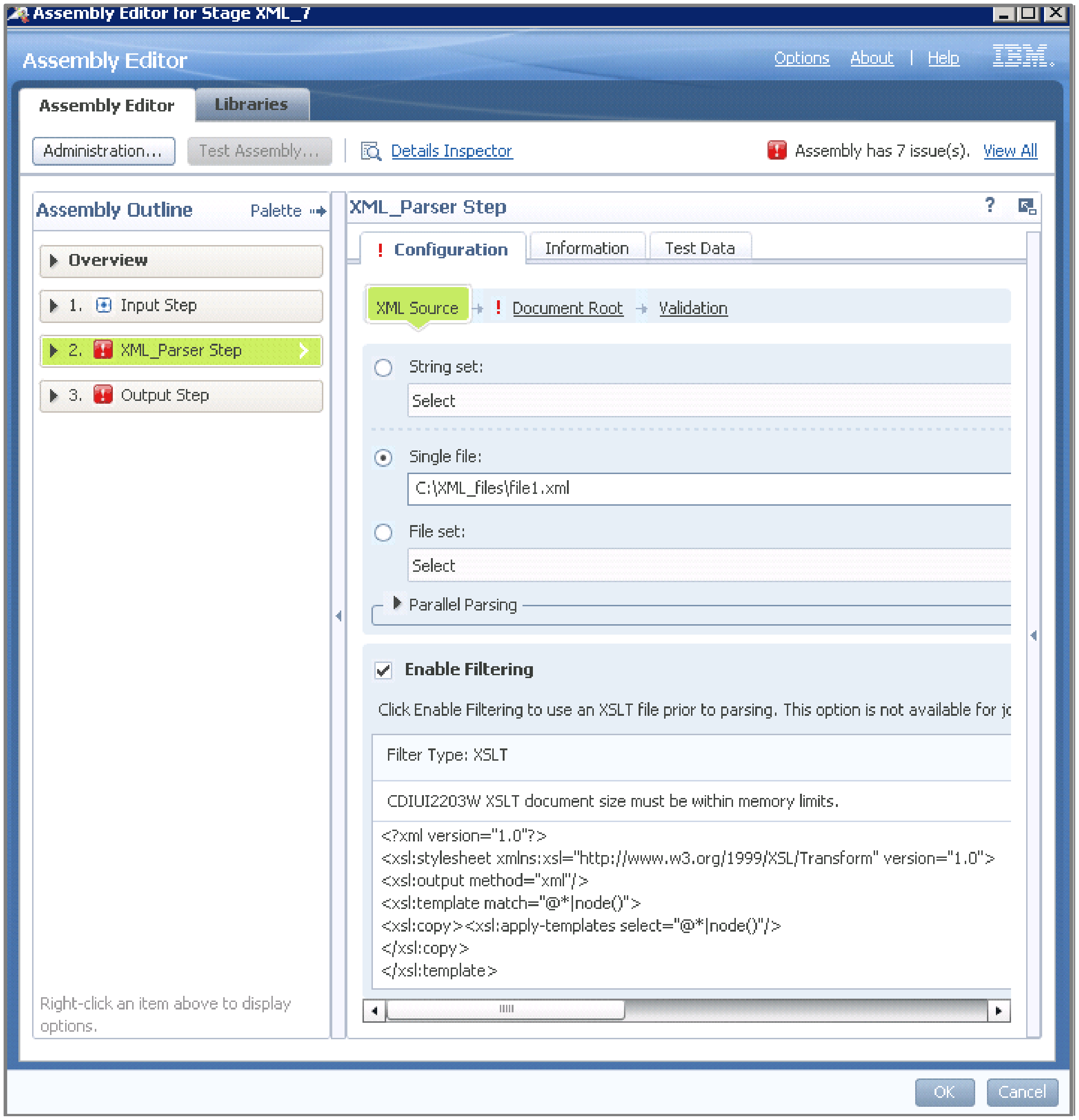


Figure 7-55 Processing XML documents that incorporate XSLT style sheets

You must paste the XSLT style sheet in the Enable Filtering section of the XML Source window in the XML Parser step. To enable the Enable Filtering option, first the XML Source needs to be defined, and then, you must select **Enable Filtering**. A default XSLT style sheet shows in the area under the Enable Filtering option, as shown in Figure 7-56 on page 199. Without selecting an XML Source, the Enable Filtering option is not enabled. You can replace the default style sheet with a style sheet of your choice.



Assembly has 7 issue(s). View All

String set: Select

Single file: C:\XML_files\file1.xml

File set: Select

Enable Filtering

Click Enable Filtering to use an XSLT file prior to parsing. This option is not available for jc

Filter Type: XSLT

CDIUI2203W XSLT document size must be within memory limits.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="xml"/>
<xsl:template match="@*|node()">
<xsl:copy><xsl:apply-templates select="@*|node()"/>
</xsl:copy>
</xsl:template>
```

Figure 7-56 Enable Filtering option in the XML Source tab of the XML Parser Step

The healthcare company uses several XML files that do not conform to the XML schema. That is, the top level of the XML file and the schema do not match. But the company still wants to parse these files by using the XML Stage. To ensure that the XML documents conform to the XML schema, the company wants to apply an XSLT style sheet to the XML documents. The XSLT must be pasted in the Enable Filtering area in the XML Source tab of the XML Parser Step. And, the output of the XSLT must conform to the XML schema defined in the Document Root of the Parser Step, as shown in Figure 7-55 on page 198.

8

Creating and composing XML documents

Our fictitious example bank is in New York. The bank was established in the year 1990 and provides services for many global IT companies. The bank has over a million account holders. The bank stores all customer information in a database. The bank sends a report to its customers every quarter. This report summarizes the activity of the customers in that quarter. The application that creates these reports needs all the information in an XML document.

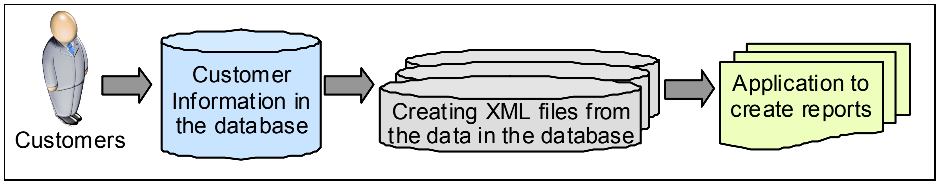

Customers
Customer Information in the database
Creating XML files from the data in the database
Application to create reports

Figure 8-1 The bank configuration

8.1 The XML Composer step

The bank stores all its customer information in a database. By using the data in the database tables, an XML file needs to be created to be input to the application that creates the reports. The bank already uses IBM Information Server for managing its business requirements and analyzing its business terms. Therefore, the bank wants to use InfoSphere DataStage to create the XML files.

8.1.1 Writing XML files

To create the XML files, the XML Output stage or the XML Stage can be used in the DataStage Designer. Because it is easier to create large XML documents with multiple levels of hierarchy by using the XML Stage, XML Stage is the preferred choice for creating the XML files. In the XML Stage, you use the Composer step to create the XML files. To add the Composer step to the Assembly Editor, follow these steps:

1. Open **Assembly Editor** and click **Palette** in the Assembly Outline bar. The steps show in the Palette, as shown in Figure 8-2 on page 203.

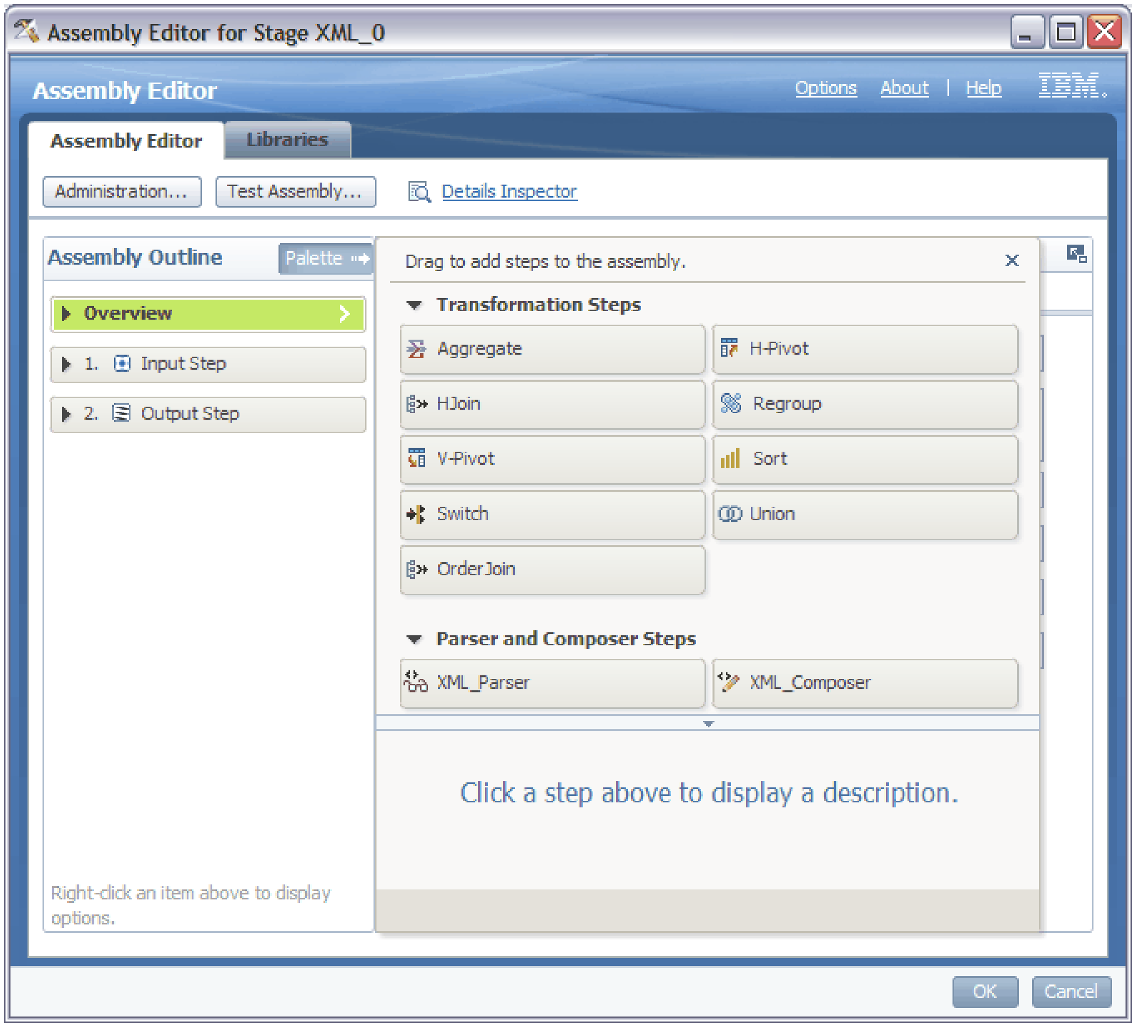
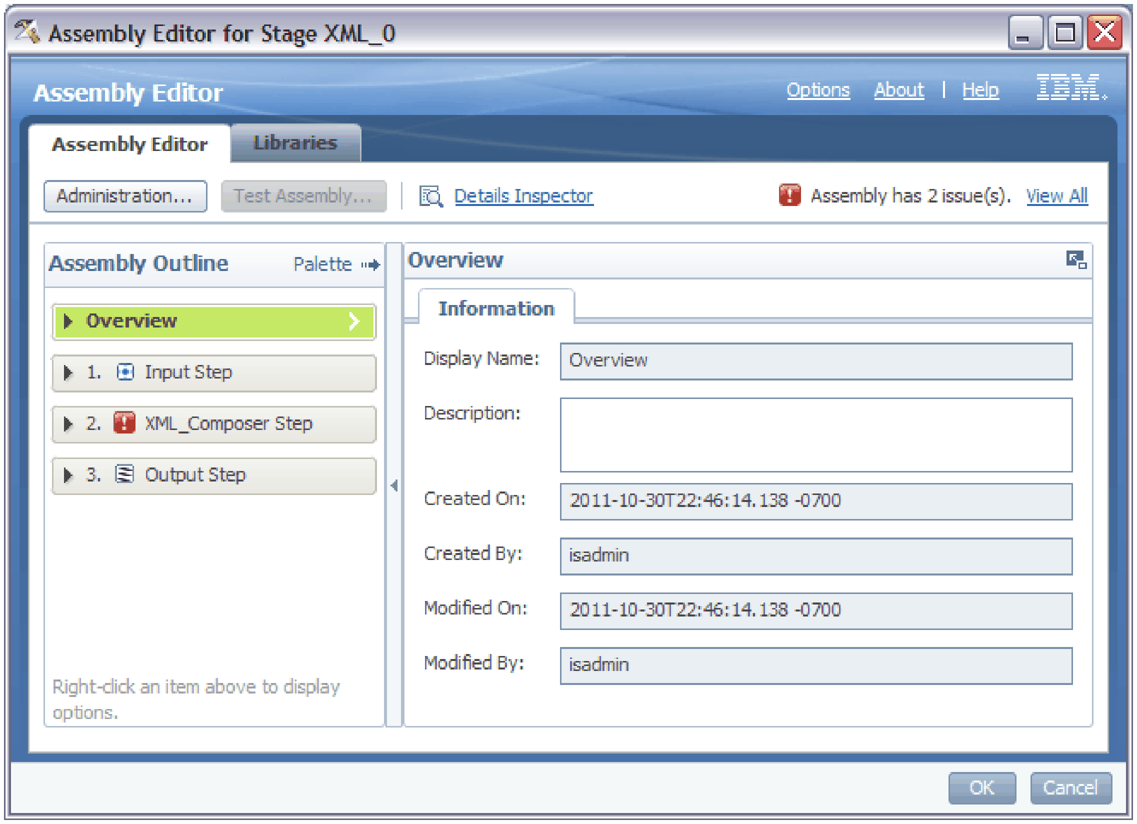


Figure 8-2 The Palette in the Assembly Editor

2. Double-click **XML_Composer** in the Palette. The XML_Composer Step is added between the Input Step and the Output Step, as shown in Figure 8-3 on page 204.



Assembly Editor for Stage XML_0

Assembly has 2 issue(s). View All

Right-click an item above to display options.

Display Name: Overview

Description:

Created On: 2011-10-30T22:46:14.138 -0700

Created By: isadmin

Modified On: 2011-10-30T22:46:14.138 -0700

Modified By: isadmin

Figure 8-3 The XML_Composer Step added to the Assembly Editor

The bank uses IBM DB2 as the database storage. The customer information is stored in multiple columns in a single database table. Therefore, in the DataStage job, the source of input data is DB2. In the DataStage Designer canvas, the DB2_Connector_Stage can be used to read the relational columns from the database. The output of the DB2_Connector_Stage must be fed to the XML Stage. The overall job design is shown in Figure 8-4.

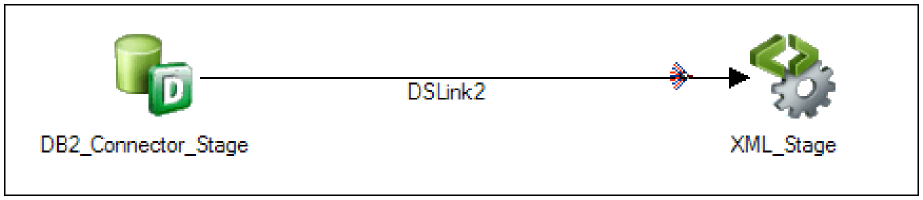
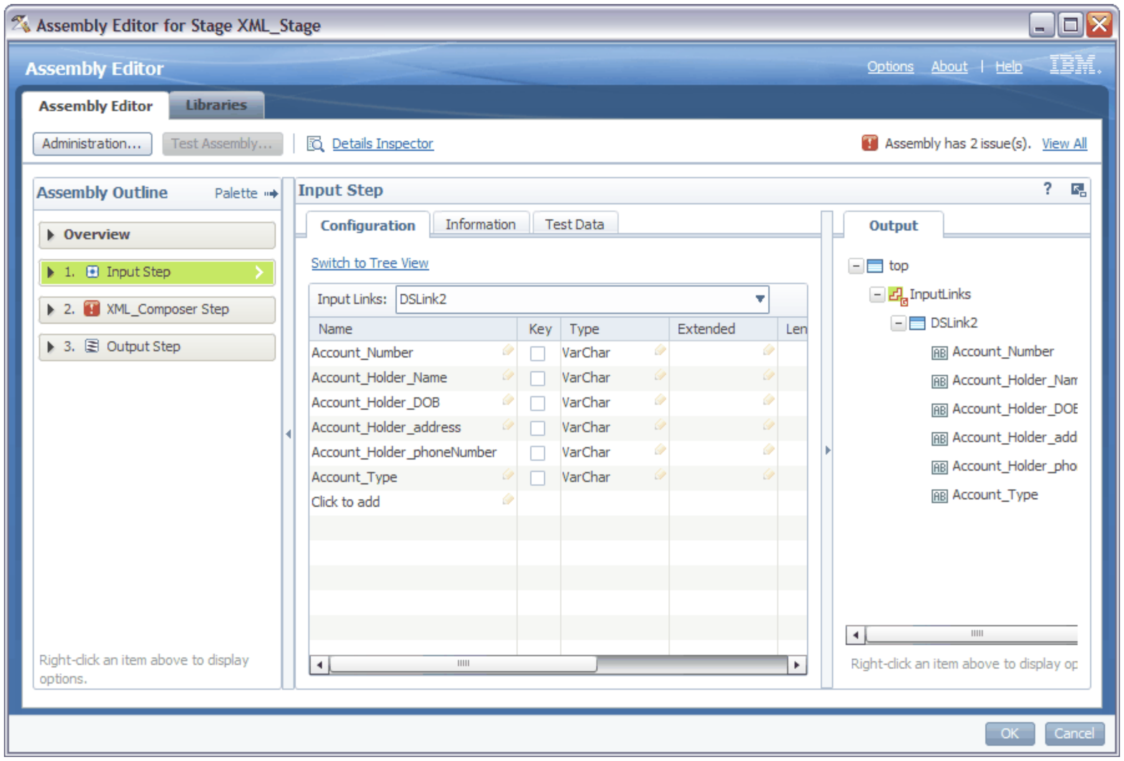


Figure 8-4 The DataStage job to read data from DB2 and create an XML file

The Input Step in the Assembly Editor describes the metadata of a relational structure. The step becomes active when an input link is provided to the XML Stage. You can see the columns that are added to the preceding stage of the XML Stage. The columns can be modified or removed, or new columns can be added. When the input is provided from the DB2 Connector to the XML Stage, the Input Step becomes active and displays the DB2 columns, as shown in Figure 8-5.



Input Links: DSLink2

Name	Key	Type	Extended	Len
Account_Number	☐	VarChar		
Account_Holder_Name	☐	VarChar		
Account_Holder_DOB	☐	VarChar		
Account_Holder_address	☐	VarChar		
Account_Holder_phoneNumber	☐	VarChar		
Account_Type	☐	VarChar		
Click to add				

Figure 8-5 The Input Step in the Assembly Editor

The bank wants to store the output XML files in two kinds of columns in the database: in a pureXML column and in a large object column. The Composer step writes the XML files in these locations. In the XML Composer step, the method of writing the XML file must be specified in the XML Target tab. A single Composer step cannot write the XML data to multiple destinations at a time.

Writing XML files to a pureXML column in the database

The bank wants to store the XML files in a single DB2 pureXML column in the database. In a pureXML column, the XML files can be stored in their native hierarchical format without the need to shred the data into relational format. Applications can readily access the XML data in this column.

In the DataStage Designer canvas, the DB2 Connector stage can be used to write the XML files into the database table. The XML files are stored in a single column xml_data in the database. The DB2 Connector can access the correct database table to write into this particular column. Because the XML Stage needs to pass the created XML files to the DB2 Connector stage, the output of the XML Stage must be given to the DB2 Connector stage, as shown in Figure 8-6.

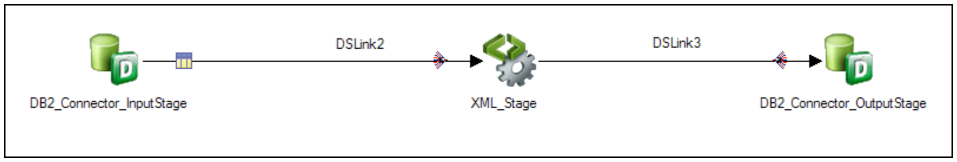
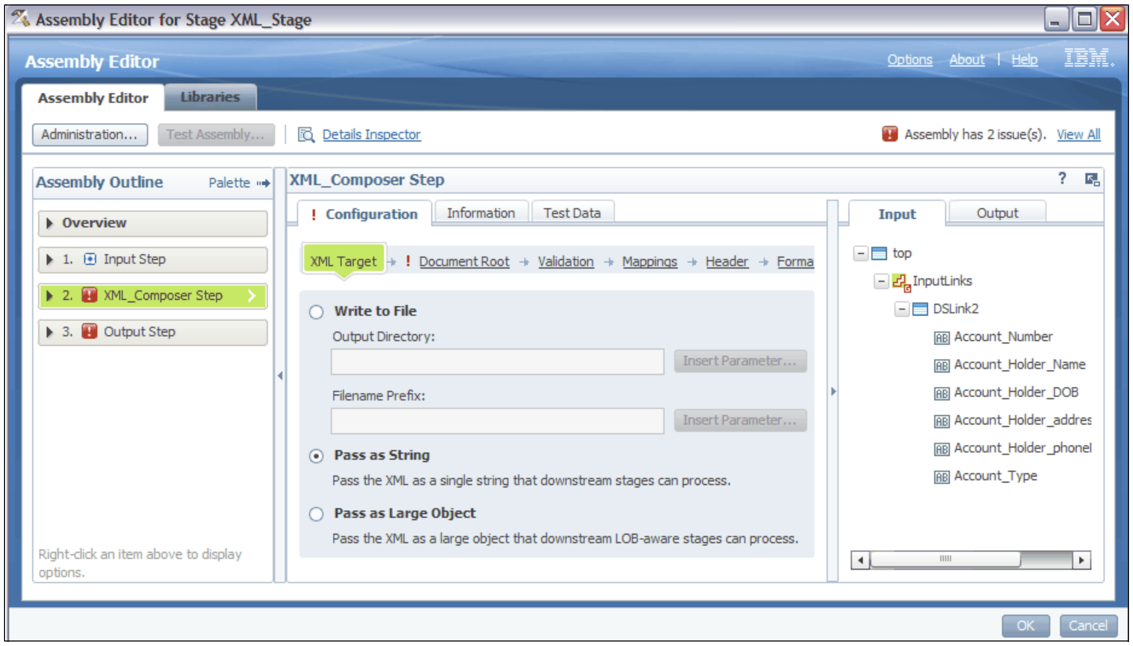


Figure 8-6 DataStage job design: XML file written into pureXML column of the database

To pass the XML files to a downstream stage within DataStage, you must use the Pass as String option in the XML Target tab of the Composer step. When the Pass as String option is used, the Composer step creates the XML files, and another stage within DataStage actually writes these files to the target.

Select **Pass as String** in the XML Target tab, as shown in Figure 8-7 on page 207. The XML Target tab is then configured to pass the created XML files to a downstream stage.



( ) Write to File

Output Directory:

Insert Parameter...

Filename Prefix:

Insert Parameter...

[✓ Pass as String]

Pass the XML as a single string that downstream stages can process.

( ) Pass as Large Object

Pass the XML as a large object that downstream LOB-aware stages can process.

Figure 8-7 The XML Target tab in the Composer Step

Writing XML files to a large object column in the database

The XML files can be stored as large object (LOB) columns in the DB2 database. We use LOB to refer to LOB, BLOB, CLOB, or DBCLOB data types in the database. See 10.6.2, "Passing XML by reference as a large object column" on page 321, where LOB is described in detail.

To pass the XML files to a LOB column in the DB2 Connector stage, the *Pass as Large Object option* must be used in the XML Target tab of the Composer Step. The Pass as Large Object option is used to pass the XML files to a LOB-aware stage. Therefore, the file is created in the Composer step and passed to a LOB-aware stage in DataStage, which actually writes the file to the target. The job design is shown in Figure 8-8.

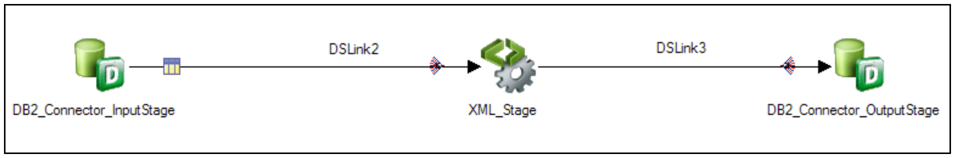
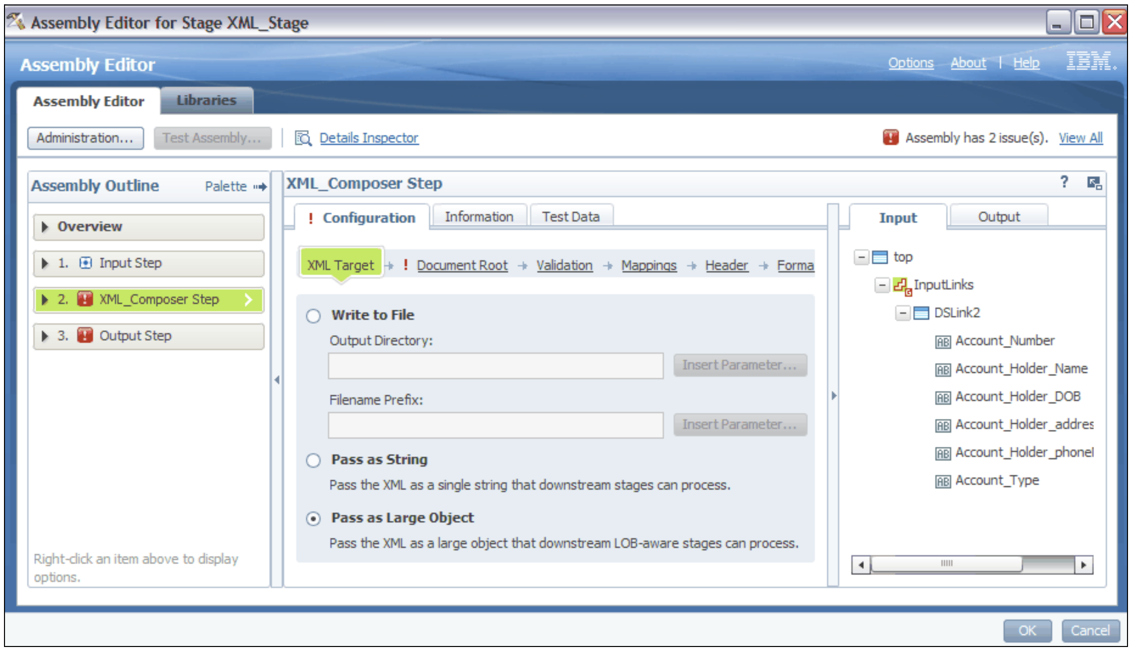


Figure 8-8 DataStage job design: XML file is written to LOB column of the database

Select **Pass as Large Object** in the XML Target tab, as shown in Figure 8-9. The XML Target tab is then configured to pass the created XML files to a downstream LOB-aware stage.



Write to File

Output Directory:

Insert Parameter...

Filename Prefix:

Insert Parameter...

Pass as String

Pass the XML as a single string that downstream stages can process.

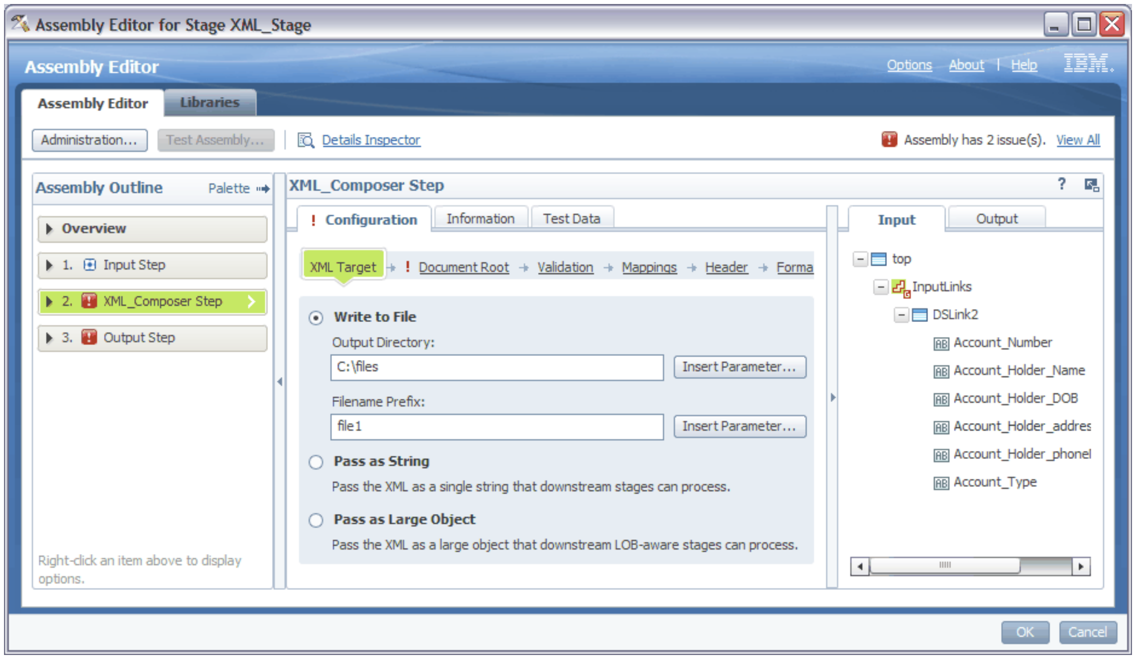
[✓ Pass as Large Object]

Pass the XML as a large object that downstream LOB-aware stages can process.

Figure 8-9 The XML Target tab LOB option in the Composer Step

Writing the XML file from the Composer step

The bank can also configure the XML Composer step to directly write the XML file to the server. The *Write to File* option can be used to write the XML file to the server. Select **Write to File**. You need to specify the absolute location where the XML files are created, as well as the file name. In the Output Directory text box, enter the absolute location where the file is to be created. Enter the XML file name in the Filename Prefix text box. For example, as shown in Figure 8-10 on page 209, to create the XML file `file1.xml` in the `C:\files` directory, you must enter `C:\files` for Output Directory. Enter `file1` for Filename Prefix. When you enter the file name, do not type the extension `.xml`. The extension is added by the Composer step when you create the file.


Assembly Editor for Stage XML_Stage

Assembly Editor

Assembly Outline: Overview; 1. Input Step; 2. XML_Composer Step; 3. Output Step

XML_Composer Step — Configuration | Information | Test Data

XML Target → ! Document Root → Validation → Mappings → Header → Forma

(●) Write to File

Output Directory: C:\files

Filename Prefix: file1

( ) Pass as String

Pass the XML as a single string that downstream stages can process.

( ) Pass as Large Object

Pass the XML as a large object that downstream LOB-aware stages can process.

Assembly has 2 issue(s). View All

Input: top / InputLinks / DSLink2 / Account_Number, Account_Holder_Name, Account_Holder_DOB, Account_Holder_addres, Account_Holder_phonel, Account_Type

Figure 8-10 The XML Target tab Write to file option in the Composer Step

In this job design, the XML Stage is the target stage in the designer canvas. No output is provided from the XML Stage, as shown in Figure 8-11.

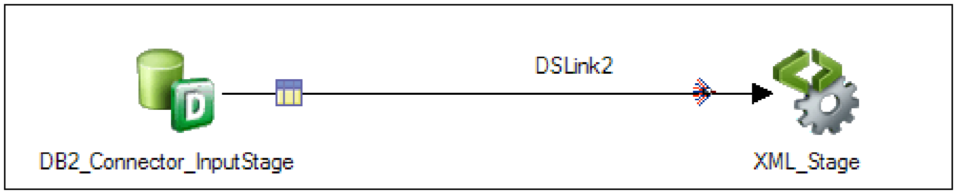
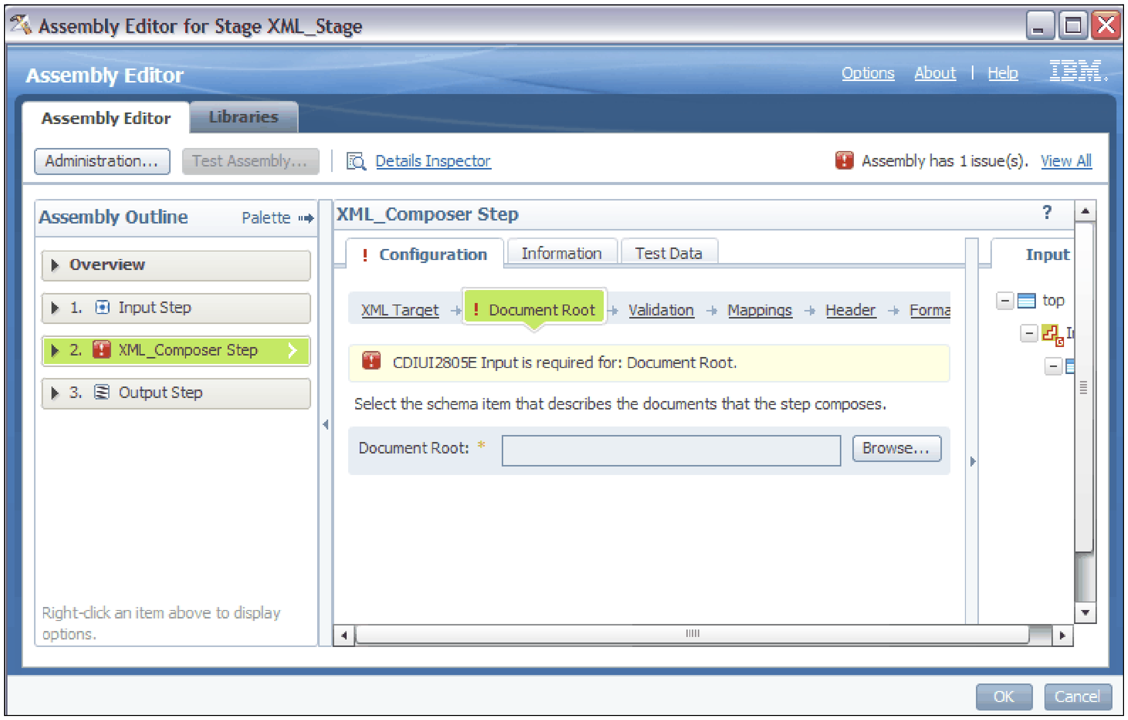


Figure 8-11 DataStage Job Design: XML file is written from the XML Stage

8.1.2 Specifying the XML schema in the Document Root tab

The XML Stage is a schema driven stage. Without a valid schema, you cannot design the assembly. For the schema to be available in the assembly, the schema (xsd files) must be imported into the *Schema Library Manager*. When you import the schema into the Schema Library Manager, the schema is converted into a simplified model. This simplified model is then available to the steps of the assembly.

The bank must create XML files from the data in the database. To create XML files, the XML schema must be defined in the Document Root tab of the Composer Step, as shown in Figure 8-12. The schema defined in the Document Root tab describes the common structure of all output XML files to be created by the Composer Step.



Assembly has 1 issue(s).

CDIUI2805E Input is required for: Document Root.

Select the schema item that describes the documents that the step composes.

Document Root: *

Right-click an item above to display options.

Figure 8-12 The Document Root tab of the Composer Step

To define the schema in the Document Root tab, follow these steps:

1. Click **Browse**. The Select Document Root window opens. This window shows all the available libraries in the Schema Library Manager. Each library shows the elements of the schema that are imported into it.

The bank created the library `Account_Info` in the Schema Library Manager and imported the `xsd` file `Account.xsd` into it. When you click Browse in the Document Root tab of the Composer Step, you can see the library in the Select Document Root window, as shown in Figure 8-13 on page 211.

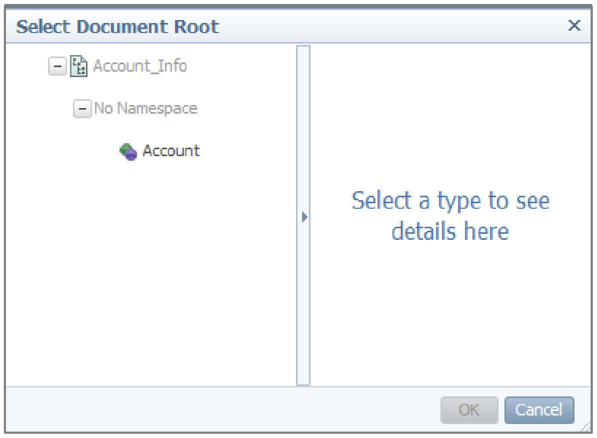


Figure 8-13 The Select Document Root window

2. In the Select Document Root window, select the top-level element of the XML file. The output XML files of the bank must have Account as their top-level element, as shown in Example 8-1. Therefore, you must select the item **Account** in the Select Document Root window.

   Example 8-1 Expected output XML file

```
?xml version="1.0" encoding="UTF-8"?>
<Account>
   <Customer>
      <Account_Number>C198765487</Account_Number>
      <Account_Holder_Name>John</Account_Holder_Name>
      <Account_Holder_DOB>1980-07-02</Account_Holder_DOB>
      <Account_Holder_address>San Jose,CA</Account_Holder_address>
      <Account_Holder_phoneNumber>123-456-654-4567</Account_Holder_p
      honeNumber>
      <Account_Type>Savings</Account_Type>
   </Customer>
</Account>
```

3. When you select the element Account in the Select Document Root window, you can see the structure of the element on the right half of the window, as shown in Figure 8-14 on page 212.

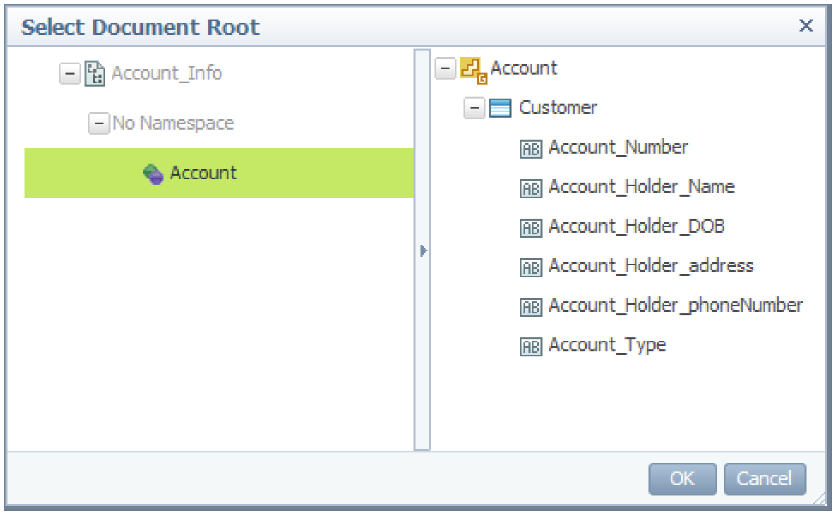
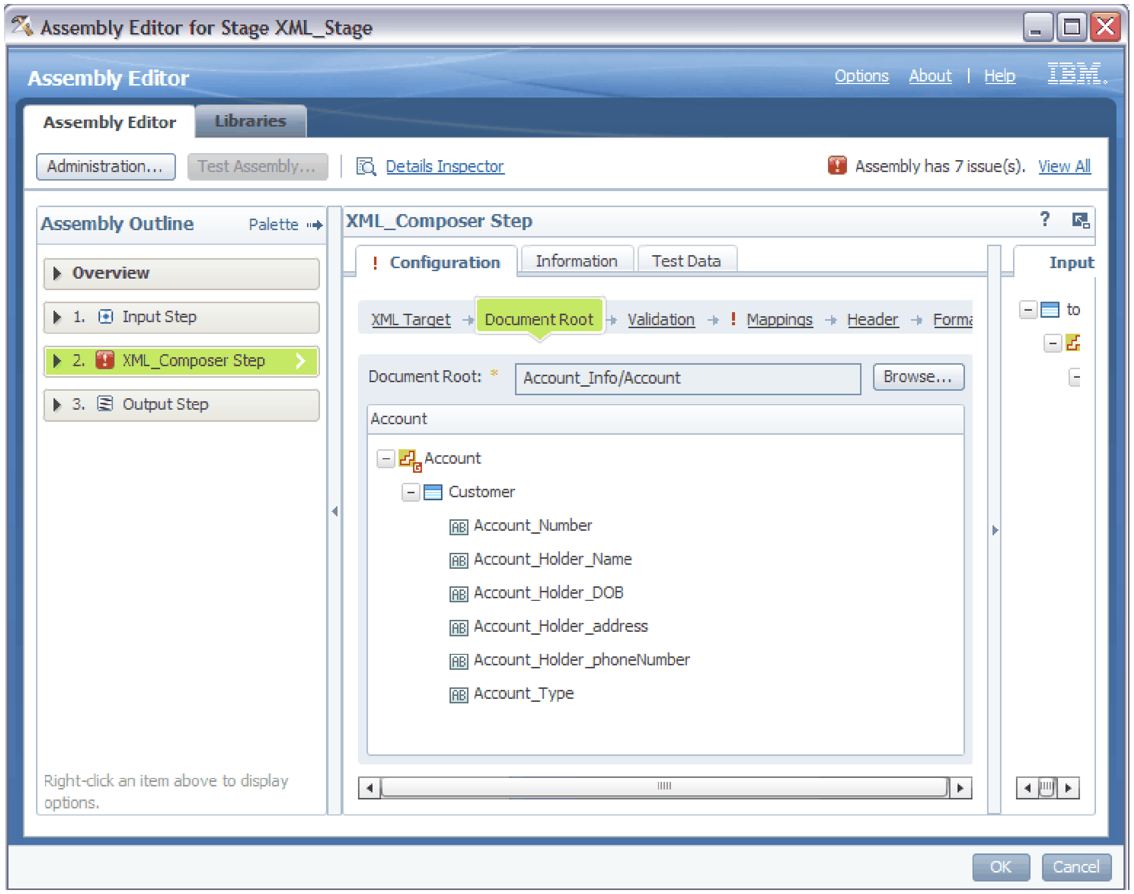


Figure 8-14 The schema structure in the Select Document Root window

4. Click **OK** to close the Select Document Root window. The schema structure is defined in the Document Root tab, as shown in Figure 8-15 on page 213.



Document Root: * Account_Info/Account

```
Account
⊟ Account
   ⊟ Customer
        Account_Number
        Account_Holder_Name
        Account_Holder_DOB
        Account_Holder_address
        Account_Holder_phoneNumber
        Account_Type
```

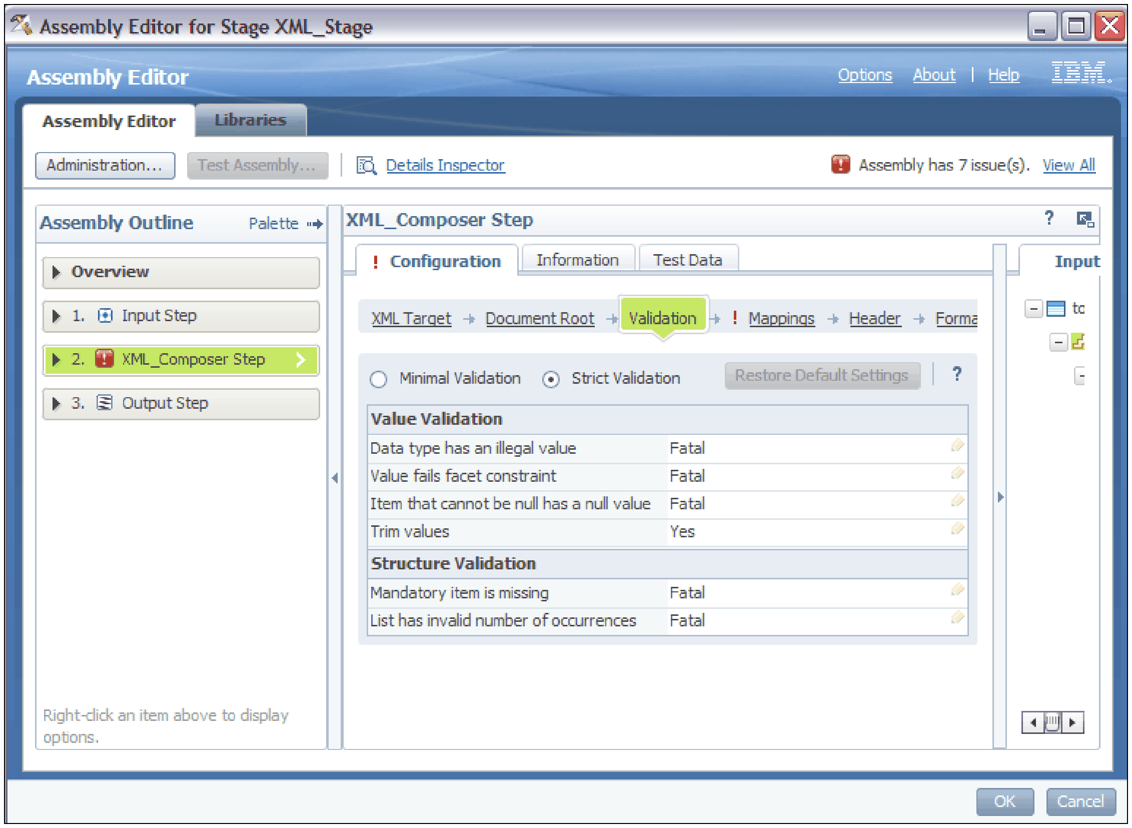
Figure 8-15 The Document Root tab in the Composer Step

8.1.3 Creating a valid XML file

The application that creates the report for the bank requires data in an XML file. The XML file needs to be valid, or the application cannot create the report. Therefore, the bank wants to validate the XML file before the file is input to the application that creates the reports.

The XML Stage offers unique validation capabilities for you to control the amount of schema validation when composing XML documents. When the XML Composer step produces the XML document, it always creates a well-formed document. It also validates the incoming data from the previous steps against the document root element structure. The spectrum of validation that ranges from minimal validation to strict schema validation allows control of the trade-off between performance and validation and allows the degree of complexity that you want.

The degree of validation can be selected through the Validation tab in the XML Composer Step, as shown in Figure 8-16. The Validation tab presents a set of validation rules. Each rule consists of a *condition* and an *action*. The condition is the validation check, and when the check fails, the action is performed.



Minimal Validation [✓ Strict Validation]

Value Validation	
Data type has an illegal value	Fatal
Value fails facet constraint	Fatal
Item that cannot be null has a null value	Fatal
Trim values	Yes
Structure Validation	
Mandatory item is missing	Fatal
List has invalid number of occurrences	Fatal

Figure 8-16 The Validation tab in the Composer Step

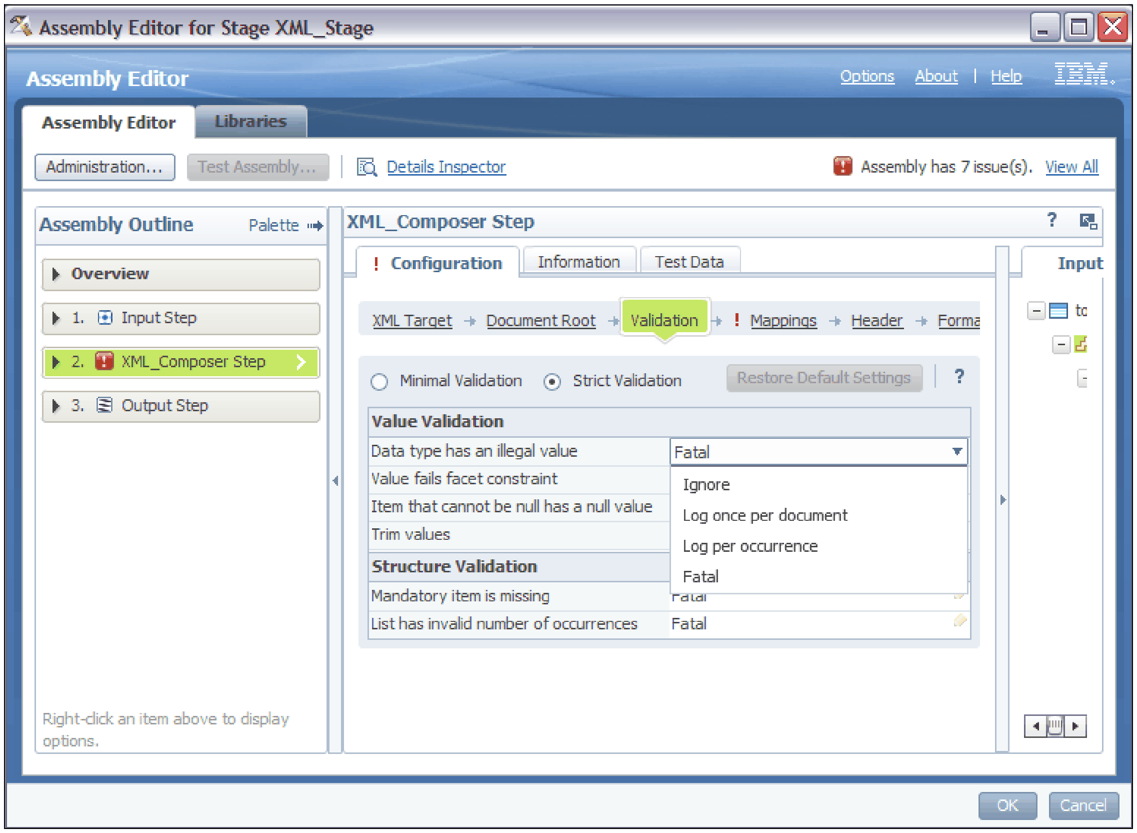
The *Validation Rules* are categorized into two categories, as shown in Figure 8-16: *Value Validation* and *Structure Validation*. Value validation rules are rules that check the actual data values in the XML document against the defined schema types. Structure Validation rules are rules that check whether the XML document conforms to the schema that was specified in the Document Root tab.

Not all rules are available for XML Composer step. Certain rules are only applicable for the XML Parser step, because the XML Composer step always adheres to these rules.

By default, *Strict* validation is selected as the validation mode in the Composer step. In Strict validation, all the validation rule actions are set to `Fatal` to ensure that the job stops on the first occurrence of invalid data.

Minimal validation mode sets all the validation rule actions to `Ignore`. When Minimal validation is selected in the Composer step, no validation occurs. The Composer step, by default, always produces a well-formed XML document. Even when Minimal Validation is selected, the Composer step creates a well-formed XML document.

Each of the validation rules can be configured to perform a specific action. When you click the action for a particular rule, the drop-down list shows the available actions that can be selected for that rule, as shown in Figure 8-17.



( ) Minimal Validation [✓ Strict Validation]

Value Validation	
Data type has an illegal value	Fatal
Value fails facet constraint	
Item that cannot be null has a null value	
Trim values	
Structure Validation	
Mandatory item is missing	Fatal
List has invalid number of occurrences	Fatal

Ignore
Log once per document
Log per occurrence
Fatal

Figure 8-17 The actions that are available for the validation rules

The bank uses the data from the database while creating the XML files. Because the data in the database is stored in varchar columns, some of the data can be invalid. You might see the following issues in the incoming data:

- Incorrect format for the date of birth (DOB) field. The correct date format in XML is *yyyy-mm-dd*.
- The Account Type field might not be the correct type.

The bank wants to take the following action for each of these errors:

- Log an error when the date formats are incorrect.
- Log an error when the Account Type field is an incorrect value.

To log error messages, the actions *Log per Occurrence* and *Log once per Document* are available. These messages are logged in the DataStage Director Log. Log per Occurrence logs the message for data that fails the validation rule. Log once per Document logs the first error message only when the validation rule fails. For example, in the XML document shown in Example 8-2, the Account Type fields contain invalid data. Log per Occurrence logs messages for each of the account types, and therefore, two error messages show in the Director log. Log once per Document logs the message only for the first invalid account type, for example, `Private`.

Example 8-2 Output XML file created by the Composer step

```
<?xml version="1.0" encoding="UTF-8"?>
<Account>
   <Customer>
      <Account_Number>C198765487</Account_Number>
      <Account_Holder_Name>John Doe1</Account_Holder_Name>
      <Account_Holder_DOB>02-07-1980</Account_Holder_DOB>
      <Account_Holder_address>San Jose,CA</Account_Holder_address>
      <Account_Holder_phoneNumber>123-456-654-4567</Account_Holder_phon
      eNumber>
      <Account_Type>Private</Account_Type>
   </Customer>
   <Customer>
      <Account_Number>C198765444</Account_Number>
      <Account_Holder_Name>Jane Smith</Account_Holder_Name>
      <Account_Holder_DOB>1975-01-01</Account_Holder_DOB>
      <Account_Holder_address>San Jose,CA</Account_Holder_address>
      <Account_Holder_phoneNumber>123-789-987-2457</Account_Holder_phon
      eNumber>
      <Account_Type>non-savings</Account_Type>
   </Customer>
</Account>
```

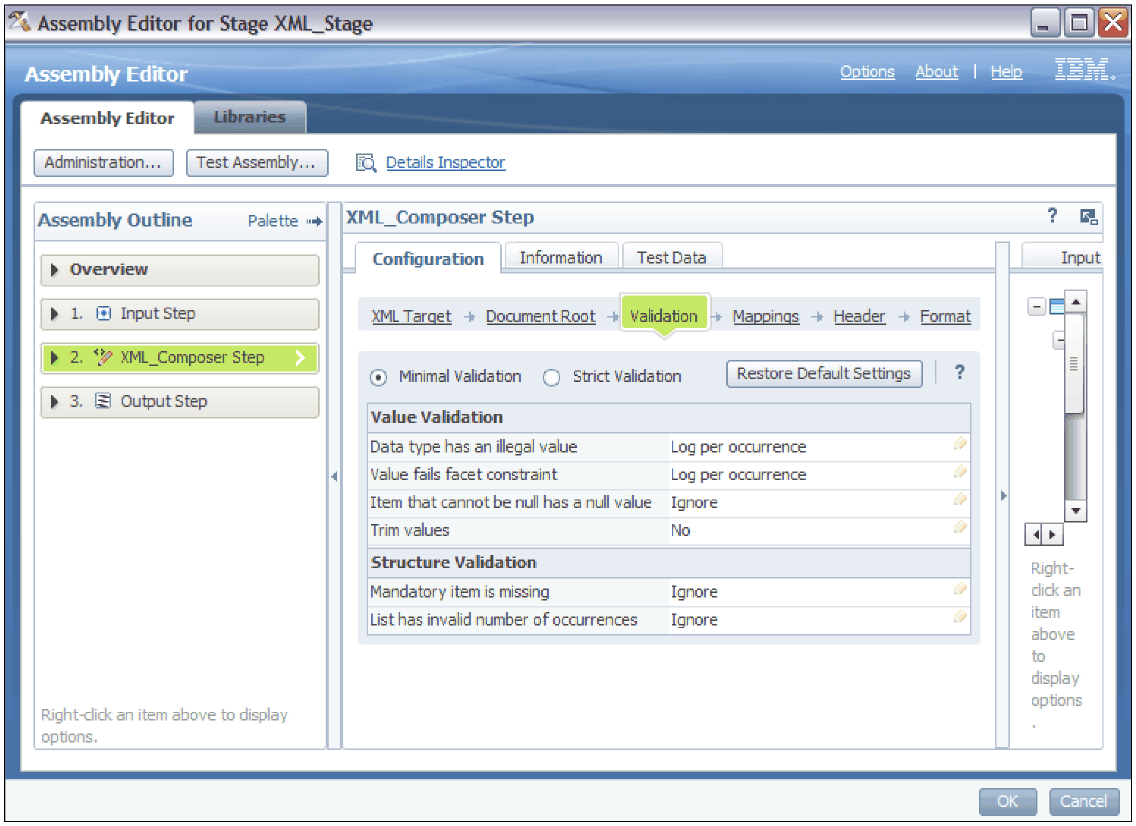
The validation rule *Data type has an illegal value* checks whether the value specified for each of the elements/attributes in the XML file is valid and within the range for the specified data type. This rule checks the format of the incoming data for the date of birth field. Set the action for this rule to Log per Occurrence to log an error message for every invalid date format.

The valid Account Type values are specified in the XML schema. The data that comes from the database must match the value in the XML schema. If the data does not match, an error message needs to be logged. To log an error message for every invalid Account Type, the validation rule *Value fails facet constraint* is set to Log per Occurrence.

For the XML document that is shown in Example 8-2 on page 216, if these validation rules are applied, the following results occur while parsing:

- An error message is logged for the date of birth field, because the value `02-07-1980` is not in the correct format.
- Two error messages are logged for the Account Type field for the values `Private` and `non-savings`.

You can see the final configuration of the Validation tab in Figure 8-18 on page 218.



[✓ Minimal Validation] Strict Validation

Value Validation	
Data type has an illegal value	Log per occurrence
Value fails facet constraint	Log per occurrence
Item that cannot be null has a null value	Ignore
Trim values	No
Structure Validation	
Mandatory item is missing	Ignore
List has invalid number of occurrences	Ignore

Figure 8-18 Configuration of the Validation tab in the XML Composer Step

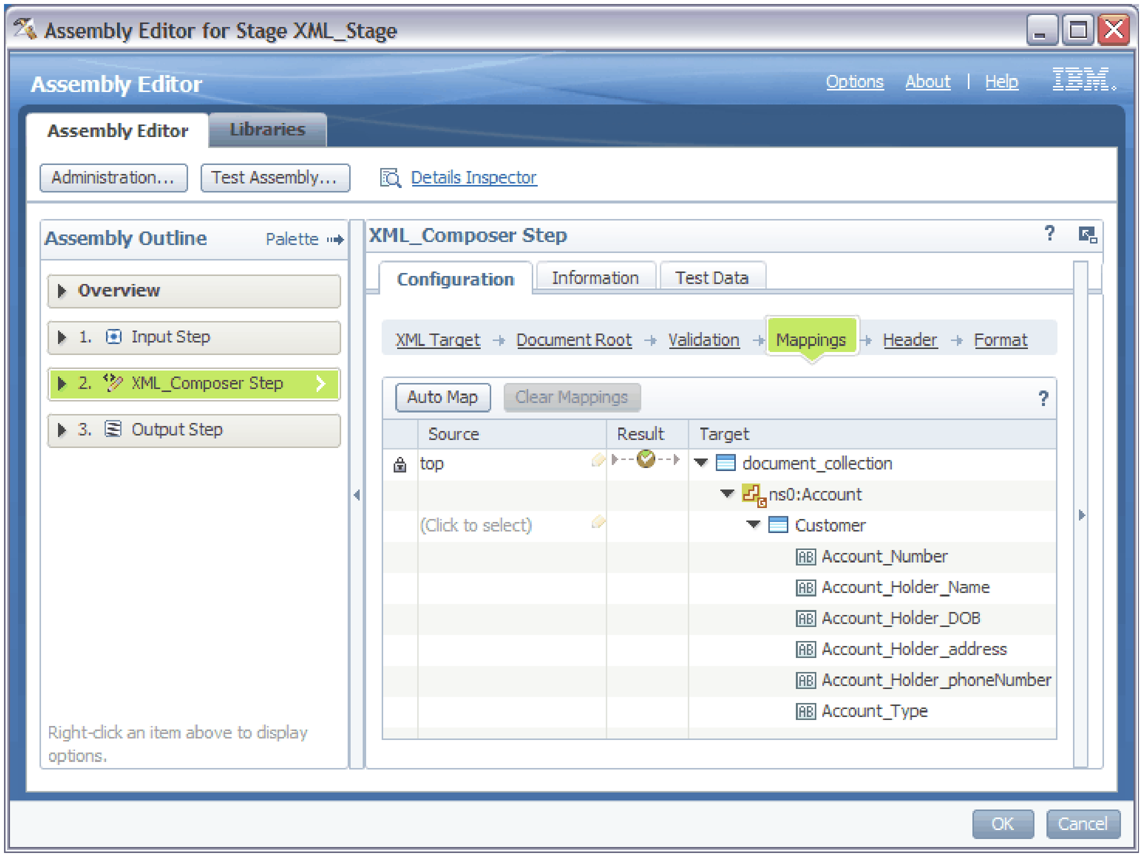
Treating all XML data types as string

The validation rule *Data type has an illegal value* checks whether the value specified for each of the elements/attributes in the XML file is valid and within the range for the specified data type. When the action for this rule is set to Ignore, Log per Occurrence, or Log once per Document, then all the elements are converted to string type. Therefore, the actual XML schema data types are not validated in the Composer step and all elements are treated as a string. For example, if an element is of integer type, and Data type has an illegal value is set to Ignore, the element is treated as a string type.

8.1.4 The Mappings tab

To map the output of the Input step to the Composer step, the Mappings tab is used. The Mappings table is a common construct to map one hierarchical structure to another. The Mappings table is used as the last step in adapting the input to the output structure. You cannot perform a join, union, or any other set operation and map to a target list. If a set operation is needed, it must be performed in a Transformation step before the Composer step.

The Mappings table, which is shown in Figure 8-19, contains three columns: Source, Result, and Target. The Target column displays the items of the target structure. The target structure is the XML schema that is defined in the Document Root tab of the Composer Step. The Source column displays the columns that need to be mapped to the target items. The Result column displays the status of the current mapping.



Source	Result	Target
top	✓	document_collection
		ns0:Account
(Click to select)		Customer
		Account_Number
		Account_Holder_Name
		Account_Holder_DOB
		Account_Holder_address
		Account_Holder_phoneNumber
		Account_Type

Figure 8-19 The Mappings tab in the Composer Step

The document_collection list is the first target list in the Mappings table. Based on the mapping to this list, one or multiple XML documents are created (see 8.3, "Multiple XML documents: Single Composer step" on page 253).

To create only one output XML document, the `top` list in the Source column needs to be mapped to document_collection in the Target column. The entire output schema structure is displayed under the document_collection list, as shown in Figure 8-20. This structure is similar to the structure that is defined in the Document Root tab of the Composer Step.

In order to map the items from the Input step to the Composer step, follow these mapping rules:

- A list in the Target column can be mapped to a list in the source column only. For example, in Figure 8-20, only a source list can be mapped to Customer. All lists are represented with blue icons. *An item with a blue icon must be mapped to another item with a blue icon.*
- A group is never mapped. *Groups create structures and do not need to be mapped*. As shown in Figure 8-20, the group Account does not need to be mapped.
- *To map a source item to a target column, first the parent list of the item needs to be mapped*. Therefore, in Figure 8-20, to map a source item to Account_Number, first the Customer list needs to be mapped.

Following these rules, the output of the Input step must be mapped to the target columns in the Composer step, as shown in Figure 8-20.

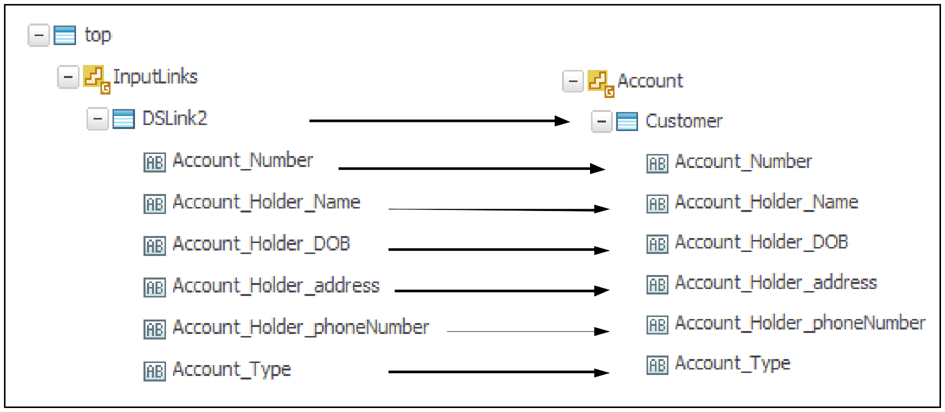
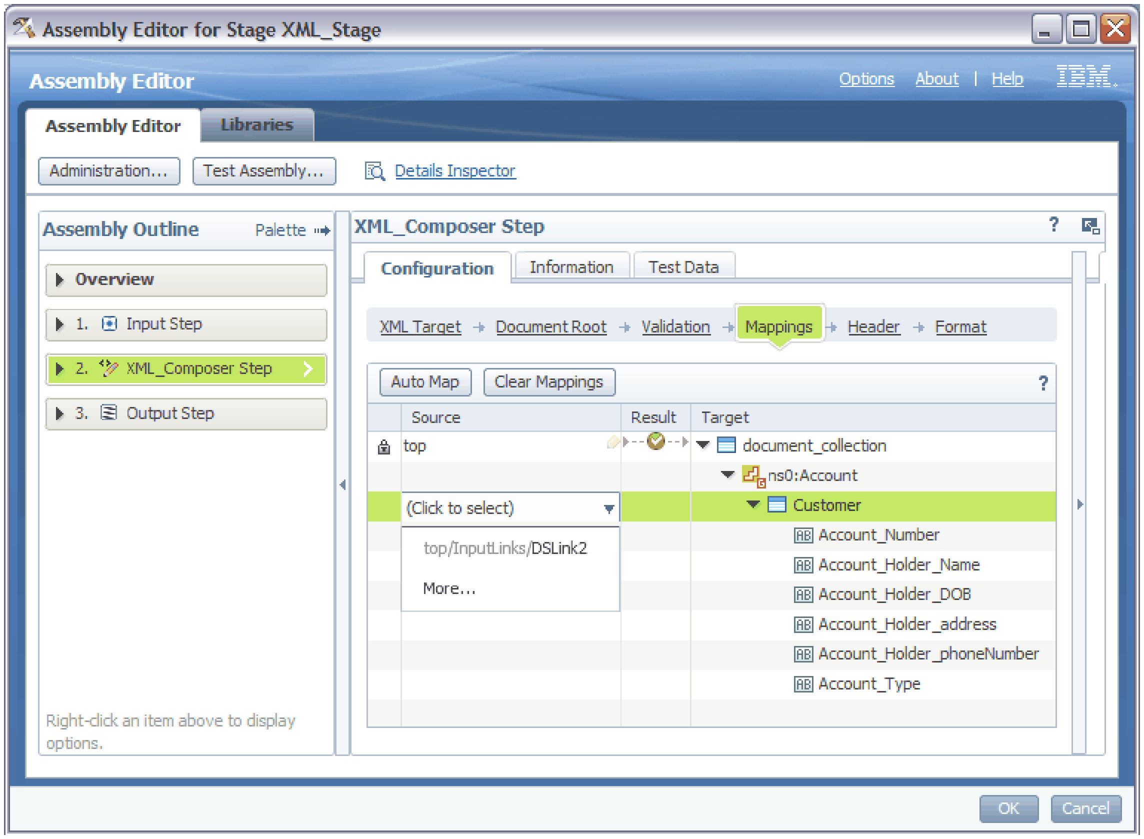


Figure 8-20 Mapping the Input Step output and the Composer Step target structure

To complete the Mappings table, follow these steps:

1. Double-click the Source column for the target list Customer to see a drop-down list. This drop-down list is called a *suggestion list*. The suggestion list shows all the available source lists that can be mapped to the target list. Only one source list, DSLink2, exists. Therefore, it is the only available source

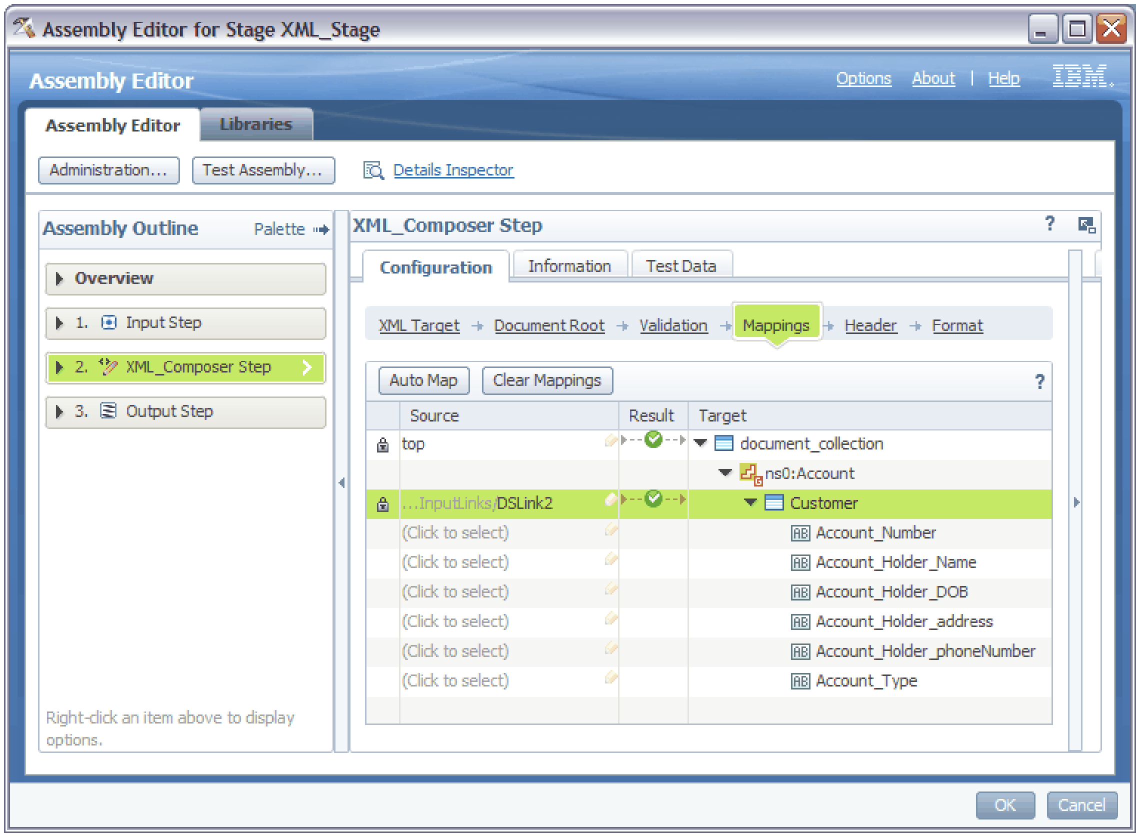
list in the suggestion list that is shown in Figure 8-21. The absolute path shows in the suggestion list, that is, `top/InputLinks/DSLink2`.



Source	Result	Target
top	✓	document_collection
		ns0:Account
(Click to select)		Customer
top/InputLinks/DSLink2		Account_Number
More...		Account_Holder_Name
		Account_Holder_DOB
		Account_Holder_address
		Account_Holder_phoneNumber
		Account_Type

Figure 8-21 The suggestion list for the target list Customer

2. Select the source list **DSLink2** from the suggestion list. The mapping occurs and a white check mark in a green circle shows in the Result column, as shown in Figure 8-22 on page 222.

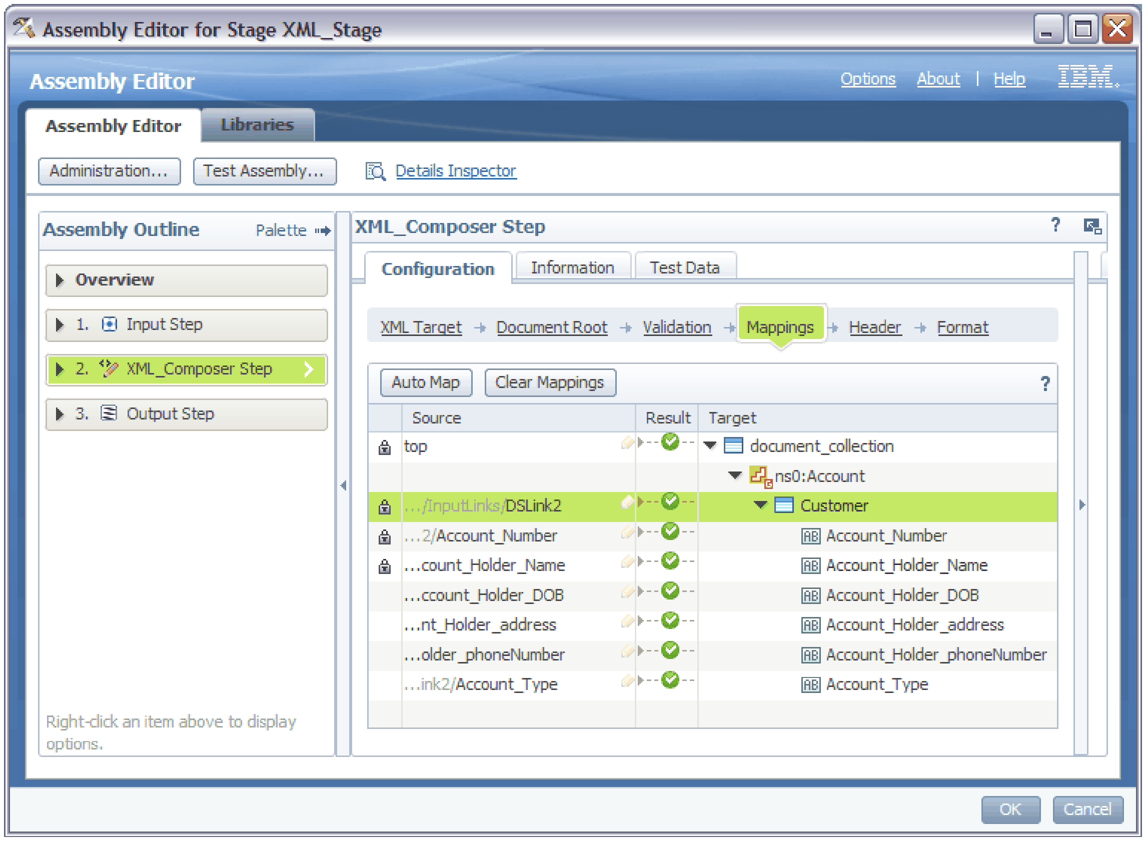


Source	Result	Target
top	✓	document_collection
		ns0:Account
...InputLinks/DSLink2	✓	Customer
(Click to select)		Account_Number
(Click to select)		Account_Holder_Name
(Click to select)		Account_Holder_DOB
(Click to select)		Account_Holder_address
(Click to select)		Account_Holder_phoneNumber
(Click to select)		Account_Type

Figure 8-22 Mappings table shows the mapping between the source and target lists

3. After you map the Customer target list, you can map the columns under it. To map each of the target columns, select the corresponding row in the Source column. From the suggestion list, select the appropriate item to map. The item recommendations in the suggestion list are based on the name and data type of the Target column.

After mapping all the columns, the Mappings table is complete, as shown in Figure 8-23 on page 223.



Source	Result	Target
top	✓	document_collection
		ns0:Account
.../InputLinks/DSLink2	✓	Customer
...2/Account_Number	✓	Account_Number
...count_Holder_Name	✓	Account_Holder_Name
...ccount_Holder_DOB	✓	Account_Holder_DOB
...nt_Holder_address	✓	Account_Holder_address
...older_phoneNumber	✓	Account_Holder_phoneNumber
...ink2/Account_Type	✓	Account_Type

Figure 8-23 The complete Mappings table

Automatically mapping the columns

One bank schema contains over a hundred columns. It is tedious to map each column in the target to the corresponding column in the source. The Composer step provides an easy mechanism to map multiple columns simultaneously by using the Auto Map function. Follow these steps to use the Auto Map feature:

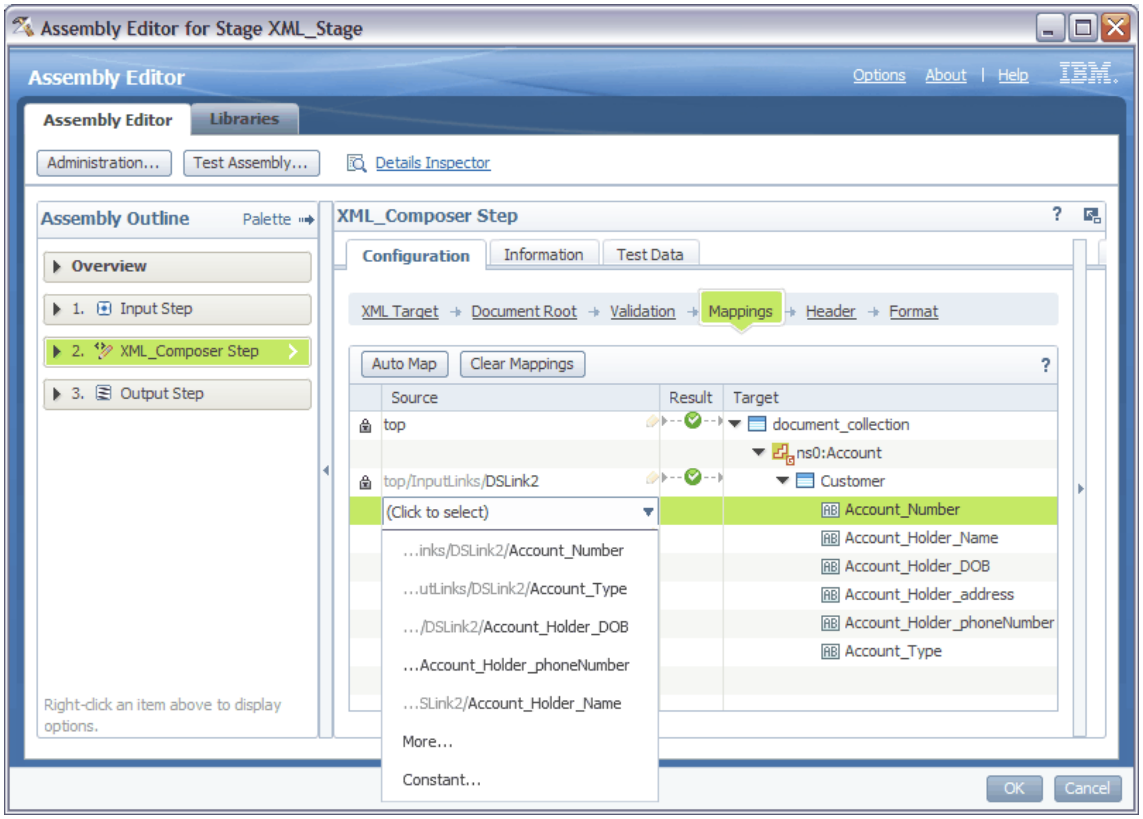
1. Map the source list **DSLink2** to the target list **Customer**.
2. Select the row corresponding to the target list **Customer**.
3. Click **Auto Map**. All the columns under the list Customer are automatically mapped.

The mapping is based on the similarity between the name and data type of the source and target columns. The automatic mapping suggestion might not always be correct. The Auto Map feature also cannot map columns of different data types. Therefore, if the target column is an integer and all source columns are of type string, then no mapping occurs for the integer column by using the Auto Map feature.

Selecting mapping candidates by using the More option

When the output of the Input step contains many columns, the suggestion list cannot list all these columns. By default, the suggestion list shows only five source items. If the required column is not available in the suggestion list, you can use the *More* option. The More option shows the entire source tree structure from which the source column to map can be selected. To use the More option, follow these steps:

1. Select the row in the Source column that you need to map to the Target column.
2. The More option is available near the end of the suggestion list, as shown in Figure 8-24. Select **More**.



Source	Result	Target
top	✓	document_collection
		ns0:Account
top/InputLinks/DSLink2	✓	Customer
(Click to select)		Account_Number
		Account_Holder_Name
		Account_Holder_DOB
		Account_Holder_address
		Account_Holder_phoneNumber
		Account_Type

```
...inks/DSLink2/Account_Number
...utLinks/DSLink2/Account_Type
.../DSLink2/Account_Holder_DOB
...Account_Holder_phoneNumber
...SLink2/Account_Holder_Name
More...
Constant...
```

Figure 8-24 The suggestion list that shows the More option

3. The Target window opens, as shown in Figure 8-25. In this window, you can see the entire schema tree.

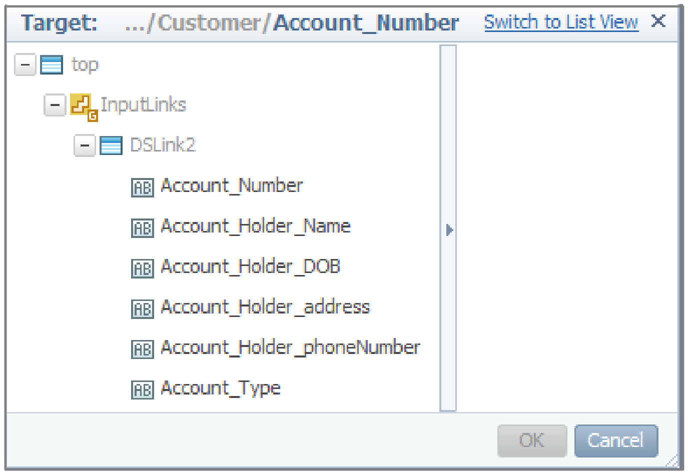


Figure 8-25 The Target window

4. Select the item to map and click **OK**.

The mapping is complete.

Mapping constants to a target column

The bank needs to create an XML file. In this file, the Account_Type needs to always be set to Savings, no matter what type of data comes from the database columns. Therefore, a constant value needs to be assigned to the output column Account_Type. Use the Composer Step Constant option. To map the target item Account_Type to the constant value Savings, follow these steps:

1. Ensure that the parent list of the target item is mapped. Therefore, to assign a constant value to the item Account_Type, Customer needs to be mapped. Map **DSLink2** to the list **Customer**.
2. Select the row in the source corresponding to the target item Account_Type. The suggestion list is shown in Figure 8-26 on page 226.

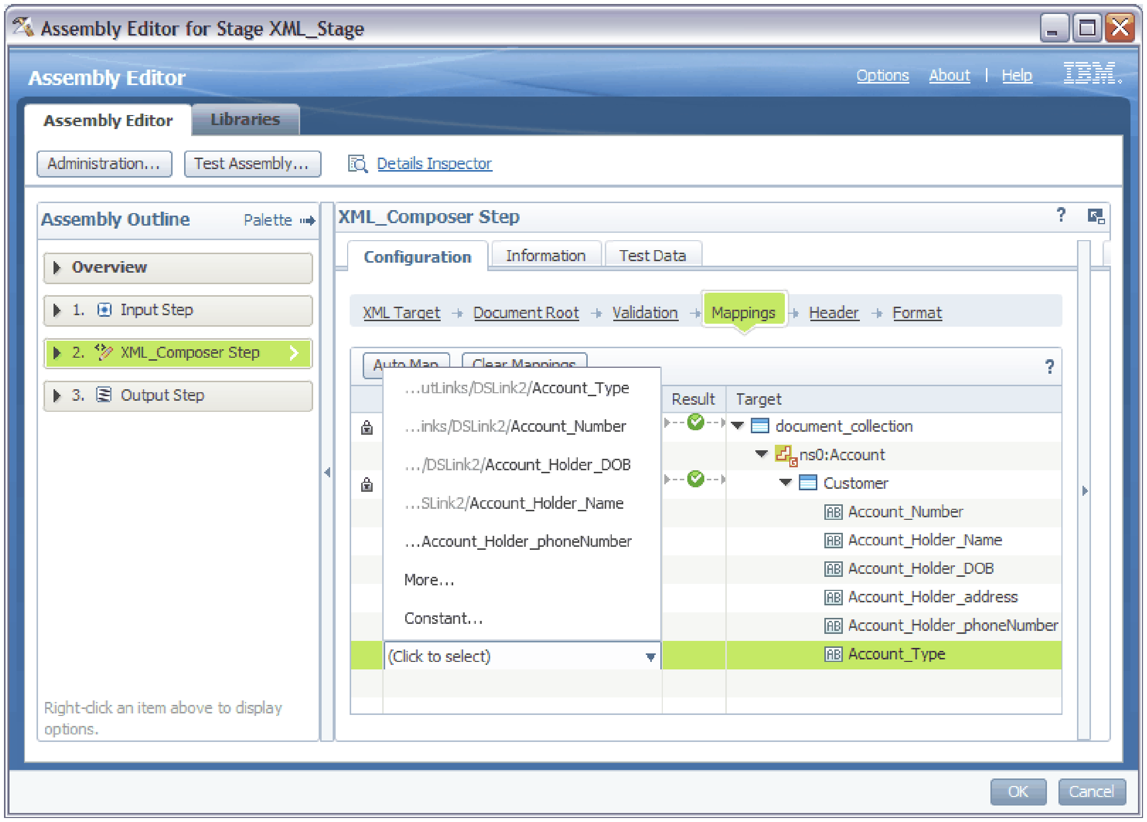


Figure 8-26 The suggestion list with the Constant option

3. The last option in the suggestion list is Constant. Select the **Constant** option to open the Constant window. In the Constant window shown in Figure 8-27, you can enter the value to assign to the target item. To assign the value Savings, enter `Savings` in the Constant window and click **OK**.

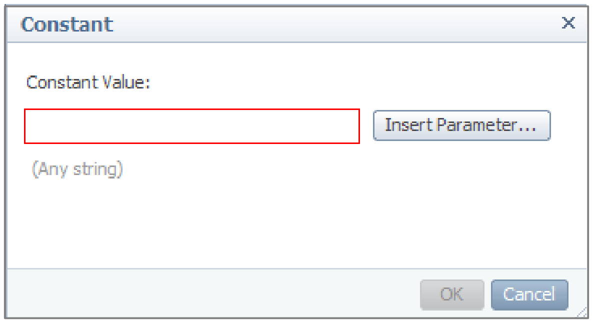
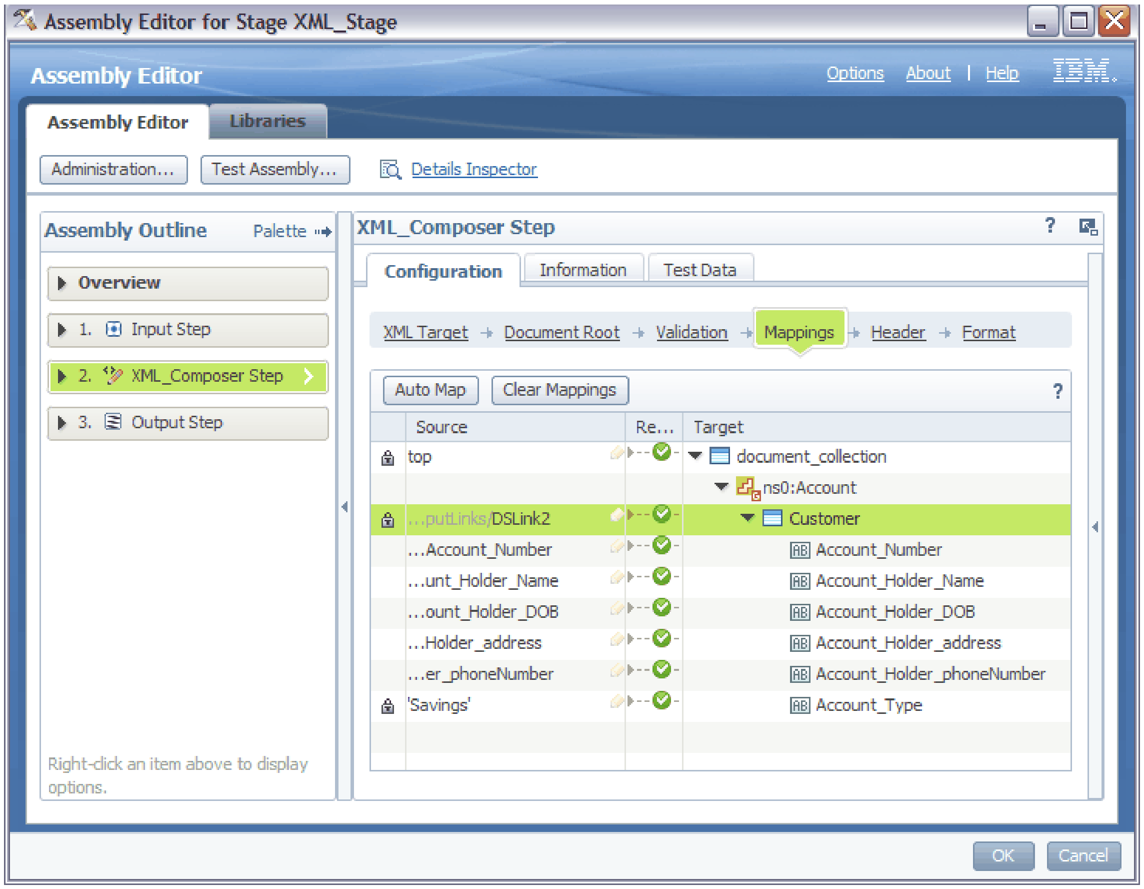


Figure 8-27 The Constant Window

Figure 8-28 shows a constant value assigned to the target column Account_Type.

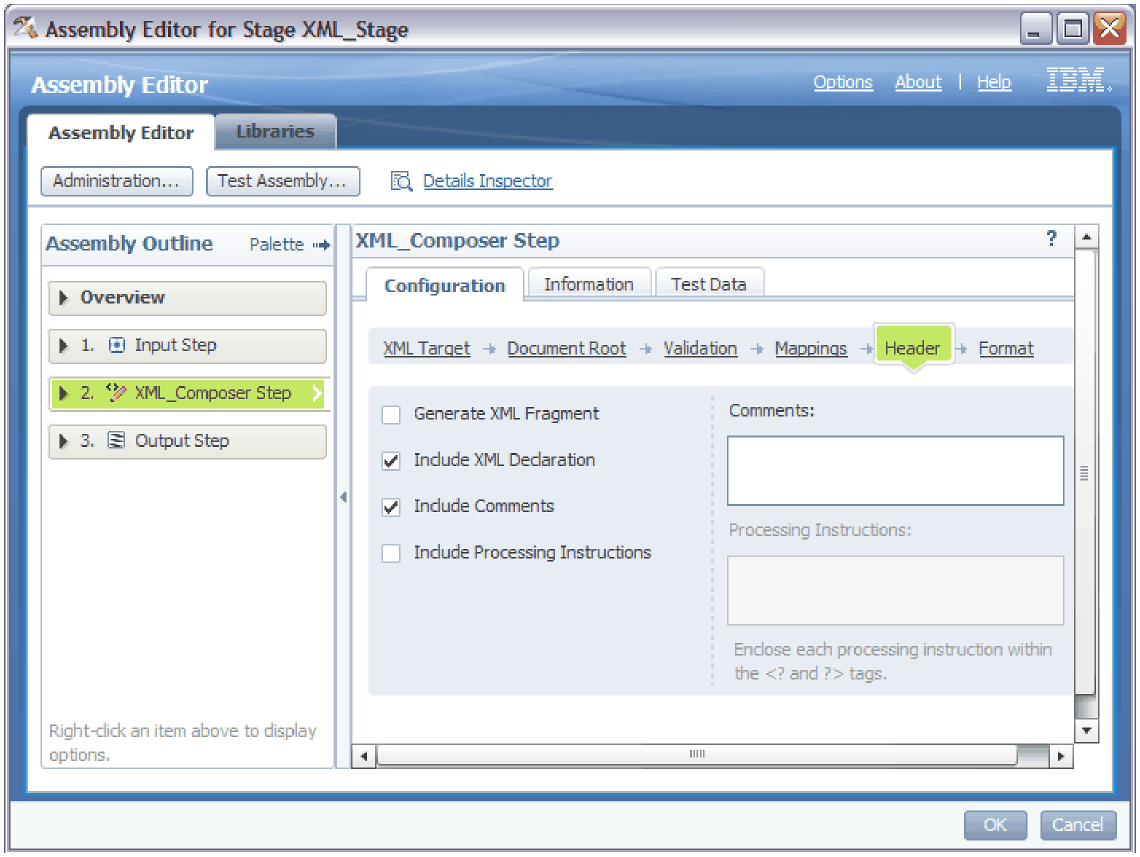


Source	Re...	Target
top	✓	document_collection
		ns0:Account
...putLinks/DSLink2	✓	Customer
...Account_Number	✓	Account_Number
...unt_Holder_Name	✓	Account_Holder_Name
...ount_Holder_DOB	✓	Account_Holder_DOB
...Holder_address	✓	Account_Holder_address
...er_phoneNumber	✓	Account_Holder_phoneNumber
'Savings'	✓	Account_Type

Figure 8-28 The Constant value Savings assigned to the target item Account_Type

8.1.5 The Header tab

The bank requires comments on the XML document. These comments are used by the application that creates the customer reports. Use the Header tab in the Composer Step to add comments to the XML document that is created by the Composer Step. Select the **Include Comments** option to add comments. The text box for Comments becomes active. You can enter your comment in the text box, as shown in Figure 8-29 on page 228.



- [ ] Generate XML Fragment
- [x] Include XML Declaration
- [x] Include Comments
- [ ] Include Processing Instructions

Comments:

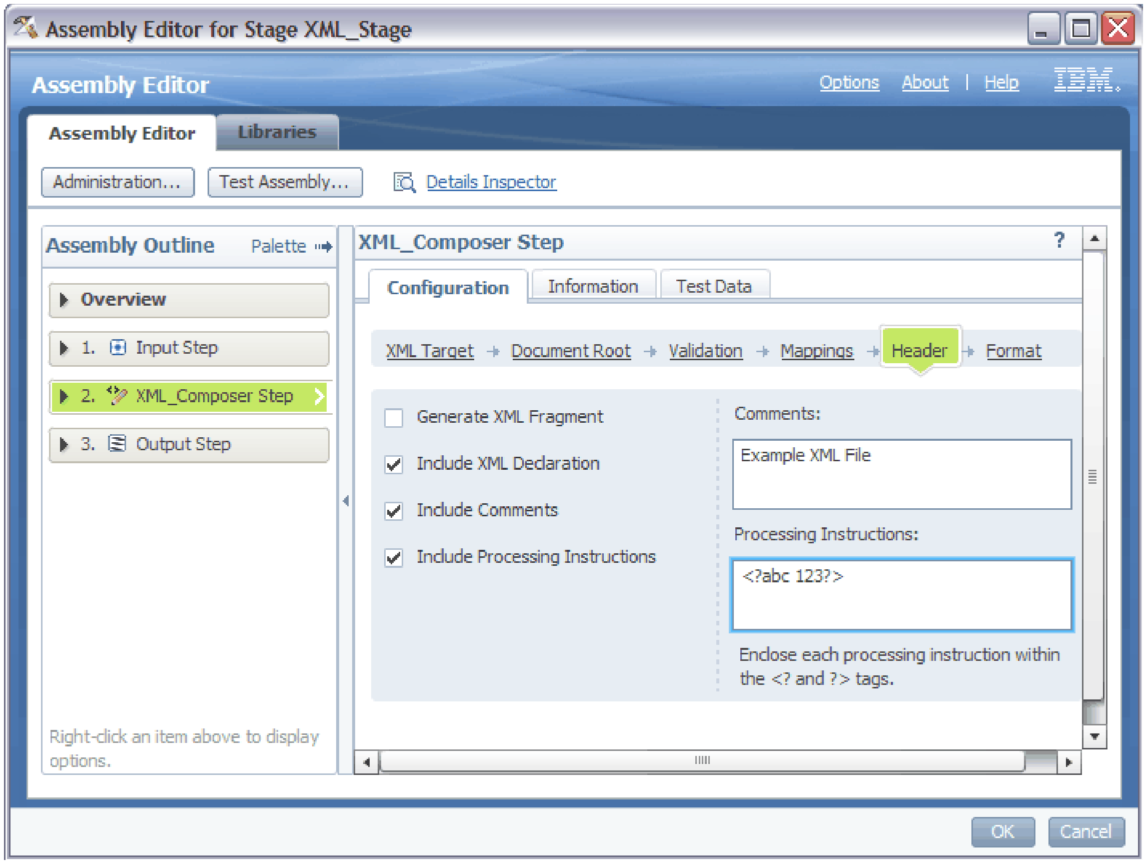
Processing Instructions:

Enclose each processing instruction within the <? and ?> tags.

Figure 8-29 The Header tab in the Composer Step

You do not need to enter the comments in the format `<!-- -->`. The tags are automatically added by the XML Composer step while you add comments.

You can also add Processing Instructions to the output XML file by choosing the Include Processing Instructions option. The Processing Instruction must be added between the tags <? ?>, as shown in Figure 8-30 on page 229.



- [ ] Generate XML Fragment
- [✓] Include XML Declaration
- [✓] Include Comments
- [✓] Include Processing Instructions

Comments:

Example XML File

Processing Instructions:

<?abc 123?>

Enclose each processing instruction within the <? and ?> tags.

Figure 8-30 The Header tab: Comments and processing instructions are added

The Include XML Declaration is selected by default in the Header tab. Use this option to add the XML declaration to the output XML file:

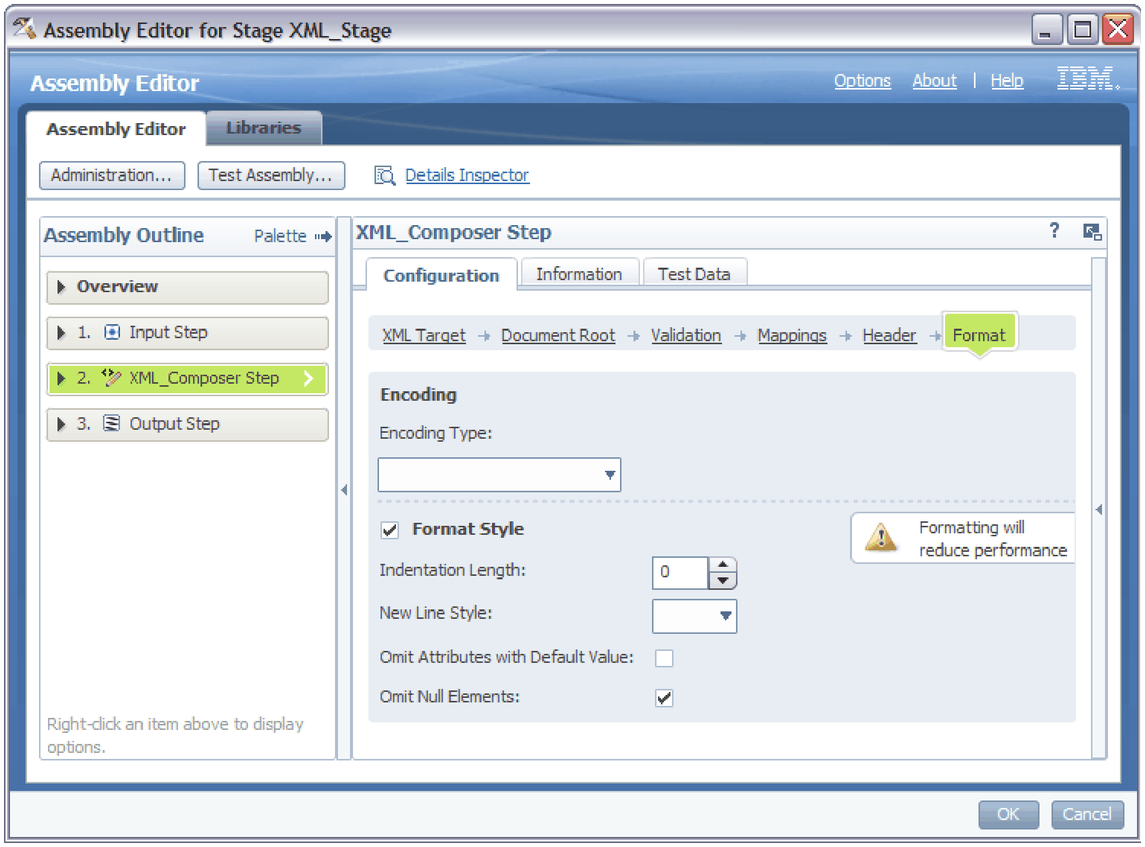
```
<?xml version="1.0" encoding="UTF-8">
```

When Include XML Declaration is not selected, the output XML file does not contain the XML declaration.

Use the Generate XML Fragment option when you do not want to include the XML declaration, comments, and processing instructions in the output XML document.

8.1.6 The Format tab

The bank requires a formatted output XML file. To add formatting, use the Format tab. In the Format tab, select **Format Style**. The formatting options are enabled, as shown in Figure 8-31.



Encoding

Encoding Type:

☑ Format Style

Formatting will reduce performance

Indentation Length: 0

New Line Style:

Omit Attributes with Default Value: ☐

Omit Null Elements: ☑

Figure 8-31 The Format tab of the Composer Step

The Indentation Length helps specify the degree of indentation that is required. The New Line Style helps specify the new line character to use within the file. Formatting affects performance.

8.1.7 The output of the Composer step

To configure the Composer step, you need to configure the XML Target, Document Root, Mappings, Header, and Format tabs. The output of the Composer Step is seen in the Output tab. The output of the Composer Step is in the group XML_Composer:result. The output depends on the option that is selected in the XML Target tab.

When the *Write to File* option is selected in the XML Target tab, the output XML file is created by the Composer step. Therefore, a single item called output-filename shows under the group XML_Composer:result, as shown in Figure 8-32. This item contains the name of the output XML file in the Filename Prefix text box in the XML Target tab.

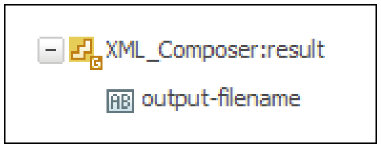


Figure 8-32 Composer step output for the Write to File option

When the *Pass as String* or *Pass as Large Object* option is selected, the XML Composer step does not write the file directly to the server. The XML file is passed to a stage after XML Stage. Therefore, the item result-string is under the group XML_Composer:result, as shown in Figure 8-33. This single item holds the entire XML file.

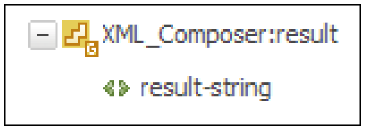


Figure 8-33 Composer step output for the Pass as String or Pass as Large Object option

8.2 Creating the hierarchy

The XML file that we created by using the Composer step in the previous sections contains a list called Customer. Therefore, you can directly map the output of the Input step to the Composer step. But if the XML schema contains multiple lists, the output of the Input step cannot directly be mapped to the structure in the Composer step. Figure 8-34 on page 232 shows an XML schema structure that contains multiple lists. To map the address and phoneNumber elements, the parent list address_info needs to be mapped first. You cannot map the input list DSLink2 to address_info, because it is already mapped to the list Customer.

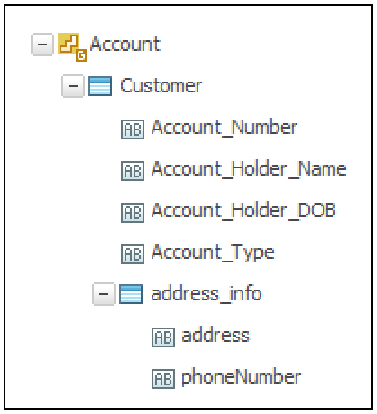


Figure 8-34 XML schema

To create an XML file based on the structure shown in Figure 8-34, the same hierarchy must be created so that the mapping is possible. The hierarchy can be created by using the Regroup step or the HJoin step in the assembly. We explain these steps in 8.2.1, "The Regroup step" on page 232, and 8.2.2, "The HJoin step" on page 240.

8.2.1 The Regroup step

The *Regroup step* is required to create a hierarchical structure. It converts a single list to two lists with a parent-child relationship. During this process, the Regroup step also removes any redundancy in the incoming data. For example, consider the input that comes from the database that contains the department code and the employee information that are shown in Table 8-1. By using the Regroup step a hierarchy can be created in which the department code is set as the parent and the employee information is set as the child. The Regroup step also removes the redundancy in the Department ID column by placing the employees `Jane Smith` and `John Doe1` under the same department.

Table 8-1 Removing redundant data elements

Department ID	Employee name	Employee ID
A100	Jane Smith	M098
A100	John Doe1	M765
B200	John Doe2	U789

The bank has a schema that contains multiple lists, as shown in Figure 8-34. To compose an XML file with this structure, use the Composer step. But, to be able to map the data from the Input step to the Composer step, you need to create a hierarchy. You can use the Regroup step to create the hierarchy.

To configure the Regroup step, it needs to be added to the assembly. To add the Regroup step, use the following steps:

1. Open **Assembly Editor** and click **Palette** in the Assembly Outline bar. You can see the various steps in the Palette, as shown in Figure 8-35.

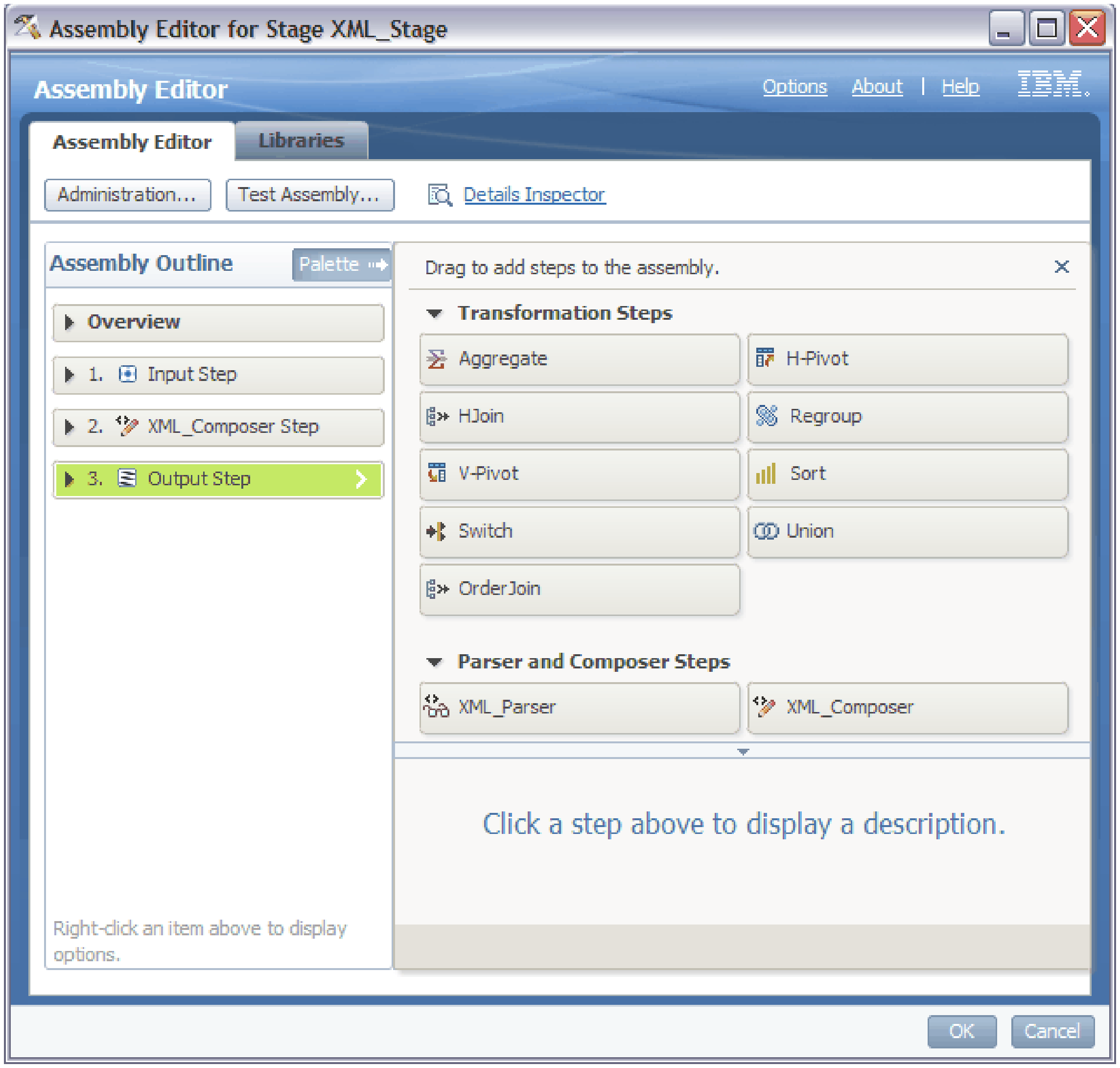
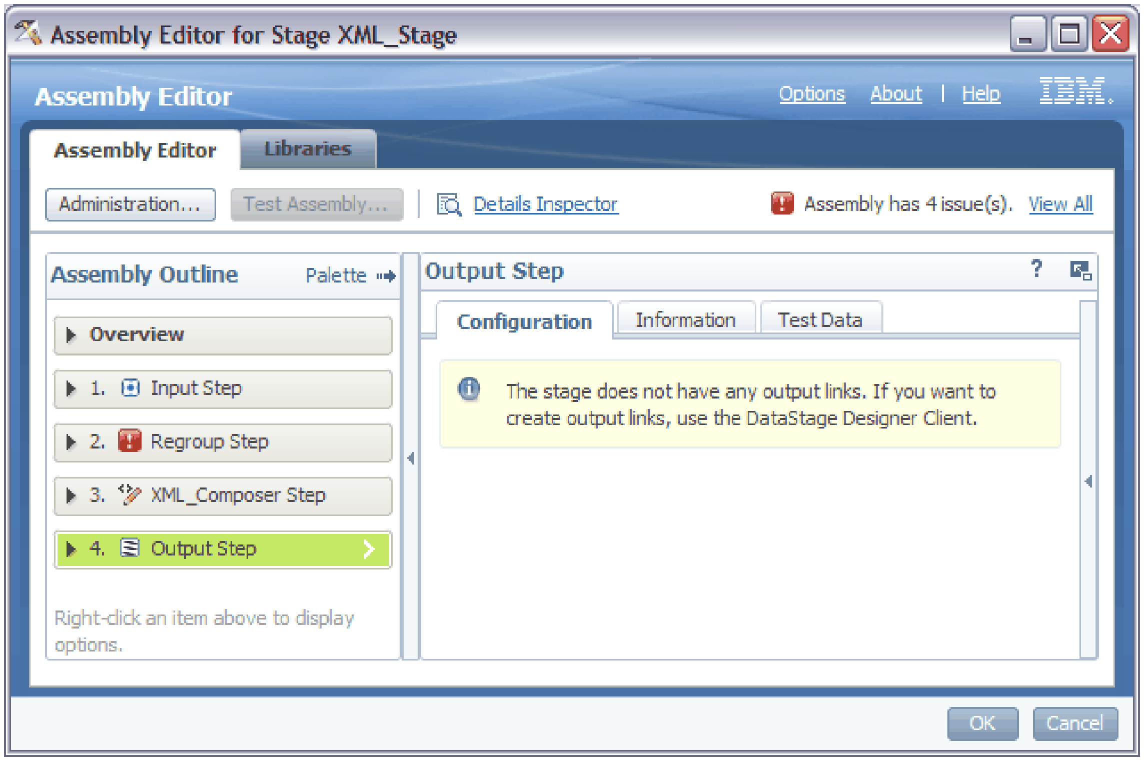


Figure 8-35 The Palette

2. Drag the **Regroup Step** from the Palette onto the Assembly Outline bar after the Input Step. The Regroup Step is added between the Input Step and the Composer Step, as shown in Figure 8-36.



The stage does not have any output links. If you want to create output links, use the DataStage Designer Client.

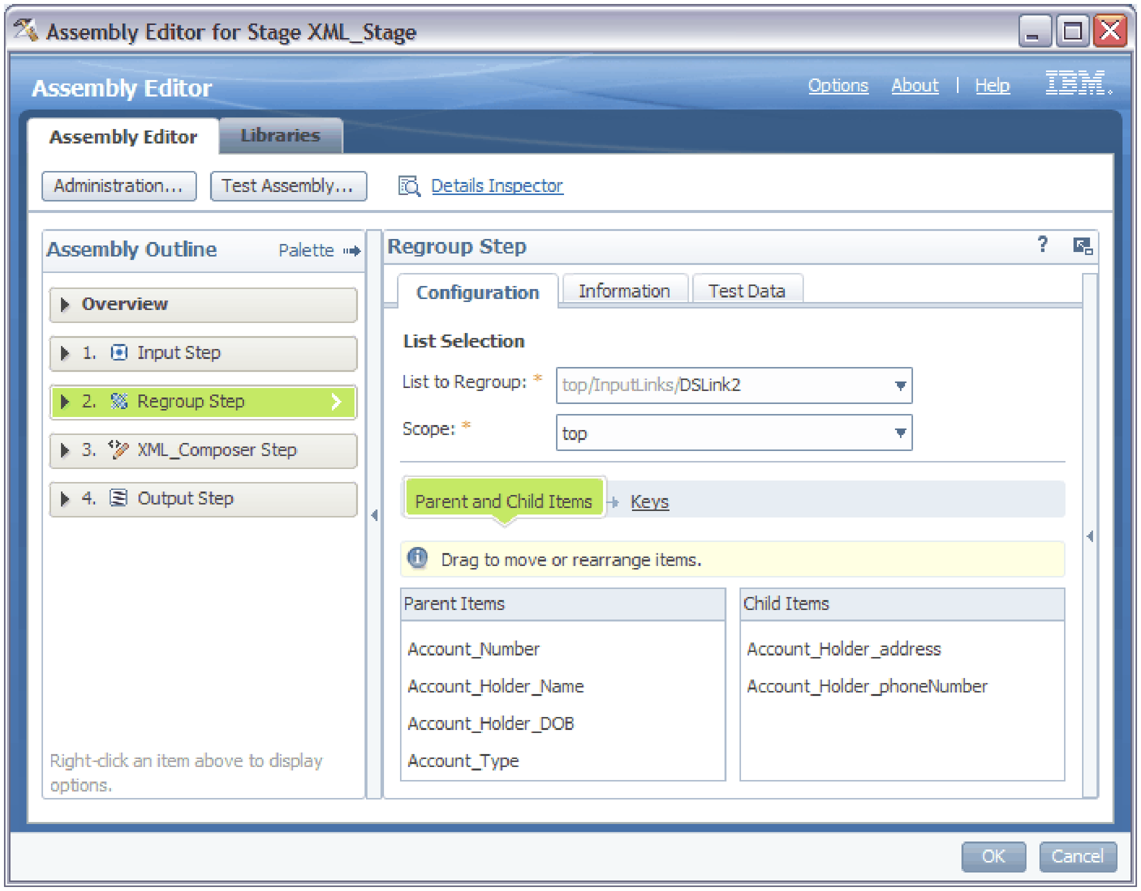
Figure 8-36 The Assembly Editor

To configure the Regroup step, follow these steps:

1. Set List to Regroup to the input list **DSLink2**. The list DSLink2 is selected, because it contains all the parent and child items under it.
2. Set the Scope to **top**. The scope defines the location of the output of the Regroup step. When `top` is defined as the scope, the output of the Regroup step is under the list `top`. The list that is selected as the scope must always be the parent of the list that is selected as the List to Regroup.
3. After you perform steps 1 and 2, all the items under the list DSLink2 can be seen in the Child Items column.
4. Because the items Account_Number, Account_Holder_Name, Account_Holder_DOB, and Account_Type are under the parent list Customer, these items are dragged into the Parent Items column.

5. The parent item Account_Number determines which child item is under it. Therefore, **Account_Number** must be set as the key in the Keys tab of the Regroup Step. Only the parent items can be selected as key values.

The Regroup Step is configured, as shown in Figure 8-37 and Figure 8-38 on page 236.



List Selection

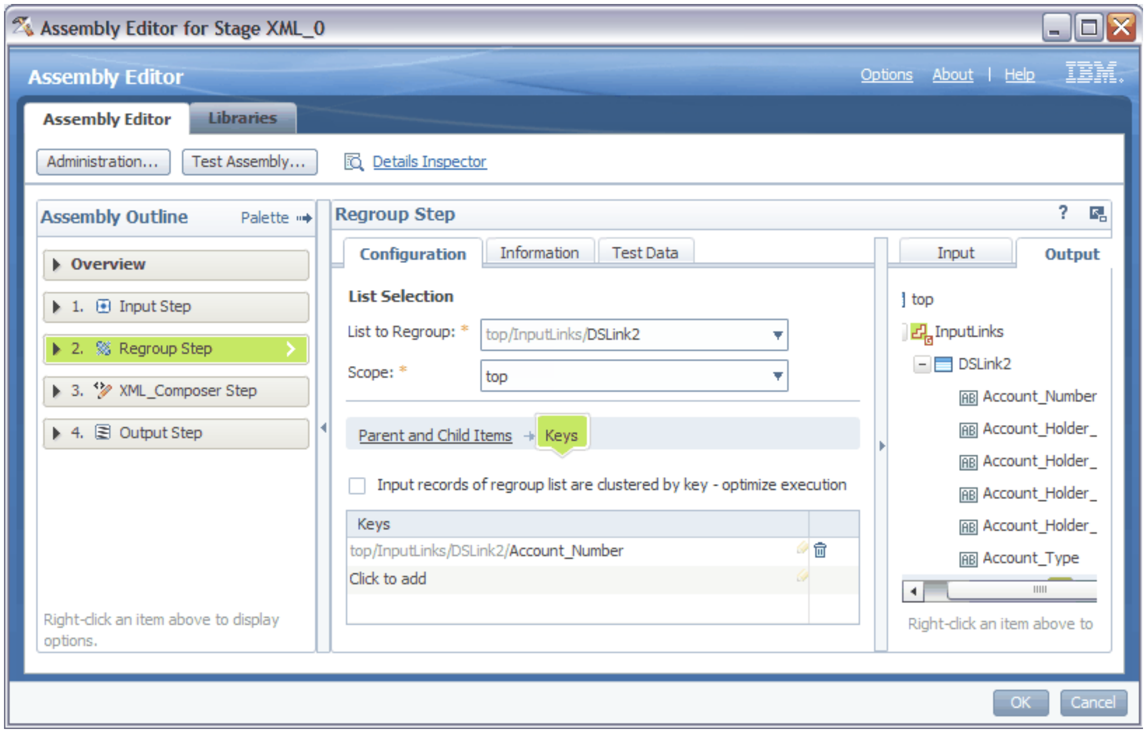
List to Regroup: * top/InputLinks/DSLink2

Scope: * top

Drag to move or rearrange items.

Parent Items	Child Items
Account_Number	Account_Holder_address
Account_Holder_Name	Account_Holder_phoneNumber
Account_Holder_DOB	
Account_Type	

Figure 8-37 The Regroup Step



List Selection

List to Regroup: * top/InputLinks/DSLink2

Scope: * top

Parent and Child Items → Keys

☐ Input records of regroup list are clustered by key - optimize execution

Keys
top/InputLinks/DSLink2/Account_Number
Click to add

Figure 8-38 Keys tab of the Regroup Step

The input and output of the Regroup step is shown in Figure 8-39 on page 237. The output of the Regroup step is under the list Regroup:result. Directly under this list are the parent items. A child list, DSLink2, is also under Regroup:result. The name of the child list is same name as the List to Regroup, which we configured in the Regroup step. As shown in Figure 8-39 on page 237, the child list DSLink2 contains all the child items.

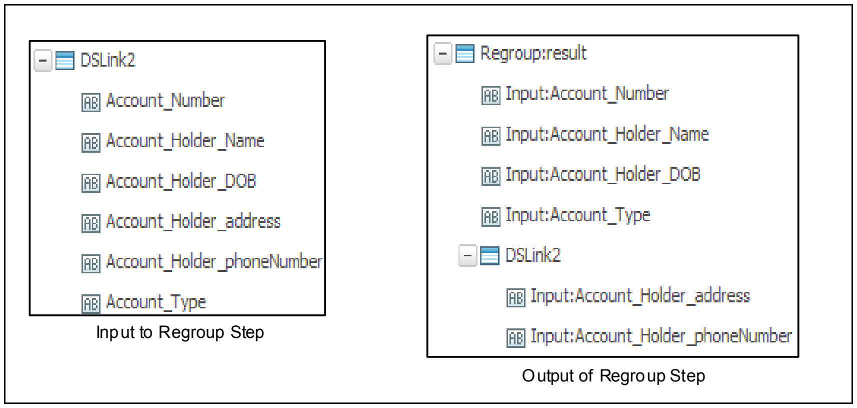


Figure 8-39 Input and output of the Regroup step

The Output of the Regroup step can now be mapped to the XML schema in the Mappings tab of the Composer step, as shown in Figure 8-40.

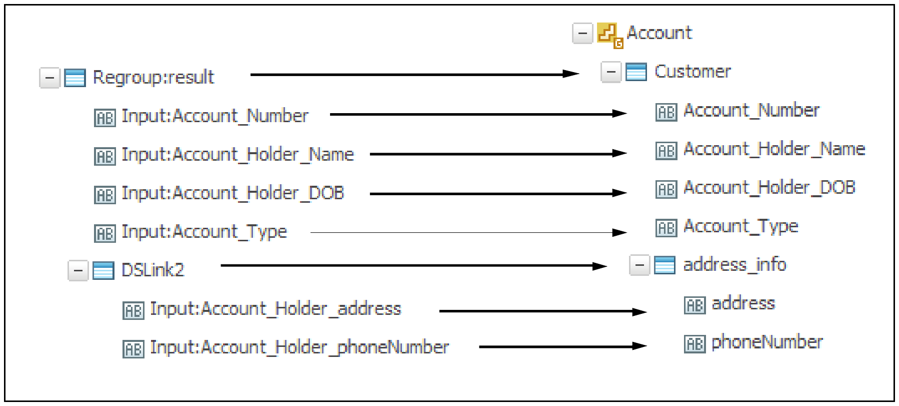
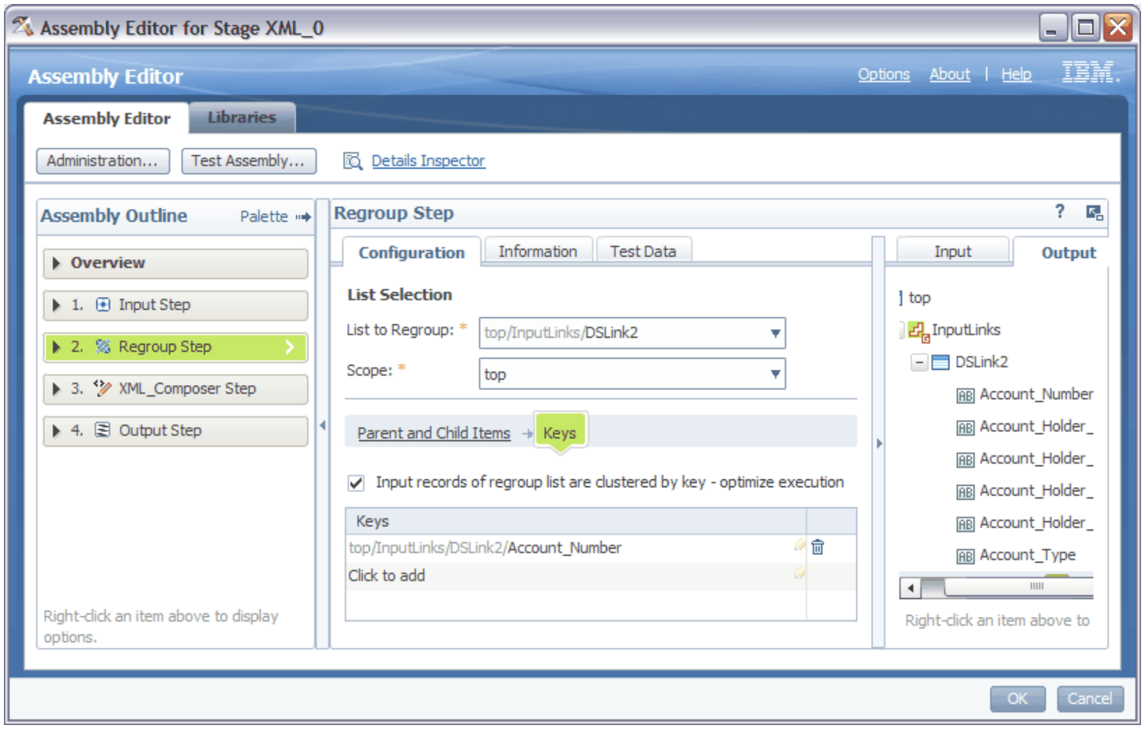


Figure 8-40 Mapping between Regroup output and Composer step

Regroup lists are clustered by key-optimize execution

When the input data is stored in a database, most often, the data is sorted. When this data is fed to the Regroup step, the Regroup step searches for all the unique key values and rearranges the data into a hierarchy. If the input data is already presorted based on the key value defined in the Regroup step, it is easy for the Regroup step to create the parent-child hierarchy. With the parent-child hierarchy, the step does not need to search through the entire input data for the unique key values.

The “Input records of regroup list are clustered by key - optimize execution” option in the Keys tab instructs the Regroup Step to regroup in a streaming fashion without storing the entire list in memory. This option also enhances the performance of the step and the overall performance of the XML Stage. This option is depicted in Figure 8-41.



List Selection

List to Regroup: * top/InputLinks/DSLink2

Scope: * top

Parent and Child Items → Keys

☑ Input records of regroup list are clustered by key - optimize execution

Keys
top/InputLinks/DSLink2/Account_Number
Click to add

top
InputLinks
DSLink2
Account_Number
Account_Holder_
Account_Holder_
Account_Holder_
Account_Holder_
Account_Type

Right-click an item above to

Figure 8-41 The Input records of regroup lists are clustered by key - optimize execution option

In case the data is not sorted in the database, the data can be sorted before feeding it to the XML Stage. We advise that you sort in DataStage rather than in XML Stage, because the DataStage Sort operates in parallel and can easily sort the relational data that comes from the database. The Sort step within the XML Stage is appropriate for sorting hierarchical data. Figure 8-42 shows the overall DataStage job design. The option “Input records of regroup lists are clustered by key - optimize execution” is used within the Regroup step.

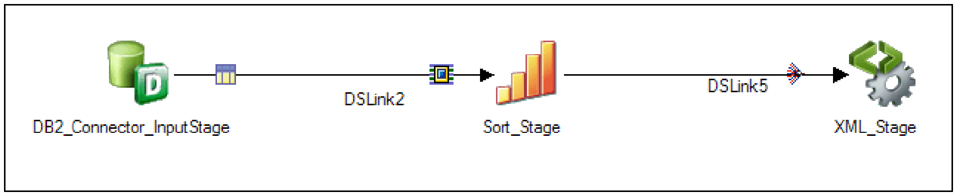
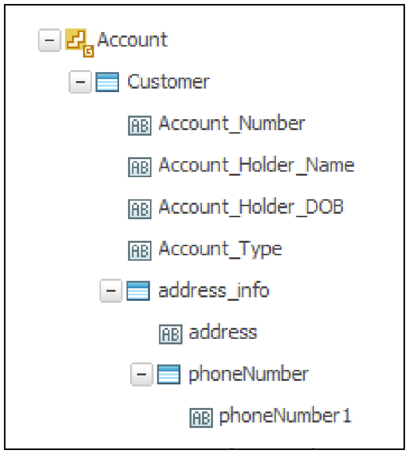


Figure 8-42 The DataStage job design

Creating multiple levels of hierarchy

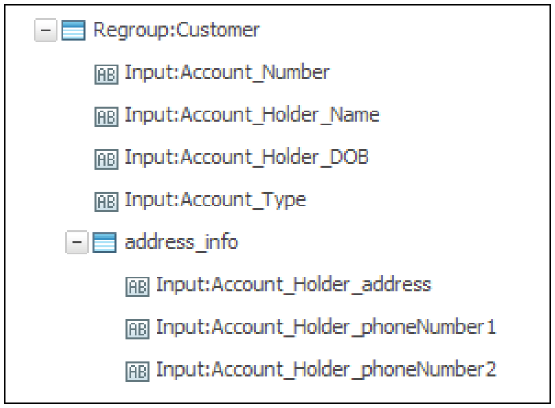
The bank also uses complicated schemas that involve multiple levels of hierarchy. Figure 8-43 shows a complex schema; the parent list Customer contains the list address_info, and address_info contains the list phoneNumber.



```
Account
  Customer
    Account_Number
    Account_Holder_Name
    Account_Holder_DOB
    Account_Type
    address_info
      address
      phoneNumber
        phoneNumber1
```

Figure 8-43 An XML schema with multiple, hierarchical levels

To create this hierarchy, two Regroup steps can be used. In the first Regroup step, create the first level of hierarchy, which is between Customer and address_info. The address_info link also contains the phoneNumber columns, as shown in Figure 8-44.

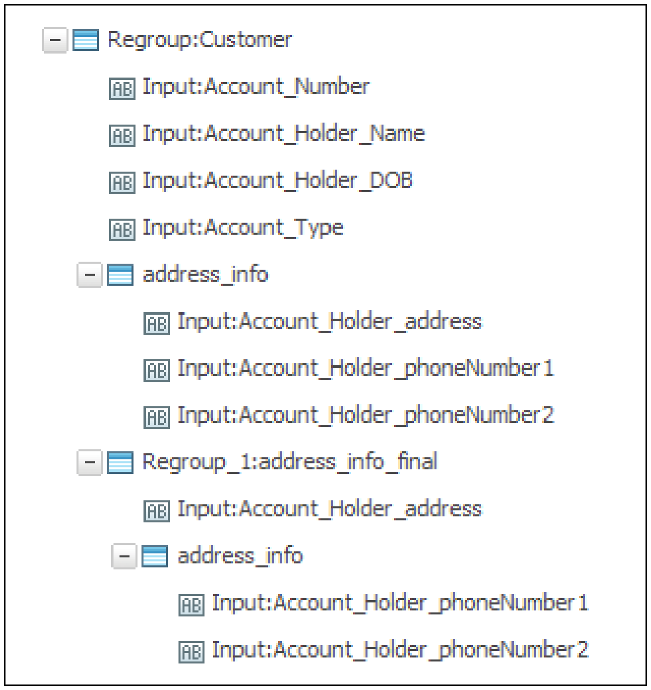


```
Regroup:Customer
  Input:Account_Number
  Input:Account_Holder_Name
  Input:Account_Holder_DOB
  Input:Account_Type
  address_info
    Input:Account_Holder_address
    Input:Account_Holder_phoneNumber1
    Input:Account_Holder_phoneNumber2
```

Figure 8-44 The hierarchy created by the first Regroup step

The second Regroup step creates the second hierarchy level, which is between address_info and phoneNumber. Remember to set the scope in the second

Regroup step to Customer so that the output of the second Regroup step is under the list Customer. Figure 8-45 shows the final hierarchy that is created.



```
Regroup:Customer
    Input:Account_Number
    Input:Account_Holder_Name
    Input:Account_Holder_DOB
    Input:Account_Type
    address_info
        Input:Account_Holder_address
        Input:Account_Holder_phoneNumber1
        Input:Account_Holder_phoneNumber2
    Regroup_1:address_info_final
        Input:Account_Holder_address
        address_info
            Input:Account_Holder_phoneNumber1
            Input:Account_Holder_phoneNumber2
```

Figure 8-45 Final hierarchy created by using two Regroup steps

The address_info list can be ignored, because the actual list is address_info_final.

8.2.2 The HJoin step

The bank must create an XML file from source data in two DB2 tables. The data in the databases must be combined to create the XML file. Therefore, two input links are necessary to the XML Stage. Figure 8-46 on page 241 shows the DataStage job design.

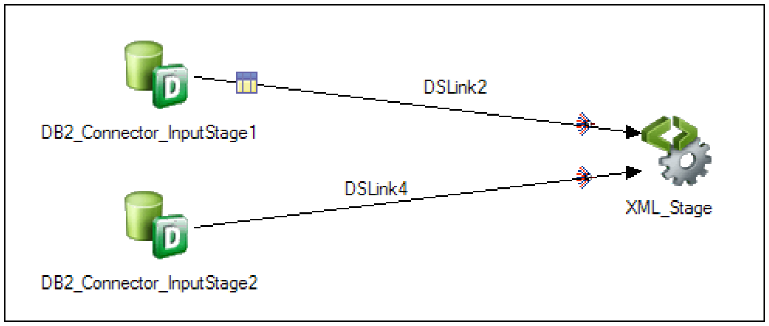


Figure 8-46 The HJoin DataStage job design

The XML Stage needs to combine the data from the two input links into a single XML structure, as shown in Figure 8-47.

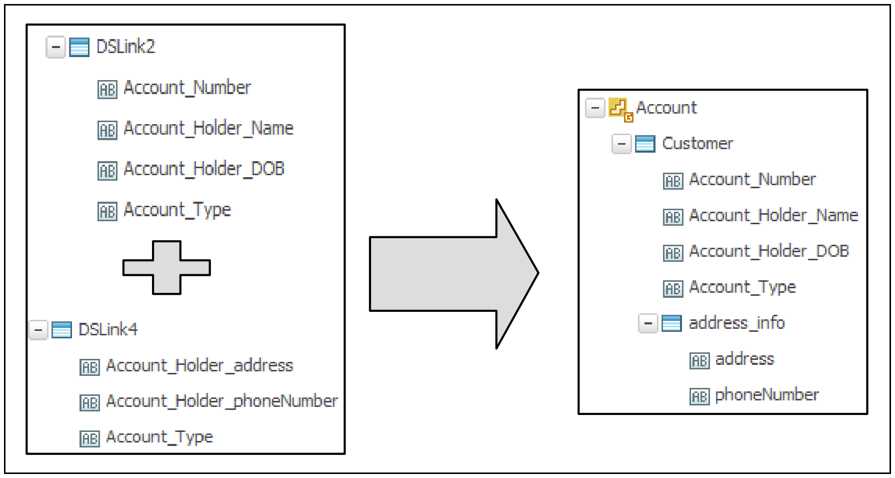


Figure 8-47 Combining two input files to create a single XML file

The HJoin step is used to join data that comes from two input links. This join is called a *Hierarchical Join*, because the step creates a hierarchical structure. Therefore, the HJoin step joins the data and creates the hierarchy. To be able to map the data from the input database to the Composer step, the HJoin step must be used.

The HJoin step must be added to the assembly to configure the HJoin step. Follow these steps to add the HJoin step to the assembly:

1. Open **Assembly Editor** and click **Palette** in the Assembly Outline bar. The steps show in the Palette, as shown in Figure 8-48 on page 242.

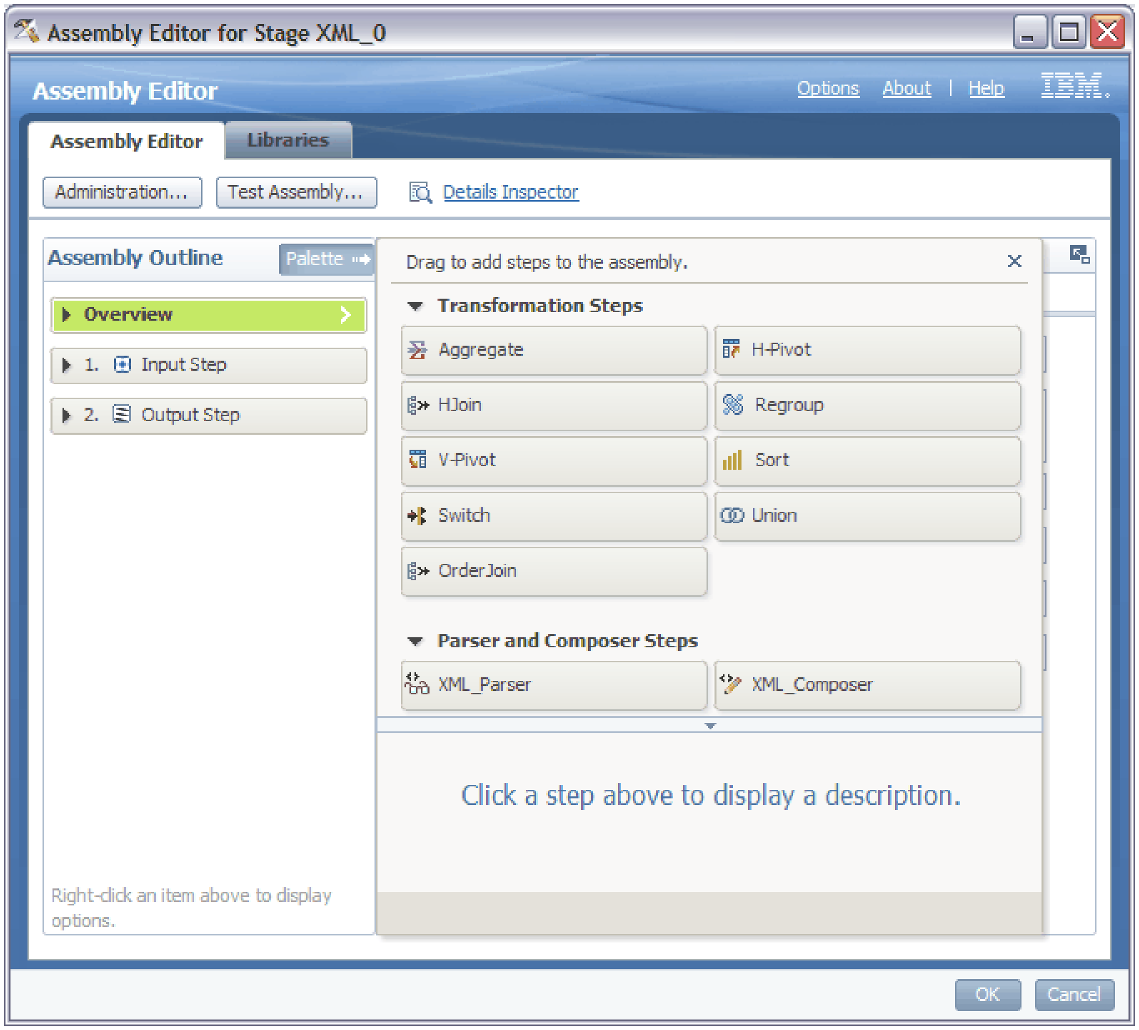
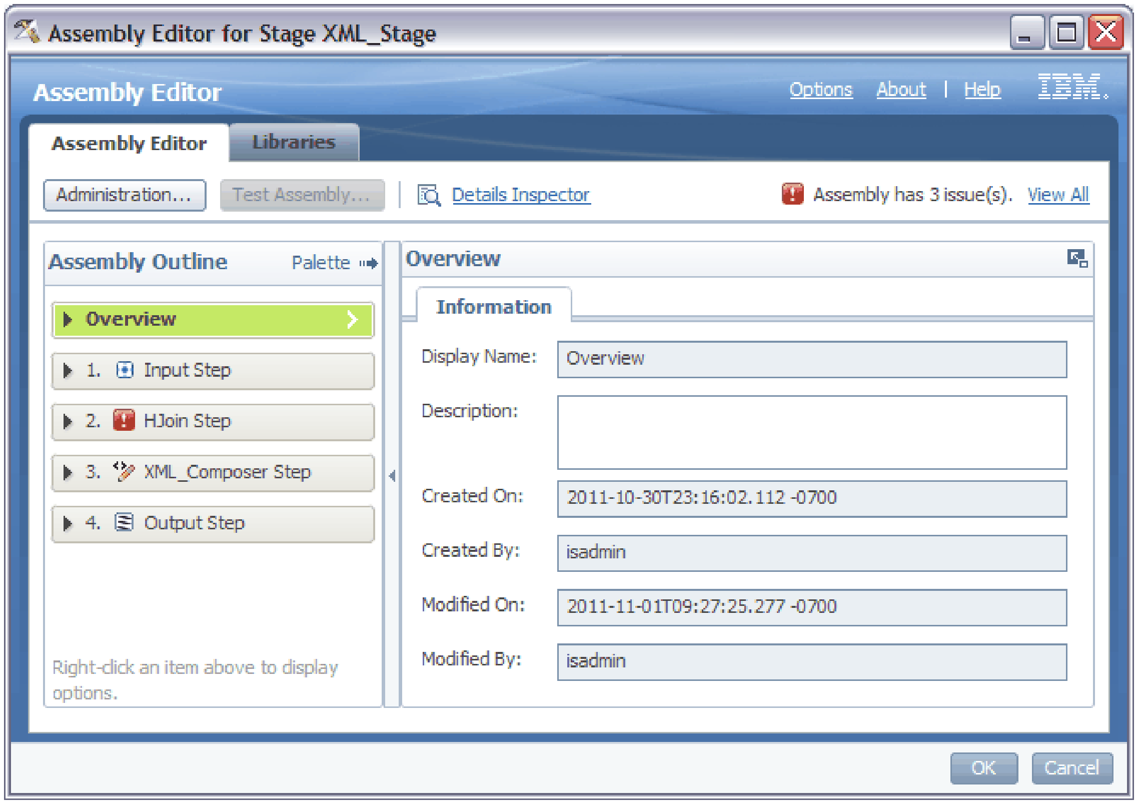


Figure 8-48 The Palette in the Assembly Editor

2. Drag the **HJoin** step from the Palette onto the Assembly Outline bar after the Input Step. The HJoin step is added between the Input Step and the Composer Step, as shown in Figure 8-49 on page 243.



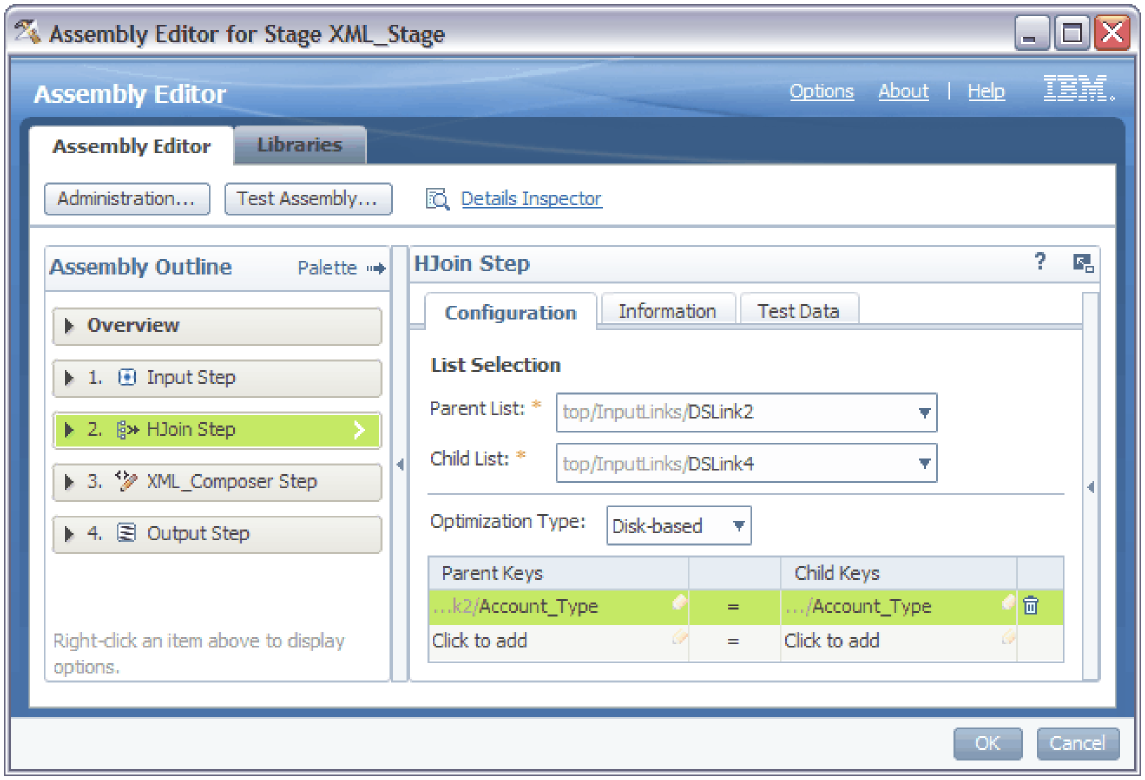
Display Name: Overview
Description:
Created On: 2011-10-30T23:16:02.112 -0700
Created By: isadmin
Modified On: 2011-11-01T09:27:25.277 -0700
Modified By: isadmin

Figure 8-49 The HJoin Step added to the Assembly Outline

To configure the HJoin step, follow these steps:

1. For the Parent List and Child List options, specify the input lists, DSLink2 and DSLink4, to the HJoin step. Because the items under list DSLink2 are the parent items in the XML schema structure defined in the Composer step, specify **DSLink2** as the Parent List. Specify **DSLink4** as the Child List.
2. Both the database tables have the common column, Account_Type, which helps to create a relationship between the two tables. This column is used to join the data that comes into the HJoin step. Therefore, select **Account_Type** for Parent Keys and Child Keys.
3. The Optimization Type determines where the join occurs. For large input data, use **Disk-based**, which is the default, for the Optimization Type.

The HJoin step is configured, as shown in Figure 8-50 on page 244.



List Selection

Parent List: * top/InputLinks/DSLink2

Child List: * top/InputLinks/DSLink4

Optimization Type: Disk-based

Parent Keys		Child Keys
...k2/Account_Type	=	.../Account_Type
Click to add	=	Click to add

Figure 8-50 The complete HJoin Step

Figure 8-51 on page 245 shows the output of the HJoin step. The output contains a nested structure. Under the input parent list DSLink2, the child list is added along with the prefix HJoin, for example, `HJoin:DSLink4`.

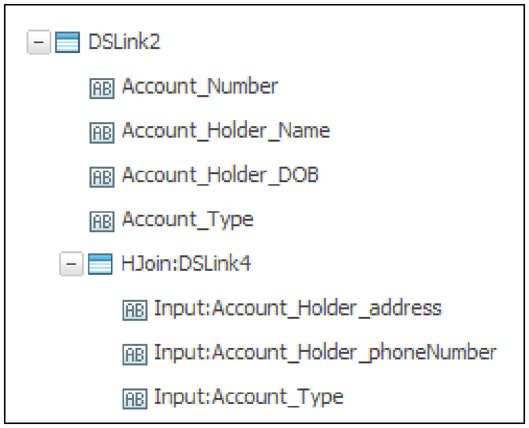

DSLink2
Account_Number
Account_Holder_Name
Account_Holder_DOB
Account_Type
HJoin:DSLink4
Input:Account_Holder_address
Input:Account_Holder_phoneNumber
Input:Account_Type

Figure 8-51 The output of the HJoin step

In several rows of the child list, the value of the column Account_Type might not match a value in the parent list. All of these rows are placed under a single group called HJoin:orphans, as shown in Figure 8-52.

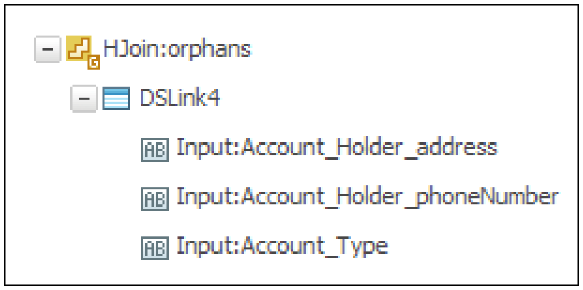

HJoin:orphans
DSLink4
Input:Account_Holder_address
Input:Account_Holder_phoneNumber
Input:Account_Type

Figure 8-52 The orphan child records

The output of the HJoin step can now be mapped to the XML schema in the Mappings tab of the Composer step, as shown in Figure 8-53 on page 246.

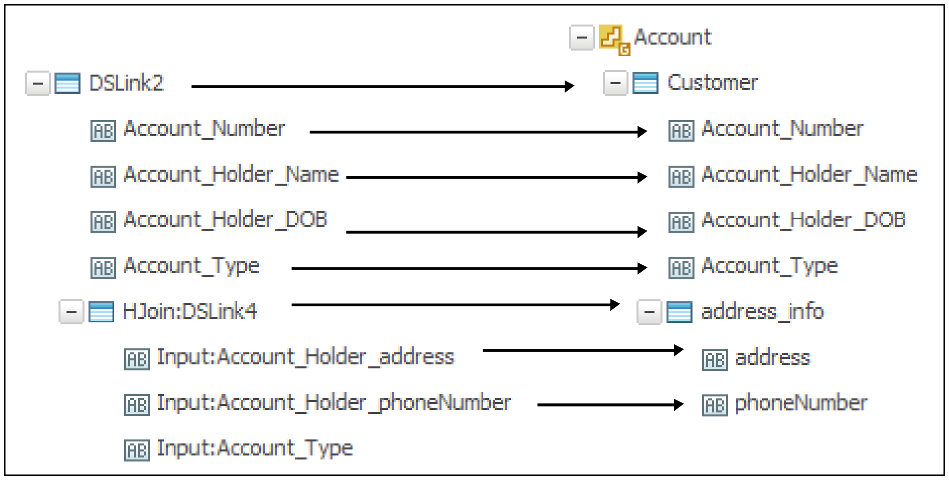


Figure 8-53 The mapping

8.2.3 Regroup step versus HJoin step

You can use both the Regroup step and the HJoin step to create a hierarchical structure.

Use the Regroup step when a single input file is provided to the XML Stage, and the input relational data has duplicate rows. For example, consider the data that is shown in Table 8-2.

Table 8-2 Single input file with duplicate rows

Department ID	Employee first name	Employee last name	Date of birth	Employee ID
A100	Jane	Smith	1986-08-06	Z1098
B200	John	Doe1	1977-10-30	Y9876
A100	John	Doe2	1969-01-01	X7656
C300	Jane	Smith1	1988-01-09	W2345
C300	Jane	Smith2	1972-05-26	T7400

To create a hierarchical structure of each department and its employees, you can use the Regroup step. Several of the Department IDs are repeated. By using the Regroup step, the redundancy in the Department IDs can be removed.

Use the HJoin step when multiple input sources are required. This step joins the data in the input source to create a hierarchical structure. Consider the two input sources that are shown in Table 8-3 and Table 8-4.

Table 8-3 Multiple sources: Input 1

Department ID	Department name
A100	Human resources
B200	Product sales
C300	Software development

Table 8-4 Multiple sources: Input2

Department ID	Employee first name	Employee last name	Date of birth	Employee ID
A100	Jane	Smith	1986-08-06	Z1098
B200	John	Doe1	1977-10-30	Y9876
A100	John	Doe2	1969-01-01	X7656
C300	Jane	Smith1	1988-01-09	W2345
C300	Jane	Smith2	1972-05-26	T7400

When an HJoin is performed between these tables, the Department IDs A100 and C300 repeat, because the HJoin step does not remove any redundancy while performing a join. Therefore, the Regroup step must be used. Because the Regroup step requires a single input list, both tables can be combined by using the Join_Stage that is available in DataStage. Figure 8-54 shows the DataStage job design.

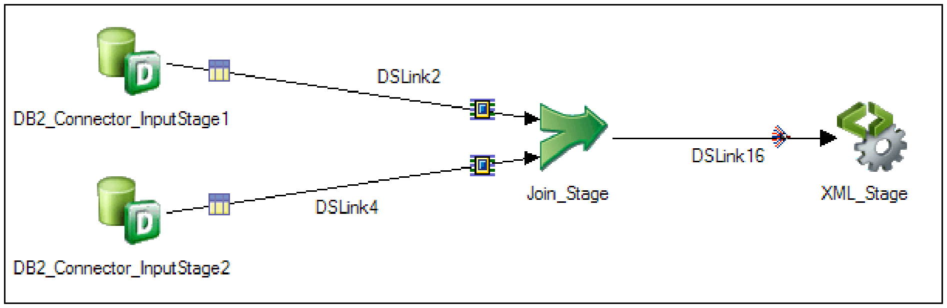


Figure 8-54 The Join DataStage Job design

To create a hierarchy with multiple levels, you can use multiple Regroup steps or HJoin steps in the assembly.

8.2.4 H-Pivot step

The H-Pivot step is used to transform multiple columns into a single hierarchical column under a list. For example, consider the input from the database, which is information about a departmental store. In the input, two columns, BeautyPr and HomePr, contain information about the products in the store. Both the columns, BeautyPr and HomePr, need to be mapped to a single column that is under a list, as shown in Figure 8-55. Use the H-Pivot step to map multiple columns in the input to a single column under a list in the output.

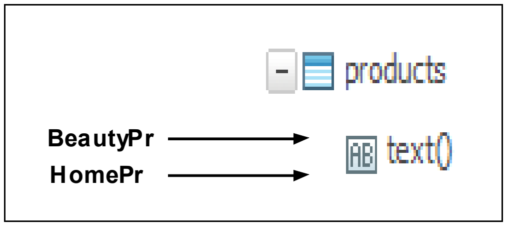


Figure 8-55 Mapping input columns to the schema structure

The bank uses a schema that contains the customer account information, as shown in Figure 8-56.

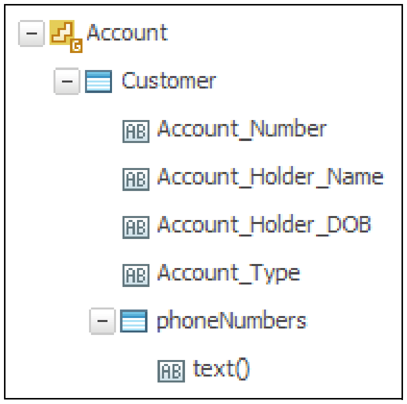
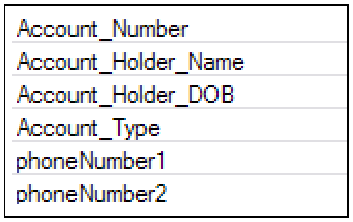


Figure 8-56 The XML schema structure

The Composer step is used to compose XML files. To map the data from the Input step to the Composer step, the hierarchy needs to be created. The input columns are shown in Figure 8-57 on page 249. Therefore, two input columns, phoneNumber1 and phoneNumber2, need to be mapped to a single output column, text(), under the list phoneNumbers.

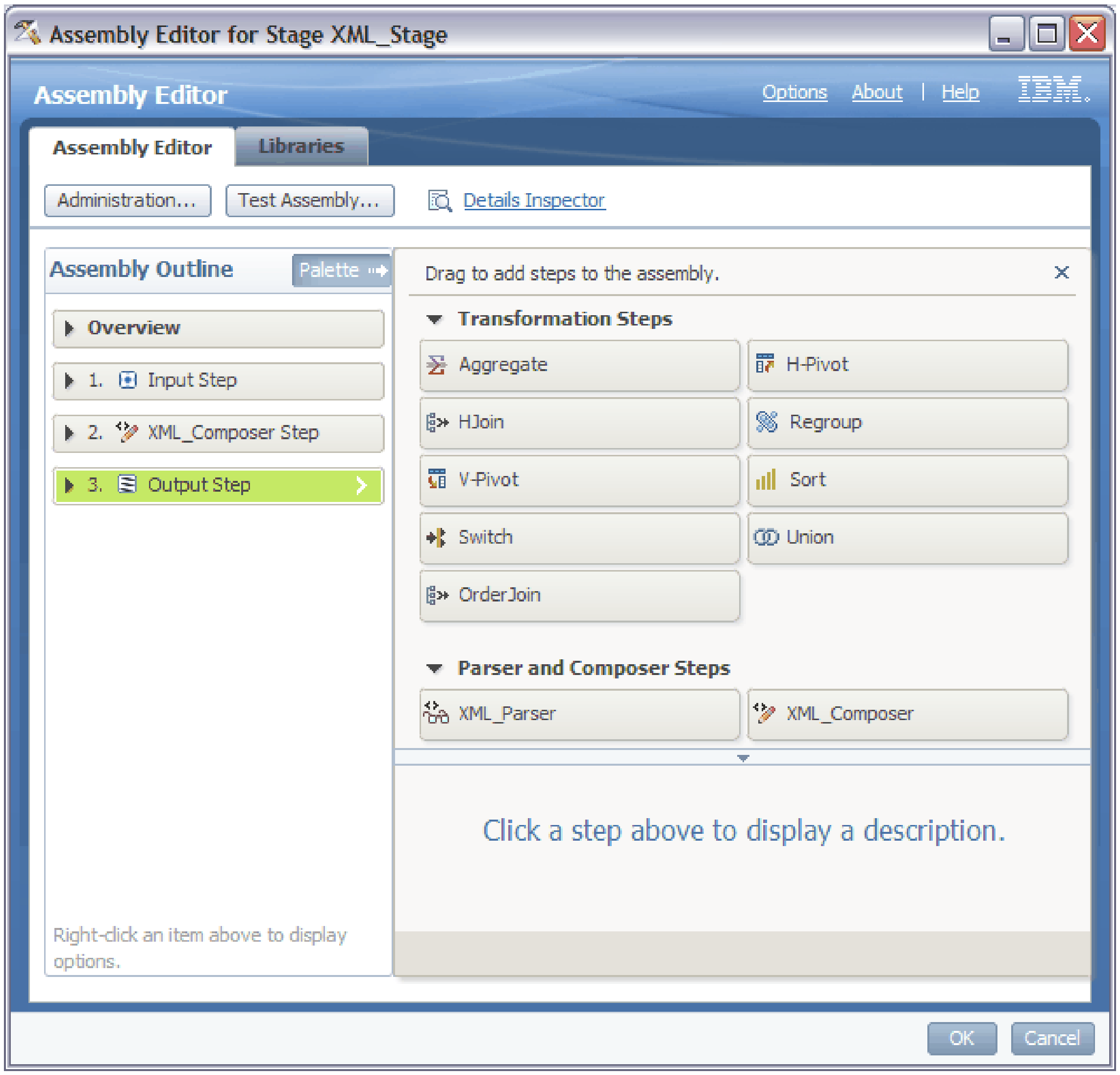


Account_Number
Account_Holder_Name
Account_Holder_DOB
Account_Type
phoneNumber1
phoneNumber2

Figure 8-57 The input columns

You need to add the H-Pivot step to the assembly before you configure it. To add the H-Pivot step, use the following steps:

1. Open **Assembly Editor** and click **Palette** in the Assembly Outline bar, which shows the various steps in the Palette, as shown in Figure 8-58 on page 250.



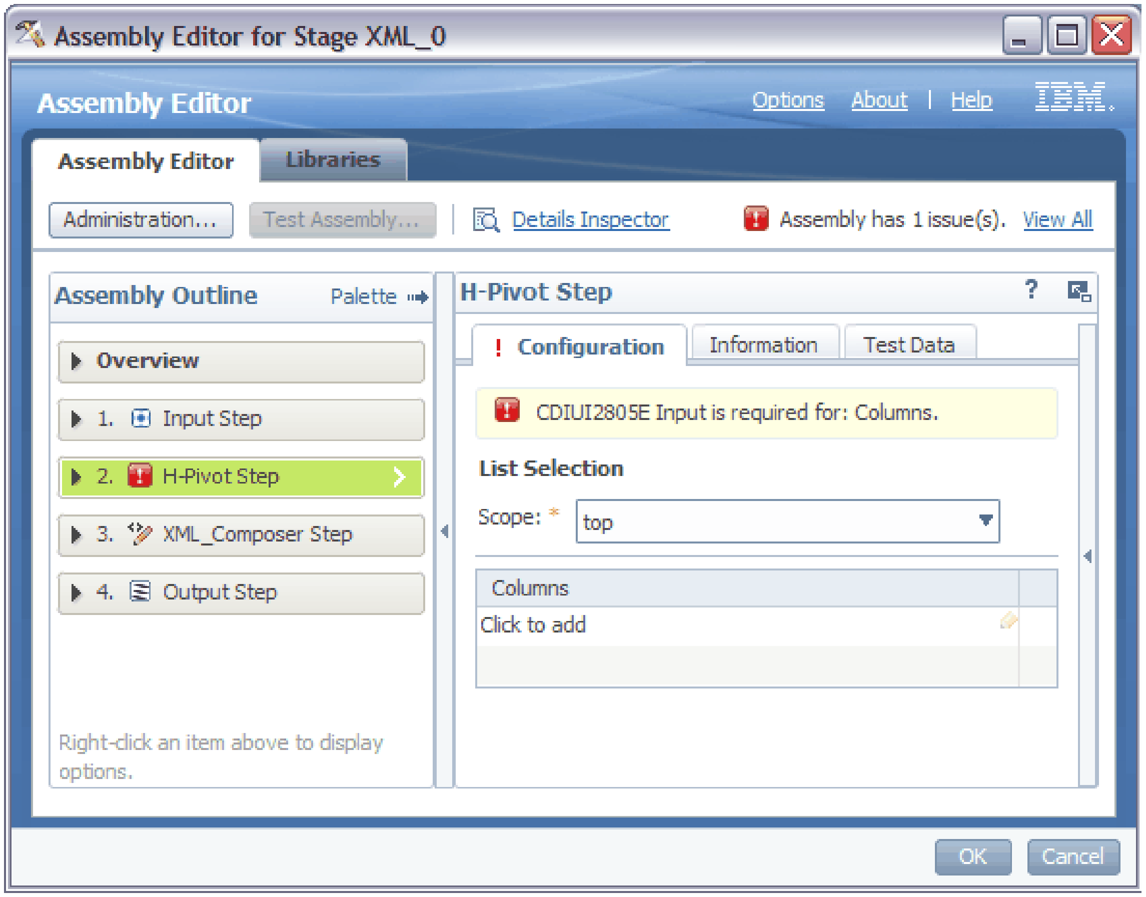
Drag to add steps to the assembly.

Click a step above to display a description.

Right-click an item above to display options.

Figure 8-58 The Palette in the Assembly Editor

2. Drag the **H-Pivot** step from the Palette onto the Assembly Outline bar after the Input Step. The H-Pivot Step is added between the Input Step and the Composer Step, as shown in Figure 8-59 on page 251.



CDIUI2805E Input is required for: Columns.

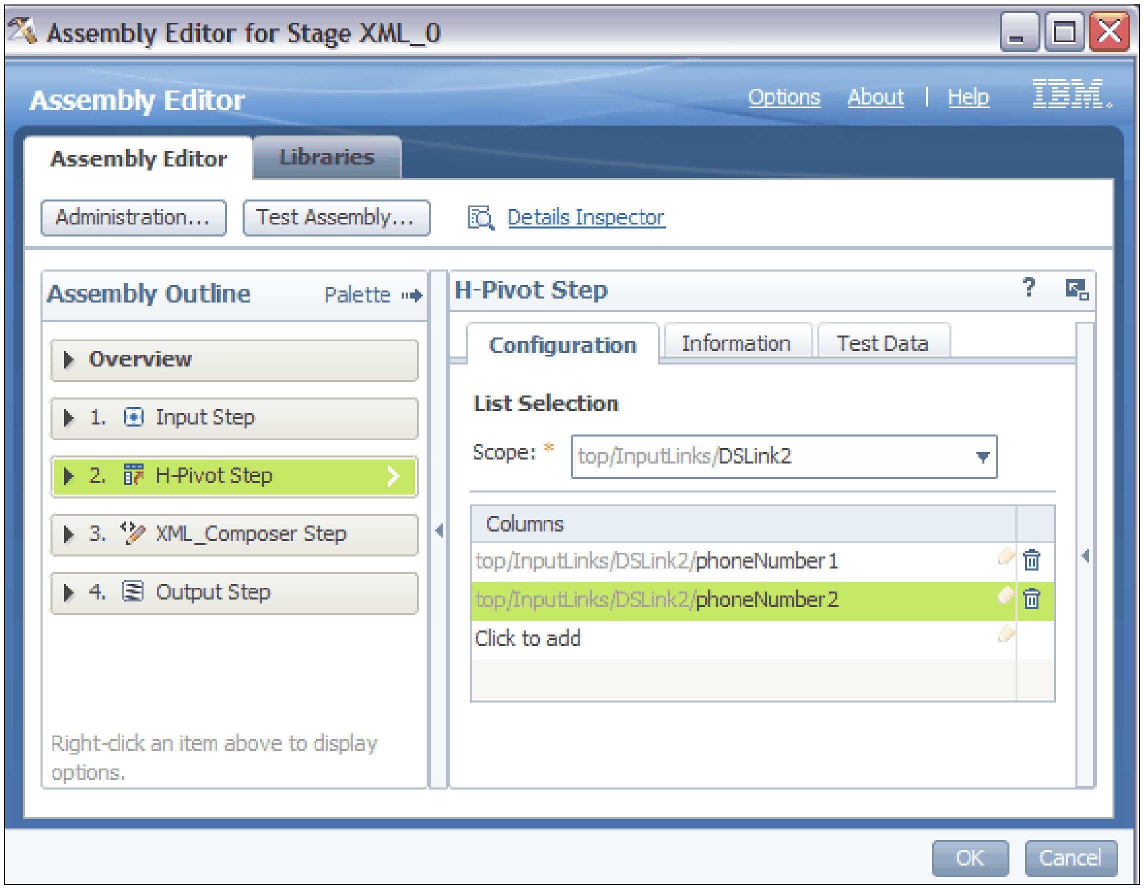
Scope: * top

Figure 8-59 The H-Pivot Step in the Assembly Editor

To configure the H-Pivot step, use the following steps:

1. Set the Scope as the list that contains the phoneNumber1 and phoneNumber2 fields. Because these fields are under the input list DSLink2, select **DSLink2** for Scope.
2. In the Columns drop-down list, select the input columns. Select **phoneNumber1** in the first drop-down list and **phoneNumber2** in the second drop-down list. Click **OK**.

The H-Pivot step is configured, as shown in Figure 8-60 on page 252.



Scope: * top/InputLinks/DSLink2

Columns
top/InputLinks/DSLink2/phoneNumber1
top/InputLinks/DSLink2/phoneNumber2
Click to add

Figure 8-60 Configured H-Pivot Step

The output of the H-Pivot step is contained under the scope DSLink2. The output of the H-Pivot step is under the group H-Pivot:result, as shown in Figure 8-61 on page 253. It contains the *rows* list, which contains the child items: name and value. *Name* contains the name of the fields selected as columns (in this scenario, Name stores phoneNumber1 and phoneNumber2). *Value* contains the data that is stored in the phoneNumber1 and phoneNumber2 columns.

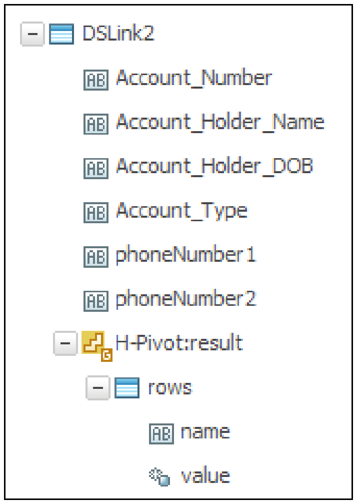


Figure 8-61 The output of the H-Pivot step

The output of the H-Pivot step can now be mapped to the phoneNumbers list, as shown in Figure 8-62.

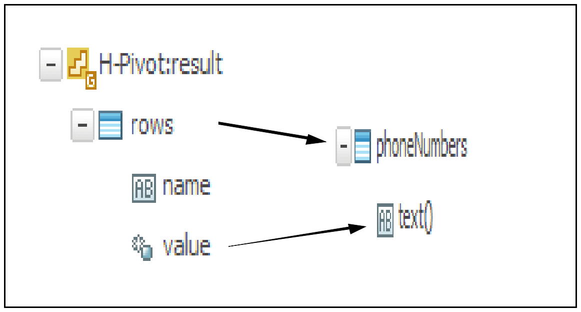
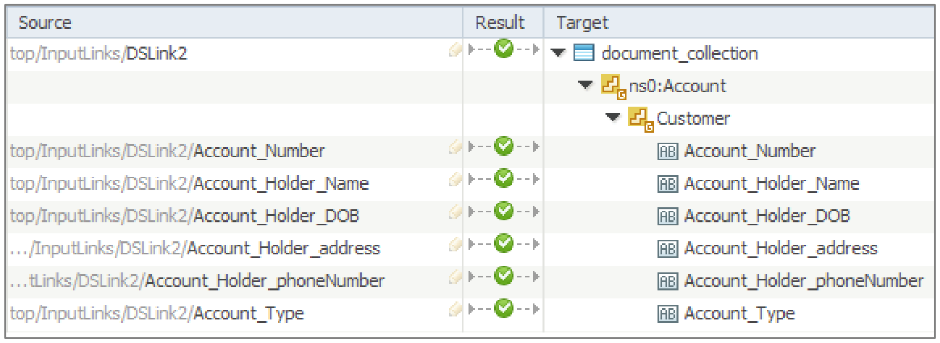


Figure 8-62 Mapping the H-Pivot step output and Composer step schema structure

8.3 Multiple XML documents: Single Composer step

Use the Mappings tab of the Composer step to map the incoming data to the XML schema structure to create the XML document. The document_collection is the first target list in the Mappings table. Based on the mapping to this list, one or multiple XML documents are created. By default, the `top` list is mapped to the document_collection list so that the XML Composer step creates only one XML file.

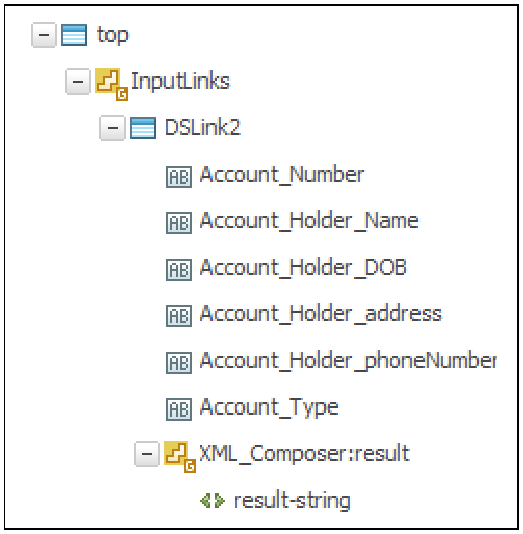
The bank must create XML files by using the data in the database. The bank wants to create one XML file for every input row that comes from the DB2 database table. The input structure must be mapped to the schema structure in the Mappings tab of the Composer step. To create one XML file for every incoming input row, map the input list DSLink2 to the document_collection list. Now, you can map the rest of the target items, as shown in Figure 8-63.



Source	Result	Target
top/InputLinks/DSLink2	✓	document_collection
		ns0:Account
		Customer
top/InputLinks/DSLink2/Account_Number	✓	Account_Number
top/InputLinks/DSLink2/Account_Holder_Name	✓	Account_Holder_Name
top/InputLinks/DSLink2/Account_Holder_DOB	✓	Account_Holder_DOB
.../InputLinks/DSLink2/Account_Holder_address	✓	Account_Holder_address
...tLinks/DSLink2/Account_Holder_phoneNumber	✓	Account_Holder_phoneNumber
top/InputLinks/DSLink2/Account_Type	✓	Account_Type

Figure 8-63 Mapping the complete document_collection list

The XML Composer result is contained in the `top` list when the `top` list is mapped to the document_collection list. When the input list DSLink2 is mapped to the document_collection list, the XML Composer result is contained in the DSLink2 list, as shown in Figure 8-64.



```
top
  InputLinks
    DSLink2
      Account_Number
      Account_Holder_Name
      Account_Holder_DOB
      Account_Holder_address
      Account_Holder_phoneNumber
      Account_Type
      XML_Composer:result
        result-string
```

Figure 8-64 The XML Composer result that is contained in the DSLink2 list

8.3.1 Target tab configuration for creating multiple XML documents

The XML Target tab describes the method of writing the XML document to the target. It provides three options for writing the XML file: Write to File, Pass as String, and Pass as Large Object. We described these options in detail in 8.1.1, “Writing XML files” on page 202. The configuration of the XML Target tab also determines how to write the multiple XML files, which we created, to the target. In the Mappings tab, the input list DSLink2 is mapped to the document_collection list. Therefore, for every input row under the list DSLink2, a new XML file is created by the Composer step.

When the Write to File option is selected in the XML Target tab of the Composer step, the Composer step directly writes the XML file onto the server. It creates one XML document for every input row. For example, if three input rows come into the XML Stage and the Filename Prefix value is file, the Composer step creates three files with the names `file.xml`, `file_1.xml`, and `file_2.xml`.

When the Pass as String or Pass as Large Object option is selected in the XML Target tab of the Composer step, the Composer step passes the created XML file to the stage after XML Stage. The Composer step does not directly write the file onto the server. By choosing this option, the Composer step creates a single item that contains both files. For example, if there are two input rows, the result of the Composer step is a single item, result_string, which contains the data that is shown in Example 8-3.

Example 8-3 Single item for multiple rows

```
<?xml version="1.0" encoding="UTF-8"?>
<Account>
   <Customer>
      <Account_Number>C198765487</Account_Number>
      <Account_Holder_Name>John Doe1</Account_Holder_Name>
      <Account_Holder_DOB>02-07-1980</Account_Holder_DOB>
      <Account_Holder_address>San Jose,CA</Account_Holder_address>
      <Account_Holder_phoneNumber>123-456-654-4567</Account_Holder_phon
      eNumber>
      <Account_Type>Private</Account_Type>
   </Customer>
</Account>
<?xml version="1.0" encoding="UTF-8"?>
<Account>
   <Customer>
      <Account_Number>C198765444</Account_Number>
      <Account_Holder_Name>Jane Smith</Account_Holder_Name>
      <Account_Holder_DOB>1975-01-01</Account_Holder_DOB>
      <Account_Holder_address>San Jose,CA</Account_Holder_address>
      <Account_Holder_phoneNumber>123-789-987-2457</Account_Holder_phon
      eNumber>
      <Account_Type>non-savings</Account_Type>
</Customer>
</Account>
```

Both the XML documents are embedded in a single file.

8.4 Configuring the Output step

When the *Pass as String* or *Pass as Large Object* option is selected in the XML Target tab of the Composer step, the Composer step passes the created XML file to the stage after the XML Stage. The Output step must be used to pass the created XML file to the downstream stage. For example, if the XML file must be written into a pureXML column of a database, the Output step is required.

When an output link is provided from the XML Stage, the link is a list within the assembly. The output columns are items in the assembly. The columns for each of the output links are visible in the Output tab of the Output Step. The Mappings tab is used to map the output of the Composer Step to the columns of the database. As shown in Figure 8-65 on page 257, the output link DSLink3 has a single column result to which the Composer Result must be mapped.

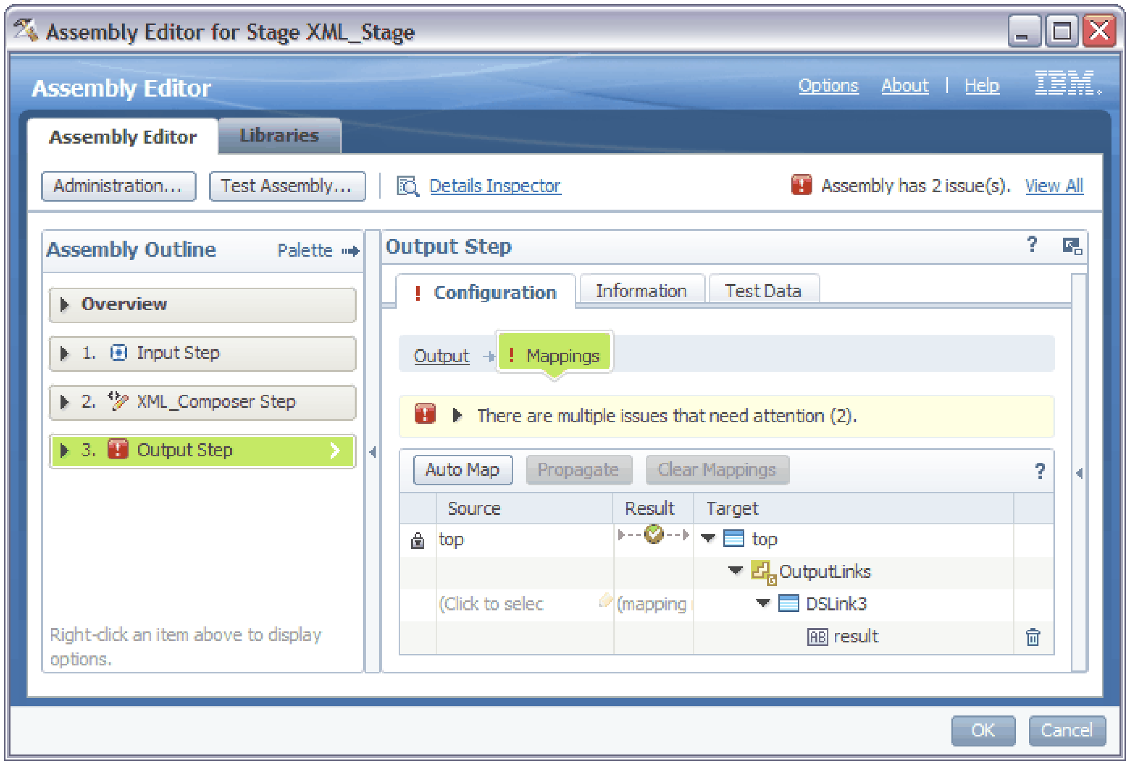
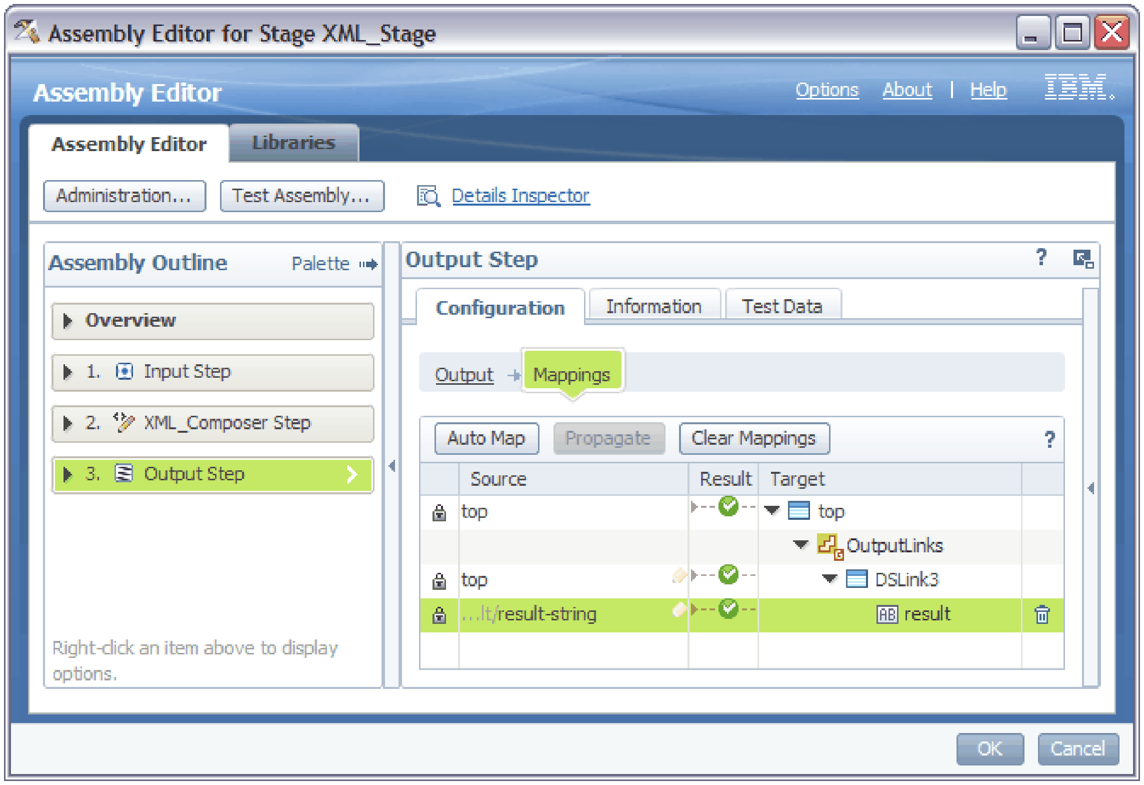


Figure 8-65 The Mappings tab for mapping the output of the Composer Step

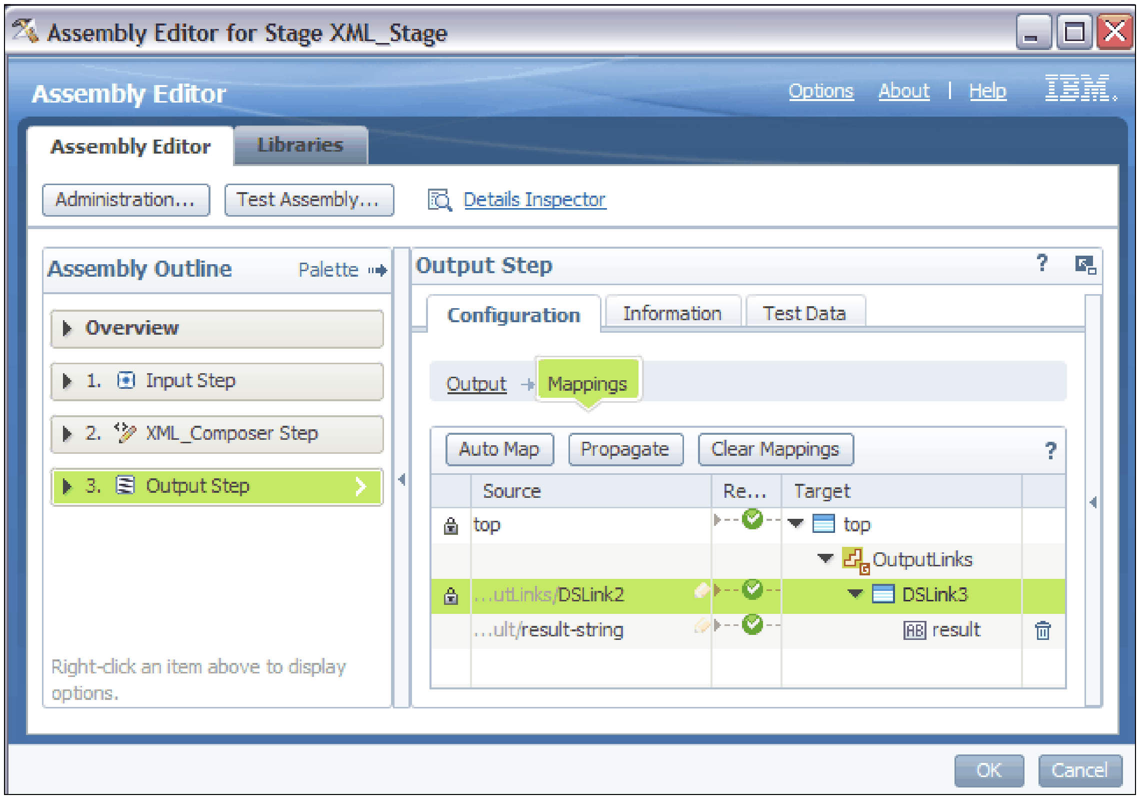
When the `top` list is mapped to the document_collection list, the output of the Composer step is under the list `top`. The Composer result is contained in a single item called result_string. To map result_string to the output column, you must first map the parent list DSLink3. Because result_string is under the list `top`, `top` is mapped to the target list DSLink3. Then, the source item result_string is mapped to the target item result. Figure 8-66 on page 258 shows the complete mapping.



Source	Result	Target
top	✓	top
		OutputLinks
top	✓	DSLink3
...lt/result-string	✓	result

Figure 8-66 Mapping the parent list to result_string

If the input link DSLink2 is mapped to the document_collection item in the Mappings tab of the Composer step, the output of the Composer step is under the input list DSLink2. To map result_string to the output column, you first must map the parent list DSLink3. Because the item result_string is under the input list DSLink2, DSLink2 is mapped to the target list DSLink3. Then, the source item result_string is mapped to the target item result. Figure 8-67 on page 259 shows the complete mapping.



Source	Re...	Target
top		top
		OutputLinks
...utLinks/DSLink2		DSLink3
...ult/result-string		result

Figure 8-67 The complete mapping for result_string

8.5 Creating XML files for various kinds of schemas

When the XML schema is imported into the Schema Library Manager, it is converted into a simplified model. This simplified model is used by the Composer step to create the XML files. We describe the representation of the schema in the assembly in detail in Chapter 6, “Introduction to XML Stage” on page 131. For certain schema definitions, the XML Stage adds its own items to the schema representation. You must pay special attention to composing the XML files for these schemas.

8.5.1 Creating an XML file for a schema with a choice definition

The bank uses a schema in which one of the complex elements is defined with *xs:choice*. The XML schema is shown in Figure 8-68 on page 260.

The item @@choiceDiscriminator is added by the XML Stage before the self_employed and hired elements to signify the xs:choice in the schema definition. The @@choiceDiscriminator holds the name of the choice branch selected at run time. For the XML schema that is shown in Figure 8-68, the output XML file can contain either the element self_employed or the element hired, but not both elements. While composing the XML file for this schema, the correct value needs to be mapped to the @@choiceDiscriminator.

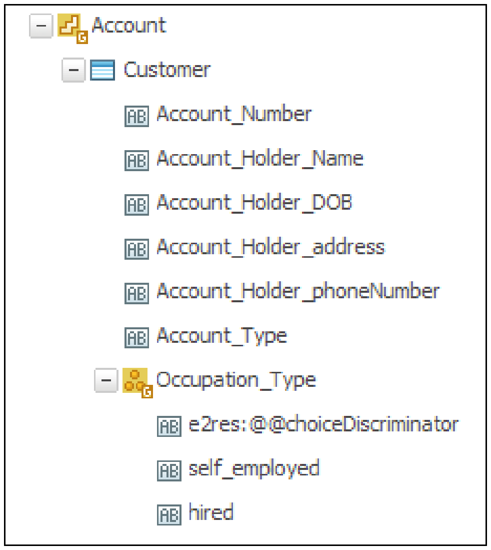
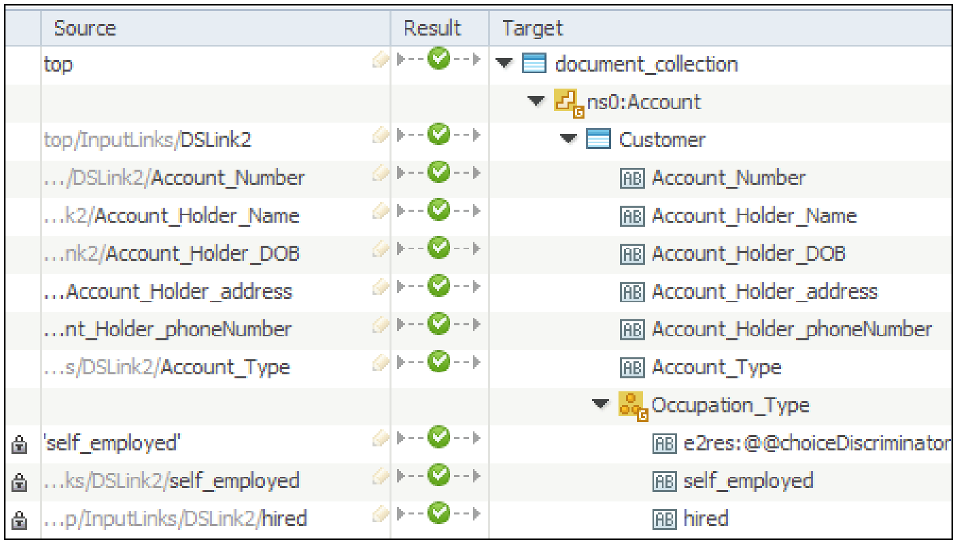


Figure 8-68 The item @@choiceDiscriminator mapped

These mapping rules apply when you map the @@choiceDiscriminator item:

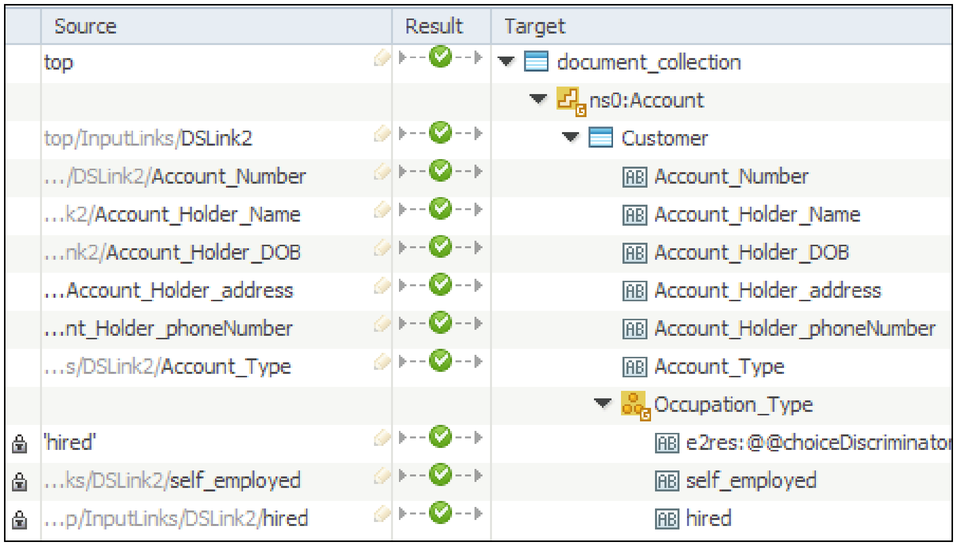
- If the input data is present only for the element self_employed, the constant value self_employed must be mapped to the target item @@choiceDiscriminator, as shown in Figure 8-69 on page 261. The constant value must be the same as the name of the choice item.



Source	Result	Target
top	✓	document_collection
		ns0:Account
top/InputLinks/DSLink2	✓	Customer
.../DSLink2/Account_Number	✓	Account_Number
...k2/Account_Holder_Name	✓	Account_Holder_Name
...nk2/Account_Holder_DOB	✓	Account_Holder_DOB
...Account_Holder_address	✓	Account_Holder_address
...nt_Holder_phoneNumber	✓	Account_Holder_phoneNumber
...s/DSLink2/Account_Type	✓	Account_Type
		Occupation_Type
'self_employed'	✓	e2res:@@choiceDiscriminator
...ks/DSLink2/self_employed	✓	self_employed
...p/InputLinks/DSLink2/hired	✓	hired

Figure 8-69 The mapping rules while mapping @@choiceDiscriminator item

- If the input data is present only for the element hired, the constant value hired must be mapped to the target item @@choiceDiscriminator, as shown in Figure 8-70.



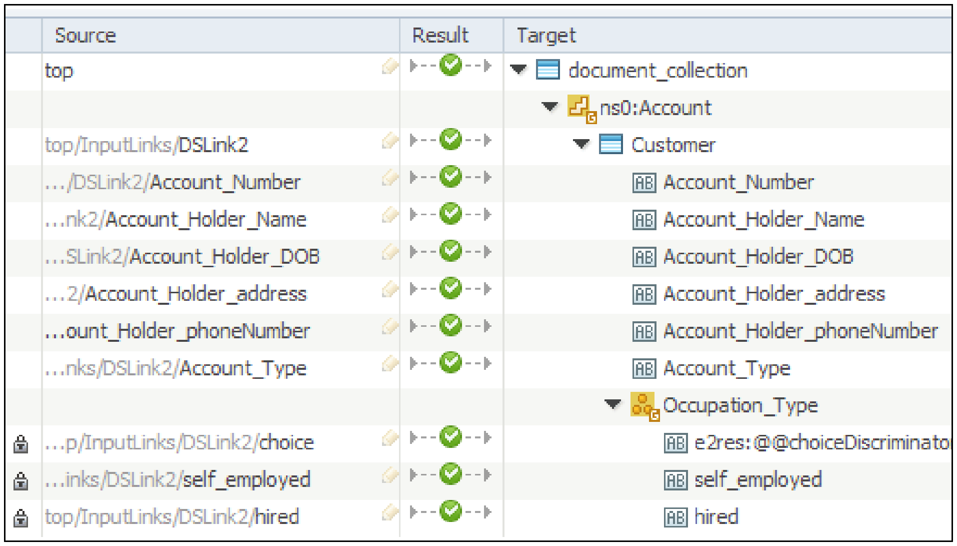
Source	Result	Target
top	✓	document_collection
		ns0:Account
top/InputLinks/DSLink2	✓	Customer
.../DSLink2/Account_Number	✓	Account_Number
...k2/Account_Holder_Name	✓	Account_Holder_Name
...nk2/Account_Holder_DOB	✓	Account_Holder_DOB
...Account_Holder_address	✓	Account_Holder_address
...nt_Holder_phoneNumber	✓	Account_Holder_phoneNumber
...s/DSLink2/Account_Type	✓	Account_Type
		Occupation_Type
'hired'	✓	e2res:@@choiceDiscriminator
...ks/DSLink2/self_employed	✓	self_employed
...p/InputLinks/DSLink2/hired	✓	hired

Figure 8-70 A constant value mapped

- Sometimes the incoming data can have both elements, that is, one row is self_employed and the other row is hired. The constant value cannot be mapped to the item @@choiceDiscriminator, because, during design time, you cannot verify which element is present in the input. In this case, a Transformer stage is used before the XML Stage. In the Transformer stage, a new column is added called *choice*. The value of this column is determined by using “If...else” conditioning. The following example shows the condition in the Transformer stage:

```
If IsNotNull(DSLink2.self_employed) Then "self_employed" Else
"hired"
```

The column choice is then mapped to the @@choiceDiscriminator item in the Mappings tab of the Composer step, as shown in Figure 8-71.



Source	Result	Target
top	✓	document_collection
		ns0:Account
top/InputLinks/DSLink2	✓	Customer
.../DSLink2/Account_Number	✓	Account_Number
...nk2/Account_Holder_Name	✓	Account_Holder_Name
...SLink2/Account_Holder_DOB	✓	Account_Holder_DOB
...2/Account_Holder_address	✓	Account_Holder_address
...ount_Holder_phoneNumber	✓	Account_Holder_phoneNumber
...nks/DSLink2/Account_Type	✓	Account_Type
		Occupation_Type
...p/InputLinks/DSLink2/choice	✓	e2res:@@choiceDiscriminato
...inks/DSLink2/self_employed	✓	self_employed
top/InputLinks/DSLink2/hired	✓	hired

Figure 8-71 The column choice mapped to the @@choiceDiscriminator item

At run time, if the input row contains data for the self_employed element, choice contains the value self_employed. The Composer step knows which element must be present in the output XML file.

- XML schemas might use choice elements with the `tns` extension. This extension means that the self_employed element is present in the namespace that is represented by `tns`. When mapping the value to the @@choiceDiscriminator, you need to take care of the namespace, too. Therefore, if the input has the data for the self_employed element and the namespace is `http://check123`, you map the constant value “`{http://check123}self_employed`” to the @@choiceDiscriminator item.

8.5.2 Creating the XML file for a schema with type derivatives

The bank uses a schema in which an element is derived from another element. Figure 8-72 shows the XML schema and its representation in the assembly. The element office_address in the schema is derived from the complex type home_address. Because office_address is derived from home_address, the element address can also be the office_address type. Therefore, the item @@type is added under the address group, so that at run time, the @@type item can determine which structure to use.

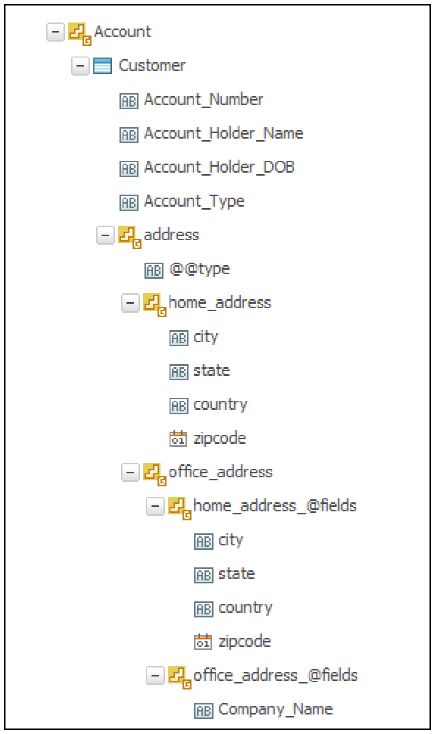
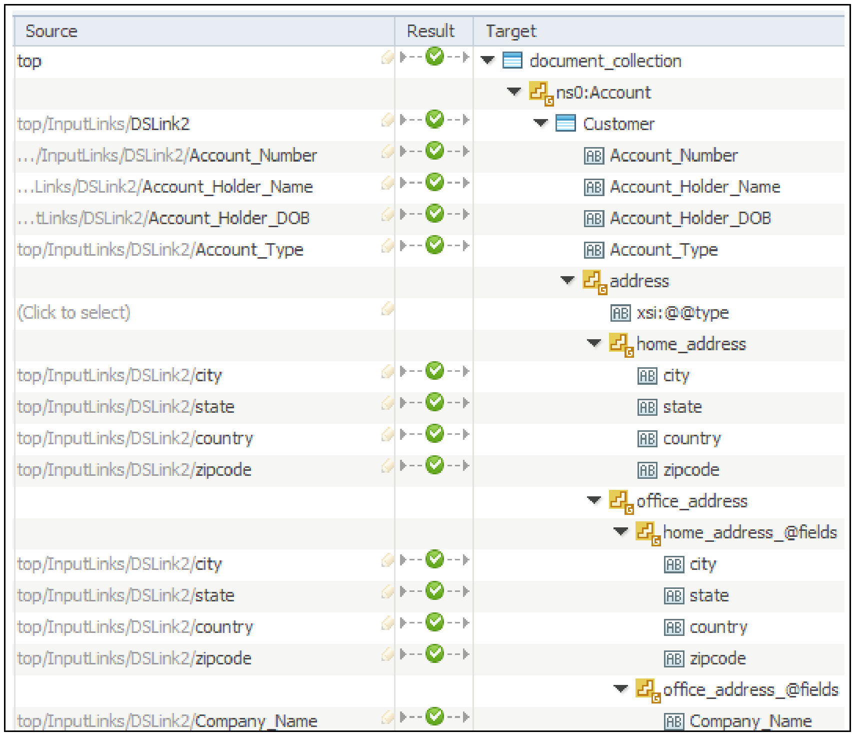


Figure 8-72 The item @@type for handling type derivatives

The following mapping rules apply when you create an XML file for a schema with @@type:

- If nothing is mapped to the @@type item, the output XML file contains only the elements of home_address. As you can see in Figure 8-73 on page 264, the mapping occurred for home and office address fields, but because nothing is mapped to @@type, the output contains only home_address.



Source	Result	Target
top	✓	document_collection
		ns0:Account
top/InputLinks/DSLink2	✓	Customer
.../InputLinks/DSLink2/Account_Number	✓	Account_Number
...Links/DSLink2/Account_Holder_Name	✓	Account_Holder_Name
...tLinks/DSLink2/Account_Holder_DOB	✓	Account_Holder_DOB
top/InputLinks/DSLink2/Account_Type	✓	Account_Type
		address
(Click to select)		xsi:@@type
		home_address
top/InputLinks/DSLink2/city	✓	city
top/InputLinks/DSLink2/state	✓	state
top/InputLinks/DSLink2/country	✓	country
top/InputLinks/DSLink2/zipcode	✓	zipcode
		office_address
		home_address_@fields
top/InputLinks/DSLink2/city	✓	city
top/InputLinks/DSLink2/state	✓	state
top/InputLinks/DSLink2/country	✓	country
top/InputLinks/DSLink2/zipcode	✓	zipcode
		office_address_@fields
top/InputLinks/DSLink2/Company_Name	✓	Company_Name

Figure 8-73 Nothing is mapped to @@type item

- If the constant value home_address is mapped to the @@type, home_address is in the output XML file.
- If the constant value office_address is mapped to the @@type, office_address is in the output XML file.
- In case the input data can have either office or home address, a Transformer stage can be used before the XML Stage. In the Transformer stage, the value for a new column type can be determined by using "If...else" conditioning. The column type can then be mapped to the @@type item in the Composer step.

9

Transforming XML documents

The XML Stage can read, write, and transform XML files. The Parser step is used for reading XML files. The Composer step is used for writing or creating XML files. Sometimes, you are required to perform certain operations on the XML data, for example, sorting the XML file or filtering the XML content to remove unwanted data. The XML Stage provides steps to transform XML files. We describe these steps in detail in this chapter.

We use the same XML structure to explain each step. The XML file contains the details of a bank account. Figure 9-1 on page 266 shows the XML structure in the assembly.

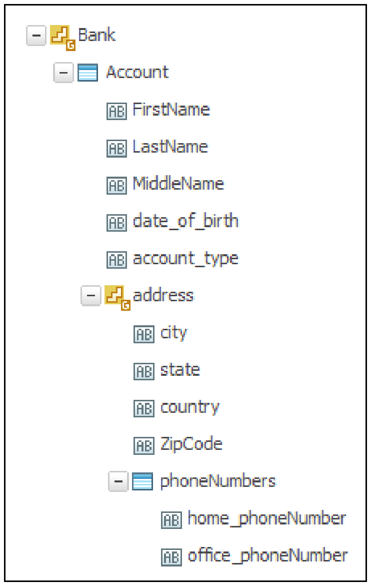

Bank
Account
FirstName
LastName
MiddleName
date_of_birth
account_type
address
city
state
country
ZipCode
phoneNumbers
home_phoneNumber
office_phoneNumber

Figure 9-1 The XML schema structure in the Assembly

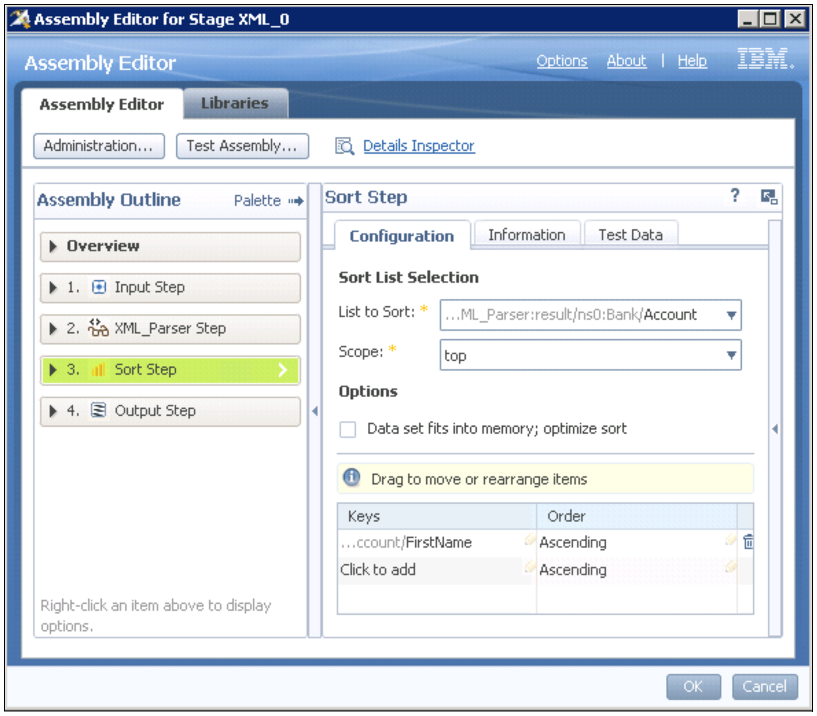
9.1 The Sort step

The Sort step is used to sort an item under a list in ascending or descending order. The Sort step works in a similar manner to the Sort stage in DataStage. The Sort stage sorts the incoming data in a link based on a key. Similarly, the Sort step sorts a list in an XML file based on an item selected as the key.

For the XML schema structure that is shown in Figure 9-1 on page 266, the Account information needs to be sorted in ascending order based on the FirstName field. To sort the XML file, first the XML file needs to be read by using the Parser step. The Sort step must be added after the Parser step to sort the content of the XML file. Follow these steps to configure the Sort step that is depicted in Figure 9-2 on page 268:

1. The List to Sort specifies the list that needs to be sorted in the XML file. In the List to Sort drop-down list, select the list **Account**.
2. The Scope in the Sort step defines the scope of the sort function. The elements are sorted within the list that is selected as the Scope. The Scope also determines the location of the output of the Sort step. The Scope must always be the parent node of the list that is selected in the List to Sort field. In this scenario, because the Account is selected as the List to Sort, only `top` can be selected as the Scope. Select **top**. The output of the Sort step is now under the list `top`. All the FirstName items that are under the `top` scope are sorted.
3. The XML file must be sorted on the FirstName item. Therefore, for Keys, select **FirstName** from the drop-down list.
4. You can select the sort order in the drop-down list under Order. The sorting options are to sort in Ascending or Descending order. Select **Ascending**.

The configuration of the Sort Step is shown in Figure 9-2.



Sort List Selection

List to Sort: * ...ML_Parser:result/ns0:Bank/Account

Scope: * top

Options

Data set fits into memory; optimize sort

Drag to move or rearrange items

Keys	Order
...ccount/FirstName	Ascending
Click to add	Ascending

Figure 9-2 The configuration of the Sort Step

Because we selected `top` as the Scope, the output of the Sort step is under `top`, as shown in Figure 9-3 on page 269. The output of the Sort step is in a group called Sort:result. The output shows the list Account sorted in ascending order based on the FirstName field. This output can then be mapped to the Composer step to create the sorted XML file.

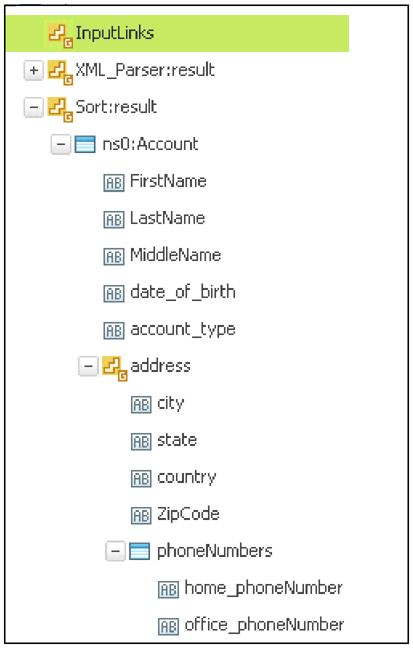

InputLinks
XML_Parser:result
Sort:result
ns0:Account
FirstName
LastName
MiddleName
date_of_birth
account_type
address
city
state
country
ZipCode
phoneNumbers
home_phoneNumber
office_phoneNumber

Figure 9-3 The output of the Sort step

Sorting has special considerations. See 10.5.1, "Sort step" on page 318 for the details.

9.2 The Aggregate step

The *Aggregate step* performs hierarchical aggregation on an item in a list. It is similar to the Aggregate stage in the DataStage Designer Canvas. The Aggregate stage groups data for an input link and performs count and sum functions. Similarly, the Aggregate step in XML Stage groups data within a list and performs functions, such as Count and Concat.

A customer at our scenario bank has either a loan account or a deposit account. You can use the Aggregate step to count the number of customers based on the type of account. To count the number of accounts of a particular account type in the XML file, first the XML file needs to be read by using the Parser step. The Aggregate step must to be added after the Parser step to aggregate the hierarchical data.

Follow these steps to configure the Aggregate step, as depicted in Figure 9-6 on page 272:

1. The List to Aggregate field specifies the list whose items need to be aggregated in the XML file. In the current scenario, the aggregation can be performed by using any of the items. We use the item called FirstName. In the List to Aggregate drop-down list, select the list that contains the FirstName field. Because FirstName is under the list Account, select **Account** in the List to Aggregate drop-down list.
2. The *Scope* field in the Aggregate step defines the scope of the Aggregate function. The elements are aggregated within the list that is selected as the scope. The scope also determines the location of the output of the Aggregate step. The scope must always be the parent node of the list that is selected in the List to Aggregate field. In this scenario, because Account is selected for the List to Aggregate, only `top` can be selected as the scope. Select **top**. Because the Scope is set as `top` and the item selected for aggregation is FirstName, the Aggregate step checks for all the FirstName fields within the `top` list and then aggregates them. Therefore, this step looks for FirstName in the input file and aggregates it.
3. In the Items columns, you select the item that needs to be grouped. From the drop-down list, select the **FirstName** field.
4. In the Functions column, select the function that you want to perform on the item FirstName. Multiple functions are available for each of the data types. In the current scenario, because we need to identify the number of accounts, we select the **Count** function from the drop-down list.
5. The Aggregate step is configured. The output of the Aggregate step is directly under `top`, which we selected as the scope. The output is in a group called Aggregate:result, as shown in Figure 9-4. All the FirstName fields in the XML file are counted without considering the account types.

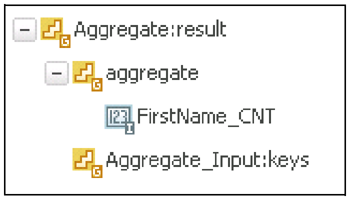


Figure 9-4 The output of the Aggregate step

6. Next, because the aggregation is based on the account_type field, we select **account_type** from in the Aggregation Keys drop-down list. The output of the Aggregate step is a list called Aggregate:result, which is under `top`, as shown in Figure 9-5 on page 271. The Aggregate:result is a list, because one output exists for each account_type value. The output also contains the

account_type field. For example, if an XML file includes 10 accounts, and four accounts are loan account_type accounts and the rest of the accounts are deposit account_type accounts, the output of the Aggregate step looks like Example 9-1.

Example 9-1 The output of the Aggregate step

```
4,loan
6,deposit
```

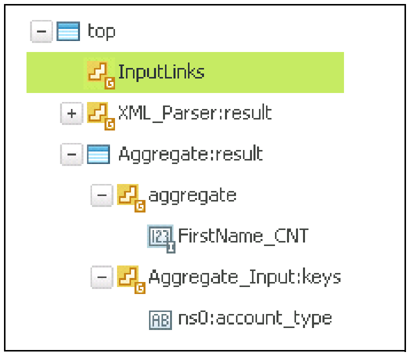
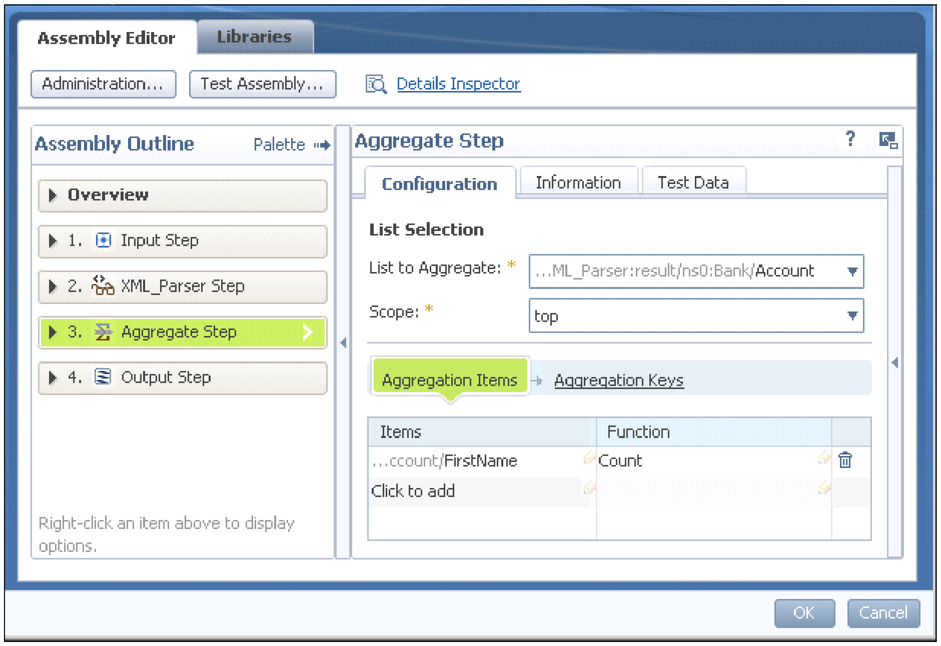


Figure 9-5 The output of the Aggregate step when a key value is specified

The configuration of the Aggregate Step is shown in Figure 9-6 on page 272.



List Selection

List to Aggregate: * ...ML_Parser:result/ns0:Bank/Account

Scope: * top

Items	Function
...ccount/FirstName	Count
Click to add	

Figure 9-6 The configuration of the Aggregate Step

You can map the output of the Aggregate step to various columns of a Sequential file by using the Output step.

Appendix B, “Transformer functions” on page 369 lists the available functions in the Aggregate step.

9.3 The Switch step

The Switch step is used to filter the XML file and classify the items into one or more targets. The filtering is based on constraints that you specify. Each constraint is associated with an output list. A default list contains the items that do not satisfy any of the constraints.

A customer at our scenario bank has either a loan account or a deposit account. You can use the Switch step to write the account information for each type into a separate output file. The XML file must be read by using the Parser step. You must add the Switch step after the Parser step to filter the items in the XML file. Follow these steps to configure the Switch step, as depicted in Figure 9-9 on page 275:

1. The List to Categorize field is the list that contains the item that is used to specify the constraint. The item account_type is under the list Account. Therefore, select **Account** from the drop-down list for the List to Categorize field.
2. The Scope defines the scope of the Switch step. The elements are filtered within the list that is selected as the scope. The scope also determines the location of the output of the Switch step. The scope must always be the parent node of the list that is selected in the List to Categorize field. In this scenario, because we selected Account for the List to Categorize field, only `top` can be selected as the scope. Select **top**. Because the scope is set to `top` and the item for filtering is account_type, the Switch step checks for all the account types within the `top` list and then filters the list. Therefore, the entire input file is filtered.
3. To specify the constraint for filtering, add a target in the Switch step. To add a target, follow these steps:

   a. Click **Add Target**. The New Target Constraint window opens, as shown in Figure 9-7.

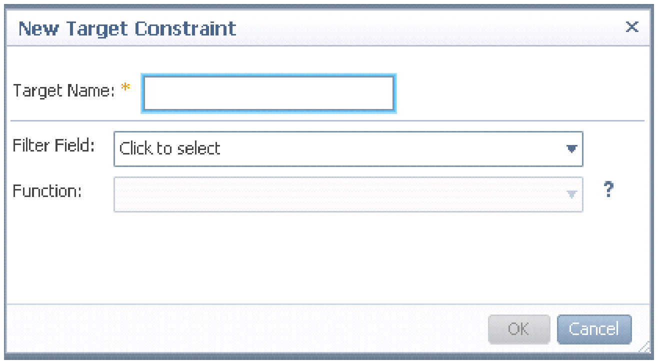


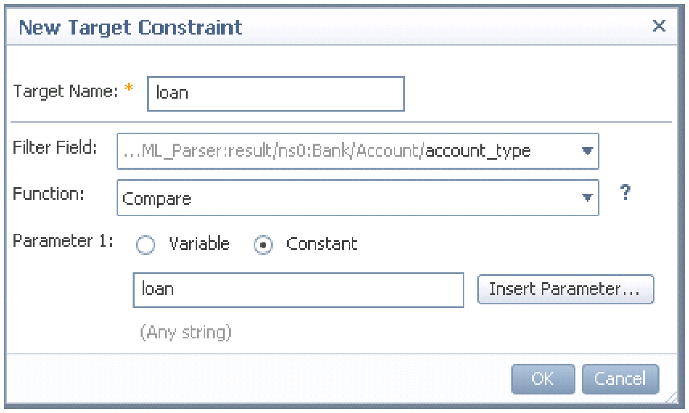
Figure 9-7 The New Target Constraint window

   b. Type `loan` for the Target Name. The Target Name is the name of the constraint, which also is the name of the output list.

   c. The Filter Field specifies the item on which the constraint is specified. Select **account_type** from the drop-down list.

   d. The Switch step provides multiple functions based on the type of filtering that you want to perform. Select **Compare** in the drop-down list for Function.

   e. The account_type can be either loan or deposit. Therefore, either one of these constant values can be specified as the constraint. For Parameter 1, select **Constant** and type `loan` to specify the string in the text box, as shown in Figure 9-8.


New Target Constraint

Target Name: * loan

Filter Field: ...ML_Parser:result/ns0:Bank/Account/account_type

Function: Compare

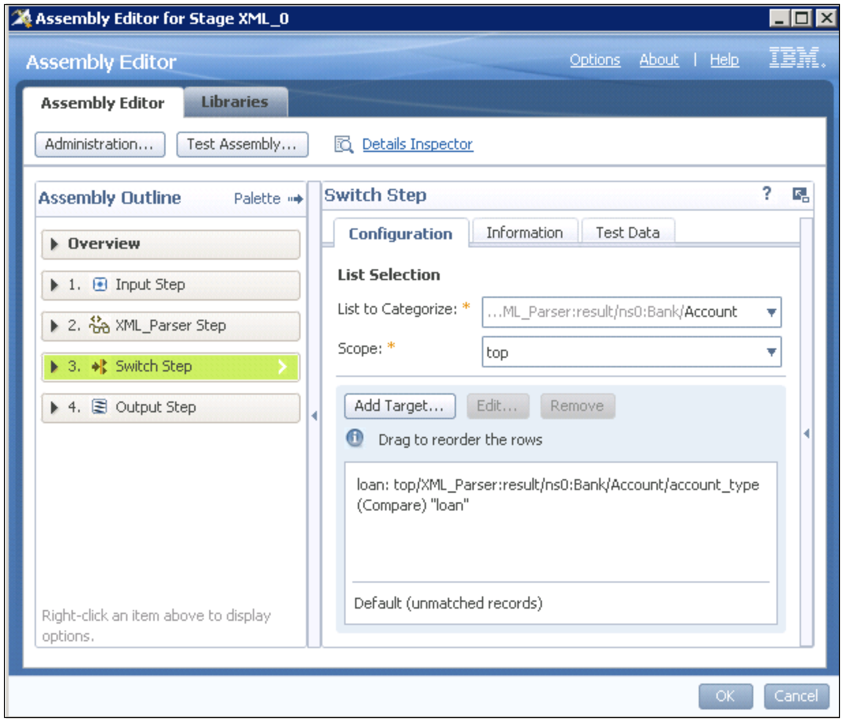
Parameter 1: Variable [✓ Constant]

loan

(Any string)

Figure 9-8 The configuration of the New Target Constraint window

Figure 9-9 on page 275 shows the configuration of the Switch Step.



List Selection

List to Categorize: * ...ML_Parser:result/ns0:Bank/Account

Scope: * top

loan: top/XML_Parser:result/ns0:Bank/Account/account_type (Compare) "loan"

Default (unmatched records)

Figure 9-9 The configuration of the Switch Step

The output of the Switch step is in a group called Switch:filtered, as shown in Figure 9-10 on page 276. Two lists are under this group: loan and default. The loan list contains all the items that satisfy the constraint for the target loan. The default list contains all the items that do not satisfy the loan constraint.

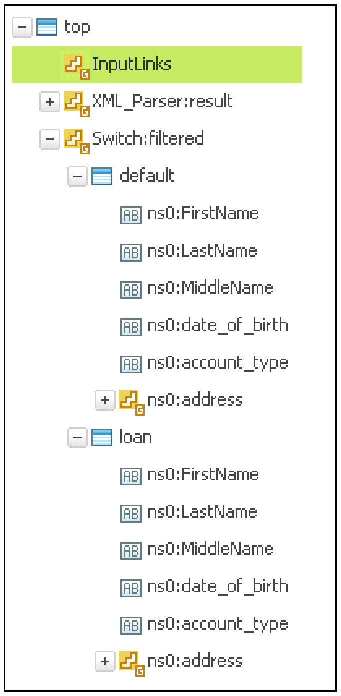

top
InputLinks
XML_Parser:result
Switch:filtered
default
ns0:FirstName
ns0:LastName
ns0:MiddleName
ns0:date_of_birth
ns0:account_type
ns0:address
loan
ns0:FirstName
ns0:LastName
ns0:MiddleName
ns0:date_of_birth
ns0:account_type
ns0:address

Figure 9-10 The output of the Switch step

Each of the lists under Switch:filtered can be mapped to a separate output link by using the Output step. Therefore, all the account information for the loan account type is in one output file. All the account information for the deposit account type is in a separate output file (the output file to which the default list is mapped).

See Appendix B, "Transformer functions" on page 369 for a list of functions that are available in the Switch step.

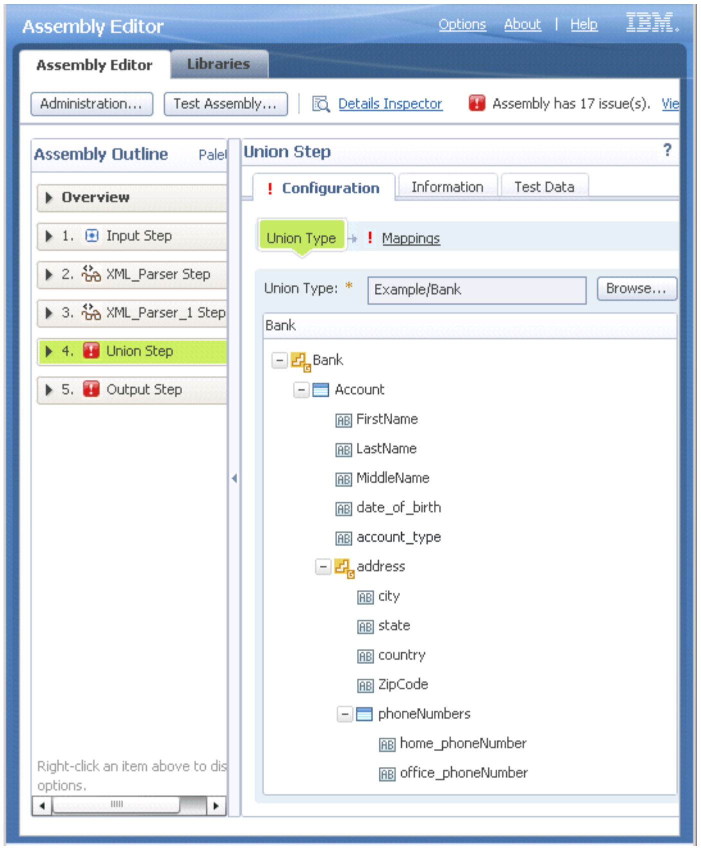
9.4 The Union step

The Union step is used to combine two lists into a single output list. Two XML files can be combined into a single file by using the Union step.

The XML files can be read by using two Parser steps. The Union step must be used after the Parser steps to combine the two files.

Follow these steps to configure the Union step:

1. Specify the target schema in the Union Type tab. The target schema defines the structure of the output of the Union step. The Union Type tab is similar to the Document Root tab of the Parser and Composer steps. To specify the Union Type, click **Browse** to select the schema element from the Schema Library Manager. When you select the schema, it is defined in the Union Type tab, as shown in Figure 9-11.



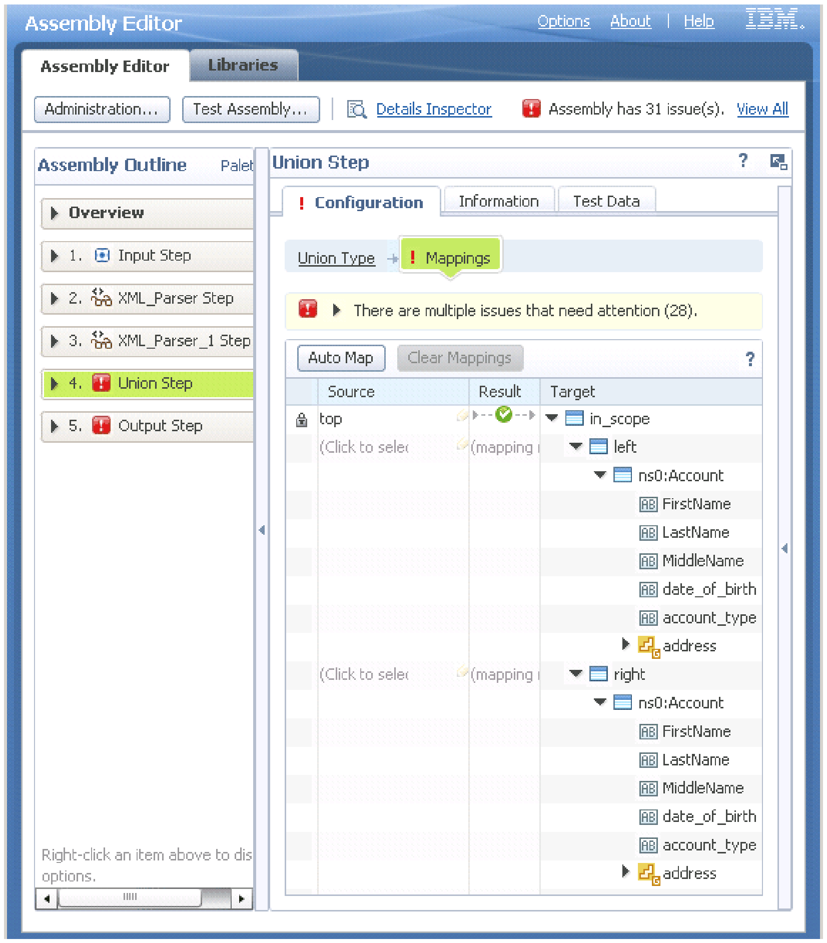
Union Type: * Example/Bank

Bank
- Bank
  - Account
    - FirstName
    - LastName
    - MiddleName
    - date_of_birth
    - account_type
    - address
      - city
      - state
      - country
      - ZipCode
      - phoneNumbers
        - home_phoneNumber
        - office_phoneNumber

Figure 9-11 The Union Type tab of the Union Step

2. The Mappings tab is similar to the Mappings tab in the Composer step. You can use the Mappings table to map the output of the Parser step to the target structure in the Union step.

   This Mappings table, as shown in Figure 9-12 on page 278, contains three columns: Source, Result, and Target. The Source column shows the columns that need to be mapped to the target items. The Result column shows the status of the current mapping. The Target column contains two lists: the left list and the right list. Each list shows the items of the target structure. The Target structure is the XML schema that is defined in the Union Type tab of the Union Step.



There are multiple issues that need attention (28).

Source	Result	Target
top	✓	in_scope
(Click to selec	(mapping	left
		ns0:Account
		FirstName
		LastName
		MiddleName
		date_of_birth
		account_type
		address
(Click to selec	(mapping	right
		ns0:Account
		FirstName
		LastName
		MiddleName
		date_of_birth
		account_type
		address

Figure 9-12 The Mappings table in the Union Step

3. The left list is mapped to the output of the first Parser step, and the right list is mapped to the output of the second Parser step, as shown in Figure 9-13 on page 279.

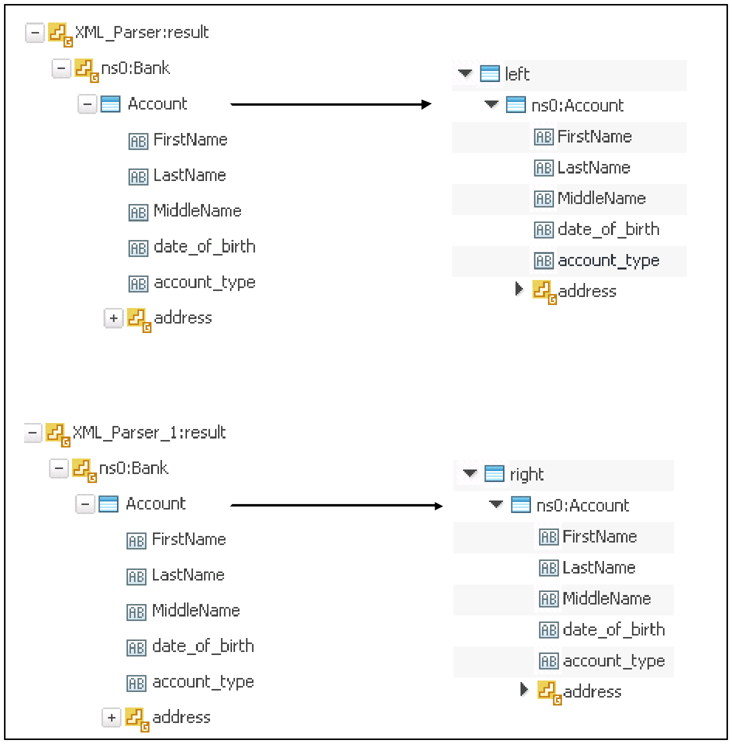


Figure 9-13 The mapping between the Parser steps and the Union step

As shown in Figure 9-13, a source list needs to be mapped to the left and right lists in the target. The Account lists under the first and second Parser steps must be mapped to the Account fields under the left and right lists. To create lists on the source side, you can use the Group to List projection. Follow these steps to create the lists on the source side:

1. Go to the Output tab of the First Parser step. Right-click the **XML_Parser:result** group, as shown in Figure 9-14 on page 280.

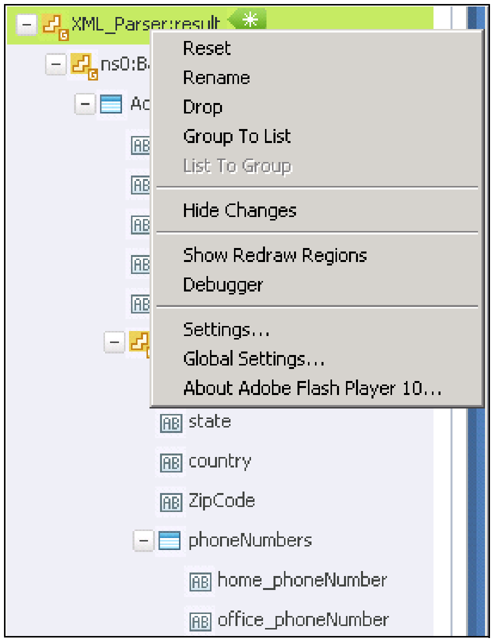


Figure 9-14 The Group To List projection option

2. Select the **Group To List** projection. The group is converted to a list, as shown in Figure 9-15 on page 281.

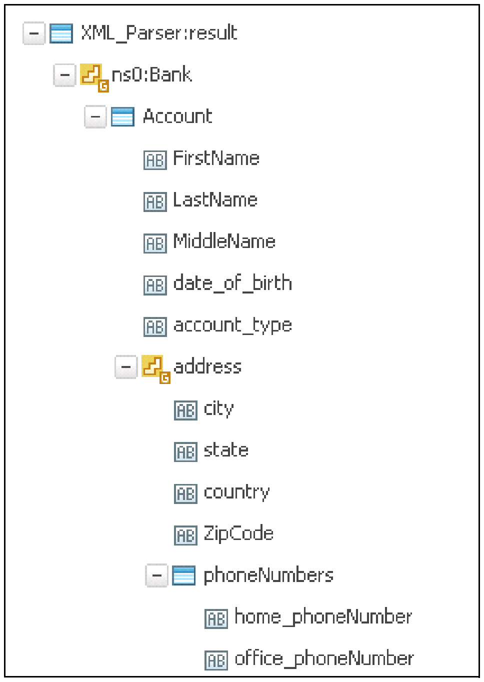


Figure 9-15 XML_Parser:result as a list

3. Go to the Output tab of the Second Parser step. Right-click the group **XML_Parser_1:result** and select **Group To List**. The group is converted to a list, as shown in Figure 9-16 on page 282.

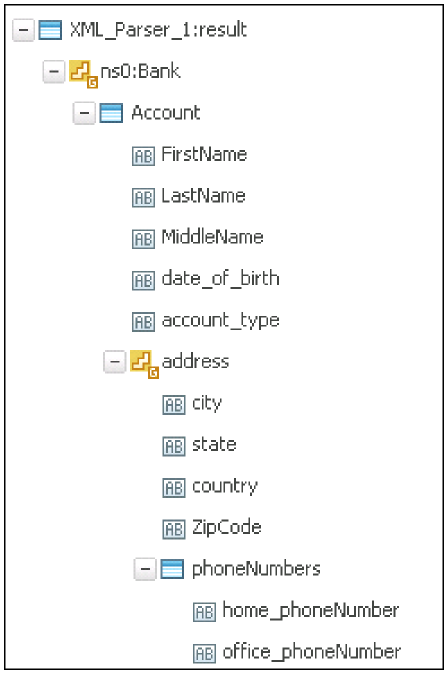
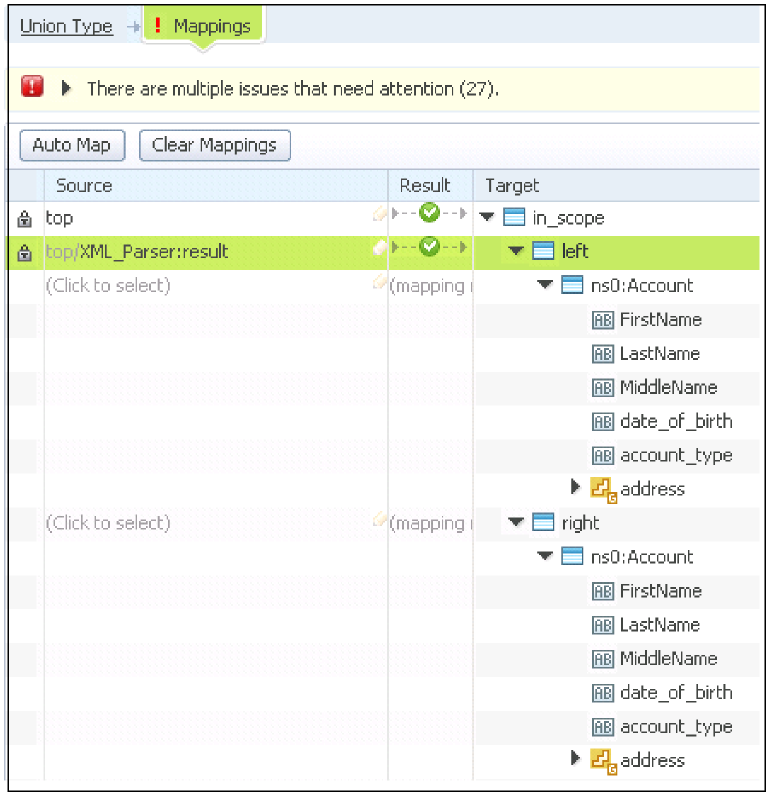


Figure 9-16 XML_Parser_1:result as a list

Follow these steps to map the data from the Parser steps to the Union step:

1. Double-click the Source column for the target list **left**. You see a drop-down list, which is the suggestion list. The *suggestion list* shows all available source lists that can be mapped to the target list. The absolute path, `top/XML_Parser:result`, is shown in the suggestion list.
2. Select the source list **top/XML_Parser:result** from the suggestion list. When you select `top/XML_Parser:result`, the mapping occurs for this row. A green circle with a white check mark shows in the Result column, as seen in Figure 9-17 on page 283.



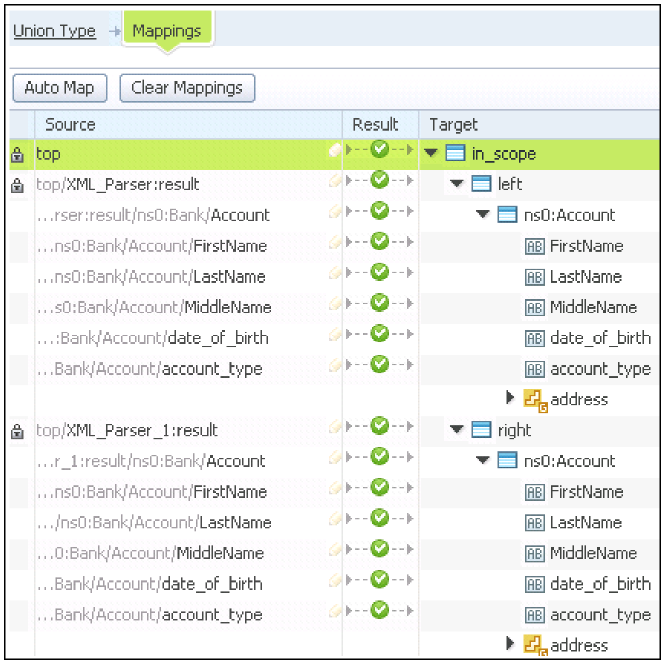
There are multiple issues that need attention (27).

Source	Result	Target
top		in_scope
top/XML_Parser:result		left
(Click to select)	(mapping	ns0:Account
		FirstName
		LastName
		MiddleName
		date_of_birth
		account_type
		address
(Click to select)	(mapping	right
		ns0:Account
		FirstName
		LastName
		MiddleName
		date_of_birth
		account_type
		address

Figure 9-17 Mapping the Source to the Target

3. Then, map **top/XM_Parser:result /Bank/Account** to the list **Account**, which is under the `left` list.
4. After mapping the target list Account, you can map the columns under it. To map each of the target columns, select the corresponding row in the Source column. From the suggestion list, select the appropriate item to map. The item recommendations in the suggestion list are based on the name and data type of the Target column.
5. Follow steps 1 - 4 to map XML_Parser_1:result and its child items to the `right` list.

Figure 9-18 on page 284 shows the Mappings table after mapping all the columns.



Source	Result	Target
top	✓	in_scope
top/XML_Parser:result	✓	left
...rser:result/ns0:Bank/Account	✓	ns0:Account
...ns0:Bank/Account/FirstName	✓	FirstName
...ns0:Bank/Account/LastName	✓	LastName
...s0:Bank/Account/MiddleName	✓	MiddleName
...:Bank/Account/date_of_birth	✓	date_of_birth
...Bank/Account/account_type	✓	account_type
		address
top/XML_Parser_1:result	✓	right
...r_1:result/ns0:Bank/Account	✓	ns0:Account
...ns0:Bank/Account/FirstName	✓	FirstName
.../ns0:Bank/Account/LastName	✓	LastName
...0:Bank/Account/MiddleName	✓	MiddleName
...Bank/Account/date_of_birth	✓	date_of_birth
...Bank/Account/account_type	✓	account_type
		address

Figure 9-18 The Mappings table

The Union step is configured. The output of the Union step is in a list called Union_result, as shown in Figure 9-19. The structure of the Union:result list is the same structure that is defined in the Union Type. The output of the Union step can be mapped to the Composer step to recreate the XML file. Therefore, two XML files are combined into a single file.

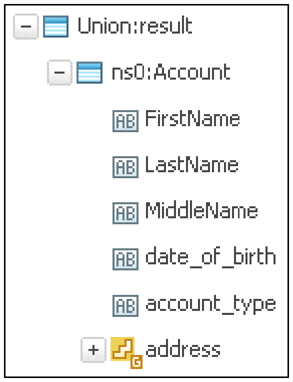


Figure 9-19 The output of the Union step

9.5 The V-Pivot step

The V-Pivot step is used to transform the values in a column into a single row.

To demonstrate the use of the V-Pivot step, the Account schema is slightly modified. The new schema representation is shown in Figure 9-20. The address is a list that can contain the account holder home address and the account holder office address. The address_type field is used to distinguish the addresses. It contains `office` for office address and `home` for home address. When this structure is parsed, it creates two output rows: one row for home address and one row for office address. To map all the account holder information and the address information in a single output row, you must use the V-Pivot step after the Parser step.

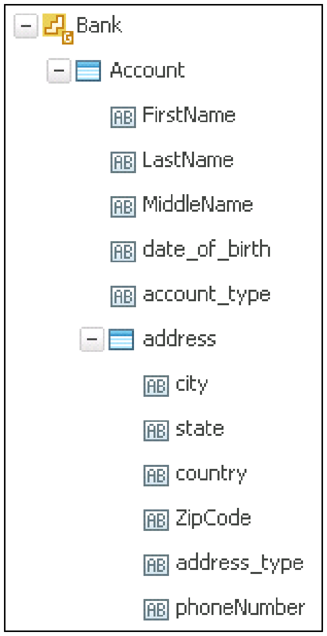


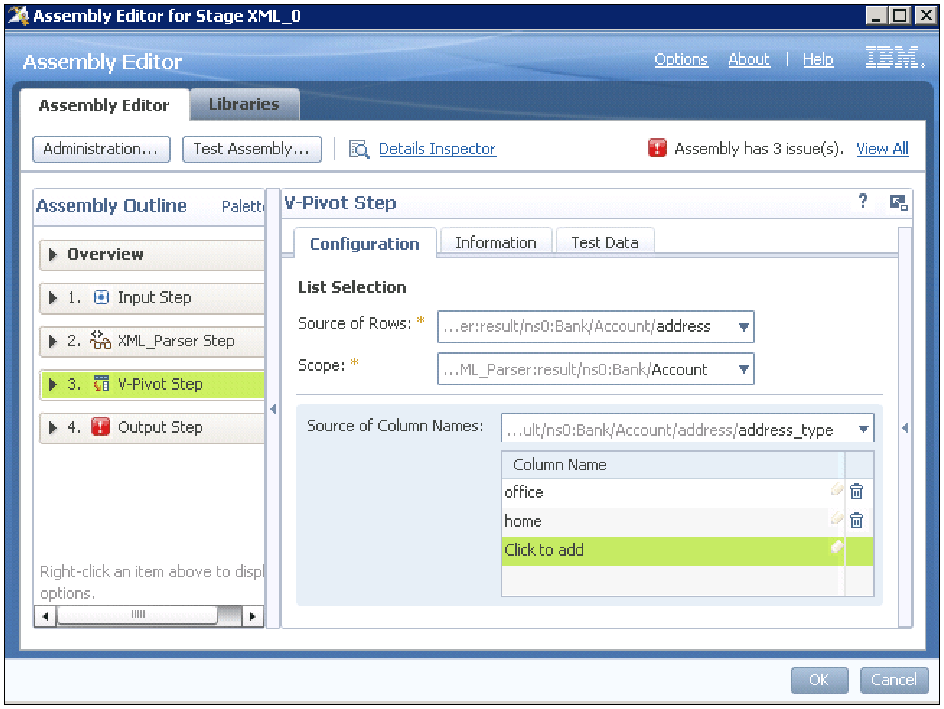
Figure 9-20 XML schema for V-Pivot example

Follow these steps to configure the V-Pivot step that is depicted in Figure 9-21 on page 287:

1. The Source of Rows field specifies the list that contains the field on which the pivot is performed. In this scenario, the address_type field is unique for each address list. Therefore, the pivot is performed on the address_type field. Because the address_type field is under the list address, select **address** in the drop-down list for the Source of Rows field.

2. The Scope in the V-Pivot step defines the scope of the V-Pivot function. The scope also determines the location of the output of the V-Pivot step. The scope must always be the parent node of the list that is selected in the Source of Rows field. For the V-Pivot step to run successfully, the pivoting field must contain unique values. The address_type field contains two values: office and home. These values are repeated for every Account. But, these values are unique for a single account. Therefore, select **Account** for the Scope to ensure that the pivoting is performed for one account at a time rather than for all the accounts. The Scope in the V-Pivot step must be selected to ensure that the values of the pivoting field are unique.
3. The Source of Column Names field specifies the pivoting field. Select **address_type** from the drop-down list.
4. The Column Name field specifies the unique values that can be in the pivoting column, for example, `address_type`. Because the address_type can be `office` and `home`, enter these values in individual rows. The Column Name field can accept alphabetical characters, numerals, and a few special characters, such as the period (.), underscore (_), and hyphen (-). The column names must start with either an alphabetical character or underscore. The column names cannot contain special characters, such as the colon (:) and the At sign (@).

The V-Pivot Step is now configured, as shown in Figure 9-21 on page 287.



List Selection

Source of Rows: * ...er:result/ns0:Bank/Account/address

Scope: * ...ML_Parser:result/ns0:Bank/Account

Source of Column Names: ...ult/ns0:Bank/Account/address/address_type

Column Name
office
home
Click to add

Figure 9-21 The configuration of the V-Pivot Step

The output of the V-Pivot step is in a group called V-Pivot:result, which contains two nodes: office and home. These names are based on the values that are specified for the Column Names. The `home` node holds the house address, and the `office` node holds the office address. The output of the V-Pivot step is under the Account list, because the Scope is set to Account. See Figure 9-22 on page 288.

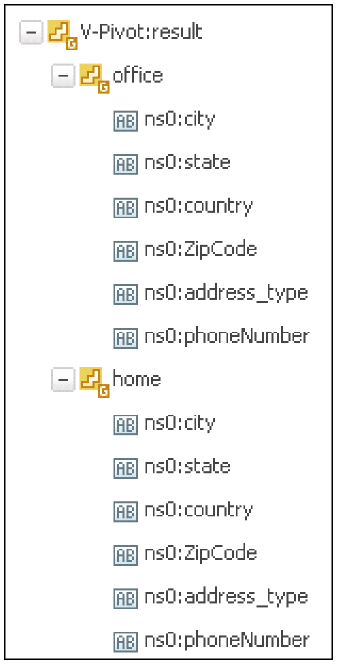


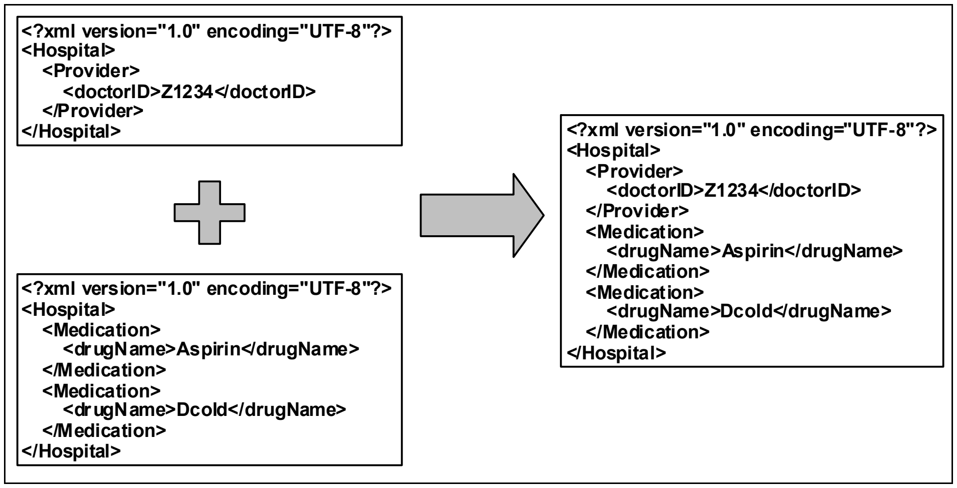
Figure 9-22 The output of the V-Pivot step

You can map the output of the V-Pivot step to a sequential file by using the Output step.

9.6 The OrderJoin step

The OrderJoin step is used to join two items in a list based on their positions.

A hospital stores its patient chart information in two XML files. One XML file stores information about the doctor who treated the patients. The other XML file stores the medications that are recommended by the doctor. We must merge both sets of details into one XML file, as shown in Figure 9-23 on page 289. We are not required to combine the exact medication that is prescribed with the doctor who prescribed it. The information merely needs to be merged so that the doctor information is followed by all the prescribed medicines. The OrderJoin step can be used to create this XML file.



```
<?xml version="1.0" encoding="UTF-8"?>
<Hospital>
   <Provider>
      <doctorID>Z1234</doctorID>
   </Provider>
</Hospital>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<Hospital>
   <Medication>
      <drugName>Aspirin</drugName>
   </Medication>
   <Medication>
      <drugName>Dcold</drugName>
   </Medication>
</Hospital>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<Hospital>
   <Provider>
      <doctorID>Z1234</doctorID>
   </Provider>
   <Medication>
      <drugName>Aspirin</drugName>
   </Medication>
   <Medication>
      <drugName>Dcold</drugName>
   </Medication>
</Hospital>
```

Figure 9-23 The output of the OrderJoin step

In this scenario, two Parser steps are required to read each of the XML files. The XML schema structure of each of the XML files is shown in Figure 9-24. The OrderJoin step is added after the Parser steps.

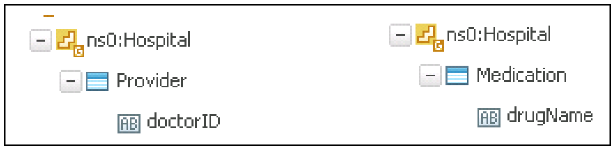
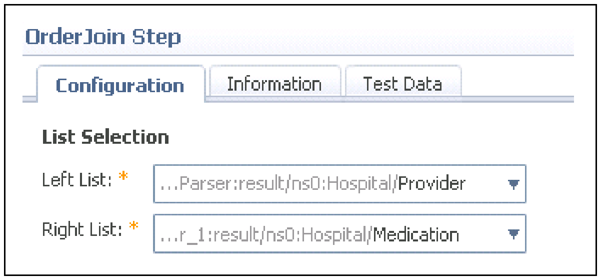


Figure 9-24 The schema structures of the input XML files

To configure the OrderJoin step, you need to specify the `Left` list and the `Right` list:

1. In the `Left` list, select the list whose content you need to list first. Select **Provider** from the drop-down list, because you need to list the doctor information first.
2. In the `Right` list, select **Medication**.

The OrderJoin Step is configured, as shown in Figure 9-25 on page 290.


OrderJoin Step

List Selection

Left List: * ...Parser:result/ns0:Hospital/Provider

Right List: * ...r_1:result/ns0:Hospital/Medication

Figure 9-25 The configuration of the OrderJoin Step

The output of the OrderJoin step is a list called OrderJoin:result. This list contains two groups: the group that you defined as the Left list and the other group that you defined as the Right list. Therefore, as seen in Figure 9-26, OrderJoin:result includes `Provider`, which contains the doctor information, and `Medication`, which contains the prescribed medications. The output of the OrderJoin step can now be mapped to a Composer step to recreate the single XML file.

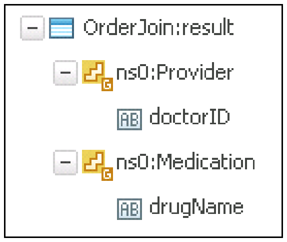


Figure 9-26 The output of the OrderJoin step

In this chapter, you configured the steps that are available in the XML Stage. We described the Parser step in detail in Chapter 7, “Consuming XML documents” on page 147. We described the Composer step, Regroup step, HJoin step, and H-Pivot step in detail in Chapter 8, “Creating and composing XML documents” on page 201.

10

DataStage performance and XML

This chapter focuses on the performance aspects of processing XML documents with InfoSphere DataStage. We begin with information about parallel processing with DataStage, and then extend those principles to processing XML. We also cover good design practices for maximizing performance with parallel jobs, and documenting those patterns to avoid that might lead to poor performance and bad resource utilization. You can obtain more information about the performance tuning of DataStage parallel jobs in the IBM Redbooks publication, *InfoSphere DataStage Parallel Framework Standard Practices*, SG24-7830, which is referenced in “IBM Redbooks” on page 375.

10.1 Partitioning and collecting methods

This section provides an overview of partitioning and collecting methods, and it provides guidelines for appropriate use in job designs. Partitioning parallelism is key for the scalability of DataStage (DS) parallel jobs. *Partitioners* distribute rows of a single link into smaller segments that can be processed independently in parallel. Partitioners exist before any stage that is running in parallel. If the prior stage ran sequentially, a fan-out icon is drawn on the link in the Designer canvas, as shown in Figure 10-1.

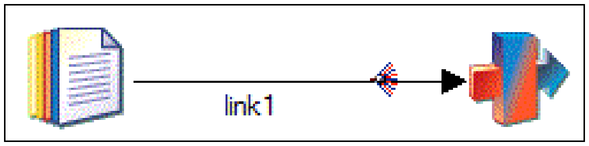


Figure 10-1 Partitioner link marking icon

Collectors combine parallel partitions of a single link for sequential processing. Collectors exist only before stages that run sequentially and when the previous stage is running in parallel. They are indicated by a fan-in icon, as shown in Figure 10-2.

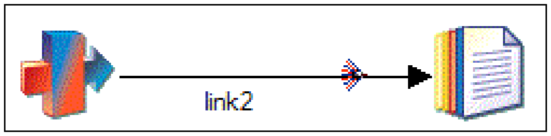


Figure 10-2 Collector link marking icon

Although partitioning allows data to be distributed across multiple processes that run in parallel, it is important that this distribution does not violate business requirements for accurate data processing. For this reason, separate types of partitioning are provided for the parallel job developer. Partitioning methods are separated into two distinct classes:

- Keyless partitioning

  Rows are distributed to each partition without regard to the actual data values.

- Keyed partitioning

  Rows are distributed by examining the data values for one or more key columns, then ensuring that records with the same values in those key columns are assigned to the same partition.

A special method called Auto partitioning is also the default method when new links are created. *Auto partitioning* specifies that the parallel framework attempts to select the appropriate partitioning method at run time, based on the configuration file, datasets, and job design (stage requirements and properties). Auto partitioning selects between keyless (same, round-robin, or entire) and keyed (hash) partitioning methods to produce functionally correct results and, in certain cases, to improve performance. The partitioning method is specified in the input stage properties by using the partitioning option, as shown in Figure 10-3.

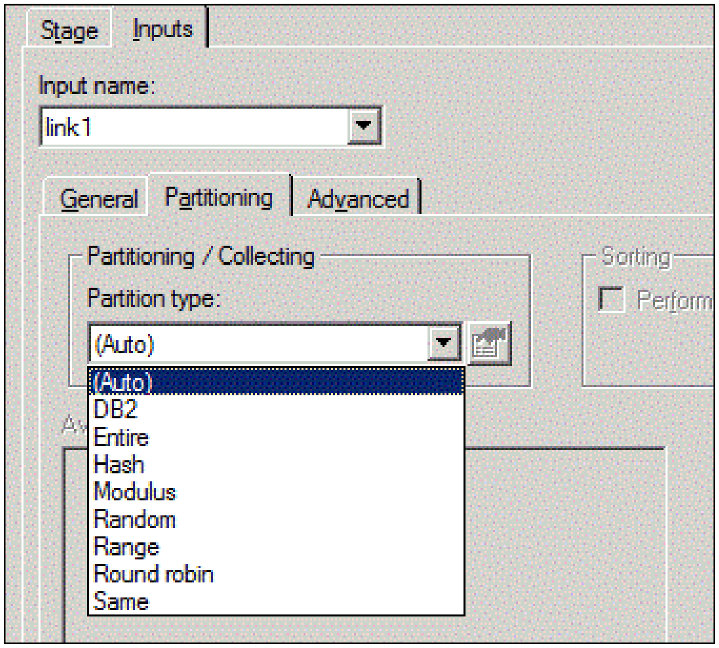


Figure 10-3 Partitioner defined on stage input link properties

Auto partitioning is designed to allow DataStage developers to focus on the business logic aspect of the job design and spend less time on parallel design principles. However, the Auto partitioning method might not be the most efficient from an overall job perspective and in certain (rare) cases can lead to wrong results. The ability of the parallel framework to determine the appropriate partitioning method depends on the information that is available to it. Auto partitioning normally ensures correct results when you use built-in stages. However, because the parallel framework has no visibility into user-specified logic (such as within a Transformer stage), it might be necessary for the user to specify a partitioning method manually. For example, if the logic defined in a Transformer stage is based on a group of related records, a keyed partitioning method must be specified to achieve the correct results.

10.1.1 Keyless partitioning methods

Keyless partitioning methods distribute rows without examining the contents of the data. The following keyless partitioning methods are available:

- Same partitioning

  Same partitioning performs no partitioning to the input dataset. Instead, it retains the partitioning from the output of the upstream stage. Same partitioning does not move data between partitions (or, in the case of a cluster or grid, between servers). Same partitioning is appropriate when trying to preserve the grouping of a previous operation (for example, a parallel sort). It is important to understand the impact of same partitioning in a specific data flow. Because same partitioning does not redistribute existing partitions, the degree of parallelism remains unchanged. If the upstream stage is running sequentially, same partitioning effectively causes a downstream parallel stage to also run sequentially. If you read a parallel dataset with same partitioning, the downstream stage runs with the degree of parallelism used to create the dataset, regardless of the current `$APT_CONFIG_FILE`.

  Important: Because of its restrictive behavior, minimize the use of same partitioning. Use it only when necessary.

- Round-robin partitioning

  Round-robin partitioning evenly distributes rows across partitions in a round-robin assignment, similar to dealing cards one at a time. Round-robin partitioning has a fairly low overhead. Because optimal parallel processing occurs when all partitions have the same workload, round-robin partitioning is useful for redistributing data that is highly skewed (an unequal number of rows in each partition).

- Random partitioning

  Random partitioning evenly distributes rows across partitions. It uses a random assignment. As a result, the order that rows are assigned to a particular partition differs between job runs. Because the random partition number must be calculated, random partitioning has a slightly higher overhead than round-robin partitioning. Although in theory random partitioning is not subject to regular data patterns that might exist in the source data, it is rarely used in functional data flows because it has a slightly larger overhead than round-robin partitioning.

- Entire partitioning

  Entire partitioning distributes a complete copy of the entire dataset to each partition. Entire partitioning is useful for distributing the reference data of a lookup task, which might or might not involve the lookup stage. On clustered and grid implementations, entire partitioning might affect performance, because the complete dataset must be distributed across the network to each node.

10.1.2 Keyed partitioning methods

Keyed partitioning examines the data values in one or more key columns, ensuring that records with the same values in those key columns are assigned to the same partition. Keyed partitioning is used when business rules (for example, remove duplicates) or stage requirements (for example, join) require processing on groups of related records. There are a number of keyed partitioning methods available, which are included in the following list:

- Hash partitioning

  Hash partitioning assigns rows with the same values in one or more key columns to the same partition by using an internal hashing algorithm. If the source data values are evenly distributed in these key columns, and many unique values exist, the resulting partitions are of relatively equal size. In the situation where the number of unique key values is low, you can get *partition skew,* where one partition receives a much larger percentage of rows than other partitions. Skew negatively affects performance. One way of correcting this partition skew is to add an additional key column that substantially increases the number of unique values of the composite key, which results in better distribution across partitions. When using hash partitioning on a composite key (more than one key column), individual key column values have no significance for partition assignment.

- Modulus partitioning

  Modulus partitioning uses a simplified algorithm for assigning related records based on a single integer key column. It performs a modulus operation on the data value by using the number of partitions as the divisor. The remainder is used to assign the value to a specific partition:

  ```
  partition = MOD (key_value / number of partitions)
  ```

  Like hash, the partition size of modulus partitioning is equally distributed as long as the data values in the key column are equally distributed. Because modulus partitioning is simpler and faster than hash, it provides a performance improvement in situations where you have a single integer key column. Modulus partitioning cannot be used for multiple or composite keys, or for non-integer key columns.

- Range partitioning

  As a keyed partitioning method, *range partitioning* assigns rows with the same values in one or more key columns to the same partition. Given enough unique values, range partitioning ensures a balanced workload by assigning an equal number of rows to each partition, unlike hash and modulus partitioning where partition skew depends on the actual data distribution. To achieve this balanced distribution, range partitioning must read the dataset twice: the first time to create a range map file, and the second time to partition the data in a flow by using the range map. A range map file is specific to a certain parallel configuration file.

The read-twice penalty of range partitioning limits its use to specific scenarios, typically where the incoming data values and distribution are consistent over time. In these instances, the range map file can be reused. It is important that if the data distribution changes without recreating the range map, partition balance is skewed, which defeats the intention of range partitioning. Also, if new data values are processed outside of the range of a specific range map, these rows are assigned to either the first or the last partition, depending on the value. In another scenario to avoid, if the incoming dataset is sequential and ordered on the key columns, range partitioning effectively results in sequential processing.

- DB2 partitioning

  The DB2/UDB Enterprise Stage (or EE Stage) matches the internal database partitioning of the source or target DB2 Enterprise Server Edition with Data Partitioning Facility database (previously called DB2/UDB EEE). By using the DB2/UDB Enterprise Stage, data is read in parallel from each DB2 node. And, by default, when writing data to a target DB2 database by using the DB2/UDB Enterprise Stage, data is partitioned to match the internal partitioning of the target DB2 table by using the DB2 partitioning method. DB2 partitioning can be specified for target DB2/UDB Enterprise stages only. To maintain partitioning on data read from a DB2/UDB Enterprise Stage, use same partitioning on the input to downstream stages.

10.1.3 Collectors

Collectors combine parallel partitions of an input data set (single link) into a single input stream to a stage that is running sequentially. The collector method is defined in the stage input/partitioning properties for any stage that is running sequentially, when the previous stage is running in parallel. The following list shows the collector types:

- Auto collector

  The *auto collector* first checks the dataset to see whether it is sorted. If so, the framework automatically inserts a sort merge collector instead. If the data is not sorted, the auto collector reads rows from all partitions in the input data set without blocking for rows to become available. For this reason, the order of rows in an auto collector is undefined, and the order might vary between job runs on the same data set. Auto is the default collector method.

- Round-robin collector

  The *round-robin collector* reads rows from partitions in the input dataset by reading input partitions in round-robin order. The round-robin collector is generally slower than an auto collector, because it blocks each partition and waits for a row to become available (or an end-of-data signal is received). However, there is a special situation where the round-robin collector might be appropriate. Assume that a job initially partitioned the data by using a round-robin partitioner, that data is not repartitioned in the job flow, and that the number of rows is not reduced (for example, through aggregation). Then, a round-robin collector can be used before the final sequential output to reconstruct a sequential output stream in the same order as the input data stream. This approach works, because a round-robin collector reads from partitions by using the same partition order that a round-robin partitioner assigns rows to parallel partitions.

- Ordered collector

  An *ordered collector* reads all rows from the first partition, blocking for data to become available. Then, it reads all rows from the next partition, and it continues this process until all rows in the dataset are collected. Ordered collectors are generally only useful if the input dataset is sorted and range-partitioned on the same key columns.

- Sort merge collector

  If the input data set is sorted in parallel, the *sort merge collector* generates a sequential stream of rows in a globally sorted order. The sort merge collector requires one or more key columns to be defined, and these key columns must be the same columns, in the same order, as used to sort the input data set in parallel. The row order is undefined for non-key columns.

10.1.4 Choosing a collector

You use the auto collector in most situations because of its non-blocking behavior. And, it intelligently inserts the sort merge collector when necessary. You need to explicitly define a collector to something other than auto if you want to force a specific collection method. Given the options for collecting data into a sequential stream, the following guidelines form a methodology for choosing the appropriate collector type:

- When output order does not matter, use the auto collector (the default).
- When the input data set is sorted in parallel, use the sort merge collector to produce a single, globally sorted stream of rows.
- When the input data set is sorted in parallel and range partitioned, the ordered collector might be more efficient.

- Use a round-robin collector to reconstruct rows in input order for round-robin-partitioned input data sets, as long as the data set is repartitioned or reduced.

10.2 Sorting relational data basics

Traditionally, the process of sorting data uses one primary key column and, optionally, one or more secondary key columns to generate a sequential ordered result set. The order of key columns determines the sequence and groupings in the result set. Each column is specified with an ascending or descending sort order. This method is the method with which you are probably familiar from using SQL databases and an ORDER BY clause in a SELECT statement. When a data set is sorted all at one time, we refer to this sort as a *global sort*. Figure 10-4 illustrates a global sort.

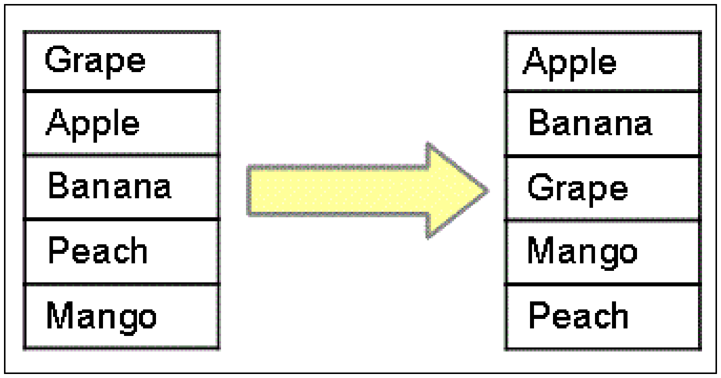


Figure 10-4 Example of a global sort operation

However, in most cases, you do not need to globally sort data to produce a single sequence of rows. Instead, sorting is most often needed to establish order in specified groups of data. This sort can be performed in parallel. The business logic, such as join, can be performed accurately on each sorted group, as long as key-based partitioning is used to ensure that all members of the group are distributed to the same partition. Figure 10-5 on page 299 illustrates key-based partitioning and a parallel sort.

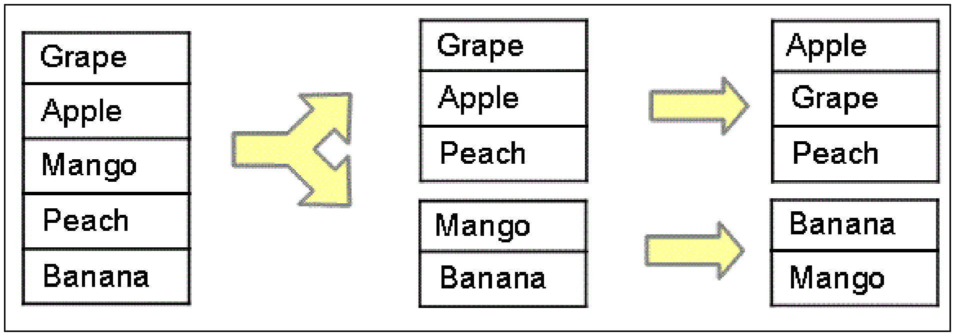


Figure 10-5 Example of parallel sort

Notice that the partitioning happens first. The data is distributed into each partition as it arrives. The sort is performed independently and in parallel on all partitions. Parallel sort yields a significant performance improvement on large data sets. Parallel sort is a high-performance solution. Perform sorting in parallel whenever possible.

10.3 Sorting relational data for composing XML documents

It is considerably more efficient to sort data within DataStage versus by using the internal sort facility within the XML Stage. Relational data must be sorted in DataStage by using the parallel Sort stage before sending it into the XML Stage and its assembly. In multiple step assemblies, where a sort needs to take place in the middle, it is advisable to split the design into two separate XML stages. This way, you perform the relational sort of large amounts of data to use the DataStage parallel sort.

When you need to compose a single XML document, the XML Stage needs to be running in sequential mode. The partitioned and sorted relational data needs to be collected to a single partition where the XML Stage processes it sequentially. To collect the data to a single partition, a sort merge collector must be used to generate a sorted sequential data set. Figure 10-6 on page 300 illustrates a job that performs a high-performance relational sort and composes the XML document from the result.

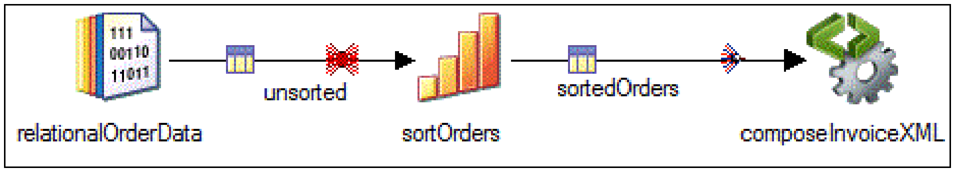
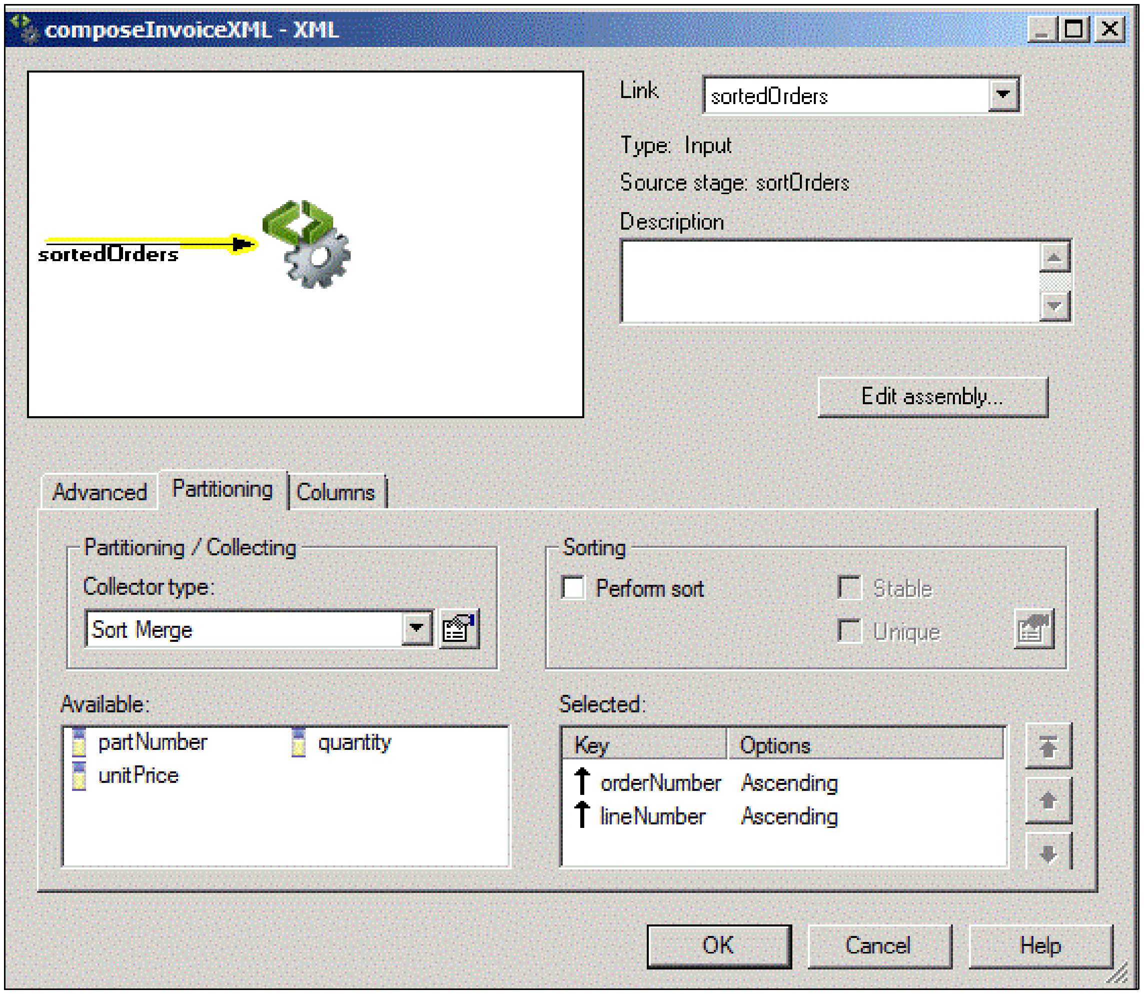


Figure 10-6 Sorting the relational data before composing

With the composeInvoiceXML Stage in the default configuration using the auto collector, the parallel engine framework automatically inserts the sort merge collector when it identifies that the upstream data is sorted. In Figure 10-6, the sort merge collector is inserted at the point where you see the collector link marking icon on the sortedOrders link. By choice, the user can also manually specify a sort merge collector, as shown in Figure 10-7 on page 301.


composeInvoiceXML - XML

Link: sortedOrders

Type: Input

Source stage: sortOrders

Description

Partitioning / Collecting

Collector type: Sort Merge

Sorting

Perform sort [ ]

Stable [ ]

Unique [ ]

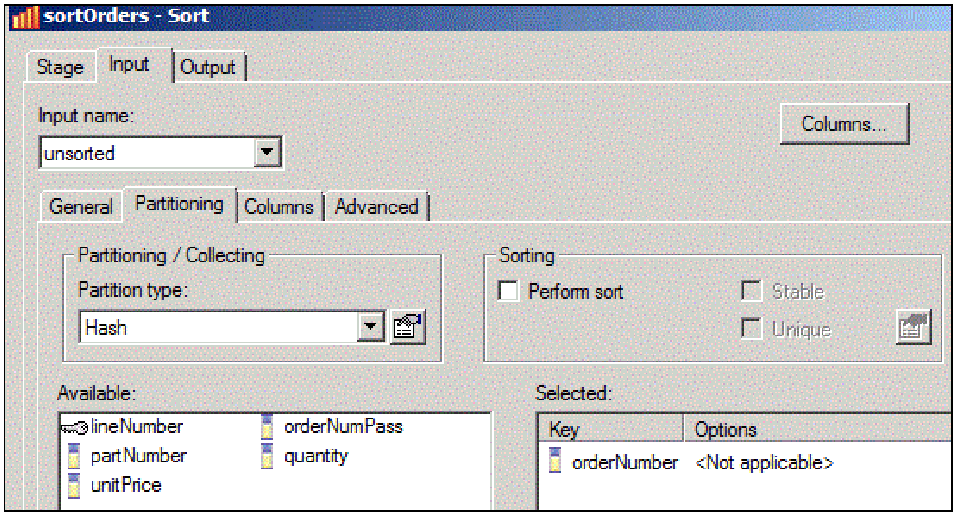
Available: partNumber, quantity, unitPrice

Selected:

Key	Options
orderNumber	Ascending
lineNumber	Ascending

Figure 10-7 Manually defining a sort merge connector

To sort the relational data, you need to specify the sort and partitioning criteria to match the columns that we use as keys within the XML assembly. Figure 10-8 on page 302 illustrates the configuration of the *hash partitioner*, which is a key-based partitioning method. Order number is selected as the hash key, which ensures that all the records that belong to each order are hashed to the same partition.


sortOrders - Sort

Stage | Input | Output

Input name: unsorted

General | Partitioning | Columns | Advanced

Partitioning / Collecting
Partition type: Hash

Sorting
☐ Perform sort ☐ Stable ☐ Unique

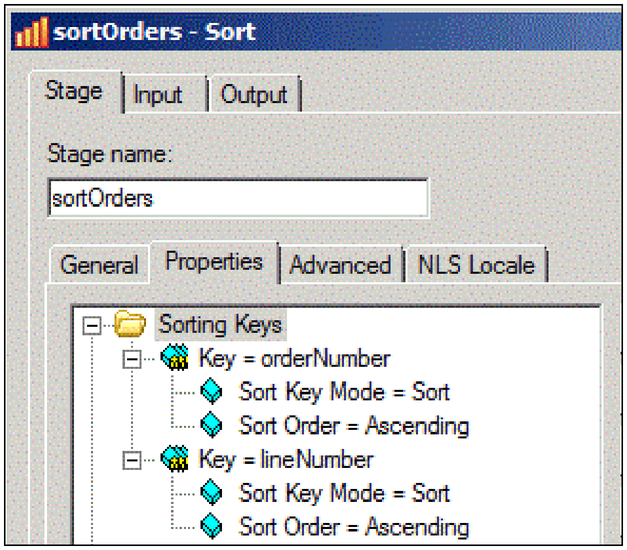
Available:
lineNumber
partNumber
unitPrice
orderNumPass
quantity

Selected:

Key	Options
orderNumber	<Not applicable>

Figure 10-8   Configuring the hash (a key-based) partitioning method

In the XML assembly, we need to regroup the lines within each order, and so we sort by using order number and line number as keys. *The sort keys and the partitioning keys do not need to match exactly*. The sort keys can be a super-set of the partitioning keys; however, the order must be preserved. Therefore, the order number must be the first sort key. Figure 10-9 illustrates the configuration of the sort keys.


sortOrders - Sort

Stage | Input | Output

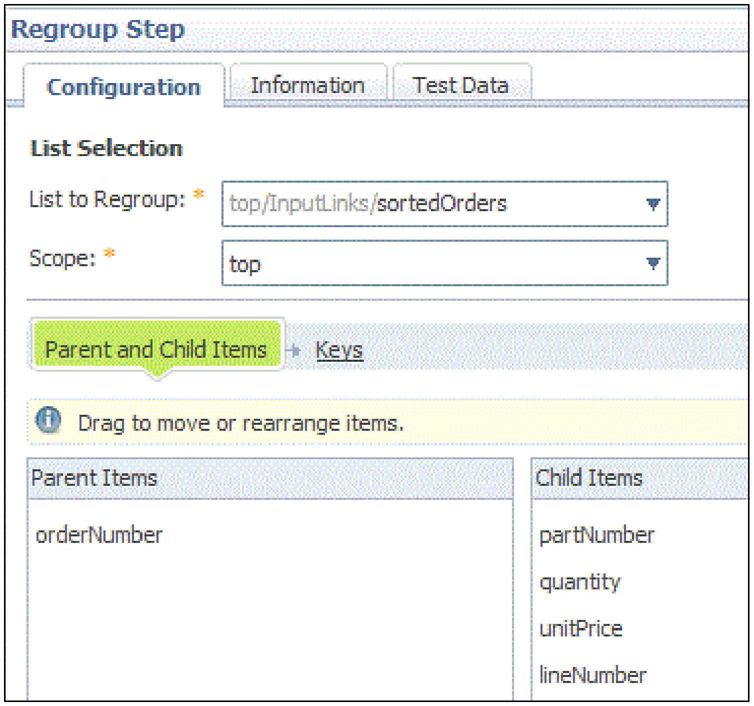
Stage name: sortOrders

General | Properties | Advanced | NLS Locale

```
Sorting Keys
  Key = orderNumber
    Sort Key Mode = Sort
    Sort Order = Ascending
  Key = lineNumber
    Sort Key Mode = Sort
    Sort Order = Ascending
```

Figure 10-9   Configuring sort keys

Inside the XML assembly, the incoming relational data is in a flat structure, with all of the columns in a single list. One incoming record exists for each line item, and the order information is repeated for each row. To create the hierarchical structure, the order elements and attributes repeat only one time per order, and the line item elements and attributes are nested within the order and repeat for each line item. Figure 10-10 illustrates the configuration of the Regroup Step within the XML assembly.



List Selection

List to Regroup: * top/InputLinks/sortedOrders

Scope: * top

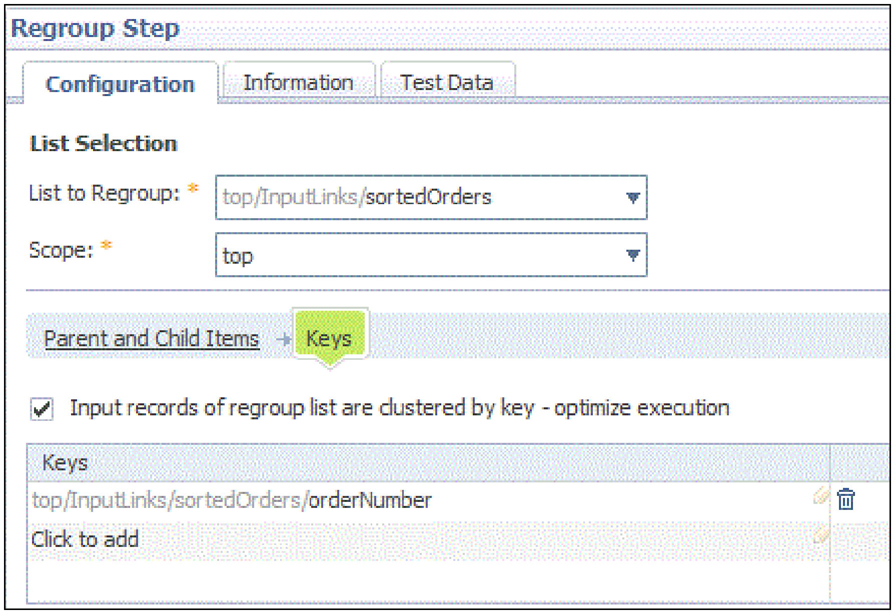
Drag to move or rearrange items.

Parent Items	Child Items
orderNumber	partNumber
	quantity
	unitPrice
	lineNumber

Figure 10-10 Configuring the list and parent-child items for the Regroup Step

For the List to Regroup option, we select the sortedOrders list, which is the incoming link that holds the relational data. We set the Scope option to `top`, which means that the regrouped output structure shows under the top level at the output. This choice is the only available choice for the scope property, because the sortedOrders input list is a flat structure. The Regroup step forms a hierarchical structure with a parent and a child list. The line items are part of the child list, so the orderNumber is configured to be part of the parent list.

The *key specification* defines when an item is added to the output regroup list. In this example, we need the regroup list to contain one item per order, so we use the order Number as the key. Figure 10-11 on page 304 illustrates the configuration of keys.


Regroup Step

List Selection

List to Regroup: * top/InputLinks/sortedOrders

Scope: * top

[✓] Input records of regroup list are clustered by key - optimize execution

Keys
top/InputLinks/sortedOrders/orderNumber
Click to add

Figure 10-11 Configuring key and optimization for the Regroup Step

The “Input records of regroup list are clustered by key - optimize execution” option instructs the Regroup step that the data is already sorted on the regroup key. This sort used the parallel sort in the DataStage job prior to the XML Stage.

Important: You must select this optimize execution option to achieve the performance benefit. If not, the XML Stage assumes that the incoming data is unsorted and starts a less efficient, non-parallel sort.

10.4 Partitioning and parallel processing with XML documents

XML can be processed in parallel with DataStage in two ways. The first method is to use the partition parallelism capabilities of the DataStage Engine. For many relatively small documents, the partition parallelism capabilities of DataStage can help improve the overall throughput of parsing documents. While each document is parsed in the conventional, sequential fashion, multiple documents can be parsed simultaneously. The second method is to use the parallel parsing feature within the XML Stage. This solution is used when a few large documents exist.

One document is parsed at a time, but multiple parsers simultaneously work on a subset of the document. The following sections discuss each method in detail.

10.4.1 Using DataStage partition parallelism

Consider a fictional coffee shop company, with more than 1,200 stores throughout the country. Every day, as each work shift ends, sales invoice data in the form of XML documents is uploaded from each store to the corporate headquarters for processing. A daily DataStage job loads these documents into the data warehouse so that figures can be available to the decision support systems that management uses to drive its business. Each time that the job runs, it processes many thousands of files.

To design a solution, we start with the XML files. Each file is stamped with a date and time code, plus the store number from where it originated. This information presents an opportunity, because many of the business calculations are driven by store number. The DataStage job is shown in Figure 10-12.

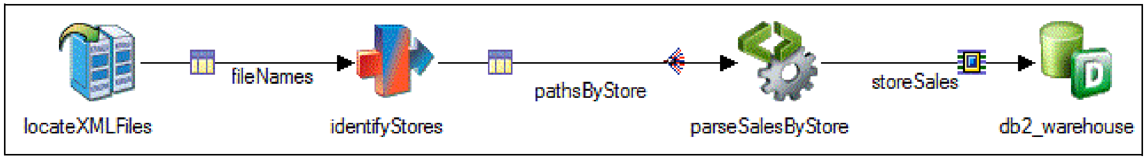


Figure 10-12 DataStage job to process many small XML files in parallel

By using an External Source stage, we obtain a data stream of file paths to the XML files, one path per record. In the transformer, we extract the store number into its own column and pass it along with the full XML file path to the XML Stage. A hash partitioner that uses the store number as the key distributes the file paths to multiple instances of the XML Stage that run in parallel. Each instance of the XML Stage on each partition parses an XML file for the data for one store. The data is loaded into an IBM DB2 database table that is partitioned by store number. Next, we describe each stage in this job.

First, the External Source stage is used, in sequential mode, to obtain a list of the XML files in the directory to be processed, as shown in Figure 10-13 on page 306.

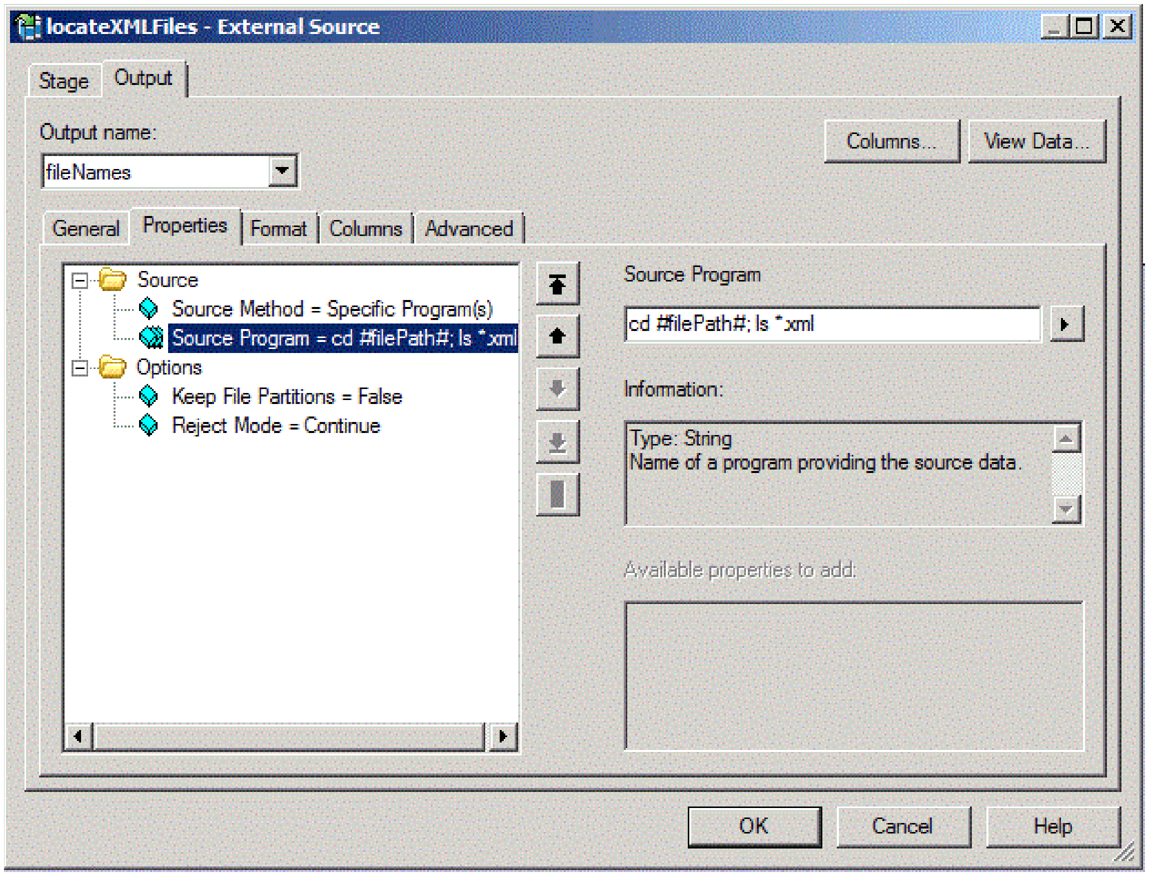
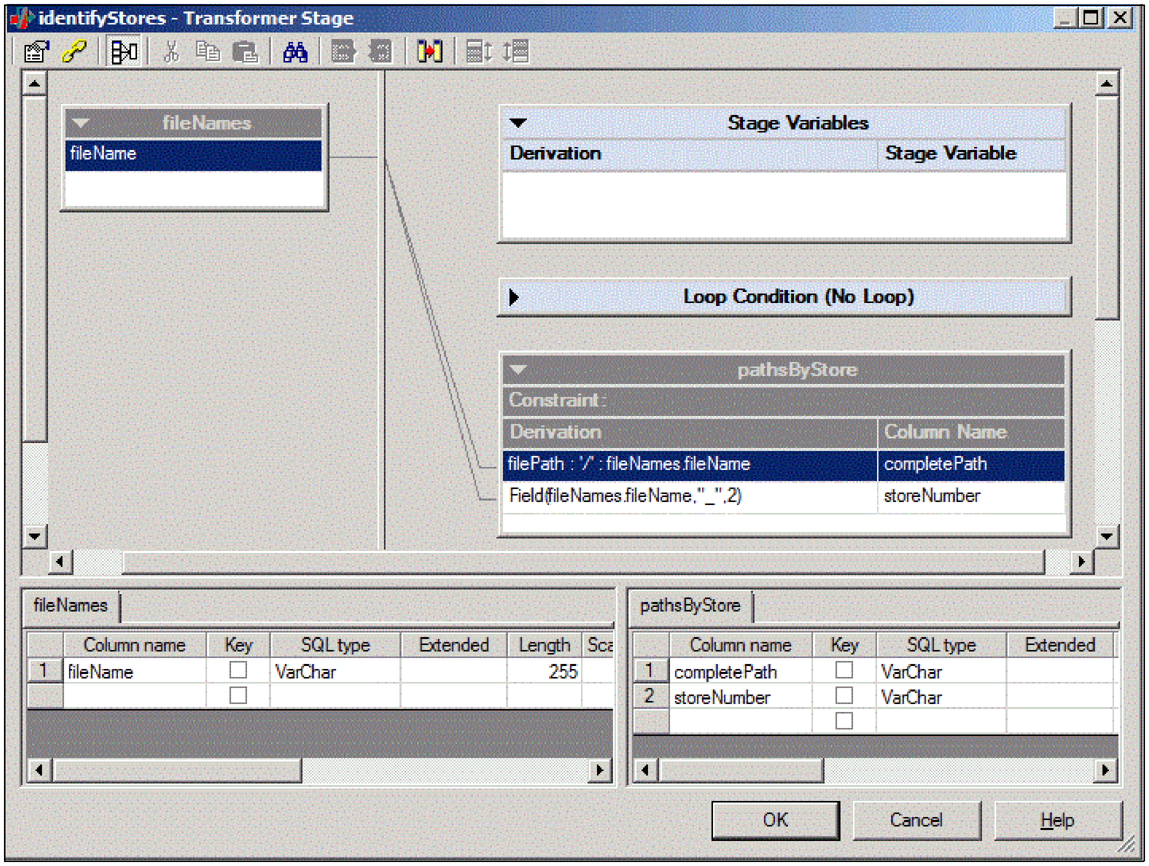

locateXMLFiles - External Source
Stage
Output
Output name:
fileNames
Columns...
View Data...
General
Properties
Format
Columns
Advanced
Source
Source Method = Specific Program(s)
Source Program = cd #filePath#; ls *xml
Options
Keep File Partitions = False
Reject Mode = Continue
Source Program
cd #filePath#; ls *xml
Information:
Type: String
Name of a program providing the source data.
Available properties to add:
OK
Cancel
Help

Figure 10-13 The external source stage can dynamically obtain a list of files

The path is defined as a job parameter by using the `#parameterName#` syntax. The source program is two simple shell commands with a semicolon as the command separator. These commands are standard for most UNIX or Linux operating systems. They also work on Microsoft Windows, because DataStage includes the MKS Toolkit, which provides shell programs similar to UNIX.

The Transformer stage also runs sequentially and extracts the store number, as shown in Figure 10-14 on page 307.


identifyStores - Transformer Stage

fileNames: fileName

Stage Variables

Derivation	Stage Variable

Loop Condition (No Loop)

pathsByStore

Constraint:

Derivation	Column Name
filePath : '/' : fileNames.fileName	completePath
Field(fileNames.fileName,"_",2)	storeNumber

fileNames

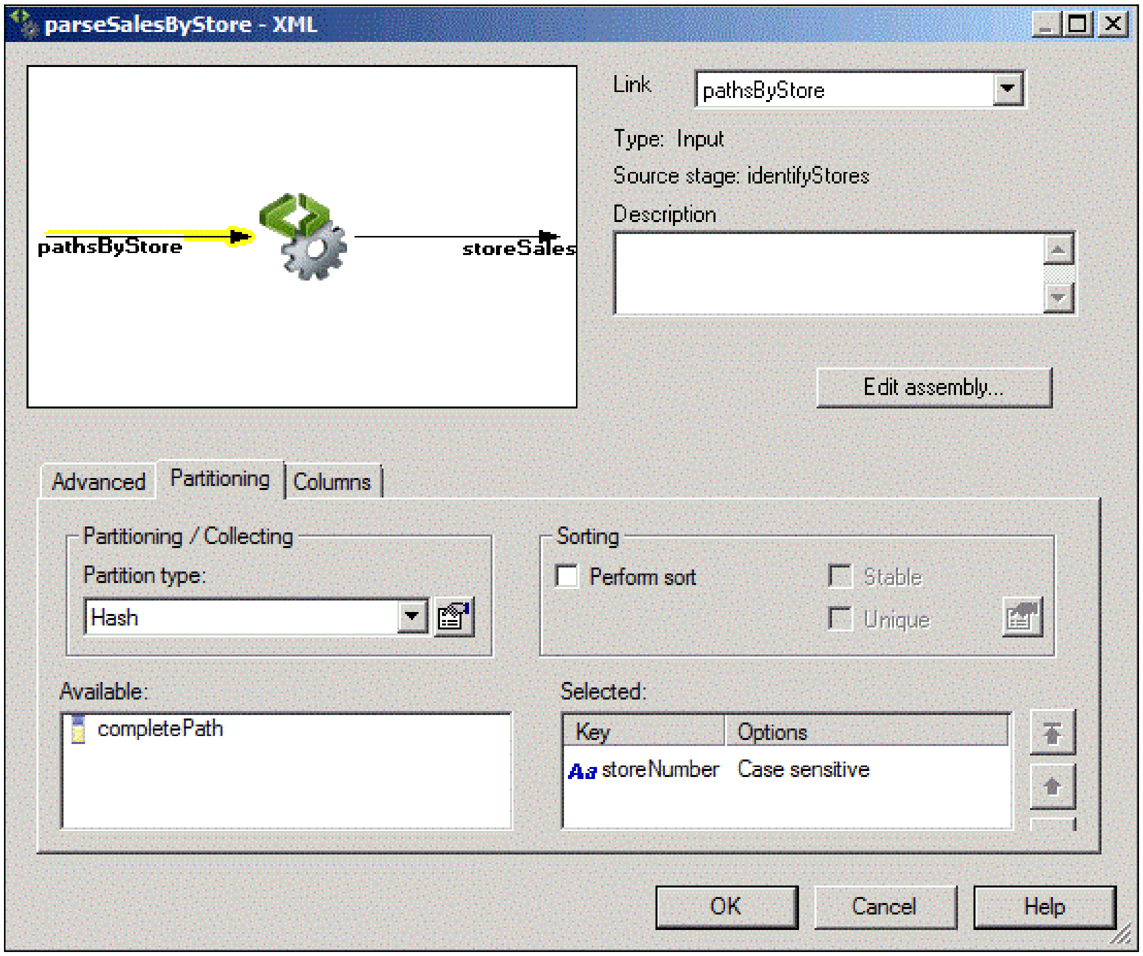
	Column name	Key	SQL type	Extended	Length	Sca
1	fileName	☐	VarChar		255	
		☐				

pathsByStore

	Column name	Key	SQL type	Extended
1	completePath	☐	VarChar	
2	storeNumber	☐	VarChar	
		☐		

Figure 10-14 Transformer stage extracts the store number from the file name

The transformer assembles the complete file path from the file name and the path from the job parameter. It extracts the pure store number from the XML file name and returns that number as a separate output column. The hash partitioner is defined on the input link on the XML Stage by using the store number field as the key, as shown in Figure 10-15 on page 308.



Link: pathsByStore

Type: Input

Source stage: identifyStores

Description

Partitioning / Collecting

Partition type: Hash

Sorting

Perform sort

Stable

Unique

Available:

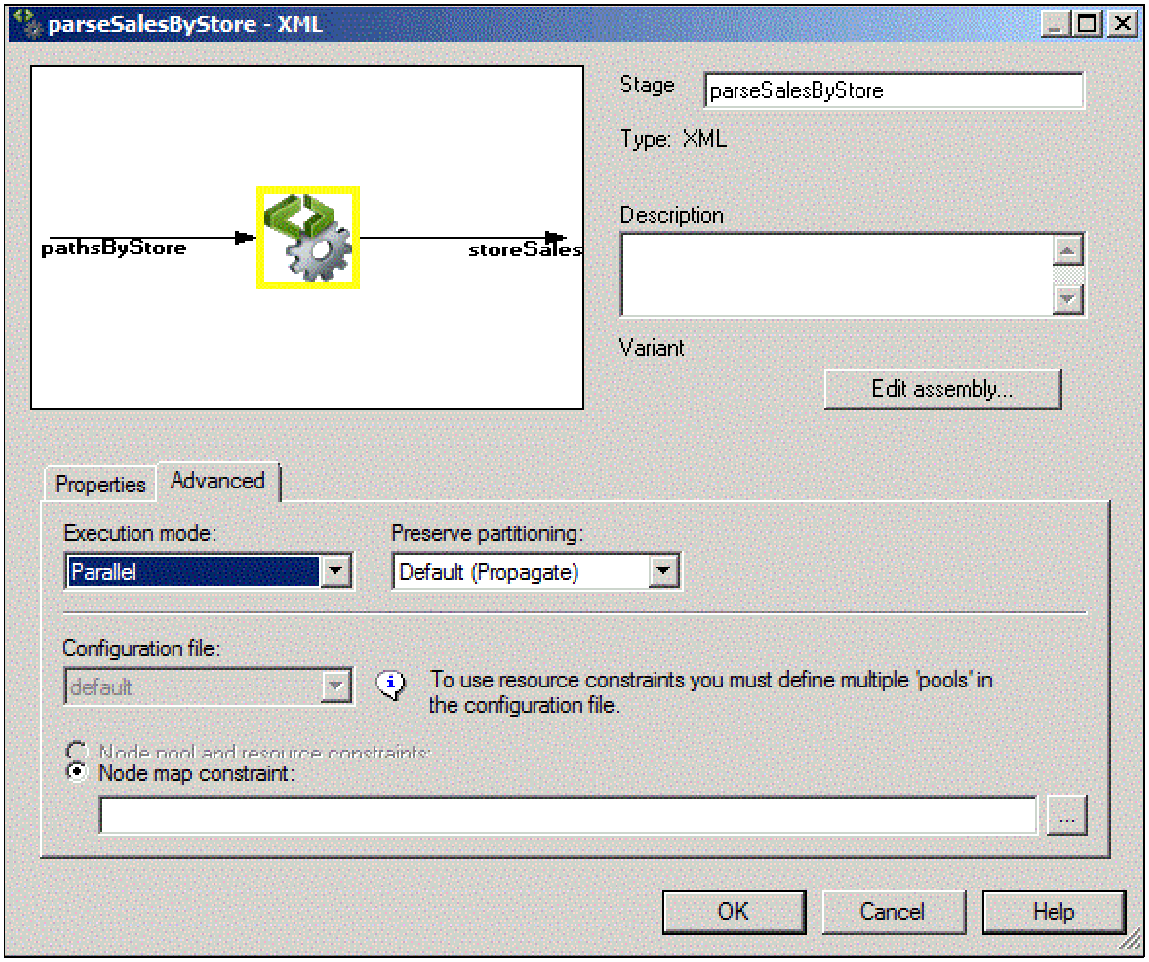
completePath

Selected:

Key	Options
storeNumber	Case sensitive

Figure 10-15 The hash partitioner configured on the input link of the XML Stage

By default, the XML Stage runs in sequential mode. To configure it to run in parallel, we open the **Advanced** tab on the XML Stage properties. We configured the Execution mode property to **Parallel**, as shown in Figure 10-16 on page 309.


parseSalesByStore - XML

Stage: parseSalesByStore

Type: XML

Description

Variant

Properties | Advanced

Execution mode: Parallel

Preserve partitioning: Default (Propagate)

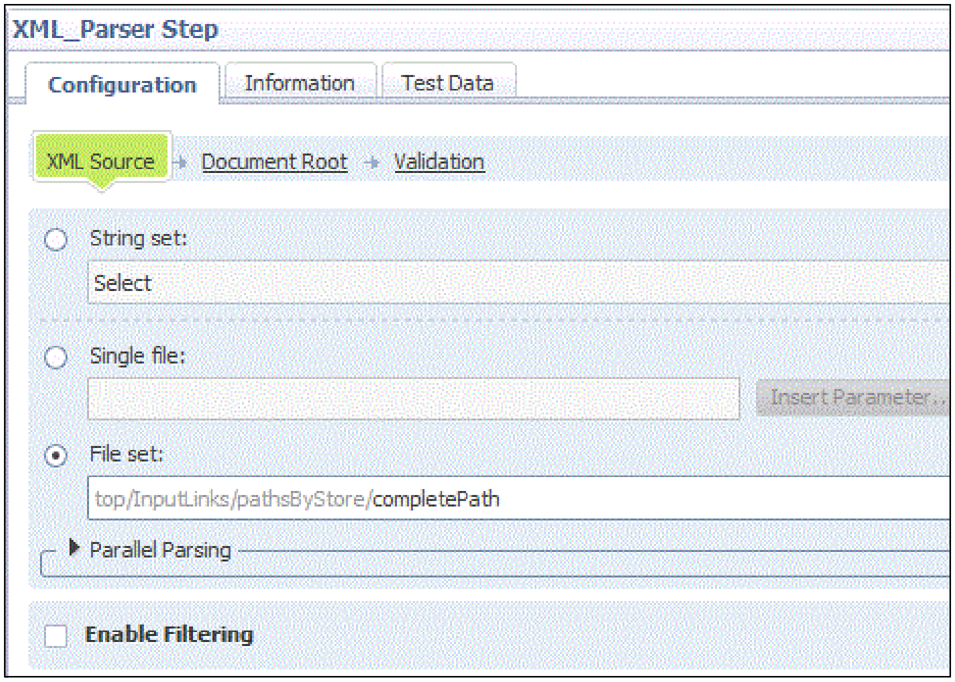
Configuration file: default

To use resource constraints you must define multiple 'pools' in the configuration file.

( ) Node pool and resource constraints:
(•) Node map constraint:

Figure 10-16 The XML Stage configured to run in parallel

In the XML assembly, the Parser step must be configured to accept the incoming path from the input link and, then, open each file for parsing. This configuration is shown in Figure 10-17 on page 310. The rest of the assembly is configured normally, exactly as you parse a single file.


XML_Parser Step

( ) String set: Select

( ) Single file:

(•) File set: top/InputLinks/pathsByStore/completePath

[ ] Enable Filtering

Figure 10-17 The Parser step is configured to read a file set from the input link

This solution is effective and benefits from the increased I/O throughput of reading multiple files simultaneously. Additionally, because the XML files store the data that is already broken down by store number, it does not need to be repartitioned. Even in other situations where repartitioning might be required, it is most often the case that the performance boost gained from increased throughput more than offsets any cost for repartitioning. You can use a similar strategy to compose XML documents, as long as each partition produces a document or stream of multiple documents.

Important: If a single XML document needs to be created with data from multiple partitions, that XML Stage must be run in sequential mode.

10.4.2 Parallel parsing large XML documents

In the previous section, we used DataStage partition parallelism to increase performance with a stream of many small XML documents. If a few large XML files exist, we need a separate approach. The XML Stage is designed to handle a specific use case that appears frequently among extract, transform, and load (ETL) scenarios that involve parsing large XML documents. The XML document contains a single repeating list element that is a direct descendant of the root element for the document, and there are many occurrences of this structure. Consider a large XML document that contains invoicing information for a business quarter. This document might contain hundreds of thousands of orders, potentially into the millions. The document needs to be parsed, and data needs to be extracted for each order. Figure 10-18 illustrates that the order element, highlighted in this image in red, meets the criteria for parallel parsing.

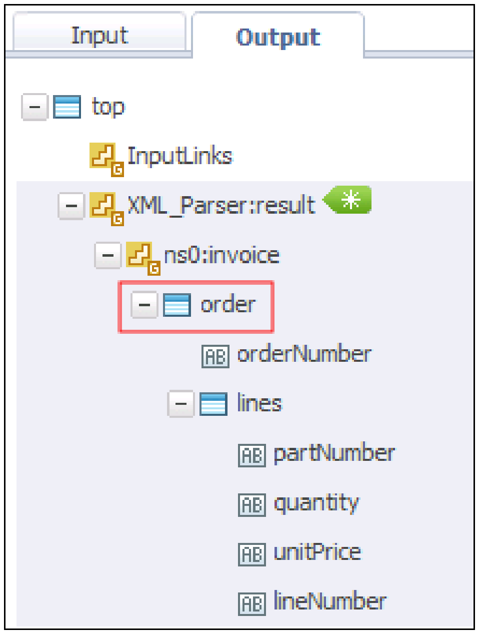


Figure 10-18 The order list is a candidate for parallel parsing

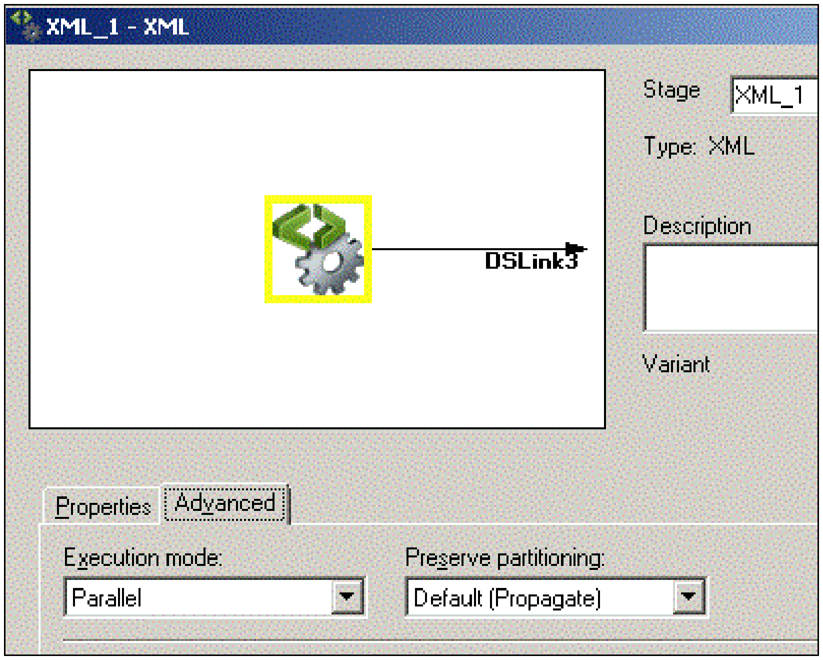
Qualifications for parallel parsing

To enable parallel parsing, all of the following qualifications must be met:

- The XML Stage must be configured to run in parallel mode.
- The XML Stage must be configured to read the file from disk. It can be an individual file, a set of files, or an input link that feeds it paths to a set of files.
- There must be a list (repeating element) immediately under the `top` level (root node) of the document that is parsed.
- Any additional steps that follow the Parser step in the assembly must be (at the time of writing this book) limited to the Switch step, H-Pivot step, and Output step.
- There must be only one output link, and the list directly under the top level must be mapped to it.

Requirement 1: Configuring the XML Stage for parallel mode

The XML Stage runs in sequential mode, by default. To switch the XML Stage to parallel mode, edit the stage properties, click the stage icon in the image map, and select the **Advanced** tab. For Execution mode, select **Parallel**, as shown in Figure 10-19.

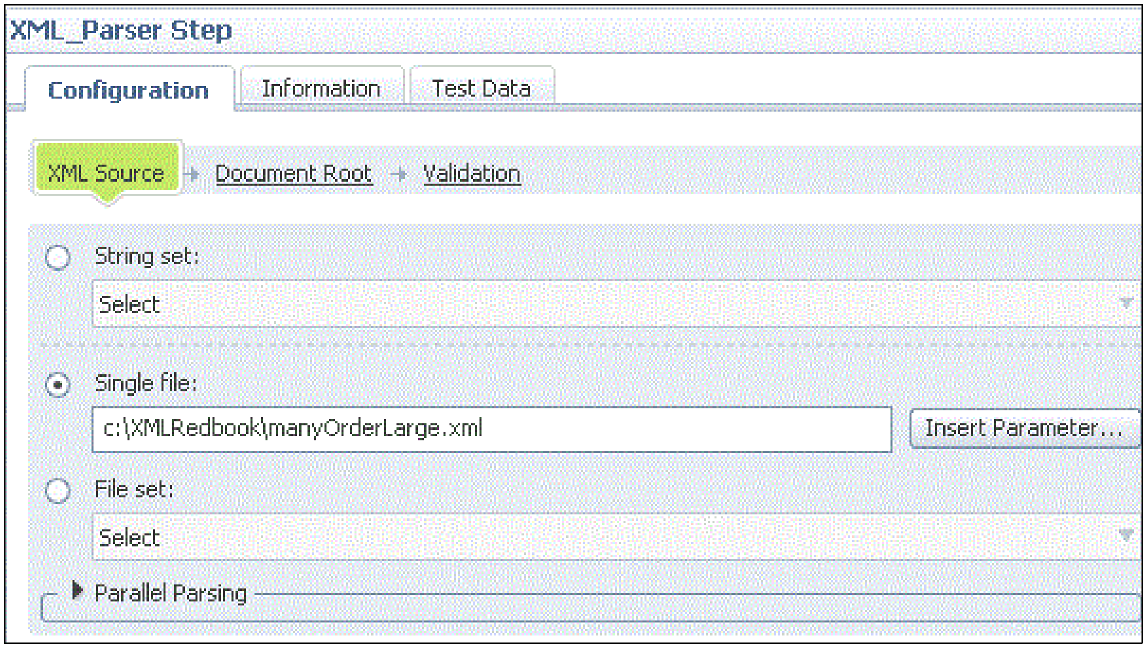


Stage: XML_1
Type: XML
Description
Variant
Execution mode: Parallel
Preserve partitioning: Default (Propagate)

Figure 10-19 Configure XML Stage to run in parallel mode

Requirement 2: Reading the XML file from disk

The stage is not capable of parsing in parallel if the document is sent to the XML Stage as an in-memory (or large object (LOB) reference) string. It must be able to pick up the XML files directly from disk. The job can be designed so that the XML Stage has zero input links, and its input files are specified directly in the parser step. A job parameter can be used, if you want, to specify the file location at run time. Figure 10-20 on page 313 illustrates this configuration.


XML_Parser Step

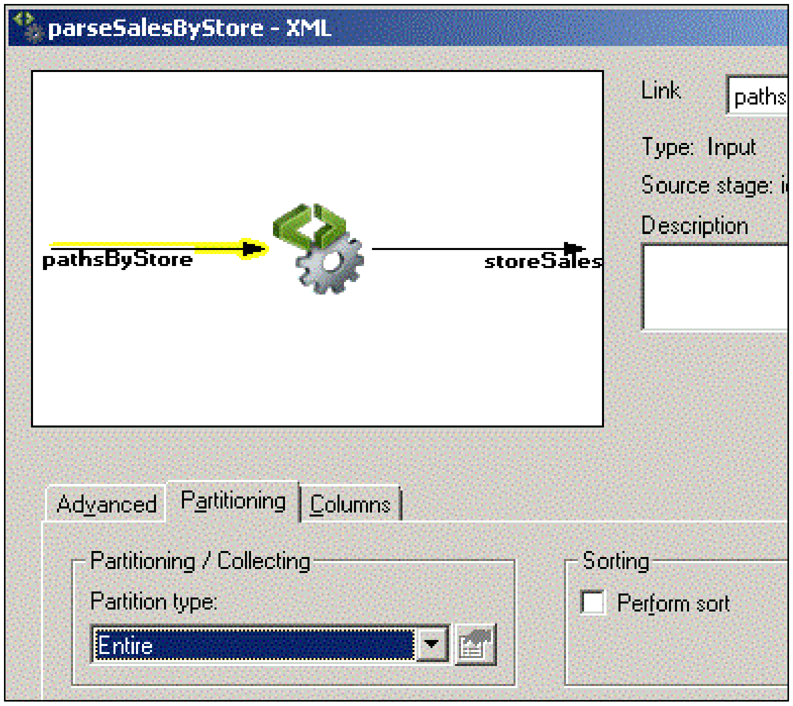
String set:
Select

Single file: [✓]
c:\XMLRedbook\manyOrderLarge.xml

File set:
Select

Figure 10-20 Specify single input file

If the XML Stage is given an input link, and the records supply the document paths, one additional configuration step is required. The partitioner on the input link to the XML Stage must be configured to use the Entire partitioning mode. Entire partitioning mode is necessary so that the file path is distributed to each instance of the XML Stage that runs on each partition. Figure 10-17 on page 310 illustrates how to configure the XML Stage to read a file path from an input link. To configure the XML Stage to use entire partitioning, edit the stage properties and click the input link in the image map. Click the **Partitioning** tab, and choose **Entire** from the Partition type choices, as shown in Figure 10-21 on page 314.



Link paths

Type: Input

Source stage: i

Description

Partitioning

Partitioning / Collecting

Partition type: Entire

Sorting

Perform sort

Figure 10-21 Configure XML Stage to use Entire partitioning

Requirement 3: Repeating element directly under the root node

It is a requirement that the repeating element must be immediately under the top level of the document to ensure that the document can be split correctly into multiple partitions. Figure 10-18 on page 311 shows the Output tab of the Parser step in the XML Stage assembly. The order list, which is highlighted, is the only structure that is a candidate for parallel parsing.

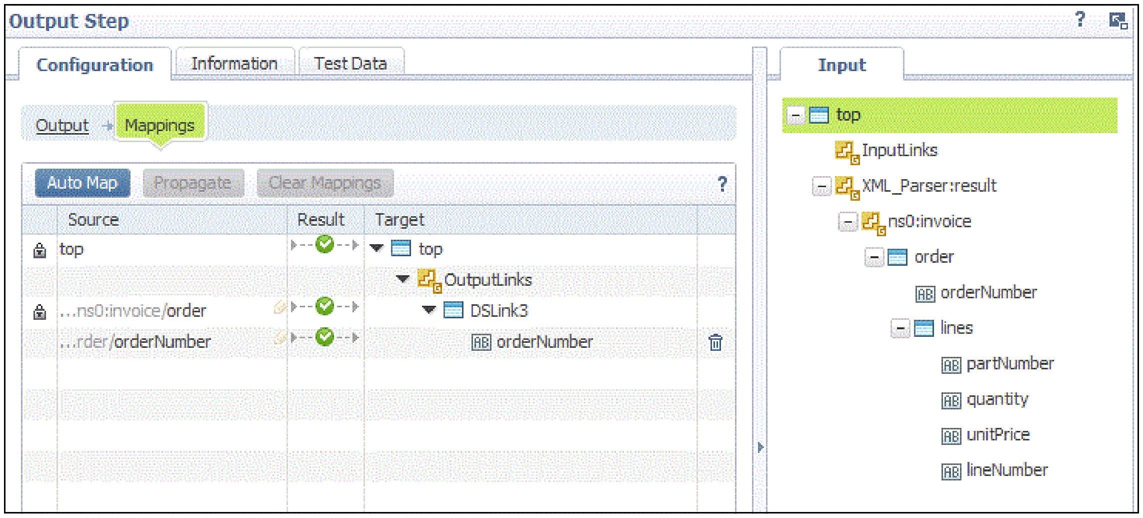
Requirement 4: Additional steps in the assembly are limited

At the time of writing this book, only the Switch step, H-Pivot step, and Output step can be used in conjunction with parallel parsing. If any other step is used, parallel parsing is not enabled.

Requirement 5: Single output link with mapped repeating element

The top-level repeating structure must be the only structure that is mapped to an output link. *One XML Stage can have only one partitioning method at a time.*

Figure 10-22 on page 315 shows that the /invoice/order list is mapped to the output link. If you add additional output links, or map a more deeply nested list, such as /invoice/order/lines, parallel parsing is not enabled. Therefore, we see a problem if we want to parse in parallel but we need to access data that is more deeply nested, such as the quantity, unitPrice, and partNumber elements from the list of lines. In the next section, we describe a solution to this problem.



Source	Result	Target
top	✓	top
		OutputLinks
...ns0:invoice/order	✓	DSLink3
...rder/orderNumber	✓	orderNumber

```
top
  InputLinks
  XML_Parser:result
    ns0:invoice
      order
        orderNumber
        lines
          partNumber
          quantity
          unitPrice
          lineNumber
```

Figure 10-22 Output step mapping for parallel processing

Mechanics of parallel parsing

The parallel parsing feature works by calculating the boundaries of the repeating element and then assigning a range of these element instances to each partition. Each partition gets assigned a range of repeating element instances to process. The number of partitions is defined by the number of logical nodes in the `APT_CONFIG_FILE`. Each instance of the Parser step parses its own range of data and then passes it along to the next step in the assembly.

DataStage partitioning considerations

The resulting output from the XML Stage is range partitioned, according to the boundaries that we set during the Parsing step. Continuing further with the invoicing example, if 100,000 orders are in the document, and a two-node configuration file, the first node gets orders 1 - 50,000 and the second node gets orders 50,001 - 100,000. In most cases, the downstream stage needs to repartition the data by using a partitioning method, such as key-based hash partitioning, to implement the rest of the business rules. Although repartitioning overhead reduces performance, the increased throughput from the parallel parsing almost always is substantially greater than the loss due to repartitioning.

Important: It is prudent to performance test your specific application with both sequential parsing and parallel parsing to determine the most effective result.

10.4.3 Advanced parallel parsing strategy

Continuing with the example from the previous section, a large XML document contains the invoicing information for a complete business quarter. This document might contain hundreds of thousands of orders, potentially millions. We need to be able to retrieve information from all of the lines within each order. As we explained in the rules for parallel parsing, we cannot process the line elements in parallel, because they are not a top-level structure. Our solution is to parse the invoice document in parallel by order, and extract each order as an XML chunk. We can then pass the XML chunk to a second downstream XML Stage, to parse the lines from within each order in the conventional way. Figure 10-23 shows the initial part of the job design.

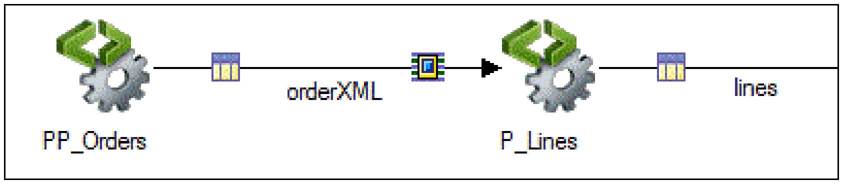
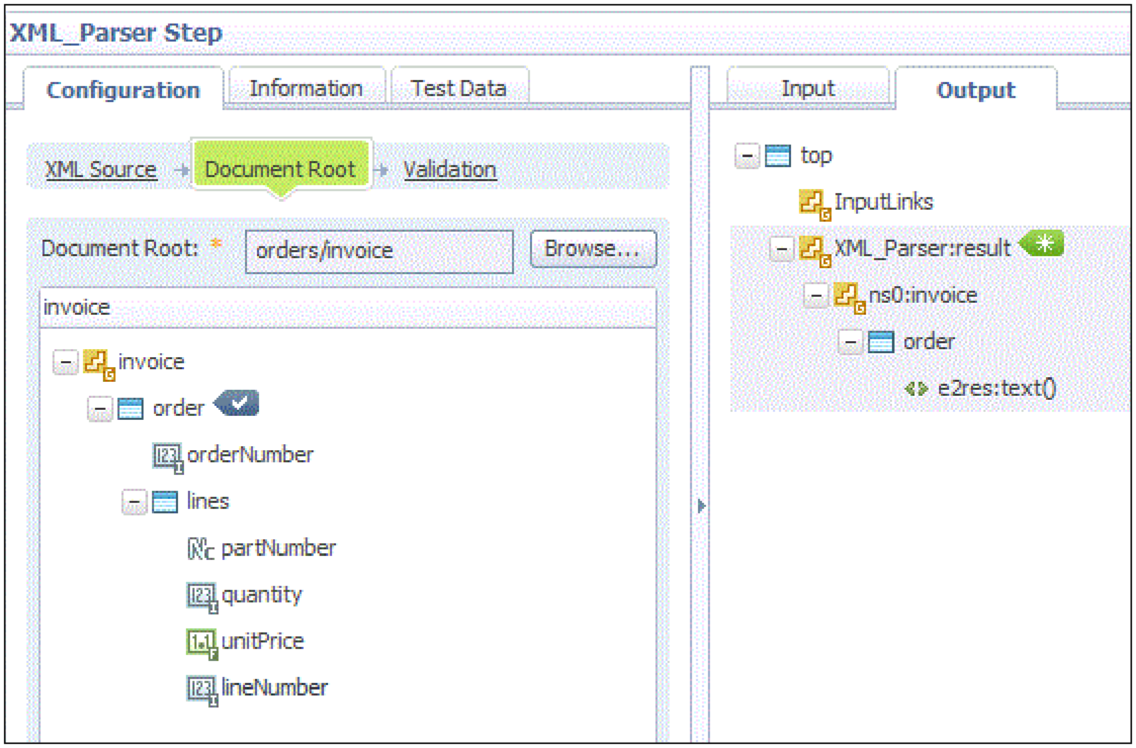


Figure 10-23 Passing a chunk of XML between stages

It is important to recognize that the second XML Stage cannot parse the lines in parallel. The actual chunk of XML containing the order must be passed as a string set from the first XML Stage to the second XML Stage. Parallel parsing is only enabled when reading an XML file directly; it is not possible to parallel parse from a string set in memory.

This design lets us get to the necessary data within the lines that we need. At the same time, this design gives us a substantial increase in performance when processing a large file. Right-click the **order** element on the Document Root tab, and choose **Chunk** from the context menu. If you then click the **Output** tab in the right pane, you can see that the order list element now shows that its only child node is <>e2res:text(). The child node is a chunk of XML that is not parsed further within this step. It is sent byte-for-byte to the next step in the assembly, which is illustrated in Figure 10-24 on page 317.



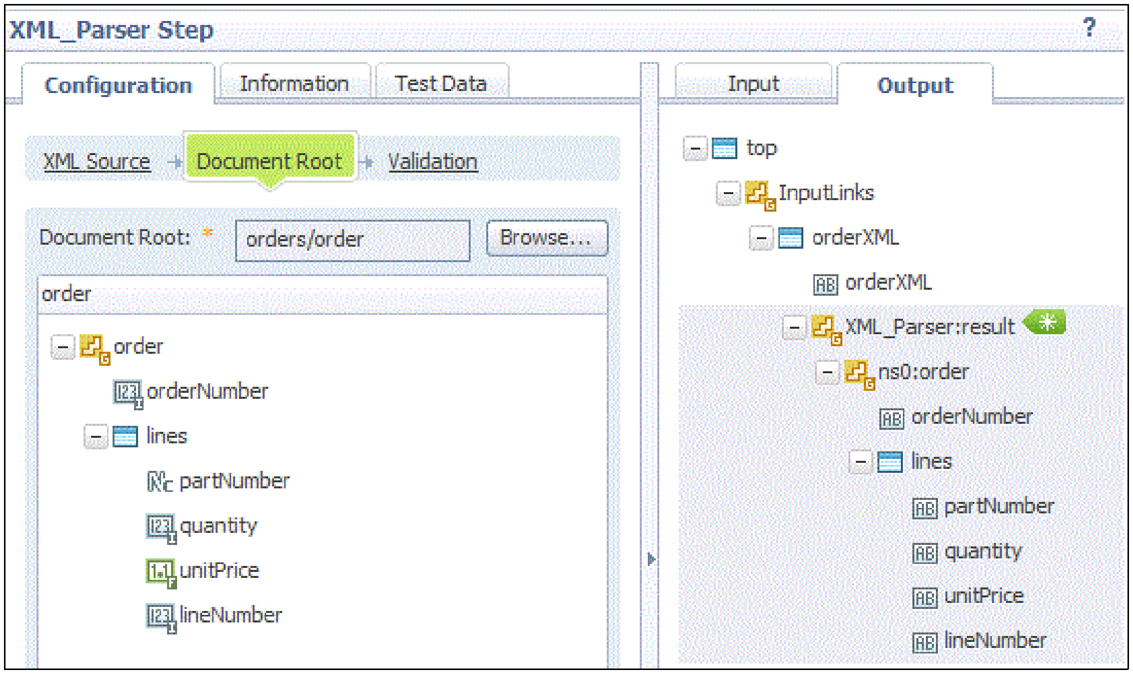
Document Root: * orders/invoice

invoice
- invoice
  - order
    - orderNumber
    - lines
      - partNumber
      - quantity
      - unitPrice
      - lineNumber

- top
  - InputLinks
  - XML_Parser:result
    - ns0:invoice
      - order
        - e2res:text()

Figure 10-24 Each order is passed to the next step as a chunk

The second XML Stage is set up in a typical parsing fashion. The Parser step reads from the input link as a string set. The document root is configured to parse the order element, as shown in Figure 10-25 on page 318.



Document Root: * orders/order

order

- order
  - orderNumber
  - lines
    - partNumber
    - quantity
    - unitPrice
    - lineNumber

- top
  - InputLinks
    - orderXML
      - orderXML
      - XML_Parser:result
        - ns0:order
          - orderNumber
          - lines
            - partNumber
            - quantity
            - unitPrice
            - lineNumber

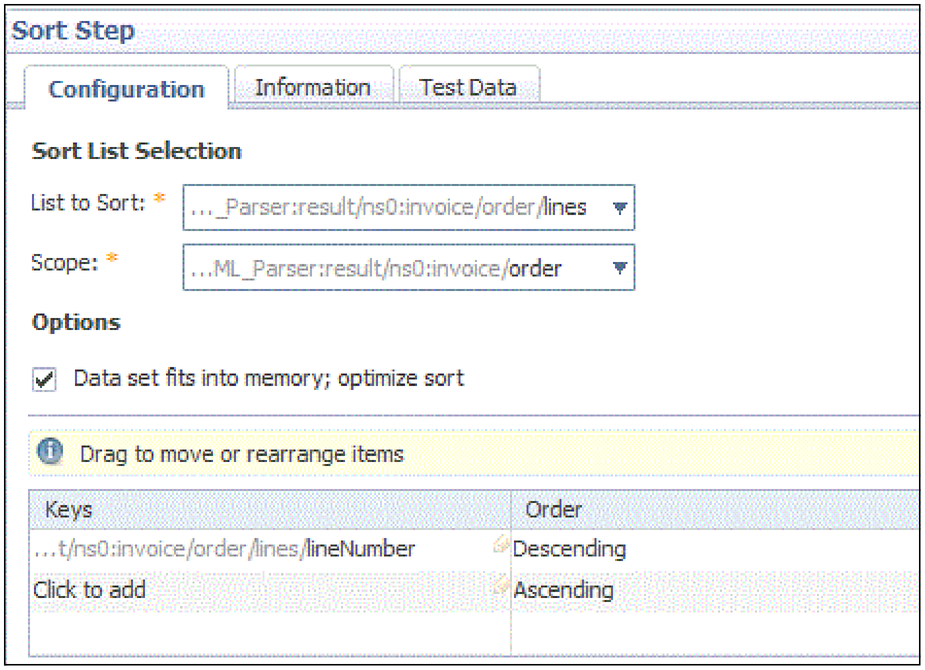
Figure 10-25 Each chunk contains one order as the document root

10.5 Tuning XML Transformation steps

Several Transformation steps in the assembly contain optimization features. To get optimal performance, ensure that the optimization features are always used. We describe the optimization features of the Sort and H-Join steps in this section.

10.5.1 Sort step

The Sort step optionally can use memory for the sort. If the data that you need to sort is so large that you cannot use this option, sort the data by using the DataStage parallel sort and not in the XML assembly. See Figure 10-26 on page 319, which illustrates this setting when enabled.


Sort Step

Configuration | Information | Test Data

Sort List Selection

List to Sort: * …_Parser:result/ns0:invoice/order/lines

Scope: * …ML_Parser:result/ns0:invoice/order

Options

[✓] Data set fits into memory; optimize sort

Drag to move or rearrange items

Keys	Order
…t/ns0:invoice/order/lines/lineNumber	Descending
Click to add	Ascending

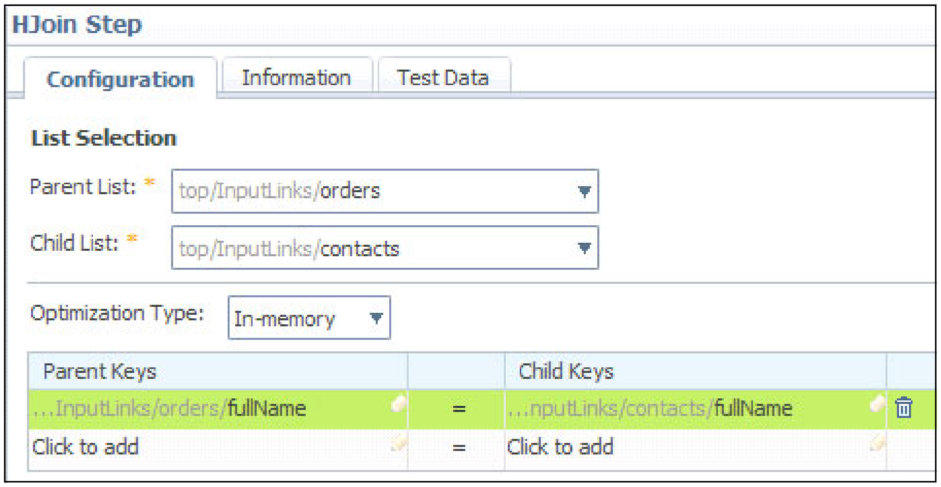
Figure 10-26 Enable in-memory optimization for the Sort step

If this option is not enabled, it uses disk-based files for sorting. It does not sort in parallel. Disk-based sorts in the XML assembly must be avoided if at all possible. The Sort step is most useful when dealing with small amounts of data that fit completely into memory. The sort must be performed in the middle of a complex hierarchical transformation. The sort cannot be performed by using DataStage parallel sort in advance of or after the XML Stage.

Important: The option to use memory is absolute. After you select the in-memory sort, you cannot fall back to disk-based sort in the event that the data does not fit in memory. If it cannot allocate enough memory to perform the sort, the job fails with a runtime error.

10.5.2 H-Join step

The hierarchical join (H-Join) step has an option to choose between disk-based and in-memory-based operation. The H-Join step is not performed in parallel. Similar to the previous statements for the Sort step, if the data does not fit into memory, it is better to perform the join in parallel by using a DataStage join stage. Figure 10-27 illustrates the in-memory optimization feature for the H-Join.


HJoin Step

Configuration | Information | Test Data

List Selection

Parent List: * top/InputLinks/orders

Child List: * top/InputLinks/contacts

Optimization Type: In-memory

Parent Keys		Child Keys
...InputLinks/orders/fullName	=	...nputLinks/contacts/fullName
Click to add	=	Click to add

Figure 10-27 Enable in-memory optimization for the Join step

Important: As with the Sort step, the in-memory H-Join option is absolute. You cannot fall back to disk-based sort in the event that the data does not fit in memory. If it cannot allocate enough memory to perform the join, the job fails with a runtime error.

10.6 Passing XML documents or chunks within the job

In certain situations, you need to pass chunks of XML or entire XML documents through the DataStage job instead of reading them from files. In 10.4.3, “Advanced parallel parsing strategy” on page 316, we described an example as an advanced parsing technique. The following situations are additional examples of this requirement:

- Reading or writing an XML document to or from an IBM WebSphere MQ message queue
- Reading or writing a SOAP message body to or from a web service
- Reading or writing an XML document to or from a column in a database table

There are performance implications with these job designs. The DataStage parallel framework is primarily designed to pass large volumes of relatively small records. Buffering is created based on a relatively small record size. To move data from one stage to another, the entire record must fit inside a transport block. The default block size is 128 KB.

If the record contains a column with large chunks of XML, the transport block size likely is exceeded, resulting in the failure of the job at run time. It is possible to increase the transport block size to accommodate larger records. It is not possible to change the transport block size for individual links. The change only can be made simultaneously for all the links in the job. Therefore, increasing the transport block can result in substantially increased memory and disk space usage. You might not experience a problem if the job has relatively few stages. For large job designs with many stages, the job might be unusable within the constraints of the available resources of the DataStage runtime environment. As a solution to this problem, the Connector framework in Information Server provides a "pass by reference" implementation for managing large character and binary objects, which we discuss in the following sections.

10.6.1 Passing XML in-line

The default behavior as described in the previous section is that all column data is passed "in-line" within the record structure. For small XML chunks or documents, passing in-line is ideal when the total record size remains less than the default block size. For small to medium job designs, increasing the block size to be larger than the maximum document size is reasonable up to about 1 MB. To increase the block size, set APT_DEFAULT_TRANSPORT_BLOCK_SIZE=*n*, where the value of *n* is in kilobytes. Although it is possible, we do not advise that you set this value higher than 1048576 (1 MB). For larger XML documents, use the "pass by reference" feature of the DataStage Connector stages.

If the individual XML documents that are processed are all consistently close to the maximum size, no further tuning is necessary. However, if the XML documents vary significantly in size, with most of them less than half of the maximum size, you might try further tuning. In this situation, set APT_OLD_BOUNDED_LENGTH=`True`. This value forces both in-memory and physical data set storage to manage all string values by their actual length and not their maximum length. This value uses more CPU resources, but it reduces memory and I/O.

10.6.2 Passing XML by reference as a large object column

The Information Server Connector framework can create a reference to a large data value, and then pass this reference, rather than the actual value, within each row. This capability allows the size of the row to remain small, so that it is unnecessary to increase the transport block size. When this field needs to be accessed within any downstream Connector stage, the data value is automatically retrieved from the original source.

Important: This feature only can be used with Connector stages with LOB column support, which includes the XML Connector.

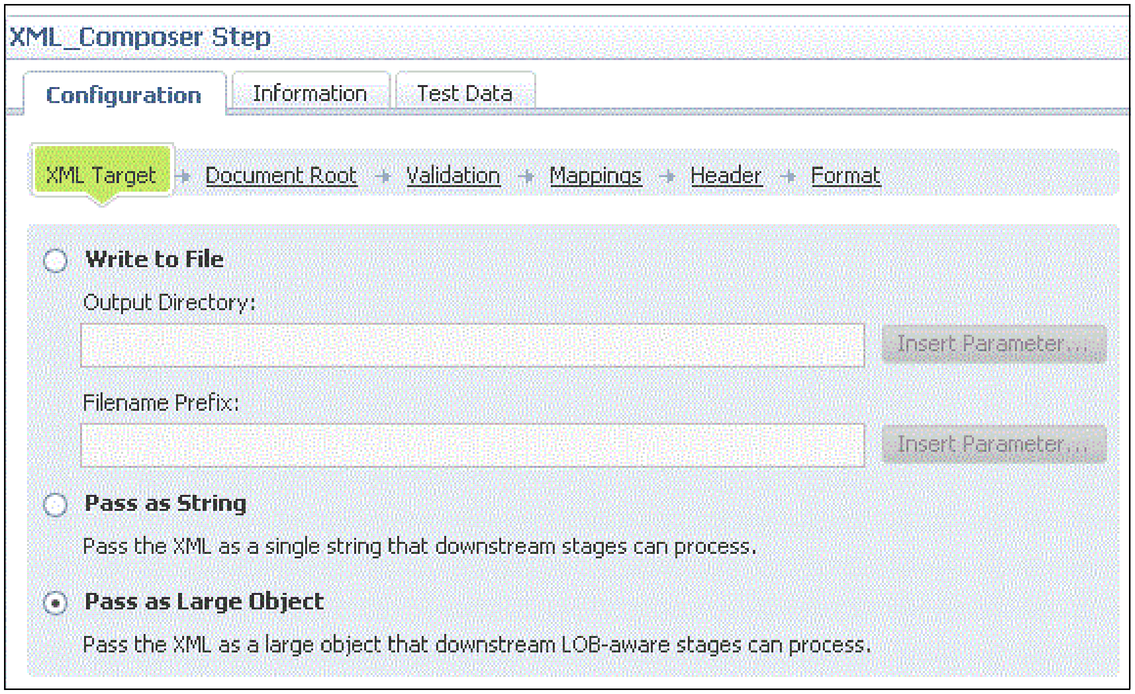
If you attempt to access a LOB column with a non-Connector stage, it is only able to see the reference string and not the actual content to which it refers. For example, you cannot use the Transformer stage to modify the XML that is passed by reference as a LOB column, because the transformer does not have LOB column support. It sees the small reference ID string only. Appendix A, “Supported type mappings” on page 363 contains a list of Connector stages, which have LOB column pass by reference support, that are available at the time of writing this book.

Important: Using references adds overhead, because the connector must retrieve the data from the source each time that you access it.

For this reason, it is important to determine in advance the size of the XML documents that are passed through the job, and design it to use LOB columns or pass data in-line as appropriate:

- LOB columns with the Composer step

  The Composer step supports writing data into a LOB column. In the XML Stage assembly, open the Composer step. As shown in Figure 10-28 on page 323, select **Pass as Large Object** on the XML Target properties page.


XML_Composer Step

Configuration | Information | Test Data

XML Target → Document Root → Validation → Mappings → Header → Format

( ) **Write to File**

Output Directory:

Filename Prefix:

( ) **Pass as String**

Pass the XML as a single string that downstream stages can process.

(●) **Pass as Large Object**

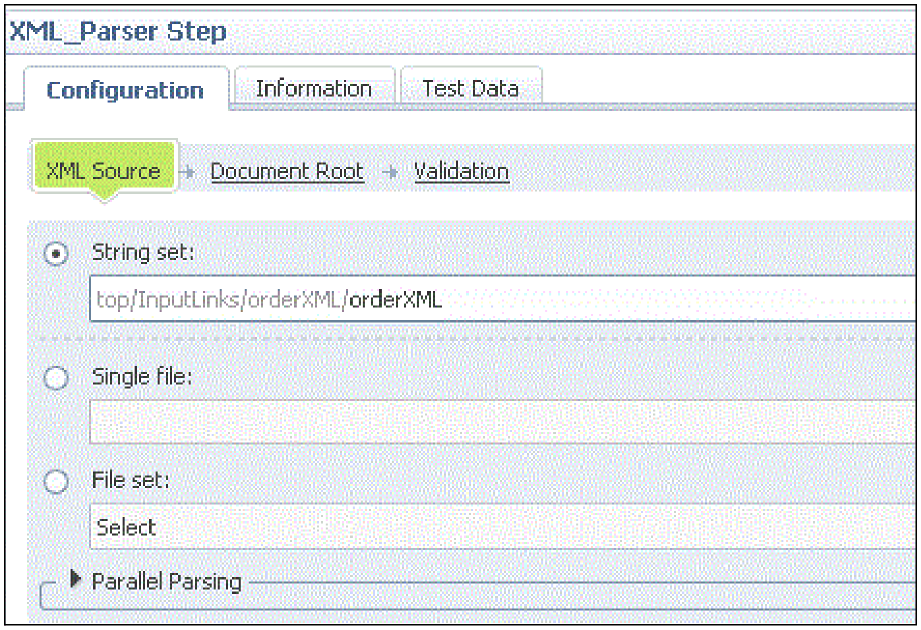
Pass the XML as a large object that downstream LOB-aware stages can process.

Figure 10-28 Output composed XML content as a LOB reference column

At the time of writing this book, the XML Connector stage uses physical disk files stored in the scratch disk location to store XML data that is passed by LOB reference. The downstream Connector stage, such as the DB2 Connector, retrieves the data from the file at access time.

- LOB columns with the Parser step

  The Parser step supports reading data from a LOB reference column. No special option is required, you use the String set option for XML Source in the XML Parser Step Configuration tab, as shown in Figure 10-29 on page 324. The XML Stage automatically detects if the column contains an in-line or pass-by-reference value. The XML Stage takes the appropriate action to retrieve the LOB data, if necessary.


XML_Parser Step

Configuration | Information | Test Data

XML Source → Document Root → Validation

(●) String set:
top/InputLinks/orderXML/orderXML

( ) Single file:

( ) File set:
Select

Parallel Parsing

Figure 10-29 LOB reference columns are handled by using string set XML source

Important: The Parser step cannot write a LOB reference column. All parser output is passed by value, even when the Chunk feature is used.

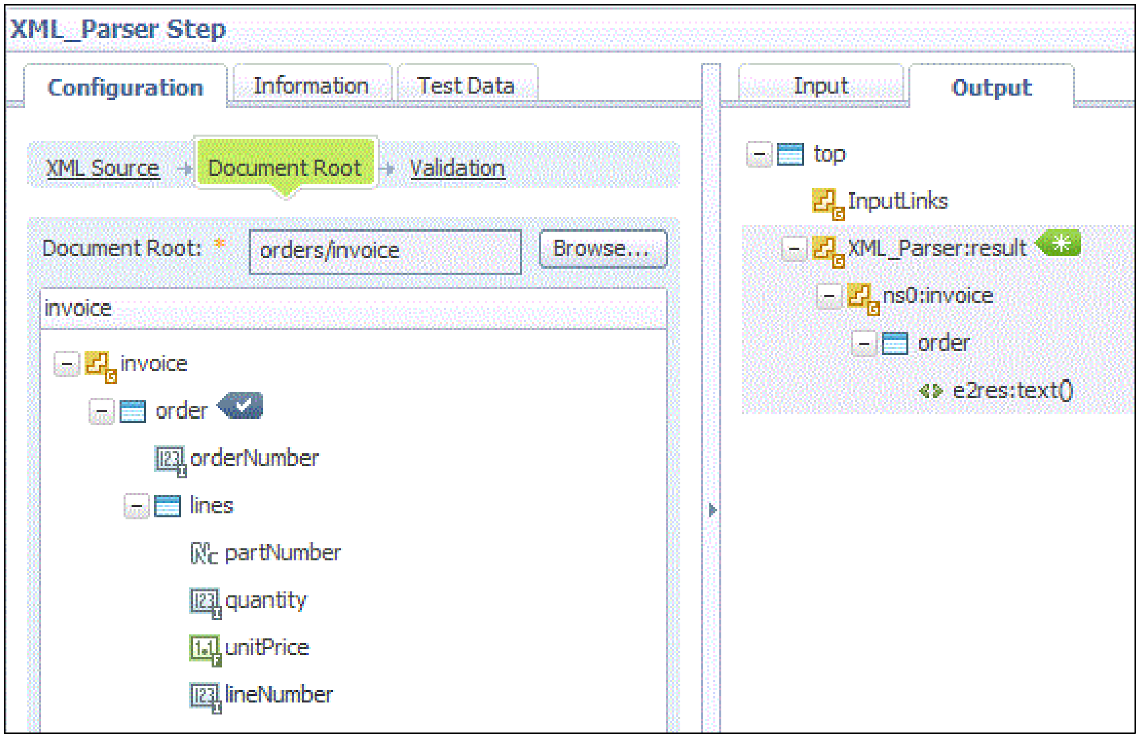
If it is necessary to extract a large chunk of XML data with the Parser step, and pass it externally to DataStage by reference, you need to add a Composer step to manage this chunk by writing the LOB reference.

10.6.3 Managing large XML schemas with LOB references

In Chapter 7, “Consuming XML documents” on page 147, we described how to use chunking to handle large schemas, and how we used a second Parser step within the same assembly to process the chunk. But what if a large chunk of XML data needs to be sent externally to the XML Stage, for example, written into a DB2 database? Large chunks require the use of a LOB reference column; however, the Parser step cannot write a LOB reference column. It is possible to use the Composer step as a solution.

Step 1: Configure Parser step to create an XML chunk

The Parser step is configured to produce a chunk of XML. It is depicted in Figure 10-30 on page 325.



Document Root: * orders/invoice

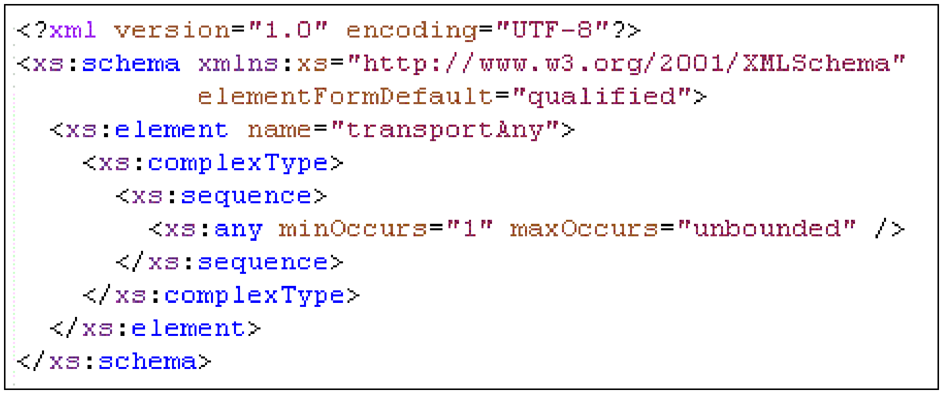
```
invoice
- invoice
  - order
      orderNumber
    - lines
        partNumber
        quantity
        unitPrice
        lineNumber
```

```
- top
    InputLinks
  - XML_Parser:result
    - ns0:invoice
      - order
          e2res:text()
```

Figure 10-30 Configure the Parser to produce a chunk of XML

Step 2: Create a generic XSD schema to hold the XML chunk

The Composer requires a document root to compose the XML. However, we cannot expand the chunk into its elements. We want to send the chunk as a string with the least amount of overhead possible. We can create a generic XML Schema Definition (XSD) schema that accepts a list of any XML elements. Figure 10-31 on page 326 contains the source for an example XSD schema that accepts a list of any XML elements.



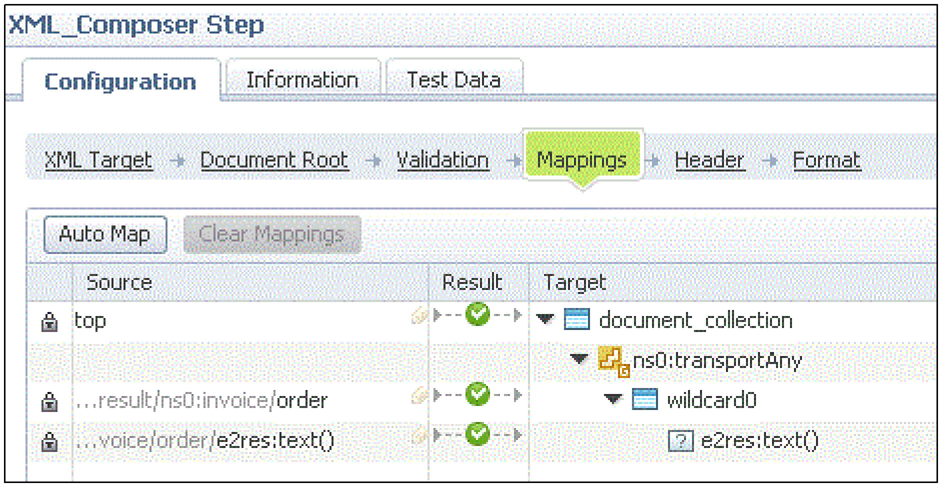
```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified">
  <xs:element name="transportAny">
    <xs:complexType>
      <xs:sequence>
        <xs:any minOccurs="1" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 10-31   Generic XML schema that contains a list of any element

The `<xs:any>` syntax allows the schema to be extensible beyond its defined elements, by allowing any XML element to be contained within it.

Step 3: Add a Composer step to write the LOB reference

The Composer step is configured to have the transportAny element from our generic schema as the document root. With this design, we can map our chunk without parsing it further, as shown in Figure 10-32.



Source	Result	Target
top	✓	document_collection
		nsO:transportAny
...result/nsO:invoice/order	✓	wildcard0
...voice/order/e2res:text()	✓	e2res:text()

Figure 10-32   Mapping Composer step by using a generic schema

The Composer step can then write its output as a LOB reference column, which is accessible by any downstream Connector stage.

10.7 Storing XML intermediately on disk

Avoid temporarily storing or staging XML documents and large XML chunks within persistent physical files, such as data sets. Instead, write XML directly to its intended target for maximum performance. Data sets, file sets, and similar stages do not have LOB column support. Therefore, XML data must be passed in-line. In situations where you must pass XML data in line, ensure that you set the length of the column to be the smallest possible value that can accommodate your largest XML document. Starting with Information Server Version 8.5, DataStage, by default, reduces I/O and storage space for physical data sets by calculating and storing only the data in variable format instead of writing the entire maximum length.

Important: This behavior can be disabled with the environment variable named APT_DONT_COMPRESS_BOUNDED_FIELDS. If you currently use this variable in your DataStage project, but you are going to write large variable-length column values, such as XML, we highly advise that you *not* set this variable at the job level.

11

Developing XML solutions

DataStage can use multiple facilities to acquire and publish XML data. In addition to file-based storage, XML data can be fetched from relational databases, delivered by message queues, or consumed from Web services. Real-time and near-real-time enterprise environments can trickle feed into DataStage jobs that are deployed as service providers that use IBM InfoSphere Information Services Director. This chapter describes integration scenarios that involve Web services, message queues, and relational databases.

Many existing production deployments of DataStage use XML Pack 2.0 for DataStage for file-based and real-time data integration. This chapter describes several key differences of which developers familiar with XML Pack 2.0 need to be aware when building new solutions or migrating existing solutions with XML Transformation stage.

11.1 Integration with Web services

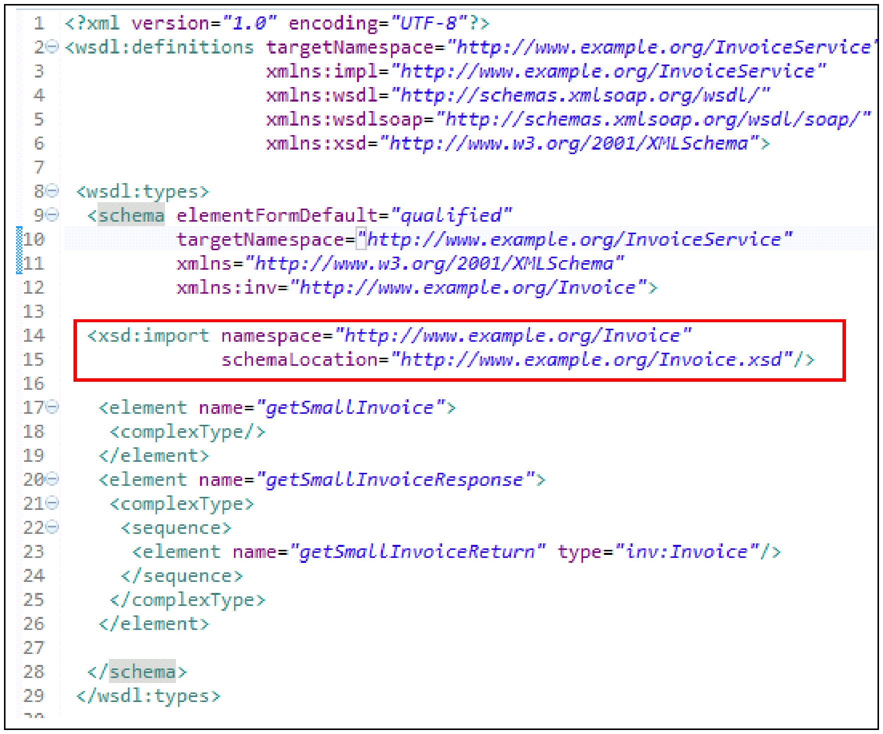
The Web Services Pack for DataStage provides stages that can read data from and write data to external Web services with InfoSphere DataStage and QualityStage jobs. The Web Services Pack supports the SOAP 1.1 binding over HTTP. The Web Services Pack can internally parse and compose simple XML structures within the SOAP header and message body of a web service request. When the SOAP message contains a complex XML structure, such as a sequence (a repeating element or group of elements), the Web Services Pack stages must be used to extract the SOAP message body as an XML document, and then use the XML Stage to parse it. The Web Services Pack also contains a Web Service MetaData Importer. The Web Service MetaData Importer can extract service and operation information from Web Service Definition Language (WSDL) files and store that information in the DataStage repository. Developers can access the information in the DataStage repository and use it to design jobs. In the following sections, we describe an example of importing a web service definition and parsing a complex SOAP message response by using the XML Stage.

11.1.1 Importing WSDL and XML schema information

A WSDL document contains both the necessary endpoint information, such as server and port number, to start the service and the metadata for the content of both the request and the response messages. SOAP is an XML-based protocol for Web services. The SOAP message is a well-formed XML document. To process the SOAP message with the XML Stage, an XML Schema Definition (XSD) schema is needed that can be imported into the schema library. There are two possibilities. Either the schema for the message body is in a separate schema document and the WSDL has a reference to it, or it is included in-line in the WSDL. We explain how to obtain an XSD file for both of these conditions in the following sections. After the XSD file is obtained, import the schema into the Schema Library Manager. To learn more about the Schema Library Manager, see Chapter 5, “Schema Library Manager” on page 113.

External XSD schema referenced in the WSDL

The following method is the simpler of the two situations that can be used to obtain the XSD file. In most cases, the XSD schema can be downloaded by using the URL in the schema location attribute from the <xsd:import> tag in the WSDL and then imported into the schema library. Figure 11-1 on page 331 is a fragment of a WSDL document that shows the <xsd:types> stanza.



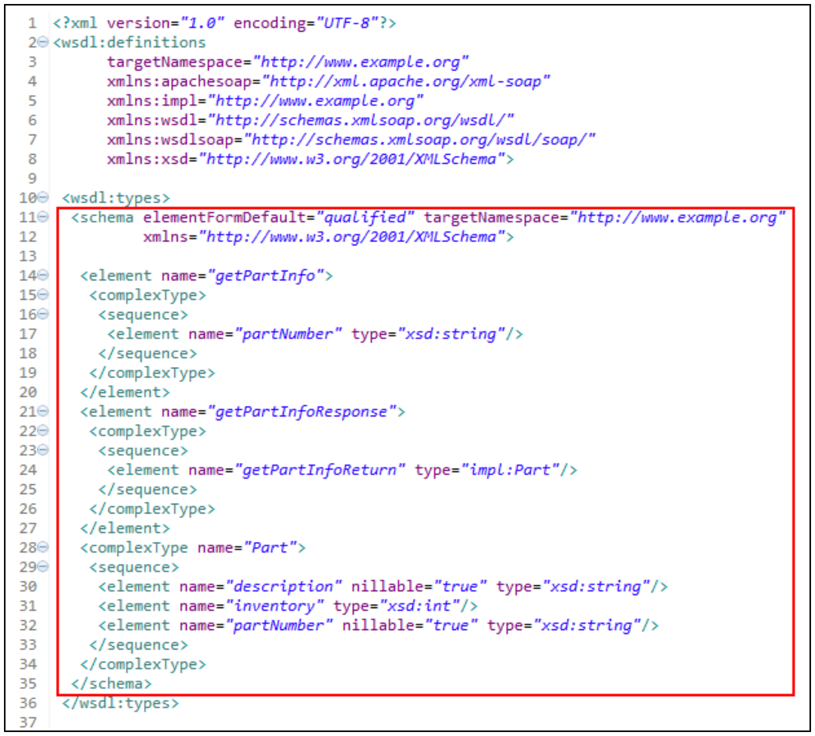
```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://www.example.org/InvoiceService"
                  xmlns:impl="http://www.example.org/InvoiceService"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <wsdl:types>
  <schema elementFormDefault="qualified"
          targetNamespace="http://www.example.org/InvoiceService"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:inv="http://www.example.org/Invoice">

  <xsd:import namespace="http://www.example.org/Invoice"
              schemaLocation="http://www.example.org/Invoice.xsd"/>

   <element name="getSmallInvoice">
    <complexType/>
   </element>
   <element name="getSmallInvoiceResponse">
    <complexType>
     <sequence>
      <element name="getSmallInvoiceReturn" type="inv:Invoice"/>
     </sequence>
    </complexType>
   </element>

  </schema>
 </wsdl:types>
```

Figure 11-1 External XSD schema referenced in a WSDL document

Figure 11-1 shows the section where the types are defined in the WSDL. We highlighted the <xsd:import> tag, and it must appear as the first element in the <xsd:schema> tag. The schemaLocation attribute of the <xsd:import> tag identifies the location of the schema file. Normally, this location is a URL location, and it can be downloaded with a web browser. Otherwise, it needs to be distributed with the WSDL file.

In-line XSD schema in a WSDL document

At the time of writing this book, the Schema Library Manager reads XSD files only, and it cannot process the WSDL directly. The schema definition needs to be manually extracted from within the WSDL and saved as an XSD file that can be imported. Figure 11-2 on page 332 contains an example WSDL file with an in-line schema. This example does not include a reference to an external XSD file. The specific contents that we need to extract are highlighted in the box.



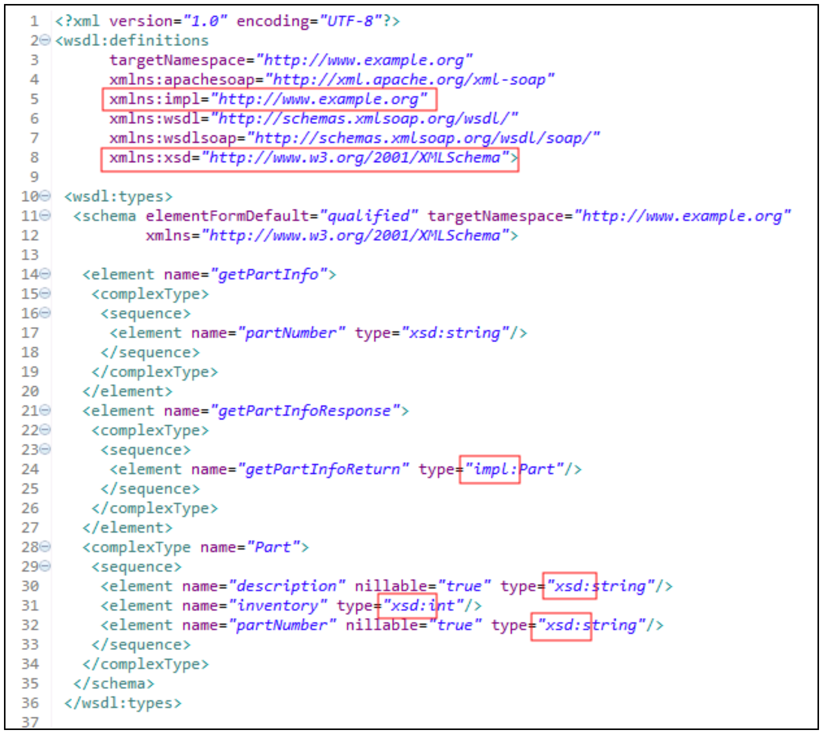
```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
      targetNamespace="http://www.example.org"
      xmlns:apachesoap="http://xml.apache.org/xml-soap"
      xmlns:impl="http://www.example.org"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
      xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <wsdl:types>
  <schema elementFormDefault="qualified" targetNamespace="http://www.example.org"
          xmlns="http://www.w3.org/2001/XMLSchema">

   <element name="getPartInfo">
    <complexType>
     <sequence>
      <element name="partNumber" type="xsd:string"/>
     </sequence>
    </complexType>
   </element>
   <element name="getPartInfoResponse">
    <complexType>
     <sequence>
      <element name="getPartInfoReturn" type="impl:Part"/>
     </sequence>
    </complexType>
   </element>
   <complexType name="Part">
    <sequence>
     <element name="description" nillable="true" type="xsd:string"/>
     <element name="inventory" type="xsd:int"/>
     <element name="partNumber" nillable="true" type="xsd:string"/>
    </sequence>
   </complexType>
  </schema>
 </wsdl:types>

```

Figure 11-2 XSD schema embedded in-line in a WSDL document

Extract the highlighted <schema> element and paste it into an empty file, and give it the `.xsd` extension. You must change a few lines. Add the standard XML declaration as the first line in the file:

```
<?xml version="1.0" encoding="UTF-8"?>
```

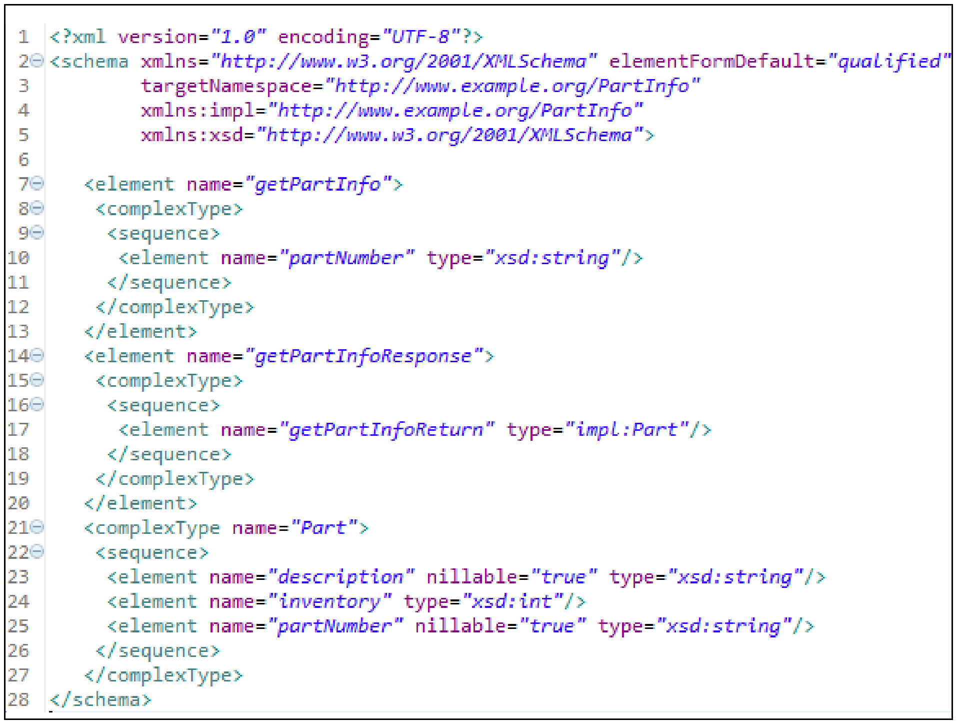
Identify all the namespace prefixes (`xsd`: and `impl`: in this example) that are used in the schema content. Look at the <wsdl:definitions> element at the top of the original WSDL file to see the attributes for xmlns:xsd= and xmlns:impl=. Copy these attributes and their assigned values into the <schema> element into the new file. Figure 11-3 on page 333 shows the highlighted namespace prefixes and attributes.



```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
      targetNamespace="http://www.example.org"
      xmlns:apachesoap="http://xml.apache.org/xml-soap"
      xmlns:impl="http://www.example.org"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
      xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <wsdl:types>
  <schema elementFormDefault="qualified" targetNamespace="http://www.example.org"
          xmlns="http://www.w3.org/2001/XMLSchema">

   <element name="getPartInfo">
    <complexType>
     <sequence>
      <element name="partNumber" type="xsd:string"/>
     </sequence>
    </complexType>
   </element>
   <element name="getPartInfoResponse">
    <complexType>
     <sequence>
      <element name="getPartInfoReturn" type="impl:Part"/>
     </sequence>
    </complexType>
   </element>
   <complexType name="Part">
    <sequence>
     <element name="description" nillable="true" type="xsd:string"/>
     <element name="inventory" type="xsd:int"/>
     <element name="partNumber" nillable="true" type="xsd:string"/>
    </sequence>
   </complexType>
  </schema>
 </wsdl:types>

```

Figure 11-3 Namespace information in the WSDL definition

After you identify all the namespaces and copy everything into a new file, save the new file with the `.xsd` extension. This file can now be imported into the Schema Manager. If you receive any errors during import, the likely cause is a missed namespace definition. Go back through these steps to verify that you located all of the namespace prefixes and defined them in the new document. Figure 11-4 on page 334 contains the complete XSD schema file after the manual extraction process.

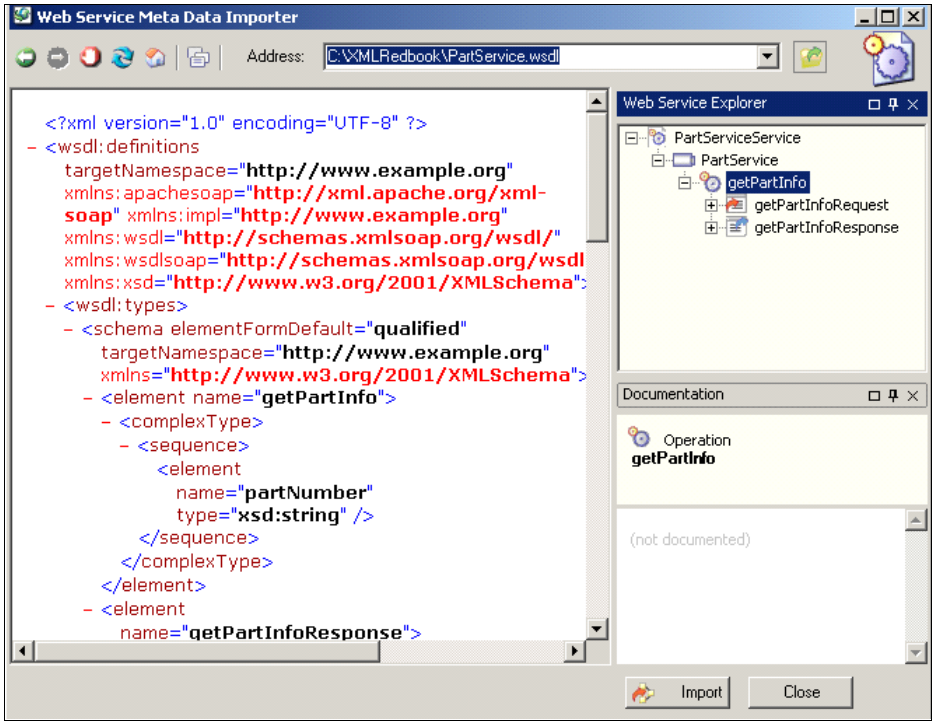


```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
        targetNamespace="http://www.example.org/PartInfo"
        xmlns:impl="http://www.example.org/PartInfo"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <element name="getPartInfo">
    <complexType>
     <sequence>
      <element name="partNumber" type="xsd:string"/>
     </sequence>
    </complexType>
   </element>
   <element name="getPartInfoResponse">
    <complexType>
     <sequence>
      <element name="getPartInfoReturn" type="impl:Part"/>
     </sequence>
    </complexType>
   </element>
   <complexType name="Part">
    <sequence>
     <element name="description" nillable="true" type="xsd:string"/>
     <element name="inventory" type="xsd:int"/>
     <element name="partNumber" nillable="true" type="xsd:string"/>
    </sequence>
   </complexType>
</schema>
```

Figure 11-4 Complete XSD document extracted in-line from a WSDL file

Importing the WSDL document with the Web Service MetaData Importer

To start the Web Service MetaData Importer, open the **DataStage Designer** and click **Import** from the main menu, expand **Table Definitions**, and click **Web Services WSDL Definitions**. Provide the URL or file path to the WSDL document to browse it. To import an operation, expand the plus sign (**+**) next to the entry for the operation in the Web Services Explorer pane. Operations appear with the gears icon next to them. Right-click the operation that you want and click **Import**. Figure 11-5 on page 335 illustrates the Web Service MetaData Importer.


Web Service Meta Data Importer
Address: C:\XMLRedbook\PartService.wsdl
Web Service Explorer
Documentation
Import
Close

```
<?xml version="1.0" encoding="UTF-8" ?>
- <wsdl:definitions
    targetNamespace="http://www.example.org"
    xmlns:apachesoap="http://xml.apache.org/xml-
    soap" xmlns:impl="http://www.example.org"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  - <wsdl:types>
    - <schema elementFormDefault="qualified"
        targetNamespace="http://www.example.org"
        xmlns="http://www.w3.org/2001/XMLSchema">
      - <element name="getPartInfo">
        - <complexType>
          - <sequence>
              <element
                name="partNumber"
                type="xsd:string" />
            </sequence>
          </complexType>
        </element>
      - <element
          name="getPartInfoResponse">
```

```
PartServiceService
  PartService
    getPartInfo
      getPartInfoRequest
      getPartInfoResponse
```

Operation
getPartInfo

(not documented)

Figure 11-5 Web Service MetaData Importer to import WSDL file

Important: You might see multiple entries for the web service in the Web Services Explorer pane. Browse and import operations that do not end in the suffix `12`. Any operations with the `12` suffix are for SOAP 1.2, which is not supported by the Web services stages at the time of writing this book.

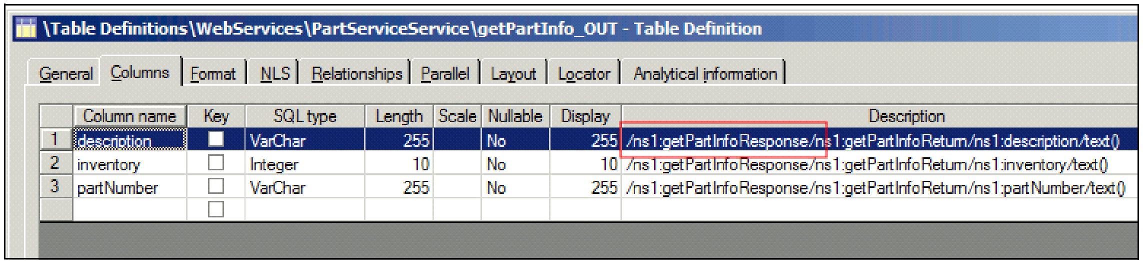
11.1.2 Designing a DataStage job

After you import the XSD file for the SOAP message of the WSDL into the Schema Library Manager, and the WSDL document is imported into the repository, the job can be constructed. Use these high-level steps:

1. Identify the SOAP message definition and the required document root.
2. Create the DataStage job with Web Service Client and XML stages.
3. Configure the Web Service Client stage to return the entire SOAP message.
4. Configure XML parsing by using the document root identified in step 1.

Step 1: Identify the SOAP message definition and the required document root

At the time of writing this book, DataStage does not have an automatic method to identify the SOAP message definition needed, or its document root. This step is performed manually by examining the imported definition that is stored in the DataStage repository. By using the DataStage Designer, browse the repository under table definitions. Expand the **WebServices** folder, and expand the entry for the web service. Locate the definition that matches the naming convention "[*Operation Name*]_OUT". Double-click to open it, and then select the **Columns** tab. Look at the Description field to identify the name of the root element in the path shown. Figure 11-6 highlights the root element.



\Table Definitions\WebServices\PartServiceService\getPartInfo_OUT - Table Definition

	Column name	Key	SQL type	Length	Scale	Nullable	Display	Description
1	description	☐	VarChar	255		No	255	/ns1:getPartInfoResponse/ns1:getPartInfoReturn/ns1:description/text()
2	inventory	☐	Integer	10		No	10	/ns1:getPartInfoResponse/ns1:getPartInfoReturn/ns1:inventory/text()
3	partNumber	☐	VarChar	255		No	255	/ns1:getPartInfoResponse/ns1:getPartInfoReturn/ns1:partNumber/text()
		☐						

Figure 11-6 Identify document root for the SOAP message

Step 2: Create the DataStage job with Web Service Client and XML stages

In the DataStage Designer, create a job and add the Web Service Client stage (WS_Client) and XML stages. Map the output of the WS_Client stage directly to the XML Stage. Figure 11-7 illustrates the required mapping between the WS_Client and the XML stages.

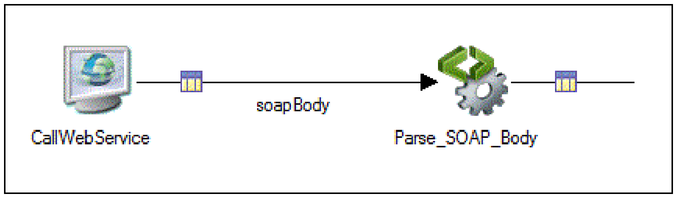
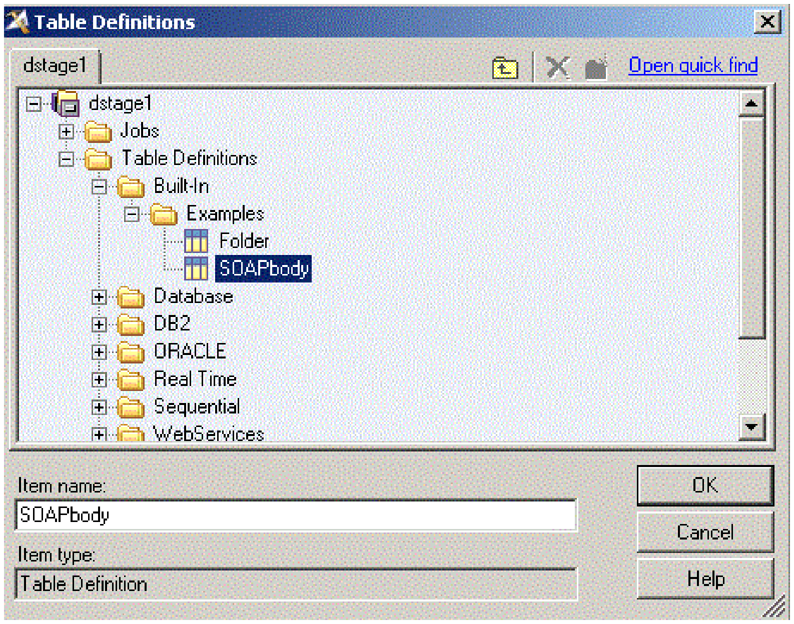


Figure 11-7 Job design fragment that shows the WS_Client and XML Stage mapping

The output link from the XML Stage can be directed to whatever downstream stage is needed, such as a Sequential File stage or a Peek stage for testing purpose.

Step 3: Configure the Web Service Client stage to return the entire SOAP message

By default, the WS_Client stage tries to parse the SOAP message XML. To pass it to the XML Stage, you need to configure a single output column to hold the output message. A built-in table definition that is specifically for this purpose, which is called `SOAPbody`, is in the DataStage repository. Open the **WS_Stage**, select the **Stage** tab, and select the **General** tab. Use **Select Web Service Operation** to browse for the operation to be used. Next, go to the **Output** tab, and then, go to the **Columns** tab. Use **Load** to retrieve the SOAPbody column definition from the DataStage repository, as shown in Figure 11-8.

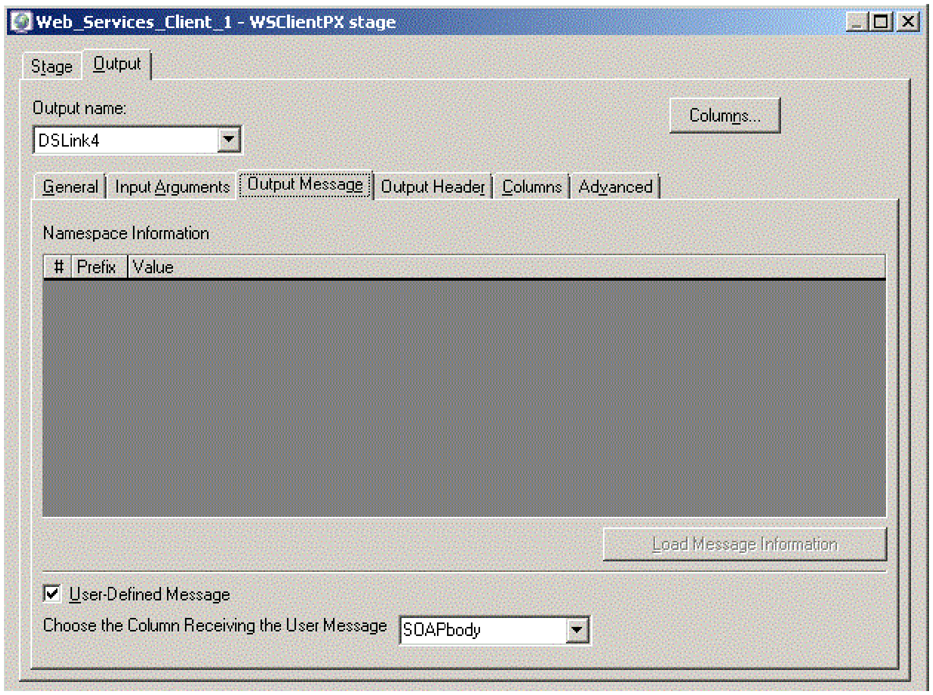


Item name: SOAPbody

Item type: Table Definition

Figure 11-8 Load the SOAPbody column definition

Now, click the **Output Message** tab. At the bottom, select **User-defined Message**, and select the **SOAPbody** column to receive the user message, as shown in Figure 11-9 on page 338. With this configuration, the WS_Client stage does not attempt to parse the SOAP message body. Instead, it loads the entire message body in-line into the SOAPbody column. Finally, complete the configuration of the WS_Client stage as required for the web service operation. For example, use the Input Arguments tab to define input parameter values for the operation that is called.



☑ User-Defined Message

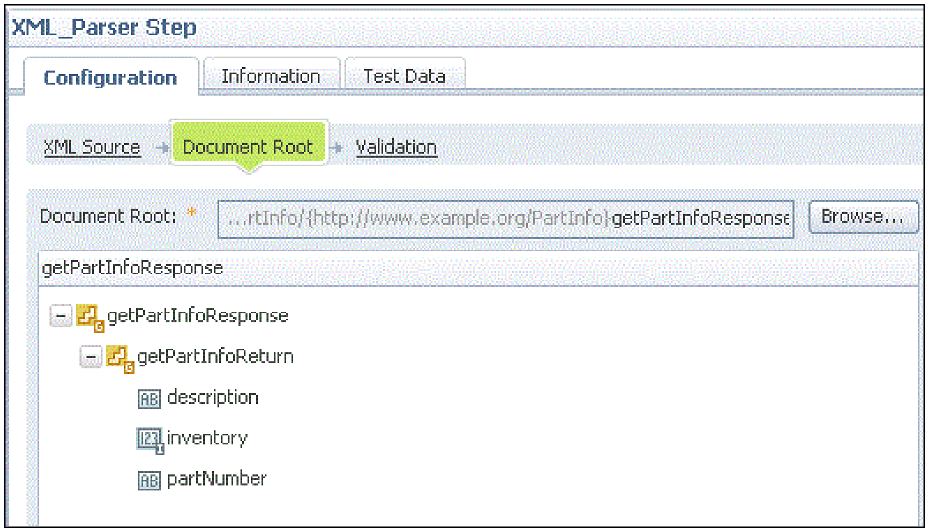
Choose the Column Receiving the User Message SOAPbody

Figure 11-9 Configure SOAPbody to receive the entire SOAP message

Important: The Web Service Pack stages do not have large object (LOB) column (pass by reference) support. The data in the SOAPbody column is passed in-line as part of the record, which means that there are limits and performance implications. For more information about passing XML data in-line, refer to Chapter 10, "DataStage performance and XML" on page 291.

Step 4: Configure the XML Stage parsing by using the document root from step 1

In the Assembly of the XML Stage, add a Parser step. Configure it to use **String Set** as the XML Source, and select **SOAPbody** as the input column. For the document root, browse to the XSD schema that was imported in the previous section, and select the root element that was identified in step 1, as illustrated in Figure 11-10 on page 339.


XML_Parser Step

Configuration | Information | Test Data

XML Source → Document Root → Validation

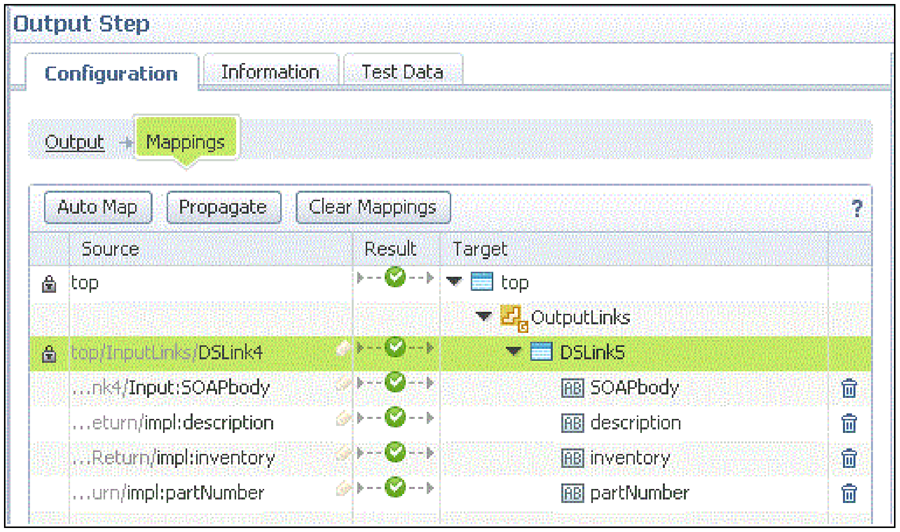
Document Root: * ...rtInfo/{http://www.example.org/PartInfo}getPartInfoResponse Browse...

getPartInfoResponse

- getPartInfoResponse
  - getPartInfoReturn
    - description
    - inventory
    - partNumber

Figure 11-10 Configure document root for parsing SOAP message body

In the Output step, configure the mapping to obtain the list and element data that is needed, as shown in Figure 11-11.


Output Step

Configuration | Information | Test Data

Output → Mappings

Auto Map | Propagate | Clear Mappings

Source	Result	Target
top	✓	top
		OutputLinks
top/InputLinks/DSLink4	✓	DSLink5
...nk4/Input:SOAPbody	✓	SOAPbody
...eturn/impl:description	✓	description
...Return/impl:inventory	✓	inventory
...urn/impl:partNumber	✓	partNumber

Figure 11-11 Map output data from the SOAP message body

11.2 Information Services Director: DataStage jobs as services

When DataStage jobs are published as providers for services, you must address a few additional job design constraints. See 3.8 “Publishing jobs as services” on page 89 for a detailed description of designing jobs for service deployment. One of the most important concepts is *request-based processing*. The submission of a request can consist of a single row or multiple rows of data. Also, you can have a single input row that has a column that contains XML data that is parsed into multiple rows. In these situations, DataStage must be able to determine which rows belong to which request. To determine which rows belong to which request, DataStage inserts a marker in the job data stream after the last row for each request. This marker is commonly referred to as the *end-of-wave marker.*

11.2.1 Understanding how end-of-wave works

Records move through the DataStage job in a first-in, first-out order. DataStage places a marker in the stream of data, after the last row of each group of rows. By watching for this marker, DataStage stages can act to ensure that each group of records is treated independently, mimicking what happens if the job runs with only those rows as input. The end-of-wave marker is not actually part of the data, and it does not get written to any external target. However, it is visible in the Peek stage for debugging purposes, and it is shown as the following sequence of characters: “`:-- `”. You can obtain additional information about end-of-wave behavior for real-time jobs in 3.7.2 “End-of-wave” on page 80.

When working with XML documents inside of a job that is deployed as a service, the end-of-wave construct is critical. If a group of rows is consumed to compose an XML document, it must stop when the end-of-wave marker is reached. When parsing incoming XML documents, it must output the end-of-wave marker at the end of each input document that is consumed.

Important: DataStage parallel jobs must be configured to use only a single partition when they are used as providers for Information Services Director.

At the time of writing this book, the DataStage parallel engine framework cannot automatically force stages to run on a single partition when the job is deployed as a service provider. It is the responsibility of the developer to ensure that the job is configured to use a single-node configuration file at all times. Failure to configure the job to use a single-node configuration file at all times results in potentially incorrect output from the service operation.

11.2.2 Designing and testing DataStage jobs with end-of-wave

Test-modify development cycles can be cumbersome for a job that must be deployed as a service. Before going into production, always use a test harness with the same type of binding that you use in production. A simpler solution that is effective for use during the development cycle is to use a stage called *Wave Generator*, which is included with DataStage. Wave Generator is on the Processing tab of the stage palette in the DataStage Designer. You can use this stage in a classic non-service mode DataStage job to simulate the end-of-wave that is produced in real-time mode. Then, you can verify in advance that the jobs work as designed. An example is shown in Figure 11-12.

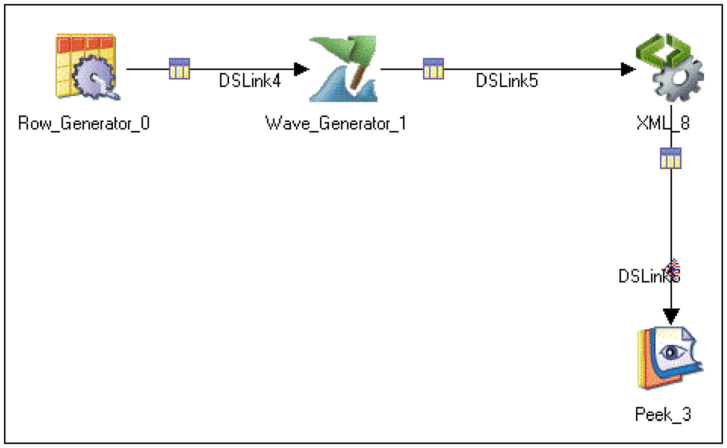
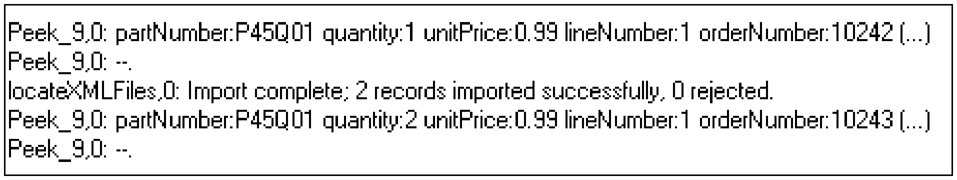


Figure 11-12 Non-Information Services Director DataStage Wave Generator testing job

You do not need to perform any specific configuration in the XML Connector stage to handle the end-of-wave, it is handled automatically. Insert Peek stages at various points in the job design so that you can observe the end-of-wave marker and verify that the correct output is generated.

11.2.3 Expected Peek stage output when parsing a document

The Peek stage shows all of the relational records that are parsed from the first document, followed by the end-of-wave marker, followed by all the records from the next document, and so on. See Figure 11-13 on page 342 for an example of how the output appears in the DataStage Director log.

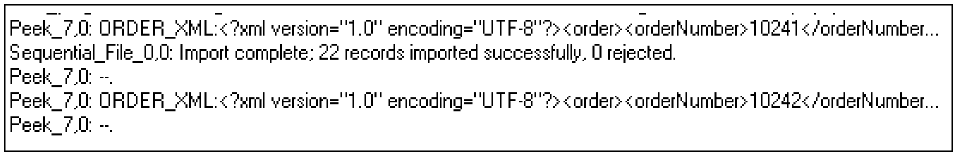


```
Peek_9,0: partNumber:P45Q01 quantity:1 unitPrice:0.99 lineNumber:1 orderNumber:10242 (...)
Peek_9,0: --.
locateXMLFiles,0: Import complete; 2 records imported successfully, 0 rejected.
Peek_9,0: partNumber:P45Q01 quantity:2 unitPrice:0.99 lineNumber:1 orderNumber:10243 (...)
Peek_9,0: --.
```

Figure 11-13 Peek results from parsing with end-of-wave

11.2.4 Expected Peek stage output when composing a document

The Peek stage shows the first entire XML document, whose contents consist of only the relational records from the first request, followed by the end-of-wave marker, followed by the next document, and so on. See Figure 11-14 for an example of this output in the DataStage Director log.



```
Peek_7,0: ORDER_XML:<?xml version="1.0" encoding="UTF-8"?><order><orderNumber>10241</orderNumber...
Sequential_File_0,0: Import complete; 22 records imported successfully, 0 rejected.
Peek_7,0: --.
Peek_7,0: ORDER_XML:<?xml version="1.0" encoding="UTF-8"?><order><orderNumber>10242</orderNumber...
Peek_7,0: --.
```

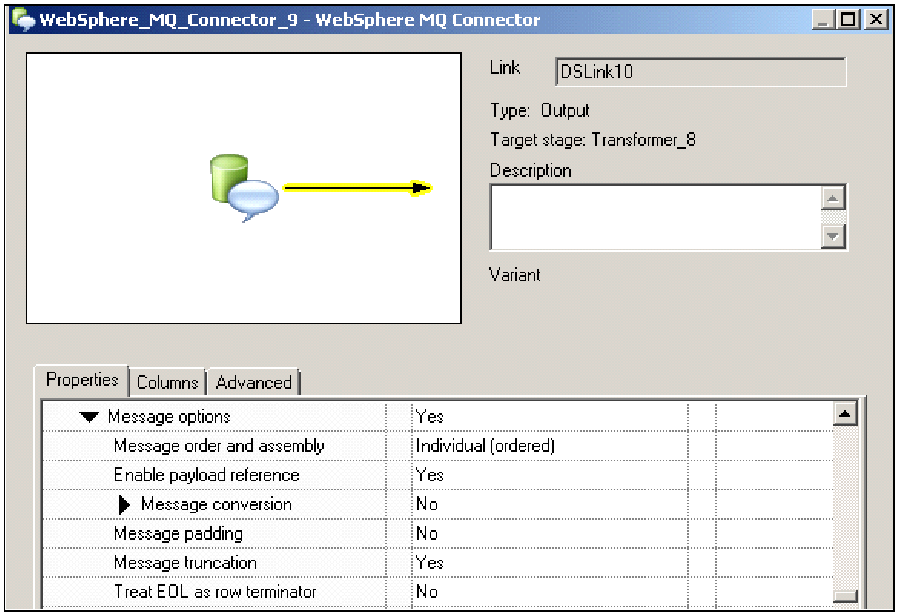
Figure 11-14 Peek results from composing with end-of-wave

Important: If your XML Stage has multiple output links, the end-of-wave marker is included in the output. The marker follows the records from each document on each link.

11.3 Event-driven processing with IBM WebSphere MQ

In addition to Information Services Director, you can also implement real-time and near-real-time applications by using IBM WebSphere MQ and InfoSphere DataStage. The DataStage MQ Connector stages can browse and write to WebSphere MQ message queues, Java Message Service (JMS) queues, and topics. When used in conjunction with the Distributed Transaction Stage, a unit of work (XA transaction) can be managed between multiple MQ queues or a supported relational database. For more information about transactional event-driven processing, see 3.8 “Publishing jobs as services” on page 89, which discusses real-time job designs in detail.

You can use XML as the payload for MQ messages as an architecture for event-driven applications. The MQ Connector stage supports LOB reference columns when used both as a source and a target. This support is useful for processing larger XML documents that cannot be stored in-line. Figure 11-15 illustrates an MQ Connector stage that is used as a source to browse a message queue and retrieve XML documents.


WebSphere_MQ_Connector_9 - WebSphere MQ Connector

Link: DSLink10

Type: Output

Target stage: Transformer_8

Description

Variant

Message options	Yes
Message order and assembly	Individual (ordered)
Enable payload reference	Yes
Message conversion	No
Message padding	No
Message truncation	Yes
Treat EOL as row terminator	No

Figure 11-15 MQ Connector stage that is configured to read XML documents

The Message options parameter is set to **Yes**. Set this parameter first so that it enables other important parameters:

- *Message truncation* must always be set to Yes with XML documents. If this parameter is set to No, and the message on the queue is larger than the maximum length defined for the column, it creates multiple output records each with part of the document. These records can cause undesirable behavior in the rest of the job. It is better to truncate the message. Then, allow the validation capabilities of the XML Stage to handle an incomplete document scenario.
- *Message padding* must always be set to No with XML documents. This parameter is important when LOB references are not used, and the data is passed in-line. White space in an XML document is ignored, but these bytes still must be stored when moving the document. Because not every message is likely to be close to the maximum size, you waste a substantial amount of resources by transferring irrelevant white space characters if this option is set to Yes.
- *Treat EOL as row terminator* must always be set to No with XML documents. If this parameter is incorrectly set to Yes, and XML contains any line

termination characters, the MQ Connector stage produces multiple output rows, each with a chunk of incomplete XML, instead of a single record with the full XML document.

In Figure 11-15 on page 343, *Enable payload reference parameter* is set to Yes. This value is appropriate for large XML documents. It results in a LOB reference in the output column instead of the data. If for small documents, you want to pass the data in-line, set *Enable payload reference* to No.

11.4 XML data integration with relational databases

It is a preferred practice to use the DataStage Connector stages for all database connectivity. The Connectors have the richest capabilities for handling Unicode data and large-sized columns, both of which are prevalent when working with XML. Many of the major commercial database vendors include specific XML processing functionality in recent versions. DataStage can integrate with these features by accessing the raw data type in binary or string format, by using LOB handling, and by using XML query functionality that is available through SQL statements. In this document, we introduce the XML capabilities that are available at the time of writing this book for IBM DB2 Version 9.7 and Oracle Database 11g (11.1). We describe any specifics of which you need to be aware when integrating these types in DataStage.

11.4.1 IBM DB2 V9 and DB2 pureXML

IBM DB2 V9 introduced extended functionality, which is called *pureXML*. This function is called pureXML because DB2 can store XML in a native, hierarchical format without breaking it down into relational components. Columns in DB2 tables that are declared with the XML data type are stored in a proprietary binary hierarchical format. This format provides faster and more feature-rich indexing and XML query capabilities. You can also store XML documents in a pure character LOB (CLOB) column, which offers the fastest I/O performance if your operations are largely limited to reading or writing the entire document. DataStage supports both types.

The DB2 Connector can read and write to DB2 XML data type columns without any special handling requirements. DB2 automatically converts between strings and its internal binary storage type. If a CLOB column is used, no conversion is required, because the XML is stored byte for byte in string format. DataStage adds the capability to validate the XML before loading it into the database. This capability eliminates the overhead of configuring an XML schema in the database for validation, which impacts load performance. It also allows reject handling to be managed in the ETL process.

DB2 Connector XML query support

The DB2 Connector supports full and partial XML document queries by using SQL and SQL with XML extensions (SQL/XML). The result can be retrieved either in-line or by using LOB reference columns:

- Pure SQL queries can return the complete contents of the column only, as depicted in Example 11-1.

  Example 11-1 SQL query that retrieves an XML column

  ```
  select xml_order from orders_table
  ```

- With XML query extensions, you can return a specific piece of XML from within the column or reference a value of a specific element in a `where clause` predicate. Example 11-2 shows this type of a query.

  Example 11-2 SQL/XML query

  ```
  select xmlquery('$c/Order/Customer/Address' passing xml_order as
  “c”) from orders_table
  ```

> **Important:** The DB2 Connector needs to be able to reconcile each object in the query result with a column defined in the stage. When the developer uses expressions, such as **`xmlquery`**, the developer must either use an `'as'` clause in the SQL query to assign a name to the expression or use the expression prefix feature in the DB2 Connector stage.

It is also possible to use the XQuery language instead of SQL in the DB2 Connector. XQuery restricts the result to be retrieved as an in-line value; the DB2 Connector is unable to use a LOB reference column in this case.

11.4.2 Oracle 11g and Oracle XML DB

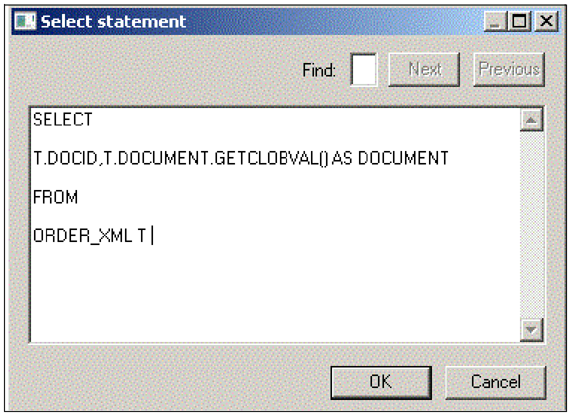
Oracle identifies its XML features by using the terminology XML DB. The features are similar to the pureXML features for IBM DB2. Oracle 11g (11.1.0) introduced new capabilities, such a binary storage type, and new XML indexing capabilities. The Oracle data type XMLType allows the Oracle database to be aware that XML is stored in the column and enables XML-specific features. With Oracle XML DB 11.1, the XMLType has three storage types: Structured (object relational), Unstructured (CLOB), and Unstructured Binary XML. See “Related publications” on page 375 for links to Oracle-published reference documents to learn more about choosing the storage type that is best for your type of application.

Retrieval: Regardless of the underlying storage type that you use for the XMLType column, you can bring back the data at query time in multiple ways.

If the data is small enough to fit in the standard Oracle string buffer, you can bring it back as *varchar*. Otherwise, you need to bring it back by converting it to a CLOB or binary LOB (BLOB). Check your Oracle database documentation to see the size of the Oracle string buffer for your specific release.

Reading XMLType data as CLOB

For data that is larger than the standard string buffer, you must retrieve it by converting to an Oracle CLOB type. Use the getClobVal() function in the SQL statement. DataStage can either store the CLOB in-line, or pass by reference. See Chapter 10, "DataStage performance and XML" on page 291 for more information about whether to use pass by reference or passing in-line. With the getClobVal() function, you must also define table and column aliases in the statement. Oracle requires the table alias as part of its SQL syntax. The column alias is required by the Connector stage, so that it can reconcile the objects in the SQL result against the objects defined on the link. Figure 11-16 illustrates how to compose the SQL statement in the Oracle connector.



```
SELECT

T.DOCID,T.DOCUMENT.GETCLOBVAL() AS DOCUMENT

FROM

ORDER_XML T
```

Figure 11-16 Retrieve XMLType object as a CLOB

The following examples show incorrectly formed queries. The query in Example 11-3 is incorrect. It contains the column alias, but it is missing the table alias.

Example 11-3 Query without table alias

```
select docid, document as document from order_xml;
```

The query in Example 11-4 is incorrect. It contains the table alias, but it is missing the column alias.

Example 11-4 Query with table alias but missing column alias

```
select t.docid, t.document.getClobVal() from order_xml t;
```

The query must use both alias types, as shown in Figure 11-16 on page 346, or a runtime error message occurs.

Using XPATH in an SQL query

If you use XPATH statements as part of the SQL query, for example, by using the extract() function to retrieve a specific portion of a document, you must also use both a column alias and a table alias. Example 11-4 shows an example SQL statement that extracts the first order in the invoice XML document.

Example 11-5 Query that extracts the first order from an invoice XML document

```
select t.docid, t.document.extract('/invoice/order[1]') as document
from order_xml t;
```

Inserting or updating XMLType columns

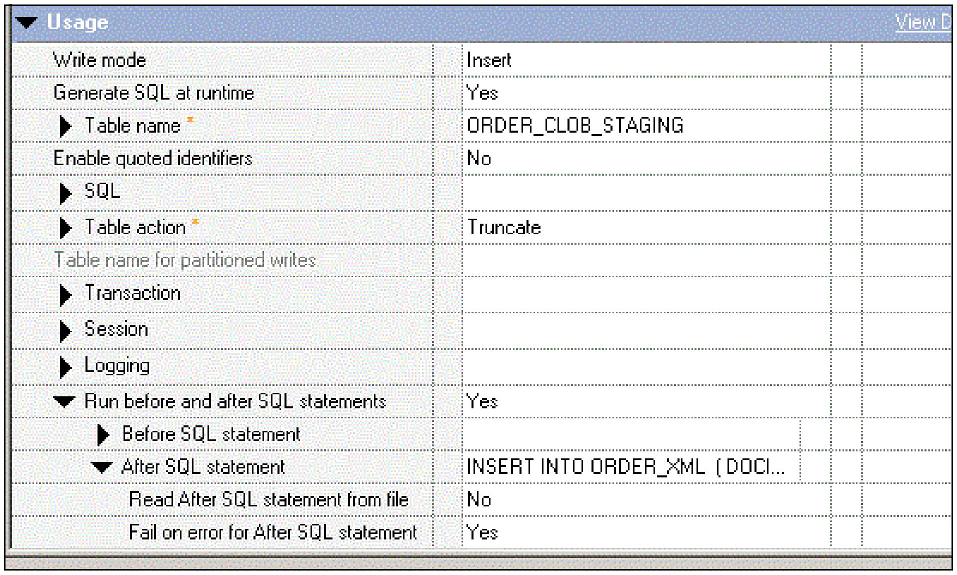
At the time of writing this book, the DataStage Oracle Connector is limited to inserting or updating 4,000 bytes or less directly into an XMLType column, regardless of the storage type. This limitation is imposed by the application programming interface (API) that DataStage uses to transfer the data to Oracle. The examples are solutions to this problem.

Solution 1: Use the ODBC Connector

The Open Database Connectivity (ODBC) Connector uses the DataDirect Wire Protocol ODBC driver, which is included with the software. Both LOB reference and in-line data transfer methods are supported, and they are not limited to 4,000 bytes. The ODBC Connector is the preferred approach to write large XML documents into Oracle.

Solution 2: Implement a staging table

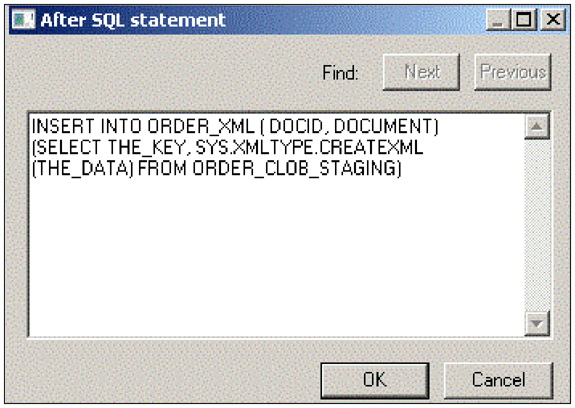
If you cannot use the ODBC Connector, you can use the Oracle Connector to implement a staging table. This solution involves creating a table in Oracle that uses a pure CLOB column instead of an XMLType column. The Oracle Connector is configured to stage the data by inserting the pure CLOB column into the table first. An `After SQL Statement` is used to insert or update the table in the database that contains the XMLType column. Figure 11-17 on page 348 illustrates the configuration of the Oracle Connector properties to perform a staged insert.



Usage	
Write mode	Insert
Generate SQL at runtime	Yes
▶ Table name *	ORDER_CLOB_STAGING
Enable quoted identifiers	No
▶ SQL	
▶ Table action *	Truncate
Table name for partitioned writes	
▶ Transaction	
▶ Session	
▶ Logging	
▼ Run before and after SQL statements	Yes
▶ Before SQL statement	
▼ After SQL statement	INSERT INTO ORDER_XML (DOCI...
Read After SQL statement from file	No
Fail on error for After SQL statement	Yes

Figure 11-17 Oracle Connector configured for a staged insert of a large XML document

The full content of the `After SQL statement` property is shown in Figure 11-18. This statement inserts the staged data into the final table that contains the XMLType column.



```
INSERT INTO ORDER_XML ( DOCID, DOCUMENT)
(SELECT THE_KEY, SYS.XMLTYPE.CREATEXML
(THE_DATA) FROM ORDER_CLOB_STAGING)
```

Figure 11-18 Insert staged CLOB data into XMLType

The job developer can customize the SYS.XMLTYPE.CREATEXML function. It can be customized with various arguments based the storage type. It can be customized with various arguments based on the schema that is defined for the column in the database.

11.4.3 Performance considerations

For most databases, the highest throughput is obtained by writing directly to string or binary storage types, which do not perform any conversion or XML-related processing. Specialty XML data types are advantageous if you actively query, modify, or index the XML data when it is stored in the database. For more details about planning your database architecture with XML, see "Related publications" on page 375 for links to additional resources.

11.5 Migrating from XML Pack 2.0 for DataStage

Deployments that use XML with DataStage for multiple years and across multiple releases can have a considerable investment in jobs that use the XML Input, XML Output, and XML Transformer stages that are part of XML Pack 2.0. These stages are still part of the most recent DataStage release, and are available for the maintenance of existing jobs and the development of new jobs. In most situations, it is preferable to develop new jobs with the new XML Stage.

We explain how to move from XML Pack 2.0 stages, provide tips, and identify pitfalls. At the time of writing this book, the Connector Migration Tool does not provide any automated conversion of XML stages in DataStage jobs. This conversion is a manual process. We describe the areas in which you can see the greatest return on your investment for refactoring existing jobs to use the new technology.

11.5.1 XML document validation and reject functionality

The XML Input, XML Output, and XML Transformer stages from XML Pack 2.0 each offered the capability to identify an output link as a reject link. This capability can capture the reject message and the document that was rejected by failed validation. The functionality differs substantially with the new XML Transformation stage. As described in Chapter 6, "Introduction to XML Stage" on page 131, the XML Transformation stage provides much finer-grained control over validation errors, and how the stage responds to them.

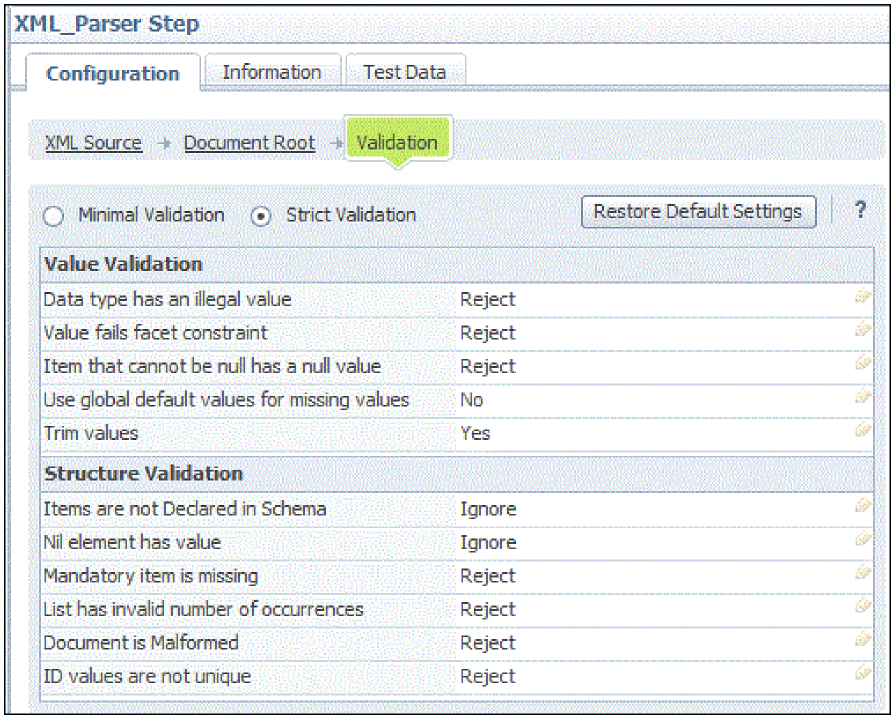
Important: With the XML Transformation stage, reject functionality is limited to the Parser step. The previous XML Pack 2.0 provided reject functionality for both composing and parsing documents.

In most cases, validation issues can be avoided when composing documents. You control the data that is sent into the XML Stage, and the data is scrubbed by the ETL process. For this reason, we focus on validation during the Parser step. If you need reject behavior when you compose an XML document, you can use the XML Pack 2.0 stages for this purpose. You need two types of information for reject processing for an XML document that fails validation during parsing. You need the error message that describes the failure and the document that failed. The XML Transformation stage can capture both pieces of information.

Important: The Parser step continues to write data ahead to the next step, regardless of whether a failure occurred.

This behavior differs from the XML Input stage, which produced no output (except for the reject link) when a validation failure occurred. In most situations, the job developer does not want the partially processed, and usually incorrect, data written to the output link. In this section, we explain how to use the Switch step in conjunction with the Parser step to implement reject functionality that is equivalent to the XML Input stage reject function.

To use reject functionality, schema validation must be enabled. Figure 11-19 on page 351 illustrates schema validation that is enabled and configured to perform a reject in most conditions. This process is highly flexible, and the job developer controls the validation.



( ) Minimal Validation [✓ Strict Validation]

Value Validation	
Data type has an illegal value	Reject
Value fails facet constraint	Reject
Item that cannot be null has a null value	Reject
Use global default values for missing values	No
Trim values	Yes
Structure Validation	
Items are not Declared in Schema	Ignore
Nil element has value	Ignore
Mandatory item is missing	Reject
List has invalid number of occurrences	Reject
Document is Malformed	Reject
ID values are not unique	Reject

Figure 11-19   Reject function enabled for validation

In this example, we send an XML instance document that contains an *X* character in a numeric field. This character causes a validation failure, because the data type has an illegal value. After Reject is selected as an outcome, the ParserStatus value is appended to the step output, as shown in Figure 11-20 on page 352.

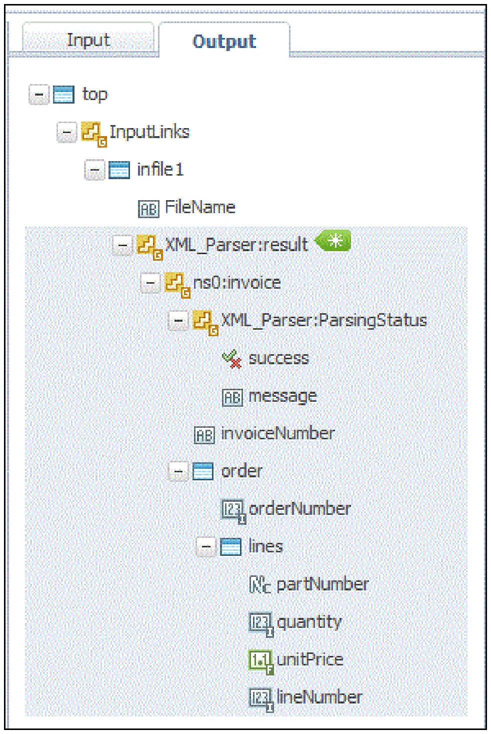
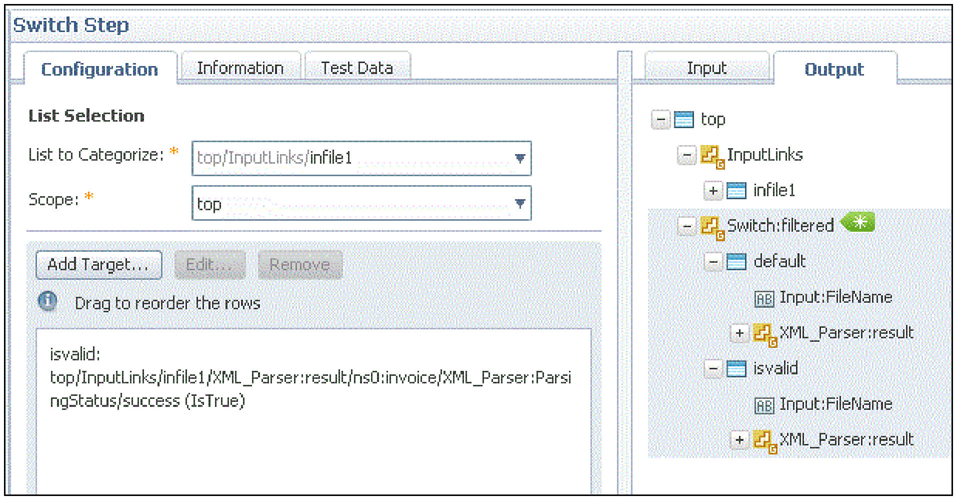


Figure 11-20 Parsing status information in the step output

To prevent output data when a validation failure occurs, we add a Switch step to after the Parser step. In the Switch Step, we add a single target, as shown in Figure 11-21 on page 353.



List Selection

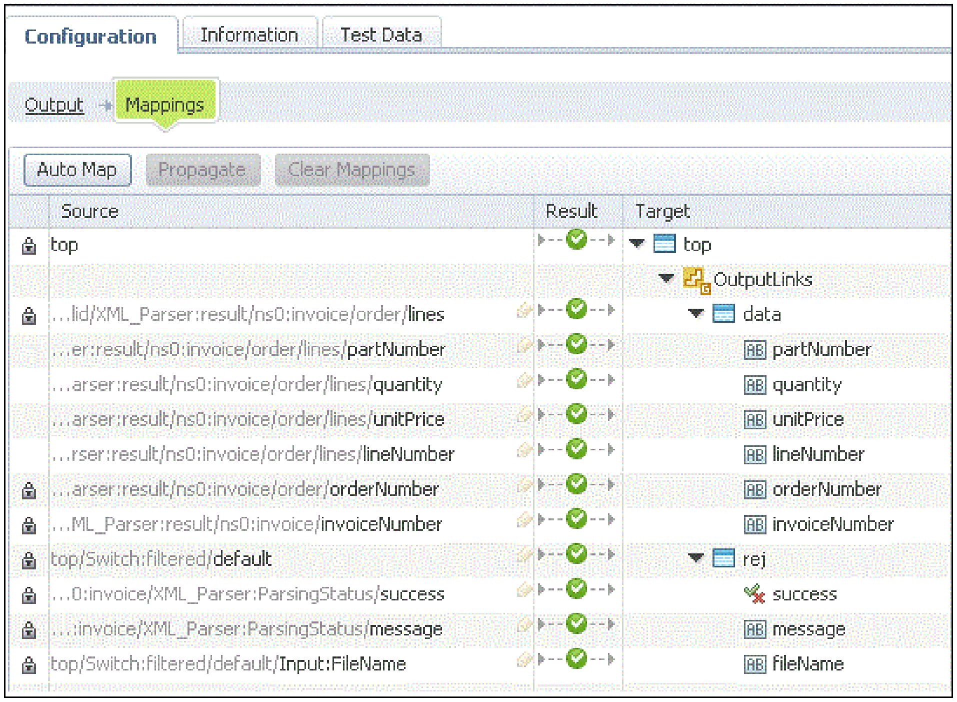
List to Categorize: * top/InputLinks/infile1

Scope: * top

isvalid:
top/InputLinks/infile1/XML_Parser:result/ns0:invoice/XML_Parser:ParsingStatus/success (IsTrue)

Figure 11-21 Using Switch step to check for parser validation results

The Switch step splits the output into two lists based on the specified target condition. The `isvalid` list contains the output when the document passes successful validation. The `default` list contains the output when the document fails validation. The input file name is included in each list, so that you can identify the processed document. Figure 11-22 on page 354 illustrates the mapping for the output of the Switch step. The output link named `rej` gets the validation failure message and the file name, which is mapped from the `default` list. The data output link receives valid parsed data from the `isvalid` list.



Source	Result	Target
top	✓	top
		OutputLinks
...lid/XML_Parser:result/ns0:invoice/order/lines	✓	data
...er:result/ns0:invoice/order/lines/partNumber	✓	partNumber
...arser:result/ns0:invoice/order/lines/quantity	✓	quantity
...arser:result/ns0:invoice/order/lines/unitPrice	✓	unitPrice
...rser:result/ns0:invoice/order/lines/lineNumber	✓	lineNumber
...arser:result/ns0:invoice/order/orderNumber	✓	orderNumber
...ML_Parser:result/ns0:invoice/invoiceNumber	✓	invoiceNumber
top/Switch:filtered/default	✓	rej
...0:invoice/XML_Parser:ParsingStatus/success	✓	success
...:invoice/XML_Parser:ParsingStatus/message	✓	message
top/Switch:filtered/default/Input:FileName	✓	fileName

Figure 11-22 Mapping primary output and reject links from the Switch step

This design pattern achieves the same result as the reject function in the XML Pack 2.0. At the same time, it provides additional control over the reject conditions during validation.

11.5.2 Design patterns for composing XML documents

In this section, we describe the major design pattern differences that DataStage developers must understand when changing jobs that compose XML documents over to the new XML Transformation stage. The stages in XML Pack 2.0 did not act based on the structure of the XSD schema. Instead, they had specific hard-coded behaviors based on the XPATH and the output mode chosen by the user at design time. The new XML Transformation stage directly depends on the XSD schema rather than fixed behaviors. In this section, we investigate three examples, one for each output mode from XML Pack 2.0. We demonstrate how the equivalent functionality is achieved with the XML Transformation stage.

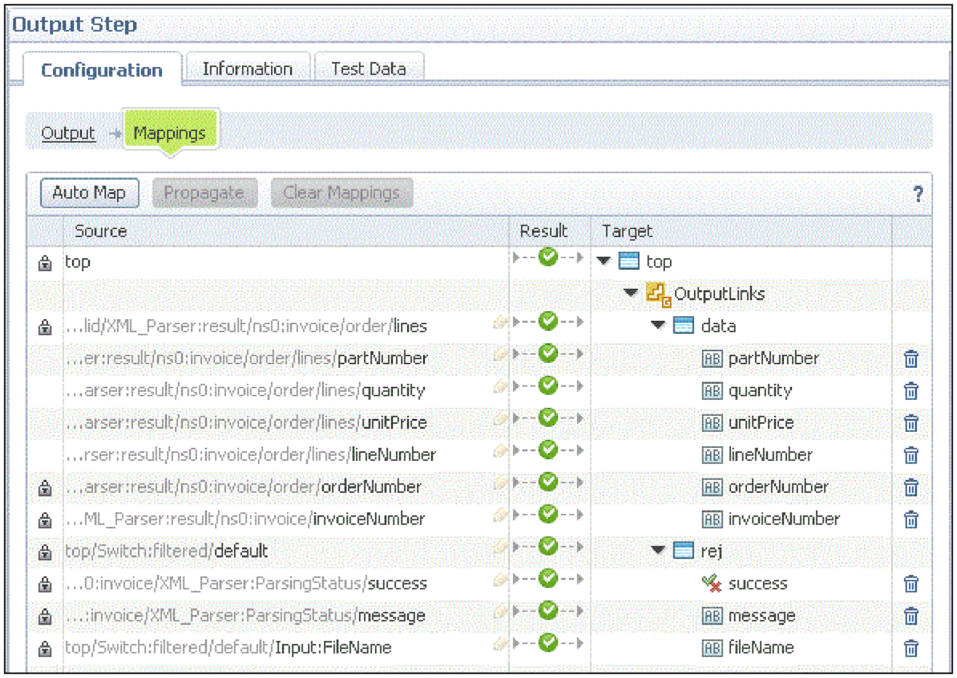
First, we quickly review XML Pack 2.0. The XMLOutput stage required document boundaries (when to finalize a document and begin a new) to be managed by a combination of the XPATH defined in the input columns and a single control named *output mode*. Output mode allowed one of three choices: Aggregate all Rows, Single row, and Trigger Column. This output mode, in combination with the XPATH for the columns, controlled the document creation.

The equivalent for these behaviors in the new XML Transformation stage is based on the use of specific steps in the assembly and the XSD. The structure defined in the XML schema must match the instance document that you are trying to create exactly. In the following scenarios, we demonstrate how to configure the correct steps in the assembly, map the correct list to the *top* of the document output structure, and select the correct scope on which to apply the step.

Aggregation step in the new XML Transformation stage: XML Pack 2.0 has an output mode named *Aggregate all rows*. Job developers must be careful not to confuse this mode with the similarly named Aggregation step in the new XML Transformation stage assembly. The Aggregation step in the assembly performs true aggregation functions, such as sum and max, on the data. It is not used to control the creation of output documents.

XML Output: Aggregate all rows output mode

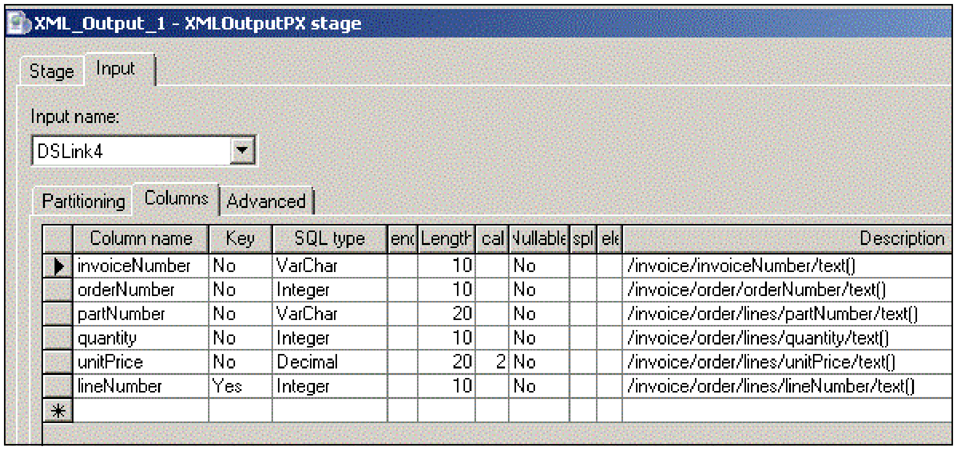
The XMLOutput stage uses a single input link only and requires a normalized relational pattern for the data that is sent. The hierarchical relationship needs to be represented in a single, flat, relational data structure on that input link. The *Aggregate all rows* option results in a single XML document as output, with the multiple input rows grouped as a repeating structure in the document.
Figure 11-23 on page 356 shows how the *Aggregate all rows* mode is selected in the XML Output stage.



Source	Result	Target
top	✓	top
		OutputLinks
…lid/XML_Parser:result/ns0:invoice/order/lines	✓	data
…er:result/ns0:invoice/order/lines/partNumber	✓	partNumber
…arser:result/ns0:invoice/order/lines/quantity	✓	quantity
…arser:result/ns0:invoice/order/lines/unitPrice	✓	unitPrice
…rser:result/ns0:invoice/order/lines/lineNumber	✓	lineNumber
…arser:result/ns0:invoice/order/orderNumber	✓	orderNumber
…ML_Parser:result/ns0:invoice/invoiceNumber	✓	invoiceNumber
top/Switch:filtered/default	✓	rej
…0:invoice/XML_Parser:ParsingStatus/success	✓	success
…:invoice/XML_Parser:ParsingStatus/message	✓	message
top/Switch:filtered/default/Input:FileName	✓	fileName

Figure 11-23 "Aggregate all rows" output mode in the XML Output stage

The following example is based on a schema for an invoice. Each invoice contains multiple orders. Each order contains multiple lines. Figure 11-24 on page 357 shows how this example invoice structure is built with XML Output. The XML Output stage automatically opens and closes parent elements based on data values. When the XML Output stage reaches the element designated as key, it produces a new element for each value of that key column. The job developer can control the sorting of the data and the column identified as key, but the rest is fixed behavior that is built into the stage.



XML_Output_1 - XMLOutputPX stage

Stage | Input

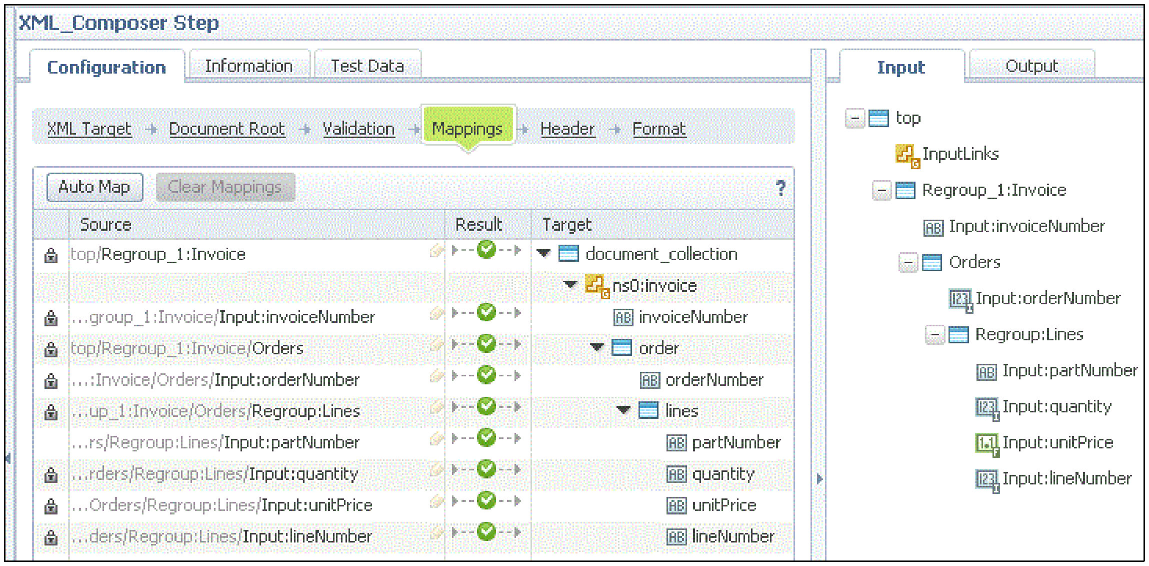
Input name: DSLink4

Partitioning | Columns | Advanced

	Column name	Key	SQL type	enc	Length	cal	Nullable	spl	ele	Description
▶	invoiceNumber	No	VarChar		10		No			/invoice/invoiceNumber/text()
	orderNumber	No	Integer		10		No			/invoice/order/orderNumber/text()
	partNumber	No	VarChar		20		No			/invoice/order/lines/partNumber/text()
	quantity	No	Integer		10		No			/invoice/order/lines/quantity/text()
	unitPrice	No	Decimal		20	2	No			/invoice/order/lines/unitPrice/text()
	lineNumber	Yes	Integer		10		No			/invoice/order/lines/lineNumber/text()
*										

Figure 11-24 XML Output stage used a key column to create a nested relationship

The new XML Transformation stage continues to support this design pattern. As described in Chapter 10, "DataStage performance and XML" on page 291, high performance can be obtained with relational joins taking place in the parallel job and sending sorted, normalized data into the XML Stage. In the assembly, the normalized input data is converted to a hierarchical relationship, and it is mapped to the target document. Figure 11-25 demonstrates how to achieve the same invoice example with the XML StageXML Stage.



XML_Composer Step

Configuration | Information | Test Data

XML Target → Document Root → Validation → Mappings → Header → Format

Auto Map | Clear Mappings | ?

Source	Result	Target
top/Regroup_1:Invoice	✓	document_collection
		ns0:invoice
...group_1:Invoice/Input:invoiceNumber	✓	invoiceNumber
top/Regroup_1:Invoice/Orders	✓	order
...:Invoice/Orders/Input:orderNumber	✓	orderNumber
...up_1:Invoice/Orders/Regroup:Lines	✓	lines
...rs/Regroup:Lines/Input:partNumber	✓	partNumber
...rders/Regroup:Lines/Input:quantity	✓	quantity
...Orders/Regroup:Lines/Input:unitPrice	✓	unitPrice
...ders/Regroup:Lines/Input:lineNumber	✓	lineNumber

Input | Output

```
top
  InputLinks
  Regroup_1:Invoice
    Input:invoiceNumber
    Orders
      Input:orderNumber
      Regroup:Lines
        Input:partNumber
        Input:quantity
        Input:unitPrice
        Input:lineNumber
```

Figure 11-25 Multiple Regroup steps used to create a nested relationship

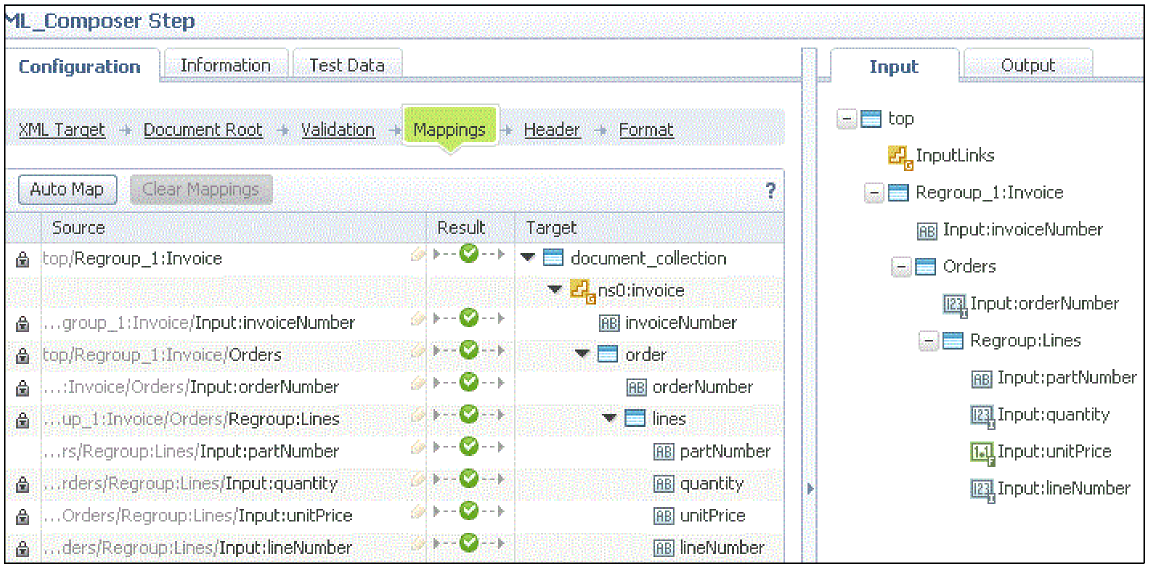
The target list `document_collection` identifies the context for creating an XML document. In this example, the top-level `Invoice` list is mapped to the document_collection, which results in a single document that is produced for each set of invoice data. Also, the target schema contains the lists `orders` and `lines`, which contain the repeating (one or more elements) structure.

In the assembly before the Composer step, multiple Regroup steps are used to create the hierarchical relationship from the flat, normalized input link. You can see the results of these Regroup steps in the input tree view of Figure 11-25 on page 357, labeled Regroup:Invoice and Regroup:Lines. The Regroup steps take multiple input rows and group them into multiple element lists. The Composer step can then easily map these lists into the output document.

Important: The configuration of the input link remains the same for both stages. Therefore, existing jobs can be modified easily without needing to change the job design. Merely replace the XML Output stage with the XML Stage on the job canvas. The rest is performed in the XML Stage assembly.

XML Output: Single row output mode

In the XML Output stage, this mode results in a new XML document that is created for each relational input row. The XPATH controls the resulting document structure. The following example is based on a schema for an address book. The address book stores data hierarchically by state, city, and zip code. The address book can have multiple states; each state can have multiple cities; each city can have multiple zip codes; and each zip code can have multiple addresses. XPATH for the nesting relationship in the XML Output stage looks similar to the "Aggregate all rows" mode example, as you can see in Figure 11-26 on page 359.

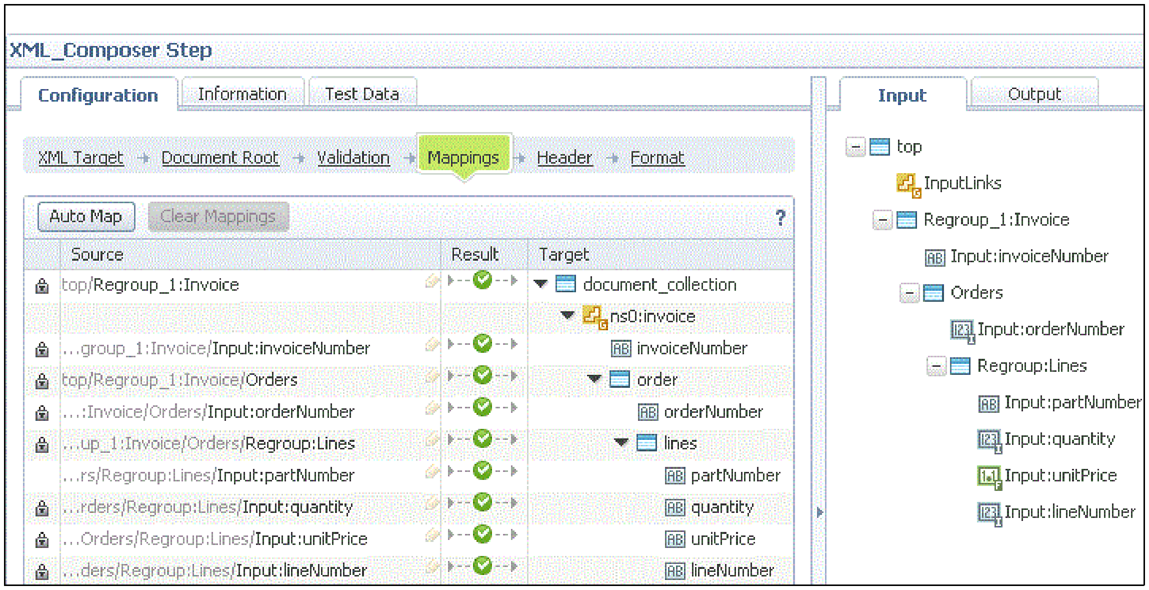


Source	Result	Target
top/Regroup_1:Invoice	✓	document_collection
		ns0:invoice
...group_1:Invoice/Input:invoiceNumber	✓	invoiceNumber
top/Regroup_1:Invoice/Orders	✓	order
...:Invoice/Orders/Input:orderNumber	✓	orderNumber
...up_1:Invoice/Orders/Regroup:Lines	✓	lines
...rs/Regroup:Lines/Input:partNumber	✓	partNumber
...rders/Regroup:Lines/Input:quantity	✓	quantity
...Orders/Regroup:Lines/Input:unitPrice	✓	unitPrice
...ders/Regroup:Lines/Input:lineNumber	✓	lineNumber

```
top
    InputLinks
    Regroup_1:Invoice
        Input:invoiceNumber
        Orders
            Input:orderNumber
            Regroup:Lines
                Input:partNumber
                Input:quantity
                Input:unitPrice
                Input:lineNumber
```

Figure 11-26 Configuring XML Output nesting relationship for single row mode

The XPATH looks similar, because it represents only the nesting structure of the target XML document. It does not represent any of the context, such as minimum/maximum occurrence. The XML Output stage used the single row output mode to enforce a single occurrence of the XML structure in each document and the creation of multiple output documents. The key to implementing the equivalent functionality in the XML Transformation stage is to use the proper XSD schema. The target schema must define its hierarchy by using groups rather than lists. In the XSD schema, set the maxOccurs attribute to `1` for the element to define the hierarchy. Figure 11-27 on page 360 illustrates the configuration of the Composer step in the XML assembly, for the equivalent of single row mode.



Source	Result	Target
top/Regroup_1:Invoice	✓	document_collection
		ns0:invoice
...group_1:Invoice/Input:invoiceNumber	✓	invoiceNumber
top/Regroup_1:Invoice/Orders	✓	order
...:Invoice/Orders/Input:orderNumber	✓	orderNumber
...up_1:Invoice/Orders/Regroup:Lines	✓	lines
...rs/Regroup:Lines/Input:partNumber	✓	partNumber
...rders/Regroup:Lines/Input:quantity	✓	quantity
...Orders/Regroup:Lines/Input:unitPrice	✓	unitPrice
...ders/Regroup:Lines/Input:lineNumber	✓	lineNumber

```
top
    InputLinks
    Regroup_1:Invoice
        Input:invoiceNumber
        Orders
            Input:orderNumber
            Regroup:Lines
                Input:partNumber
                Input:quantity
                Input:unitPrice
                Input:lineNumber
```

Figure 11-27 Mapping to a non-repeating structure

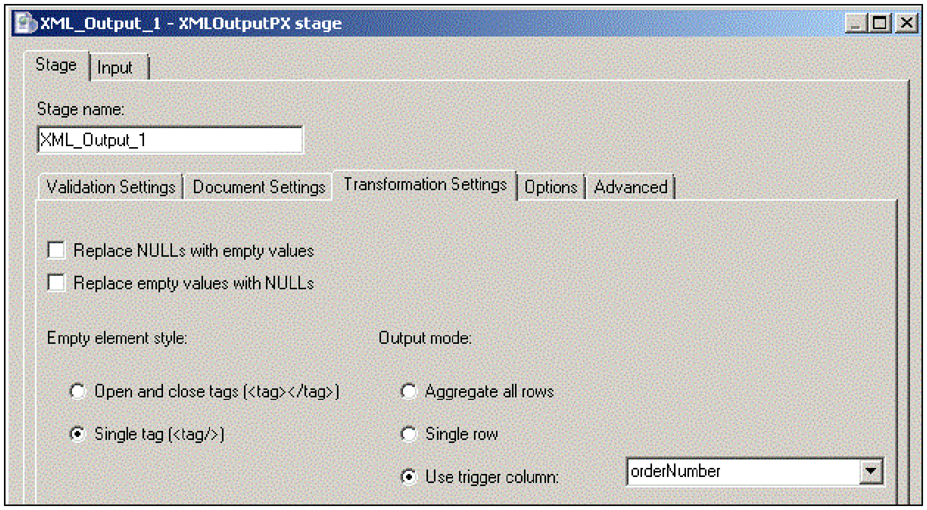
The only list in the target structure is the document_collection list, which identifies when a new document is created. The DSLink4 input link list is mapped to the document_collection; thus, one document is created for each input row. The rest of the elements are then mapped. There is no Regroup step. Even though the output target defines a hierarchical relationship, it can be managed through mapping because of the one-to-one relationships of all the fields.

The configuration of the input link remains the same. Jobs that use the XML Output stage with single row mode can be modified easily to use the XML Transformation stage without having to change the rest of the job design.

XML Output: Trigger column

In the XML Output stage, this mode results in the creation of a new document when the value of a specific column changed. The job developer can use a column as a flag, computing the value of the column, such as in an upstream Transformer stage, so that it changes precisely when a new document must be created. To demonstrate this functionality, we use the previous example from “XML Output: Aggregate all rows output mode” on page 355. In the original example, each invoice contains multiple orders, and each order contains multiple lines. The job creates a single invoice document that contains multiple orders. For this scenario, we modify the example so that it creates a separate document for each order.

The configuration of the XML Output stage, as shown in Figure 11-28, results in the creation of a new output XML document whenever the value of the orderNumber column changes.



Stage name: XML_Output_1

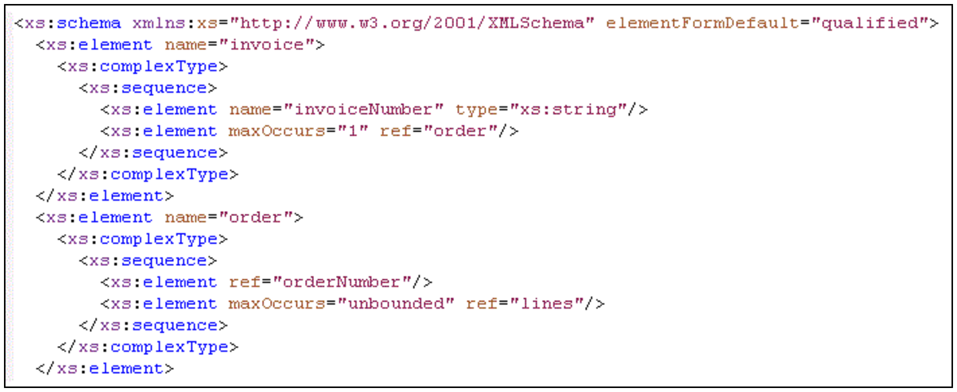
[ ] Replace NULLs with empty values
[ ] Replace empty values with NULLs

Empty element style:
- ( ) Open and close tags (<tag></tag>)
- (●) Single tag (<tag/>)

Output mode:
- ( ) Aggregate all rows
- ( ) Single row
- (●) Use trigger column: orderNumber

Figure 11-28 XML Output stage configured to create a document for each order

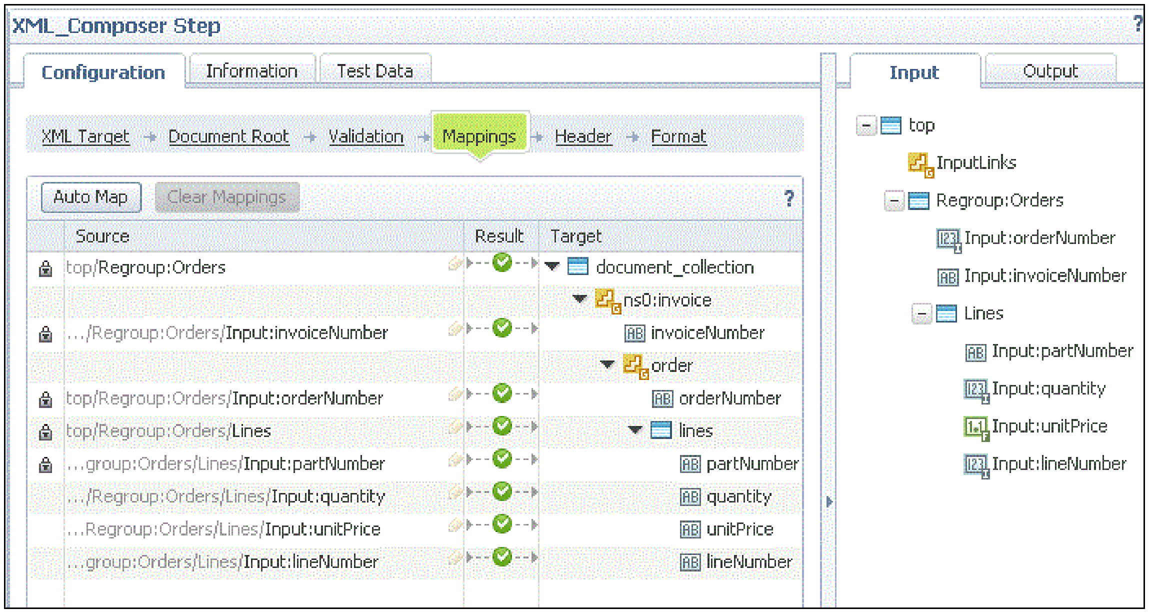
To replicate this functionality in the XML Transformation stage, we need to change the mapping to the document_collection target list, so that it receives a list of orders. We also need to modify the schema so that the definition for the invoice element contains only a single order element. To modify the schema, set `maxOccurs=”1”` as shown in Figure 11-29. The order element still contains an unlimited number of lines (`maxOccurs=”unbounded”`), which is the required behavior.



```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="invoice">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="invoiceNumber" type="xs:string"/>
        <xs:element maxOccurs="1" ref="order"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="order">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="orderNumber"/>
        <xs:element maxOccurs="unbounded" ref="lines"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

Figure 11-29 Schema fragment that shows one order in each invoice document

With this modified schema, the Order element appears as a group instead of a list in the assembly, as shown in Figure 11-30. Now, the Composer step creates only a single order in each XML document.



Source	Result	Target
top/Regroup:Orders	✓	document_collection
		ns0:invoice
.../Regroup:Orders/Input:invoiceNumber	✓	invoiceNumber
		order
top/Regroup:Orders/Input:orderNumber	✓	orderNumber
top/Regroup:Orders/Lines	✓	lines
...group:Orders/Lines/Input:partNumber	✓	partNumber
.../Regroup:Orders/Lines/Input:quantity	✓	quantity
...Regroup:Orders/Lines/Input:unitPrice	✓	unitPrice
...group:Orders/Lines/Input:lineNumber	✓	lineNumber

Figure 11-30 Mapping to create one document per order

A Regroup step is used to group the lines into a list in each order. The regrouped Orders are then mapped to the document_collection, which results in one output document that is created for each order.

A

Supported type mappings

In this appendix, we provide the following tables with the supported type mappings:

- DataStage data type to XML data type mapping
- XML data type to DataStage data type mapping
- Connector stage large object (LOB) reference support matrix

A.1 DataStage data type to XML data type mapping

Table A-1 provides a listing of the DataStage data type to XML data type mapping.

Table A-1 DataStage data type to XML data type mapping

DataStage data type	XML data type
BigInt	long
Unsigned BigInt	unsignedLong
Binary	binary
Bit	boolean
Char	string
Date	date
Decimal	decimal
Double	double
Float	double
Integer	int
Unsigned Integer	unsignedInt
LongNVarChar	string
LongVarBinary	binary
LongVarChar	string
Nchar	string
Numeric	decimal
NvarChar	string
Real	float
SmallInt	short
Unsigned SmallInt	unsignedShort
Time	time
Timestamp	dateTime
TinyInt	byte

DataStage data type	XML data type
Unsigned TinyInt	unsignedByte
VarBinary	binary
VarChar	string

A.2 XML data type to DataStage data type mapping

Table A-2 provides a listing of the XML data type to DataStage data type mapping.

Table A-2 XML data type to DataStage data type mapping

XML data type	DataStage data type
duration	VarChar
dateTime	TimeStamp
time	Time
date	Date
gYear	Date
gYearMonth	Date
gMonth	Date
gMonthDay	Date
gDay	Date
anyURI	VarChar
ENTITY	VarChar
ENTITIES	VarChar
ID	VarChar
IDREF	VarChar
IDREFS	VarChar
QName	VarChar
token	VarChar

XML data type	DataStage data type
language	VarChar
Name	VarChar
NCName	VarChar
NMTOKEN	VarChar
NMTOKENS	VarChar
NOTATION	VarChar
normalizedString	VarChar
string	VarChar
float	Real
double	Double
decimal	Decimal
integer	Decimal
long	BigInt
int	Integer
short	SmallInt
byte	TinyInt
positiveInteger	Decimal
nonPositiveInteger	Decimal
negativeInteger	Decimal
nonNegativeInteger	Decimal
unsignedLong	Unsigned BigInt
unsignedInt	Unsigned Integer
unsignedShort	Unsigned SmallInt
UnsignedByte	Unsigned TinyInt
hexBinary	VarChar
base64Binary	VarChar
boolean	Bit

A.3 Connector stage LOB reference support matrix

Table A-3 provides a listing of the Connector stage large object (LOB) references.

Table A-3 Connector stage LOB reference support matrix

Connector stage	Accepts LOB reference	Creates LOB reference
CDC Connector	Yes	No
DB2 Connector	Yes	Yes
DRS Connector	Yes	Yes
MQ Connector	Yes	Yes
Netezza Connector	No	No
ODBC Connector	Yes	Yes
Oracle Connector	Yes	Yes
Teradata Connector	Yes	Yes
XML Connector	Yes	Yes

B

Transformer functions

In this appendix, we present lists of the Transformer functions, which include the Aggregate functions and Switch functions. All functions are not available for all data types. The functions that are presented in Table B-1 are data type specific.

Table B-1 Aggregate functions

Function	Description
Average	Calculates the average value in the list.
Count	Counts the elements in the list. Ignores null values.
Concatenate	Concatenates all strings in the list, starting with the first element.
First	Selects first value in the list.
Last	Selects last value in the list.
Maximum	Selects the maximum value in the list. If the list contains Boolean strings, True is greater than False.
Minimum	Selects the minimum value in the list. If the list contains Boolean strings, False is less than True.
Sum	Calculates the sum of all of the values in the list.
Variance	Calculates the variance value of all of the values in the list.

All functions are not available for all data types. The functions that are presented in Table B-2 are data type specific.

Table B-2 Switch functions

Function	Description
isNull	Returns True if the value is null.
Greater than	Returns true if the value is greater than the value of the parameter.
Greater than or equal	Returns true if the value is greater than or equal to the value of the parameter.
Less than	Returns true if the value is less than the value of the parameter.
Less than or equal	Returns true if the value is less than or equal to the value of the parameter.
Equals	Returns true if the two values are the same.
Between	Returns true if the value is within the specified range.
IsTrue	Returns true if the value is true.
IsFalse	Returns true if the value is false.
Compare	Returns true if the string value is the same as the string value of the parameter.
CompareNoCase	Returns true if the string value is the same as the string value of the parameter. Ignores the case.
Contains	Returns true if the string value contains the string value of the parameter.
ContainsCaseInsensitive	Returns true if the string value contains the string value of the parameter. Ignores the case.
IsBlank	Returns true if the string is empty or null.
IsNotBlank	Returns true if the string is not empty and not null.
Like	Returns true if the string value matches the pattern defined by the parameter value. Use a percent sign (%) to define missing letters before and after the pattern.

Glossary

Access mode. The access mode of an IBM solidDB parameter defines whether the parameter can be changed dynamically via an ADMIN COMMAND, and when the change takes effect. The possible access modes are RO, RW, RW/Startup, and RW/Create.

Application Programming Interface. An interface provided by a software product that enables programs to request services.

BLOB. Binary large object. A block of bytes of data (for example, the body of a message) that has no discernible meaning, but is treated as one entity that cannot be interpreted.

DDL. (Data Definition Language). An SQL statement that creates or modifies the structure of a table or database, for example, CREATE TABLE, DROP TABLE, ALTER TABLE, and CREATE DATABASE.

Disk-based table (D-table). A table whose contents are stored primarily on disk so that the server copies only small amounts of data at a time into memory.

DML. (Data Manipulation Language). An INSERT, UPDATE, DELETE, or SELECT SQL statement.

Distributed Application. A set of application programs that collectively constitute a single application.

Dynamic SQL. SQL that is interpreted during execution of the statement.

Metadata. Typically called data (or information) about data. It describes or defines data elements.

Multi-Threading. Capability that enables multiple concurrent operations to use the same process.

Open Database Connectivity (ODBC). A standard application programming interface for accessing data in both relational and non-relational database management systems. Using this API, database applications can access data stored in database management systems on a variety of computers even if each database management system uses a different data storage format and programming interface. ODBC is based on the call level interface (CLI) specification of the X/Open SQL Access Group.

Optimization. The capability to enable a process to execute and perform in such a way as to maximize performance, minimize resource utilization, and minimize the process execution response time delivered to the user.

Partition. Part of a database that consists of its own data, indexes, configuration files, and transaction logs.

Primary Key. Field in a table that is uniquely different for each record in the table.

Process. An instance of a program running in a computer.

Server. A computer program that provides services to other computer programs (and their users) in the same or other computers. However, the computer in which a server program runs is also frequently referred to as a server.

Shared nothing. A data management architecture where nothing is shared between processes. Each process has its own processor, memory, and disk space.

SQL passthrough. The act of passing SQL statements to the back end, instead of executing statements in the front end.

Static SQL. SQL that has been compiled prior to execution. Typically, it provides the best performance.

Abbreviations and acronyms

IBM	International Business Machines Corporation
ITSO	International Technical Support Organization
ACID	Atomicity, Consistency, Isolation, Durability
ACS	access control system
AIX®	Advanced Interactive eXecutive from IBM
API	Application programming interface
ASCII	American Standard Code for Information Interchange
ASP	Active Server Pages
B2B	Business-to-Business
B2C	Business-to-Consumer
BE	Back-end data server
BLOB	Binary large object
CDC	Change Data Capture
CLI	Call-Level Interface
CLOB	Character large object
CPU	Central processing unit
CRM	Customer Relationship Management
DBA	Database Administrator
DBMS	DataBase Management System
DDL	Data Definition Language. An SQL statement that creates or modifies the structure of a table or database, for example, CREATE TABLE or DROP TABLE.
DES	Data Encryption Standard
DLL	Dynamic Link Library
DML	Data Manipulation Language. An INSERT, UPDATE, DELETE, or SELECT SQL statement.
DS	DataStage
DSN	Data Source Name
D-table	Disk-based Table
DTD	Document Type Definition
DTS	Distributed Transaction Stage
EAI	Enterprise Architecture Integration
EAR	Enterprise Archive
EJB	Enterprise JavaBeans
EOW	End-of-wave
ERE	External Reference Entity (solidDB HSB link failure detection method)
ETL	Extract-Transform-Load
FE	Front-end data server
FP	Fix Pack
FTP	File Transfer Protocol
Gb	Gigabits
GB	Gigabytes
GIS	Geographical Information Systems
GUI	Graphical user interface
HAC	solidDB High Availability Controller
HADR	High Availability Disaster Recovery - DB2
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
I/O	Input/Output

IBM	International Business Machines Corporation
ID	Identifier
IMS	Information Management System
IS	Information Server
ISV	Independent Software Vendor
ISD	Information Services Director
IT	Information Technology
ITSO	International Technical Support Organization
JMS	Java Message Services
JSP	Java Server Pages
KB	Kilobyte (1024 bytes)
LDAP	Lightweight Directory Access Protocol
LLA	Linked Library Access
LOB	Large objects
Mb	Megabits
MB	Megabytes
MBCS	Multibyte Character Set
MDM	Master Data Management
MPP	Massively Parallel Processing
MQ	Message Queue
MVC	Model-View-Controller
MWB	Metadata WorkBench
ODBC	Open DataBase Connectivity
OS	Operating System
O/S	Operating System
OSH	Orchestrate SHell
PDA	Personal Device Assistant
RDBMS	Relational DataBase Management System
REST	Representational State Transfer
RSS	Really Simple Syndication
RTF	Rich Text Format
SBCS	Single Byte Character Set
SDK	Software Developers Kit
SGML	Standard Generalized Markup Language
SL	Section Leader
SMP	Symmetric MultiProcessing
SOA	Service-oriented architecture
SOAP	Simple Object Access Protocol
SQL	Structured Query
SVG	Scalable Vector Graphics
UDDI	Universal Description, Discovery and Integration
UOW	Unit-of-Work
URL	Uniform Resource Locator
VRML	Virtual Reality Modeling Language
VSAM	Virtual Sequential Access Method
W3C	Worldwide Web Consortium
WISD	WebSphere Information Services Director
WSDL	Web Services Description Language
XHTML	eXtensible HyperText Markup Language
XML	eXtensible Markup Language
XPATH	XML Path Language
XSL	Extensible Stylesheet Language
XSLT	XSL Transformations

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

The following IBM Redbooks publications provide additional information about the topic in this document. Several publications referenced in this list might be available in softcopy only.

- *InfoSphere DataStage Parallel Framework Standard Practices,* SG24-7830
- *IBM InfoSphere DataStage Data Flow and Job Design*, SG24-7576
- *The XML Files: Development of XML/XSL Applications Using WebSphere Studio Version 5*, SG24-6586
- *Implementing IBM InfoSphere Change Data Capture for DB2 z/OS V6.5*, REDP-4726
- *DB2 9 pureXML Guide*, SG24-7315
- *Cloud Computing and the Value of zEnterprise*, REDP-4763
- *Master Data Management IBM InfoSphere Rapid Deployment Package*, SG24-7704

You can search for, view, download or order these documents and other Redbooks, Redpapers, Web Docs, draft and additional materials, at the following website:

ibm.com/redbooks

Online resources

These websites are also relevant as further information sources:

- Information Server, Version 8.7 Information Center: Oracle connector and LOB /XML types:

  http://publib.boulder.ibm.com/infocenter/iisinfsv/v8r7/topic/com.ibm.swg.im.iis.conn.oracon.usage.doc/topics/suppt_lob_oracc.html

- Oracle Database documentation: Overview of XML DB:

  http://download.oracle.com/docs/cd/E11882_01/appdev.112/e23094/xdb01int.htm#ADXDB0100
- Articles from IBM developerWorks

  http://www.ibm.com/developerworks/views/data/libraryview.jsp?search_by=infosphere+datastage+8.5
- IBM Information Center

  http://publib.boulder.ibm.com/infocenter/iisinfsv/v8r5/topic/com.ibm.swg.im.iis.ds.stages.xml.usage.doc/topics/introduction_to_xml_stage.html
- Blog by Ernie Ostic

  http://dsrealtime.wordpress.com/table-of-contents-for-reasonably-useful-postings/

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

InfoSphere DataStage for Enterprise XML Data Integration

(0.5” spine)
0.475”<->0.875”
250 <-> 459 pages

InfoSphere DataStage for Enterprise XML Data Integration

Addresses the complexities of hierarchical data types

Reads huge documents using streaming technology

Spans both batch and real-time run times

XML is one of the most common standards for the exchange of information. However, organizations find challenges in how to address the complexities of dealing with hierarchical data types, particularly as they scale to gigabytes and beyond. In this IBM Redbooks publication, we discuss and describe the new capabilities in IBM InfoSphere DataStage 8.5. These capabilities enable developers to more easily manage the design and processing requirements presented by the most challenging XML sources. Developers can use these capabilities to create powerful hierarchical transformations and to parse and compose XML data with high performance and scalability. Spanning both batch and real-time run times, these capabilities can be used to solve a broad range of business requirements.

As part of the IBM InfoSphere Information Server 8.5 release, InfoSphere DataStage was enhanced with new hierarchical transformation capabilities called XML Stage. XML Stage provides native XML schema support and powerful XML transformation functionality. These capabilities are based on a unique state-of-the-art technology that allows you to parse and compose any complex XML structure from and to a relational form, as well as to a separate hierarchical form.

SG24-7987-00 ISBN 0738436720