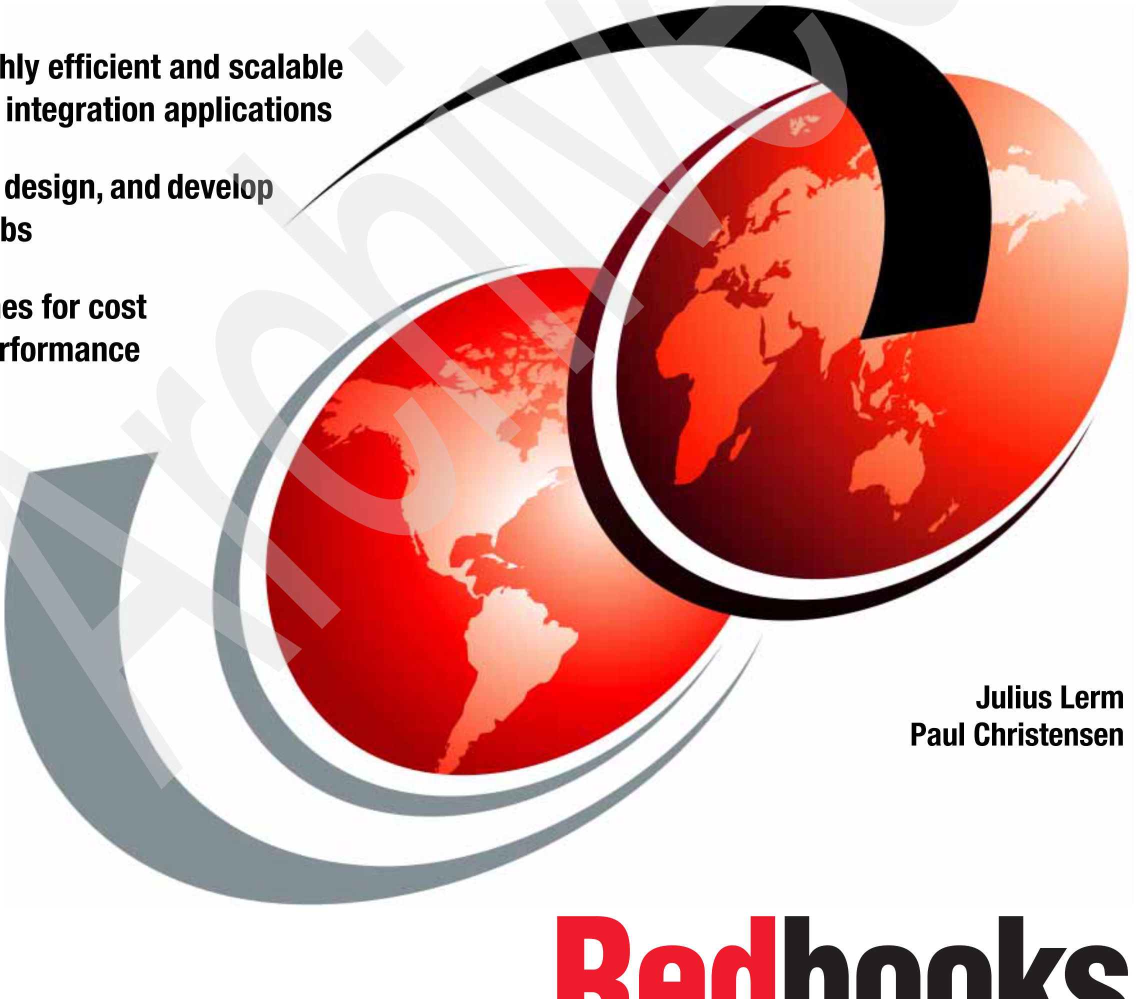
IBM

InfoSphere DataStage Parallel Framework Standard Practices



- Develop highly efficient and scalable information integration applications
- Investigate, design, and develop data flow jobs
- Get guidelines for cost effective performance

Julius Lerm
Paul Christensen

Redbooks

ibm.com/redbooks

International Technical Support Organization

InfoSphere DataStage: Parallel Framework Standard Practices

September 2010

SG24-7830-00

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

First Edition (September 2010)

This edition applies to Version 8, Release 1 of IBM InfoSphere Information Server (5724-Q36) and Version 9, Release 0, Modification 1 of IBM InfoSphere Master Data Management Server (5724-V51), and Version 5.3.2 of RDP.

Contents

Notices

Trademarks

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

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

AIX®	Information Agenda™	MVS™
ClearCase®	Informix®	Orchestrate®
DataStage®	InfoSphere™	Redbooks®
DB2®	Iterations®	Redbooks (logo) ®
IBM®	MQSeries®	WebSphere®

The following terms are trademarks of other companies:

Java, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

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

UNIX is a registered trademark of The Open Group in the United States and other countries.

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

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

Preface

In this IBM® Redbooks® publication, we present guidelines for the development of highly efficient and scalable information integration applications with InfoSphere™ DataStage® (DS) parallel jobs.

InfoSphere DataStage is at the core of IBM Information Server, providing components that yield a high degree of freedom. For any particular problem there might be multiple solutions, which tend to be influenced by personal preferences, background, and previous experience. All too often, those solutions yield less than optimal, and non-scalable, implementations.

This book includes a comprehensive detailed description of the components available, and descriptions on how to use them to obtain scalable and efficient solutions, for both batch and real-time scenarios.

The advice provided in this document is the result of the combined proven experience from a number of expert practitioners in the field of high performance information integration, evolved over several years.

This book is intended for IT architects, Information Management specialists, and Information Integration specialists responsible for delivering cost-effective IBM InfoSphere DataStage performance on all platforms.

This book is organized as follows:

- Chapter 1, “Data integration with Information Server and DataStage” on page 1

  This chapter presents an overview of the Information Integration components and services offered by IBM Information Management. This includes a discussion of Information Server 8 and its layers (client, services, metadata, and engine).

- Chapter 2, “Data integration overview” on page 11

  In this chapter we describe the use of job sequences and other types of jobs commonly found in data integration applications.

- Chapter 3, “Standards” on page 21

  In chapter 3, we describe and discuss recommendations for the adoption of the following development and deployment standards:

  – Directory structures
  – Naming conventions

  - Graphical job layouts
  - source control systems
  - categorization standards

- Chapter 4, “Job parameter and environment variable management” on page 55

  In this chapter we describe how to use job parameters, parameter sets, and environment variables.

- Chapter 5, “Development guidelines” on page 69

  In this chapter, we discuss development guidelines for modular designs, adequate error handling, and how to facilitate metadata management.

- Chapters 6–14, beginning on page 91

  In this sequence of chapters of the book, we provide detailed description of a number of the features and related stage types of DataStage. The following list details the features covered:

  - Partitioning and collecting
  - Sorting
  - File stage Usage
  - Transformation Languages
  - Combining Data
  - Restructuring Data
  - Performance Tuning Job Designs
  - Existing Database stage Guidelines
  - Connector stage Guidelines

- Chapter 15, “Batch data flow design” on page 259

  In this chapter we present recommendations on how to obtain efficient and scalable job designs to process large amounts of data in batch mode. We also list the primary design goals for batch applications and discuss a number of common bad design patterns.

  The emphasis is on proven job design patterns that apply to most, if not all, large volume batch processing applications.

- Chapter 16, “Real-time data flow design” on page 293

  In this chapter we present recommendations on how to obtain efficient and scalable job designs to process large amounts of data in real-time mode.

  Though partitioning and pipeline parallelism enables scalability and efficiency, it introduces challenges in real-time scenarios. We present a comprehensive discussion on what real-time means in the context of InfoSphere DataStage:

  - Job topologies
  - Message-oriented processing with MQSeries®
  - SOA applications with Information Services Director (ISD)

   - Real-time scalability

   In addition, we present techniques to overcome those challenges introduced by parallelism.

- Appendices, beginning on page 375

  We provide a number of supporting topics in the appendix:

  - Runtime topologies for distributed transaction jobs
  - Standard practices summary
  - DataStage naming reference
  - Example job template
  - Understanding the parallel job score
  - Estimating the size of a parallel dataset
  - Environmental variables reference
  - DataStage data types

Document history

Prior to this Redbooks publication, there was DataStage release documentation made available dated April 15, 2009, written by Julius Lerm. This book updates and extends the information in that initial release documentation with terminology, import/export mechanisms, and job parameter handling (including parameter sets).

The team who wrote this book

This book was produced by the following authors, along with contributions from a number of their colleagues. The authors are listed, along with a short biographical sketch of each.

Julius Lerm is a Technical Architect, and a member of the Center of Excellence of IBM Information Management, Analytics & Optimization Software Services. He has 16 years experience in solution architecting, and developing and performance tuning of large scale parallel database and information integration applications. His experience includes extensive development of custom components and extensions to the DataStage parallel framework, as well as tools that bridge product functionality gaps and address complex integration problems. He has provided guidance, mentoring, and training to customers worldwide in transactional, data warehousing, and service-oriented and message-based processing projects. He has also presented at several conferences. Julius holds Bachelor's and Master's degrees in Computer Science from Federal University of Rio Grande do Sul (UFRGS, Brazil).

Paul Christensen is a Technical Architect and member of the worldwide IBM Information Agenda™ Architecture team. With 19 years of experience in enterprise data management and parallel computing technologies, he has led the successful design, implementation, and management of large-scale Data Integration and Information Management solutions using the IBM Information Agenda and partner portfolios. Paul's experience includes early hardware-based parallel computing platforms, massively parallel databases including Informix® and DB2®, and the parallel framework of IBM Information Server and DataStage. To facilitate successful customer and partner deployments using IBM Information Server, he has helped to develop standard practices, course material, and technical certifications. Paul holds a Bachelor's degree in Electrical Engineering from Drexel University, and is an IBM Certified Solution Developer.

Other Contributors

We would like to give special thanks to the following contributing authors whose input added significant value to this publication.

Mike Carney - Technical Architect, IBM Software Group, Information Management, Westford, MA
Tony Curcio - DataStage Product Manager, IBM Software Group, Information Management, Charlotte, NC
Patrick Owen - Technical Architect, IBM Software Group, Information Management, Little Rock, AR
Steve Rigo - Technical Architect, IBM Software Group, Information Management, Atlanta, GA
Ernie Ostic - Technical Sales Specialist, IBM Software Group, Worldwide Sales, Newark, NJ
Paul Stanley - Product Development Engineer, IBM Software Group, Information Management, Boca Raton, FL

In the following sections we thank others who have contributed to the development and publication of this IBM Redbooks publication.

From IBM Locations Worldwide
Tim Davis - Executive Director, Information Agenda Architecture Group, IBM Software Group, Information Management, Littleton, MA
Susan Laime - IM Analytics and Optimization Software Services, IBM Software Group, Information Management, Littleton, MA
Margaret Noel - Integration Architect, IBM Software Group, Information Management, Atlantic Beach, FL

From the International Technical Support Organization
Chuck Ballard - Project Manager at the International Technical Support organization, in San Jose, California. Chuck managed the processes required to format and publish this IBM Redbooks Publication.
Mary Comianos - Publications Management
Emma Jacobs - Graphics
James Hoy - Editor

Now you can become a published author, too!

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

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

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

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

- Use the online **Contact us** review Redbooks form found at:

  ibm.com/redbooks

- Send your comments in an e-mail to:

  redbooks@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

Stay connected to IBM Redbooks

- Find us on Facebook:

  `http://www.facebook.com/pages/IBM-Redbooks/178023492563?ref=ts`
- Follow us on twitter:

  `http://twitter.com/ibmredbooks`
- Look for us on LinkedIn:

  `http://www.linkedin.com/groups?home=&gid=2130806`
- Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:

  `https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm`
- Stay current on recent Redbooks publications with RSS Feeds:

  `http://www.redbooks.ibm.com/rss.html`

1

Data integration with Information Server and DataStage

In this chapter we discuss and describe the concepts, education, and services that are available to help you get started with your data integration activities. 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 life cycle. Core to those activities is the InfoSphere Information Server platform, and InfoSphere DataStage (DS).

IBM InfoSphere Information Server is a software platform that helps organizations derive more value from the complex, heterogeneous information 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 volumes 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.

DataStage can manage data arriving in real-time as well as data received on a periodic or scheduled basis. This 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.

With these components and a great set of standard practices, you are on your way to a highly successful data integration effort. To help you further along the way, this book also provides a brief overview of a number of services and education offerings by IBM.

But first, to aid in understanding, we provide a brief overview of the InfoSphere Information Server 8.

1.1 Information Server 8

Information Server 8 implements a new architecture that differs from earlier versions of DataStage.

DataStage 7.X consisted of a two-tier infrastructure, with clients connected directly to the DSEngine. The DSEngine stored all of the metadata and runtime information, as well as controlled the execution of jobs.

IBM InfoSphere Information Server 8 is installed in layers that are mapped to the physical hardware. In addition to the main product modules, product components are installed in each tier as needed.

1.1.1 Architecture and information tiers

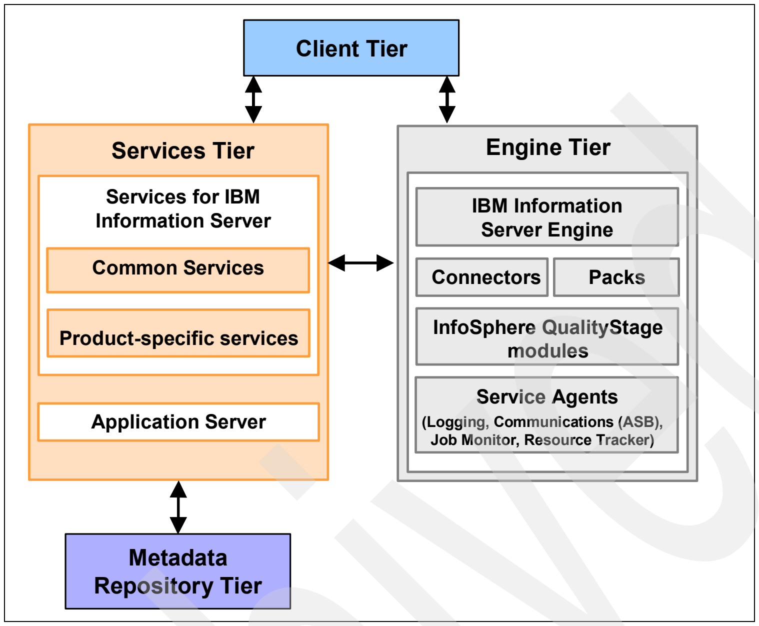
This new architecture does not affect the way DataStage jobs are developed. The DataStage parallel framework remains the same, with a few minimal changes to internal mechanisms that do not impact job designs in any way.

From a job design perspective, the product has interesting new features:

- New stages, such as Database Connectors, Slowly Changing Dimensions, and Distributed Transaction.
- Job Parameter Sets
- Balanced optimization, a capability that automatically or semi-automatically rewrites jobs to make use of RDBMS capabilities for transformations.

Information Server also provides new features for job developers and administrators, such as a more powerful import/export facility, a job comparison tool, and an impact analysis tool.

The information tiers work together to provide services, job execution, metadata, and other storage, as shown in Figure 1-1.



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 1-1 Information Server tiers

The following list describes the information tiers shown in Figure 1-1:

- Client

  Product module clients that are not Web-based and that are used for development and administration in IBM InfoSphere Information Server. The client tier includes the IBM InfoSphere Information Server console, IBM InfoSphere DataStage and QualityStage clients, and other clients.

- Engine

  Runtime engine that runs jobs and other tasks for product modules that require the engine.

- Metadata repository

  Database that stores the shared metadata, data, and configuration for IBM InfoSphere Information Server and the product modules.

- Services

  Common and product-specific services for IBM InfoSphere Information Server along with IBM WebSphere® Application Server (application server).

In this document we focus on aspects related to parallel job development, which is directly related to the Client and Engine layers.

1.2 IBM Information Management InfoSphere Services

IBM Information Management InfoSphere Professional Services offers a broad range of workshops and services designed to help you achieve success in the design, implementation, and rollout of critical information integration projects. An overview of this range of services is depicted in Figure 1-2.

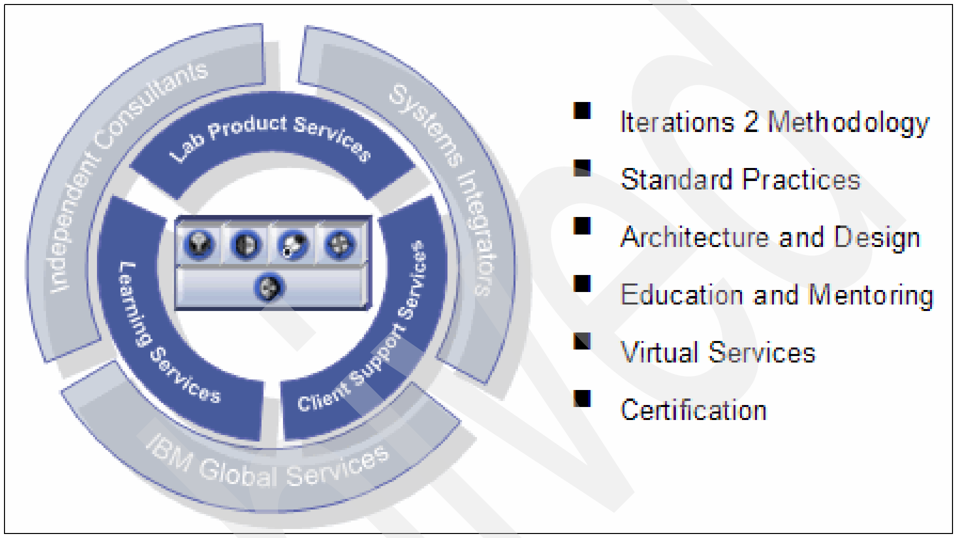


Figure 1-2 IBM InfoSphere Services overview

Based on that overview, we have listed a number of services offerings and their description in Table 1-1.

Table 1-1 Services description

Services offerings	**Description**
Staff Augmentation and Mentoring	Whether through workshop delivery, project leadership, or mentored augmentation, the Professional Services staff of IBM Information Platform and Solutions use IBM methodologies, standard practices, and experience developed through thousands of successful engagements in a wide range of industries and government entities.
Learning Services	IBM offers a variety of courses covering the IBM Information Management product portfolio. The IBM blended learning approach is based on the principle that people learn best when provided with a variety of learning methods that build upon and complement each other. With that in mind, courses are delivered through a variety of mechanisms: classroom, on-site, and Web-enabled FlexLearning.
Certification	IBM offers a number of professional certifications through independent testing centers worldwide. These certification exams provide a reliable, valid, and fair method of assessing product skills and knowledge gained through classroom and real-world experience.
Client Support Services	IBM is committed to providing our customers with reliable technical support worldwide. All client support services are available to customers who are covered under an active IBM InfoSphere maintenance agreement. Our worldwide support organization is dedicated to assuring your continued success with IBM InfoSphere products and solutions.
Virtual Services	The low cost Virtual Services offering is designed to supplement the global IBM InfoSphere delivery team, as needed, by providing real-time, remote consulting services. Virtual Services has a large pool of experienced resources that can provide IT consulting, development, migration, and training services to customers for IBM InfoSphere DataStage.

1.3 Center of Excellence for Data Integration (CEDI)

Establishing a CEDI in your enterprise can increase efficiency and drive down the cost of implementing data integration projects. A CEDI can be responsible for competency, readiness, accelerated mentored learning, common business rules, standard practices, repeatable processes, and the development of custom methods and components tailored to your business.

IBM InfoSphere Professional Services offerings can be delivered as part of a strategic CEDI initiative, or on an as-needed basis across a project life cycle. The IBM InfoSphere services offerings in an information integration project life cycle are illustrated in Figure 1-3.

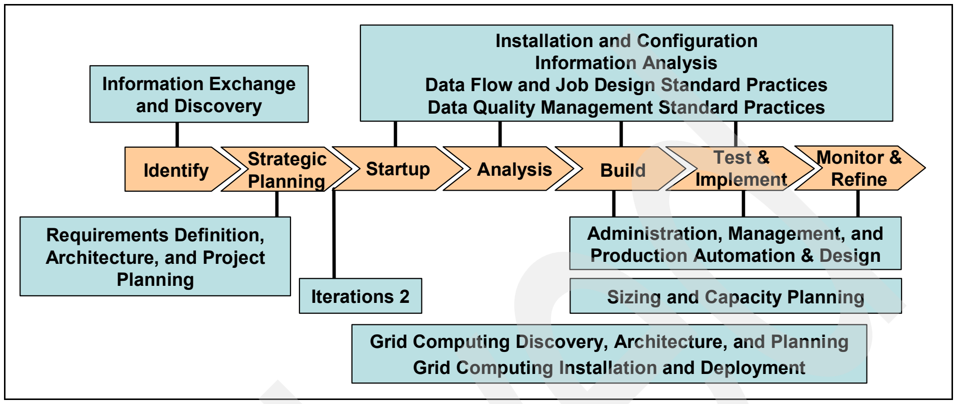

Installation and Configuration
Information Analysis
Data Flow and Job Design Standard Practices
Data Quality Management Standard Practices
Information Exchange and Discovery
Identify
Strategic Planning
Startup
Analysis
Build
Test & Implement
Monitor & Refine
Requirements Definition, Architecture, and Project Planning
Iterations 2
Administration, Management, and Production Automation & Design
Sizing and Capacity Planning
Grid Computing Discovery, Architecture, and Planning
Grid Computing Installation and Deployment

Figure 1-3 InfoSphere Services offerings and project life cycle

Table 1-2 lists the workshop offerings and descriptions for project startup.

Table 1-2 Project startup workshops

Project Startup Workshops	**Description**
Information Exchange and Discovery Workshop	Targeted for clients new to the IBM InfoSphere product portfolio, this workshop provides high-level recommendations on how to solve a customer's particular problem. IBM analyzes the data integration challenges outlined by the client and develops a strategic approach for addressing those challenges.
Requirements Definition, Architecture, and Project Planning Workshop	Guiding clients through the critical process of establishing a framework for a successful future project implementation, this workshop delivers a detailed project plan, as well as a project blueprint. These deliverables document project parameters, current and conceptual end states, network topology, and data architecture and hardware and software specifications. It also outlines a communication plan, defines scope, and captures identified project risk.
Iterations® 2	IBM Iterations 2 is a framework for managing enterprise data integration projects that integrate with existing customer methodologies. Iterations 2 is a comprehensive, iterative, step-by-step approach that leads project teams from initial planning and strategy through tactical implementation. This workshop includes the Iterations 2 software, along with customized mentoring.

The workshop offerings and descriptions for standard practices are shown in Table 1-3.

Table 1-3 Standard practices workshops

Standard Practices Workshops	**Description**
Installation and Configuration Workshop	This workshop establishes a documented, repeatable process for installation and configuration of IBM InfoSphere Information Server components. This might involve review and validation of one or more existing Information Server environments, or planning, performing, and documenting a new installation.
Information Analysis Workshop	This workshop provides clients with a set of standard practices and a repeatable methodology for analyzing the content, structure, and quality of data sources using a combination of IBM InfoSphere Information Analyzer, QualityStage, and Audit stage.
Data Flow and Job Design Standard Practices Workshop	This workshop helps clients establish standards and templates for the design and development of parallel jobs using IBM InfoSphere DataStage through practitioner-led application of IBM standard practices to a client's environment, business, and technical requirements. The delivery includes a customized standards document as well as custom job designs and templates for a focused subject area.
Data Quality Management Standard Practices Workshop	This workshop provides clients with a set of standard processes for the design and development of data standardization, matching, and survivorship processes using IBM InfoSphere QualityStage The data quality strategy formulates an auditing and monitoring program to ensure on-going confidence in data accuracy, consistency, and identification through client mentoring and sharing of IBM standard practices.
Administration, Management, and Production Automation Workshop	This workshop provides customers with a customized tool kit and a set of proven standard practices for integrating IBM InfoSphere DataStage into a client's existing production infrastructure (monitoring, scheduling, auditing/logging, change management) and for administering, managing and operating DataStage environments.

The workshop offerings and descriptions for advanced deployment are shown in Table 1-4.

Table 1-4 Advanced deployment workshops

Advanced Deployment Workshops	**Description**
Health Check Evaluation	This workshop is targeted for clients currently engaged in IBM InfoSphere development efforts that are not progressing according to plan, or for clients seeking validation of proposed plans prior to the commencement of new projects. It provides review of, and recommendations for, core Extract Transform and Load (ETL) development and operational environments by an IBM expert practitioner.
Sizing and Capacity Planning Workshop	The workshop provides clients with an action plan and set of recommendations for meeting current and future capacity requirements for data integration. This strategy is based on analysis of business and technical requirements, data volumes and growth projections, existing standards and technical architecture, and existing and future data integration projects.
Performance Tuning Workshop	This workshop guides a client's technical staff through IBM standard practices and methodologies for review, analysis, and performance optimization using a targeted sample of client jobs and environments. This workshop identifies potential areas of improvement, demonstrates IBM processes and techniques, and provides a final report with recommended performance modifications and IBM performance tuning guidelines.
High-Availability Architecture Workshop	Using IBM InfoSphere Standard Practices for high availability, this workshop presents a plan for meeting a customer's high availability requirements using the parallel framework of IBM InfoSphere DataStage. It implements the architectural modifications necessary for high availability computing.
Grid Computing Discovery, Architecture and Planning Workshop	This workshop teaches the planning and readiness efforts required to support a future deployment of the parallel framework of IBM InfoSphere DataStage on Grid computing platforms. This workshop prepares the foundation on which a follow-on grid installation and deployment is executed, and includes hardware and software recommendations and estimated scope.
Grid Computing Installation and Deployment Workshop	In this workshop, the attendee installs, configures, and deploys the IBM InfoSphere DataStage Grid Enabled Toolkit in a client's grid environments, and provides integration with Grid Resource Managers, configuration of DataStage, QualityStage, and Information Analyzer.

For more details on any of these IBM InfoSphere Professional Services offerings, and to find a local IBM Information Management contact, visit: the following web page:

http://www.ibm.com/software/data/services/ii.html

1.4 Workshops for IBM InfoSphere DataStage

Figure 1-4 illustrates the various IBM InfoSphere Services workshops around the parallel framework of DataStage. The DataFlow Design workshop is part of the Investigate, Design and Develop track of IBM InfoSphere DataStage workshop.

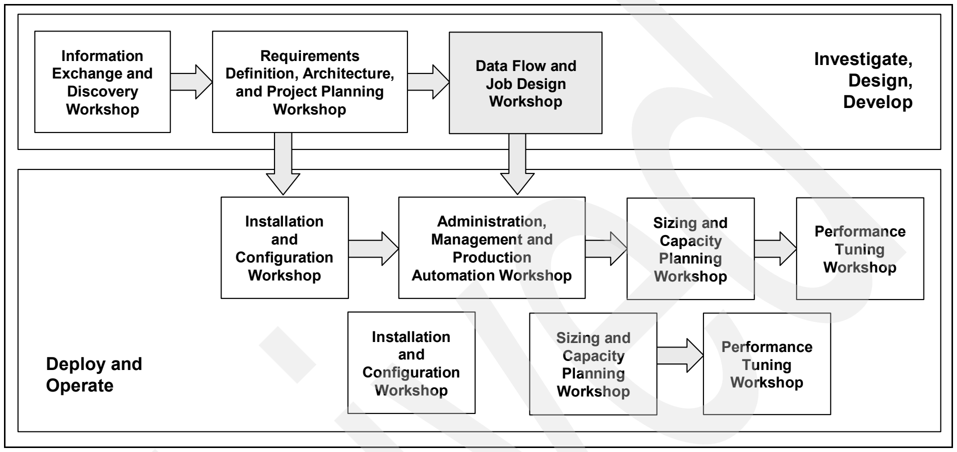

Information Exchange and Discovery Workshop
Requirements Definition, Architecture, and Project Planning Workshop
Data Flow and Job Design Workshop
Investigate, Design, Develop
Installation and Configuration Workshop
Administration, Management and Production Automation Workshop
Sizing and Capacity Planning Workshop
Performance Tuning Workshop
Deploy and Operate
Installation and Configuration Workshop
Sizing and Capacity Planning Workshop
Performance Tuning Workshop

Figure 1-4 Services workshops for the IBM InfoSphere DataStage Parallel Framework

2

Data integration overview

Work performed by data integration jobs fall into four general categories:

- Reading input data, including sequential files, databases and DataStage (DS) Parallel Datasets
- Performing row validation to support data quality
- Performing transformation from data sources to data targets
- Provisioning data targets

Figure 2-1 depicts the general flow diagram for DataStage parallel jobs.

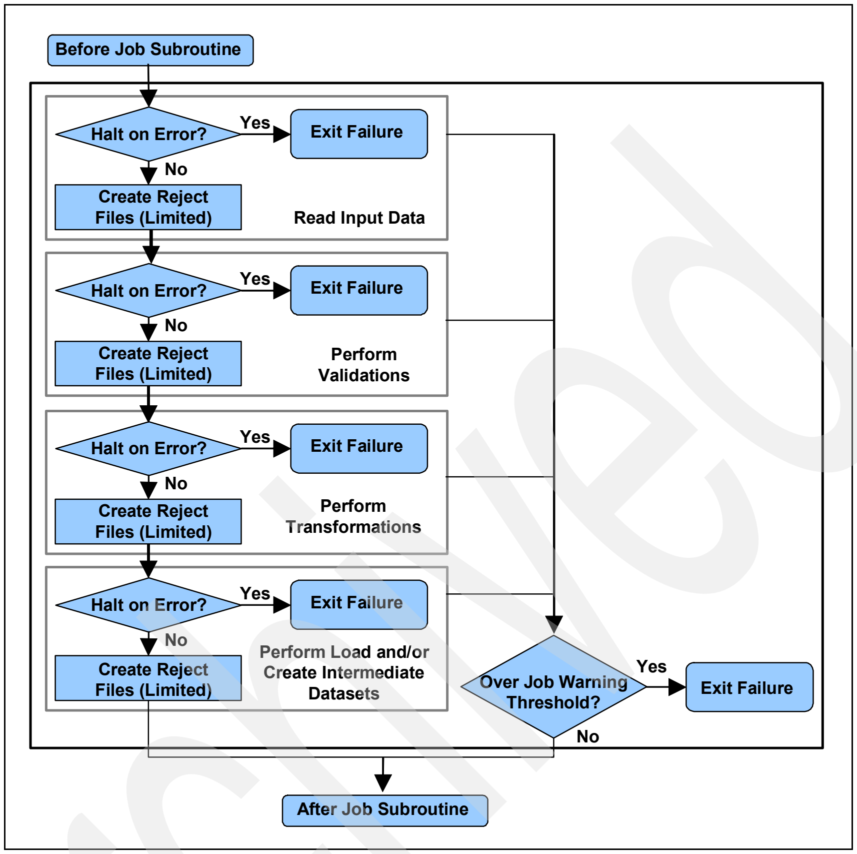


Figure 2-1 General flow for parallel job

2.1 Job sequences

As shown in Figure 2-1 on page 12, data integration development is intended to be modular. It is built from individual parallel jobs assembled in IBM InfoSphere DataStage, controlled as modules from master DataStage Sequence jobs. This is illustrated in Figure 2-2.

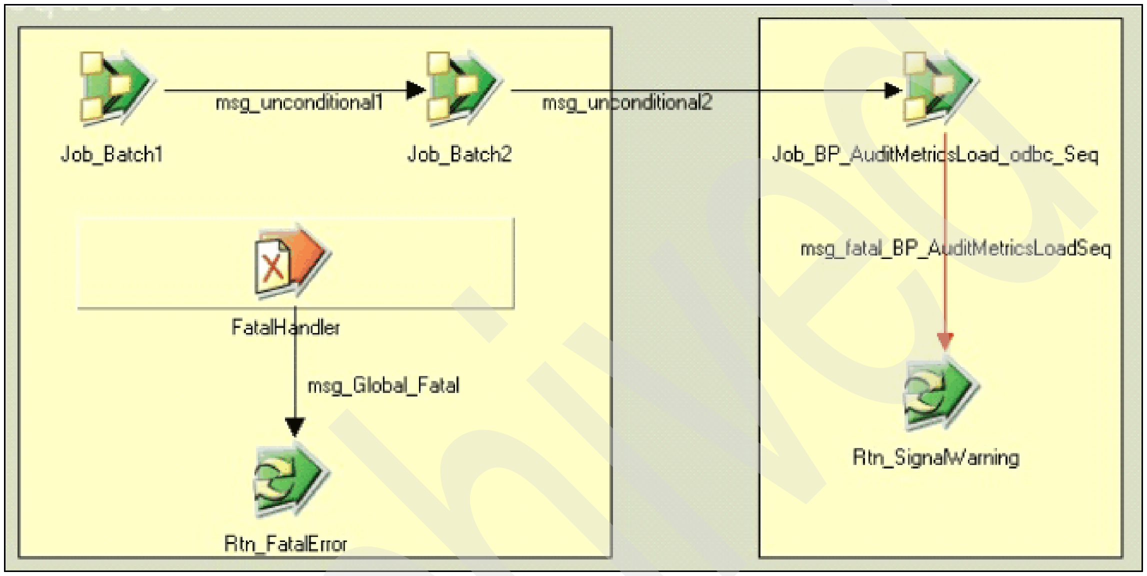


Figure 2-2 Sample job sequence

These job sequences control the interaction and error handling between individual DataStage jobs, and together form a single end-to-end module in a DataStage application.

Job sequences also provide the recommended level of integration with external schedulers (such as AutoSys, Cron, CA7, and so forth). This provides a level of granularity and control that is easy to manage and maintain, and provides an appropriate use of the respective technologies.

In most production deployments, job sequences require a level of integration with various production automation technologies (such as scheduling, auditing/capture, and error logging).

2.2 Job types

Nearly all data integration jobs fall into three major types:

- Transformation jobs

  Transformation jobs prepare data for provisioning jobs
- Provisioning jobs

  Provisioning jobs load transformed data, and
- Hybrid jobs

  Hybrid jobs do both.

Table 2-1 defines when each job type is used.

Table 2-1 Job type use

Type	**Data requirements**	**Example**
Transformation	Data must not be changed by any method unless jobs transforming an entire subject area have successfully completed, or where the resource requirements for data transformation are large.	Reference tables upon which all subsequent jobs and the current data target (usually a database) depend, or long running provisioning processes. This prevents partial replacement of reference data in the event of transformation failure, and preserves the compute effort of long running transformation jobs.
Hybrid	Data can be changed regardless of success or failure.	Non-reference data or independent data are candidates. The data target (usually a database) must allow subsequent processing of error or reject rows and tolerate partial or complete non-update of targets. Neither the transformation nor provisioning requirements are large.
Provisioning	Data must not be changed by any method unless jobs transforming an entire subject area have successfully completed, or where the resource requirements for data provisioning are large.	Any target where either all sources have been successfully transformed or where the resources required to transform the data must be preserved in the event of a load failure.

2.2.1 Transformation jobs

In transformation jobs, data sources, which might be write-through cache datasets, are processed to produce a load-ready dataset that represents either the entire target table or new records to be appended to the target table.

If the entire target table is regenerated with each run, and no other external or subsequent processing alters the contents of the target table, the output dataset qualifies as a write-through cache that can be used by subsequent DataStage jobs instead of reading the entire target table.

When we say "subsequent jobs" we mean any jobs executed as part of the same transformation cycle, until the target table is updated in the target database by a provisioning job. A transformation cycle might correspond, for instance, to daily, weekly, or monthly batch processes.

As a transformation cycle progresses, any real-time updates to the target tables in the target database (by online or real-time applications, for instance), must cease, so as not to yield invalid results at the end of the processing cycle. Otherwise, at the final steps of the processing window, provisioning jobs would invalidate or overwrite modifications performed by the online applications. The target table would then become inconsistent.

The following example transformation job demonstrates the use of write-through cache parallel datasets: The source Oracle stages read data from a couple of source tables, which are joined and updated by a Parallel Transformer. This transformed join result is funneled with the content of the target table (extracted by the "TargetTable_tbl" stage). The result of the Funnel is saved to the Seg1_DS parallel dataset.

The content of the Seg1_DS parallel dataset can be used by subsequent jobs, in the same transformation cycle, as input to other joins, lookup, or transform operations, for instance. Those jobs can thus avoid re-extracting data from that table unnecessarily. These operations might result in new versions of this dataset. Those new versions replace Seg1_DS (with different names) and are used by other subsequent downstream jobs.

An example job is illustrated in Figure 2-3.

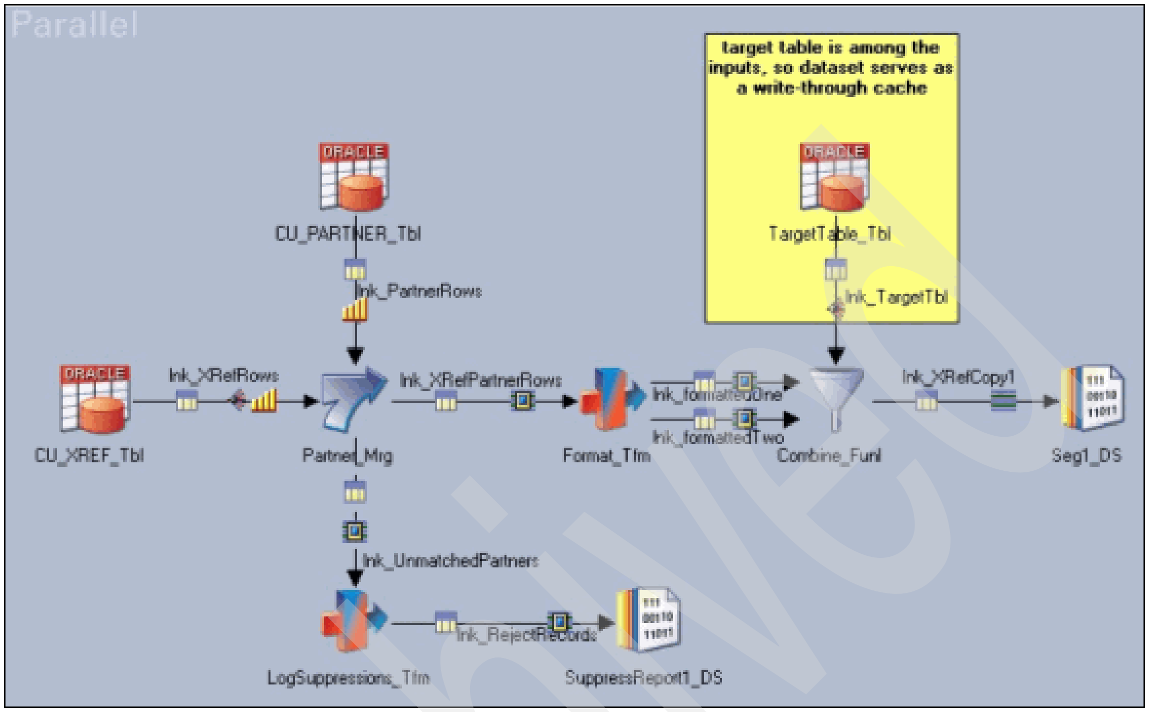


Figure 2-3 Sample transformation job with write-through cache

In Figure 2-4, the sample transformation job does not produce write-through cache. Its sources do not include the target table.

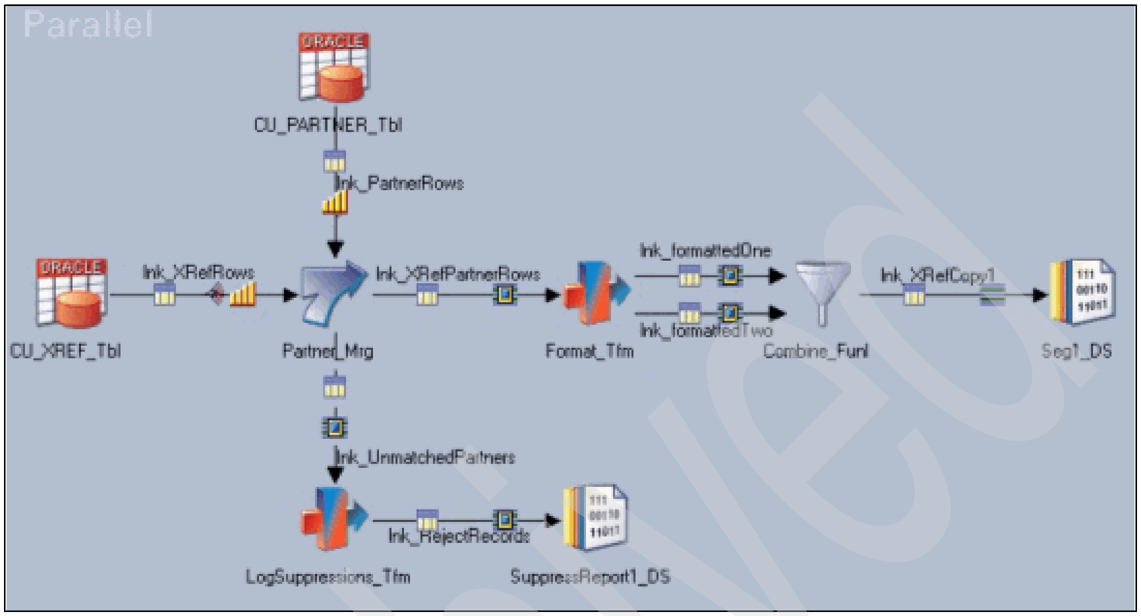


Figure 2-4 Sample transformation job without write-through cache

2.2.2 Hybrid jobs

The sample hybrid job depicted in Figure 2-5 demonstrates a job that transforms and combines source data, creating a dataset of target rows, and loading the target table. Because the target table is not sourced, this is an insert of new rows instead of creating a complete write-through cache.

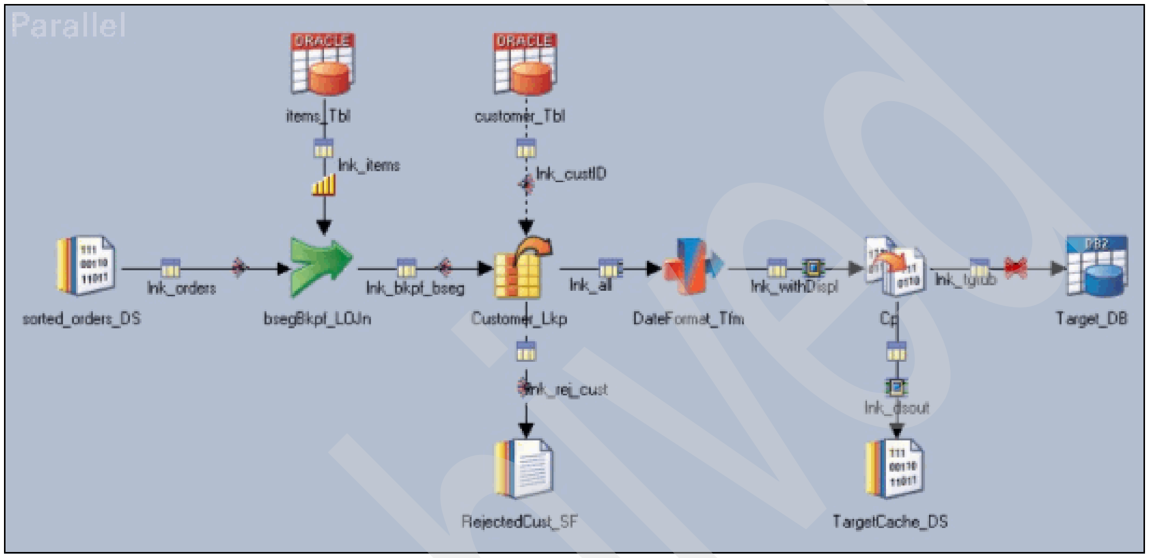


Figure 2-5 Sample hybrid job

2.2.3 Provisioning jobs

The sample provisioning job depicted in Figure 2-6 demonstrates the straightforward approach to simple provisioning tasks. In general, it is a good idea to separate the loading of a database from the ETL process, as a database load can often fail due to external reasons. Creating the load-ready dataset allows the provisioning job to be rerun without reprocessing the source extracts and transformations.

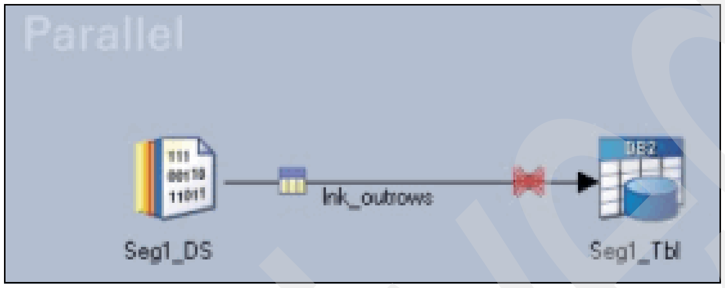


Figure 2-6 Sample provisioning job

3

Standards

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.

3.1 Directory structures

IBM InfoSphere DataStage (DS) requires file systems to be available for the following elements:

- Software Install Directory

  IBM InfoSphere DataStage executables, libraries, and pre-built components.

- DataStage Project Directory

  Runtime information, such as compiled jobs, OSH (Orchestrate® Shell) scripts, generated BuildOps and Transformers, and logging information.

- Data Storage

  This includes the following storage types:

  – DataStage temporary storage: Scratch, temp, buffer
  – DataStage parallel dataset segment files
  – Staging and archival storage for any source files

By default, these directories (except for file staging) are created during installation as subdirectories under the base InfoSphere DataStage installation directory.

3.1.1 Metadata layer

In addition to the file systems listed in the previous section, a DataStage project also requires a proper amount of space in the Metadata layer (which is a relational database system). As opposed to the Project directory, the Metadata layer stores the design time information, including job and sequence designs, table definitions, and so on.

In this section we do not discuss requirements and recommendations for other Information Server layers (Metadata and Services). The discussion here is strictly about the Engine layer.

A single Information Server instance (Services + Metadata) might manage multiple Engines. The following discussion pertains to the setup of a single Engine instance.

3.1.2 Data, install, and project directory structures

In Figure 3-1 on page 24 we illustrate how you might configure the file systems to satisfy the requirements of each class of DataStage storage. These directories are configured during product installation.

By default, the DataStage Administrator client creates its projects (repositories) in the Projects directory of the DataStage installation directory. In general, it is a bad practice to create DataStage projects in the default directory, as disk space is typically limited in production install directories. For this reason, a separate file system is created and mounted over the Projects subdirectory.

DataStage parallel configuration files are used to assign resources (such as processing nodes, disk, and scratch file systems) at runtime when a job is executed. The DataStage administrator creates parallel configuration files that define the degree of parallelism (number of nodes, node pools), and resources used. Parallel configuration files are discussed in detail in the OEM document *Orchestrate 7.5 User Guide*, which can be obtained from the IBM CEDI portal, located at the following web page:

http://www.haifa.ibm.com/ilsl/metadata/cedi.shtml

Data file systems store individual segment files of DataStage parallel datasets. Scratch file systems are used by the DataStage parallel framework to store temporary files such as sort and buffer overflow.

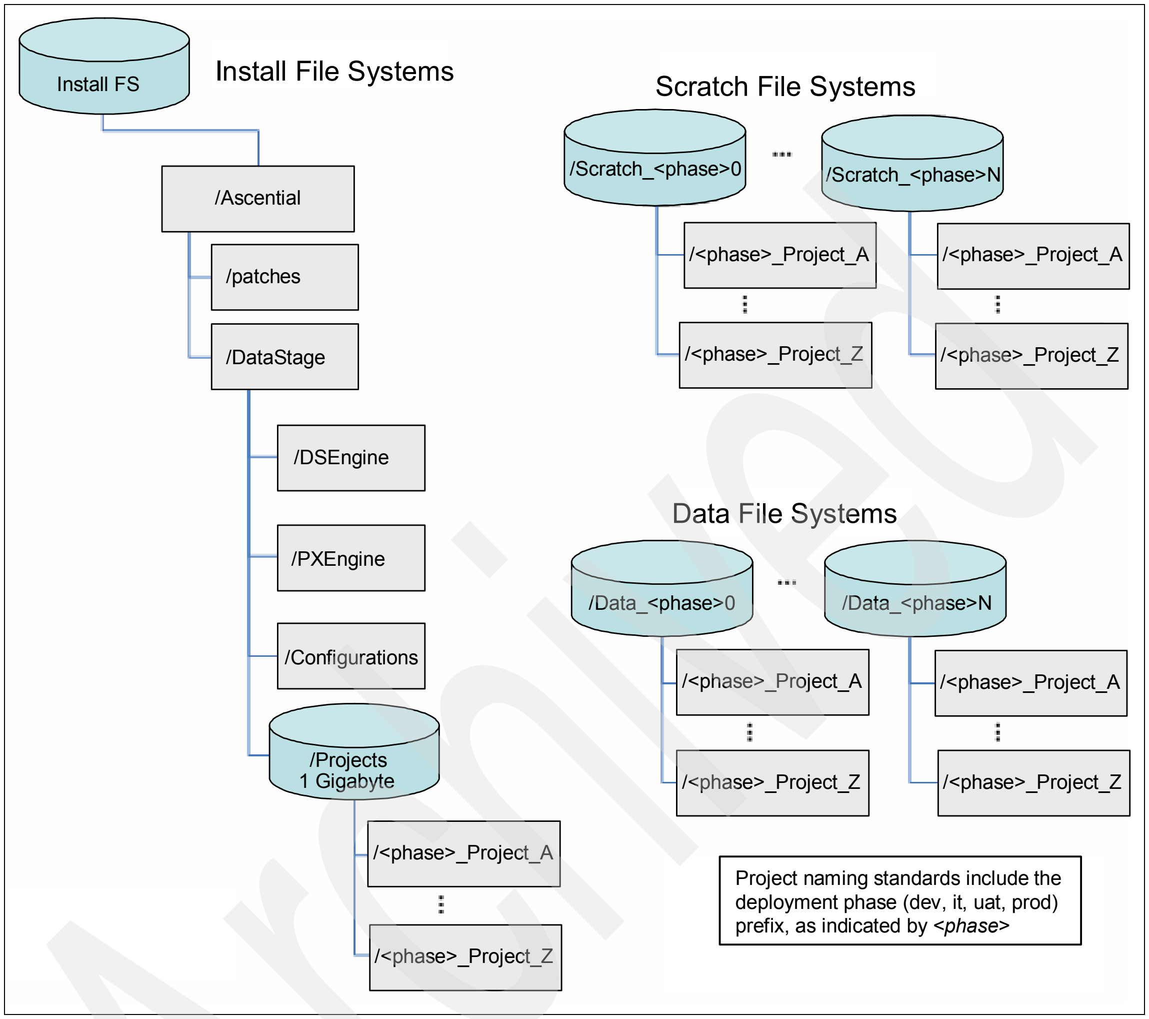


Figure 3-1 Suggested DataStage install, scratch, and data directories

3.1.3 Extending the DataStage project for external entities

It is suggested that another directory structure, referred to as Project_Plus, be created to integrate all aspects of a DataStage application that are managed outside of the DataStage Projects repository. The Project_Plus hierarchy includes directories for secured parameter files, dataset header files, custom components, Orchestrate schema, SQL and shell scripts. It might also be useful to support custom job logs and reports.

The Project_Plus directories provide a complete and separate structure in the same spirit as a DataStage project, organizing external entities in a structure that is associated with one corresponding DataStage project. This provides a convenient vehicle to group and manage resources used by a project.

It is quite common for a DataStage application to be integrated with external entities, such as the operating system, enterprise schedulers and monitors, resource managers, other applications, and middle ware. The Project_Plus directory provides an extensible model that can support this integration through directories for storing source files, scripts, and other components.

Project_Plus and change management

Project naming conventions recommend naming a project with a prefix to indicate the deployment phase (dev, it, uat, and prod). Following this naming convention also separates the associated files in the corresponding Project_Plus hierarchy.

However, to isolate support files completely and in a manner that is easy to assign to separate file systems, an additional level of directory structure can be used to enable multiple phases of application deployment (development, integration testing, user acceptance testing, and production) as appropriate. If the file system is not shared across multiple servers, not all of these development phases might be present on a local file system.

Project_Plus file system

The Project_Plus directory is often stored in the `/usr/local` home directory (for example, `/usr/local/dstage`), but this can be in any location as long as permissions and file system access are permitted to the DataStage developers and applications.

Note: The file system where the Project_Plus hierarchy is stored must be expandable without requiring destruction and re-creation.

Project_Plus directory structure

Figure 3-2 shows typical components and the structure of the Project_Plus directory hierarchy.

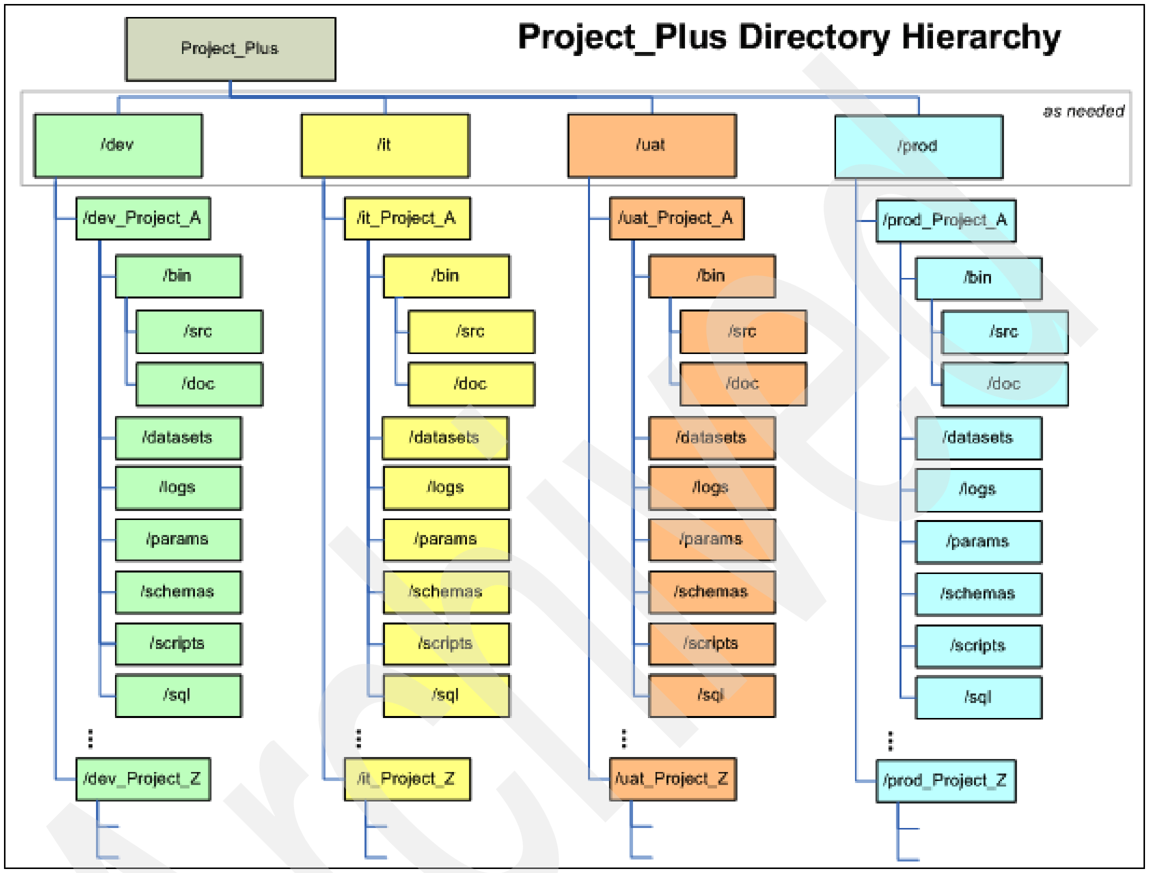


Figure 3-2 Project_Plus directory structure

In Table 3-1 we list the Project_Plus variable descriptions.

Table 3-1 Project_Plus variable descriptions

Directory	**Description**
Project_Plus	Top-level of directory hierarchy
/dev	Development phase directory tree (if applicable)
/dev_Project_A	Subdirectory created for each DataStage project (the actual directory name "dev_Project_A" should match the corresponding DataStage Project Name).
/bin	Location of custom programs, DataStage routines, BuildOps, utilities, and shells
/doc	Documentation for programs in /bin subdirectory
/src	Source code and Makefiles for items in /bin subdirectory Depending on change management policies, this directory might only be present in the /dev development phase directory tree
/datasets	Location of parallel dataset header files (.ds files)
/logs	Location of custom job logs and reports
/params	Location of parameter files for automated program control, a backup copy of *dsenv* and backup copies of DSParams:$ProjectName project files
/schemas	Location of Orchestrate schema files
/it	Integration Test phase directory tree (if applicable)
/uat	User Acceptance Test phase directory tree (if applicable)
/prod	Production phase directory tree (if applicable)

Project_Plus environment variables

The Project_Plus directory structure is made to be transparent to the DataStage application, through the use of environment variable parameters used by the DataStage job developer. Environment variables are a critical portability tool that enables DataStage applications to be deployed through the life cycle without any code changes.

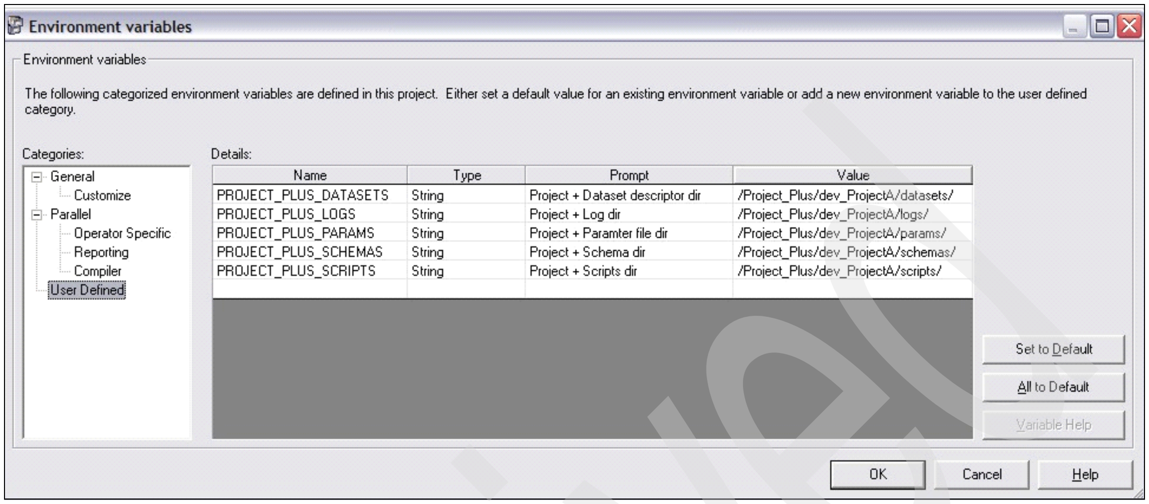
In support of a Project_Plus directory structure, the user-defined environment variable parameters, depicted in Table 3-2, is configured for each project using the DataStage Administrator, substituting your Project_Plus file system and project name in the value column:

Table 3-2 Project_Plus environment variables

Name	Type	Prompt	Example Value
PROJECT_PLUS_DATASETS	String	Project + Dataset descriptor dir	/Project_Plus/devProject_A/datasets/
PROJECT_PLUS_LOGS	String	Project + Log dir	/Project_Plus/devProject_A/logs/
PROJECT_PLUS_PARAMS	String	Project + Parameter file dir	/Project_Plus/devProject_A/params/
PROJECT_PLUS_SCHEMAS	String	Project + Schema dir	/Project_Plus/devProject_A/schemas/
PROJECT_PLUS_SCRIPTS	String	Project + Scripts dir	/Project_Plus/devProject_A/scripts/

Note: the PROJECT_PLUS default values include a trailing directory separator, to avoid having to specify in the stage properties. This is optional, but whichever standard the administrator chooses, it is consistently deployed across projects and job designs.

The Project_Plus environment variables are depicted in Figure 3-3 as they would appear when loaded into the project dataset.


Environment variables

Environment variables

The following categorized environment variables are defined in this project. Either set a default value for an existing environment variable or add a new environment variable to the user defined category.

Categories:
- General
  - Customize
- Parallel
  - Operator Specific
  - Reporting
  - Compiler
- User Defined

Details:

Name	Type	Prompt	Value
PROJECT_PLUS_DATASETS	String	Project + Dataset descriptor dir	/Project_Plus/dev_ProjectA/datasets/
PROJECT_PLUS_LOGS	String	Project + Log dir	/Project_Plus/dev_ProjectA/logs/
PROJECT_PLUS_PARAMS	String	Project + Paramter file dir	/Project_Plus/dev_ProjectA/params/
PROJECT_PLUS_SCHEMAS	String	Project + Schema dir	/Project_Plus/dev_ProjectA/schemas/
PROJECT_PLUS_SCRIPTS	String	Project + Scripts dir	/Project_Plus/dev_ProjectA/scripts/

Figure 3-3 Project_Plus environment variables

In parallel job designs, the Project_Plus parameters are added as job parameters using the $PROJDEF default value. These parameters are used in the stage properties to specify the location of DataSet header files, job parameter files, orchestrate schemas, and external scripts in job flows.

Using Project_Plus with grid or cluster deployments

When deploying a DataStage application in cluster or grid environments, or when configuring for high availability and disaster recovery scenarios, careful consideration is made when sharing the Project_Plus file system configuration.

In general, custom components, dataset header files, and other components of the Project_Plus directory must be visible to all members of the cluster or grid, using the same mount point on all servers. Creation of small individual mount points is generally not desirable.

Mount this directory on all members of the cluster after installing IBM InfoSphere DataStage, but before creating any DataSets.

3.1.4 File staging

Project naming conventions recommend naming a project with a suffix to indicate the deployment phase (dev, it, uat, prod). Following this naming convention also separates the associated files in the corresponding Staging hierarchy.

However, to completely isolate support files in a manner that is easy to assign to separate file systems, an additional level of directory structure can be used to enable multiple phases of application deployment (development, integration test, user acceptance test, and production) as appropriate. If the file system is not shared across multiple servers, not all of these development phases might be present on a local file system.

In support of the Staging directory structure, the user-defined environment variable parameters, shown in Table 3-3, is configured for each project using the DataStage Administrator, substituting your staging file system and project name in the value column.

Table 3-3 Environment variable parameters

Name	**Type**	**Prompt**	**Example Value**
STAGING_DIR	String	Staging directory	/Staging/
PROJECT_NAME	String	Project name	devProject_A
DEPLOY_PHASE	String	Deployment phase	dev

The Project_Name and Deploy_Phase variables are used to parameterize the directory location in job designs properly.

Note: The STAGING_DIR default value includes a trailing directory separator, to avoid having to specify in the stage properties. This is optional, but whatever standard the administrator chooses, it must be consistently deployed across projects and job designs.

It is suggested that a separate staging file system and directory structure be used for storing, managing, and archiving various source data files, as illustrated in Figure 3-4 on page 31.

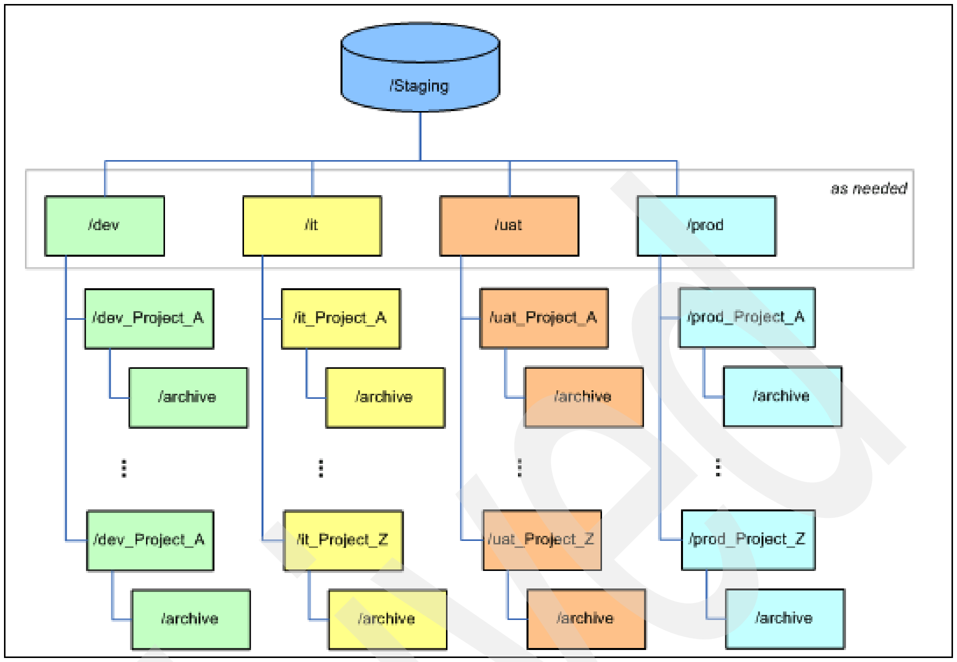


Figure 3-4 DataStage staging directories

In each deployment directory, files are separated by project name. See Table 3-4.

Table 3-4 Directory file descriptions

Directory	**Description**
Staging	Top-level of directory hierarchy
/dev	Development phase directory tree (if applicable)
/dev_Project_A	Subdirectory created for each DataStage project (the actual directory name `dev_Project_A` should match the corresponding DataStage Project Name) location of source data files, target data files, error and reject files
/archive	Location of compressed archives created by archive process of previously processed files
/it	Integration Test phase directory tree (if applicable)
/uat	User Acceptance Test phase directory tree (if applicable)
/prod	Production phase directory tree (if applicable)

3.2 Naming conventions

As a graphical development environment, DataStage offers (in certain restrictions) flexibility to developers when naming various objects and components used to build a data flow. By default, the Designer tool assigns default names based on the object type, and the order the item is placed on the design canvas. Though the default names might create a functional data flow, they do not facilitate ease of maintenance over time, nor do they adequately document the business rules or subject areas. Providing a consistent naming standard is essential to achieve the following results:

- Maximize the speed of development
- Minimize the effort and cost of downstream maintenance
- Enable consistency across multiple teams and projects
- Facilitate concurrent development
- Maximize the quality of the developed application
- Increase the readability of the objects in the visual display medium
- Increase the understanding of components when seen in external systems (for example in IBM InfoSphere Business Glossary, Metadata Workbench, or an XML extract)

In this section we present a set of standards and guidelines to apply to developing data integration applications using IBM InfoSphere DataStage.

Any set of standards needs to take on the culture of an organization, to be tuned according to needs, so it is envisaged that these standards shall develop and adapt over time to suit both the organization and the purpose.

Throughout this section, the term *standard* refers to those principles that are required. The term *guideline* refers to recommended, but not required, principles.

3.2.1 Key attributes of the naming convention

This naming convention is based on the following three-part convention:

Subject, Subject Modifier, and Class Word

In a DataStage job, there might be multiple stages that correlate to a certain area or task. The naming convention described in the previous section is meant to reflect that logical organization in a straightforward way. A *subject* corresponds to a so-called area in a job, a *subject modifier* corresponds to a more specific operation in that area, and the *class word* indicates the nature of an object.

One example might be a job in which records for data sources such as accounts and customers are prepared, correlated, credit scores calculated and results written out as a parallel dataset. In this scenario, as part of the same job, we can identify a few subjects, such as accounts, customers, and the report generation. Subject modifiers denote the subset of the logic inside that area. Class word indicates the type of stage or link.

A few examples, in which subject, subject modifier and class word are separated by underscores, are shown in Table 3-5.

Table 3-5 Subject examples

Object Name	Description
ACCT_FMTCUSTID_TFM	A Transformer (TFM) that formats the Customer ID of account records
RPT_ACCTCUST_LOJN	A Left-Outer Join stage (LOJN) that correlates account and customer records for the report
RPT_SCORE_TFM	A Transformer that calculates credit scores
RPT_EXPORT_DS	A Dataset stage that exports the report results

In the context of DataStage, the class word is used to identify either a type of object, or the function that a particular type of object performs. In certain cases objects can be sub-typed (for example, a Left Outer Join). In these cases the class word represents the subtype.

For example, in the case of a link object, the class word refers to the functions of reading, reference (Lookup), moving or writing data (or in a Sequence Job, the moving of a message).

In the case of a data store the class word refers to the type of data store (as examples Dataset, Sequential File, Table, View, and so forth).

Where there is no sub classification required, the class word refers to the object. As an example, a Transformer might be named `Data_Block_Split_Tfm`.

As a guideline, the class word is represented as a two, three, or four letter abbreviation. If it is a three or four letter abbreviation, it is word capitalized. If it is a two letter abbreviation, both letters are capitalized.

A list of frequently-used class word abbreviations is provided in Appendix C, “DataStage naming reference” on page 391.

One benefit of using the subject, subject modifier, class word approach instead of using the prefix approach, is to enable two levels of sorting or grouping. In InfoSphere Metadata Workbench, the object type is defined in a separate field. There is a field that denotes whether the object is a column, a derivation, a link, a stage, a job design, and so forth. This is the same or similar information that is carried in a prefix approach. Carrying this information as a separate attribute enables the first word of the name to be used as the subject matter, allowing sort either by subject matter or by object type. In addition, the class word approach enables sub-classification by object type to provide additional information.

For the purposes of documentation, all word abbreviations are referenced by the long form to get used to saying the name in full even if reading the abbreviation. Like a logical name, however, when creating the object, the abbreviated form is used. This re-enforces wider understanding of the subjects.

The key issue is readability. Though DataStage imposes limitations on the type of characters and length of various object names, the standard, where possible, is to separate words by an underscore, which allows clear identification of each work in a name. This is enhanced by also using word capitalization (for example, the first letter of each word is capitalized).

3.2.2 Designer object layout

The effective use of naming conventions means that objects need to be spaced appropriately on the DataStage Designer canvas. For stages with multiple links, expanding the icon border can significantly improve readability. This approach takes extra effort at first, so a pattern of work needs to be identified and adopted to help development. The Snap to Grid feature of Designer can improve development speed.

When development is more or less complete, attention must be given to the layout to enhance readability before it is handed over to versioning.

Where possible, consideration must be made to provide DataStage developers with higher resolution screens, as this provides them with more monitor display real-estate. This can help make them more productive and makes their work more easily read.

3.2.3 Documentation and metadata capture

One of the major problems with any development effort, whatever tool you use, is maintaining documentation. Despite best intentions, and often due to time constraints, documentation is often something that is left until later or is inadequately implemented. Establishing a standard method of documentation with examples and enforcing this as part of the acceptance criteria is strongly recommended. The use of meaningful naming standards (as outlined in this section) compliments these efforts.

DataStage provides the ability to document during development with the use of meaningful naming standards (as outlined in this section). Establishing standards also eases use of external tools and processes such as InfoSphere Metadata Workbench, which can provide impact analysis, as well as documentation and auditing.

3.2.4 Naming conventions by object type

In this section we describe the object type naming conventions.

Projects

Each DataStage Project is a standalone repository. It might have a one-to-one relationship with an organizations' project of work. This factor can cause terminology issues especially in teamwork where both business and developers are involved.

The name of a DataStage project is limited to a maximum of 18 characters. The project name can contain alphanumeric characters and underscores.

Projects names must be maintained in unison with source code control. As projects are promoted through source control, the name of the phase and the project name should reflect the version, in the following form:

```
<Phase>_<ProjectName>_<version>
```

In this example, `Phase` corresponds to the phase in the application development life cycle, as depicted in Table 3-6.

Table 3-6 Project phases

Phase Name	Phase Description
Dev	Development
IT	Integration Test
UAT	User Acceptance Test
Prod	Production

Development Projects

Shared development projects should contain the phase in the life cycle, the name, and a version number. Examples are shown in the following list:

- Dev_ProjectA_p0 (initial development phase 0…phase N)
- Dev_ProjectA_v1 (maintenance)
- Dev_ProjectA_v2

Individual developers can be given their own sandbox projects, which should contain the user ID or initials, the application name, the phase in the life cycle and a version number. This is difficult to do with 18 characters. The following list shows some examples:

- JoeDev_ProjectA_v2
- SueDev_ProjectA_v1

Test projects

Test project names should contain the phase in the life cycle, project name, and version. The following project names are intended for Integration Testing (IT) and User Acceptance Testing (UAT):

- IT_ProjectA_v1_0 (first release)
- IT_ProjectA_v1_1 (patch or enhancement)
- UAT_ProjectA_v1_0

Production projects

Although it is preferable to create a new project for each minor and major change to a production project, making a change to the project name could require changes to external objects. For example, an enterprise scheduler requires the project name. Therefore, it is not a requirement that a project name contain version information.

Using version numbers could allow you to run parallel versions of a DataStage application, without making changes to the always-on system.

The following list shows examples of acceptable names for the production project:

- Prod_ProjectA_v1
- ProjectA_v1
- ProjectA

The following examples are project names where the project is single application focused:

- Accounting Engine NAB Development is named Dev_AcctEngNAB_v1_0
- Accounting Engine NAB Production is named Prod_AcctEngNAB

The following examples are project names where the project is multiapplication focused:

- Accounting Engine Development or Dev_AcctEngine_v1_0
- Accounting Engine Production or Prod_AcctEngine

Folder hierarchy

The DataStage repository is organized in a folder hierarchy, allowing related objects to be grouped together. Folder names can be long, are alpha numeric and can also contain both spaces and underscores. Therefore, directory names are word capitalized and separated by either an underscore or a space.

DataStage 7.5 enforced the top level directory structure for various types of objects, such as jobs, routines, shared containers, and table definitions. Developers had the flexibility to define their own directory or category hierarchy beneath that level.

With Information Server 8, objects can be organized in an arbitrary hierarchy of folders, each folder containing objects of any type. The repository top level is not restricted to a set of fixed folder names. New top level folders can be added, such as one per user, subject area, development phase, and so on.

Figure 3-5 presents the top level view, with a list of default folders. As stated before, objects are not restricted to a top level folder named after the corresponding type.

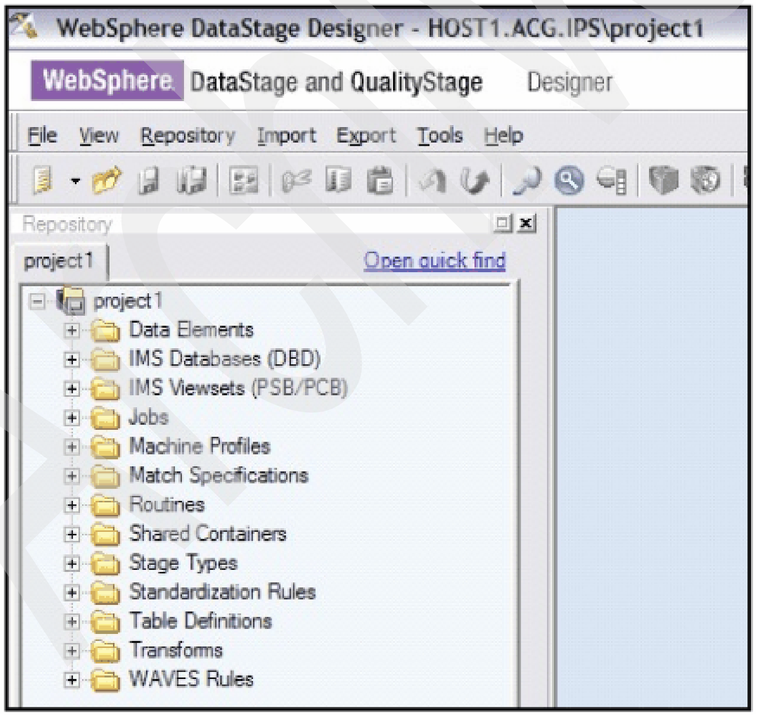
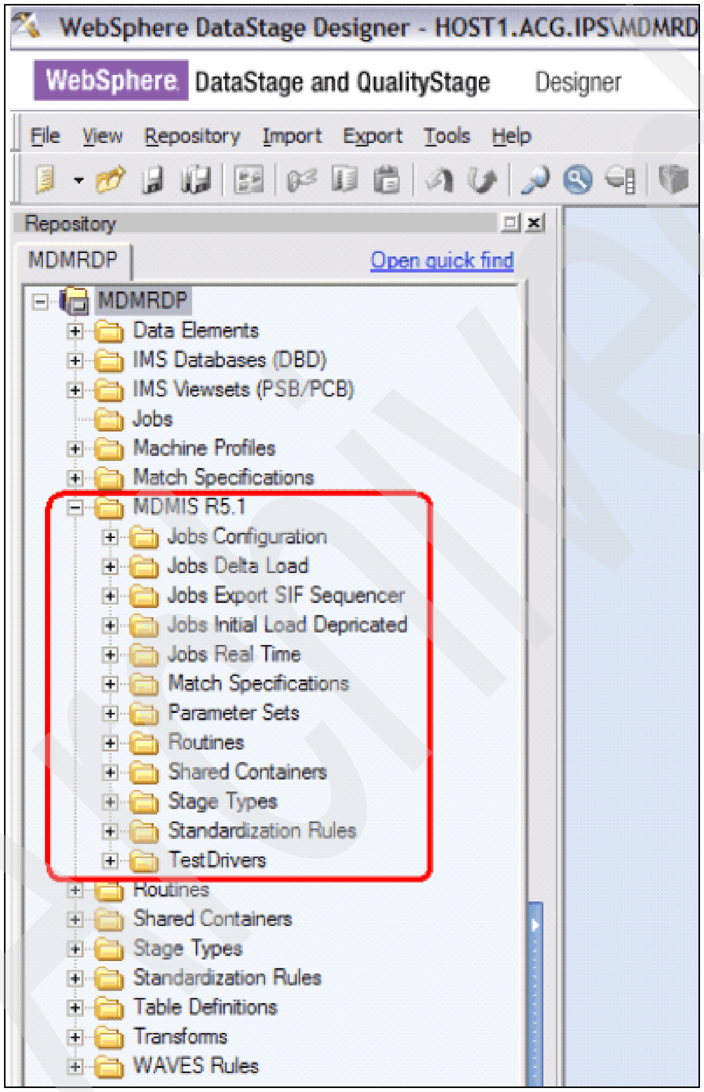


Figure 3-5 Default top level folders

Figure 3-6 shows an example of a custom top-level folder that aggregates objects of several types.

Information Server 8 maintains the restriction that there can only be a single object of a certain type with a given name.


WebSphere DataStage Designer - HOST1.ACG.IPS\MDMRD

WebSphere DataStage and QualityStage Designer

File View Repository Import Export Tools Help

Repository

MDMRDP Open quick find

- MDMRDP
  - Data Elements
  - IMS Databases (DBD)
  - IMS Viewsets (PSB/PCB)
  - Jobs
  - Machine Profiles
  - Match Specifications
  - MDMIS R5.1
    - Jobs Configuration
    - Jobs Delta Load
    - Jobs Export SIF Sequencer
    - Jobs Initial Load Depricated
    - Jobs Real Time
    - Match Specifications
    - Parameter Sets
    - Routines
    - Shared Containers
    - Stage Types
    - Standardization Rules
    - TestDrivers
  - Routines
  - Shared Containers
  - Stage Types
  - Standardization Rules
  - Table Definitions
  - Transforms
  - WAVES Rules

Figure 3-6 A custom top-level repository folder

Object creation

In Information Server 8, object creation is simplified. To create a new object, right-click the target parent folder, select **New** and the option for the desired object type, as shown in Figure 3-7. Objects of any type can be created in any folder in the repository hierarchy.

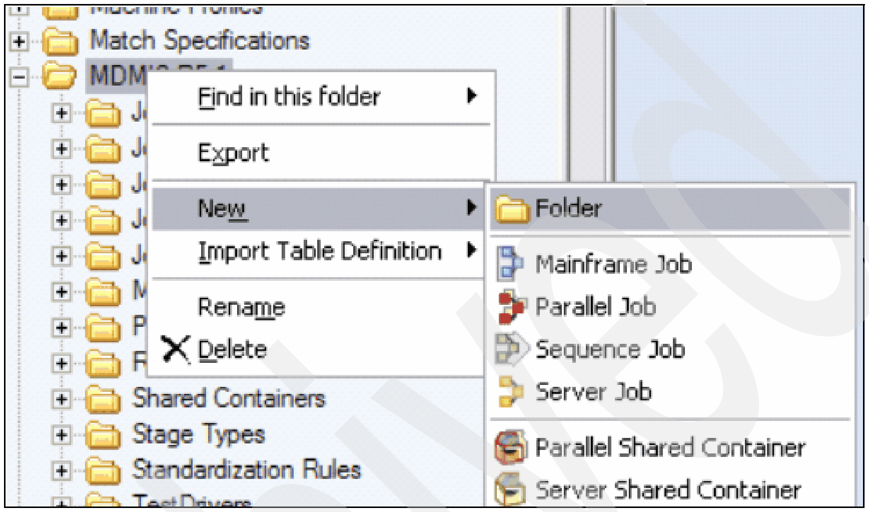


Figure 3-7 Creating a new folder

Categorization by functional module

For a given application or functional module, all objects can be grouped in a single top-level folder, with sub-levels for separate object types, as in Figure 3-6 on page 38. Job names must be unique in a DataStage project, not only in a folder.

Categorization by developer

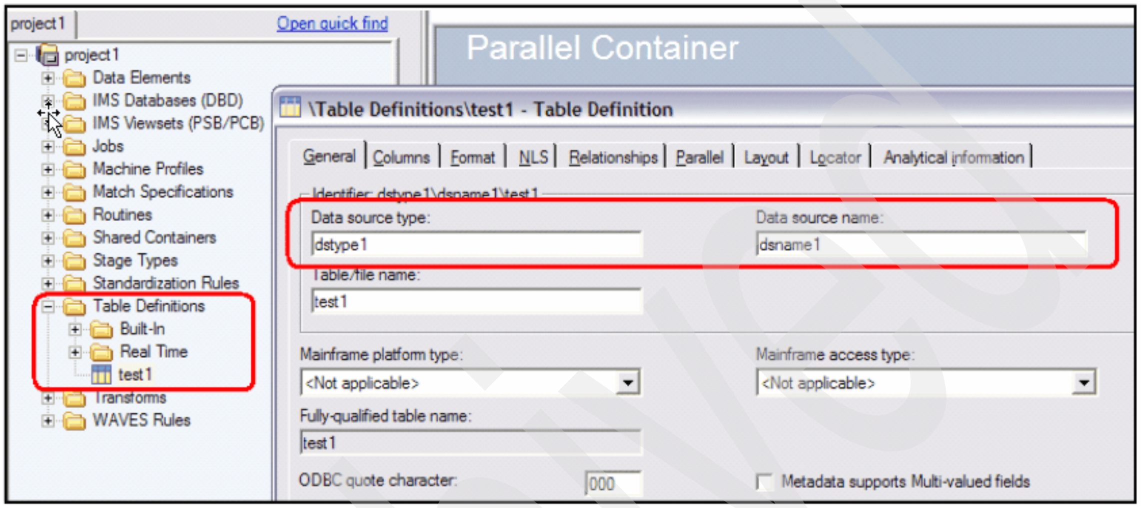
In development projects, folders might be created for each developer as their personal sandbox. That is the place where they perform unit test activities on jobs they are developing.

It is the responsibility of each developer to delete unused or obsolete code. The development manager, to whom is assigned the DataStage Manager role, must ensure that projects are not inflated with unused objects (such as jobs, sequences, folders, and table definitions).

Again, object names must be unique in a given project for the given object type. Two developers cannot save a copy of the same job with the same name in their individual sandbox categories. A unique job name must be given.

Table definition categories

Unlike DataStage 7.5, in which table definitions were categorized using two level names (based on the data source type and the data source name), Information Server 8 allows them to be placed anywhere in the repository hierarchy. This is depicted in Figure 3-8.



- project1
  - Data Elements
  - IMS Databases (DBD)
  - IMS Viewsets (PSB/PCB)
  - Jobs
  - Machine Profiles
  - Match Specifications
  - Routines
  - Shared Containers
  - Stage Types
  - Standardization Rules
  - Table Definitions
    - Built-In
    - Real Time
    - test1
  - Transforms
  - WAVES Rules

\Table Definitions\test1 - Table Definition

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

Identifier: dstype1\dsname1\test1

Data source type: dstype1

Data source name: dsname1

Table/file name: test1

Mainframe platform type: <Not applicable>

Mainframe access type: <Not applicable>

Fully-qualified table name: test1

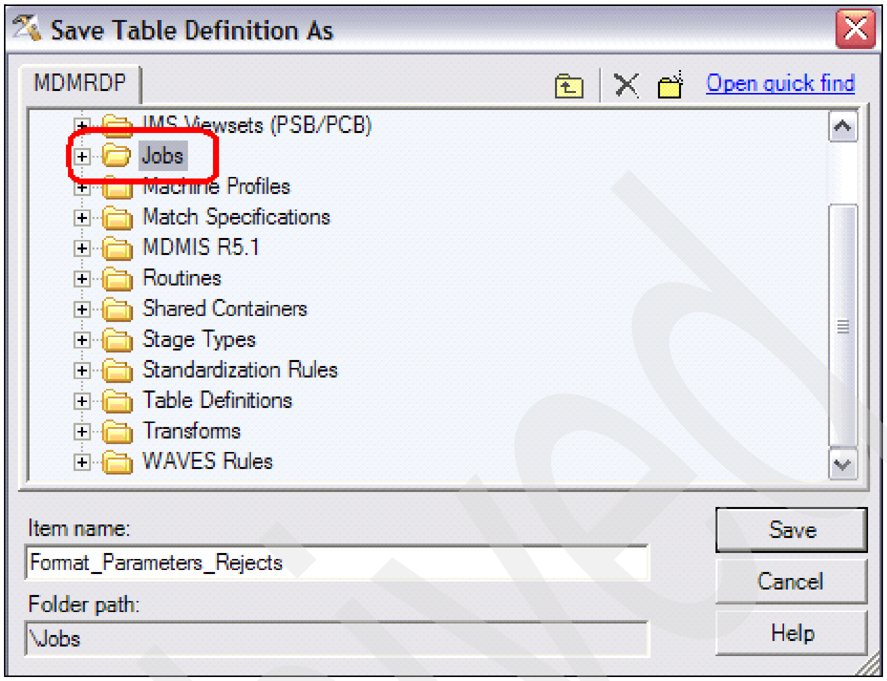
ODBC quote character: 000

Metadata supports Multi-valued fields

Figure 3-8 A table definition in the repository hierarchy

Figure 3-9 on page 41 shows an example of a table definition stored directly underneath Table Definitions. Its data source type and data source name properties do not determine names for parent subfolders.

When saving temporary TableDefs (usually created from output link definitions to assist with job creation), developers are prompted for the folder in the "Save Table Definition As" window. The user must pay attention to the folder location, as these objects are no longer stored in the Table Definition category by default.



Item name: Format_Parameters_Rejects

Folder path: \Jobs

Figure 3-9 Table definition categories

Jobs and job sequences

Job names must begin with a letter and can contain letters, numbers, and underscores only. Because the name can be long, job and job sequence names must be descriptive and should use word capitalization to make them readable.

Jobs and job sequences are all held under the Category Directory Structure, of which the top level is the category Jobs.

A job is suffixed with the class word Job and a job sequence is suffixed with the class word Seq.

The following items are examples of job naming:

- CodeBlockAggregationJob
- CodeBlockProcessingSeq

Jobs must be organized under category directories to provide grouping such that a directory should contain a sequence job and all the jobs that are contained in that sequence. This is discussed further in “Folder hierarchy” on page 37.

Shared containers

Shared containers have the same naming constraints as jobs in that the name can be long but cannot contain underscores, so word capitalization must be used for readability. Shared containers might be placed anywhere in the repository tree and consideration must be given to a meaningful directory hierarchy. When a shared container is used, a character code is automatically added to that instance of its use throughout the project. It is optional as to whether you decide to change this code to something meaningful.

To differentiate between parallel shared containers and server shared containers, the following class word naming is recommended:

- Psc = Parallel Shared Container
- Ssc = Server Edition Shared Container

> **Note:** Use of Server Shared Containers is discouraged in a parallel job.

Examples of Shared Container naming are as follows:

- AuditTrailPsc (original naming as seen in the Category Directory)
- AuditTrailPscC1 (an instance of use of the previously mentioned shared container)
- AuditTrailPscC2 (another instance of use of the same shared container)

In the aforementioned examples the characters C1 and the C2 are automatically applied to the Shared Container stage by DataStage Designer when dragged onto the design canvas.

Parameters

A parameter can be a long name consisting of alphanumeric characters and underscores. The parameter name must be made readable using capitalized words separated by underscores. The class word suffix is `parm`.

The following examples are of parameter naming:

- Audit_Trail_Output_Path_parm
- Note where this is used in a stage property, the parameter name is delimited by the # sign:

```
#Audit_Trail_Output_Path_parm#
```

Links

In a DataStage job, links are objects that represent the flow of data from one stage to the next. In a job sequence, links represent the flow of a message from one activity or step to the next.

It is particularly important to establish a consistent naming convention for link names, instead of using the default DSLink# (where # is an assigned number). In the graphical Designer environment, stage editors identify links by name. Having a descriptive link name reduces the chance for errors (for example, during link ordering). Furthermore, when sharing data with external applications (for example, through job reporting), establishing standardized link names makes it easier to understand results and audit counts.

To differentiate link names from stage objects, and to identify in captured metadata, the prefix `lnk_` is used before the subject name of a link.

The following rules can be used to establish a link name:

- The link name should define the subject of the data that is being moved.
- For non-stream links, the link name should include the link type (reference, reject) to reinforce the visual cues of the Designer canvas:
  - `Ref` for reference links (Lookup)
  - `Rej` for reject links (such as Lookup, Merge, Transformer, Sequential File, and Database)
- The type of movement might optionally be part of the Class Word. As examples:
  - `In` for input
  - `Out` for output
  - `Upd` for updates
  - `Ins` for inserts
  - `Del` for deletes
  - `Get` for shared container inputs
  - `Put` for shared container output
- As data is enriched through stages, the same name might be appropriate for multiple links. In this case, specify a unique link name in a particular job or job sequence by including a number. (The DataStage Designer does not require link names on stages to be unique.)

  The following list provides sample link names:

  - Input Transactions: *lnk_Txn_In*
  - Reference Account Number Rejects: *lnk_Account_Ref_Rej*
  - Customer File Rejects: *lnk_Customer_Rej*

Stage names

DataStage assigns default names to stages as they are dragged onto the Designer canvas. These names are based on the type of stage (Object) and a unique number, based on the order the object was added to the flow. In a job or job sequence, stage names must be unique.

Instead of using the full object name, a 2, 3, or 4 character abbreviation must be used for the class word suffix, after the subject name and subject modifier. A list of frequently-used stages and their corresponding class word abbreviation can be found in Appendix C, “DataStage naming reference” on page 391.

Sequencer object naming

In a job Sequencer, links are actually messages. Proceed sequencer links with the class word `msg_` followed by the type of message (as examples, fail and unconditional), and followed by the ClassName. The following lists shows some examples:

- Reception Succeeded Message: `msg_ok_Reception`
- Reception Failed Message: `msg_fail_Reception`

Data stores

For the purposes of this section, a data store is a physical piece of disk storage where data is held for a period of time. In DataStage terms, this can be either a table in a database structure or a file contained in a disk directory or catalog structure. Data held in a database structure is referred to as either a table or a view. In data warehousing, two additional subclasses of table might be used: dimension and fact. Data held in a file in a directory structure is classified according to its type, for example: Sequential File, Parallel Dataset, Lookup File Set, and so on.

The concepts of “source” and “target” can be applied in a couple of ways. Every job in a series of jobs could consider the data it gets in to be a source and the data it writes out as being a target. However, for the sake of this naming convention a source is only data that is extracted from an original system. A target is the data structures that are produced or loaded as the final result of a particular series of jobs. This is based on the purpose of the project: to move data from a source to a target.

Data stores used as temporary structures to land data between jobs, supporting restart and modularity, should use the same names in the originating job and any downstream jobs reading the structure.

Examples of data store naming are as follows:

- Transaction Header Sequential File or `Txn_Header_SF`
- Customer Dimension or `Cust_Dim` (This optionally can be further qualified as `Cust_Dim_Tgt` if you want to qualify it as a final target)
- Customer Table or `Cust_Tab`
- General Ledger Account Number View or *`GL_Acctno_View`*

Transformer stage and stage variables

A Transformer stage name can be long (over 50 characters) and can contain underscores. Therefore, the name can be descriptive and readable through word capitalization and underscores. DataStage supports two types of Transformers:

- Tfm: Parallel Transformer
- BTfm: BASIC Transformer

> **Note:** For maximum performance and scalability, BASIC Transformers must be avoided in DS parallel data flows.

A Transformer stage variable can have a long name (up to 32 characters) consisting of alphanumeric characters but not underscores. Therefore, the stage variable name must be made readable only by using capitalized words. The class word suffix is stage variable or SV. Stage variables must be named according to their purpose.

Examples of Transformer stage and stage variable naming are as follows:

- `Alter_Constraints_Tfm`
- `recNumSV`

When developing Transformer derivation expressions, it is important to remember stage variable names are case sensitive.

DataStage routines

DataStage BASIC routine names should indicate their function and be grouped in sub-categories by function under a main category of that corresponds to the subject area. For example:

```
Routines/Automation/SetDSParamsFromFile
```

A how-to document describing the appropriate use of the routine must be provided by the author of the routine, and placed in a documentation repository.

DataStage custom Transformer routine names should indicate their function and be grouped in sub-categories by function under a main category that corresponds to the subject area. For example:

```
Routines/Automation/DetectTeradataUnicode
```

Source code, a makefile, and the resulting object for each Custom Transformer routine must be placed in the Project_Plus source directory. For example:

```
/Project_Plus/projectA_dev/bin/source
```

File names

Source file names should include the name of the source database or system and the source table name or copybook name. The goal is to connect the name of the file with the name of the storage object on the source system. Source flat files have a unique serial number composed of the date, "_ETL_" and time. For example:

```
Client_Relationship_File1_In_20060104_ETL_184325.psv
```

Intermediate datasets are created between modules. Their names include the name of the module that created the dataset or the contents of the dataset in that more than one module might use the dataset after it is written. For example:

```
BUSN_RCR_CUST.ds
```

Target output files include the name of the target subject area or system, the target table name or copybook name. The goal is the same as with source files—to connect the name of the file with the name of the file on the target system. Target flat files have a unique serial number composed of the date, *_ETL_* and time. For example:

```
Client_Relationship_File1_Out_20060104_ETL_184325.psv
```

Files and datasets have suffixes that allow easy identification of the content and type. DataStage proprietary format files have required suffixes. They are identified in italics in Table 3-7, which defines the types of files and their suffixes.

Table 3-7 File suffixes

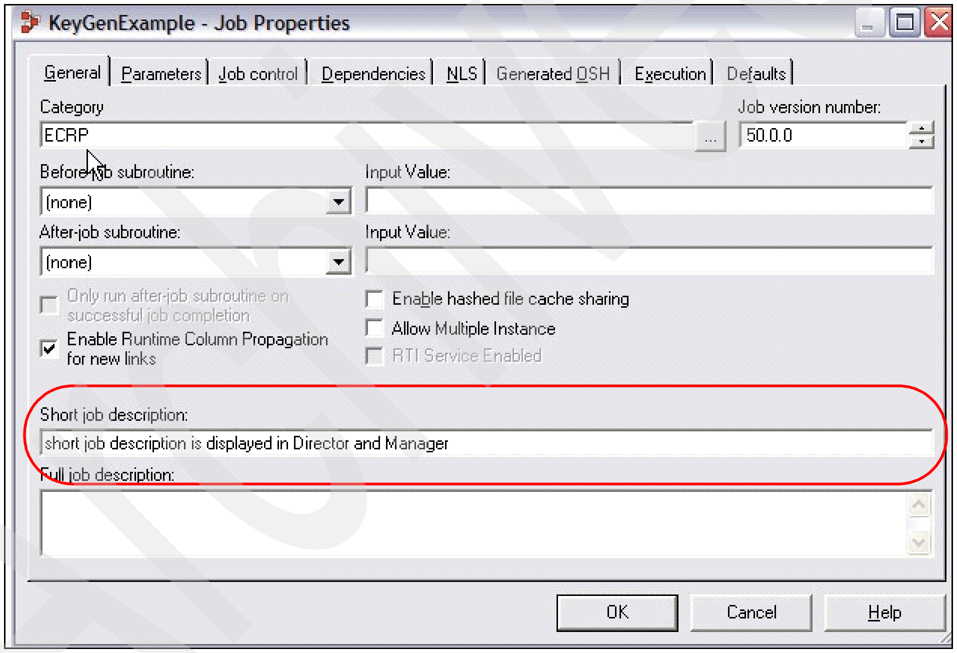
File Type	**Suffix**
Flat delimited and non-delimited files	.dat.
Flat pipe (|) delimited files	.psv
Flat comma-and-quote delimited files	.csv.
DataStage datasets	*.ds.*
DataStage filesets	*.fs*
DataStage hash files	*.hash.*
Orchestrate schema files	.schema.
Flat delimited or non-delimited REJECT files	.rej.
DataStage REJECT datasets	_rej.*ds*.
Flat delimited or non-delimited ERROR files	.err.
DataStage ERROR datasets	_err.*ds*.

Flat delimited or non-delimited LOG files	.log.

3.3 Documentation and annotation

DataStage Designer provides description fields for each object type. These fields allow the developer to provide additional descriptions that can be captured and used by administrators and other developers.

The “Short Description” field is also displayed on summary lines in the Director and Designer clients. At a minimum, although there is no technical requirement to do so, job developers should provide descriptive annotations in the “Short Description” field for each job and job sequence, as in Figure 3-10.



KeyGenExample - Job Properties
General | Parameters | Job control | Dependencies | NLS | Generated OSH | Execution | Defaults
OK
Cancel
Help

Category: ECRP
Job version number: 50.0.0
Before-job subroutine: (none)
Input Value:
After-job subroutine: (none)
Input Value:
[ ] Only run after-job subroutine on successful job completion
[✓] Enable Runtime Column Propagation for new links
[ ] Enable hashed file cache sharing
[ ] Allow Multiple Instance
[ ] RTI Service Enabled
Short job description: short job description is displayed in Director and Manager
Full job description:

Figure 3-10 Job level short description

In a job, the Annotation tool must be used to highlight steps in a given job flow. By changing the vertical alignment properties (for example, Bottom) the annotation can be drawn around the referenced stages, as in Figure 3-11 on page 48.

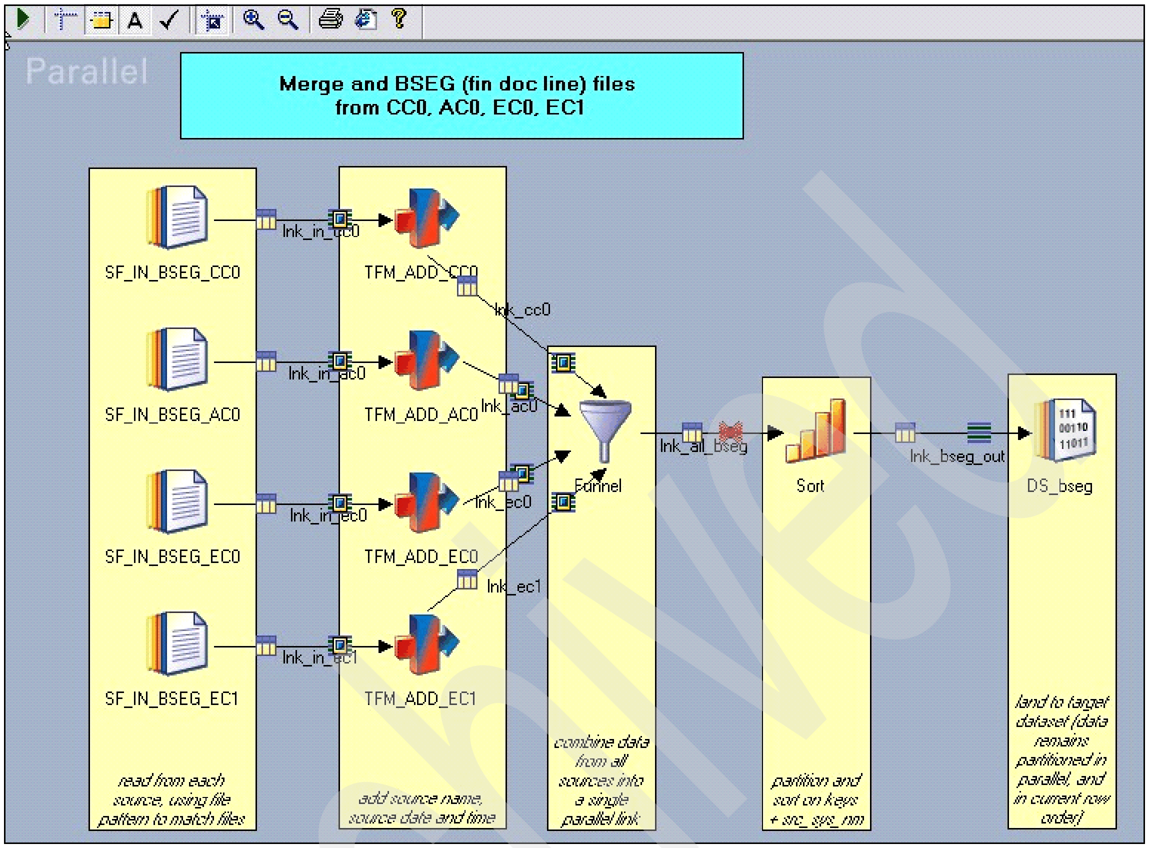

Parallel
Merge and BSEG (fin doc line) files from CC0, AC0, EC0, EC1
SF_IN_BSEG_CC0
SF_IN_BSEG_AC0
SF_IN_BSEG_EC0
SF_IN_BSEG_EC1
lnk_in_cc0
lnk_in_ac0
lnk_in_ec0
lnk_in_ec1
TFM_ADD_CC0
TFM_ADD_AC0
TFM_ADD_EC0
TFM_ADD_EC1
lnk_cc0
lnk_ac0
lnk_ec0
lnk_ec1
Funnel
lnk_all_bseg
Sort
lnk_bseg_out
DS_bseg
read from each source, using file pattern to match files
add source name, source date and time
combine data from all sources into a single parallel link
partition and sort on keys + src_sys_nm
land to target dataset (data remains partitioned in parallel, and in current row order)

Figure 3-11 Sample job annotation

DataStage also allows descriptions to be attached to each stage in the General tab of the stage properties.

Each stage should have a short description of its function specified in the stage properties. These descriptions appear in the job documentation automatically generated from jobs and sequencers adhering to the standards in this document. More complex operators or operations should have correspondingly longer and more complex explanations on this tab.

The following list details some examples of such annotations:

- Job short description

  This job takes the data from GBL Oracle Table AD_TYP and does a truncate load into Teradata Table AD_TYP.

- ODBC Enterprise stage read

Read the GLO.RcR_GLOBAL_BUSN_CAT_TYP table from ORACLE_SERVER_parm using the ODBC driver. There are no selection criteria in the WHERE clause.

- Oracle Enterprise stage read

  Read the GLOBAL.GLOBAL_REST_CHAR table from ORACLE_SERVER_parm using the Oracle Enterprise operator. There are no selection criteria in the WHERE clause.

- Remove Duplicates stage

  Removes all but one record with duplicate BUSN_OWN_TYP_ID keys.

- Lookup stage
  - Validates the input and writes rejects
  - Validates the input and continues
  - Identifies changes and drops records not matched (not updated)
- Copy stage
  - Sends data to the Teradata MultiLoad stage for loading into Teradata, and to a dataset for use as write-through cache
  - Renames and drops columns and is not optimized out
  - Is cosmetic and is optimized out
- Sequential file stage
  - Source file for the LANG table
  - Target file for business qualification process rejects
- Transformer stage
  - Generates sequence numbers that have a less-than file scope
  - Converts null dates
- Modify stage

  Performs data conversions not requiring a Transformer

- Teradata MultiLoad stage

  Loads the RcR_GLOBAL_LCAT_TYP table

- Dataset stage
  - Writes the GLOBAL_Ad_Typ dataset, which is used as write-through cache to avoid the use of database calls in subsequent jobs.
  - Reads the GLOBAL_Lcat dataset, which is used as write-through cache to avoid the use of database calls.

3.4 Working with source code control systems

The DataStage built-in repository manages objects (jobs, sequences, table definitions, routines, custom components) during job development. However, this repository is not capable of managing non-DataStage components (as examples, UNIX® shell scripts, environment files, and job scheduler configurations) that might be part of a completed application.

Source code control systems (such as ClearCase®, PVCS and SCCS) are useful for managing the development life cycle of all components of an application, organized into specific releases for version control.

As of release 8.1, DataStage does not directly integrate with source code control systems, but it does offer the ability to exchange information with these systems. It is the responsibility of the DataStage developer to maintain DataStage objects in the source code system.

The DataStage Designer client is the primary interface to the DataStage object repository. Using Designer, you can export objects (such as job designs, table definitions, custom stage types, and user-defined routines) from the repository as clear-text format files. These files can be checked into the external source code control system.

There are three export file format for DataStage 8.X objects:

- DSX (DataStage eXport format),
- XML
- ISX

DSX and XML are established formats that have remained the same since pre-8.X versions. ISX is a new format introduced in Information Server 8, which can be imported and exported with the new command-line utility ISTool.

3.4.1 Source code control standards

The first step to effective integration with source code control systems is to establish standards and rules for managing this process:

- Establish category naming and organization standard

  DataStage objects can be exported individually or by category (folder hierarchy). Grouping related objects by folder can simplify the process of exchanging information with the external source code control system. This object grouping helps establish a manageable middle ground between an entire project exports and individual object exports.

- Define rules for exchange with source code control

As a graphical development environment, Designer facilitates iterative job design. It is cumbersome to require the developer to check-in every change to a DataStage object in the external source code control system. Rather, rules must be defined for when this transfer should take place. Typically, milestone points on the development life cycle are a good point for transferring objects to the source code control system. For example, when a set of objects has completed initial development, unit test, and so on.

- Do not rely on the source code control system for backups

  Because the rules defined for transfer to the source code control system are typically only at milestones in the development cycle, they are not an effective backup strategy. Furthermore, operating system backups of the project repository files only establish a point in time, and cannot be used to restore individual objects.

For these reasons, it is important that an identified individual maintains backup copies of the important job designs using .DSX file exports to a local or (preferably) shared file system. These backups can be done on a scheduled basis by an operations support group, or by the individual DataStage developer. In either case, the developer should create a local backup prior to implementing any extensive changes.

3.4.2 Using object categorization standards

As discussed in “Folder hierarchy” on page 37, establishing and following a consistent naming and categorization standard is essential to the change management process. The DataStage Designer can export at the project, folder, and individual object levels. Assigning related objects to the same category provides a balanced level of granularity when exporting and importing objects with external source code control systems.

3.4.3 Export to source code control system

The process of exporting DataStage objects to a source code control system is a straightforward process. It can be done interactively by the developer or project manager using the Designer client, as explained in this section.

Information Server 8 introduces ISTool, which is a deployment and import/export tool that can be invoked either on the client or the server side. It extracts and imports repository objects in a new ISX format.

The major purpose of ISTool is to assist the deployment of DataStage applications, but this section focuses on its use as an import/export tool.

Client-only tools

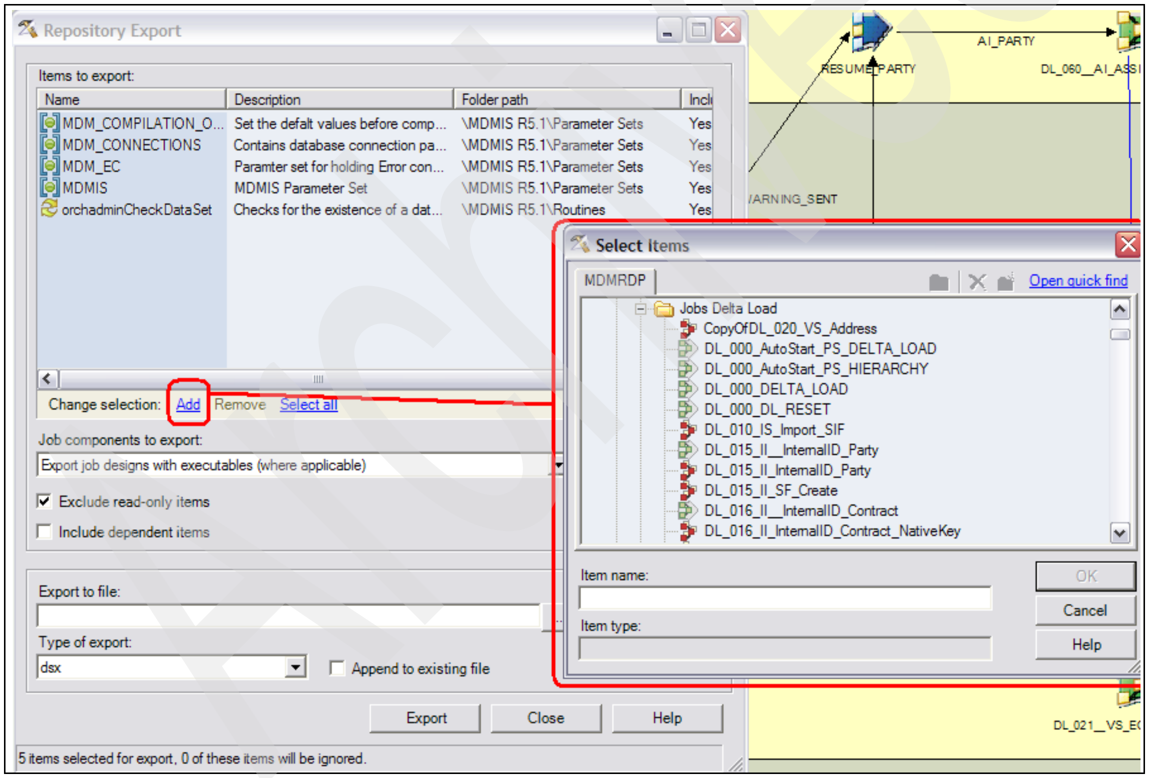
The DataStage client includes Windows® command-line utilities for automating the export process. These utilities (dsexport, dscmdexport, dsimport and XML2DSX) are documented in the *DataStage Designer Client Guide*, LC18-9893.

All exports from the DataStage repository to DSX or XML format are performed on the Windows workstation.

Exporting with DS Client Tools

The Export menu from DS Designer facilitates the selection of multiple folders and object types as part of the same export step.

In the example of Figure 3-12, the add link opens up a dialog box from which individual items or folders can be selected.



Repository Export

Items to export:

Name	Description	Folder path	Incl
MDM_COMPILATION_O...	Set the defalt values before comp...	\MDMIS R5.1\Parameter Sets	Yes
MDM_CONNECTIONS	Contains database connection pa...	\MDMIS R5.1\Parameter Sets	Yes
MDM_EC	Paramter set for holding Error con...	\MDMIS R5.1\Parameter Sets	Yes
MDMIS	MDMIS Parameter Set	\MDMIS R5.1\Parameter Sets	Yes
orchadminCheckDataSet	Checks for the existence of a dat...	\MDMIS R5.1\Routines	Yes

Change selection: Add Remove Select all

Job components to export: Export job designs with executables (where applicable)

☑ Exclude read-only items

☐ Include dependent items

Export to file:

Type of export: dsx ☐ Append to existing file

Export Close Help

5 items selected for export, 0 of these items will be ignored.

Select Items

MDMRDP Open quick find

- Jobs Delta Load
  - CopyOfDL_020_VS_Address
  - DL_000_AutoStart_PS_DELTA_LOAD
  - DL_000_AutoStart_PS_HIERARCHY
  - DL_000_DELTA_LOAD
  - DL_000_DL_RESET
  - DL_010_IS_Import_SIF
  - DL_015_II__InternalID_Party
  - DL_015_II_InternalID_Party
  - DL_015_II_SF_Create
  - DL_016_II__InternalID_Contract
  - DL_016_II_InternalID_Contract_NativeKey

Item name:

Item type:

OK Cancel Help

Figure 3-12 DS Designer Export facility

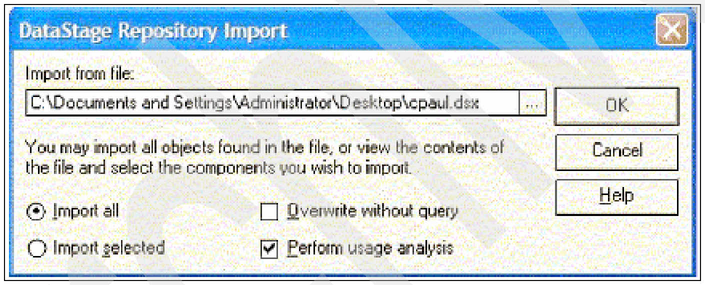
The user has already selected the MDMIS R5.1/Parameter Sets category and one DS routine. Additional items can be selected out of the Select Items window, and the user can also select the output format of DSX or XML.

Importing with DS client tools

In a similar manner, the import of objects from an external source code control system is a straightforward process. Import can be interactive through the DS Designer client or automated through command-line utilities.

In Information Server 8, the DS Designer import interface is basically the same as the one from DataStage 7.5.

- Use the source code control system to check-out (or export) the .DSX file to your client workstation.
- Import objects in the .DSX file using DS Designer. Choose **Import DataStage Components** from the Import menu. Select the file you checked out of your source code control system by clicking the ellipsis ("…") next to the filename field in the import dialog box. This is depicted in Figure 3-13. After selecting your file, click **OK** to import.



C:\Documents and Settings\Administrator\Desktop\cpaul.dsx

You may import all objects found in the file, or view the contents of the file and select the components you wish to import.

[✓ Import all]
Import selected
Overwrite without query
[✓ Perform usage analysis]

Figure 3-13 DS Designer import options for DSX files

- The import of the .DSX file places the object in the same DataStage folder from which it originated. This means that if necessary it creates the job folder if it does not already exist.
- If the objects were not exported with the job executables, then compile the imported objects from Designer, or from the multi-job compile tool.

There is an equivalent GUI option to import XML files. The import of XML files first converts the input file from XML to DSX by means of a XSL stylesheet (this is done behind the scenes). The DSX file is then finally imported into the repository.

The Windows workstation utilities (dsimport, dscmdimport and XML2DSX) are documented in the *DataStage Designer Client Guide*. Job designs in DSX or

XML format can be imported using the Designer client or the dsimport, dscmdimport or XML2DSX client tools.

Client and server tools

Information Server 8 provides for a new import/export tool named ISTool. It can be invoked either on the client or the server side. The ability to import and export objects on the server side is a feature long awaited by the user community.

One can now export an entire project with the following syntax (the same syntax applies to both the client and server environments):

```
istool export -domain <domain> -username <user> -password <passwd>
-archive <archive_name> -datastage '<hostname>/project/*.*‘
```

However, to export just the jobs category the wild card syntax changes a little (notice the extra /*/). Without this, the folders in are skipped.

```
istool export -do <domain> -u <user> -p <passwd> -ar <archive_name> -ds
'<hostname>/project/Jobs/*/*.*'
```

The output or the archive file is a compressed file. If uncompressed, it creates the directory structure similar to the GUI. The import option can be used to import the .isx (archive suffix), similar to the .xml or .dsx files.

It is much faster to run the export/import on the server side with the ISTool when compared to the DS Designer client tools described in 3.4.3, “Export to source code control system” on page 51.

The ISTool is documented in the *IBM Information Server Manager User Guide*, LC19-2453-00.

4 Job parameter and environment variable management

An overview of DataStage (DS) job parameters can be found in the *IBM InfoSphere DataStage and QualityStage Designer Client Guide*, LC18-9893.

DataStage jobs can be parameterized to allow for portability and flexibility. Parameters are used to pass values for variables into jobs at run time. There are two types of parameters supported by DataStage jobs:

- Standard job parameters
  - Are defined on a per job basis in the job properties dialog box.
  - The scope of a parameter is restricted to the job.
  - Used to vary values for stage properties, and arguments to before/after job routines at runtime.
  - No external dependencies, as all parameter metadata is a sub element to a single job.

- Environment variable parameters:
  - Use operating system environment variable concept.
  - Provide a mechanism for passing the value of an environment variable into a job as a job parameter (Environment variables defined as job parameters start with a $ sign).
  - Are similar to a standard job parameter in that it can be used to vary values for stage properties, and arguments to before/after job routines.
  - Provide a mechanism to set the value of an environment variable at runtime. DataStage provides a number of environment variables to enable / disable product features, fine-tune performance, and to specify runtime and design time functionality (for example, $APT_CONFIG_FILE).

4.1 DataStage environment variables

In this section we discuss the DataStage environment variables, including scope, special values, an overview of predefined variables and recommended settings for all jobs, and the migration of project-level settings across projects.

4.1.1 DataStage environment variable scope

Although operating system environment variables can be set in multiple places, there is a defined order of precedence that is evaluated when a job's actual environment is established at runtime. The scope of an environment variable is dependent upon where it is defined. Table 4-1 shows where environment variables are set and order of precedence.

Table 4-1 Environment variables

Where Defined	**Scope (* indicates highest precedence)**
System profile	System wide
dsenv	All DataStage processes
Shell script (if dsjob –local is specified)	Only DataStage processes spawned by dsjob
Project	All DataStage processes for a project
Job Sequencer	Job sequence and sequence sub processes
Job	* Current job's environment and sub processes

The daemon for managing client connections to the DataStage engine is called *dsrpcd*. By default (in a root installation), dsrpcd is started when the server is installed, and should start whenever the machine is restarted. dsrpcd can also be manually started and stopped using the **`$DSHOME/uv –admin`** command. For more information, see the *IBM InfoSphere DataStage and QualityStage Administrator Client Guide*, LC18-9895.

By default, DataStage jobs inherit the dsrpcd environment which, on UNIX platforms, is set in the `etc/profile` and `$DSHOME/dsenv` scripts. On Windows, the default DataStage environment is defined in the registry. Client connections do not pick up per-user environment settings from their `$HOME/.profile` script.

Environment variable settings for particular projects can be set in the DataStage Administrator client. Any project-level settings for a specific environment variable override any settings inherited from dsrpcd.

In DataStage Designer, environment variables might be defined for a particular job using the Job Properties dialog box. Any job-level settings for a specific environment variable override any settings inherited from dsrpcd or from project-level defaults. Project-level environment variables are set and defined in DataStage Administrator.

4.1.2 Special values for DataStage environment variables

To avoid hard-coding default values for job parameters, there are three special values that can be used for environment variables in job parameters. They are described in Table 4-2.

Table 4-2 Environment variable special values

Value	Use
$ENV	Causes the value of the named environment variable to be retrieved from the operating system of the job environment. Typically this is used to pickup values set in the operating system outside of DataStage.
$PROJDEF	Causes the project default value for the environment variable (as shown on the Administrator client) to be picked up and used to set the environment variable and job parameter for the job.
$UNSET	Causes the environment variable to be removed completely from the runtime environment. Several environment variables are evaluated only for their presence in the environment (for example, APT_SORT_INSERTION_CHECK_ONLY).

Note: $ENV must not be used for specifying the default $APT_CONFIG_FILE value because, during job development, Designer parses the corresponding parallel configuration file to obtain a list of node maps and constraints (Advanced stage properties)

4.1.3 Environment variable settings

In this section we call discuss a number of key environment variables, and provide descriptions of their settings for parallel jobs.

Environment variable settings for all jobs

IBM suggests the following environment variable settings for all DataStage parallel jobs. These settings can be made at the project level, or set on an individual basis in the properties for each job. It might be helpful to create a parameter set that includes the environment variables described in Table 4-3.

Table 4-3 Environment variable descriptions

Environment Variable	Setting	Description
$APT_CONFIG_FILE	filepath	Specifies the full path name to the parallel configuration file. This variable must be included in all job parameters so that it can be easily changed at runtime.
$APT_DUMP_SCORE		Outputs parallel score dump to the DataStage job log, providing detailed information about actual job flow including operators, processes, and Datasets. Extremely useful for understanding how a job actually ran in the environment. See Appendix E, "Understanding the parallel job score" on page 401.
$OSH_ECHO		Includes a copy of the generated osh (Orchestrate Shell) in the job's DataStage log.
$APT_RECORD_ COUNTS		Outputs record counts to the DataStage job log as each operator completes processing. The count is per operator per partition. This setting must be disabled by default, but part of every job design so that it can be easily enabled for debugging purposes.
$APT_PERFORMANCE _DATA	$UNSET	If set, specifies the directory to capture advanced job runtime performance statistics.
$OSH_PRINT_ SCHEMAS		Outputs actual runtime metadata (schema) to DataStage job log. This setting must be disabled by default, but part of every job design so that it can be easily enabled for debugging purposes.
$APT_PM_SHOW_ PIDS		Places entries in DataStage job log showing UNIX process ID (PID) for each process started by a job. Does not report PIDs of DataStage "phantom" processes started by Server shared containers.
$APT_BUFFER_ MAXIMUM_TIMEOUT		Maximum buffer delay in seconds

On Solaris platforms only: When working with large parallel datasets (where the individual data segment files are larger than 2 GB), you must define the environment variable `$APT_IO_NOMAP`.

On Tru64 5.1A platforms only: On Tru64 platforms, the environment variable $APT_PM_NO_SHARED_MEMORY must be set to 1 to work around a performance issue with shared memory MMAP operations. This setting instructs the parallel framework to use named pipes rather than shared memory for local data transport.

Additional environment variable settings

Throughout this document, a number of environment variables are mentioned for tuning the performance of a particular job flow, assisting in debugging, or changing the default behavior of specific DS Parallel stages. The environment variables mentioned in this document are summarized in Appendix G, “Environment variables reference” on page 413. An extensive list of environment variables is documented in the IBM *InfoSphere DataStage and QualityStage Parallel Job Advanced Developer Guide*, LC18-9892.

4.1.4 Migrating project-level environment variables

When migrating projects between machines or environments, it is important to note that project-level environment variable settings are not exported when a project is exported. These settings are stored in a file named DSPARAMS in the project directory. If an environment variable has not been configured for the project, the migrated job fails during startup.

Any project-level environment variables must be set for new projects using the Administrator client, or by carefully editing the DSPARAMS file in the project.

Note: Copying DSPARAMS between versions of DataStage might result in failure.

In addition, migrating lists of environment variable parameters and their values from one project to another can be managed by the administrator using a CEDI developed tool called *Environment Variable Transporter*.

4.2 DataStage job parameters

Parameters are passed to a job as either DataStage job parameters or as environment variables. The naming standard for job parameters uses the suffix `_parm` in the variable name. Environment variables have a prefix of `$` when used as job parameters.

Job parameters are passed from a job sequencer to the jobs in its control as though a user were answering the runtime dialog questions displayed in the DataStage Director job-run dialog. Job parameters can also be specified using a parameter set.

The scope of job parameters depends on their type. The scope depends on the following factors:

- Is specific to the job in which it is defined and used. Job parameters are stored internally in DataStage for the duration of the job, and are not accessible outside that job.
- Can be extended by the use of a job sequencer, which can manage and pass the job parameter among jobs in the sequence.

4.2.1 When to use parameters

As a standard practice, file names, database names, passwords, and message queue names must be parameterized. It is left to the discretion of the developer to parameterize other properties. When deciding on what to parameterize, ask the following questions:

- Could this stage or link property be required to change from one project to another?
- Could this stage or link property change from one job execution to another?

If you answer yes to either of these questions, you should create a job parameter and set the property to that parameter.

To facilitate production automation and file management, the Project_Plus and Staging file paths are defined using a number of environment variables that must be used as job parameters. (See 3.1.3, "Extending the DataStage project for external entities" on page 24 and 3.1.4, "File staging" on page 29.)

Job parameters are required for the following DataStage programming elements:

- File name entries in stages that use files or datasets must never use a hard-coded operating system path name.
  - Staging area files must always have path names as follows:
    `/#$STAGING_DIR##$DEPLOY_PHASE_parm#[filename.suffix]`
  - DataStage datasets always have path names as follows:
    `/#$PROJECT_PLUS_DATASETS#[headerfilename.ds]`
- Database stages must always use variables for the server name, schema (if appropriate), user ID and password.

A list of recommended job parameters is summarized in Appendix D, "Example job template" on page 397.

4.2.2 Parameter standard practices

File Name stage properties must be configured using two parameters, one for directory path and the second for file name. The directory path delimiter must be specified in the property to avoid errors. Do not assume the runtime value of the directory parameter includes the appropriate delimiter. If the user supplies it the operating system accepts // as a delimiter, and if it is not provided, which is common, the file name property specification is correct.

Example of standard practice for file name properties:

```
#Dir_Path#/#File_Name#
```

Similar to directory path delimiters, database schema names, etc. should contain any required delimiter.

Example of standard practice for table name properties:

```
#DatabaseSchemaName#.TableName
```

User Accounts and passwords must be specified as environment variables.

Passwords must be set to type encrypted, and the default value maintained using the DataStage Administrator.

4.2.3 Specifying default parameter values

Default values of job parameters migrate with the job. If the value is not overridden then you run the risk of unintentionally using the wrong resources, such as connecting to the wrong database or referencing the wrong file. To mitigate this risk developers should follow this standard practice when setting the default value of a parallel job's parameter:

- Standard parameters: Ensure the default value is empty
- Environment variable parameters: Ensure the default value is empty, $ENV, $PROJDEF or $UNSET

The intent of this standard practice is to ensure a job is portable. It thus requires the value of a parameter to be set independent of the job. During development of a job, consider using the standard practice of always using a test harness sequencer to execute a parallel job. The test harness allows the job to be run independently and ensures the parameter values are set. When the job is ready for integration into a production sequencer, the test harness can be cut, pasted, and linked into the production sequencer. The test harness is also useful in test environments, as it allows you to run isolated tests on a job.

In rare cases, normal parameter values must be allowed default values. For example, a job might be configured with parameters for `array_size` or `commit_size` and the corresponding Database stage properties set to these parameters. The default value must be set to an optimal value as determined by performance testing. This value must be relatively static. The value can always be overridden by job control. This exception also minimizes the number of parameters. Consider that `array_size` is different for every job. You can have a unique parameter for every job, but it is difficult to manage the values for every job.

4.2.4 Parameter sets

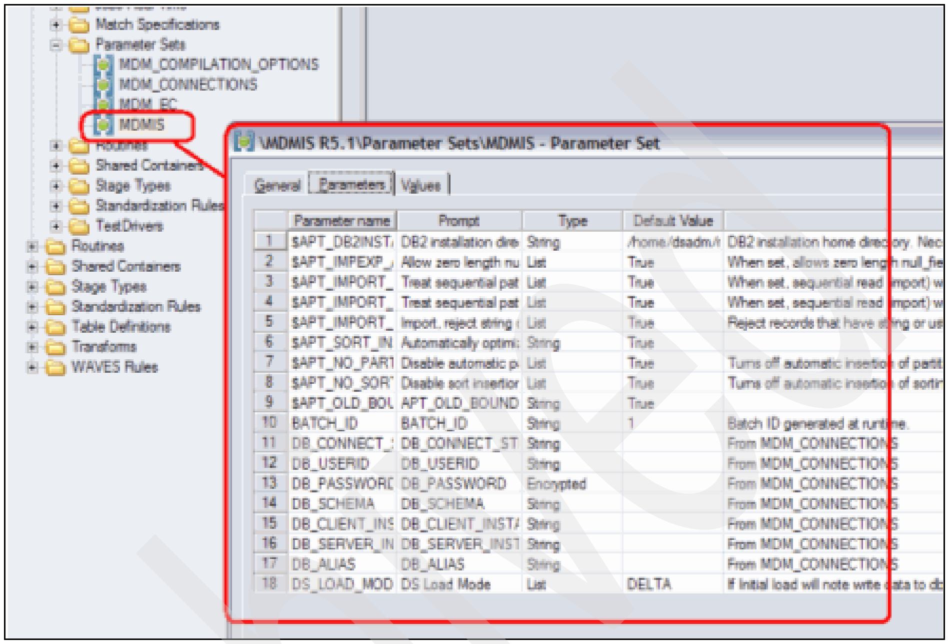
One of the great new features in Information Server 8 is the concept of parameter sets. They greatly simplify the management of job parameters by providing the following elements:

- Central location for the management of large lists of parameters
- Simplification of the propagation of parameter sets into jobs, sequences and shared containers
- Convenient way of defining multiple sets of values in the form of value files

Parameter sets are such an advance compared to earlier versions of DataStage that they actually render obsolete the techniques previously devised and recommended (such as Basic routines to read values from files and the use of job templates or shared containers as a way of incorporating predefined sets of parameters into new jobs).

A parameter set is assigned a name, and as such can be passed into jobs, shared containers and sequences collectively. It is an entity on its own, stored anywhere in the repository tree. We recommend creating a folder named “parameter sets” for this purpose.

Figure 4-1 shows an example of a parameter set named MDMIS. The Parameters tab lists the individual parameter names and their default values, types, prompts and descriptions.



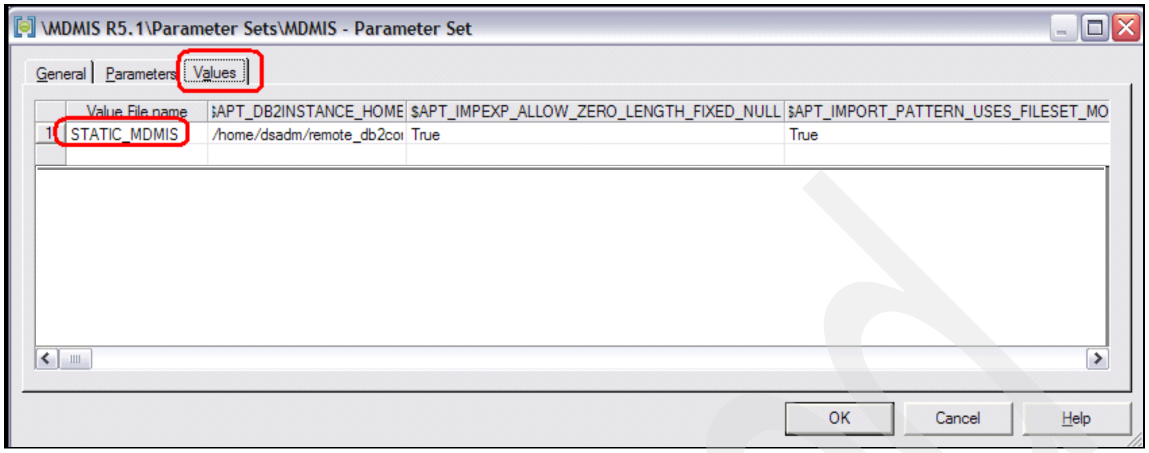
	Parameter name	Prompt	Type	Default Value	
1	$APT_DB2INST	DB2 installation dire	String	/home/dsadm/	DB2 installation home directory. Nec
2	$APT_IMPEXP_	Allow zero length nu	List	True	When set, allows zero length null_fie
3	$APT_IMPORT_	Treat sequential pat	List	True	When set, sequential read import) w
4	$APT_IMPORT_	Treat sequential pat	List	True	When set, sequential read import) w
5	$APT_IMPORT_	Import. reject string	List	True	Reject records that have string or us
6	$APT_SORT_IN	Automatically optimi	String	True	
7	$APT_NO_PART	Disable automatic p	List	True	Turns off automatic insertion of parti
8	$APT_NO_SORT	Disable sort insertior	List	True	Turns off automatic insertion of sortir
9	$APT_OLD_BOU	APT_OLD_BOUND	String	True	
10	BATCH_ID	BATCH_ID	String	1	Batch ID generated at runtime.
11	DB_CONNECT_	DB_CONNECT_ST	String		From MDM_CONNECTIONS
12	DB_USERID	DB_USERID	String		From MDM_CONNECTIONS
13	DB_PASSWORD	DB_PASSWORD	Encrypted		From MDM_CONNECTIONS
14	DB_SCHEMA	DB_SCHEMA	String		From MDM_CONNECTIONS
15	DB_CLIENT_INS	DB_CLIENT_INSTA	String		From MDM_CONNECTIONS
16	DB_SERVER_IN	DB_SERVER_INST	String		From MDM_CONNECTIONS
17	DB_ALIAS	DB_ALIAS	String		From MDM_CONNECTIONS
18	DS_LOAD_MOD	DS Load Mode	List	DELTA	If Initial load will note write data to db

Figure 4-1 A sample parameter set

The multiple values that parameter set MDMIS can assume are defined in the Values tab. The list of parameters is presented horizontally. The first column is the name of a value file. Subsequent columns contain values for each individual parameter.

In this example, there is a single value file, but there might be multiple such value files for the same parameter set. This is depicted in Figure 4-2 on page 65.

Parameter sets are stored in the metadata layer along with the rest of the project's design metadata. They might be exported and imported individually or with other objects using the DS Designer's Import/Export facilities.

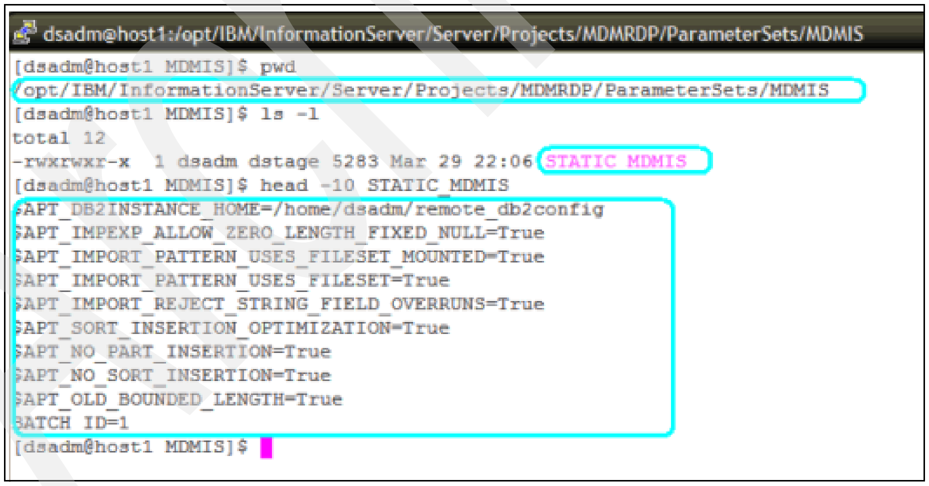


\MDMIS R5.1\Parameter Sets\MDMIS - Parameter Set

	Value File name	$APT_DB2INSTANCE_HOME	$APT_IMPEXP_ALLOW_ZERO_LENGTH_FIXED_NULL	$APT_IMPORT_PATTERN_USES_FILESET_MO
1	STATIC_MDMIS	/home/dsadm/remote_db2co	True	True

Figure 4-2 Value files of a parameter set

Value files are actual flat files stored in the DS Project in the DSEngine host file system. Figure 4-2 presents an example of a shell session displaying the location and content of the STATIC_MDMIS value file for the MDMIS parameter set. There is a directory named ParameterSets/MDMIS under the MDMRDP project directory. The value file STATIC_MDMIS is stored in that directory. The output showing the first 10 lines of output is depicted in Figure 4-3.



dsadm@host1:/opt/IBM/InformationServer/Server/Projects/MDMRDP/ParameterSets/MDMIS

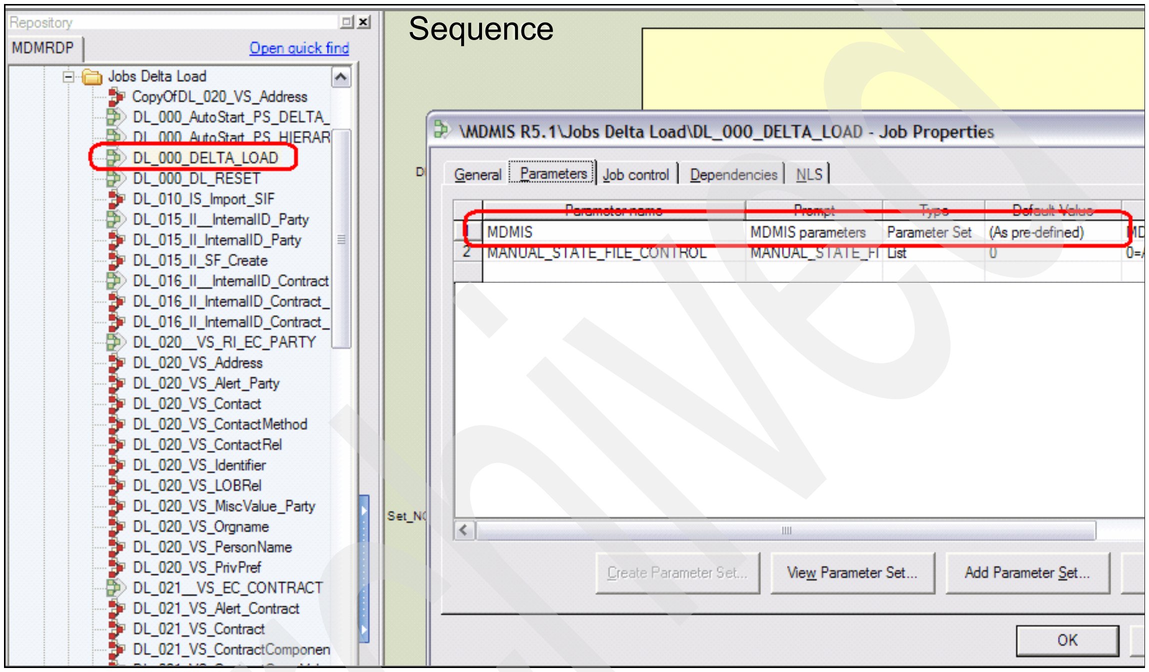
```
[dsadm@host1 MDMIS]$ pwd
/opt/IBM/InformationServer/Server/Projects/MDMRDP/ParameterSets/MDMIS
[dsadm@host1 MDMIS]$ ls -l
total 12
-rwxrwxr-x  1 dsadm dstage 5283 Mar 29 22:06 STATIC_MDMIS
[dsadm@host1 MDMIS]$ head -10 STATIC_MDMIS
$APT_DB2INSTANCE_HOME=/home/dsadm/remote_db2config
$APT_IMPEXP_ALLOW_ZERO_LENGTH_FIXED_NULL=True
$APT_IMPORT_PATTERN_USES_FILESET_MOUNTED=True
$APT_IMPORT_PATTERN_USES_FILESET=True
$APT_IMPORT_REJECT_STRING_FIELD_OVERRUNS=True
$APT_SORT_INSERTION_OPTIMIZATION=True
$APT_NO_PART_INSERTION=True
$APT_NO_SORT_INSERTION=True
$APT_OLD_BOUNDED_LENGTH=True
BATCH_ID=1
[dsadm@host1 MDMIS]$
```

Figure 4-3 Location and content of a value file in the DS project directory

Propagating parameter sets

Propagating large sets of parameters is as easy as defining a single parameter. Instead of repeatedly defining individual parameters in each sequence, job, and shared container, all it takes is defining a single parameter of type `Parameter Set` (click the **Add Parameter Set** button).

Figure 4-4 shows a sample job sequence, with a parameter set named `MDMIS`.



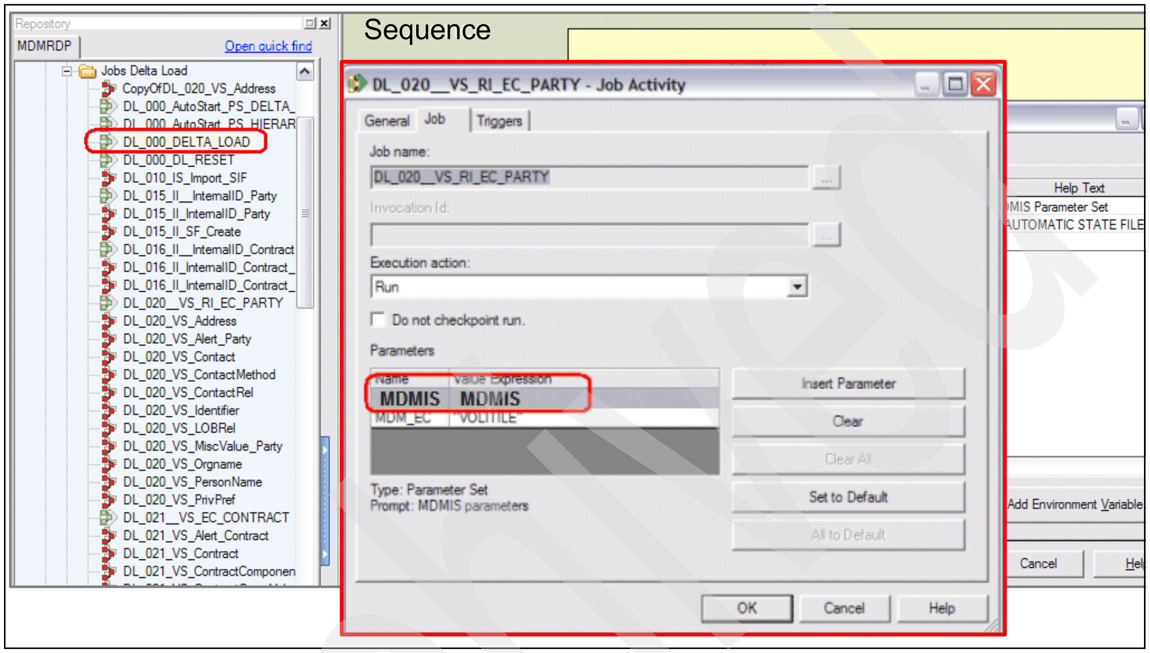
Jobs Delta Load
- CopyOfDL_020_VS_Address
- DL_000_AutoStart_PS_DELTA_
- DL_000_AutoStart_PS_HIERAR
- DL_000_DELTA_LOAD
- DL_000_DL_RESET
- DL_010_IS_Import_SIF
- DL_015_II__InternalID_Party
- DL_015_II_InternalID_Party
- DL_015_II_SF_Create
- DL_016_II__InternalID_Contract
- DL_016_II_InternalID_Contract_
- DL_016_II_InternalID_Contract_
- DL_020__VS_RI_EC_PARTY
- DL_020_VS_Address
- DL_020_VS_Alert_Party
- DL_020_VS_Contact
- DL_020_VS_ContactMethod
- DL_020_VS_ContactRel
- DL_020_VS_Identifier
- DL_020_VS_LOBRel
- DL_020_VS_MiscValue_Party
- DL_020_VS_Orgname
- DL_020_VS_PersonName
- DL_020_VS_PrivPref
- DL_021__VS_EC_CONTRACT
- DL_021_VS_Alert_Contract
- DL_021_VS_Contract
- DL_021_VS_ContractComponen

	Parameter name	Prompt	Type	Default Value	
1	MDMIS	MDMIS parameters	Parameter Set	(As pre-defined)	MD
2	MANUAL_STATE_FILE_CONTROL	MANUAL_STATE_FI	List	0	0=/

Figure 4-4 Job sequence with a MDMIS parameter set

Figure 4-5 displays how a sequence passes the parameter set down to one of its job activities. The elimination of clutter is evident.

The parameters for the job invoked by this job activity are defined in a way similar to the one depicted in Figure 4-4 on page 66.


Repository
MDMRDP
Open quick find

Jobs Delta Load
- CopyOfDL_020_VS_Address
- DL_000_AutoStart_PS_DELTA_
- DL_000_AutoStart_PS_HIERAR
- DL_000_DELTA_LOAD
- DL_000_DL_RESET
- DL_010_IS_Import_SIF
- DL_015_II__InternalID_Party
- DL_015_II_InternalID_Party
- DL_015_II_SF_Create
- DL_016_II__InternalID_Contract
- DL_016_II_InternalID_Contract_
- DL_016_II_InternalID_Contract_
- DL_020__VS_RI_EC_PARTY
- DL_020_VS_Address
- DL_020_VS_Alert_Party
- DL_020_VS_Contact
- DL_020_VS_ContactMethod
- DL_020_VS_ContactRel
- DL_020_VS_Identifier
- DL_020_VS_LOBRel
- DL_020_VS_MiscValue_Party
- DL_020_VS_Orgname
- DL_020_VS_PersonName
- DL_020_VS_PrivPref
- DL_021__VS_EC_CONTRACT
- DL_021_VS_Alert_Contract
- DL_021_VS_Contract
- DL_021_VS_ContractComponen

Sequence

DL_020__VS_RI_EC_PARTY - Job Activity

General | Job | Triggers

Job name: DL_020__VS_RI_EC_PARTY

Invocation Id:

Execution action: Run

Do not checkpoint run.

Parameters

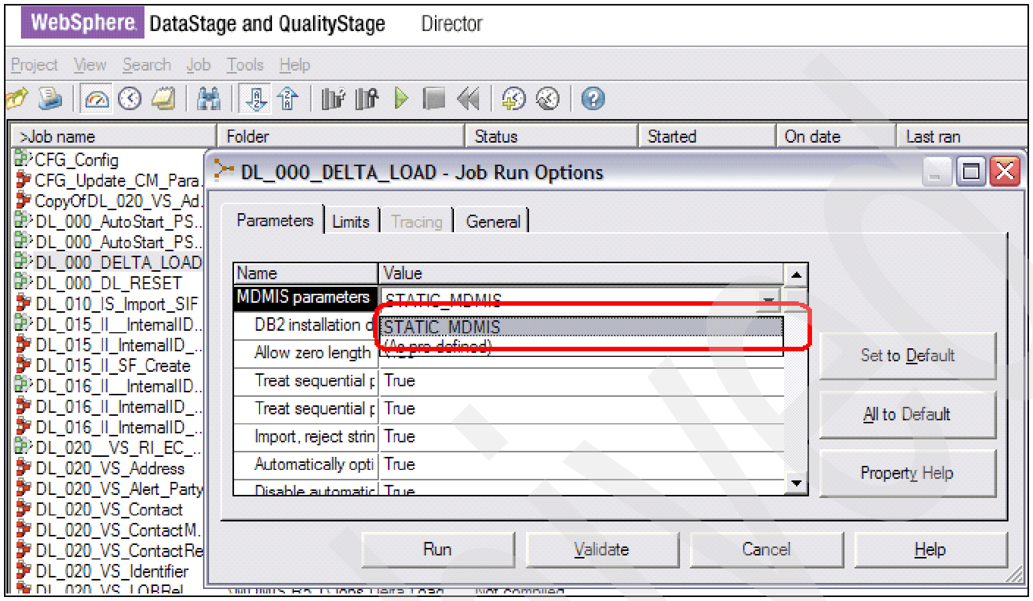
Name	Value Expression
MDMIS	MDMIS
MDM_EC	"VOLITILE"

Type: Parameter Set
Prompt: MDMIS parameters

Figure 4-5 Passing a parameter set to a job activity

Setting the runtime value for a parameter set

Upon job or sequence invocation, the value file for a parameter set might be specified on the director job run options, as depicted in Figure 4-6.



Name	Value
MDMIS parameters	STATIC_MDMIS
DB2 installation d	STATIC_MDMIS
Allow zero length	(As pre-defined)
Treat sequential	True
Treat sequential	True
Import, reject strin	True
Automatically opti	True
Disable automatic	True

Figure 4-6 Setting the value file upon job sequence invocation

Values for individual parameters might be overridden manually by the operator, in addition to the ones assumed by the choice of value file. For dsjob, the command line would follow a syntax similar to setting an individual parameter. For the example, the choice of value file is specified as:

```
param MDMIS=STATIC_MDMIS
```

Using parameter sets is strongly encouraged in all types of DataStage projects.

Environment variable parameter lists

When a job is migrated from one project to another, the job fails during startup if the environment variable has not been configured for the project. Migrating lists of environment variable parameters and their values from one project to another can be managed by the administrator using a CEDI developed tool called the *Environment Variable Transporter*.

If the tool is not available then you must enter the environment variable parameters one at a time in the DataStage Administrator. If this becomes too cumbersome, consider the fact that environment variable parameters are stored in the DSParams file. The DSParams is a text file that can be modified by hand. However, if you choose to modify this file by hand you do so at your own risk.

5

Development guidelines

In this chapter we discuss and describe guidelines for developing DataStage (DS) parallel jobs. The guidelines in this chapter relate to the following subjects:

- Modular development
- Establishing job boundaries
- The use of job design templates
- Parallel shared containers
- Error and reject record handling
- Considerations for usage and impact analysis

In addition, we also include a discussion on the usage of components, such as Server Edition components, Copy stage, Parallel Datasets, Parallel Transformer and BuildOps.

We then delve into a greater level of detail on how to use the various stage types in subsequent chapters of this book.

5.1 Modular development

Modular development techniques must be used to maximize re-use of DataStage jobs and components:

- Job parameterization allows a single job design to process similar logic instead of creating multiple copies of the same job. The multiple-instance job property allows multiple invocations of the same job to run simultaneously.
- A set of standard job parameters must be used in DataStage jobs for source and target database parameters (such as DSN and user, password) and directories where files are stored. To ease re-use, these standard parameters and settings must be made part of a Designer job template.
- Create a standard directory structure outside of the DataStage project directory for source and target files, intermediate work files, and so forth.
- Where possible, create reusable components such as parallel shared containers to encapsulate frequently used logic.

5.2 Establishing job boundaries

It is important to establish appropriate job boundaries when developing parallel jobs. In certain cases, functional requirements might dictate job boundaries. For example, it might be appropriate to update all dimension values before inserting new entries in a data warehousing fact table. But functional requirements might not be the only factor driving the size of a given DataStage job.

Though it might be possible to construct a large, complex job that satisfies given functional requirements, this might not be appropriate. The following list details factors to consider when establishing job boundaries:

- Establishing job boundaries through intermediate datasets creates checkpoints that can be used in the event of a failure when processing must be restarted. Without these checkpoints, processing must be restarted from the beginning of the job flow. It is for these reasons that long-running tasks are often segmented into separate jobs in an overall sequence.
  - For example, if the extract of source data takes a long time (such as an FTP transfer over a wide area network) land the extracted source data to a parallel data set before processing. To continue processing to a database can cause conflict with locking tables when waiting for the FTP to complete.
  - As another example, land data to a parallel dataset before loading to a target database unless the data volume is small, the overall time to process the data is minimal, or if the data volume is so large that it cannot be staged on the extract, transform, and load (ETL) server.

- Larger, more complex jobs require more system resources (CPU, memory, swap) than a series of smaller jobs, sequenced together through intermediate datasets. Resource requirements are further increased when running with a greater degree of parallelism specified by a given configuration file. However, the sequence of smaller jobs generally requires more disk space to hold intermediate data, and the speed of the I/O subsystem can impact overall end-to-end throughput.

  In 12.3, "Minimizing runtime processes and resource requirements" on page 179, we provide recommendations for minimizing resource requirements of a given job design, especially when the volume of data does not dictate parallel processing.
- Breaking large job flows into smaller jobs might further facilitate modular development and re-use if business requirements for more than one process depend on intermediate data created by an earlier job.
- The size of a job directly impacts the speed of development tasks such as opening, saving, and compiling. These factors might be amplified when developing across a wide-area or high-latency network connection. In extreme circumstances, this can significantly impact developer productivity and ongoing maintenance costs.
- The startup time of a given job is directly related to the number of stages and links in the job flow. Larger more complex jobs require more time to startup before actual data processing can begin. Job startup time is further impacted by the degree of parallelism specified by the parallel configuration file.
- Remember that the number of stages in a parallel job includes the number of stages in each shared container used in a particular job flow.

As a rule of thumb, keeping job designs to less than 50 stages is a good starting point. But this is not a hard-and-fast rule. The proper job boundaries are ultimately dictated by functional/restart/performance requirements, expected throughput and data volumes, degree of parallelism, number of simultaneous jobs and their corresponding complexity, and the capacity and capabilities of the target hardware environment.

Combining or splitting jobs is relatively easy, so do not be afraid to experiment and see what works best for your jobs in your environment.

5.3 Job design templates

DataStage Designer provides the developer with re-usable job templates, which can be created from an existing parallel job or job sequence using the **`New Template from Job`** command.

Template jobs must be created with the following elements:

- Standard parameters (for example, source and target file paths, database login properties, and so forth)
- Environment variables and their default settings
- Project_Plus environment variables (see 3.1.3, “Extending the DataStage project for external entities” on page 24.)
- Annotation blocks

In addition, template jobs might contain any number of stages and pre-built logic, allowing multiple templates to be created for various types of standardized processing.

5.4 Default job design

Default job designs include all of the capabilities detailed in Chapter 3, “Standards” on page 21. Template jobs should contain all the default characteristics and parameters the project requires. These defaults provide at a minimum:

- Development phase neutral storage (as examples: dev, it, uat and prod)
- Support for Teradata, Oracle, DB2/UDB and SQL Server login requirements
- Enforced project standards
- Optional operational metadata (runtime statistics) suitable for loading into a database
- Optional auditing capabilities

The default job design specifically supports the creation of write-through cache in which data in load-ready format is stored in parallel datasets for use in the load process or in the event the target table becomes unavailable.

The default job design incorporates several features and components of DataStage that are used together to support tactical and strategic job deployment.

These features include:

- Restartable job sequencers that manage one or more jobs, detect and report failure conditions, provide monitoring and alert capabilities, and support checkpoint restart functionality.
- Custom routines written in DataStage BASIC (DS Basic) that detect external events, manage and manipulate external resources, provide enhanced notification and alert capabilities, and interface to the UNIX operating system
- DataStage Parallel jobs that exploit job parameterization, runtime UNIX environment variables, and conditional execution.

Each subject area is broken into sub-areas and each sub-area might be further subdivided. These sub-areas are populated by a DataStage job sequencer using two types of DataStage jobs at a minimum:

- A job that reads source data and then perform one of the following tasks
  - Transforms it to load-ready format
  - Optionally stores its results in a write-through cache DataStage dataset or loads the data to the target table.
- A job that reads the DataStage dataset and loads it to the target table.

Other sections discuss in detail each of the components and give examples of their use in a working example job sequencer.

5.5 Parallel shared containers

Parallel shared containers allow common logic to be shared across multiple jobs. For maximum component re-use, enable Runtime Column Propagation (RCP) at the project level and for every stage in the parallel shared container. This allows the container input and output links to contain only the columns relevant to the container processing. When using RCP, any additional columns are passed through the container at runtime without the need to separate and remerge.

Because parallel shared containers are inserted when a job is compiled, all jobs that use a shared container must be recompiled when the container is changed. The Usage Analysis and Multi-Job Compile tools can be used to recompile jobs that use a shared container.

5.6 Error and reject record handling

Reject rows are those rows that fail active or passive business rule driven validation as specified in the job design document. Error rows are those rows caused by unforeseen data events such as values too large for a column or text in an unsupported language.

The exact policy for each reject is specified in the job design document, and further, whether the job or ETL processing is to continue is specified on a per-job and per-sequence and per-script basis based on business requirements.

Reject files include those records rejected from the ETL stream due to Referential Integrity failures, data rule violations or other reasons that would disqualify a row from processing. The presence of rejects might indicate that a job has failed and prevent further processing. Specification of this action is the responsibility of the Business Analyst and is published in the design document.

Error files include those records from sources that fail quality tests. The presence of errors might not prevent further processing. Specification of this action is the responsibility of the Business Analyst and is published in the design document.

Both rejects and errors are archived and placed in a special directory for evaluation or other action by support staff. The presence of rejects and errors are detected and notification sent by email to selected staff. These activities are the responsibility of job sequencers used to group jobs by reasonable grain or by a federated scheduler.

ETL actions to be taken for each record type is specified for each stage in the job design document. These actions include:

1. Ignore – some process or event downstream of the ETL process is responsible for handling the error.
2. Reprocess – rows are reprocessed and re-enter the data stream.
3. Push back – rows are sent to a Data Steward for corrective action.

The default action is to push back reject and error rows to a Data Steward.

5.6.1 Reject handling with the Sequential File stage

The Sequential File stage can optionally include a reject link, which outputs rows that do not match the given table definition and format specifications. By default, rows that cannot be read are dropped by the Sequential File stage. A message is always written to the Director log which details the count of rows successfully read and rows rejected.

The Sequential File stage offers the reject options listed in Table 5-1:

Table 5-1 Reject options

Option	Description
Continue	Drop read failures from input stream. Pass successful reads to the output stream. (No reject link exists)
Fail	Abort job on read format failure (No reject link exists)
Output	Reject switch failures to the reject stream. Pass successful reads to the output stream. (Reject link exists)

The reject option must be used in all cases where active management of the rejects is required.

If a file is created by this option, it must have a *.rej file extension. Alternatively, a shared container error handler can be used.

Rejects are categorized in the ETL job design document using the ranking listed in Table 5-2.

Table 5-2 Reject ranking

Category	Description	Sequential File stage Option
	Rejects are expected and can be ignored	Use the Continue option. Only records that match the given table definition and format are output. Rejects are tracked by count only.
	Rejects should not exist but should not stop the job, and must be reviewed by the Data Steward.	Use the Output option. Send the reject stream to a *.rej file.
	Rejects should not exist and should stop the job.	Use the Fail option.

5.6.2 Reject handling with the Lookup stage

The Lookup stage compares a single input stream to one or more reference streams using keys, and rejects can occur if the key fields are not found in the reference data. This behavior makes the Lookup stage valuable for positive (reference is found) and negative (reference is not found) business rule validation. DataStage offers options in a Parallel Lookup stage, as listed in Table 5-3.

Table 5-3 Parallel Lookup stage options

Option	Description
Continue	Ignore Lookup failures and pass Lookup fields as nulls to the output stream. Pass successful Lookups to the output stream.
Drop	Drop Lookup failures from the input stream. Pass successful Lookups to the output stream.
Fail	Abort job on Lookup failure
Reject	Reject Lookup failures to the reject stream. Pass successful Lookups to the output stream.

The reject option must be used in all cases where active management of the rejects is required. Furthermore, to enforce error management only one reference link is allowed on a Lookup stage. If there are multiple validations to perform, each must be done in its own Lookup.

If a file is created by this option, it must have a *.rej or *.err file extension. The *.rej extension is used when rejects require investigation after a job run, the *.err extension when rejects can be ignored but need to be recorded. Alternatively, a local error handler based on a shared container can be used.

Rejects are categorized in the ETL job design document using the ranking in Table 5-4.

Table 5-4 Reject ranking: Lookup stage

Category	Description	Lookup stage option
	Rejects are expected and can be ignored	Drop if lookup fields are necessary down stream or Continue if lookup fields are optional
	Rejects can exist in the data, however, they only need to be recorded but not acted on.	Send the reject stream to an *.err file or tag and merge with the output stream.
	Rejects should not exist but should not stop the job, and must be reviewed by the Data Steward.	Send the reject stream to an *.rej file or tag and merge with the output stream.
	Rejects should not exist and should stop the job.	Use the Fail option.

5.6.3 Reject handling with the Transformer stage

Rejects occur when a Transformer stage is used and a row:

1. satisfies requirements for a reject conditional output stream.
2. cannot satisfy requirements of any conditional output stream and is rejected by the default output stream.

If a file is created from the reject stream, it must have a *.rej or *.err file extension. The *.rej extension is used when rejects require investigation after a job run, the *.err extension when rejects can be ignored but need to be recorded. Alternatively, a shared container error handler can be used.

Rejects are categorized in the ETL job design document using the ranking listed in Table 5-5.

Table 5-5 Reject ranking: Transformer stage

Category	Description	Transformer stage option
	Rejects are expected and can be ignored.	Funnel the reject stream back to the output streams.
	Rejects can exist in the data, however, they only need to be recorded but not acted on.	Send the reject stream to an *.err file or tag and merge with the output stream.
	Rejects should not exist but should not stop the job, and be reviewed by the Data Steward.	Send the reject stream to an *.rej file or tag and merge with the output stream.
	Rejects should not exist and should stop the job.	Send the reject stream to a reject file and halt the job.

5.6.4 Reject handling with Target Database stages

Database stages (such as DB2/UDB Enterprise, ODBC Enterprise, and Oracle Enterprise) offer an optional reject link that can be used to capture rows that cannot be written to the target database. To capture rejects from a target database, a reject link must exist on that stage. Otherwise, reject rows are not captured. A message is always written to the Director log which details the count of rows successfully read and rows rejected.

Target Database stages offer the reject options listed in Table 5-6.

Table 5-6 Reject options

Option	Description
No reject link exists	Do not capture rows that fail to be written.
Reject link exists	Pass rows that fail to be written to the reject stream.

The reject option must be used in all cases where active management of the rejects is required.

If a file is created by this option, it must have a *.rej file extension. Alternatively, a shared container error handler is used.

Rejects are categorized in the ETL job design document using the ranking listed in Table 5-7.

Table 5-7 Reject ranking

Category	Description	Target Database stage Option
	Rejects are expected and can be ignored	No reject link exists. Only records that match the given table definition and database constraints are written. Rejects are tracked by count only.
	Rejects should not exist but should not stop the job, and must be reviewed by the Data Steward.	Reject link exists. Send the reject stream to a *.rej file.

5.6.5 Error processing requirements

Jobs produce flat files containing rejects and errors. They might alternatively process rows on reject ports and merge these rows with the normal output stream. This section deals with both methods of handling errors.

Processing errors and rejects to a flat file

Each job produces a flat file for errors and a flat file for rejects with a specific naming convention:

1. The project name ($PROJECT_NAME) and a underscore "_";
2. The job name (JOB_NAME_parm) and a underscore "_";
3. The project phase ($DEPLOY_PHASE) and a underscore "_";
4. The job run identifier (RUN_ID_parm) and a period "."; and
5. The appropriate file type, one of "rej" or "err".

For example, job DECRP_N_XformClients in the ECR_FACTS project in the development environment with a run identifier of 20060201-ETL-091504 would have the following reject and error file names:

- `ECR_FACTS_DECRP_N_XformClients_dev_20060201-ETL-091504.rej`
- `ECR_FACTS_DECRP_N_XformClients_dev_20060201-ETL-091504.err`

Rows are converted to the common file record format with 9 columns (as shown in Figure 5-2 on page 81) using Column Export and Transformer stages for each reject port, and gathered using a Funnel stage that feeds a Sequential File stage. The Column Export and Transformer stages might be kept in a template Shared Container the developer makes local in each job.

The standard columns for error and reject processing are listed in Table 5-8.

Table 5-8 Standard columns for error and reject processing

Column Name	Key?	Data Source
HOST_NAME	Yes	DSHostName Transformer macro in the error handler
PROJECT_NAME	Yes	DSProjectName Transformer macro in the error handler
JOB_NAME	Yes	DSJobName Transformer macro in the error handler
STAGE_NAME	Yes	The name of the stage from which the error came
DATA_OBJ_NAME	Yes	The source table or file data object name
RUN_ID	Yes	RUN_ID_parm
ETL_ROW_NUM	Yes	Data stream coming in to the error handler
ROW_DATA	No	The columns from the upstream stages reject port exported to a single pipe-delimited "|" varchar column using the Column Export stage in the error handler Length to be determined by max length of record to be maintained as row data.

In Figure 5-1 we depict the stages that process the errors produced by a job.

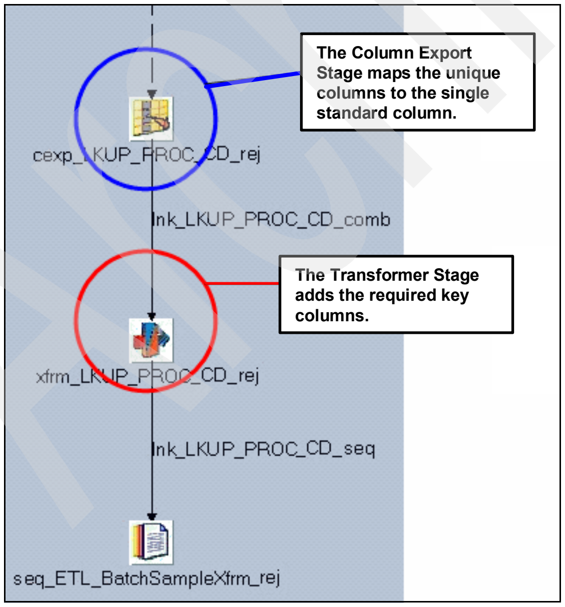
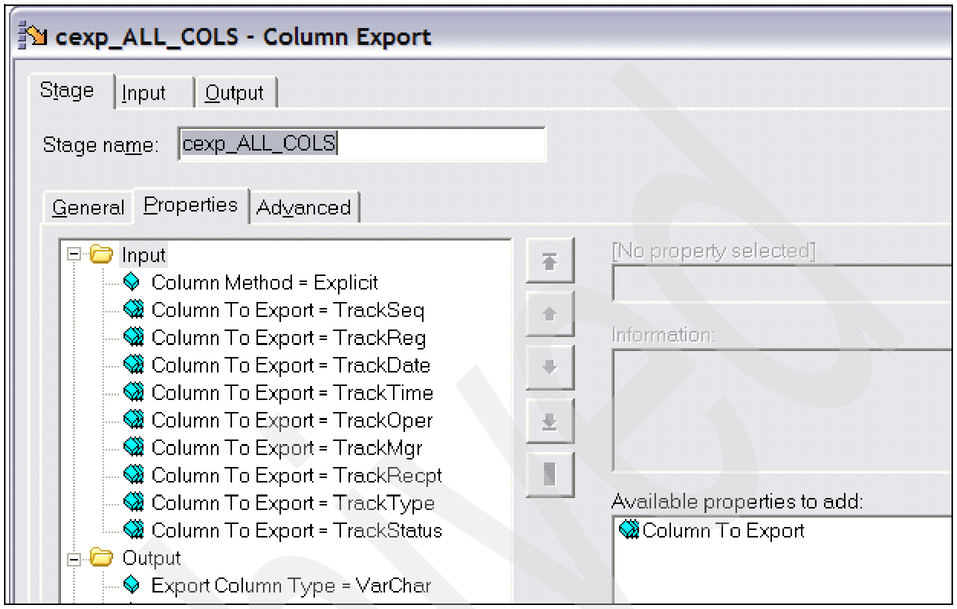

The Column Export Stage maps the unique columns to the single standard column.
cexp_LKUP_PROC_CD_rej
lnk_LKUP_PROC_CD_comb
The Transformer Stage adds the required key columns.
xfrm_LKUP_PROC_CD_rej
lnk_LKUP_PROC_CD_seq
seq_ETL_BatchSampleXfrm_rej

Figure 5-1 Error processing components

The input to the Column Export stage explicitly converts the data unique to the reject stream (in this case, Track*) to a single output column, ROW_DATA, as depicted in Figure 5-2.



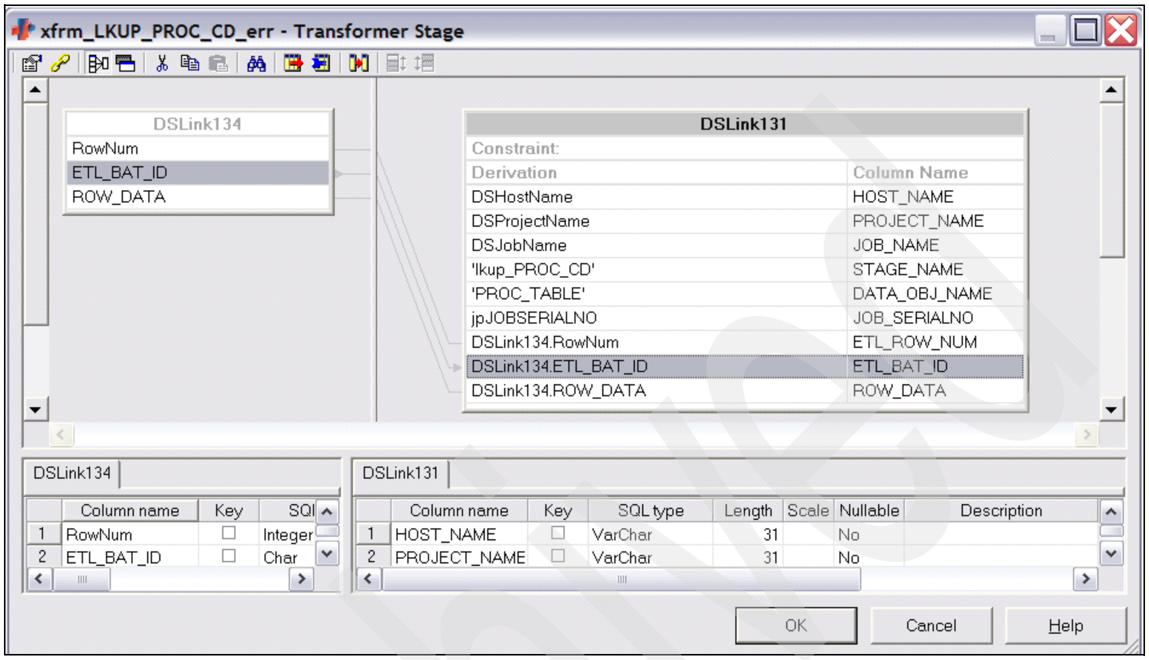
cexp_ALL_COLS - Column Export

Stage name: cexp_ALL_COLS

- Input
  - Column Method = Explicit
  - Column To Export = TrackSeq
  - Column To Export = TrackReg
  - Column To Export = TrackDate
  - Column To Export = TrackTime
  - Column To Export = TrackOper
  - Column To Export = TrackMgr
  - Column To Export = TrackRecpt
  - Column To Export = TrackType
  - Column To Export = TrackStatus
- Output
  - Export Column Type = VarChar

Figure 5-2 Error processing Column Export stage

And the downstream Transformer stage builds the standard output record by creating the required keys, as depicted in Figure 5-3.


xfrm_LKUP_PROC_CD_err - Transformer Stage

DSLink134
RowNum
ETL_BAT_ID
ROW_DATA

DSLink131

Constraint:

Derivation	Column Name
DSHostName	HOST_NAME
DSProjectName	PROJECT_NAME
DSJobName	JOB_NAME
'lkup_PROC_CD'	STAGE_NAME
'PROC_TABLE'	DATA_OBJ_NAME
jpJOBSERIALNO	JOB_SERIALNO
DSLink134.RowNum	ETL_ROW_NUM
DSLink134.ETL_BAT_ID	ETL_BAT_ID
DSLink134.ROW_DATA	ROW_DATA

DSLink134

	Column name	Key	SQl
1	RowNum	☐	Integer
2	ETL_BAT_ID	☐	Char

DSLink131

	Column name	Key	SQL type	Length	Scale	Nullable	Description
1	HOST_NAME	☐	VarChar	31		No	
2	PROJECT_NAME	☐	VarChar	31		No	

Figure 5-3 Error processing Transformer stage

Processing errors and rejects and merging with an output stream

There might be processing requirements specifying that rejected or error rows be tagged as having failed a validation and be merged back into the output stream. This is done by processing the rows from the reject ports and setting the value of a specific column with a value specified by the design document. In Table 5-9 we identify the tagging method to be used for the previously cited operators.

Table 5-9 Tagging method

stage	Description	Method
Lookup	A failed Lookup rejects an intact input row whose key fails to match the reference link key. One or more columns might have been selected for replacement when a reference key is found.	Connect the reject port to a Transformer stage where those columns selected for replacement are set to specific values. Connect the output stream of the Transformer and Lookup stages to a Funnel stage to merge the two streams.
Switch	A failed switch rejects an intact input row show key fails to resolve to one of the switch output stream.	Connect the reject port to a Transformer stage where columns are set to specific values. Connect the output stream of the Transformer Stage and one or more output streams of the Switch stage to a Funnel stage to merge the two (or more) streams.
Transformer	A Transformer rejects an intact input row that cannot pass conditions specified on the output streams, OR with columns contain illegal values for operations performed on said columns. In either case, attaching a non-specific reject stream (referred to as the stealth reject stream) gathers rows from either condition to the reject stream.	Connect the reject port to a Transformer stage where columns are set to specific values. Connect the output stream of the corrective Transformer stage and one or more output streams of the original Transformer stage to a Funnel stage to merge the two (or more) streams.

Figure 5-4 depicts how rows that are rejected by the Lookup stage are processed by a corrective Transformer stage, where the failed references are set to a specific value and then merged with the output of the Lookup stage.

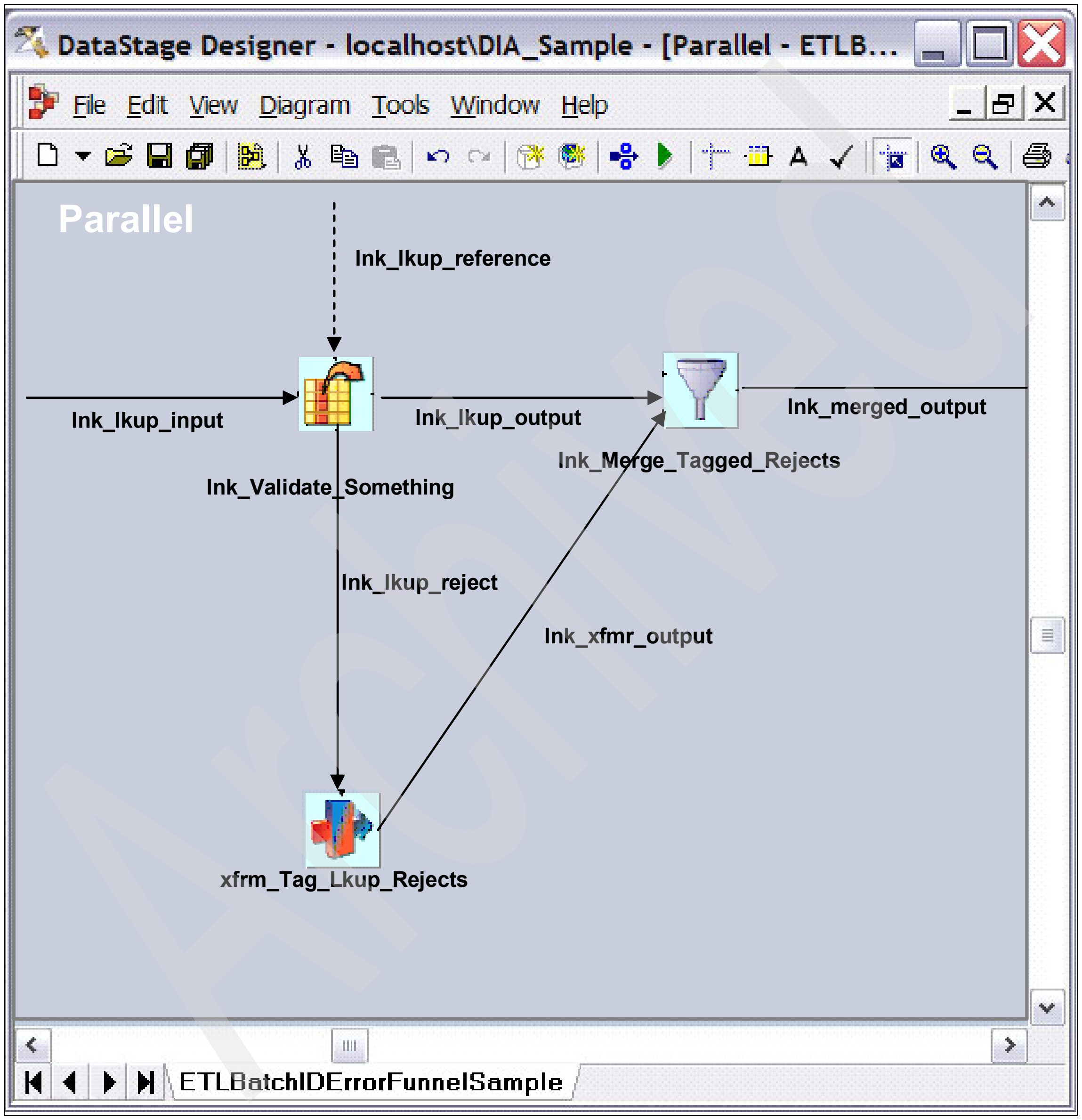


Figure 5-4 Error processing Lookup example

5.7 Component usage

DataStage offers a wealth of component types for building parallel ETL flows. This section provides guidelines appropriate use of various stages when building a parallel job flows.

5.7.1 Server Edition components

Avoid the use of Server Edition components in parallel job flows. DataStage Server components limit overall performance of large-volume job flows because many components such as the BASIC Transformer use interpreted pseudo-code. In clustered and MPP environments Server Edition components only run on the primary (conductor) node, severely impacting scalability and network resources.

The ability to use a DataStage Server component in a parallel job is intended only as a migration option for existing DataStage Server applications that might benefit by making use of parallel capabilities on SMP platforms.

The following DataStage Server components must be avoided in parallel job flows:

- BASIC Transformers
- BASIC Routines
- Server shared containers

BASIC routines are still appropriate, and necessary, for the job control components of a DataStage Job Sequence and Before/After Job Subroutines for parallel jobs.

5.7.2 Copy stage

For complex data flows, it is best to develop a job iteratively using the Copy stage as a placeholder. Because the Copy stage does not require an output link, it can be used at the end of a data flow in the following circumstances:

- For simple jobs with only two stages, the Copy stage must be used as a placeholder so that new stages can be inserted easily should future requirements change.
- Unless the force property is set to **True**, a Copy stage with a single input link and a single output link is optimized out of the final job flow at runtime.

5.7.3 Parallel datasets

When writing intermediate results between DataStage parallel jobs, always write to parallel datasets. Datasets achieve end-to-end parallelism across job boundaries by writing data in partitioned form, in sort order, and in native format. Used in this manner, parallel datasets effectively establish restart points in the event that a job (or sequence) needs to be re-run. Datasets offer parallel I/O on read and write operations, without overhead for format or data type conversions.

There might be instances when setting the environment variable APT_OLD_BOUNDED_LENGTH might be beneficial. When this environment variable is set (present in the environment), varchar columns are only stored using the actual data length. This might improve I/O performance (and reduce disk use) when processing a large number of varchar columns with a large maximum length and highly-variable data lengths.

Note: Because parallel datasets are platform and configuration-specific, they must not be used for long-term archive of source data.

5.7.4 Parallel Transformer stages

The DataStage parallel Transformer stage generates C code, which is compiled into a parallel component. For this reason, it is important to minimize the number of Transformers, and to use other stages (such as Copy) when derivations are not needed.

- The Copy stage must be used instead of a Transformer for simple operations:
  - Job Design placeholder between stages (unless the force option is set to **True**, the parallel framework optimizes this out at runtime)
  - Renaming columns
  - Dropping columns
- Default type conversions (see “Default and explicit type conversions” on page 426.)

  Rename, drop (if runtime column propagation is disabled), and default type conversion can also be performed by the output mapping tab of any stage.
- Never use the BASIC Transformer stage in large-volume job flows. Instead, user-defined functions and routines can expand Parallel Transformer capabilities. The BASIC Transformer is intended as a temporary-use migration choice for existing DataStage Server jobs containing complex routines. Even then its use must be restricted and the routines must be converted as soon as possible.

- Consider, if possible, implementing complex derivation expressions using regular patterns by Lookup tables instead of using a Transformer with nested derivations. For example, the derivation expression:

  ```
  If A=0,1,2,3 Then B=”X” If A=4,5,6,7 Then B=”C”
  ```

  In this situation, the expression could also be implemented with a lookup table containing values in column A and the corresponding values in column B.
- Optimize the overall job flow design to combine derivations from multiple Transformers into a single Transformer stage when possible.
- Because the Parallel Transformer is compiled, it is faster than the Interpreted Filter and Switch stages. The only time that Filter or Switch must be used is when the selection clauses need to be parameterized at runtime.
- The Modify stage can be used for non-default type conversions, null handling, and character string trimming. See 9.2, “Modify stage” on page 146.

5.7.5 BuildOp stages

BuildOps can only be used in the following circumstances:

- Complex reusable logic cannot be implemented using the Transformer.
- Existing Transformers do not meet performance requirements.

As always, performance must be tested in isolation to identify specific causes of bottlenecks.

5.8 Job design considerations for usage and impact analysis

It is important to capture the table definitions prior to construction of a job flow so that linkages are set up between the table definitions and the DataStage job whenever a table definition is loaded into a stage from the repository or saved into the repository from a stage. These relationships are the key to the DataStage usage analysis mechanism as well as the impact analysis. In other words, to identify which DataStage jobs in which a particular table definition is used, that table definition is required to be loaded into the job/stage from the repository or saved from the job/stage into the repository.

5.8.1 Maintaining JobDesign:Table definition connection

The following development procedures ensure that the linkage between job design and table definition is properly established and maintained.

To preserve the job design:table definition connection, DataStage developers should populate metadata on passive stages or links in the following ways:

- Performing a load to populate the stage with an existing table definition
- Performing a save from in the stage when changing or creating a new table definition
- Performing a drag and drop from the repository window onto the link

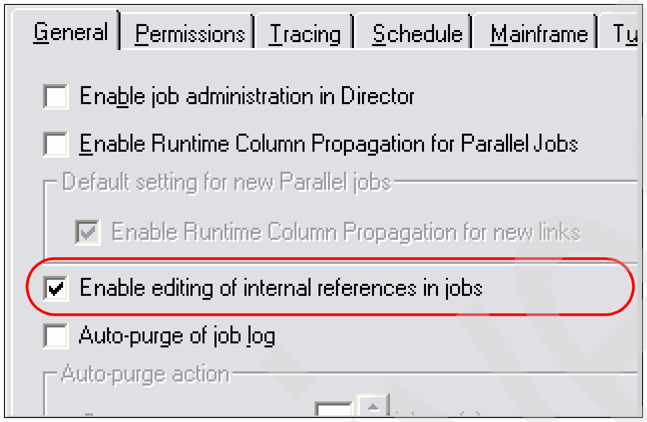
To preserve the job design:table definition connection, DataStage developers should not:

- In the Transformer stage, populate output links by dragging input columns to output columns
- Delete a table definition that is attached to a job design
- Rename table definition folders (careful, names are case-sensitive) containing table definitions that are in use. If the folder must be renamed, then the affected definitions must be dropped and reacquired
- Manually create a table definition in the designer without a save
- Modify table definitions without a save operation
- Change the column order in a sequential table definition

Changing the name of a properly acquired table or file does not break the metadata connection, neither does deleting it and recreating it.

5.8.2 Verifying the job design:table definition connection

To view the job design:table definition relationships in DataStage Designer, the "Enable editing of internal references in jobs" option must be selected in the project properties administrator client, as shown in Figure 5-5.



☐ Enable job administration in Director

☐ Enable Runtime Column Propagation for Parallel Jobs

Default setting for new Parallel jobs

☑ Enable Runtime Column Propagation for new links

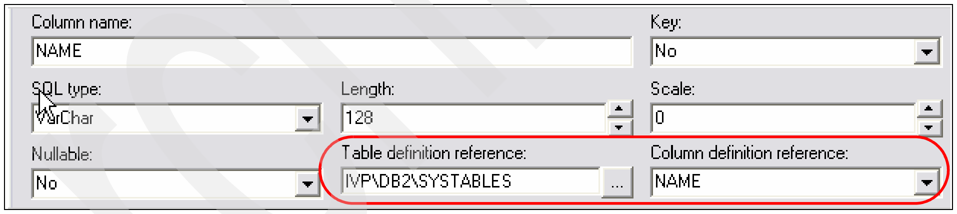
☑ Enable editing of internal references in jobs

☐ Auto-purge of job log

Auto-purge action

Figure 5-5 Enabling internal references in administrator

Once enabled, the relationships can be viewed in the stage editor on the Edit Column panel, as shown in Figure 5-6.



Column name: NAME		Key: No
SQL type: VarChar	Length: 128	Scale: 0
Nullable: No	Table definition reference: IVP\DB2\SYSTABLES	Column definition reference: NAME

Figure 5-6 Internal references in designer

Partitioning and collecting

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 was running sequentially, a *fan-out* icon is drawn on the link in the Designer canvas, as shown in Figure 6-1.

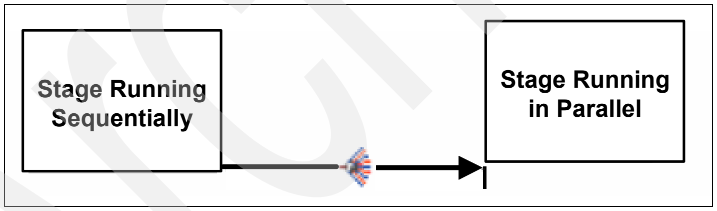


Figure 6-1 Fan-out icon

Collectors combine parallel partitions of a single link for sequential processing. Collectors only exist before stages running sequentially and when the previous stage is running in parallel, and are indicated by a fan-in icon as shown in Figure 6-2.

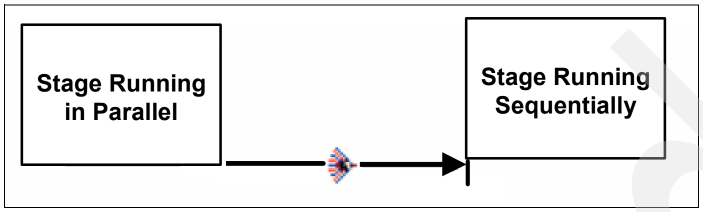


Figure 6-2 Collector icon

This section provides an overview of partitioning and collecting methods, and provides guidelines for appropriate use in job designs. It also provides tips for monitoring jobs running in parallel.

6.1 Partition types

Though partitioning allows data to be distributed across multiple processes running 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

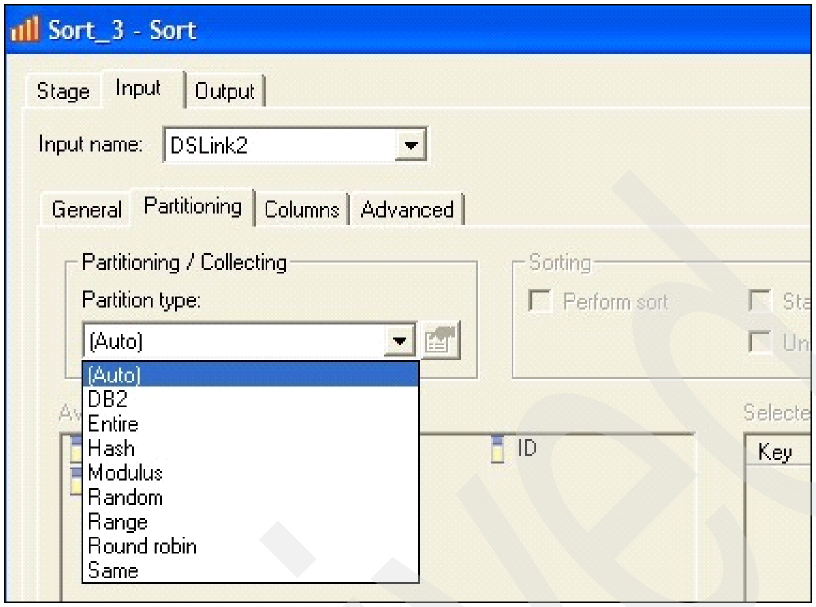
  Keyless partitioning distributes rows without regard to the actual data values. Separate types of keyless partitioning methods define the method of data distribution.

- Keyed partitioning

  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.

The default partitioning method used when links are created is *Auto partitioning*.

The partitioning method is specified in the input stage properties using the partitioning option, as shown in Figure 6-3 on page 94.



Input name: DSLink2

Partition type: (Auto)

Figure 6-3 Specifying partition method

6.1.1 Auto partitioning

This is the default partitioning method for newly-drawn links, Auto partitioning specifies that the parallel framework attempts to select the appropriate partitioning method at runtime. Based on the configuration file, datasets, and job design (stage requirements and properties), Auto partitioning selects between keyless (same, round-robin, entire) and keyed (hash) partitioning methods to produce functionally correct results and, in certain cases, to improve performance.

In the Designer canvas, links with Auto partitioning are drawn with the link icon, depicted in Figure 6-4.

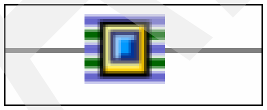
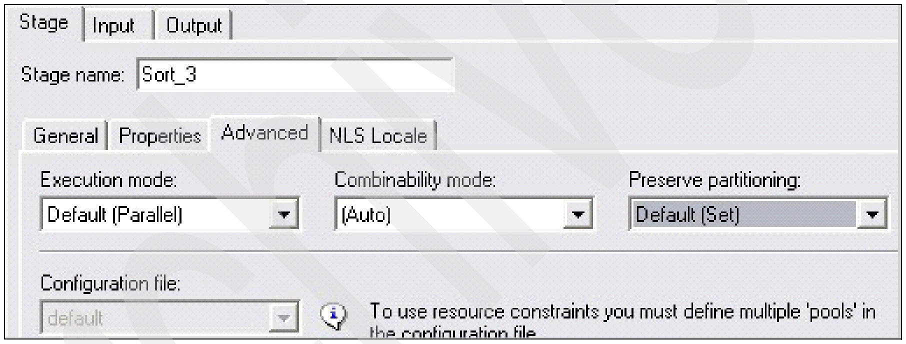


Figure 6-4 Auto partitioning icon

Auto partitioning is designed to allow beginner DataStage developers to construct simple data flows without having to understand the details of parallel design principles. However, the Auto partitioning method might not be the most efficient from an overall job perspective and in certain cases can lead to wrong results.

Furthermore, the parallel framework's ability to determine the appropriate partitioning method depends on the information available to it. In general, Auto partitioning ensures correct results when using built-in stages. However, because the parallel framework has no visibility into user-specified logic (such as Transformer or BuildOp stages) it might be necessary to specify a partitioning method for certain stages. 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 correct results.

The Preserve Partitioning flag is an internal hint that Auto partitioning uses to attempt to preserve previously ordered data (for example, on the output of a parallel sort). This flag is set automatically by certain stages (sort, for example), although it can be explicitly set or cleared in the advanced stage properties of a given stage, as shown in Figure 6-5.



Stage name: Sort_3

Execution mode: Default (Parallel)

Combinability mode: (Auto)

Preserve partitioning: Default (Set)

Configuration file: default

To use resource constraints you must define multiple 'pools' in the configuration file

Figure 6-5 Preserve Partitioning option

The Preserve Partitioning flag is part of the dataset structure, and its state is stored in persistent datasets.

There are cases when the input stage requirements prevent partitioning from being preserved. For example, when the upstream partitioning scheme is round-robin, but the stage at hand is a Join. In this case, the Join requires the data to be partitioned by hash on the Join key. In these instances, if the Preserve Partitioning flag was set, a warning is placed in the Director log indicating the parallel framework was unable to preserve partitioning for a specified stage.

6.1.2 Keyless partitioning

Keyless partitioning methods distribute rows without examining the contents of the data. The partitioning methods are described in Table 6-1 on page 96.

Table 6-1 Partitioning methods

Keyless Partition Method	Description
Same	Retains existing partitioning from previous stage.
Round-robin	Distributes rows evenly across partitions, in a round-robin partition assignment.
Random	Distributes rows evenly across partitions in a random partition assignment.
Entire	Each partition receives the entire dataset.

Same partitioning

Same partitioning performs no partitioning to the input dataset. Instead, it retains the partitioning from the output of the upstream stage, as shown in Figure 6-6.

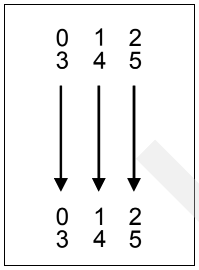


Figure 6-6 Same partitioning

Same partitioning does not move data between partitions (or, in the case of a cluster or grid, between servers), and is appropriate when trying to preserve the grouping of a previous operation (for example, a parallel Sort).

In the Designer canvas, links that have been specified with Same partitioning are drawn with a horizontal line partitioning icon, as in Figure 6-7.

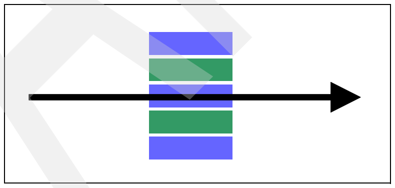


Figure 6-7 Same partitioning icon

It is important to understand the impact of Same partitioning in a given data flow. Because Same 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.

Note: Minimize the use of SAME partitioning, using only when necessary.

Round-robin partitioning

Round-robin partitioning evenly distributes rows across partitions in a round-robin assignment, similar to dealing cards. Round-robin partitioning has a fairly low overhead. It is shown in Figure 6-8.

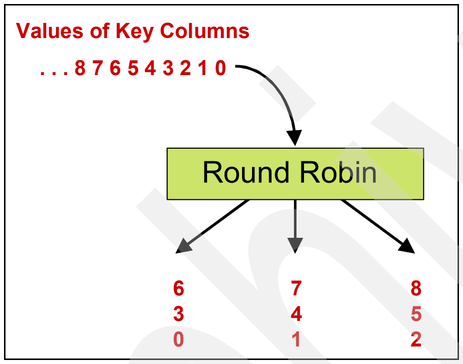


Figure 6-8 Round-robin partitioning

Because optimal parallel processing occurs when all partitions have the same workload, round-robin partitioning is useful for redistributing data that is highly skewed (there are an unequal number of rows in each partition).

Random partitioning

Like Round-robin, Random partitioning evenly distributes rows across partitions, but using a random assignment. As a result, the order that rows are assigned to a particular partition differ between job runs.

Because the random partition number must be calculated, Random partitioning has a slightly higher overhead than Round-robin partitioning.

Though 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, though it shares basic principle of Round-robin partitioning, it has a slightly larger overhead.

Entire partitioning

Entire partitioning distributes a complete copy of the entire dataset to each partition, and is illustrated in Figure 6-9.

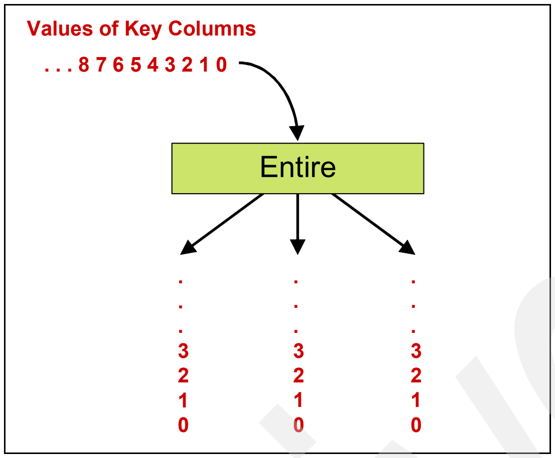


Figure 6-9 Entire partitioning

Entire partitioning is useful for distributing the reference data of a Lookup task (this might or might not involve the Lookup stage).

On clustered and grid implementations, Entire partitioning might have a performance impact, as the complete dataset must be distributed across the network to each node.

6.1.3 Keyed partitioning

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. Keyed partitioning is described in Table 6-2 on page 99.

Table 6-2 Keyed partitioning

Keyed Partitioning	**Description**
Hash	Assigns rows with the same values in one or more key columns to the same partition using an internal hashing algorithm.
Modulus	Assigns rows with the same values in a single integer key column to the same partition using a simple modulus calculation.
Range	Assigns rows with the same values in one or more key columns to the same partition using a specified *range map* generated by pre-reading the dataset.
DB2	For DB2 Enterprise Server Edition with DPF (DB2/UDB) only Matches the internal partitioning of the specified source or target table.

6.1.4 Hash partitioning

Hash partitioning assigns rows with the same values in one or more key columns to the same partition using an internal hashing algorithm. This is depicted in Figure 6-10.

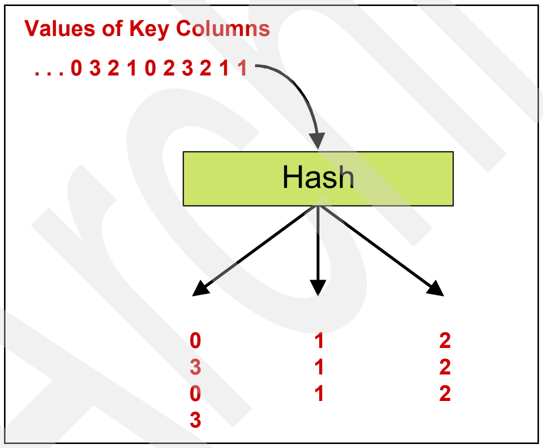


Figure 6-10 Hash partitioning

If the source data values are evenly distributed in these key columns, and there are a large number of unique values, then the resulting partitions are of relatively equal size.

As an example of hashing, consider the sample dataset in Table 6-3.

Table 6-3 Sample hash dataset

ID	LName	FName	Address
	Ford	Henry	66 Edison Avenue
	Ford	Clara	66 Edison Avenue
	Ford	Edsel	7900 Jefferson
	Ford	Eleanor	7900 Jefferson
	Dodge	Horace	17840 Jefferson
	Dodge	John	75 Boston Boulevard
	Ford	Henry	4901 Evergreen
	Ford	Clara	4901 Evergreen
	Ford	Edsel	1100 Lakeshore
10	Ford	Eleanor	1100 Lakeshore

Hashing on the LName key column produces the results depicted in Table 6-4 and Table 6-5.

Table 6-4 Partition 0

ID	LName	FName	Address
	Dodge	Horace	17840 Jefferson
	Dodge	John	75 Boston Boulevard

Table 6-5 Partition 1

ID	LName	FName	Address
	Ford	Henry	66 Edison Avenue
	Ford	Clara	66 Edison Avenue
	Ford	Edsel	7900 Jefferson
	Ford	Eleanor	7900 Jefferson
	Ford	Henry	4901 Evergreen
	Ford	Clara	4901 Evergreen
	Ford	Edsel	1100 Lakeshore
10	Ford	Eleanor	1100 Lakeshore

In this case, there are more instances of Ford than Dodge, producing partition skew, which would impact performance. In this example the number of unique values limit the degree of parallelism, regardless of the actual number of nodes in the parallel configuration file.

Using the same source dataset, hash partitioning on the LName and FName key columns yields the distribution with a 4-node configuration file depicted in Table 6-6, Table 6-7, Table 6-8, and Table 6-9.

Table 6-6 Partition 0

ID	LName	FName	Address
	Ford	Clara	66 Edison Avenue
	Ford	Clara	4901 Evergreen

Table 6-7 Partition 1

ID	LName	FName	Address
	Ford	Edsel	7900 Jefferson
	Dodge	Horace	17840 Jefferson
	Ford	Edsel	1100 Lakeshore

Table 6-8 Partition 2

ID	LName	FName	Address
	Ford	Eleanor	7900 Jefferson
	Dodge	John	75 Boston Boulevard
10	Ford	Eleanor	1100 Lakeshore

Table 6-9 Partition 3

ID	LName	FName	Address
	Ford	Henry	66 Edison Avenue
	Ford	Henry	4901 Evergreen

In this example, the key column combination of LName and FName yields improved data distribution and a greater degree of parallelism. Only the unique combination of key column values appear in the same partition when used for hash partitioning. 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 using the number of partitions as the divisor. The remainder is used to assign the value to a given 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 must be used if you have a single integer key column. Modulus partitioning cannot be used for composite keys, or for a non-integer key column.

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 a sufficient number of unique values, Range partitioning ensures balanced workload by assigning an approximately equal number of rows to each partition, unlike Hash and Modulus partitioning where partition skew is dependent on the actual data distribution. This is depicted in Figure 6-11.

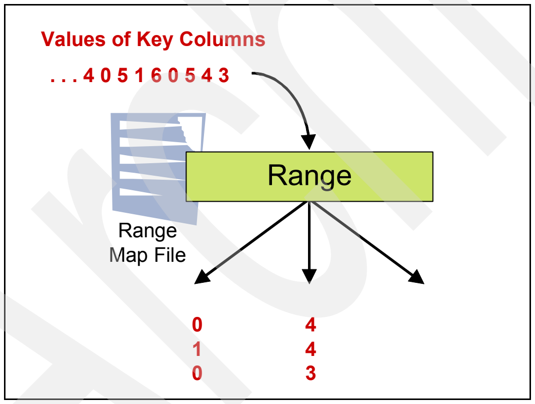


Figure 6-11 Range partitioning

To achieve this balanced distribution, Range partitioning must read the dataset twice: the first to create a Range Map file, and the second to actually partition the data in a flow using the Range Map. A Range Map file is specific to a given 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 re-used.

It is important to note that if the data distribution changes without recreating the Range Map, partition balance is skewed, defeating the intention of Range partitioning. Also, if new data values are processed outside of the range of a given 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 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). 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 using the DB2/UDB Enterprise stage, data is partitioned to match the internal partitioning of the target DB2 table using the *DB2* partitioning method.

DB2 partitioning can only be specified for target DB2/UDB Enterprise stages. To maintain partitioning on data read from a DB2/UDB Enterprise stage, use Same partitioning on the input to downstream stages.

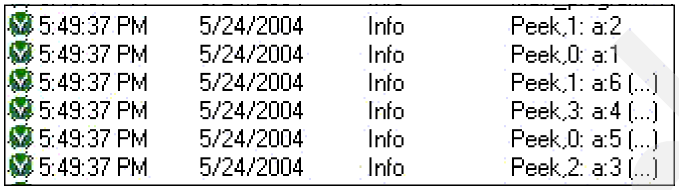
6.2 Monitoring partitions

At runtime, the DataStage parallel framework determines the degree of parallelism for each stage using:

- The parallel configuration file (APT_CONFIG_FILE)
- The degree of parallelism of existing source and target datasets (and, in certain cases, databases)
- If specified, a stage’s node pool (stage/Advanced properties)

This information is detailed in the parallel job score, which is output to the Director job log when the environment variable APT_DUMP_SCORE is set to **True**. Specific details on interpreting the parallel job score can be found in Appendix E, “Understanding the parallel job score” on page 401.

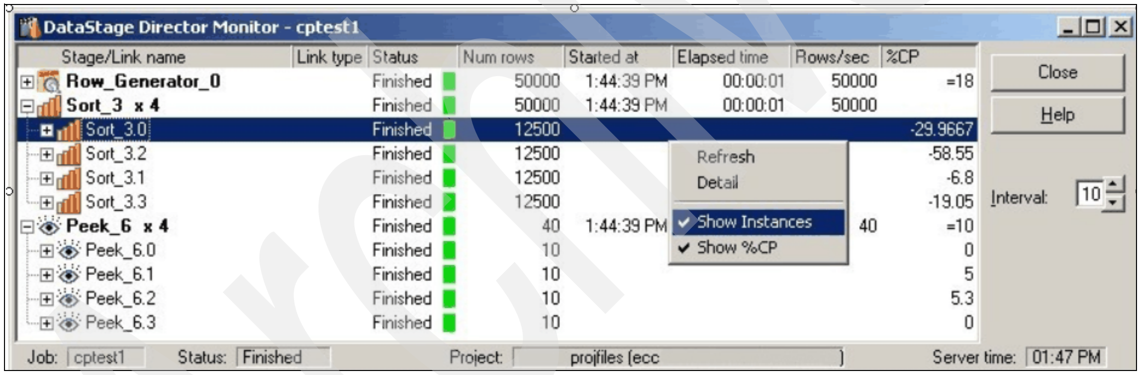
Partitions are assigned numbers, starting at zero. The partition number is appended to the stage name for messages written to the Director log, as shown in Figure 6-12, where the Peek stage is running with four degrees of parallelism (partition numbers zero through 3).



```
5:49:37 PM   5/24/2004   Info   Peek,1: a:2
5:49:37 PM   5/24/2004   Info   Peek,0: a:1
5:49:37 PM   5/24/2004   Info   Peek,1: a:6 [...]
5:49:37 PM   5/24/2004   Info   Peek,3: a:4 [...]
5:49:37 PM   5/24/2004   Info   Peek,0: a:5 [...]
5:49:37 PM   5/24/2004   Info   Peek,2: a:3 [...]
```

Figure 6-12 Partition numbers as shown in Director log

To display row counts per partition in the Director Job Monitor window, right-click anywhere in the window, and select the **Show Instances** option, as shown in Figure 6-13. This is useful in determining the distribution across parallel partitions (skew). In this instance, the stage named Sort_3 is running across four partitions (x 4 next to the stage name), and each stage is processing an equal number (12,500) of rows for an optimal balanced workload.



DataStage Director Monitor - cptest1

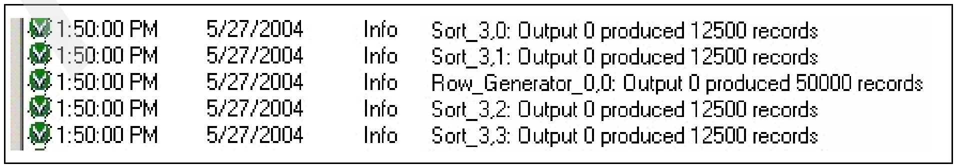
Stage/Link name	Link type	Status	Num rows	Started at	Elapsed time	Rows/sec	%CP
Row_Generator_0		Finished	50000	1:44:39 PM	00:00:01	50000	=18
Sort_3 x 4		Finished	50000	1:44:39 PM	00:00:01	50000	
Sort_3.0		Finished	12500				-29.9667
Sort_3.2		Finished	12500				-58.55
Sort_3.1		Finished	12500				-6.8
Sort_3.3		Finished	12500				-19.05
Peek_6 x 4		Finished	40	1:44:39 PM		40	=10
Peek_6.0		Finished	10				0
Peek_6.1		Finished	10				5
Peek_6.2		Finished	10				5.3
Peek_6.3		Finished	10				0

Job: cptest1 Status: Finished Project: projfiles (ecc) Server time: 01:47 PM

Interval: 10

Figure 6-13 Director job monitor row counts by partition

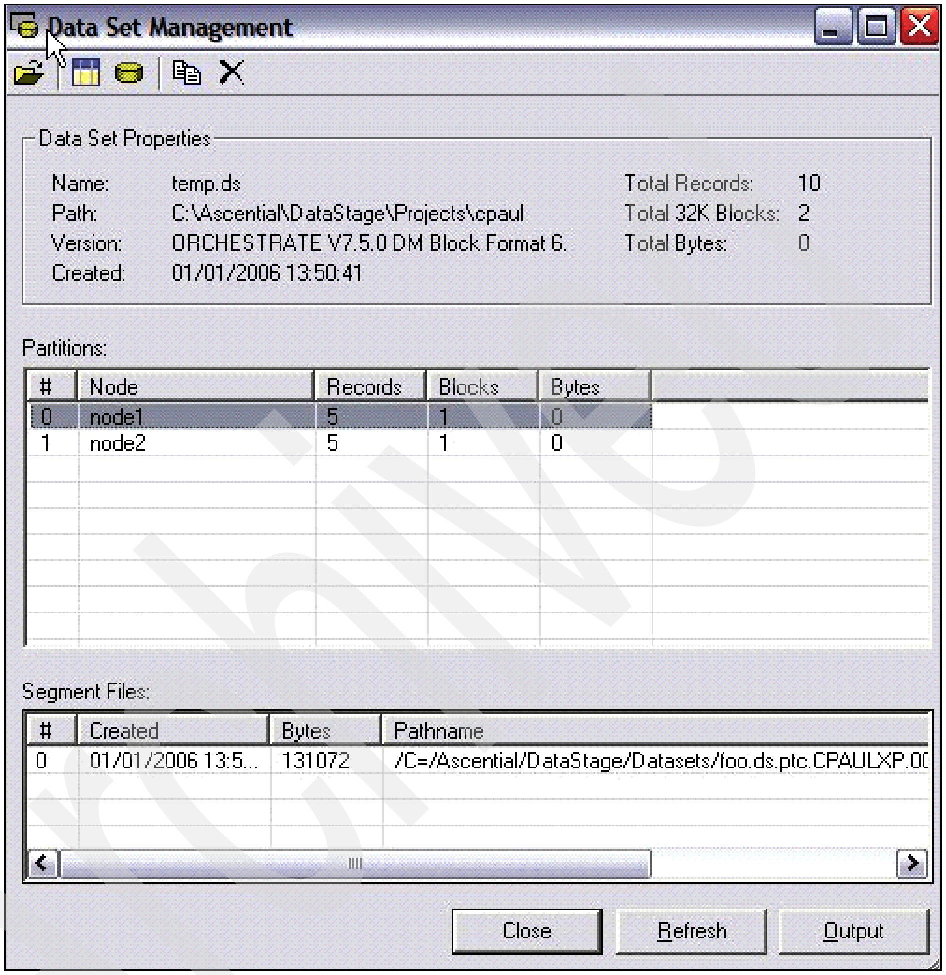
Setting the environment variable APT_RECORD_COUNTS outputs the row count per link per partition to the Director log as each stage/node completes processing, as illustrated in Figure 6-14.



```
1:50:00 PM   5/27/2004   Info   Sort_3,0: Output 0 produced 12500 records
1:50:00 PM   5/27/2004   Info   Sort_3,1: Output 0 produced 12500 records
1:50:00 PM   5/27/2004   Info   Row_Generator_0,0: Output 0 produced 50000 records
1:50:00 PM   5/27/2004   Info   Sort_3,2: Output 0 produced 12500 records
1:50:00 PM   5/27/2004   Info   Sort_3,3: Output 0 produced 12500 records
```

Figure 6-14 Output of APT_RECORD_COUNTS in Director log

The Dataset Management tool (available in the Tools menu of Designer or Director) can be used to identify the degree of parallelism and number of rows per partition for an existing persistent dataset, as shown in Figure 6-15.



Data Set Properties

Name:	temp.ds	Total Records:	10
Path:	C:\Ascential\DataStage\Projects\cpaul	Total 32K Blocks:	2
Version:	ORCHESTRATE V7.5.0 DM Block Format 6.	Total Bytes:	0
Created:	01/01/2006 13:50:41		

Partitions:

#	Node	Records	Blocks	Bytes
0	node1	5	1	0
1	node2	5	1	0

Segment Files:

#	Created	Bytes	Pathname
0	01/01/2006 13:5...	131072	/C=/Ascential/DataStage/Datasets/foo.ds.ptc.CPAULXP.0(

Figure 6-15 Dataset Management tool

In a non-graphical way, the **`orchadmin`** command line utility on the DataStage server can also be used to examine a given parallel dataset.

6.3 Partition methodology

Given the numerous options for keyless and keyed partitioning, the following objectives help to form a methodology for assigning partitioning:

- Choose a partitioning method that gives close to an equal number of rows in each partition, and which minimizes overhead.

  This ensures that the processing workload is evenly balanced, minimizing overall run time.

- The partition method must match the business requirements and stage functional requirements, assigning related records to the same partition if required

  Any stage that processes groups of related records (generally using one or more key columns) must be partitioned using a keyed partition method.

  This includes, but is not limited to the following stages:

  – Aggregator
  – Change Capture
  – Change Apply
  – Join, Merge
  – Remove Duplicates
  – Sort

  It might also be necessary for Transformers and BuildOps that process groups of related records.

  In satisfying the requirements of this second objective, it might not be possible to choose a partitioning method that gives close to an equal number of rows in each partition.

- Unless partition distribution is highly skewed, minimize repartitioning, especially in cluster or grid configurations

  Repartitioning data in a cluster or grid configuration incurs the overhead of network transport.

- Partition method must not be overly complex

  The simplest method that meets these objectives generally is the most efficient and yields the best performance.

Using these objectives as a guide, the following methodology can be applied:

1. Start with Auto partitioning (the default)
2. Specify Hash partitioning for stages that require groups of related records
   a. Specify only the key columns that are necessary for correct grouping as long as the number of unique values is sufficient
   b. Use Modulus partitioning if the grouping is on a single integer key column
   c. Use Range partitioning if the data is highly skewed and the key column values and distribution do not change significantly over time (Range Map can be reused)
3. If grouping is not required, use round-robin partitioning to redistribute data equally across all partitions

   This is especially useful if the input dataset is highly skewed or sequential
4. Use Same partitioning to optimize end-to-end partitioning and to minimize repartitioning

   Be mindful that Same partitioning retains the degree of parallelism of the upstream stage

   In a flow, examine up-stream partitioning and sort order and attempt to preserve for down-stream processing. This might require re-examining key column usage in stages and re-ordering stages in a flow (if business requirements permit).

   Across jobs, persistent datasets can be used to retain the partitioning and sort order. This is particularly useful if downstream jobs are run with the same degree of parallelism (configuration file) and require the same partition and sort order.

6.4 Partitioning examples

In this section, we apply the partitioning methodology defined earlier to several example job flows.

6.4.1 Partitioning example 1: Optimized partitioning

The Aggregator stage only outputs key column and aggregate result columns. To add aggregate columns to every detail row, a Copy stage is used to send the detail rows to an Inner Join and an Aggregator. The output of the Aggregator is sent to the second input of the Join. The standard solution is to Hash partition (and Sort) the inputs to the Join and Aggregator stages as shown in Figure 6-16.

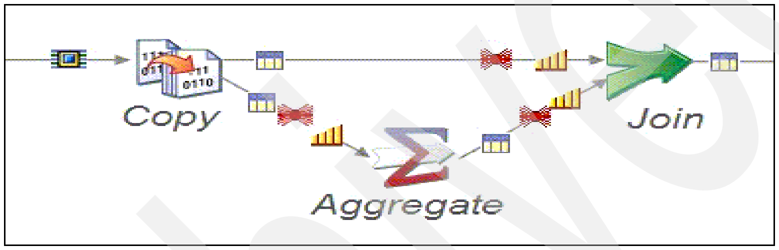


Figure 6-16 Standard Partitioning assignment

However, on closer inspection, the partitioning and sorting of this scenario can be optimized. Because the Join and Aggregator use the same partition keys and sort order, we can move the Hash partition and Sort before the Copy stage, and apply Same partitioning to the downstream links, as shown in Figure 6-17.

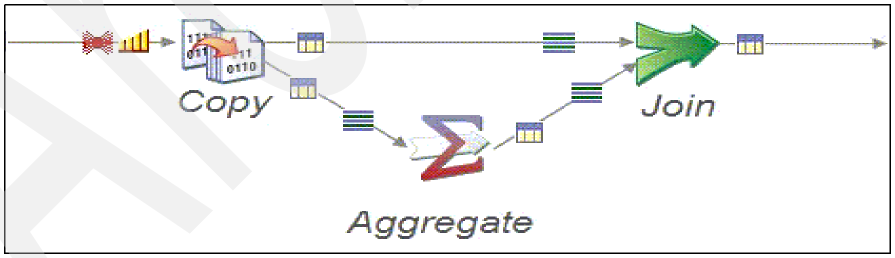


Figure 6-17 Optimized Partitioning assignment

This example is revisited in Chapter 7, “Sorting” on page 115 because there is one final step necessary to optimize the sorting in this example.

6.4.2 Partitioning example 2: Use of Entire partitioning

In this example, a Transformer is used to extract data from a single header row of an input file. In the Transformer, a new output column is defined on the header and detail links using a single constant value derivation. This column is used as the key for a subsequent Inner Join to attach the header values to every detail row. Using a standard solution, both inputs to the Join are Hash partitioned and sorted on this single join column (either explicitly, or through Auto partitioning). This is depicted in Figure 6-18.

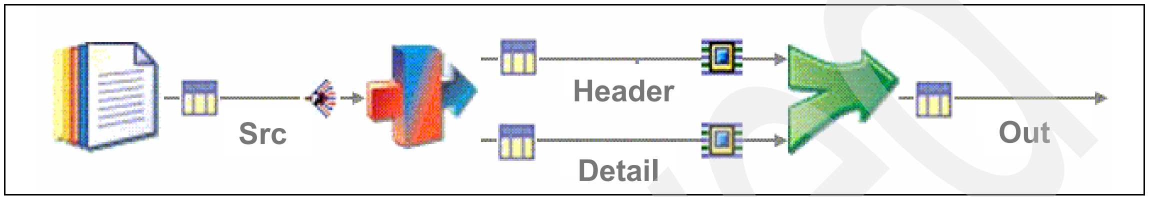


Figure 6-18 Standard partitioning assignment for a Join stage

Although Hash partitioning guarantees correct results for stages that require groupings of related records, it is not always the most efficient solution, depending on the business requirements. Although functionally correct, the solution has one serious limitation. Remembering that the degree of parallel operation is limited by the number of distinct values, the single value join column assigns all rows to a single partition, resulting in sequential processing.

To optimize partitioning, consider that the single header row is really a form of reference data. An optimized solution is to alter the partitioning for the input links to the Join stage, as depicted in Figure 6-19.

- Use round-robin partitioning on the detail input to distribute rows across all partitions evenly.
- Use Entire partitioning on the header input to copy the single header row to all partitions.

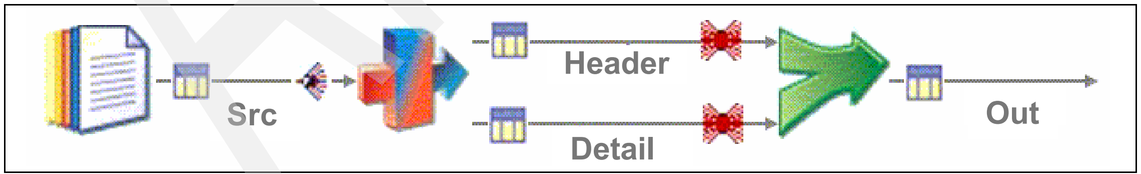
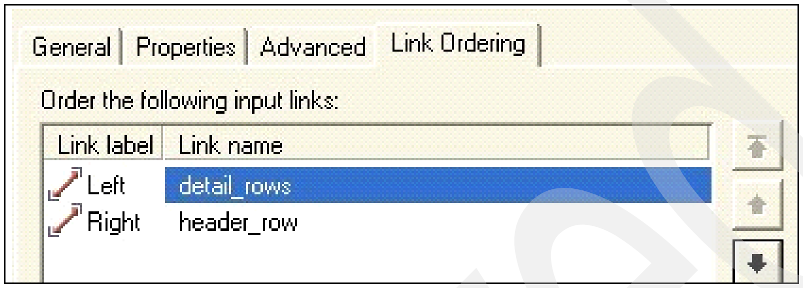


Figure 6-19 Optimized Partitioning assignment based on business requirements

Because we are joining on a single value, there is no need to pre-sort the input to the Join. We revisit this in the Sorting discussion.

To process a large number of detail records, the link order of the Inner Join is significant. The Join stage operates by reading a single row from the Left input and reading all rows from the Right input that match the key values. For this reason, the link order in this example must be set so that the single header row is assigned to the Right input, and the detail rows are assigned to the Left input, as shown in Figure 6-20.



Order the following input links:

Link label	Link name
Left	detail_rows
Right	header_row

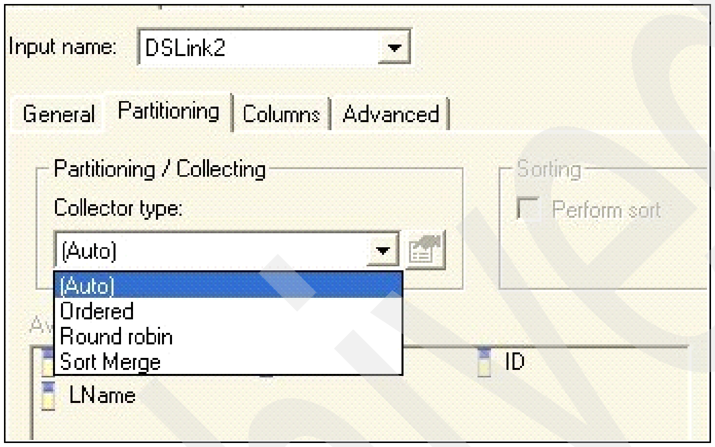
Figure 6-20 Specifying link order in Join stage

If defined in reverse of this order, the Join attempts to read all detail rows from the right input (because they have the same key column value) into memory.

For advanced users, there is one further detail in this example. Because the Join waits until it receives an End of Group (new key value) or End of Data (no more rows on the input dataset) from the Right input, the detail rows in the Left input buffer to disk to prevent a deadlock. (See 12.4, “Understanding buffering” on page 180). Changing the output derivation on the header row to a series of numbers instead of a constant value establishes the End of Group and prevent buffering to disk.

6.5 Collector types

Collectors combine parallel partitions of an input dataset (single link) into a single input stream to a stage running sequentially. Like partitioning methods, the collector method is defined in the stage Input/Partitioning properties for any stage running sequentially, when the previous stage is running in parallel, as shown in Figure 6-21.


Input name: DSLink2

Collector type: (Auto)

Figure 6-21 Specifying collector method

6.5.1 Auto collector

The Auto collector reads rows from partitions in the input dataset without blocking if a row is unavailable on a particular partition. For this reason, the order of rows in an Auto collector is undefined, and might vary between job runs on the same dataset. Auto is the default collector method.

6.5.2 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 must wait for a row to appear in a particular partition.

However, there is a specialized example where the Round-robin collector might be appropriate. Consider Figure 6-22, where data is read sequentially and passed to a round-robin partitioner.

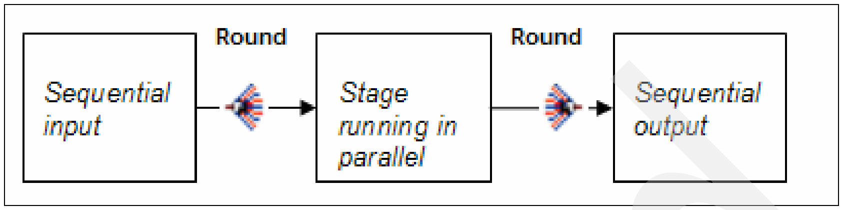


Figure 6-22 Round-robin collector example

Assuming the 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 is because a Round-robin collector reads from partitions using the same partition order that a Round-robin partitioner assigns rows to parallel partitions.

6.5.3 Ordered collector

An Ordered collector reads all rows from the first partition, then reads all rows from the next partition until all rows in the dataset have been collected.

Ordered collectors are generally only useful if the input dataset has been Sorted and Range partitioned on the same key columns. In this scenario, an Ordered collector generates a sequential stream in sort order.

6.5.4 Sort Merge collector

If the input dataset is sorted in parallel, the Sort Merge collector generates a sequential stream of rows in globally sorted order. The Sort Merge collector requires one or more key columns to be defined, and these must be the same columns, in the same order, as used to sort the input dataset in parallel. Row order is undefined for non-key columns.

6.6 Collecting methodology

Given the options for collecting data into a sequential stream, the following guidelines form a methodology for choosing the appropriate collector type:

1. When output order does not matter, use Auto partitioning (the default)
2. When the input dataset has been sorted in parallel, use Sort Merge collector to produce a single, globally sorted stream of rows
3. When the input dataset has been sorted in parallel and Range partitioned, the Ordered collector might be more efficient
4. Use a Round-robin collector to reconstruct rows in input order for Round-robin partitioned input datasets, as long as the dataset has not been repartitioned or reduced.

7

Sorting

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 is the method the SQL databases use for an ORDER BY clause, as illustrated in the following example, sorting on primary key LName (ascending), secondary key FName (descending).

The input data is shown in Table 7-1.

Table 7-1 Input data

ID	LName	FName	Address
	Ford	Henry	66 Edison Avenue
	Ford	Clara	66 Edison Avenue
	Ford	Edsel	7900 Jefferson
	Ford	Eleanor	7900 Jefferson
	Dodge	Horace	17840 Jefferson
	Dodge	John	75 Boston Boulevard
	Ford	Henry	4901 Evergreen

ID	LName	FName	Address
	Ford	Clara	4901 Evergreen
	Ford	Edsel	1100 Lakeshore
10	Ford	Eleanor	1100 Lakeshore

After Sorting by LName, FName, the results are as in Table 7-2.

Table 7-2 Sort results

ID	LName	FName	Address
	Dodge	John	75 Boston Boulevard
	Dodge	Horace	17840 Jefferson
	Ford	Henry	66 Edison Avenue
	Ford	Henry	4901 Evergreen
	Ford	Eleanor	7900 Jefferson
10	Ford	Eleanor	1100 Lakeshore
	Ford	Edsel	7900 Jefferson
	Ford	Edsel	1100 Lakeshore
	Ford	Clara	66 Edison Avenue
	Ford	Clara	4901 Evergreen

However, in most cases there is no 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 done in parallel.

For example, the Remove Duplicates stage selects either the first or last row from each group of an input dataset sorted by one or more key columns. Other stages (for example, Sort Aggregator, Change Capture, Change Apply, Join, Merge) require pre-sorted groups of related records.

7.1 Partition and sort keys

For parallel sort in DataStage (DS) parallel jobs, the following qualifications apply:

- Partitioning is used to gather related records, assigning rows with the same key column values to the same partition.
- Sorting is used to establish group order in each partition, based on one or more key columns.

Note: By definition, when data is re-partitioned, sort order is not maintained. To restore row order and groupings, a sort is required after repartitioning.

In the following example, the input dataset from Table 7-1 on page 115 is partitioned on the LName and FName columns. Given a 4-node configuration file, you would see the results depicted in Table 7-3, Table 7-4, Table 7-5, and Table 7-6 on page 118.

Table 7-3 Partition 0

ID	LName	FName	Address
	Ford	Clara	66 Edison Avenue
	Ford	Clara	4901 Evergreen

Table 7-4 Partition 1

ID	LName	FName	Address
	Ford	Edsel	7900 Jefferson
	Dodge	Horace	17840 Jefferson
	Ford	Edsel	1100 Lakeshore

Table 7-5 Partition 2

ID	LName	FName	Address
	Ford	Eleanor	7900 Jefferson
	Dodge	John	75 Boston Boulevard
10	Ford	Eleanor	1100 Lakeshore

Table 7-6 Partition 3

ID	LName	FName	Address
	Ford	Henry	66 Edison Avenue
	Ford	Henry	4901 Evergreen

Applying a parallel sort to this partitioned input dataset, using the primary key column LName (ascending) and secondary key column FName (descending), would generate the resulting datasets depicted in Table 7-7, Table 7-8, Table 7-9, and Table 7-10.

Table 7-7 Partition 0

ID	LName	FName	Address
	Ford	Clara	66 Edison Avenue
	Ford	Clara	4901 Evergreen

Table 7-8 Partition 1

ID	LName	FName	Address
	Dodge	Horace	17840 Jefferson
	Ford	Edsel	7900 Jefferson
	Ford	Edsel	1100 Lakeshore

Table 7-9 Partition 2

ID	LName	FName	Address
	Dodge	John	75 Boston Boulevard
	Ford	Eleanor	7900 Jefferson
10	Ford	Eleanor	1100 Lakeshore

Table 7-10 Partition 3

ID	LName	FName	Address
	Ford	Henry	66 Edison Avenue
	Ford	Henry	4901 Evergreen

The partition and sort keys do not have to match. For example, secondary sort keys can be used to establish order in a group for selection with the Remove Duplicates stage (which can specify first or last duplicate to retain). Say that an input dataset consists of order history based on CustID and Order Date. Using Remove Duplicates, you can select the most recent order for a given customer.

To satisfy those requirements you could perform the following steps:

1. Partition on CustID to group related records
2. Sort on OrderDate in Descending order
3. Remove Duplicates on CustID, with Duplicate To Retain=First

7.2 Complete (Total) sort

If a single, sequential ordered result is needed, in general it is best to use a two step process:

1. Partition and parallel Sort on key columns.
2. Use a Sort Merge collector on these same key columns to generate a sequential, ordered result set.

This is similar to the way parallel database engines perform their parallel sort operations.

7.3 Link sort and Sort stage

DataStage provides two methods for parallel sorts:

- Standalone Sort stage

  This is used when execution mode is set to Parallel.

- Sort on a link

  This is used when using a keyed input partitioning method.

By default, both methods use the same internal sort package (the tsort operator).

The Link sort offers fewer options, but is easier to maintain in a DataStage job, as there are fewer stages on the design canvas. The Standalone sort offers more options, but as a separate stage makes job maintenance more complicated.

In general, use the Link sort unless a specific option is needed on the stand-alone stage. Most often, the standalone Sort stage is used to specify the Sort Key mode for partial sorts.

7.3.1 Link sort

Sorting on a link is specified on the Input/Partitioning stage options, when specifying a keyed partitioning method. (Sorting on a link is not available with Auto partitioning, although the DS parallel framework might insert a Sort if required). When specifying key columns for partitioning, the Perform Sort option is checked. In the Designer canvas, links that have sort defined have a Sort icon in addition to the Partitioning icon, as shown in Figure 7-1.

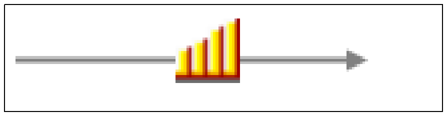
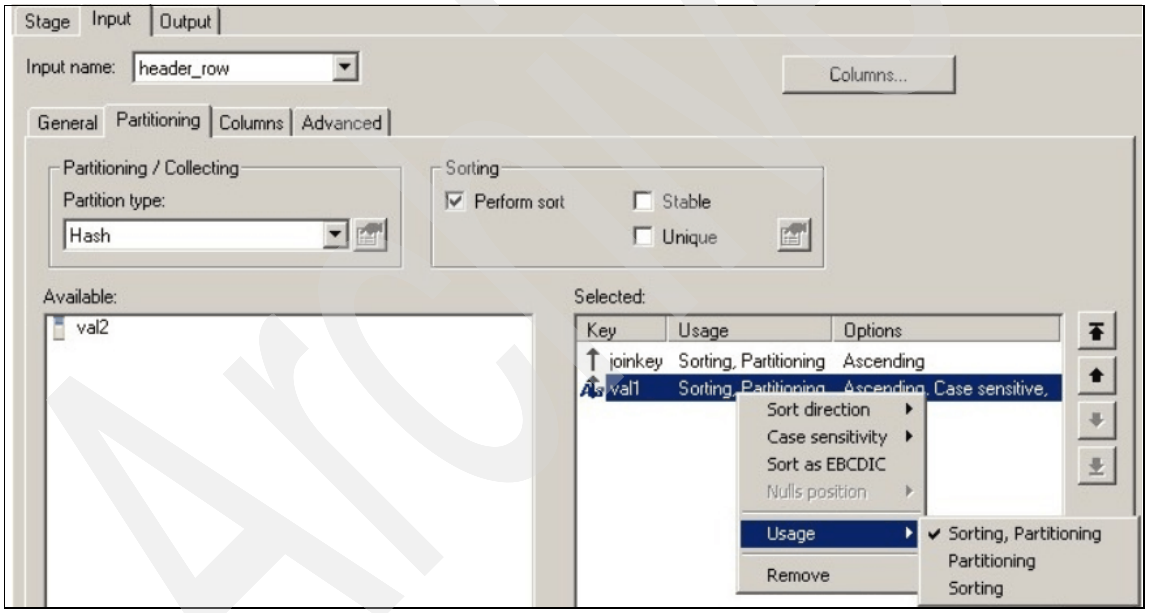


Figure 7-1 Link Sort Icon

Additional properties can be specified by right-clicking the key column, as shown in Figure 7-2.



Input name: header_row

Partitioning / Collecting
Partition type: Hash

Sorting
[✓] Perform sort
[ ] Stable
[ ] Unique

Available:
val2

Selected:

Key	Usage	Options
joinkey	Sorting, Partitioning	Ascending
val1	Sorting, Partitioning	Ascending, Case sensitive,

Usage: ✓ Sorting, Partitioning

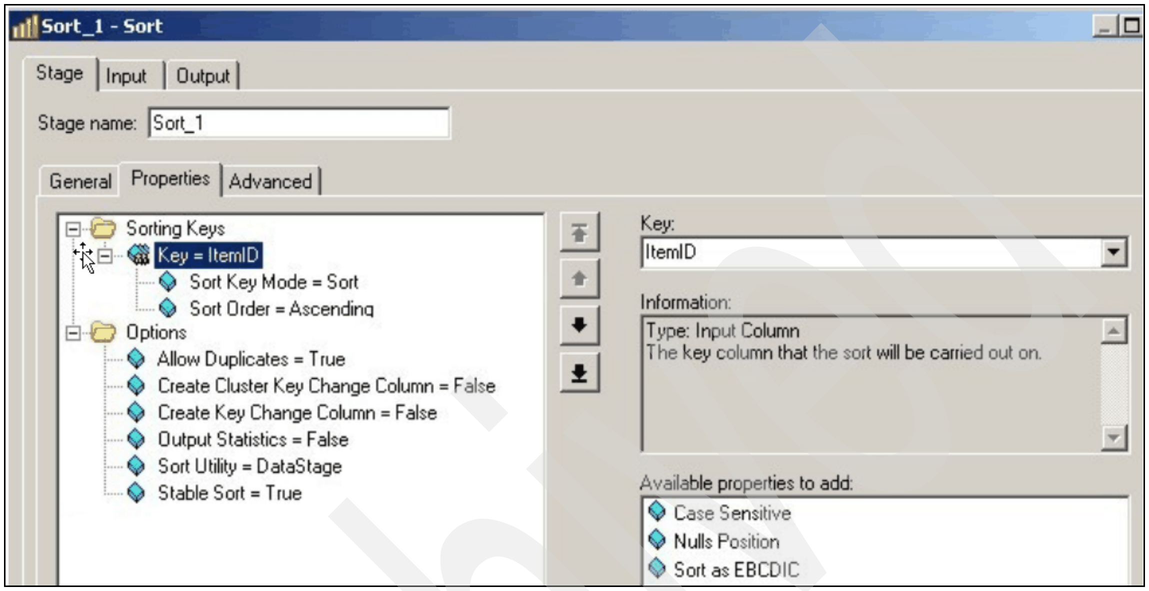
Figure 7-2 Specifying Link Sort options

Key column options let the developer specify the following options:

- Key column usage: sorting, partitioning, or both
- Sort direction: Ascending or Descending
- Case sensitivity (strings)
- Sorting character set: ASCII (default) or EBCDIC (strings)
- Position of nulls in the result set (for nullable columns)

7.3.2 Sort stage

The standalone Sort stage offers more options than the sort on a link, as depicted in Figure 7-3.



Stage name: Sort_1

- Sorting Keys
  - Key = ItemID
    - Sort Key Mode = Sort
    - Sort Order = Ascending
- Options
  - Allow Duplicates = True
  - Create Cluster Key Change Column = False
  - Create Key Change Column = False
  - Output Statistics = False
  - Sort Utility = DataStage
  - Stable Sort = True

Key: ItemID

Information:
Type: Input Column
The key column that the sort will be carried out on.

Available properties to add:
- Case Sensitive
- Nulls Position
- Sort as EBCDIC

Figure 7-3 Sort stage options

Specifically, the following properties are not available when sorting on a link:

- ► Sort Key Mode (a particularly important performance optimization)
- ► Create Cluster Key Change Column
- ► Create Key Change Column
- ► Output Statistics
- ► Sort Utility (do not change this)
- ► Restrict Memory Usage

Of the options only available in the standalone Sort stage, the Sort Key Mode is most frequently used.

Note: The Sort Utility option is an artifact of previous releases. Always specify the DataStage Sort Utility, which is significantly faster than a UNIX sort.

7.4 Stable sort

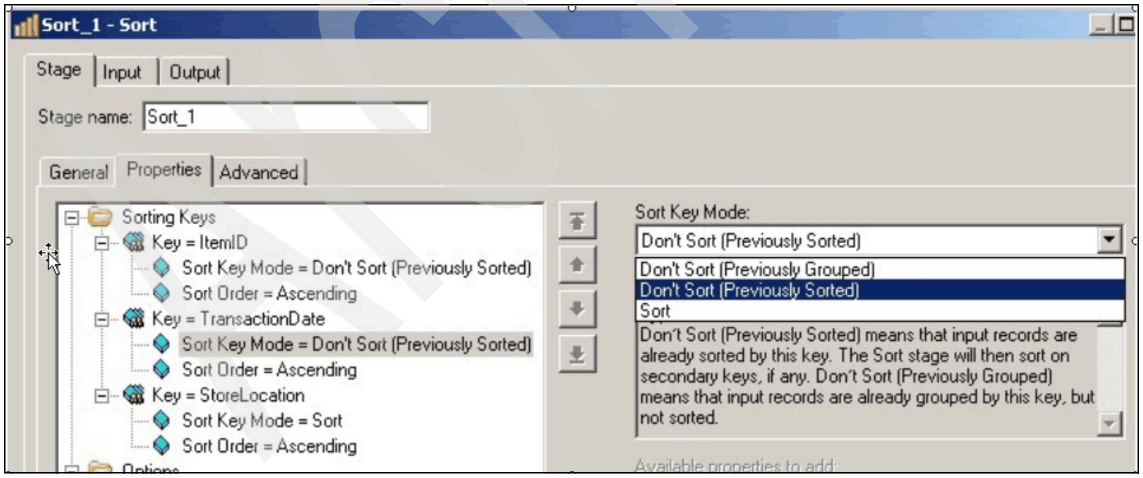
Stable sorts preserve the order of non-key columns in each sort group. This requires additional overhead in the sort algorithm, and thus a stable sort is generally slower than a non-stable sort for the same input dataset and sort keys. For this reason, disable Stable sort unless needed.

By default, the Stable sort option is disabled for sorts on a link and enabled with the standalone Sort stage.

7.5 Subsorts

In the standalone Sort stage, the key column property, Sort Key Mode, is a particularly powerful feature and a significant performance optimizer. It is used when resorting a sub-grouping of a previously sorted input dataset, instead of performing a complete sort. This subsort uses significantly less disk space and CPU resource, and can often be performed in memory (depending on the size of the new subsort groups).

To resort based on a sub-group, all key columns must still be defined in the Sort stage. Re-used sort keys are specified with the "Do not Sort (Previously Sorted)" property. New sort keys are specified with the Sort Key Mode property, as shown in Figure 7-4.



Stage name: Sort_1

```
Sorting Keys
  Key = ItemID
    Sort Key Mode = Don't Sort (Previously Sorted)
    Sort Order = Ascending
  Key = TransactionDate
    Sort Key Mode = Don't Sort (Previously Sorted)
    Sort Order = Ascending
  Key = StoreLocation
    Sort Key Mode = Sort
    Sort Order = Ascending
Options
```

Sort Key Mode: Don't Sort (Previously Sorted)

- Don't Sort (Previously Grouped)
- Don't Sort (Previously Sorted)
- Sort

Don't Sort (Previously Sorted) means that input records are already sorted by this key. The Sort stage will then sort on secondary keys, if any. Don't Sort (Previously Grouped) means that input records are already grouped by this key, but not sorted.

Figure 7-4 Sort Key Mode property

To perform a sub-sort, keys with the “Do not Sort (Previously Sorted)” property must be at the top of the list, without gaps between them. The key column order for these keys must match the key columns and order defined in the previously-sorted input dataset.

If the input data does not match the key column definition for a sub-sort, the job aborts.

7.6 Automatically-inserted sorts

By default, the parallel framework inserts sort operators as necessary to ensure correct results. The parallel job score (see Appendix E, “Understanding the parallel job score” on page 401) can be used to identify automatically-inserted sorts, as shown in Figure 7-5.

```
op1[4p] {(parallel inserted tsort operator
      {key={value=LastName}, key={value=FirstName}}(0))
    on nodes (
```

Figure 7-5 Inserted Sort operator

Typically, the parallel framework inserts sorts before any stage that requires matched key values or ordered groupings (Join, Merge, Remove Duplicates, Sort Aggregator). Sorts are only inserted automatically when the flow developer has not explicitly defined an input sort.

Though ensuring correct results, inserted sorts can be a significant performance impact if they are not necessary. There are two ways to prevent the parallel framework from inserting an un-necessary sort:

- Insert an upstream Sort stage on each link, define all sort key columns with the “Do not Sort (Previously Sorted)” Sort Mode key property.
- Set the environment variable APT_SORT_INSERTION_CHECK_ONLY. This verifies sort order but does not perform a sort, aborting the job if data is not in the required sort order.

Revisiting the partitioning examples in 6.4, “Partitioning examples” on page 108, the environment variable $APT_SORT_INSERTION_CHECK_ONLY must be set to prevent the DS parallel framework from inserting unnecessary sorts before the Join stage.

7.7 Sort methodology

Using the rules and behavior outlined in the previous section, the following methodology must be applied when sorting in a parallel data flow:

1. Start with a link sort.
2. Specify only necessary key columns.
3. Do not use Stable Sort unless needed.
4. Use a stand-alone Sort stage instead of a Link sort for options that are not available on a Link sort:
   - Sort Key Mode, Create Cluster Key Change Column, Create Key Change Column, Output Statistics
   - Always specify the DataStage Sort Utility for standalone Sort stages
   - Use the "Do not Sort (Previously Sorted)" Sort Key Mode to resort a sub-group of a previously-sorted input dataset
5. Be aware of automatically-inserted sorts.

   Set $APT_SORT_INSERTION_CHECK_ONLY to verify but do not establish required sort order.
6. Minimize the use of sorts in a job flow.
7. To generate a single, sequential ordered result set use a parallel Sort and a Sort Merge collector

7.8 Tuning sort

Sort is a particularly expensive task which requires CPU, memory, and disk resources.

To perform a sort, rows in the input dataset are read into a memory buffer on each partition. If the sort operation can be performed in memory (as is often the case with a sub-sort) no disk I/O is performed.

By default, each sort uses 20 MB of memory per partition for its memory buffer. This value can be changed for each standalone Sort stage using the Restrict Memory Usage option (the minimum is 1 MB/partition). On a global basis, the APT_TSORT_STRESS_BLOCKSIZE environment variable can be use to specify the size of the memory buffer, in MB, for all sort operators (link and standalone), overriding any per-sort specifications.

If the input dataset cannot fit into the sort memory buffer, results are temporarily spooled to disk in the following order:

1. Scratch disks defined in the current configuration file (APT_CONFIG_FILE) in the sort named disk pool
2. Scratch disks defined in the current configuration file default disk pool
3. The default directory specified by the environment variable TMPDIR
4. The directory */tmp* (on UNIX) or *C:/TMP* (on Windows) if available

The file system configuration and number of scratch disks defined in parallel configuration file can impact the I/O performance of a parallel sort. Having a greater number of scratch disks for each node allows the sort to spread I/O across multiple file systems.

7.8.1 Sorts and variable-length fields

Although it is usually recommended to define the maximum length of variable length fields, there are situations when it is better to leave their lengths unbound.

The parallel framework always allocates the space equivalent to their maximum specified lengths. If most values are much shorter than their maximum lengths, there will be a large amount of unused space being moved around between operators as well as to/from datasets and fixed format files. That happens, for instance, when an address field is defined as "varchar(500)" but most addresses are 30 characters long.

This severely impacts the performance of sort operations: the more unused bytes a record holds, the more unnecessary data is moved to and from scratch space.

In those situations, it is better to leave the length of those fields unspecified, as the records will only allocate the exact space to hold the field values.

This rule must be applied judiciously, but it may result in great performance gains.

8

File Stage usage

DataStage (DS) offers various parallel stages for reading from and writing to files. In this chapter we provide suggestions for when to use a particular stage, and any limitations that are associated with that stage.

A summary of the various stages are provided in Table 8-1.

Table 8-1 File stages

File stage	Suggested Usage	Limitations
Sequential File	Read and write standard files in a single format.	Cannot write to a single file in parallel, performance penalty of conversion, does not support hierarchical data files.
Complex Flat File	Need to read source data in complex (hierarchical) format, such as mainframe sources with COBOL copybook file definitions.	Cannot write in parallel; performance penalty of format conversion.
Data Set	Intermediate storage between DataStage parallel jobs.	Can only be read from and written to by DataStage parallel jobs or *orchadmin* command.

File stage	Suggested Usage	Limitations
File Set	Need to share information with external applications, can write in parallel (generates multiple segment files).	Slightly higher overhead than dataset.
SAS Parallel	Need to share data with an external Parallel SAS application. (Requires SAS connectivity license for DataStage.)	Requires Parallel SAS, can only be read from / written to by DataStage or Parallel SAS.
Lookup File Set	Rare instances where Lookup reference data is required by multiple jobs and is not updated frequently.	Can only be written – contents cannot be read or verified. Can only be used as reference link on a Lookup stage.

No parallel file stage supports update of existing records. Certain stages (parallel dataset) support Append, to add new records to an existing file. But this is not recommended, as it imposes risks for failure recovery.

We provide further information about these File stage types in the remaining sections of this chapter.

8.1 Dataset usage

Parallel datasets are the persistent (on-disk) representation of the in-memory data structures of the parallel framework. As such, datasets store data in partitioned form, using the internal format of the parallel engine. In general, datasets provide maximum performance for reading and writing data from disk, as no overhead is needed to translate data to the internal parallel framework representation.

Data is stored in datasets in *fixed-length format* and variable length fields are padded up to their maximum length. This allows the parallel framework to determine field boundaries quickly without having to scan the entire record looking for field delimiters.

This yields the best performance when most of the fields are of fixed length and unused positions in variable length fields tend to be minimal.

However, when the overall amount of unused space in variable length fields is significant, the dataset advantages tend to be offset by the cost of storing that much space. For nstance, if an address field is defined as "varchar(500)" and most addresses are 30 characters long, there will be a significant amount of unused space across the entire dataset. When dealing with millions or billions of records, this cost is significant.

Under those circumstances, there are two alternatives:

1. Define the environment variable APT_OLD_BOUNDED_LENGTH

   When this environment variable is set varchar columns will only be stored using the actual data length.

2. Use Filesets instead

   Filesets are stored in text format and, if defined as delimited format, only the exact amount of space required to hold he actual values is used.

Datasets can only be read from and written to using a DataStage parallel job. If data is to be read or written by other applications (including DS Server jobs), then a different parallel stage such as a SequentialFile should be adopted instead.

Also, archived datasets can only be restored to DataStage instances that are on the exact same OS platform.

8.2 Sequential File stages (Import and export)

The Sequential File stage can be used to read from or write to one or more flat files of the same format. Unlike the Complex Flat File stage, the Sequential File stage can only read and write data that is in flattened (row/column) format.

8.2.1 Reading from a sequential file in parallel

The ability to read sequential files in parallel in DS parallel jobs depends on the read method and the options specified. These are depicted for sequential and parallel read in Table 8-2.

Table 8-2 Read method options

Sequential File - Options to read Sequentially:
Read Method: Specific Files, only one file specified might be a file or named pipe
Read Method: File Pattern
Sequential File - Options to read in Parallel:
Read Method: Specific Files, only one file specified, **Readers Per Node** option greater than 1 *useful for **SMP** configurations* - ***file may be either fixed or variable-width***
Read Method: Specific Files, more than one file specified, *each file specified within a single Sequential File stage must be of the same format*
Read Method: File Pattern, set environment variable $APT_IMPORT_PATTERN_USES_FILESET
Read Method: Specific Files, **Read From Multiple Nodes** option is set to Yes, *useful for **cluster** and **Grid** configurations* - ***file may only be fixed-width***

When reading in parallel, input row order is not maintained across readers.

8.2.2 Writing to a sequential file in parallel

It is only possible to write in parallel from a Sequential File stage when more than one output file is specified. In these instances, the degree of parallelism of the write corresponds to the number of file names specified.

A better option for writing to a set of sequential files in parallel is to use the FileSet stage. This creates a single header file (in text format) and corresponding data files, in parallel, using the format options specified in the FileSet stage. The FileSet stage writes in parallel.

8.2.3 Separating I/O from column import

If the sequential file input cannot be read in parallel, performance can still be improved by separating the file I/O from the column parsing operation. In a job, we define a single large string column for the non-parallel sequential file read, and pass this to a Column Import stage to parse the file in parallel. The formatting and column properties of the Column Import stage match those of the Sequential File stage. An example is depicted in Figure 8-1.

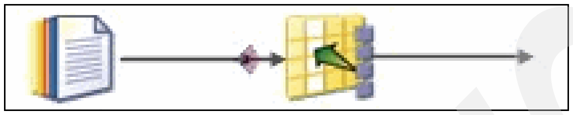


Figure 8-1 Column Import example

This method is also useful for External Source and FTP Sequential Source stages.

8.2.4 Partitioning sequential file reads

Care must be taken to choose the appropriate partitioning method from a sequential file read:

- Do not read from a sequential file using SAME partitioning in the downstream stage. Unless more than one source file is specified, SAME reads the entire file into a single partition, making the entire downstream flow run sequentially (unless it is later repartitioned).
- When multiple files are read by a single Sequential File stage (using multiple files, or by using a file pattern), each file's data is read into a separate partition. It is important to use Round-robin partitioning (or other partitioning appropriate to downstream components) to distribute the data in the flow evenly.

8.2.5 Sequential file (Export) buffering

By default, the Sequential File (export operator) stage buffers its writes to optimize performance. When a job completes successfully, the buffers are flushed to disk. The environment variable $APT_EXPORT_FLUSH_COUNT allows the job developer to specify how frequently (in number of rows) that the Sequential File stage flushes its internal buffer on writes.

Setting this value to a low number (such as 1) is useful for real-time applications, but there is a small performance penalty associated with increased I/O. It is also important to remember that this setting applies to all Sequential File stages in the data flow.

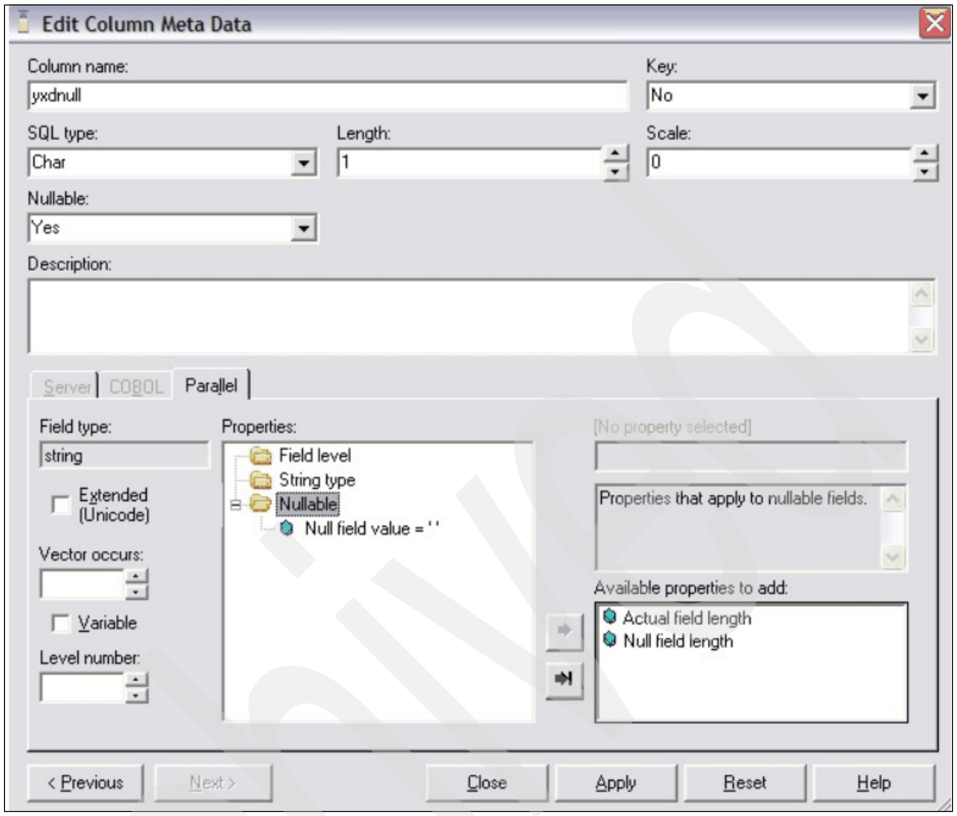
8.2.6 Parameterized sequential file format

The Sequential File stage supports a schema file option to specify the column definitions and file format of the source file. Using the schema file option allows the format of the source file to be specified at runtime, instead of statically through table definitions.

The format of the schema file, including sequential file import/export format properties is documented in the *Orchestrate Record Schema* manual, which is included with the product documentation. This document is required, because the Import/Export properties used by the Sequential File and Column Import stages are not documented in the *DataStage Parallel Job Developers Guide*.

8.2.7 Reading and writing nullable columns

When reading from or writing to sequential files or file sets, the in-band (value) must be explicitly defined in the extended column attributes for each nullable column, as shown in Figure 8-2 on page 133.


Edit Column Meta Data

Column name: yxdnull
Key: No
SQL type: Char
Length: 1
Scale: 0
Nullable: Yes
Description:

Server | COBOL | Parallel

Field type: string
Extended (Unicode)
Vector occurs:
Variable
Level number:

Properties:
- Field level
- String type
- Nullable
  - Null field value = ' '

[No property selected]

Properties that apply to nullable fields.

Available properties to add:
- Actual field length
- Null field length

Figure 8-2 Extended column metadata (nullable properties)

By default, DataStage does not allow a zero-length null_field setting for fixed-width columns. Setting the environment variable $APT_IMPEXP_ALLOW_ZERO_LENGTH_FIXED_NULL permits this. This must be used with care, as poorly formatted data causes incorrect results.

8.2.8 Reading from and writing to fixed-length files

Particular attention must be taken when processing fixed-length fields using the Sequential File stage:

- If the incoming columns are variable-length data types (for example, Integer, Decimal, Varchar), the field width column property must be set to match the fixed-width of the input column. Double-click the column number in the grid dialog box to set this column property.
- If a field is nullable, you must define the null field value and length in the nullable section of the column property. Double-click the column number in

the grid dialog box or right-click the column and select **Edit Column** to set these properties.

- When writing fixed-length files from variable-length fields (for example, Integer, Decimal, and Varchar), the field width and pad string column properties must be set to match the fixed-width of the output column. Double-click the column number in the grid dialog box to set this column property.
- To display each field value, use the print_field import property. Use caution when specifying this option as it can generate an enormous amount of detail in the job log. All import and export properties are listed in the *Import/Export Properties* chapter of the *Orchestrate Operators Referen*ce, which is included with the product documentation..

8.2.9 Reading bounded-length VARCHAR columns

Care must be taken when reading delimited, bounded-length varchar columns (varchars with the length option set). By default, if the source file has fields with values longer than the maximum varchar length, these extra characters are silently truncated.

The $APT_IMPORT_REJECT_STRING_FIELD_OVERRUNS environment variable directs parallel jobs to reject records with strings longer than their declared maximum column length.

8.2.10 Tuning sequential file performance

On heavily-loaded file servers or certain RAID/SAN array configurations, the $APT_IMPORT_BUFFER_SIZE and $APT_EXPORT_BUFFER_SIZE environment variables can be used to improve I/O performance. These settings specify the size of the read (import) and write (export) buffer size in Kbytes, with a default of 128 (128 K). Increasing this size can improve performance.

Finally, in certain disk array configurations, setting the environment variable $APT_CONSISTENT_BUFFERIO_SIZE to a value equal to the read/write size in bytes can significantly improve performance of Sequential File operations.

8.3 Complex Flat File stage

The Complex Flat File (CFF) stage can be used to read or write one or more files in the same hierarchical format. When used as a source, the stage allows you to read data from one or more complex flat files, including MVS™ datasets with QSAM and VSAM files. A complex flat file can contain one or more GROUPs, REDEFINES, or OCCURS clauses. Complex Flat File source stages execute in parallel mode when they are used to read multiple files. You can configure the stage to execute sequentially if it is only reading one file with a single reader.

Note: The Complex Flat File stage cannot read from sources with OCCURS DEPENDING ON clauses. (This is an error in the DataStage documentation.)

When used as a target, the stage allows you to write data to one or more complex flat files. It does not write to MVS datasets.

8.3.1 CFF stage data type mapping

When you work with mainframe data using the CFF stage, the data types are mapped to internal parallel data types as depicted in Table 8-3.

Table 8-3 Data types

COBOL Type	Description	Size	Internal Type	Internal Options
S9(1-4) COMP/COMP-5	binary, native binary	2 bytes	int16	
S9(5-9) COMP/COMP-5	binary, native binary	4 bytes	int32	
S9(10-18) COMP/COMP-5	binary, native binary	2 bytes	int64	
9(1-4) COMP/COMP-5	binary, native binary	2 bytes	uint16	
9(5-9) COMP/COMP-5	binary, native binary	4 bytes	uint32	
9(10-18) COMP/COMP-5	binary, native binary	8 bytes	uint64	
X(n)	character	n bytes	string(n)	
X(n)	character for filler	n bytes	raw(n)	
X(n)	varchar	n bytes	string(max=n)	
9(x)V9(y)COMP-3	decimal	(x+y)/2+1 bytes	decimal[x+y,y]	packed
S9(x)V9(y)COMP-3	decimal	(x+y)/2+1 bytes	decimal[x+y,y]	packed
9(x)V9(y)	display_numeric	x+y bytes	decimal[x+y,y] or string[x+y]	zoned

S9(x)V9(y)	display_numeric	x+y bytes	decimal[x+y,y] or string[x+y]	zoned, trailing
S9(x)V9(y) SIGN IS TRAILING	display_numeric	x+y bytes	decimal[x+y,y]	zoned, trailing
S9(x)V9(y) SIGN IS LEADING	display_numeric	x+y bytes	decimal[x+y,y]	zoned, leading
S9(x)V9(y) SIGN IS TRAILING SEPARATE	display_numeric	x+y+1 bytes	decimal[x+y,y]	separate, trailing
S9(x)V9(y) SIGN IS LEADING SEPARATE	display_numeric	x+y+1 bytes	decimal[x+y,y]	separate, leading
COMP-1	float	4 bytes	sfloat	
COMP-2	float	8 bytes	dfloat	
N(n) or G(n) DISPLAY-1	graphic_n, graphic_g	n*2 bytes	ustring[n]	
N(n) or G(n) DISPLAY-1	vargraphic_g/n	n*2 bytes	ustring[max=n]	
Group			subrec	

8.4 Filesets

Filesets are a type of hybrid between datasets and sequential files.

Just like a dataset, records are stored in several partition files. Also, there is a descriptor file that lists the paths to those partition files as well as the schema for the records contained in them. However, data is stored in text format as in SequentialFiles, as opposed to datasets that stored column values in internal binary format.

Filesets are the way of writing data to text files in parallel.

They are the best alternative under one of the following two circumstances:

- When the overall amount of unused space in variable length fields is significant.

  For instance, when an address field is defined as "varchar(500)" but most records have addresses 30 characters long.

  If the Fileset is defined as delimited format, varchar columns will only be stored using the actual data length, yielding significant savings over datasets.

- Data must be archived to be later restored to a DataStage instance on a different OS platform.

  Records are stored in text format, so writing and reading them across platforms is not an issue.

9

Transformation languages

In this chapter we discuss the transformation languages available to DS developers. We start by describing what we consider to be the most important one, which is the Parallel Transformer stage. It is by far the most used transformation language in parallel jobs. The other stages also play an important part in parallel design, such as the Modify, Filter, and Switch stages.

9.1 Transformer stage

The DataStage (DS) Parallel Transformer stage generates C++ code that is compiled into a parallel component. For this reason, it is important to minimize the number of Transformers, and to use other stages (such as Copy) when derivations are not needed. See 5.7.4, "Parallel Transformer stages" on page 86 for guidelines on Transformer stage usage.

9.1.1 Transformer NULL handling and reject link

When evaluating expressions for output derivations or link constraints, the Transformer rejects (through the reject link indicated by a dashed line) any row that has a NULL value used in the expression. When rows are rejected by a Transformer, entries are placed in the Director job log.

Always include reject links in a parallel Transformer. This makes it easy to identify reject conditions (by row counts). To create a Transformer reject link in Designer, right-click an output link and choose **Convert to Reject**, as in Figure 9-1.

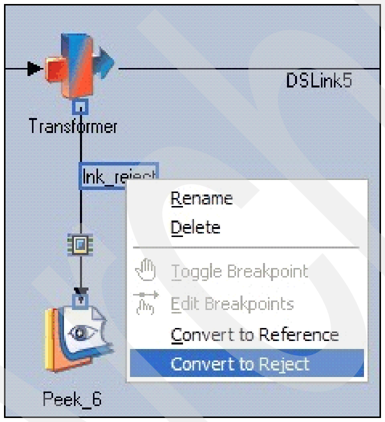


Figure 9-1 Transformer reject link

The parallel Transformer rejects NULL derivation results (including output link constraints) because the rules for arithmetic and string handling of NULL values are, by definition, undefined. Even if the target column in an output derivation allows nullable results, the Transformer rejects the row instead of sending it to the output links.

For this reason, if you intend to use a nullable column in a Transformer derivation or output link constraint, it must be converted from its out-of-band (internal) null representation to an in-band (specific value) null representation using stage variables or the Modify stage.

For example, the following stage variable expression would convert a null value to a specific empty string:

```
If ISNULL(link.col) Then “” Else link.col
```

If an incoming column is only used in an output column mapping, the Transformer allows this row to be sent to the output link without being rejected.

9.1.2 Parallel Transformer system variables

The system variable @ROWNUM behaves differently in the Parallel Transformer stage than in the Server Transformer. Because the Parallel Transformer runs in parallel, @ROWNUM is assigned to incoming rows for each partition. When generating a sequence of numbers in parallel, or performing parallel derivations, the system variables @NUMPARTITIONS and @PARTITIONNUM must be used.

9.1.3 Transformer derivation evaluation

Output derivations are evaluated before any type conversions on the assignment. For example, the PadString function uses the length of the source type, not the target. Therefore, it is important to make sure the type conversion is done before a row reaches the Transformer.

For example, TrimLeadingTrailing(string) works only if “string” is a VarChar field. Thus, the incoming column must be of VarChar type before it is evaluated in the Transformer.

9.1.4 Conditionally aborting jobs

The Transformer can be used to conditionally abort a job when incoming data matches a specific rule. Create a new output link that handles rows that match the abort rule. In the link constraints dialog box, apply the abort rule to this output link, and set the Abort After Rows count to the number of rows allowed before the job must be aborted (for example, 1).

Because the Transformer aborts the entire job flow immediately, it is possible that valid rows have not yet been flushed from sequential file (export) buffers, or committed to database tables. It is important to set the database commit parameters or adjust the Sequential File buffer settings (see 8.2.5, “Sequential file (Export) buffering” on page 131.

9.1.5 Using environment variable parameters

Job parameters can be used in any derivation expression in the parallel Transformer stage. However, if the job parameter is an Environment Variable (starting with a $ sign in the Job Properties/Parameters list), it cannot be used directly in the parallel Transformer.

To use an environment variable job parameter in a Parallel Transformer, define a new job parameter and assign its default value to the environment variable.

9.1.6 Transformer decimal arithmetic

When decimal data is evaluated by the Transformer stage, there are times when internal decimal variables need to be generated to perform the evaluation. By default, these internal decimal variables have a precision and scale of 38 and 10.

If more precision is required, the APT_DECIMAL_INTERM_PRECISION and APT_DECIMAL_INTERM_SCALE environment variables can be set to the desired range, up to a maximum precision of 255 and scale of 125. By default, internal decimal results are rounded to the nearest applicable value. The APT_DECIMAL_INTERM_ROUND_MODE environment variable can be used to change the rounding behavior using one of the following keywords:

- ceil

  Rounds towards positive infinity. Examples:

  `1.4 ∅ 2.1.6  ∅ -1`

- floor

  Rounds towards positive infinity. Examples:

  `1.6  ∅1.-1.4  ∅ -2`

- round_inf

  Rounds or truncates towards nearest representable value, breaking ties by rounding positive values toward positive infinity and negative values toward negative infinity. Examples:

  `1.4  ∅ 1, 1.5  ∅ 2, -1.4  ∅ -1, -1.5  ∅ -2`

- trunc_zero

  Discard any fractional digits to the right of the rightmost fractional digit supported regardless of sign. For example, if $APT_DECIMAL_INTERM_SCALE is smaller than the results of the internal calculation, round or truncate to the scale size. Examples:

  `1.56  ∅ 1.5, -1.56  ∅ -1.5.`

9.1.7 Optimizing Transformer expressions and stage variables

To write efficient Transformer stage derivations, it is useful to understand what items get evaluated and when. The evaluation sequence is as follows:

Evaluate each stage variable initial value.

For each input row to process:

Evaluate each stage variable derivation value, unless the derivation is empty

For each output link:

– Evaluate the link constraint; if true
  - Evaluate each column derivation value
  - Write the output record
  - Else skip the link

Next output link

Next input row

The stage variables and the columns in a link are evaluated in the order in which they are displayed in the Transformer editor. Similarly, the output links are also evaluated in the order in which they are displayed.

From this sequence, it can be seen that there are certain constructs that are inefficient to include in output column derivations, as they are evaluated once for every output column that uses them. Such constructs are where the same part of an expression is used in multiple column derivations

For example, if multiple columns in output links want to use the same substring of an input column, the following test might appear in a number of output columns derivations:

```
IF (DSLINK1.col[1,3] = “001”) THEN ...
```

In this case, the evaluation of the substring of DSLINK1.col[1,3] is evaluated for each column that uses it.

This can be made more efficient by moving the substring calculation into a stage variable. By doing this, the substring is evaluated once for every input row. In this case, the stage variable definition would as follows:

```
DSLINK1.col1[1,3]
```

Each column derivation would start with:

```
IF (stageVar1 = “001” THEN ...
```

In fact, this example can be improved further by moving the string comparison into the stage variable. The stage variable is as follows:

```
IF (DSLink1.col[1,3] = “001” THEN 1 ELSE 0
```

Each column derivation would start as follows:

```
IF (stageVar1) THEN
```

This reduces both the number of substring functions evaluated and string comparisons made in the Transformer.

Where an expression includes calculated constant values

For example, a column definition might include a function call that returns a constant value, for example:

```
Str(“ “,20)
```

This returns a string of 20 spaces. In this case, the function is evaluated every time the column derivation is evaluated. It is more efficient to calculate the constant value just once for the whole Transformer.

This can be achieved using stage variables. This function can be moved into a stage variable derivation. In this case, the function is still evaluated once for every input row. The solution here is to move the function evaluation into the initial value of a stage variable.

A stage variable can be assigned an initial value from the stage Properties dialog box/Variables tab in the Transformer stage editor. In this case, the variable would have its initial value set as follows:

```
Str(“ “,20)
```

Leave the derivation of the stage variable on the main Transformer page empty. Any expression that previously used this function is changed to use the stage variable instead.

The initial value of the stage variable is evaluated once, before any input rows are processed. Because the derivation expression of the stage variable is empty, it is not re-evaluated for each input row. Therefore, its value for the whole Transformer processing is unchanged from the initial value.

In addition to a function value returning a constant value, another example is part of an expression, as follows:

```
"abc" : "def"
```

As with the function-call example, this concatenation is evaluated every time the column derivation is evaluated. Because the subpart of the expression is constant, this constant part of the expression can be moved into a stage variable, using the initial value setting to perform the concatenation just once.

Where an expression requiring a type conversion is used as a constant, or it is used in multiple places.
For example, an expression might include something such as this:

```
DSLink1.col1+"1"
```

In this case, the 1 is a string constant. To add it to DSLink1.col1, it must be converted from a string to an integer each time the expression is evaluated. The solution in this case is to change the constant from a string to an integer:

```
DSLink1.col1+1
```

In this example, if DSLINK1.col1 were a string field, a conversion is required every time the expression is evaluated. If this only appeared once in one output column expression, this is fine. However, if an input column is used in more than one expression, where it requires the same type conversion in each expression, it is more efficient to use a stage variable to perform the conversion once. In this case, you would create, for example, an integer stage variable, specify its derivation to be DSLINK1.col1, and use the stage variable in place of DSLink1.col1, where that conversion would have been required.

When using stage variables to evaluate parts of expressions, the data type of the stage variable must be set correctly for that context. Otherwise, needless conversions are required wherever that variable is used.

9.2 Modify stage

The Modify stage is the most efficient stage available, because it uses low-level functionality that is part of every DataStage parallel component.

As noted in the previous section, the Output Mapping properties for any parallel stage generate an underlying modify for default data type conversions, dropping and renaming columns.

The standalone Modify stage can be used for non-default type conversions (nearly all date and time conversions are non-default), null conversion, and string trim. The Modify stage uses the syntax of the underlying modify operator, documented in the *Parallel Job Developers Guide*, LC18-9892, as well as the *Orchestrate Operators Reference.*

9.2.1 Modify and null handling

The Modify stage can be used to convert an out-of-band null value to an in-band null representation and vice-versa.

Note: The DataStage *Parallel Job Developers Guide* gives incorrect syntax for converting an out-of-band null to an in-band null (value) representation.

To convert from an out-of-band null to an in-band null (value) representation in Modify, the syntax is as follows:

```
destField[:dataType] = make_null(sourceField,value)
```

The elements of this syntax are defined as follows:

- *destField*: Destination field's name.
- *dataType*: The optional data type; use it if you are also converting types.
- *sourceField*: Source field's name.
- *value*: The value of the source field when it is null.

To convert from an in-band null to an out-of-band null, the syntax is as follows:

```
destField[:dataType] = handle_null (sourceField,value)
```

The elements of this syntax are defined as follows:

- *destField*: Destination field's name.
- *dataType*: The optional data type; use it if you are also converting types.
- *sourceField*: Source field's name
- *value*: The value you want to represent a null in the output.

The destField is converted from an Orchestrate out-of-band null to a value of the field's data type. For a numeric field, the value can be a numeric value. For decimal, string, time, date, and time stamp fields, the value can be a string.

9.2.2 Modify and string trim

The function string_trim has been added to Modify, with the following syntax:

```
stringField=string_trim[character, direction, justify] (string)
```

Use this function to remove the characters used to pad variable-length strings when they are converted to fixed-length strings of greater length. By default, these characters are retained when the fixed-length string is converted back to a variable-length string.

The character argument is the character to remove. By default, this is NULL. The value of the direction and justify arguments can be either begin or end. Direction defaults to end, and justify defaults to begin. Justify has no affect when the target string has variable length.

The following example removes all leading ASCII NULL characters from the beginning of name and places the remaining characters in an output variable-length string with the same name:

```
name:string = string_trim[NULL, begin](name)
```

The following example removes all trailing Z characters from color, and left-justifies the resulting hue fixed-length string:

```
hue:string[10] = string_trim['Z', end, begin](color)
```

9.3 Filter and Switch stages

The Filter and Switch stages evaluate their expressions at runtime for every input row. Because it is compiled, a parallel Transformer with output link constraints is faster than a Filter or Switch.

Use of Filter and Switch stages must be limited to instances where the entire filter or switch expression must be parameterized at runtime. In a Parallel Transformer, link constraint expressions, but not data, is fixed by the developer.

10

Combining data

In this chapter we discuss stages related to the combination of records. Records can be combined both horizontally and vertically. For horizontal combination (that is, fields values from multiple records from various input links, related through a common key, are combined into a single record image) there are three standard stage types available, Lookup, Join, and Merge. The vertical combination, on the other hand, is represented by the Aggregator stage. Here, multiple records from the same input link are combined and aggregated values are appended to the output record definition.

10.1 Lookup versus Join versus Merge

The Lookup stage is most appropriate when the reference data for all Lookup stage s in a job is small enough to fit into available physical memory. Each lookup reference requires a contiguous block of shared memory. If the datasets are larger than available memory resources, the JOIN or MERGE stage must be used.

Limit the use of database Sparse Lookups (available in the DB2 Enterprise, Oracle Enterprise, and ODBC Enterprise stages) to scenarios where the number of input rows is significantly smaller (for example, 1:100 or more) than the number of reference rows. (see 13.1.7, "Database sparse lookup versus join" on page 197).

Sparse Lookups might also be appropriate for exception-based processing when the number of exceptions is a small fraction of the main input data. It is best to test both the Sparse and Normal to see which actually performs best, and to retest if the relative volumes of data change dramatically.

10.2 Capturing unmatched records from a Join

The Join stage does not provide reject handling for unmatched records (such as in an InnerJoin scenario). If unmatched rows must be captured or logged, an OUTER join operation must be performed. In an OUTER join scenario, all rows on an outer link (for example, Left Outer, Right Outer, or both links in the case of Full Outer) are output regardless of match on key values.

During an Outer Join, when a match does not occur, the Join stage inserts values into the unmatched non-key columns using the following rules:

- If the non-key column is defined as nullable (on the Join input links), the DS parallel framework inserts NULL values in the unmatched columns
- If the non-key column is defined as not-nullable, the parallel framework inserts default values based on the data type. For example, the default value for an Integer is zero, the default value for a Varchar is an empty string (""), and the default value for a Char is a string of padchar characters equal to the length of the Char column.

For this reason, care must be taken to change the column properties to allow NULL values before the Join. This is done by inserting a Copy stage and mapping a column from NON-NULLABLE to NULLABLE.

A Transformer stage can be used to test for NULL values in unmatched columns.

In most cases, it is best to use a Column Generator to add an indicator column, with a constant value, to each of the inner links and test that column for the constant after you have performed the join. This isolates your match/no-match logic from any changes in the metadata. This is also handy with Lookups that have multiple reference links.

10.3 The Aggregator stage

In this section we discuss and describe the Aggregator stage.

10.3.1 Aggregation method

By default, the default Aggregation method is set to Hash, which maintains the results of each key-column value/aggregation pair in memory. Because each key value/aggregation requires approximately 2 K of memory, the Hash Aggregator can only be used when the number of distinct key values is small and finite.

The Sort Aggregation method must be used when the number of key values is unknown or large. Unlike the Hash Aggregator, the Sort Aggregator requires presorted data, but only maintains the calculations for the current group in memory.

10.3.2 Aggregation data type

By default, the output data type of a parallel Aggregator stage calculation or recalculation column is floating point (Double). To aggregate in decimal precision, set the “Aggregations/Default to Decimal Output” optional property in the Aggregator stage.

You can also specify that the result of an individual calculation or recalculation is decimal by using the optional Decimal Output sub-property.

Performance is typically better if you let calculations occur in floating point (Double) data type and convert the results to decimal downstream in the flow. An exception to this is financial calculations, which must be done in decimal to preserve appropriate precision.

10.3.3 Performing total aggregations

The Aggregator counts and calculates based on distinct key value groupings. To perform a total aggregation, use the stages shown in Figure 10-1:

1. Generate a single constant-value key column using the Column Generator or an upstream Transformer.
2. Aggregate in parallel on the generated column (partition Round-robin, aggregate on generated key column) there is no need to sort or hash-partition the input data with only one key column value.
3. Aggregate sequentially on the generated column.

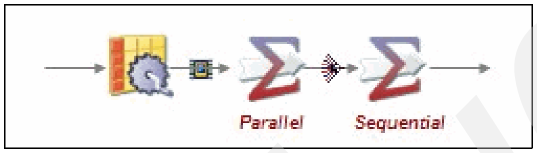


Figure 10-1 Aggregate stage example

In Figure 10-1 two Aggregators are used to prevent the sequential aggregation from disrupting upstream processing. Therefore, they came into existence long before DataStage (DS) itself. Those stages perform field-by-field comparisons on two pre-sorted input datasets.

In 10.5, “Checksum” on page 155, we discuss the use of the Checksum stage, which implements the MD5 Checksum algorithm. Although originally developed for quite different purposes, its common use in the DataStage world tends to be as a more efficient record comparison method.

The Slowly Changing Dimension (SCD) stage is included in this chapter as well. It packs a considerable amount of features specifically tailored to the SCD problem. This stage helps identify, for instance, if new records must be created or updated.

10.4 Comparison stages

There are a number of stages that can be used to identify changes in records, as part of incremental data warehouse loads:

- Change Capture
- Difference
- Compare

These stages are described in detail in the *DataStage Parallel Job Developer Guide*, LC18-9892.

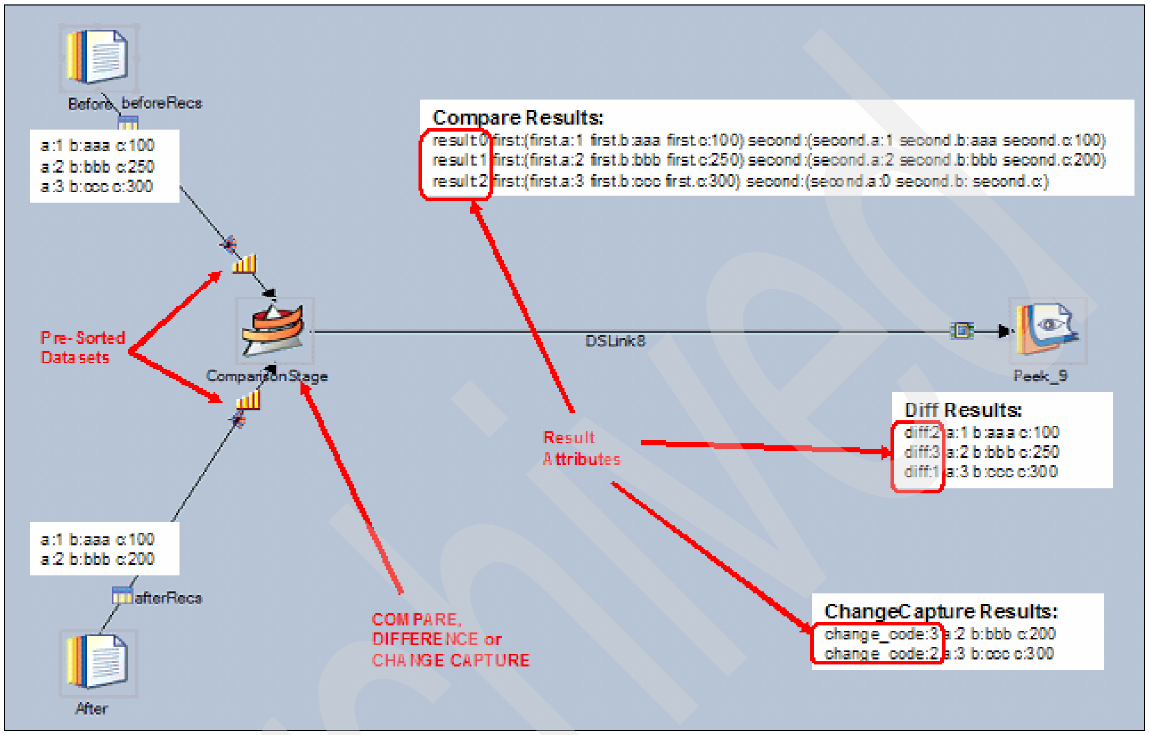
The input datasets are expected to be pre-sorted by a common key.

Comparisons are done on a column-by-column basis. The net effect in terms of performance is a higher processing overhead as the number of attributes to be compared and the size of the individual columns increase.

There are differences between the workings of each stage, so one type might be better suited than the others depending on the task at hand. For instance, the ChangeCapture stage is supposed to be used in conjunction with the ChangeApply. The ChangeCapture produces a set of records containing what needs to be applied by ChangeApply to the before records to produce the after records.

The Difference stage is similar to ChangeCapture. It is seldom used and projects tend to resort to ChangeCapture instead, as this one is closely associated with the ChangeApply stage.

In Figure 10-2 we present an example of comparison of the results from these three stages. The icon for the ComparisonStage is of ChangeCapture, but any of these three stages can be used in its place, yielding different results (depending on the specific business scenario).



```
a:1 b:aaa c:100
a:2 b:bbb c:250
a:3 b:ccc c:300
```

```
a:1 b:aaa c:100
a:2 b:bbb c:200
```

Compare Results:

```
result:0 first:(first.a:1 first.b:aaa first.c:100) second:(second.a:1 second.b:aaa second.c:100)
result:1 first:(first.a:2 first.b:bbb first.c:250) second:(second.a:2 second.b:bbb second.c:200)
result:2 first:(first.a:3 first.b:ccc first.c:300) second:(second.a:0 second.b: second.c:)
```

Diff Results:

```
diff:2 a:1 b:aaa c:100
diff:3 a:2 b:bbb c:250
diff:1 a:3 b:ccc c:300
```

ChangeCapture Results:

```
change_code:3 a:2 b:bbb c:200
change_code:2 a:3 b:ccc c:300
```

Figure 10-2 Column-by-column comparison stages

This example reads data from sequential files. The data can be stored in a database, a persistent dataset or a fileset. Note that we are not discussing the various techniques and optimizations for the extraction of data for the before dataset. This is left for subsequent chapters of this book.

The before dataset is typically extracted from a data mart or a data warehouse. The after represents the set of input records for a given processing cycle.

One could implement similar logic in various ways using other standard components, such as a normal lookup followed by a Transformer. In this scenario, the datasets do not have to be pre-sorted. The comparison of individual columns is implemented by expressions inside the Transformer.

An important disadvantage of the normal lookups is that they are limited to the amount of memory available to hold the reference dataset. Stages that require pre-sorted datasets provide for a much better scalability, as they are not constrained by the amount of available memory.

The point here is there are several ways of implementing the same thing in DataStage. However, there are components that are suited for specific tasks. In the case of comparing records, the Compare, Change Capture, and Difference stages are specific to the problem of identifying differences between records. They reduce the coding effort and clutter.

10.5 Checksum

The Checksum stage is a new component in Information Server8.1 that generates an MD5 Checksum value for a set of columns in a give dataset.

The RSA MD5 algorithm was originally developed and historically used for encryption and integrity checking. However, in the realm of DataStage applications for business intelligence and data warehousing, the most common use tends to be record comparison for incremental loads.

In Figure 10-3 on page 156 we present an example of the Checksum stage for record comparison. The flow assumes the Before dataset already contains a pre-calculated checksum, most likely stored in a target DW table. A new checksum value is calculated for incoming records (After records). The checksum must be calculated on the exact same set of fields for both datasets.

Records from the Before and After datasets are correlated with a Left Outer Join (the left side being the After dataset). The subsequent Transformer contains the logic to compare the checksums and route records to appropriate datasets.

There are a few aspects to consider:

- Checksum values must be calculated for incoming records.
- Checksum values must be pre-calculated and possibly pre-stored for the before dataset in a source dataset or database.
- Other applications that update the same target tables must use the same algorithm as the Checksum stage. That is, the RSA MD5 algorithm.
- The checksum values are specific to a set of attributes taken as input to the MD5 algorithm. If the set of attributes relevant to the checksum calculation changes, all pre-calculated checksum values must be re-calculated.

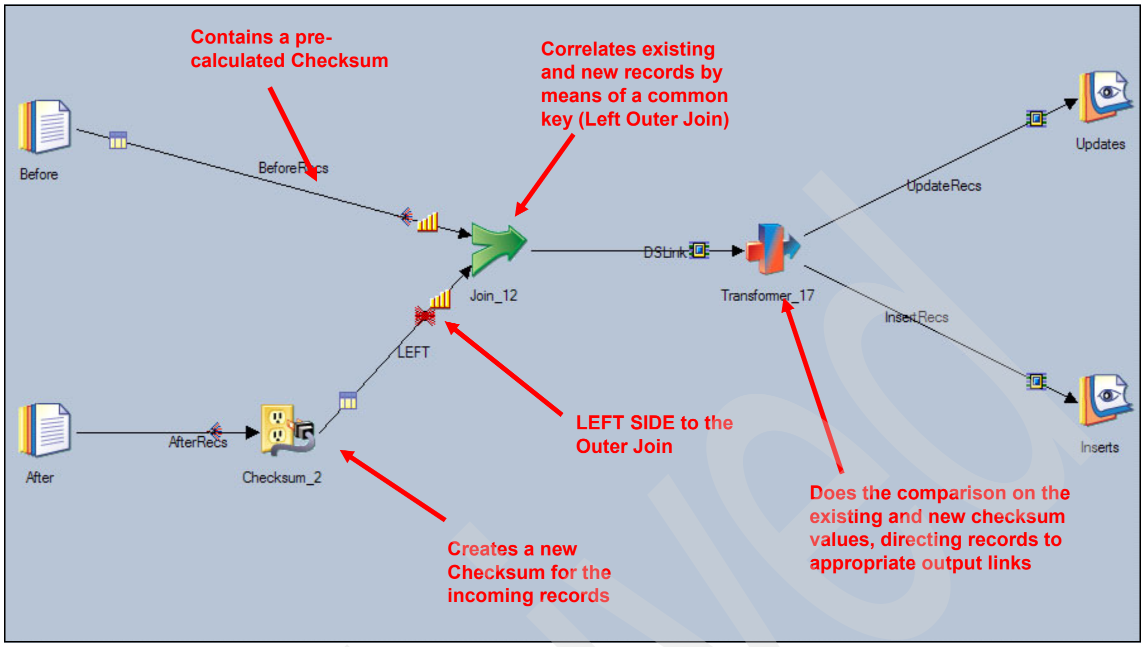

Contains a pre-calculated Checksum
Correlates existing and new records by means of a common key (Left Outer Join)
Before
BeforeRecs
Join_12
DSLink
Transformer_17
UpdateRecs
Updates
InsertRecs
Inserts
LEFT
After
AfterRecs
Checksum_2
LEFT SIDE to the Outer Join
Creates a new Checksum for the incoming records
Does the comparison on the existing and new checksum values, directing records to appropriate output links

Figure 10-3 Using the Checksum stage to identify dataset differences

The Checksum-based approach tends to be more efficient than the column-by-column comparison.

The approach to use depends on how frequently the set of relevant attributes changes and number and size of columns involved in the comparison.

10.6 SCD stage

The SCD stage is another stage that in our opinion falls into the category of identifying changes.

It actually contains more capabilities than to compare records. It has features that, as the name says, are specifically tailored to the processing of slowly changing dimensions:

- Ability to process SCDs type 1 and 2
- Supports special fields:
  - Business Keys
  - Surrogate Keys
  - Current Indicators
  - Effective and Expiration Date fields

- There are two input links (one for input data, one for reference [dimension] records) and two output links (one for fact records and another for updates to the dimension table).

The functionality of the SCD stage can be implemented by using other DataStage stages, such as Lookups, Transformers, and Surrogate Key Generators. However, the SCD stage combines features into a single component. The net effect from a job design perspective is less clutter and ease of development.

You can find a detailed description of the SCD stage in the *InfoSphere DataStage 8.1 Parallel Job Developer Guide,* LC18-9892. The SCD stage is depicted in Figure 10-4.

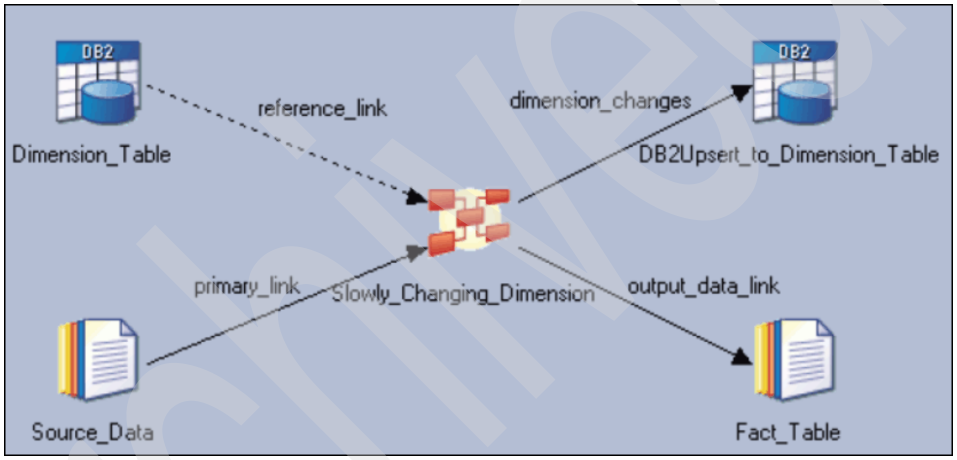


Figure 10-4 The SCD stage

The input stream link contains records from which data for both Fact and Dimension tables are derived. There is one output link for fact records and a second output link for updates to the dimension table.

The entire set of reference records is read from the reference link and stored in memory in the form of a lookup table. This is done at initialization. The reference data remains in memory throughout the duration of the entire job.

This reference table cannot be refreshed dynamically. As a result, the SCD cannot be used in real-time jobs. Because the reference data is kept entirely in memory, it is important to restrict the set of reference records for a given job run.

This release of the Data Flow Standard Practices proposes techniques for batch incremental loads that include a discussion on how to restrict reference datasets (See Chapter 15, “Batch data flow design” on page 259).

That technique consists of the following elements:

- Loading a set of unique source system keys into a target temp table.
- A subsequent job extracts to a dataset the result of a database join between the temp and the target table:
  - This extracted dataset containing the result of the join serves as the reference data for Lookups, Joins, ChangeCapture or, in this case, the SCD stage
  - The Database Join reduces significantly the set of reference records, to only what is relevant for a given run.

There is a chance that this technique will not make the reference dataset small enough to fit in physical memory. That depends on how much memory is available and how large are the data volumes involved.

Avoid overflowing the physical memory when running parallel jobs. This must be avoided for any type of application, not only DataStage jobs. There must be enough room for all processes (OS, DataStage jobs and other applications) and user data such lookup tables to fit comfortably in physical memory.

The moment the physical memory fills up, the operating system starts paging in and out to swap space, which is bad for performance.

Even more grave is when the entire swap space is consumed. That is when DataStage starts throwing fatal exceptions related to fork() errors.

As previously stated, the SCD packs in lots of features that make slowly changing dimension easier, so we are not necessarily ruling out its use.

One might consider making use of the SCD stage when the following circumstances are in affect:

- The SCD stage meets the business requirements (that is, the processing of slowly changing dimensions).
- It is guaranteed the reference dataset fits in memory
  - Use the techniques outlined in Chapter 15, “Batch data flow design” on page 259.
    - Avoid extracting and loading the entire contents of a dimension table
    - Make sure only the relevant reference subset is extracted and loaded into memory through the “reference link”.

If there is a possibility the reference data is too large to fit in memory, another technique, sorted joins, must be adopted instead.

11

Restructuring data

In this chapter, assume you are familiar with the most common record format in DataStage (DS), consisting of a set of atomic fields of types such as integers, char/varchar, float, decimal and so forth. That is the record format that most stages understand and are what can be referred to, in relational terms, as the First Normal Form (1NF). However, DataStage can handle record formats that are more complex than 1NF.

One of the original purposes of the underlying DataStage parallel framework was to parallelize COBOL applications. As such, it has been implemented with ways of importing/exporting and representing those COBOL record formats.

Those capabilities are still present in the DataStage today.

11.1 Complex data types

DataStage can represent de-normalized data records, with fields consisting of:

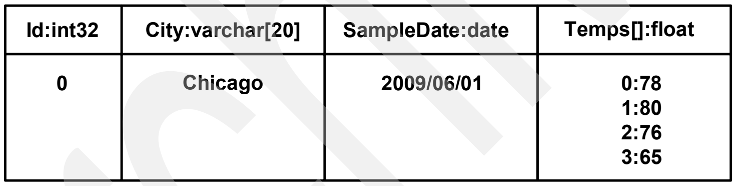
- Subrecords
- Vectors
- Tagged fields (which roughly correspond to C union structures)
- Arbitrary levels of nesting of these types

Complex data types are described in detail in the follow documents:

- *Orchestrate 7.5 Operators Reference*
- *IBM Information Server 8.1 DataStage Parallel Job Developer Guide,* LC18-9892 (Be aware that tagged fields are not described in this document.)

11.1.1 Vectors

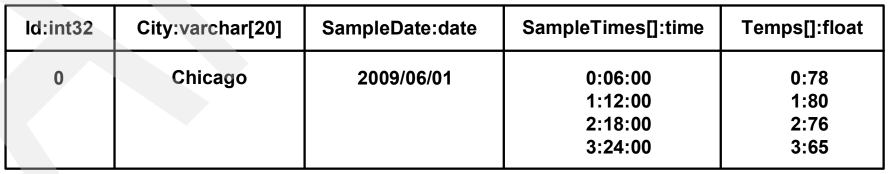
In Figure 11-1 we show an example of a single record, whose Temps field consists of a vector of float values. The notation used here for vector elements consists of a integer indicating the element index, followed by a colon and the actual value.



Id:int32	City:varchar[20]	SampleDate:date	Temps[]:float
0	Chicago	2009/06/01	0:78 1:80 2:76 3:65

Figure 11-1 A record with a vector field

A record might have multiple fields of vector types, as shown in Figure 11-2.



Id:int32	City:varchar[20]	SampleDate:date	SampleTimes[]:time	Temps[]:float
0	Chicago	2009/06/01	0:06:00 1:12:00 2:18:00 3:24:00	0:78 1:80 2:76 3:65

Figure 11-2 A record with two vector fields

There is one additional field for SampleTimes. But, SampleTimes and Temps are paired fields, so perhaps there can be a better representation.

11.1.2 Subrecords

Although valid as is, an alternate schema grouping sample times and corresponding temperatures can be adopted. This can be accomplished by means of a subrecord field type, as depicted in Figure 11-3.

Id:int32	City:varchar[20]	SampleDate:date	TempSamples[]:subrec(SampleTime[]:time;Temp:float)	
			SampleTime	Temp
0	Chicago	2009/06/01	0:06:00	78
			1:12:00	80
			2:18:00	76
			3:24:00	65

Figure 11-3 A record with a vector of subrecords

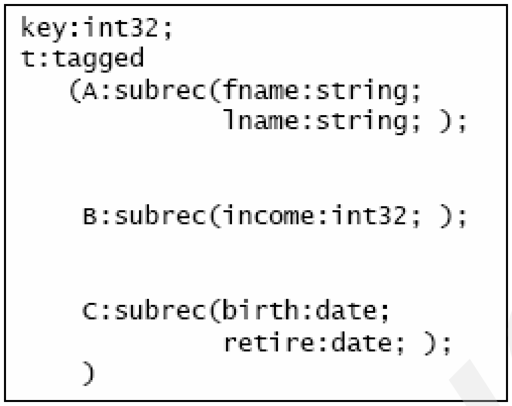
This method of presenting data represents an evolution of the previous two vector scenarios, in which values previously paired are contained in subrecords. The top level field for the vector is named TempSamples. Each subrecord in this vector contains two fields of types time and float.

11.1.3 Tagged fields

The tagged field complex data type allows a given field to assume one of a number of possible subrecord definitions.

One typically uses tagged fields when importing data from a COBOL data file when the COBOL data definition contains a REDEFINES statement. A COBOL REDEFINES statement specifies alternative data types for a single field.

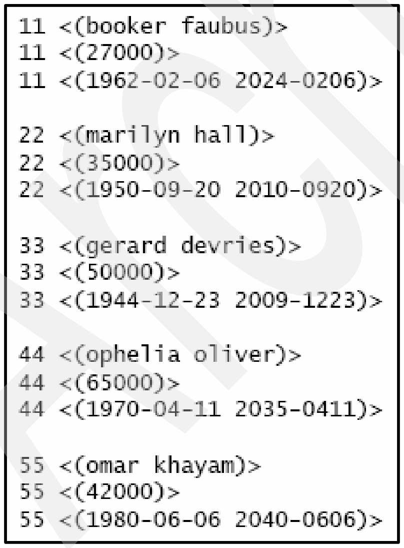
In Figure 11-4 we show an example of a tagged field. The top-level field “t” consists of subrecords, which can assume, in this example, three distinct subrecord types.



```
key:int32;
t:tagged
   (A:subrec(fname:string;
            lname:string; );


    B:subrec(income:int32; );


    C:subrec(birth:date;
            retire:date; );
    )
```

Figure 11-4 A tagged record structure

In Figure 11-5, we show examples of records for the definition in Figure 11-4. Each line represents a separate record. More than one record has the same value for the key field.



```
11 <(booker faubus)>
11 <(27000)>
11 <(1962-02-06 2024-0206)>

22 <(marilyn hall)>
22 <(35000)>
22 <(1950-09-20 2010-0920)>

33 <(gerard devries)>
33 <(50000)>
33 <(1944-12-23 2009-1223)>

44 <(ophelia oliver)>
44 <(65000)>
44 <(1970-04-11 2035-0411)>

55 <(omar khayam)>
55 <(42000)>
55 <(1980-06-06 2040-0606)>
```

Figure 11-5 Sample tagged records

You can create more complex record structures by nesting tagged fields inside vectors of subrecords. Also, tagged fields might be of subrecord or vector type, for example.

Tagged data types are not described by the Information Server documentation. They are described in *Orchestrate 7.5 Operators Reference*. However, tagged types can still be used by means of the Orchestrate record schema definition, as depicted in the previous examples.

11.2 The Restructure library

DataStage supports a number of stages for the conversion of record structures. They allow for normalization and de-normalization of vectors for instance, as well as a combination of distinct fields into vectors and vice-versa.

In Table 11-1 we show a list of DataStage operators and corresponding stage types.

Table 11-1 DataStage operators

Stage type	**Orchestrate operator**
Column Import	Field_import
Column Export	Field_export
Make SubRecords	Makesubrec
Split SubRecord	Splitsubrec
Combine Records	Aggtorec
Promote SubRecord	Promotesubrec
Make Vector	Makevect
Split Vector	Splitvect
N/A	Tagbatch
N/A	Tagswitch

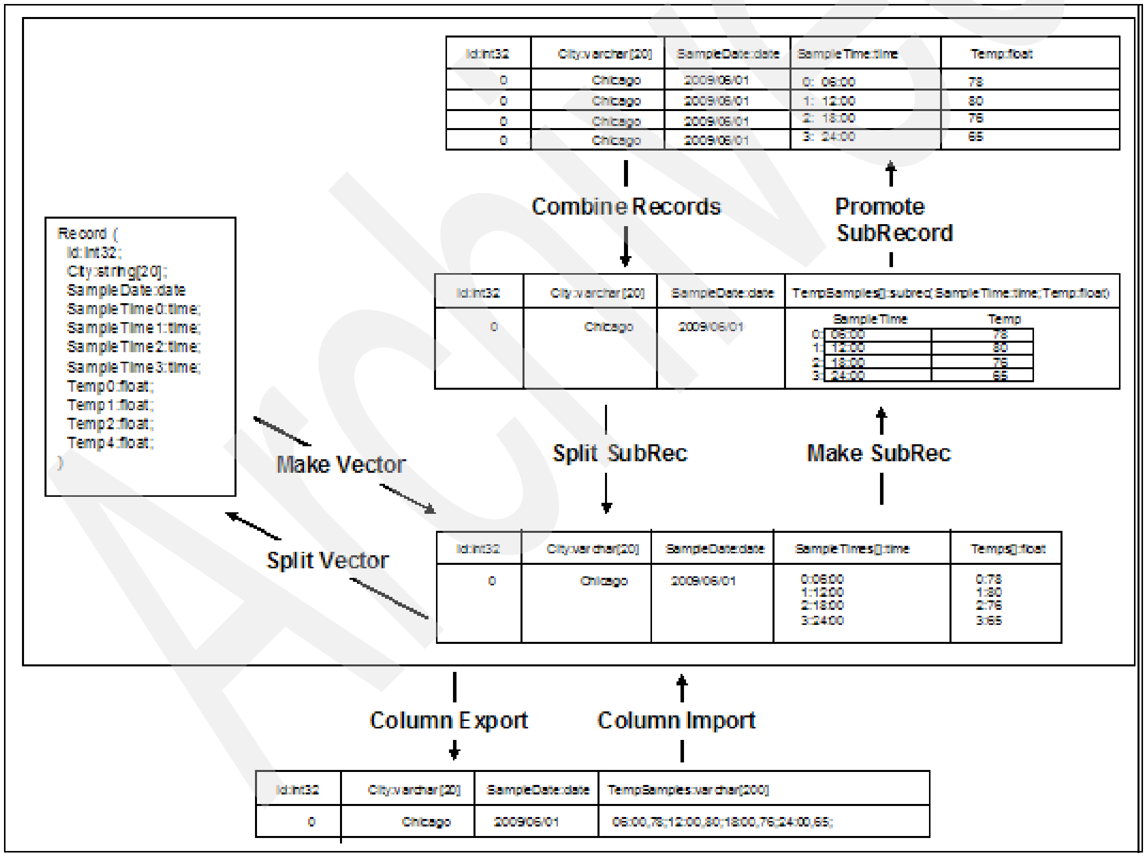
For a full detailed description of those stages and operators, see the follow documents:

- ► *Orchestrate 7.5 Operators Reference*
- ► *IBM Information Server 8.1 DataStage Parallel Job Developer Guide,* LC18-9892

These documents describe the corresponding stage types, with the exception of tagbatch and tagswitch.

In Figure 11-6 we put all Restructure stages and Operators in perspective. You can see the paths that can be followed to convert from one type to another.

The Column Import and Column Export stages can import into and export from any of the record types.



```
Record (
  Id:Int32;
  City:string[20];
  SampleDate:date
  SampleTime0:time;
  SampleTime1:time;
  SampleTime2:time;
  SampleTime3:time;
  Temp0:float;
  Temp1:float;
  Temp2:float;
  Temp4:float;
)
```

Id:Int32	City:varchar[20]	SampleDate:date	SampleTime:time	Temp:float
0	Chicago	2009/06/01	0: 06:00	78
0	Chicago	2009/06/01	1: 12:00	80
0	Chicago	2009/06/01	2: 18:00	76
0	Chicago	2009/06/01	3: 24:00	65

Id:Int32	City:varchar[20]	SampleDate:date	TempSamples[]:subrec(SampleTime:time;Temp:float)	
			SampleTime	Temp
0	Chicago	2009/06/01	0: 06:00	78
			1: 12:00	80
			2: 18:00	76
			3: 24:00	65

Id:Int32	City:varchar[20]	SampleDate:date	SampleTimes[]:time	Temps[]:float
0	Chicago	2009/06/01	0:06:00 1:12:00 2:18:00 3:24:00	0:78 1:80 2:76 3:65

Id:Int32	City:varchar[20]	SampleDate:date	TempSamples:varchar[200]
0	Chicago	2009/06/01	06:00,78;12:00,80;18:00,76;24:00,65;

Figure 11-6 The Restructure operators in perspective

11.2.1 Tagbatch and Tagswitch

The Restructure operators for tagged fields are seldom used and are not exposed as DS stage types, although they are still available in the underlying Orchestrate framework. One might use a generic stage in a DS flow to make use of them.

When using OSH (Orchestrate Shell) scripting, Restructure operators can be referenced as usual.

The tagswitch operator writes each tag case to a separate output link, along with a copy of the top-level fields.

In Figure 11-7, we show the input and output schemas of a tagswitch operator. The input schema contains a top level key field, along with three tag cases. Instead of the nested subrecord fields being flattened out to the same output record format, each tag case is redirected to a separate output link. The top level fields are copied along to the corresponding output link.

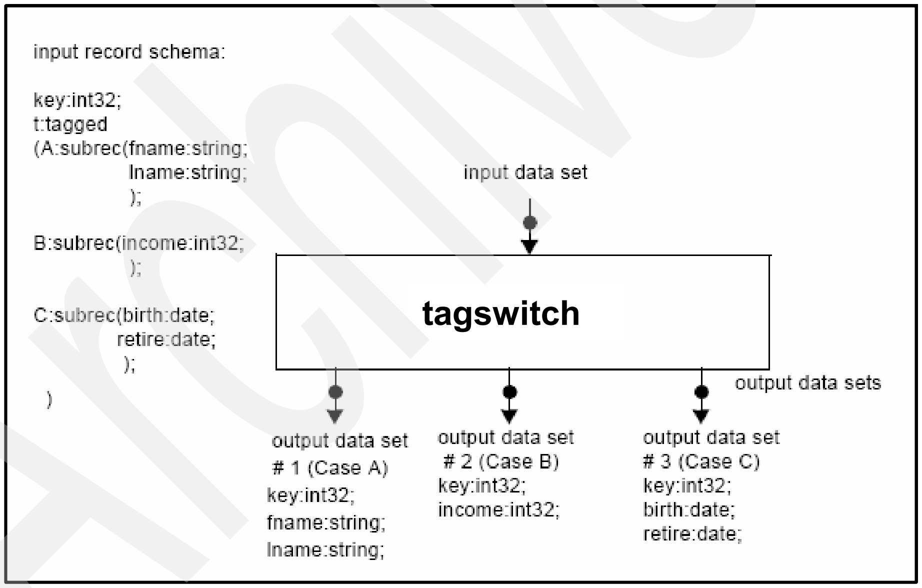
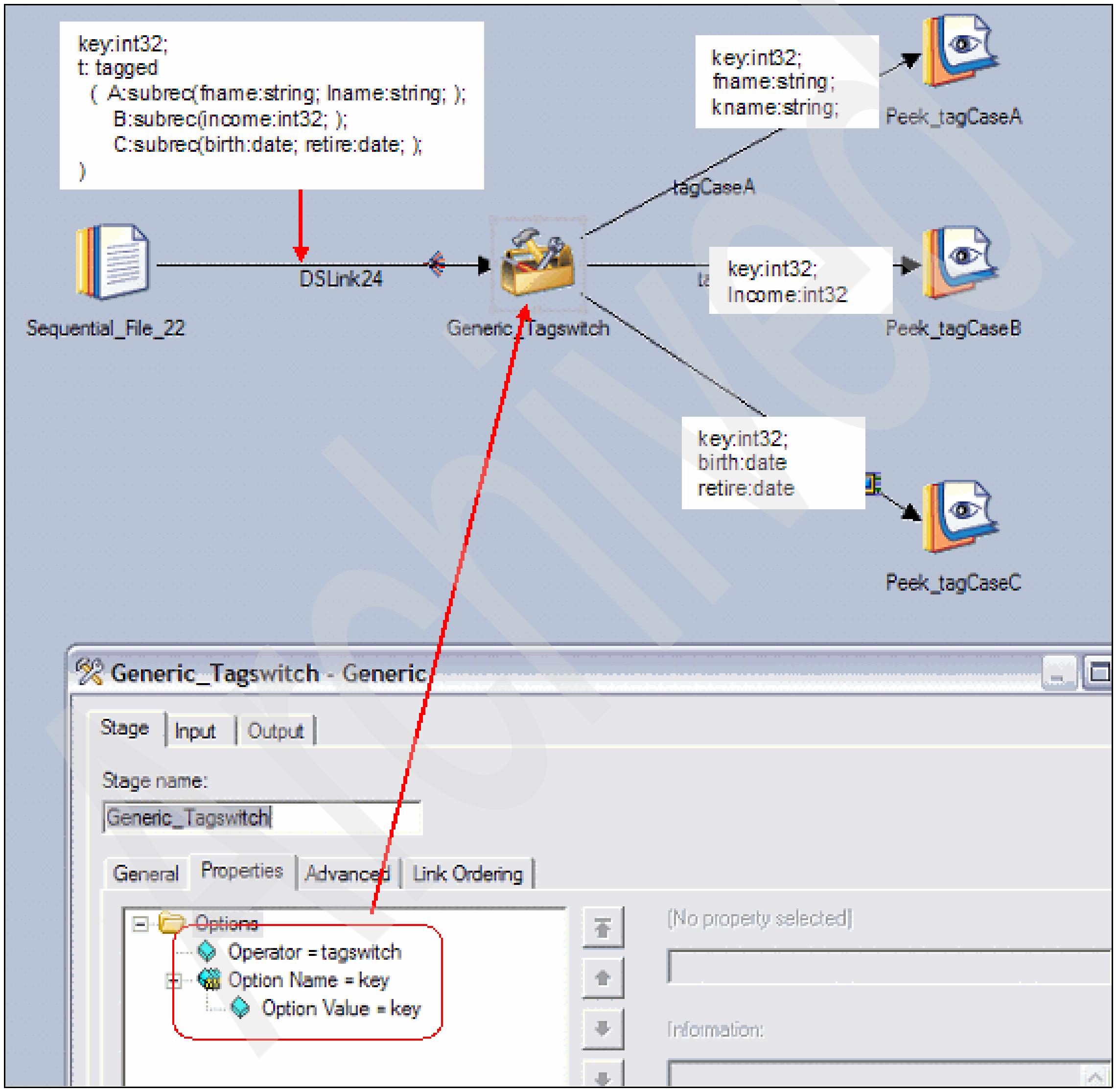


Figure 11-7 Input and output schemas of a tagswitch operator

Figure 11-8 illustrates invoking a tagswitch operator from a DS job. The flow includes a generic stage, which allows one to reference any underlying parallel operator. The stage properties must include the operator name (tagswitch) and the properties as expected by the operator argument syntax.

The record is imported using a complex record schema syntax.



```
key:int32;
t: tagged
 ( A:subrec(fname:string; lname:string; );
   B:subrec(income:int32; );
   C:subrec(birth:date; retire:date; );
 )
```

```
key:int32;
fname:string;
kname:string;
```

```
key:int32;
Income:int32
```

```
key:int32;
birth:date
retire:date
```

Stage name: Generic_Tagswitch

- Options
  - Operator = tagswitch
  - Option Name = key
    - Option Value = key

Figure 11-8 Using the generic stage to Invoke the Tagswitch operator

In Figure 11-9, we present an alternative solution using standard DataStage stages. The input records must be imported with a Type indicator. A Transformer or a Switch stage directs each record case to a separate branch. Each branch has a separate Column Import stage, which does the parsing of each case separately.

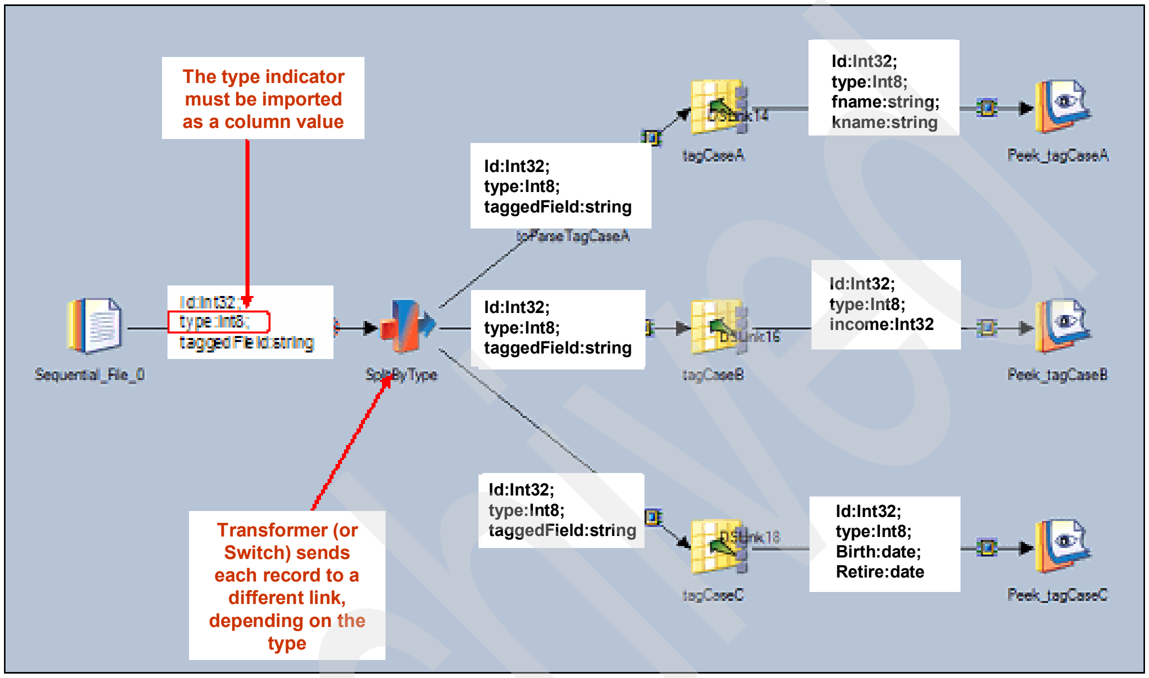
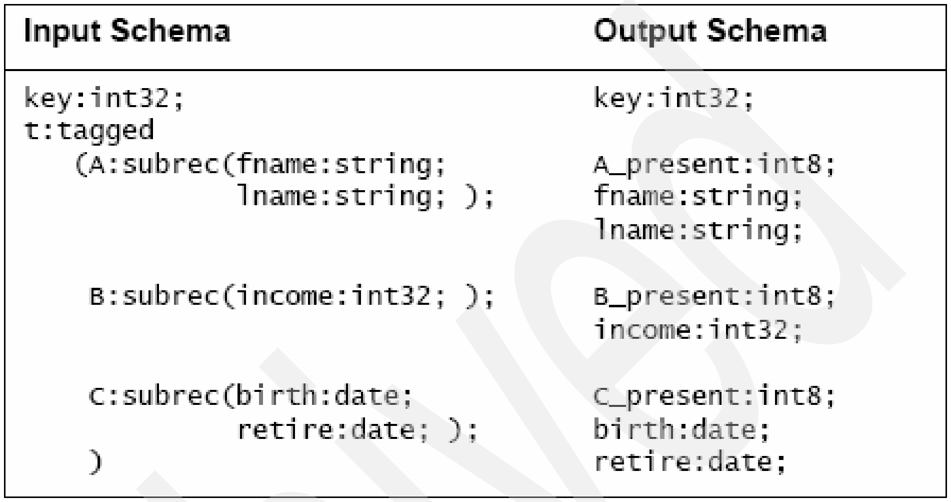


Figure 11-9 The Tagswitch functionality implemented with DS stages

The Tagbatch operator flattens the record definitions of tagged fields so all fields from the tag cases are promoted to top level fields.

In Figure 11-10 we present an example from the *Orchestrate 7.5 Operators Reference*. Each record of the input schema might assume one of three possible formats. When running those records through a Tagbatch operator, the schema is flattened so all fields are promoted to the top level. There are fields that indicate the original type, named "*_present".



Input Schema	Output Schema
`key:int32;`	`key:int32;`
`t:tagged`	
`(A:subrec(fname:string;`	`A_present:int8;`
`lname:string; );`	`fname:string;`
	`lname:string;`
`B:subrec(income:int32; );`	`B_present:int8;`
	`income:int32;`
`C:subrec(birth:date;`	`C_present:int8;`
`retire:date; );`	`birth:date;`
`)`	`retire:date;`

Figure 11-10 Input and output schemas of a Tagbatch operator

The OSH invocation for the example is as follows:

```
osh "... tagbatch -key key ..."
```

By default, multiple records with the same key value are combined into the same output record. Or they might be written out as separate records.

In Figure 11-11 we present a way of implementing the Tagbatch functionality using DS stages. It is similar to the solution for Tagswitch, except that it includes a Left Outer Join to assemble a single output record out of the multiple input tag cases.

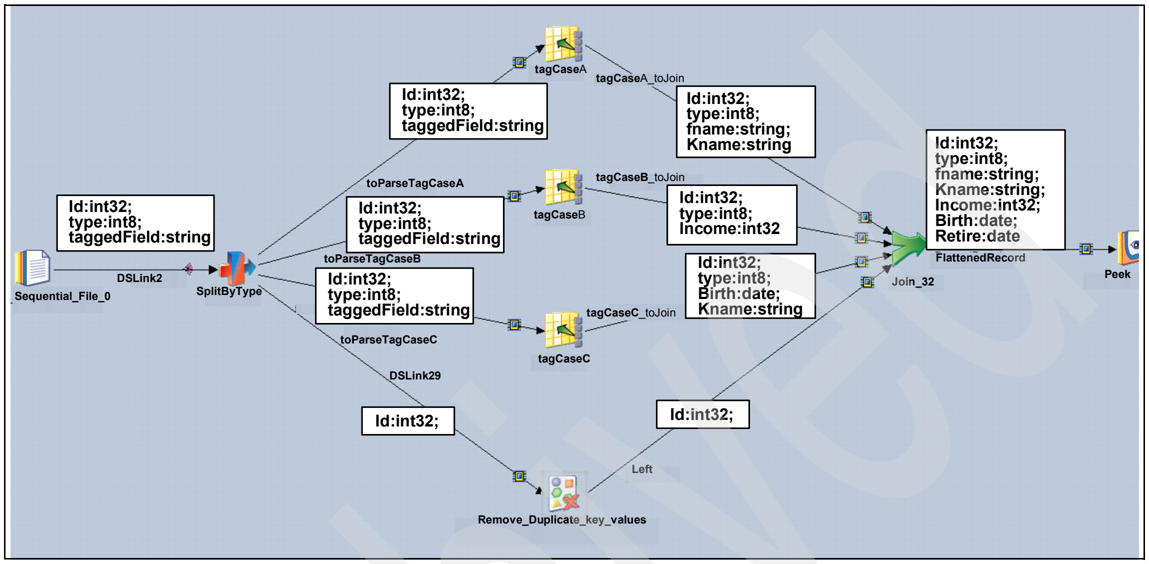

tagCaseA
tagCaseA_toJoin
Id:int32; type:int8; taggedField:string
Id:int32; type:int8; fname:string; Kname:string
Id:int32; type:int8; fname:string; Kname:string; Income:int32; Birth:date; Retire:date
toParseTagCaseA
tagCaseB_toJoin
tagCaseB
Id:int32; type:int8; Income:int32
Id:int32; type:int8; taggedField:string
Id:int32; type:int8; taggedField:string
FlattenedRecord
Peek
toParseTagCaseB
Id:int32; type:int8; Birth:date; Kname:string
Join_32
Sequential_File_0
DSLink2
SplitByType
Id:int32; type:int8; taggedField:string
tagCaseC_toJoin
toParseTagCaseC
tagCaseC
DSLink29
Id:int32;
Id:int32;
Left
Remove_Duplicate_key_values

Figure 11-11 The Tagbatch Functionality Implemented with DS stages

You can see from the examples that the native Tagbatch and Tagswitch operators provide a convenient way of dealing with tagged record structures. Instead of having multiple stages, the flow can be simplified by using components that are specific to the task at hand. This is often the case when dealing with complex COBOL record types.

11.2.2 Importing complex record types

Complex values can be imported into the types described in the previous sections. There are two ways this can be done:

- Sequential File stage
  - The representation of complex record structures in the DataStage table definition tends to be cluttered. Details such as nesting level, number of occurrences, subrecord tagging are harder to define from scratch.
  - This stage allows the use of a record schema file. With record schema files, one can achieve reusable job designs, with a great deal of independence from record types.

  - Record schema files allow the specification of record structures to a degree of complexity beyond what the Complex Flat File stage can support. That is typically the case when dealing with tagged record structures and complex binary feeds.
- Complex Flat File
  - The Complex Flat File exposes the details of complex flat files as stage properties.
  - It does not allow the use of record schema files, thus job designs with this stage is specific to a certain record definition.
  - You can import complex input records to flattened record types. The input records are multiplied depending on the nested fields and number of occurrences.
  - Complex Flat Files can be used in tandem to cascade the parsing of nested fields.

The Sequential File stage can be used in conjunction with the Column Import stage. This might be done for two purposes:

- De-couple the file import process from the parsing of input records: the record parsing can be done in parallel, with a higher degree of parallelism than the one used to read the input file.
- Simplify the parsing by using multiple column import and Transformer stages in tandem.

The full import/export record syntax is described in the DataStage 7.5 Operators Reference (Chapter 25: Import/Export Properties).

11.3 The Pivot Enterprise stage

The Pivot Enterprise stage is used to pivot data horizontally. The Pivot stage maps a set of columns in an input row to a single column in multiple output rows.

You can generate a pivot index that assigns an index number to each row with a set of pivoted data.

This stage was introduced in Information Server 8.1 and complements the set of Restructure stages that were already present in previous releases, as discussed in the previous sections.

In Figure 11-12 we provide examples of data before and after a horizontal pivot operation. These examples were taken from the *IBM Information Server 8.1 DataStage Parallel Job Developer Guide*.

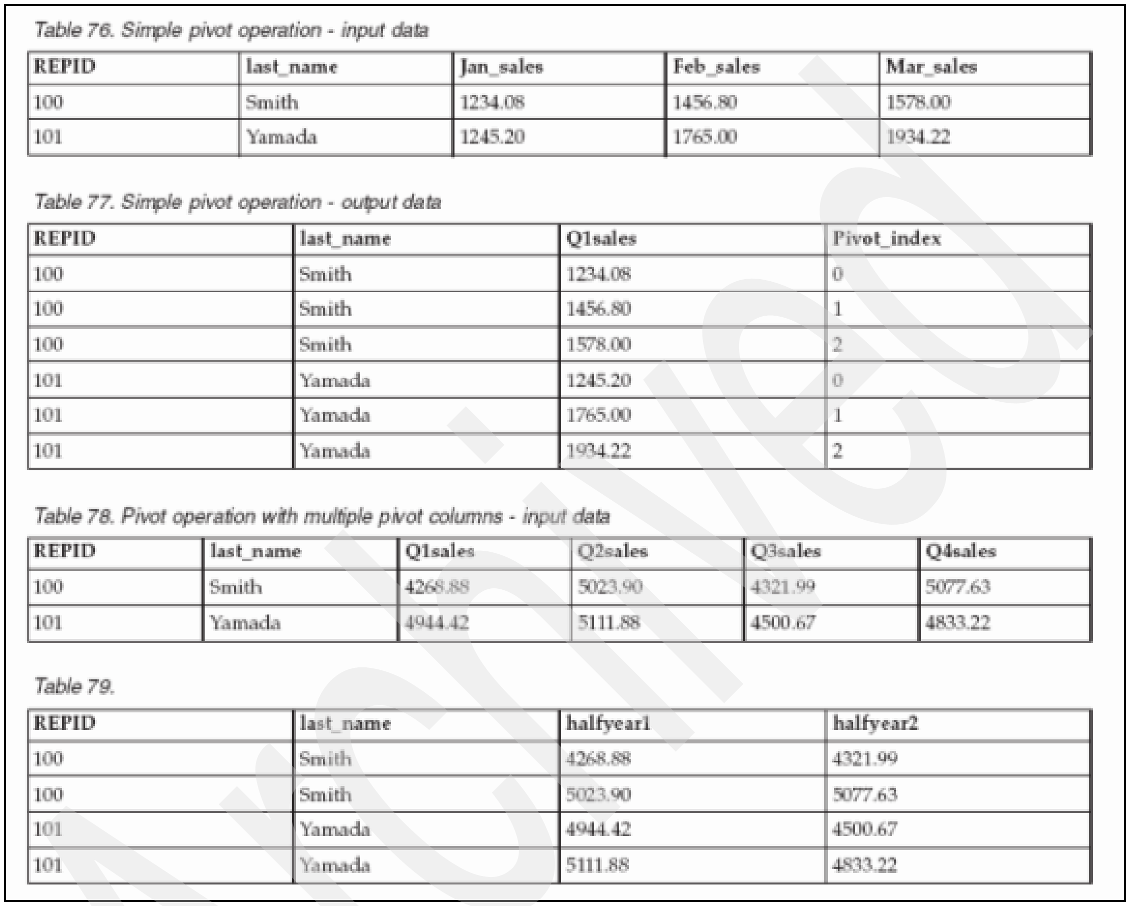


Table 76. Simple pivot operation - input data

REPID	last_name	Jan_sales	Feb_sales	Mar_sales
100	Smith	1234.08	1456.80	1578.00
101	Yamada	1245.20	1765.00	1934.22

Table 77. Simple pivot operation - output data

REPID	last_name	Q1sales	Pivot_index
100	Smith	1234.08	0
100	Smith	1456.80	1
100	Smith	1578.00	2
101	Yamada	1245.20	0
101	Yamada	1765.00	1
101	Yamada	1934.22	2

Table 78. Pivot operation with multiple pivot columns - input data

REPID	last_name	Q1sales	Q2sales	Q3sales	Q4sales
100	Smith	4268.88	5023.90	4321.99	5077.63
101	Yamada	4944.42	5111.88	4500.67	4833.22

Table 79.

REPID	last_name	halfyear1	halfyear2
100	Smith	4268.88	4321.99
100	Smith	5023.90	5077.63
101	Yamada	4944.42	4500.67
101	Yamada	5111.88	4833.22

Figure 11-12 Pivot examples

One could mimic the functionality of the Pivot stage by using a combination of Restructure stages depicted in Figure 11-6 on page 164. To go from the record structure with fields “SampleTime0…SampleTime3” and “Temp0…3” the following sequence of stages can be implemented:

1. MakeVector
2. MakeSubRecord
3. PromoteSubRecord

This must be done twice. First for SampleTimeN, and again for TempN.

The MakeVector stage has the following drawbacks:

- Naming requirements that lead to additional field renaming;
- It does not support the generation of Pivot Index in a straightforward way.

The Pivot stage overcomes those limitations and does the transformation from a single record into multiple records in a single stage. It provides for a more natural and user-friendly restructure mechanism for certain scenarios.

12

Performance tuning job designs

The ability to process large volumes of data in a short period of time depends on all aspects of the flow and environment being optimized for maximum throughput and performance. Performance tuning and optimization is an iterative process that begins at job design and unit tests, proceeds through integration and volume testing, and continues throughout an application's production life cycle.

This section provides tips for designing a job for optimal performance, and for optimizing the performance of a given data flow using various settings and features in DataStage (DS).

12.1 Designing a job for optimal performance

Overall job design can be the most significant factor in data flow performance. This section outlines performance-related tips that can be followed when building a parallel data flow using DataStage.

- Use parallel datasets to land intermediate result between parallel jobs.
  - Parallel datasets retain data partitioning and sort order, in the DS parallel native internal format, facilitating end-to-end parallelism across job boundaries.
  - Datasets can only be read by other DS parallel jobs (or the **`orchadmin`** command line utility). If you need to share information with external applications, File Sets facilitate parallel I/O at the expense of exporting to a specified file format.
- Lookup File Sets can be used to store reference data used in subsequent jobs. They maintain reference data in DS parallel internal format, pre-indexed. However, Lookup File Sets can only be used on reference links to a Lookup stage. There are no utilities for examining data in a Lookup File Set.
- Remove unneeded columns as early as possible in the data flow. Every unused column requires additional memory that can impact performance (it also makes each transfer of a record from one stage to the next more expensive).
  - When reading from database sources, use a select list to read needed columns instead of the entire table (if possible)
  - Be alert when using runtime column propagation (RCP). It might be necessary to disable RCP for a particular stage to ensure that columns are actually removed using that stage's Output Mapping.
- Specify a maximum length for Varchar columns.

  Unbounded strings (Varchars without a maximum length) can have a significant negative performance impact on a job flow.

  There are limited scenarios when the memory overhead of handling large Varchar columns would dictate the use of unbounded strings:
  - Varchar columns of a large (for example, 32 K) maximum length that are rarely populated.
  - Varchar columns of a large maximum length with highly varying data sizes.

- Avoid type conversions, if possible.

  When working with database sources and targets, use **orchdbutil** to ensure that the design-time metadata matches the actual runtime metadata (especially with Oracle databases).

  Enable $OSH_PRINT_SCHEMAS to verify runtime schema matches job design column definitions

  Verify that the data type of defined Transformer stage variables matches the expected result type
- Minimize the number of Transformers. For data type conversions, renaming and removing columns, other stages (for example, Copy, Modify) might be more appropriate. Note that unless dynamic (parameterized) conditions are required, a Transformer is always faster than a Filter or Switch stage.
- Avoid using the BASIC Transformer, especially in large-volume data flows. External user-defined functions can expand the capabilities of the parallel Transformer.
- Use BuildOps only when existing Transformers do not meet performance requirements, or when complex reusable logic is required.

  Because BuildOps are built in C++, there is greater control over the efficiency of code, at the expense of ease of development (and more skilled developer requirements).
- Minimize the number of partitioners in a job. It is usually possible to choose a smaller partition-key set, and to re-sort on a differing set of secondary/tertiary keys.
- When possible, ensure data is as evenly distributed as possible. When business rules dictate otherwise and the data volume is large and sufficiently skewed, re-partition to a more balanced distribution as soon as possible to improve performance of downstream stages.
- Know your data. Choose hash key columns that generate sufficient unique key combinations (when satisfying business requirements).
- Use SAME partitioning carefully. Remember that SAME maintains the degree of parallelism of the upstream operator.

- Minimize and combine use of Sorts where possible.
  - It is frequently possible to arrange the order of business logic in a job flow to make use of the same sort order, partitioning, and groupings.
  - If data has already been partitioned and sorted on a set of key columns, specifying the **Do not sort, previously sorted** option for those key columns in the Sort stage reduces the cost of sorting and take greater advantage of pipeline parallelism.
  - When writing to parallel datasets, sort order and partitioning are preserved. When reading from these datasets, try to maintain this sorting, if possible, by using SAME partitioning.
  - The stable sort option is much more expensive than non-stable sorts, and can only be used if there is a need to maintain an implied (that is, not explicitly stated in the sort keys) row order.
  - Performance of individual sorts can be improved by increasing the memory usage per partition using the **Restrict Memory Usage (MB)** option of the standalone Sort stage. The default setting is 20 MB per partition. In addition, the APT_TSORT_STRESS_BLOCKSIZE environment variable can be used to set (in units of MB) the size of the RAM buffer for *all* sorts in a job, even those that have the **Restrict Memory Usage** option set.

12.2 Understanding operator combination

At runtime, the DataStage parallel framework analyzes a given job design and uses the parallel configuration file to build a job score, which defines the processes and connection topology (datasets) between them used to execute the job logic.

When composing the score, the DS parallel framework attempts to reduce the number of processes by combining the logic from two or more stages (operators) into a single process (per partition). Combined operators are generally adjacent to each other in a data flow.

The purpose behind operator combination is to reduce the overhead associated with an increased process count. If two processes are interdependent (one processes the output of the other) and they are both CPU-bound or I/O-bound, there is nothing to be gained from pipeline partitioning.

Exception: One exception to this guideline is when operator combination generates too few processes to keep the processors busy. In these configurations, disabling operator combination allows CPU activity to be spread across multiple processors instead of being constrained to a single processor. As with any example, test your results in your environment.

However, the assumptions used by the parallel framework optimizer to determine which stages can be combined might not always be the most efficient. It is for this reason that combination can be enabled or disabled on a per-stage basis, or globally.

When deciding which operators to include in a particular combined operator (also referred to as a Combined Operator Controller), the framework includes all operators that meet the following rules:

- Must be contiguous
- Must be the same degree of parallelism
- Must be combinable.

  The following is a partial list of non-combinable operators:

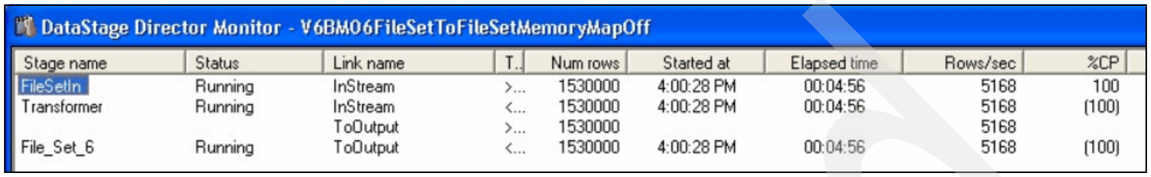
  – Join
  – Aggregator
  – Remove Duplicates
  – Merge
  – BufferOp
  – Funnel
  – DB2 Enterprise stage
  – Oracle Enterprise stage
  – ODBC Enterprise stage
  – BuildOps

In general, it is best to let the framework decide what to combine and what to leave uncombined. However, when other performance tuning measures have been applied and still greater performance is needed, tuning combination might yield additional performance benefits.

There are two ways to affect operator combination:

- The environment variable APT_DISABLE_COMBINATION disables all combination in the entire data flow, this is only recommended on pre-7.0 versions of DS.
- In Designer, combination can be set on a per-stage basis (on the stage/Advanced tab)

The job score identifies what components are combined. (For information about interpreting a job score dump, see Appendix E, “Understanding the parallel job score” on page 401.) In addition, if the %CPU column is displayed in a job monitor window in Director, combined stages are indicated by parenthesis surrounding the %CPU, as shown in Figure 12-1.

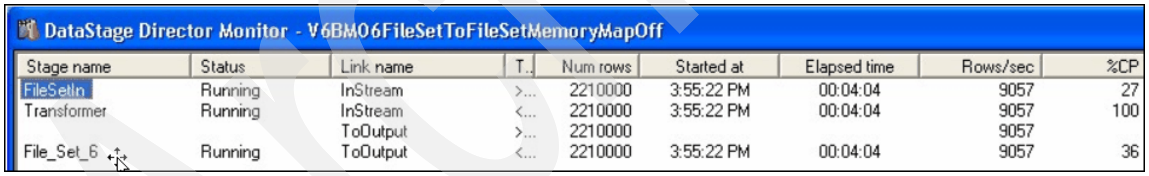

DataStage Director Monitor - V6BM06FileSetToFileSetMemoryMapOff

Stage name	Status	Link name	T..	Num rows	Started at	Elapsed time	Rows/sec	%CP
FileSetIn	Running	InStream	>...	1530000	4:00:28 PM	00:04:56	5168	100
Transformer	Running	InStream	<...	1530000	4:00:28 PM	00:04:56	5168	(100)
		ToOutput	>...	1530000			5168	
File_Set_6	Running	ToOutput	<...	1530000	4:00:28 PM	00:04:56	5168	(100)

Figure 12-1 CPU-bound combined process in job monitor

Choosing which operators to disallow combination for is as much art as science. However, in general, it is good to separate I/O heavy operators (Sequential File, Full Sorts, and so on.) from CPU-heavy operators (Transformer, Change Capture, and so on). For example, if you have several Transformers and database operators combined with an output Sequential File, it might be a good idea to set the sequential file to be non-combinable. This prevents IO requests from waiting on CPU to become available and vice-versa.

In fact, in this job design, the I/O-intensive FileSet is combined with a CPU-intensive Transformer. Disabling combination with the Transformer enables pipeline partitioning, and improves performance, as shown in this subsequent job monitor (Figure 12-2) for the same job.


DataStage Director Monitor - V6BM06FileSetToFileSetMemoryMapOff

Stage name	Status	Link name	T..	Num rows	Started at	Elapsed time	Rows/sec	%CP
FileSetIn	Running	InStream	>...	2210000	3:55:22 PM	00:04:04	9057	27
Transformer	Running	InStream	<...	2210000	3:55:22 PM	00:04:04	9057	100
		ToOutput	>...	2210000			9057	
File_Set_6	Running	ToOutput	<...	2210000	3:55:22 PM	00:04:04	9057	36

Figure 12-2 Throughput in job monitor after disabling combination

12.3 Minimizing runtime processes and resource requirements

The architecture of the parallel framework is well-suited for processing massive volumes of data in parallel across available resources. Toward that end, DataStage executes a given job across the resources defined in the specified configuration file.

There are times, however, when it is appropriate to minimize the resource requirements for a given scenario:

- Batch jobs that process a small volume of data
- Real-time jobs that process data in small message units
- Environments running a large number of jobs simultaneously on the same servers

In these instances, a single-node configuration file is often appropriate to minimize job startup time and resource requirements without significantly impacting overall performance.

There are many factors that can reduce the number of processes generated at runtime:

- Use a single-node configuration file
- Remove all partitioners and collectors (especially when using a single-node configuration file)
- Enable runtime column propagation on Copy stages with only one input and one output
- Minimize join structures (any stage with more than one input, such as Join, Lookup, Merge, Funnel)
- Minimize non-combinable stages (as outlined in the previous section) such as Join, Aggregator, Remove Duplicates, Merge, Funnel, DB2 Enterprise, Oracle Enterprise, ODBC Enterprise, BuildOps, BufferOp
- Selectively (being careful to avoid deadlocks) disable buffering. (Buffering is discussed in more detail in section 12.4, “Understanding buffering” on page 180.)

12.4 Understanding buffering

There are two types of buffering in the DS parallel framework:

- Inter-operator transport
- Deadlock prevention

12.4.1 Inter-operator transport buffering

Though it might appear so from the performance statistics, and documentation might discuss record streaming, records do not stream from one stage to another. They are actually transferred in blocks (as with magnetic tapes) called *Transport Blocks*. Each pair of operators that have a producer/consumer relationship share at least two of these blocks.

The first block is used by the upstream/producer stage to output data it is done with. The second block is used by the downstream/consumer stage to obtain data that is ready for the next processing step. After the upstream block is full and the downstream block is empty, the blocks are swapped and the process begins again.

This type of buffering (or record blocking) is rarely tuned. It usually only comes into play when the size of a single record exceeds the default size of the transport block. Setting APT_DEFAULT_TRANSPORT_BLOCK_SIZE to a multiple of (or equal to) the record size resolves the problem. Remember, there are two of these transport blocks for each partition of each link, so setting this value too high can result in a large amount of memory consumption.

The behavior of these transport blocks is determined by the following environment variables:

- APT_DEFAULT_TRANSPORT_BLOCK_SIZE

  This variable specifies the default block size for transferring data between players. The default value is 8192, with a valid value range for between 8192 and 1048576. If necessary, the value provided by a user for this variable is rounded up to the operating system's nearest page size.

- APT_MIN_TRANSPORT_BLOCK_SIZE

  This variable specifies the minimum allowable block size for transferring data between players. The default block size is 8192. The block size cannot be less than 8192, or greater than 1048576. This variable is only meaningful when used in combination with APT_LATENCY_COEFFICIENT, APT_AUTO_TRANSPORT_BLOCK_SIZE, and APT_MAX_TRANSPORT_BLOCK_SIZE

- APT_MAX_TRANSPORT_BLOCK_SIZE

  This variable specifies the maximum allowable block size for transferring data between players. The default block size is 1048576. It cannot be less than 8192, or greater than 1048576. This variable is only meaningful when used in combination with APT_LATENCY_COEFFICIENT, APT_AUTO_TRANSPORT_BLOCK_SIZE, and APT_MMIN_TRANSPORT_BLOCK_SIZE

- APT_AUTO_TRANSPORT_BLOCK_SIZE

  If set, the framework calculates the block size for transferring data between players according to the following algorithm:

  ```
  If (recordSize * APT_LATENCY_COEFFICIENT <
  APT_MIN_TRANSPORT_BLOCK_SIZE), then blockSize =
  APT_MIN_TRANSPORT_BLOCK_SIZE
  Else, if (recordSize * APT_LATENCY_COEFFICIENT >
  APT_MAX_TRANSPORT_BLOCK_SIZE), then blockSize =
  APT_MAX_TRANSPORT_BLOCK_SIZE
  Else, blockSize = recordSize * APT_LATENCY_COEFFICIENT
  ```

- APT_LATENCY_COEFFICIENT

  This variable specifies the number of records to be written to each transport block.

- APT_SHARED_MEMORY_BUFFERS

  This variable specifies the number of Transport Blocks between a pair of operators. It must be at least two.

Note: The following environment variables are used only with fixed-length records:

- *APT_MIN/MAX_TRANSPORT_BLOCK_SIZE*
- APT_LATENCY_COEFFICIENT
- APT_AUTO_TRANSPORT_BLOCK_SIZE

12.4.2 Deadlock prevention buffering

The other type of buffering, Deadlock Prevention, comes into play anytime there is a Fork-Join structure in a job. Figure 12-3 is an example job fragment.

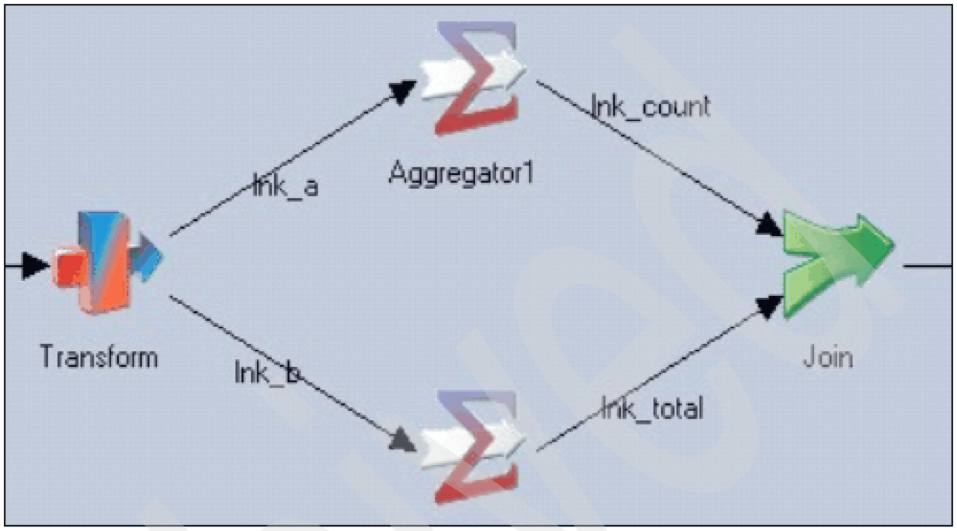


Figure 12-3 Fork-Join example

In this example, the Transformer creates a fork with two parallel Aggregators, which go into an Inner Join. Note however, that Fork-Join is a graphical description, it does not necessarily have to involve a Join stage.

To understand deadlock-prevention buffering, it is important to understand the operation of a parallel pipeline. Imagine that the Transformer is waiting to write to Aggregator1, Aggregator2 is waiting to read from the Transformer, Aggregator1 is waiting to write to the Join, and Join is waiting to read from Aggregator2. This operation is depicted in Figure 12-4. The arrows represent dependency direction, instead of data flow.

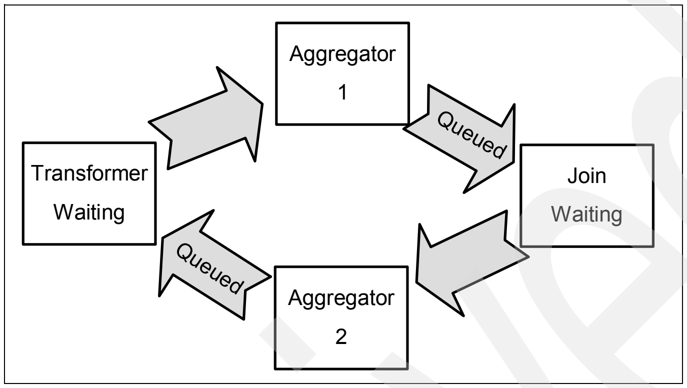


Figure 12-4 Dependency direction

Without deadlock buffering, this scenario would create a circular dependency where Transformer is waiting on Aggregator1, Aggregator1 is waiting on Join, Join is waiting on Aggregator2, and Aggregator2 is waiting on Transformer. Without deadlock buffering, the job would deadlock, bringing processing to a halt (though the job does not stop running, it would eventually time out).

To guarantee that this problem never happens in parallel jobs, there is a specialized operator called BufferOp. BufferOp is always ready to read or write and does not allow a read/write request to be queued. It is placed on all inputs to a join structure (again, not necessarily a Join stage) by the parallel framework during job startup. So the job structure is altered to look like that in Figure 12-5. Here the arrows now represent data-flow, not dependency.

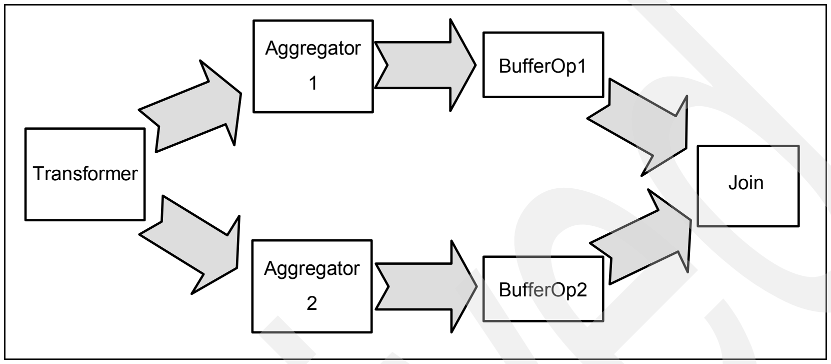


Figure 12-5 Data flow

Because BufferOp is always ready to read or write, Join cannot be stuck waiting to read from either of its inputs, breaking the circular dependency and guaranteeing no deadlock occurs.

BufferOps is also placed on the input partitions to any Sequential stage that is fed by a Parallel stage, as these same types of circular dependencies can result from partition-wise Fork-Joins.

By default, BufferOps allocates 3 MB of memory each (remember that this is per operator, per partition). When that is full (because the upstream operator is still writing but the downstream operator is not ready to accept that data yet) it begins to flush data to the scratch disk resources specified in the configuration file (detailed in the *DataStage Designer Client Guide*).

Tip: For wide rows, it might be necessary to increase the default buffer size (APT_BUFFER_MAXIMUM_MEMORY) to hold more rows in memory.

The behavior of deadlock-prevention BufferOps can be tuned through these environment variables:

- APT_BUFFER_DISK_WRITE_INCREMENT

  This environment variable controls the blocksize written to disk as the memory buffer fills. The default 1 MB. This variable cannot exceed two-thirds of the APT_BUFFER_MAXIMUM_MEMORY value.

- APT_BUFFER_FREE_RUN

  This environment variable details the maximum capacity of the buffer operator before it starts to offer resistance to incoming flow, as a nonnegative (proper or improper) fraction of APT_BUFFER_MAXIMUM_MEMORY. Values greater than 1 indicate that the buffer operator free runs (up to a point) even when it has to write data to disk.

- APT_BUFFER_MAXIMUM_MEMORY

  This environment variable details the maximum memory consumed by each buffer operator for data storage. The default is 3 MB.

- APT_BUFFERING_POLICY

  This environment variable specifies the buffering policy for the entire job. When it is not defined or is defined to be the null string, the default buffering policy is AUTOMATIC_BUFFERING. Valid settings are as follows:

  – AUTOMATIC_BUFFERING

    Buffer as necessary to prevent dataflow deadlocks

  – FORCE_BUFFERING

    Buffer all virtual datasets

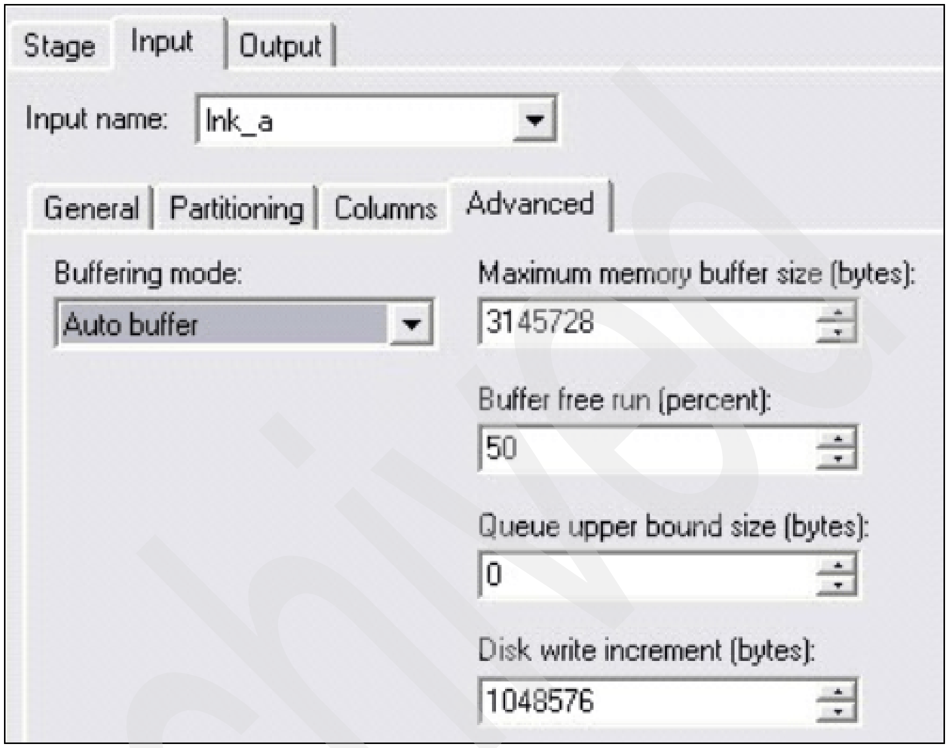
  – NO_BUFFERING

    Inhibit buffering on all virtual datasets

Attention: Inappropriately specifying NO_BUFFERING can cause dataflow deadlock during job execution. This setting is only recommended for advanced users.

Note: FORCE_BUFFERING can be used to reveal bottlenecks in a job design during development and performance tuning, but degrades performance and therefore must not be used in production job runs.

Additionally, the buffer mode, buffer size, buffer free run, queue bound, and write increment can be set on a per-stage basis from the Input/ Advanced tab of the stage properties, as shown in Figure 12-6.



Input name: lnk_a

Buffering mode: Auto buffer

Maximum memory buffer size (bytes): 3145728

Buffer free run (percent): 50

Queue upper bound size (bytes): 0

Disk write increment (bytes): 1048576

Figure 12-6 Stage properties Input Advanced tab

Aside from ensuring that no deadlock occurs, BufferOps also have the effect of smoothing out production/consumption spikes. This allows the job to run at the highest rate possible even when a downstream stage is ready for data at various times than when its upstream stage is ready to produce that data. When attempting to address these mismatches in production/consumption, it is best to tune the buffers on a per-stage basis, instead of globally through environment variable settings.

Important: Choosing which stages to tune buffering for and which to leave alone is as much art as a science, and must be considered among the last resorts for performance tuning.

Stages upstream/downstream from high-latency stages (such as remote databases, NFS mount points for data storage, and so forth) are a good place to start. If that does not yield enough of a performance boost (remember to test iteratively, changing only one thing at a time), setting the buffering policy to FORCE_BUFFERING causes buffering to occur everywhere.

By using the performance statistics in conjunction with this buffering, you might be able identify points in the data flow where a downstream stage is waiting on an upstream stage to produce data. Each place might offer an opportunity for buffer tuning.

As implied, when a buffer has consumed its RAM, it asks the upstream stage to slow down. This is called pushback. Because of this, if you do not have force buffering set and APT_BUFFER_FREE_RUN set to at least approximately 1000, you cannot determine that any one stage is waiting on any other stage, as another stage far downstream can be responsible for cascading pushback all the way upstream to the place you are seeing the bottleneck.

13

Database stage guidelines

In this chapter we present guidelines for existing database stages. That is, non-Connector database stage types that were available in DataStage (DS) 7.X and Information Server 8.0.1:

- ▸ Native Parallel (Enterprise) stages
- ▸ Plug-in (API) stages

Existing database stages are discussed in this chapter for ease of reference, especially when dealing with earlier releases of Information Server (IS) and DataStage.

Connector stages are discussed in Chapter 14, “Connector stage guidelines” on page 221.

As a general guideline, new projects should give preference to Connector stages, and take advantage of existing Enterprise equivalents in specific cases where these have an edge in terms of performance.

13.1 Existing database development overview

In this section we provide guidelines for accessing any database from in DataStage Parallel Jobs using existing database stages. Subsequent sections provide database-specific tips and guidelines.

13.1.1 Existing database stage types

DataStage offers database connectivity through native parallel and plug-in stage types. For certain databases (DB2, Informix, Oracle, and Teradata), multiple stage types are available"

- Native Parallel Database stages
  - DB2/UDB Enterprise
  - Informix Enterprise
  - ODBC Enterprise
  - Oracle Enterprise
  - Netezza Enterprise (load only)
  - SQLServer Enterprise
  - Teradata Enterprise
- Plug-In Database stages

Dynamic RDBMS

  - DB2/UDB API
  - DB2/UDB Load
  - Informix CLI
  - Informix Load
  - Informix XPS Load
  - Oracle OCI Load
  - RedBrick Load
  - Sybase IQ12 Load
  - Sybase OC
  - Teradata API
  - Teradata MultiLoad (MultiLoad)
  - Teradata MultiLoad (TPump)

Note: Not all database stages (for example, Teradata API) are visible in the default DataStage Designer palette. You might need to customize the palette to add hidden stages.

Native Parallel Database stages

In general, for maximum parallel performance, scalability, and features it is best to use the native parallel database stages in a job design, if connectivity requirements can be satisfied.

Because there are exceptions to this rule (especially with Teradata), specific guidelines for when to use various stage types are provided in the database-specific topics in this section.

Because of their integration with database technologies, native parallel stages often have more stringent connectivity requirements than plug-in stages. For example, the DB2/UDB Enterprise stage is only compatible with DB2 Enterprise Server Edition with DPF on the same UNIX platform as the DataStage Engine.

Native parallel stages always pre-query the database for actual runtime metadata (column names, types, attributes). This allows DataStage to match return columns by name, not position in the stage Table Definitions. However, care must be taken to assign the correct data types in the job design.

ODBC Enterprise stage

In general, native database components (such as the Oracle Enterprise stage) are preferable to ODBC connectivity if both are supported on the database platform, operating system, and version. Furthermore, the ODBC Enterprise stage is not designed to interface with database-specific parallel load technologies.

The benefit of ODBC Enterprise stage comes from the large number of included and third-party ODBC drivers to enable connectivity to all major database platforms. ODBC also provides an increased level of data virtualization that can be useful when sources and targets (or deployment platforms) change.

DataStage bundles OEM versions of ODBC drivers from DataDirect. On UNIX, the DataDirect ODBC Driver Manager is also included. Wire Protocol ODBC Drivers generally do not require database client software to be installed on the server platform.

Plug-In database stages

Plug-in stage types are intended to provide connectivity to database configurations not offered by the native parallel stages. Because plug-in stage types cannot read in parallel, and cannot span multiple servers in a clustered or grid configuration, they can only be used when it is not possible to use a native parallel stage.

From a design perspective, plug-in database stages match columns by order, not name, so Table Definitions must match the order of columns in a query.

13.1.2 Database metadata

In this section we describe use and types of metadata.

Runtime metadata

At runtime, the DS native parallel database stages always pre-query the database source or target to determine the actual metadata (column names, data types, null-ability) and partitioning scheme (in certain cases) of the source or target table.

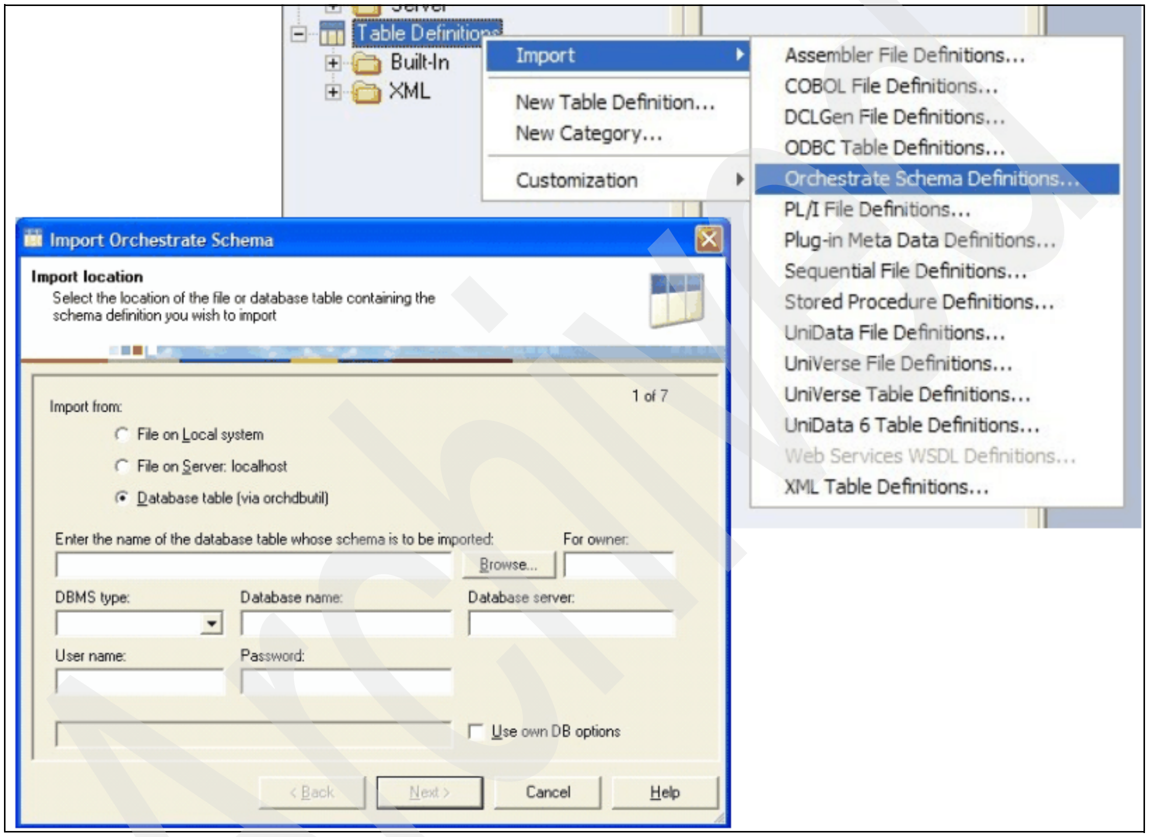
For each native parallel database stage, the following elements are true:

- Rows of the database result set correspond to records of a parallel dataset.
- Columns of the database row correspond to columns of a record.
- The name and data type of each database column corresponds to a parallel dataset name and data type using a predefined mapping of database data types to parallel data types.
- Both the DataStage Parallel Framework and relational databases support null values, and a null value in a database column is stored as an out-of-band NULL value in the DataStage column.

The actual metadata used by a native parallel database stage is always determined at runtime, regardless of the table definitions assigned by the DataStage developer. This allows the database stages to match return values by column name instead of position. However, care must be taken that the column data types defined by the DataStage developer match the data types generated by the database stage at runtime. Database-specific data type mapping tables are included in the following sections.

Metadata import

When using the native parallel DB2 Enterprise, Informix Enterprise or Oracle Enterprise stages, use **orchdbutil** to import metadata to avoid type conversion issues. This utility is available as a server command line utility and in Designer by clicking **Table Definitions** ∅ **Import** ∅ **Orchestrate Schema Definitions**, and selecting the **Import from Database Table** option in the wizard, as illustrated in Table 13-1 on page 198.


Server
Table Definitions
Built-In
XML
Import
New Table Definition...
New Category...
Customization
Assembler File Definitions...
COBOL File Definitions...
DCLGen File Definitions...
ODBC Table Definitions...
Orchestrate Schema Definitions...
PL/I File Definitions...
Plug-in Meta Data Definitions...
Sequential File Definitions...
Stored Procedure Definitions...
UniData File Definitions...
UniVerse File Definitions...
UniVerse Table Definitions...
UniData 6 Table Definitions...
Web Services WSDL Definitions...
XML Table Definitions...
Import Orchestrate Schema
< Back
Next >
Cancel
Help
Browse...

Import location

Select the location of the file or database table containing the schema definition you wish to import

1 of 7

Import from:

( ) File on Local system

( ) File on Server: localhost

[✓ Database table (via orchdbutil)]

Enter the name of the database table whose schema is to be imported:

For owner:

DBMS type:

Database name:

Database server:

User name:

Password:

☐ Use own DB options

Figure 13-1 Orchdbutil metadata import

One disadvantage to the graphical **orchdbutil** metadata import is that the user interface requires each table to be imported individually.

When importing a large number of tables, it is easier to use the corresponding **orchdbutil** command-line utility from the DataStage server machine. As a command, **orchdbutil** can be scripted to automate the process of importing a large number of tables.

Defining metadata for database functions

When using database functions in a SQL SELECT list in a Read or Lookup, it is important to use SQL aliases to name the calculated columns so that they can be referenced in the DataStage job. The alias names must be added to the Table Definition in DataStage.

For example, the following SQL assigns the alias “Total” to the calculated column:

```
SELECT store_name, SUM(sales) Total
FROM store_info
GROUP BY store_name
```

In many cases it might be more appropriate to aggregate using the Parallel Aggregator stage. However, there might be cases where user-defined functions or logic need to be executed on the database server.

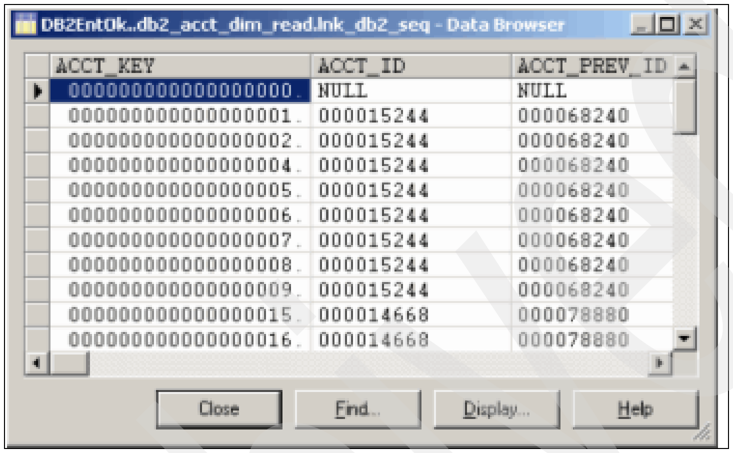
13.1.3 Optimizing select lists

For best performance and optimal memory usage, it is best to specify column names on all source database stages, instead of using an unqualified Table or SQL SELECT * read. For the Table read method, always specify the Select List subproperty. For Auto-Generated SQL, the DataStage Designer automatically populates the select list based on the stage’s output column definition.

The only exception to this rule is when building dynamic database jobs that use runtime column propagation to process all columns in a source table.

13.1.4 Testing database connectivity

The **View Data** button on the Output/Properties tab of source database stages lets you verify database connectivity and settings without having to create and run a job. Test the connection using **View Data** button. If the connection is successful, you see a window with the result columns and data, similar to Figure 13-2.

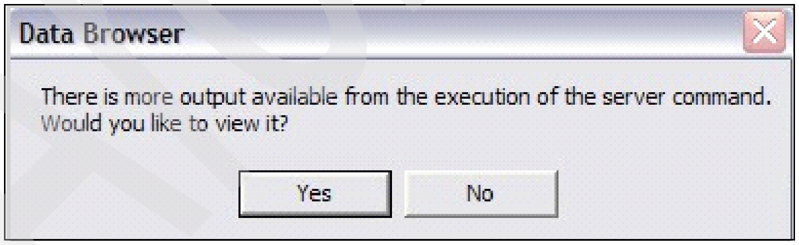


DB2EntOk..db2_acct_dim_read.lnk_db2_seq - Data Browser

ACCT_KEY	ACCT_ID	ACCT_PREV_ID
000000000000000000.	NULL	NULL
000000000000000001.	000015244	000068240
000000000000000002.	000015244	000068240
000000000000000004.	000015244	000068240
000000000000000005.	000015244	000068240
000000000000000006.	000015244	000068240
000000000000000007.	000015244	000068240
000000000000000008.	000015244	000068240
000000000000000009.	000015244	000068240
000000000000000015.	000014668	000078880
000000000000000016.	000014668	000078880

Figure 13-2 Sample view: Data output

If the connection fails, an error message might appear. You are prompted to view additional detail. Clicking **YES**, as in Figure 13-3, displays a detailed dialog box with the specific error messages generated by the database stage that can be useful in debugging a database connection failure.



Data Browser

There is more output available from the execution of the server command. Would you like to view it?

Figure 13-3 View additional error detail

13.1.5 Designing for restart

To enable restart of high-volume jobs, it is important to separate the transformation process from the database write (Load or Upsert) operation. After transformation, the results must be landed to a parallel dataset. Subsequent jobs should read this dataset and populate the target table using the appropriate database stage and write method.

As a further optimization, a Lookup stage (or Join stage, depending on data volume) can be used to identify existing rows before they are inserted into the target table.

13.1.6 Database OPEN and CLOSE commands

The native parallel database stages provide options for specifying `OPEN` and `CLOSE` commands. These options allow commands (including SQL) to be sent to the database before (`OPEN`) or after (`CLOSE`) all rows are read/written/loaded to the database. `OPEN` and `CLOSE` are not offered by plug-in database stages. For example, the `OPEN` command can be used to create a temporary table, and the `CLOSE` command can be used to select all rows from the temporary table and insert into a final target table.

As another example, the `OPEN` command can be used to create a target table, including database-specific options (such as table space, logging, and constraints) not possible with the Create option. In general, it is not a good idea to let DataStage generate target tables unless they are used for temporary storage. There are limited capabilities to specify Create table options in the stage, and doing so might violate data-management (DBA) policies.

It is important to understand the implications of specifying a user-defined `OPEN` and `CLOSE` command. For example, when reading from DB2, a default `OPEN` statement places a shared lock on the source. When specifying a user-defined `OPEN` command, this lock is not sent and must be specified explicitly if appropriate.

Further details are outlined in the respective database sections of the *Orchestrate Operators Reference*, which is part of the Orchestrate OEM documentation.

13.1.7 Database sparse lookup versus join

Data read by any database stage can serve as the reference input to a Lookup Operation. By default, this reference data is loaded into memory like any other reference link (Normal Lookup).

When directly connected as the reference link to a Lookup stage, the DB2/UDB Enterprise, ODBC Enterprise, and Oracle Enterprise stages allow the Lookup type to be changed to Sparse, sending individual SQL statements to the reference database for each incoming Lookup row. Sparse Lookup is only available when the database stage is directly connected to the reference link, with no intermediate stages.

Important: The individual SQL statements required by a Sparse Lookup are an expensive operation from a performance perspective. In most cases, it is faster to use a DataStage JOIN stage between the input and DB2 reference data than it is to perform a Sparse Lookup.

For scenarios where the number of input rows is significantly smaller (for example, 1:100 or more) than the number of reference rows in a DB2 or Oracle table, a Sparse Lookup might be appropriate.

13.1.8 Appropriate use of SQL and DataStage

When using relational database sources, there is often a functional overlap between SQL and DataStage functionality. Although it is possible to use either SQL or DataStage to solve a given business problem, the optimal implementation involves making use of the strengths of each technology to provide maximum throughput and developer productivity.

Though there are extreme scenarios when the appropriate technology choice is clearly understood, there might be less obvious areas where the decision must be made based on factors such as developer productivity, metadata capture and re-use, and ongoing application maintenance costs. The following guidelines can assist with the appropriate use of SQL and DataStage technologies in a given job flow:

- When possible, use a SQL filter (WHERE clause) to limit the number of rows sent to the DataStage job. This minimizes impact on network and memory resources, and makes use of the database capabilities.
- Use a SQL Join to combine data from tables with a small number of rows in the same database instance, especially when the join columns are indexed. A join that reduces the result set significantly is also often appropriate to do in the database.

- When combining data from large tables, or when the source includes a large number of database tables, the efficiency of the DS parallel Sort and Join stages can be significantly faster than an equivalent SQL query. In this scenario, it can still be beneficial to use database filters (WHERE clause) if appropriate.
- Avoid the use of database stored procedures (for example, Oracle PL/SQL) on a per-row basis in a high-volume data flow. For maximum scalability and parallel performance, implement business rules using native parallel DataStage components.

13.2 Existing DB2 guidelines

In this section we provide guidelines for working with stages.

13.2.1 Existing DB2 stage types

DataStage provides access to DB2 databases using one of 5 stages, summarized in the Table 13-1.

Table 13-1 Database access using stages

DataStage Stage name	Stage type	DB2 requirement	Supports partitioned DB2?	Parallel read?	Parallel write?	Parallel Sparse Lookup	SQL Open / Close
DB2/UDB Enterprise	Native Parallel	DPF, same platform as ETL server [a]	Yes / directly to each DB2 node	Yes	Yes	Yes	Yes
DB2/UDB API	Plug-In	Any DB2 through DB2 Client or DB2-Connect	Yes / through DB2 node 0	No	Possible Limitations	No	No
DB2/UDB Load	Plug-In	Subject to DB2 Loader Limitations	No	No	No	No	No
ODBC Enterprise	Native	Any DB2 through DB2 Client or DBE-Connect	Yes / through DB2 node 0	No[b]	No	No	No
Dynamic RDBMS	Plug-In	Any DB2 through DB2 Client or DB2-Connect	Yes / through DB2 node 0	No	Possible Limitations	No	No

a. It is possible to connect the DB2 UDB stage to a remote database by cataloging the remote database in the local instance and then using it as though it were a local database. This only works when the authentication mode of the database on the remote instance is set to *client authentication*. If you use the stage in this way, you might experience data duplication when working in partitioned instances because the node configuration of the local instance might not be the same as the remote instance. For this reason, the *client authentication* configuration of a remote instance is not recommended.

b. A patched version of the ODBC Enterprise stage allowing parallel read is available from IBM InfoSphere Support for certain platforms. Check with IBM InfoSphere Support for availability.

For specific details on the stage capabilities, consult the DataStage documentation (*DataStage Parallel Job Developers Guide*, and *DataStage Plug-In* guides)

DB2/UDB Enterprise stage

DataStage provides native parallel read, lookup, upsert, and load capabilities to parallel DB2 databases on UNIX using the native parallel DB2/UDB Enterprise stage. The DB2/UDB Enterprise stage requires DB2 Enterprise Server Edition on UNIX with the Data Partitioning Facility (DPF) option. (Before DB2 V8, this was called DB2 EEE.) Furthermore, the DB2 hardware/UNIX/software platform must match the hardware/software platform of the DataStage ETL server.

As a native, parallel component the DB2/UDB Enterprise stage is designed for maximum performance and scalability. These goals are achieved through tight integration with the DB2 Relational Database Management System (RDBMS), including direct communication with each DB2 database node, and reading from and writing to DB2 in parallel (where appropriate), using the same data partitioning as the referenced DB2 tables.

ODBC and DB2 plug-In stages

The ODBC Enterprise and plug-in stages are designed for lower-volume access to DB2 databases without the DPF option installed (prior to V8, DB2 EEE). These stages also provide connectivity to non-UNIX DB2 databases, databases on UNIX platforms that differ from the platform of the DataStage Engine, or DB2 databases on Windows or Mainframe platforms (except for the Load stage against a mainframe DB2 instance, which is not supported).

By facilitating flexible connectivity to multiple types of remote DB2 database servers, the use of DataStage plug-in stages limits overall performance and scalability. Furthermore, when used as data sources, plug-in stages cannot read from DB2 in parallel.

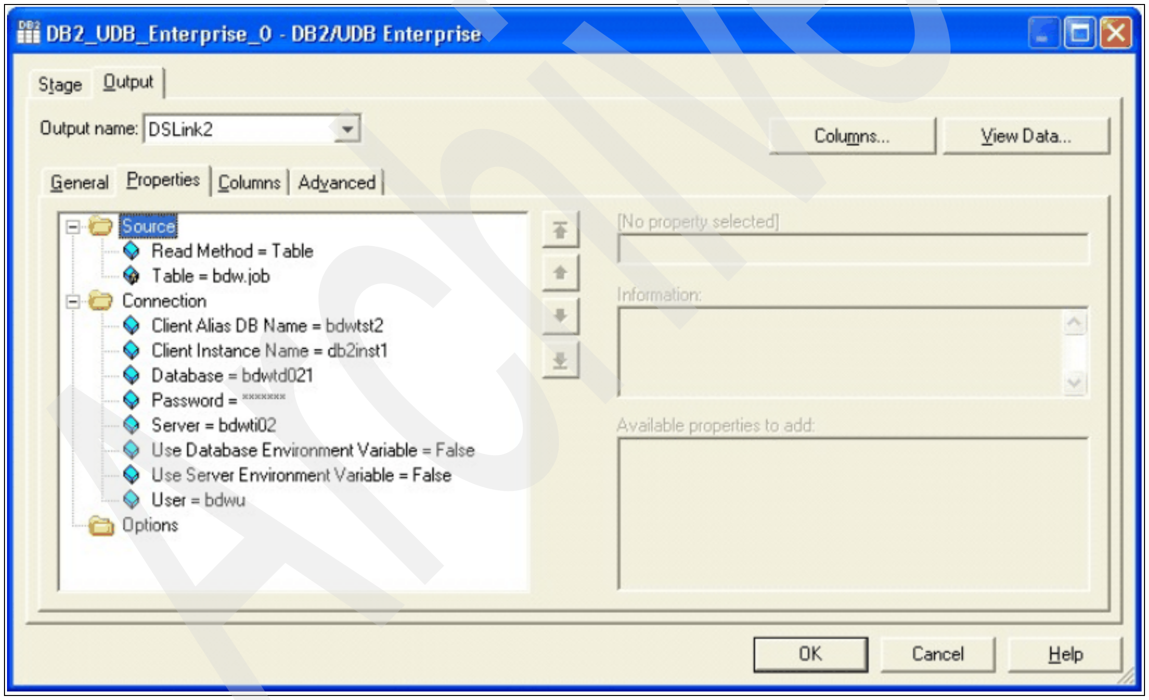
Using the DB2/UDB API stage or the Dynamic RDBMS stage, it might be possible to write to a DB2 target in parallel, because the DataStage parallel framework instantiates multiple copies of these stages to handle the data that has already been partitioned in the parallel framework. Because each plug-in invocation opens a separate connection to the same target DB2 database table, the ability to write in parallel might be limited by the table and index configuration set by the D2 database administrator.

The DB2/API (plug-in) stage must be used to read from and write to DB2 databases on non-UNIX platforms (such as mainframe editions through DB2-Connect). Sparse Lookup is not supported through the DB2/API stage.

13.2.2 Connecting to DB2 with the DB2/UDB Enterprise stage

Perform the following steps to connect to DB2 with the DB2/UDB Enterprise stage:

1. Create a Parallel job and add a DB2/UDB Enterprise stage.
2. Add the properties as depicted in Figure 13-4.



Output name: DSLink2

```
Source
    Read Method = Table
    Table = bdw.job
Connection
    Client Alias DB Name = bdwtst2
    Client Instance Name = db2inst1
    Database = bdwtd021
    Password = *******
    Server = bdwti02
    Use Database Environment Variable = False
    Use Server Environment Variable = False
    User = bdwu
Options
```

Figure 13-4 DB2/UDB Enterprise stage properties

3. For connection to a remote DB2/UDB instance, set the following properties on the DB2/UDB Enterprise stage in your parallel job:
   - Client Instance Name: Set this to the DB2 client instance name. If you set this property, DataStage assumes you require remote connection.
   - Server (Optional): Set this to the instance name of the DB2 server. Otherwise use the DB2 environment variable, DB2INSTANCE, to identify the instance name of the DB2 server.
   - Client Alias DB Name: Set this to the DB2 client's alias database name for the remote DB2 server database. This is required only if the client's alias is different from the actual name of the remote server database.
   - Database (Optional): Set this to the remote server database name. Otherwise use the environment variables $APT_DBNAME or $APT_DB2DBDFT to identify the database.
   - User: Enter the user name for connecting to DB2, this is required for a remote connection to retrieve the catalog information from the local instance of DB2 and thus must have privileges for that local instance.
   - Password: Enter the password for connecting to DB2, this is required for a remote connection to retrieve the catalog information from the local instance of DB2 and thus must have privileges for that local instance.

13.2.3 Configuring DB2 multiple instances in one DataStage job

Although it is not officially supported, it is possible to connect to more than one DB2 instance in a single job. Your job must meet one of the following configurations (The use of the word *stream* refers to a contiguous flow of one stage to another in a single job):

- Single stream - Two Instances Only

  Reading from one instance and writing to another instance with no other DB2 instances (not sure how many stages of these two instances can be added to the canvas for this configuration for lookups)

- Two Stream – One Instance per Steam

  Reading from instance A and writing to instance A and reading from instance B and writing to instance B (not sure how many stages of these two instances can be added to the canvas for this configuration for lookups)

- Multiple Stream - with N DB2 sources with no DB2 targets

  Reading from one to *n* DB2 instances in separate source stages with no downstream other DB2 stages

To get this configuration to work correctly, you must adhere to all of the directions specified for connecting to a remote instance AND the following:

- You must not set the APT_DB2INSTANCE_HOME environment variable. After this variable is set, it tries to use it for each of the connections in the job. Because a `db2nodes.cfg` file can only contain information for one instance, this creates problems.
- In order for DataStage to locate the `db2nodes.cfg` file, you must build a user on the DataStage server with the same name as the instance to which you are trying to connect. (The default logic for the DB2/UDB Enterprise stage is to use the home directory of the instance, as defined for the UNIX user with the same name as the DB2 instance.) In the user's UNIX home directory, create a sqllib subdirectory and place the `db2nodes.cfg` file of the remote instance there. Because the APT_DB2INSTANCE_HOME is not set, DS defaults to this directory to find the configuration file for the remote instance.

To connect to multiple DB2 instances, use separate jobs with their respective DB2 environment variable settings, landing intermediate results to a parallel dataset. Depending on platform configuration and I/O subsystem performance, separate jobs can communicate through named pipes, although this incurs the overhead of Sequential File stage (corresponding export/import operators), which does not run in parallel.

If the data volumes are sufficiently small, DB2 plug-in stages (DB2 API, DB2 Load, Dynamic RDBMS) can be used to access data in other instances.

13.2.4 DB2/UDB Enterprise stage column names

At runtime, the native parallel DB2/UDB Enterprise stage translates column names exactly except when a component of a DB2 column name is not compatible with DataStage column naming conventions. The naming conventions place no limit on the length of a column name, but have the following restrictions:

- The name must start with a letter or underscore character
- The name can contain only alphanumeric and underscore characters
- The name is case insensitive

When there is an incompatibility, the DB2/UDB Enterprise stage converts the DB2 column name as follows:

- If the DB2 column name does not begin with a letter or underscore, the string APT__column# (two underscores) is added to beginning of the column name, where "column#" is the number of the column. For example, if the third DB2 column is named 7dig, the DataStage column is named APT__37dig.
- If the DB2 column name contains a character that is not alphanumeric or an underscore, the character is replaced by two underscore characters.

13.2.5 DB2/API stage column names

When using the DB2/API, DB2 Load, and Dynamic RDBMS plug-in stages, set the environment variable $DS_ENABLE_RESERVED_CHAR_CONVERT if your DB2 database uses the reserved characters # or $ in column names. This converts these special characters into an internal representation DataStage can understand.

Observe the following guidelines when $DS_ENABLE_RESERVED_CHAR_CONVERT is set:

- Avoid using the strings __035__ and __036__ in your DB2 column names (these are used as the internal representations of # and $ respectively)
- Import metadata using the Plug-in Meta Data Import tool. Avoid hand editing (this minimizes the risk of mistakes or confusion).
- Once the table definition is loaded, the internal column names are displayed rather than the original DB2 names both in table definitions and in the data browser. They are also used in derivations and expressions.
- The original names are used in generated SQL statements, however. Use them if entering SQL in the job yourself.

13.2.6 DB2/UDB Enterprise stage data type mapping

The DB2 database schema to be accessed must not have any columns with User Defined Types (UDTs). Use the **`db2 describe table [table-name]`** command on the DB2 client for each table to be accessed to determine if UDTs are in use. Alternatively, examine the DDL for each schema to be accessed.

Table definitions must be imported into DataStage using `orchdbutil` to ensure accurate Table definitions. The DB2/UDB Enterprise stage converts DB2 data types to DataStage parallel data types, as shown in Table 13-2.

Table 13-2 Parallel data types

DB2 data types	DataStage parallel data types
CHAR(n)	string[n] or ustring[n]
CHARACTER VARYING(n,r)	string[max=n] or ustring[max=n]
DATE	date
DATETIME	Time or time stamp with corresponding fractional precision for time: ► If the DATETIME starts with a year component, the result is a time stamp field. ► If the DATETIME starts with an hour, the result is a time field.
DECIMAL[p,s]	decimal[p,s] where *p* is the precision and *s* is the scale
DOUBLE-PRECISION	dfloat
FLOAT	dfloat
INTEGER	int32
MONEY	decimal
NCHAR(n,r)	string[n] or ustring[n]
NVARCHAR(n,r)	string[max=n] or ustring[max=n]
REAL	sfloat
SERIAL	int32
SMALLFLOAT	sfloat
SMALLINT	int16
VARCHAR(n)	string[max=n] or ustring[max=n]

Important: DB2 data types that are not listed in the table cannot be used in the DB2/UDB Enterprise stage, and generate an error at runtime.

13.2.7 DB2/UDB Enterprise stage options

The DB2/UDB Enterprise (native parallel) stage must be used for reading from, performing lookups against, and writing to a DB2 Enterprise Server Edition database with Database Partitioning Feature (DPF)

- As a native, parallel component, the DB2/UDB Enterprise stage is designed for maximum performance and scalability against large partitioned DB2 UNIX databases.
- DB2/UDB Enterprise stage is tightly integrated with the DB2 RDBMS, communicating directly with each database node, reading from and writing to DB2 in parallel (where appropriate), and using the same data partitioning as the referenced DB2 tables.

When writing to a DB2 database in parallel, the DB2/UDB Enterprise stage offers the choice of SQL (insert/update/upsert/delete) or fast DB2 loader methods. The choice between these methods depends on required performance, database log usage, and recoverability.

- The Write Method (and corresponding insert/update/upsert/delete) communicates directly with the DB2 database nodes to execute instructions in parallel. All operations are logged to the DB2 database log, and the target tables might be accessed by other users. Time and row-based commit intervals determine the transaction size, and the availability of new rows to other applications.
- The DB2 Load method requires that the DataStage user running the job have DBADM privilege on the target DB2 database. During the load operation, the DB2 Load method places an exclusive lock on the entire DB2 table space into which it loads the data. No other tables in that table space can be accessed by other applications until the load completes. The DB2 load operator performs a non-recoverable load. That is, if the load operation is terminated before it is completed, the contents of the table are unusable and the table space is left in a load pending state. In this scenario, the DB2 Load DataStage job must be rerun in Truncate mode to clear the load pending state.

13.2.8 Performance notes

In certain cases, when using user-defined SQL without partitioning against large volumes of DB2 data, the overhead of routing information through a remote DB2 coordinator might be significant. In these instances, it might be beneficial to have the DB2 DBA configure separate DB2 coordinator nodes (no local data) on each ETL server (in clustered Extract, Transform, and Load (ETL) configurations). In this configuration, DB2 Enterprise stage should not include the Client Instance Name property, forcing the DB2 Enterprise stages on each ETL server to communicate directly with their local DB2 coordinator.

13.3 Existing Informix database guidelines

In this section we describe the Informix database guidelines.

13.3.1 Informix Enterprise stage column names

For each Informix Enterprise stage, the following elements must be in effect:

- Rows of the database result set correspond to records of a parallel dataset.
- Columns of the database row correspond to columns of a DataStage parallel record.
- The name and data type of each database column corresponds to a parallel dataset name and data type using a predefined mapping of database data types to parallel data types.
- Both DS parallel jobs and Informix support null values, and a null value in a database column is stored as an out-of-band NULL value in the DS column.

13.3.2 Informix Enterprise stage data type mapping

Table Definitions must be imported into DataStage using **`orchdbutil`** to ensure accurate Table Definitions. The Informix Enterprise stage converts Informix data types to parallel data types, as shown in Table 13-3 on page 207.

Table 13-3 Data type mapping

Informix Data Type	DS Parallel Data Type
CHAR(n)	string[n]
CHARACTER VARYING(n,r)	string[max=n]
DATE	date
DATETIME	date, time or time stamp with corresponding fractional precision for time: ► If the DATETIME starts with a year component and ends with a month, the result is a date field. ► If the DATETIME starts with a year component, the result is a time stamp field. ► If the DATETIME starts with an hour, the result is a time field.
DECIMAL[p,s]	decimal[p,s] where *p* is the precision and *s* is the scale The maximum precision is 32. A decimal with floating scale is converted to dfloat.
DOUBLE-PRECISION	dfloat
FLOAT	dfloat
INTEGER	int32
MONEY	decimal
NCHAR(n,r)	string[n]
NVARCHAR(n,r)	string[max=n]
REAL	sfloat
SERIAL	int32
SMALLFLOAT	sfloat
SMALLINT	int16
VARCHAR(n)	string[max=n]

Important: Informix data types that are not listed in the table cannot be used in the Informix Enterprise stage, and generates an error at runtime.

13.4 ODBC Enterprise guidelines

In this section we describe the ODBC Enterprise guidelines.

13.4.1 ODBC Enterprise stage column names

For each ODBC Enterprise stage the following elements must be in effect:

- Rows of the database result set correspond to records of a parallel dataset.
- Columns of the database row correspond to columns of a DS record.
- The name and data type of each database column corresponds to a parallel dataset name and data type using a predefined mapping of database data types to parallel data types.
- Names are translated exactly except when the external data source column name contains a character that DataStage does not support. In that case, two underscore characters replace the unsupported character.
- Both DS and ODBC support null values, and a null value in a database column is stored as an out-of-band NULL value in the DS column.

13.4.2 ODBC Enterprise stage data type mapping

ODBC data sources are not supported by the orcdbutil utility. It is important to verify the correct ODBC to DS parallel data type mapping, as shown in Table 13-4.

Table 13-4 Data type mapping

ODBC Data Type	DS Parallel Data Type
SQL_BIGINT	int64
SQL_BINARY	raw(n)
SQL_CHAR	string[n]
SQL_DECIMAL	decimal[p,s] where *p* is the precision and *s* is the scale
SQL_DOUBLE	decimal[p,s]
SQL_FLOAT	decimal[p,s]
SQL_GUID	string[36]
SQL_INTEGER	int32

ODBC Data Type	DS Parallel Data Type
SQL_BIT	int8 [0 or 1]
SQL_REAL	decimal[p,s]
SQL_SMALLINT	int16
SQL_TINYINT	int8
SQL_TYPE_DATE	date
SQL_TYPE_TIME	time[p]
SQL_TYPE_TIMESTAMP	timestamp[p]
SQL_VARBINARY	raw[max=n]
SQL_VARCHAR	string[max=n]
SQL_WCHAR	ustring[n]
SQL_WVARCHAR	ustring[max=n]

The maximum size of a DataStage record is limited to 32 K. If you attempt to read a record larger than 32 K, the parallel framework returns an error and abort your job.

Important: ODBC data types that are not listed in the table cannot be used in the ODBC Enterprise stage, and generates an error at runtime.

13.4.3 Reading ODBC sources in parallel

Starting with DataStage release 7.5.2, the ODBC Enterprise stage can read in parallel. By default, the ODBC Enterprise stage reads sequentially.

To read in parallel with the ODBC Enterprise stage, specify the Partition Column option. For optimal performance, this column must be indexed in the database.

13.4.4 Writing to ODBC targets in parallel

In general, avoid writing to ODBC targets in parallel, even though the ODBC Enterprise stage supports parallel execution.

Depending on the target database, and the table configuration (Row- or Page-Level lock mode if available), it might be possible to write to a target database in parallel using the ODBC Enterprise stage.

13.5 Oracle database guidelines

In this section we describe the Oracle database guidelines.

13.5.1 Oracle Enterprise stage column names

For each Oracle Enterprise stage, the following elements must be in effect:

- Rows of the database result set correspond to records of a parallel dataset.
- Columns of the database row correspond to columns of a DS record.
- The name and data type of each database column corresponds to a dataset name and data type using a predefined mapping of database data types to parallel data types.
- Names are translated exactly except when the Oracle source column name contains a character that DataStage does not support. In that case, two underscore characters replace the unsupported character.
- Both DS and Oracle support null values, and a null value in a database column is stored as an out-of-band NULL value in the DS column.

13.5.2 Oracle Enterprise stage data type mapping

Oracle table definitions must be imported into DataStage using **`orchdbutil`** to ensure accurate table definitions. This is particularly important for Oracle databases, which are not heavily typed. DataStage maps Oracle data types based on the rules given in the following table:

Table 13-5 Mapping data types

Oracle Data Type	DS Parallel Data Type
CHAR(n)	string[n] or ustring[n] a fixed-length string with length = n
DATE	timestamp
NUMBER	decimal[38,10]
NUMBER[p,s]	int32 if precision(p) < 11 and scale s = 0 decimal[p,s] if precision (p) >=11 or scale > 0
RAW(n)	not supported
VARCHAR(n)	string[max=n] or ustring[max=n] a variable-length string with maximum length = n

The maximum size of a DataStage record is limited to 32 K. If you attempt to read a record larger than 32 K, DataStage returns an error and aborts your job.

Important: Oracle data types that are not listed in the table cannot be used in the Oracle Enterprise stage, and generate an error at runtime.

13.5.3 Reading from Oracle in parallel

By default, the Oracle Enterprise stage reads sequentially from its source table or query. Setting the partition table option to the specified table enables parallel extracts from an Oracle source. The underlying Oracle table does not have to be partitioned for parallel reads in DS parallel jobs.

It is important to note that certain types of queries cannot run in parallel:

- Queries containing a GROUP BY clause that are also hash partitioned on the same field
- Queries performing a non-collocated join (an SQL JOIN between two tables that are not stored in the same partitions with the same partitioning strategy)

13.5.4 Oracle load options

When writing to an Oracle table (using Write Method = Load), the Oracle Enterprise stage uses the Parallel Direct Path Load method. When using this method, the Oracle stage cannot write to a table that has indexes (including indexes automatically generated by Primary Key constraints) on it unless you specify the Index Mode option (maintenance, rebuild).

Setting the environment variable $APT_ORACLE_LOAD_OPTIONS to OPTIONS (DIRECT=TRUE, PARALLEL=FALSE) allows loading of indexed tables without index maintenance. In this instance, the Oracle load is done sequentially.

The Upsert Write Method can be used to insert rows into a target Oracle table without bypassing indexes or constraints. To generate the SQL required by the Upsert method, the key columns must be identified using the check boxes in the column grid.

13.6 Sybase Enterprise guidelines

In this section we describe the Sybase Enterprise guidelines.

13.6.1 Sybase Enterprise stage column names

For each Sybase Enterprise stage, the following elements must be in effect:

- Rows of the database result set correspond to records of a parallel dataset.
- Columns of the database row correspond to columns of a DS record.
- The name and data type of each database column corresponds to a dataset name and data type using a predefined mapping of database data types to parallel data types.
- Names are translated exactly except when the Sybase source column name contains a character that DataStage does not support. In that case, two underscore characters replace the unsupported character.
- Both DS and Sybase support null values, and a null value in a database column is stored as an out-of-band NULL value in the DS column.

13.6.2 Sybase Enterprise stage data type mapping

Sybase databases are not supported by the orcdbutil utility. It is important to verify the correct Sybase to DS parallel data type mapping, as shown in Table 13-6.

Table 13-6 Data type mapping

Sybase Data Types	**DS Parallel Data Types**
BINARY(n)	raw(n)
BIT	int8
CHAR(n)	string[n] a fixed-length string with length n
DATE	date
DATETIME	timestamp
DEC[p,s] or DECIMAL[p,s]	decimal[p,s] where *p* is the precision and *s* is the scale
DOUBLE PRECISION or FLOAT	dfloat

Sybase Data Types	DS Parallel Data Types
INT or INTEGER	int32
MONEY	decimal[15,4]
NCHAR(n)	ustring[n] a fixed-length string with length n - only for ASE
NUMERIC[p,s]	decimal[p,s] where *p* is the precision and *s* is the scale
NVARCHAR(n,r)	ustring[max=n] a variable-length string with length n - only for ASE
REAL	sfloat
SERIAL	int32
SMALLDATETIME	timestamp
SMALLFLOAT	sfloat
SMALLINT	int16
SMALLMONEY	decimal[10,4]
TINYINT	int8
TIME	time
UNSIGNED INT	unit32
VARBINARY(n)	raw[max=n]
VARCHAR(n)	string[max=n] a variable-length string with maximum length n

Important: Sybase data types that are not listed in the table cannot be used in the Sybase Enterprise stage, and generates an error at runtime.

13.7 Existing Teradata database guidelines

In this section we describe the Teradata database guidelines.

13.7.1 Choosing the proper Teradata stage

In DataStage parallel jobs, the following stages depicted in Table can be used for reading from and writing to Teradata databases:

- Source Teradata stages
  - Teradata Enterprise
  - Teradata API
- Target Teradata stages
  - Teradata Enterprise
  - Teradata API
  - Teradata MultiLoad (MultiLoad option)
  - Teradata MultiLoad (TPump option)

For maximum performance of high-volume data flows, the native parallel Teradata Enterprise stage must be used. Teradata Enterprise uses the programming interface of the Teradata utilities FastExport (reads) and FastLoad (writes), and is subject to all these utilities' restrictions.

Note: Unlike the FastLoad utility, the Teradata Enterprise stage supports Append mode, inserting rows into an existing target table. This is done through a shadow terasync table.

Teradata has a system-wide limit to the number of concurrent database utilities. Each use of the Teradata Enterprise stages counts toward this limit.

13.7.2 Source Teradata stages

The Source Teradata stages are listed in Table 13-7.

Table 13-7 Source Teradata stages

Teradata stage	Stage type	Usage guidelines	Parallel read	Teradata utility limit
Teradata Enterprise	Native Parallel	► Reading a large number of rows in parallel ► Supports OPEN and CLOSE commands ► Subject to the limits of Teradata FastExport	Yes	applies
Teradata API	Plug-In	Reading a small number of rows sequentially	No	none

13.7.3 Target Teradata stages

The Target Teradata stages are listed in Table 13-8.

Table 13-8 Target Teradata stages

Teradata stage	Stage type	Usage guidelines	Parallel write	Teradata utility limit
Teradata Enterprise	Native Parallel	► Writing a large number of rows in parallel ► Supports OPEN and CLOSE commands ► Limited to INSERT (new table) or APPEND (existing table) ► Subject to the limits of Teradata FastLoad (but also supports APPEND) ► Locks the target table in exclusive mode	Yes	applies
Teradata MultiLoad (MultiLoad utility)	Plug-In	► Insert, Update, Delete, Upsert of moderate data volumes ► Locks the target tables in exclusive mode	No	applies
Teradata MultiLoad (TPump utility)	Plug-In	► Insert, Update, Delete, Upsert of small volumes of data in a large database ► Does not lock the target tables ► Must not be run in parallel, because each node and use counts toward system-wide Teradata utility limit	No	applies
Teradata API	Plug-In	► Insert, Update, Delete, Upsert of small volumes of data ► Allows concurrent writes (does not lock target) ► Slower than TPump for equivalent operations	Yes	none

13.7.4 Teradata Enterprise stage column names

For each Teradata Enterprise stage, the following elements must be in effect:

- Rows of the database result set correspond to records of a parallel dataset.
- Columns of the database row correspond to columns of a DS record.
- The name and data type of each database column corresponds to a DS dataset name and data type using a predefined mapping of database data types to parallel data types.
- Both DS and Teradata support null values, and a null value in a database column is stored as an out-of-band NULL value in the DS column.
- DataStage gives the same name to its columns as the Teradata column name. However, though DS column names can appear in either upper or lower case, Teradata column names appear only in upper case.

13.7.5 Teradata Enterprise stage data type mapping

Teradata databases are not supported by the orcdbutil utility. It is important to verify the correct Teradata to DS parallel data mapping, as shown in Table 13-9.

Table 13-9 Data type mapping

Teradata Data Type	**DS Parallel Data Type**
byte(n)	raw[n]
byteint	int8
char(n)	string[n]
date	date
decimal[p,s]	decimal[p,s] where *p* is the precision and S is the scale
double precision	dfloat
float	dfloat
graphic(n)	raw[max=n]
integer	int32
long varchar	string[max=n]
long vargraphic	raw[max=n]
numeric(p,s)	decimal[p,s]
real	Dfloat

Teradata Data Type	DS Parallel Data Type
smallint	int16
time	time
timestamp	timestamp
varbyte(n)	raw[max=n]
varchar(n)	string[max=n]
vargraphic(n)	raw[max=n]

Important: Teradata data types that are not listed in the table cannot be used in the Teradata Enterprise stage, and generate an error at runtime.

Aggregates and most arithmetic operators are not allowed in the SELECT clause of a Teradata Enterprise stage.

13.7.6 Specifying Teradata passwords with special characters

Teradata permits passwords with special characters and symbols. To specify a Teradata password that contains special characters, the password must be surrounded by an "escaped" single quote as shown, where *pa$$* is the example password: \'pa$$\'

13.7.7 Teradata Enterprise settings

In the Teradata Enterprise stage, the DB Options property specifies the connection string and connection properties in the following form:

```
user=username,password=password[,SessionsPerPlayer=nn][,RequestedSessio
ns=nn]
```

In this example, SesionsPerPlayer and RequestedSessions are optional connection parameters that are required when accessing large Teradata databases.

By default, RequestedSessions equals the maximum number of available sessions on the Teradata instance, but this can be set to a value between 1 and the database vprocs.

The SessionsPerPlayer option determines the number of connections each DataStage parallel player opens to Teradata. Indirectly, this determines the number of DataStage players, the number of UNIX processes, and the overall system resource requirements of the DataStage job. SessionsPerPlayer must be set in the following manner:

```
RequestedSessions = (sessions per player * the number of nodes *
players per node)
```

The default value for the SessionsPerPlayer suboption is 2.

Setting the SessionsPerPlayer too low on a large system can result in so many players that the job fails due to insufficient resources. In that case SessionsPerPlayer must be increased, and RequestedSessions must be decreased.

13.7.8 Improving Teradata Enterprise performance

Setting the environment variable, $APT_TERA_64K_BUFFERS, might significantly improve performance of Teradata Enterprise connections depending on network configuration. By default, Teradata Enterprise stage uses 32 K buffers. (Note that 64 K buffers must be enabled at the Teradata server level).

13.8 Netezza Enterprise stage

Starting with release 7.5.2, DataStage supports Netezza Performance Server (NPS) targets on AIX®, Linux® Red Hat, Linux SuSE, and Solaris platforms using the Netezza Enterprise stage. Netezza sources are supported through ODBC.

Documentation for the Netezza Enterprise stage is installed with the DataStage client, but is not referenced in the documentation bookshelf. You can find the following installation and developer documentation for the Netezza Enterprise stage in the `Docs/` subdirectory in the installed DataStage client directory:

- Connectivity Reference Guide for Netezza Servers
- Netezza Interface Library

13.8.1 Netezza write methods

The Netezza Enterprise stage is a write stage. The stage takes bulk data from a data source and writes that data to a specified destination table in NPS. You can write data to NPS using two available load methods, as listed in Table 13-10.

Table 13-10 Load methods

Load Method	Description	Requirements
Netezza Load	Uses NPS nzload utility to load directly to target NPS table	LOAD privileges for the target table; data in the source database is consistent, contains no default values, single byte characters only, uses a predefined format
External Table	Writes to an external table in NPS; data is then streamed into the target table	If the data source contains default values for table columns and uses variable format for data encoding such as UTF-8

13.8.2 Limitations of Netezza Write stage

The following list details the limitations of Netezza Write stage:

- DataStage design column order and data types must match target NPS table. During the write operation, the rows in the source database are mapped and streamed into the destination database. The columns of the source database are not mapped individually. You cannot interchange the order of the columns when writing to the destination database. The Netezza Interface write operator expects the column format in the destination database to be identical to the schema of the source.
- Netezza Performance Server supports only ASCII table names and column names. The following character sets are supported: v UTF-8 for NCHAR and NVARCHAR data types v LATIN9 for CHAR and VARCHAR data types
- If the Netezza Enterprise stage encounters bad records during the write operation, it aborts with an error. Bad input records are records that might be corrupted in the source database. To ignore the corrupted records and continue the write operation, you can specify the number of times to ignore the corrupt records before the write operation stops. The Netezza Enterprise stage ignores the bad records and continues to write data into NPS until the number of bad input records equals the number specified.

13.8.3 Netezza Enterprise error logs

You can view the error logs to identify errors that occur during any database operations, and view information about the success or failure of these operations. By default the log files are created in the `/tmp` directory. When writing data to the Netezza Performance Server by using the nzload method, the log files are created in the /tmp directory on the client computer.

The following names are used for the log files:

- `/tmp/database name.table name.nzlog`
- `/tmp/database name.table name.nzbad`

When writing data to the Netezza Performance Server by using the External Table method, the log files are created in the /tmp directory in the Netezza Performance Server. The following names are used for the log files:

- `/tmp/external table name.log`
- `/tmp/external table name.bad`

Note: The log files are appended every time an error occurs during the write operation.

14

Connector stage guidelines

This chapter presents usage guidelines for Connector stages. As a general guideline, new projects should give preference to Connector stages, and take advantage of existing Enterprise equivalents in specific cases where these have an edge in terms of performance.

The Connector library is constantly evolving, so the reader should always consult the latest documentation and release notes.

This document is not intended to serve as an installation and configuration guide for connectors, but instead focus on aspects related to data flow design.

14.1 Connectors and the connector framework

Until the advent of Connectors, DataStage (DS) relied on multiple types of stages for database access. There were two major groups:

- API (DS Server-based)
- Enterprise (parallel) stages

The Connector library provides a common framework for accessing external data sources in a reusable way across separate IS layers. The generic design of Connectors makes them independent of the specifics of the runtime environment in which they run.

The connectivity components are depicted in Figure 14-1.

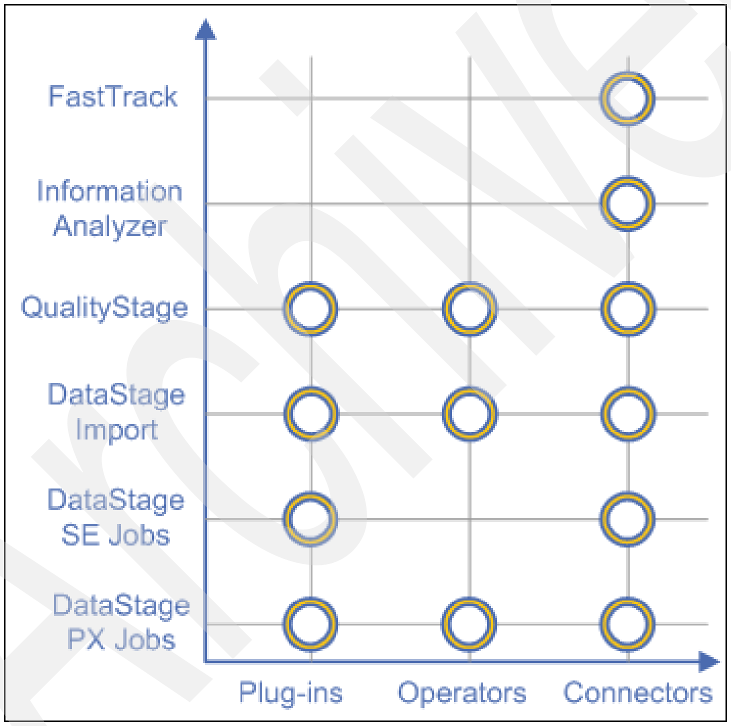


Figure 14-1 Comparing types of connectivity components

Connectivity modules are used towards the following ends:

- Discover and import metadata
- Extract data
- Load data

Information Server goes beyond job execution engines such as DS Server and parallel jobs, adding new tools such as Fast Track, Balanced Optimization and Information Analyzer. Figure 14-1 on page 222 presents a comparison of the places where each type can be used.

IS tools are implemented as J2EE applications running on top of WebSphere Application Server. These tools also have the need to access external data sources, discovering and importing metadata and loading/extracting data.

Plug-ins and Operators are components optimized for their corresponding execution environments (the parallel framework naturally being the fastest).

Connectors implement a common framework that can be made use of in DS Server and Parallel jobs, as well as Information Analyzer, FastTrack, and any future tools, using the same runtime library.

They provide consistent user experience. They have a similar look and feel, with minor GUI differences depending on the target type. The need for custom stage editors is eliminated. There is a common stage editor and metadata importer with rich capabilities.

There are a few important new concepts:

- LOB support
- Schema reconciliation
- Formalized metadata support

The connector framework is implemented in C++. However, the same runtime libraries can be used in J2EE applications such as Information Analyzer and FastTrack through JNI bridging. Design-time access to connectors is done through the Connector Access Service.

The APIs and layers are not exposed to the general public as development APIs. They are restricted to IBM, and enable the creation of new tools and components. One example of a component built on top of the Connector API is the Distributed Transaction stage (DTS), which supports the execution of distributed transactions across heterogeneous systems through the XA Protocol.

14.1.1 Connectors in parallel jobs

Figure 14-2 shows Connectors in the context of parallel jobs. PX Bridge is the API layer that allows parallel operators to access the Connector runtime.

A similar bridge for DS Server jobs is named SE Bridge. The same Connector runtime can be used in Java™ applications through a JNI layer.

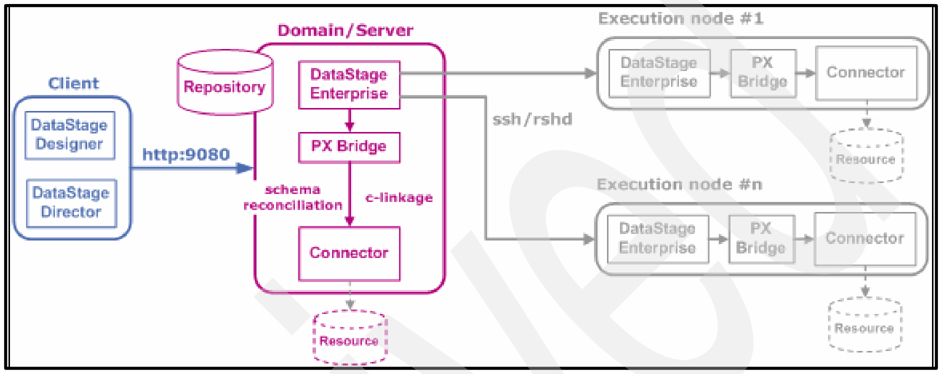


Figure 14-2 Connectors in parallel jobs

The job is designed in the DataStage Designer client application, which runs on Windows only. The job can be started and stopped from the DataStage Director client application. The clients access ISF server through internal http port 9080 on the WebSphere Application Server. The stage types, data schemas, and job definitions are saved to the common metadata repository.

When the job starts, the DataStage Enterprise engine starts a PX Bridge Operator module for each Connector stage in the job. The PX Bridge acts as the interface between the DataStage Enterprise framework and Connectors. The Connectors are unaware of the type of the runtime environment.

The PX Bridge performs schema reconciliation on the server on which the job is initiated. The schema reconciliation is the negotiation process in which the PX Bridge serves as the mediator between the framework that offers the schema definition provided by the job design (the schema defined on the Connector stage's link) and the Connector that offers the actual schema definition as defined in the data resource to which it connects. During this process, the attempt is made to agree on the schema definition to use in the job. The differences that might be acceptable are as follows:

- Separate column data types for which data can be converted without data loss
- Unused columns that can be dropped or ignored

The job might be configured to run on multiple execution nodes in parallel. The DataStage Enterprise engine runs in distributed fashion. The communication and synchronization among the running nodes is performed using the secure shell and secure shell daemon mechanism. On each of the execution nodes Connector instances are reading or writing portions of the data in parallel. This provides for a high data throughput, especially when accessing resources that support parallel and distributed data access, for example a DB2 database with a database partitioning feature (DPF).

14.1.2 Large object (LOB) support

LOB data of any size can be moved from a source to target in a piecemeal fashion. The target connector interacts with the source connector to read and write the LOB fragments. This is depicted in Figure 14-3.

Instead of propagating the LOB itself (unnecessarily moving large amounts of data across virtual links and operators all the way to the final target stage), the job can propagate an LOB reference.

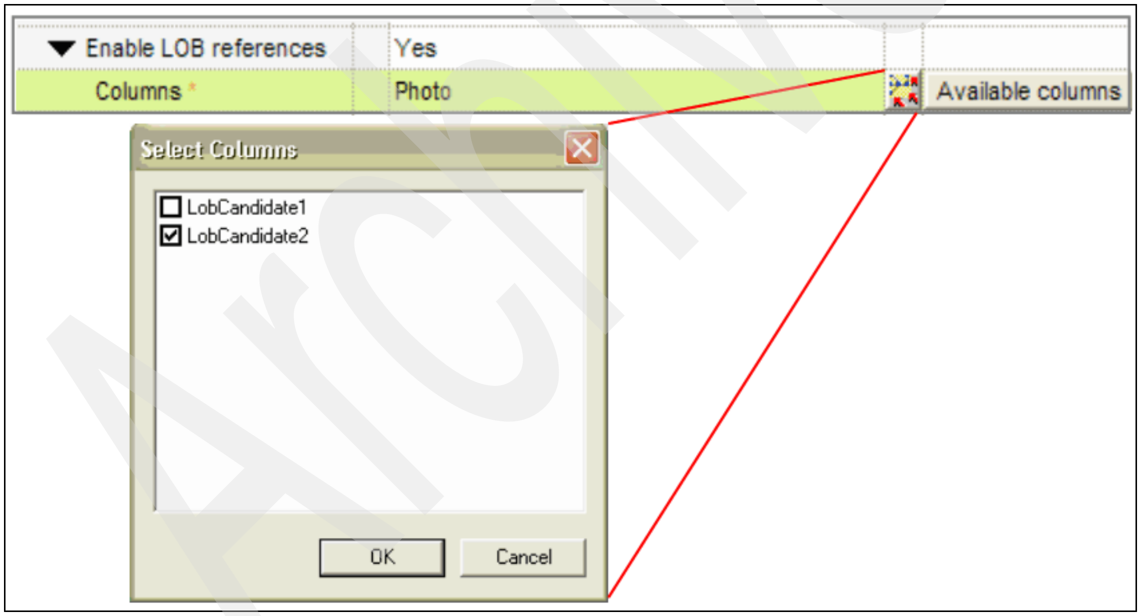

Enable LOB references: Yes
Columns *: Photo

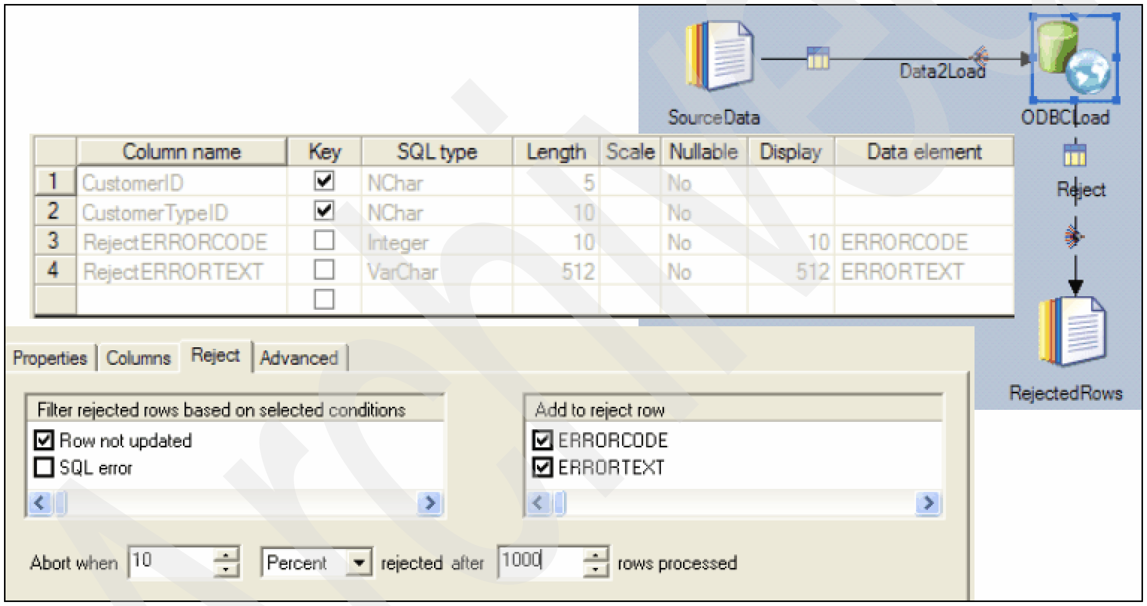
Figure 14-3 LOB Support in Connector stages

14.1.3 Reject Links

Reject functionality enhancements are as follows:

- Optionally append error code and error message data to the failing row data
- Conditionally filter which rows are rejected
- Ability to fail a job based on an absolute or percentage number of failures

The reject links in Connector stages is depicted in Figure 14-4. One major implication of this reject functionality is the ability to output all records, regardless of the presence of errors. This simplifies the development of real-time ISD jobs, which need to synchronize the output from database stages with the InfoSphere Information Services Director (ISD) Output stage. This is discussed in detail in Chapter 16, “Real-time data flow design” on page 293.



	Column name	Key	SQL type	Length	Scale	Nullable	Display	Data element
1	CustomerID	✓	NChar	5		No		
2	CustomerTypeID	✓	NChar	10		No		
3	RejectERRORCODE	☐	Integer	10		No	10	ERRORCODE
4	RejectERRORTEXT	☐	VarChar	512		No	512	ERRORTEXT
		☐						

Filter rejected rows based on selected conditions
☑ Row not updated
☐ SQL error

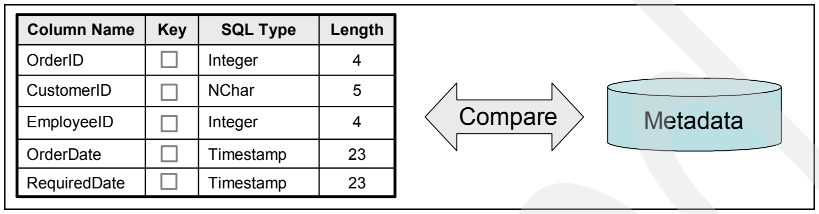
Add to reject row
☑ ERRORCODE
☑ ERRORTEXT

Abort when 10 Percent rejected after 1000 rows processed

Figure 14-4 Reject links in Connector stages

14.1.4 Schema reconciliation

Schema reconciliation is a mechanism to resolve discrepancies between design-time and actual metadata. The Connector compares design-time and propagated metadata with external metadata on a field-by-field basis. This is depicted in Figure 14-5.



Column Name	Key	SQL Type	Length
OrderID	☐	Integer	4
CustomerID	☐	NChar	5
EmployeeID	☐	Integer	4
OrderDate	☐	Timestamp	23
RequiredDate	☐	Timestamp	23

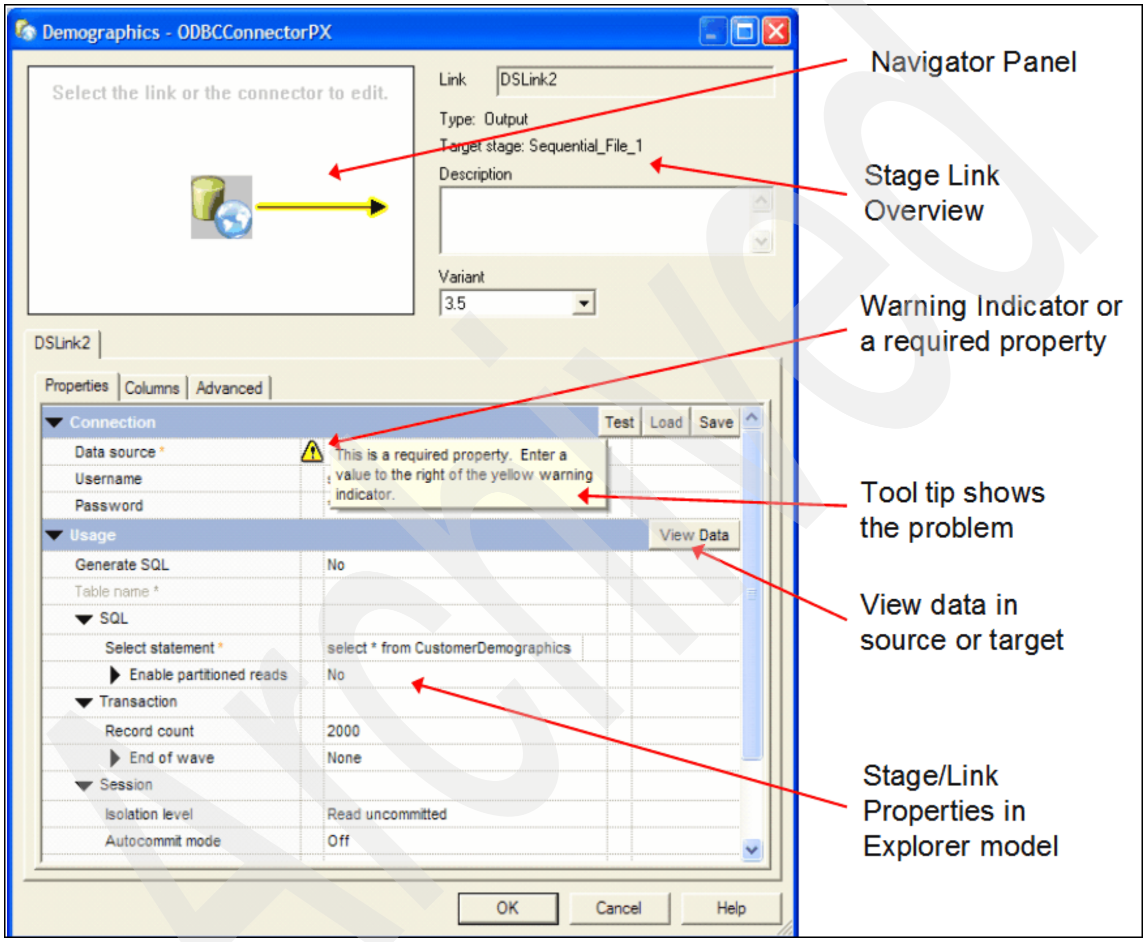
Figure 14-5 Schema reconciliation

The goal is to reduce unpredictable problems that can stem from non-matching metadata in the following ways:

- Identifying matching fields
- Performing data conversions for compatible data types
- Failing jobs that cannot be automatically corrected

14.1.5 Stage editor concepts

Each DS Server API and Enterprise stage has its own custom editor. For Connectors, there is a Common Stage Editor that makes the design experience much the same across all database connector types. The Connector Stage Editor is depicted in Figure 14-6.


Navigator Panel
Stage Link Overview
Warning Indicator or a required property
Tool tip shows the problem
View data in source or target
Stage/Link Properties in Explorer model
Demographics - ODBCConnectorPX
Select the link or the connector to edit.
DSLink2
Properties
Columns
Advanced
Test
Load
Save
View Data
OK
Cancel
Help

Link: DSLink2
Type: Output
Target stage: Sequential_File_1
Description
Variant: 3.5

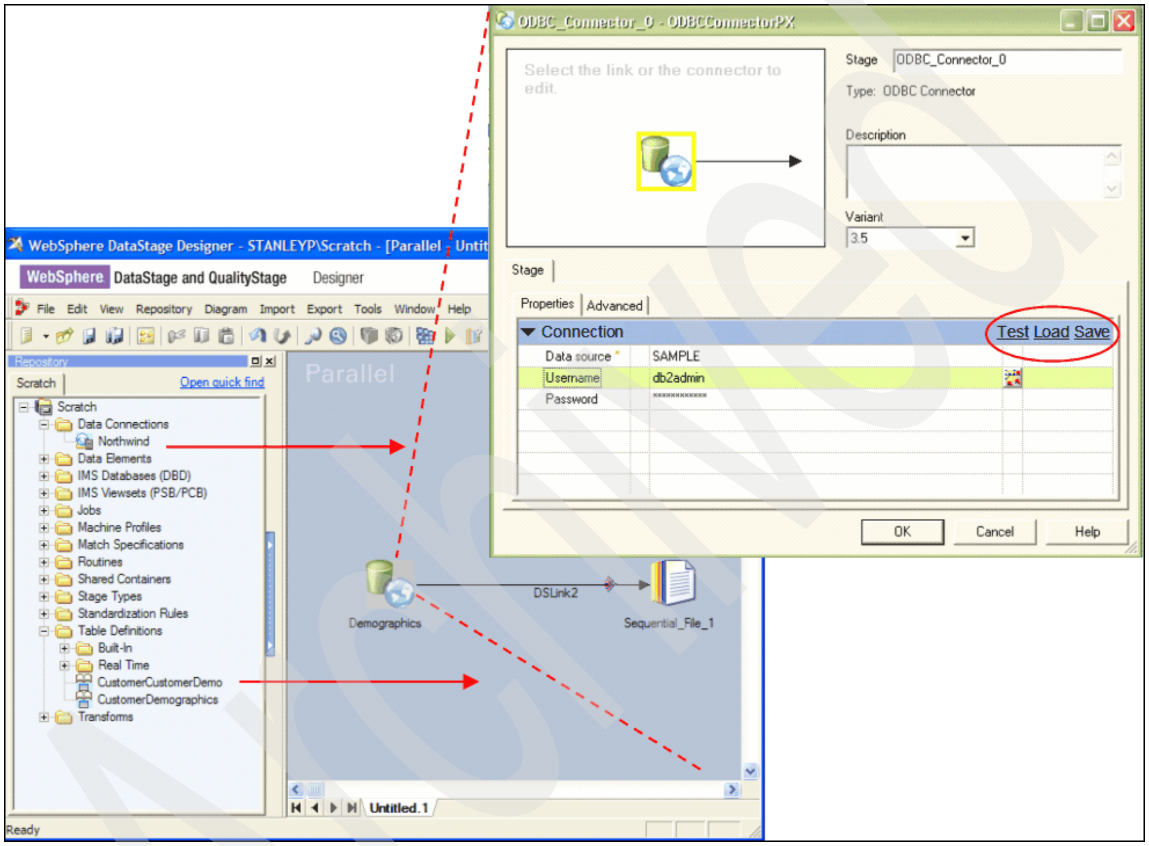
Connection	
Data source *	This is a required property. Enter a value to the right of the yellow warning indicator.
Username	
Password	
Usage	
Generate SQL	No
Table name *	
SQL	
Select statement *	select * from CustomerDemographics
Enable partitioned reads	No
Transaction	
Record count	2000
End of wave	None
Session	
Isolation level	Read uncommitted
Autocommit mode	Off

Figure 14-6 Connector Stage Editor

The Common Stage Editor for Connectors presents properties as a hierarchical, contextual view. The properties, their list of valid values and their hierarchy are defined by Connector definition files. There is one definition for each Connector type. Definition describes the elements to be displayed. The Common Stage Editor renders the presentation and drives the interaction with the user.

14.1.6 Connection objects

Connection objects encapsulate information to access external data sources. Data such as user name, password and database name might be kept at a central location and reused across multiple stages. This is depicted in Figure 14-7.



Stage: ODBC_Connector_0
Type: ODBC Connector
Description:
Variant: 3.5

Connection

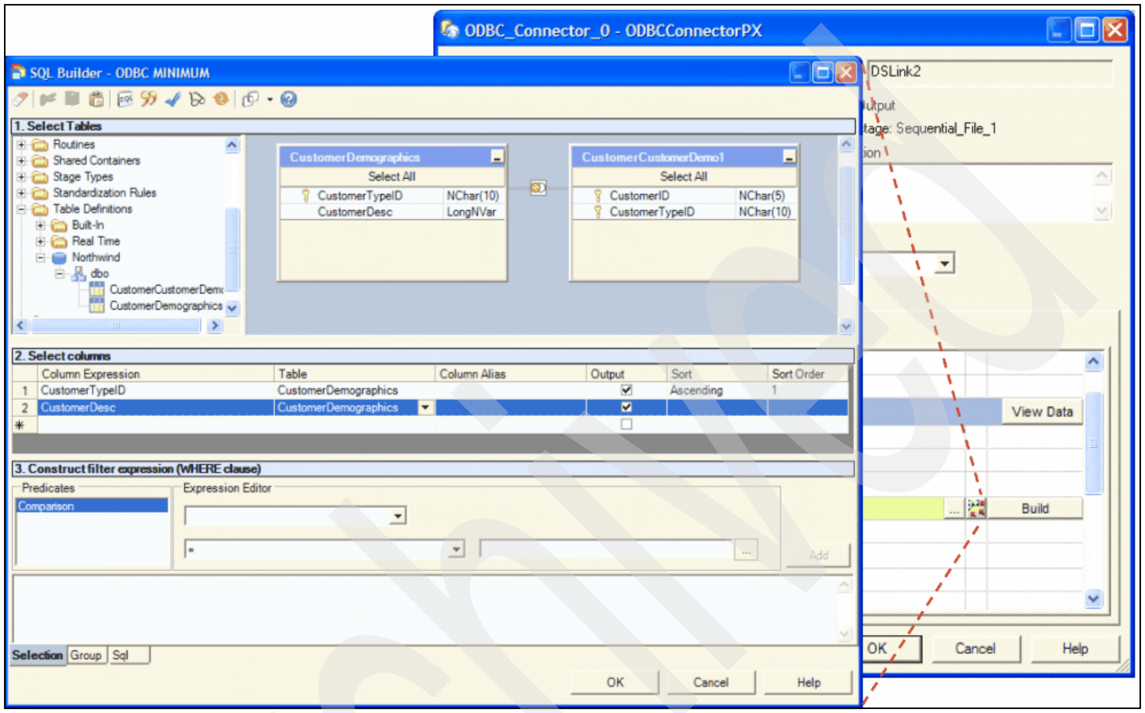
Data source *	SAMPLE
Username	db2admin
Password	************

Figure 14-7 Dragging connection objects

Connection objects might be dragged onto existing Connector stages, or onto the canvas. When dragged into the canvas, a new Connector stage is created.

14.1.7 SQL Builder

Figure 14-8 presents a picture of the SQL Builder. The SQL Builder is accessible by clicking an icon adjacent to the User-Defined SQL field.



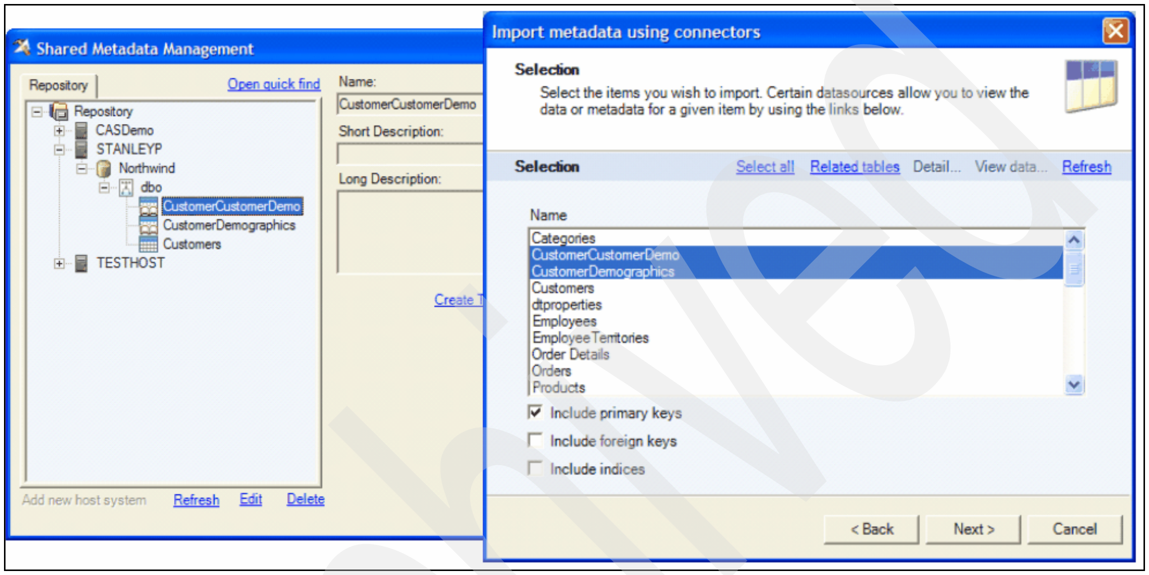
	Column Expression	Table	Column Alias	Output	Sort	Sort Order
1	CustomerTypeID	CustomerDemographics		✓	Ascending	1
2	CustomerDesc	CustomerDemographics		✓		
*				☐		

Figure 14-8 The SQL builder

14.1.8 Metadata importation

Metadata imported through connectors creates shared metadata as well as DataStage table definitions.

Table definitions are associated with the data connection used to import the metadata. This is illustrated in Figure 14-9.


Shared Metadata Management

Repository — Open quick find

- Repository
  - CASDemo
  - STANLEYP
    - Northwind
      - dbo
        - CustomerCustomerDemo
        - CustomerDemographics
        - Customers
  - TESTHOST

Name: CustomerCustomerDemo

Short Description:

Long Description:

Add new host system · Refresh · Edit · Delete

Import metadata using connectors

Selection

Select the items you wish to import. Certain datasources allow you to view the data or metadata for a given item by using the links below.

Selection — Select all · Related tables · Detail... · View data... · Refresh

Name
- Categories
- CustomerCustomerDemo
- CustomerDemographics
- Customers
- dtproperties
- Employees
- EmployeeTerritories
- Order Details
- Orders
- Products

[✓] Include primary keys
[ ] Include foreign keys
[ ] Include indices

< Back · Next > · Cancel

Figure 14-9 Metadata importation

Certain Connector stages take advantage of the Dynamic Metadata Interface (DMDI). DMDI is an interface provided by the connector framework employed by connectors to perform metadata importation to populate properties and column definitions. The following stages use DMDI:

- Essbase
- Salesforce
- XML Pack

These connectors use DMDI to display a GUI to browse metadata objects, then to import them and to set stage properties. The DMDI is launched from the stage editor.

14.2 ODBC Connector

The ODBC Connector provides access to database resources through the industry-standard ODBC interface. It is compliant with the ODBC standard version 3.5, level 3. ODBC Connector accesses databases through the user-defined ODBC data source name (DSN) definitions, which can be created for a variety of ODBC drivers. On Windows, DSN definitions are created through the built-in ODBC Driver Manager application and are stored in the Windows registry. On UNIX systems, ODBC DSN definitions are provided in a special ODBC initialization file pointed to by the ODBCINI environment variable.

The Connector supports three execution contexts for DataStage jobs:

- Source
- Target
- Lookup

When used in the source context, the Connector extracts data from the database by executing SELECT SQL statements. It provides this data to other stages in the job for transformation and load functions. When used in the target context, the Connector loads, updates, and deletes data in the database by executing INSERT, UPDATE, and DELETE SQL statements. The lookup context is similar to the source context, with the difference that the SELECT SQL statements are parameterized. The parameter values are provided dynamically on the input link by the other stages in the job.

The ODBC Connector supports a special type of link called *reject link*. The Connector can be configured to direct the data that it cannot process to the reject link, from which it can be sent to any stage in the job (For example, the Sequential File stage). The rejected data can later be inspected and re-processed. Reject links provide for the option to continue running the job when an error is detected, instead of interrupting the job at that moment. As shown in Figure 14-10 on page 233, reject links are supported in lookup and target context.

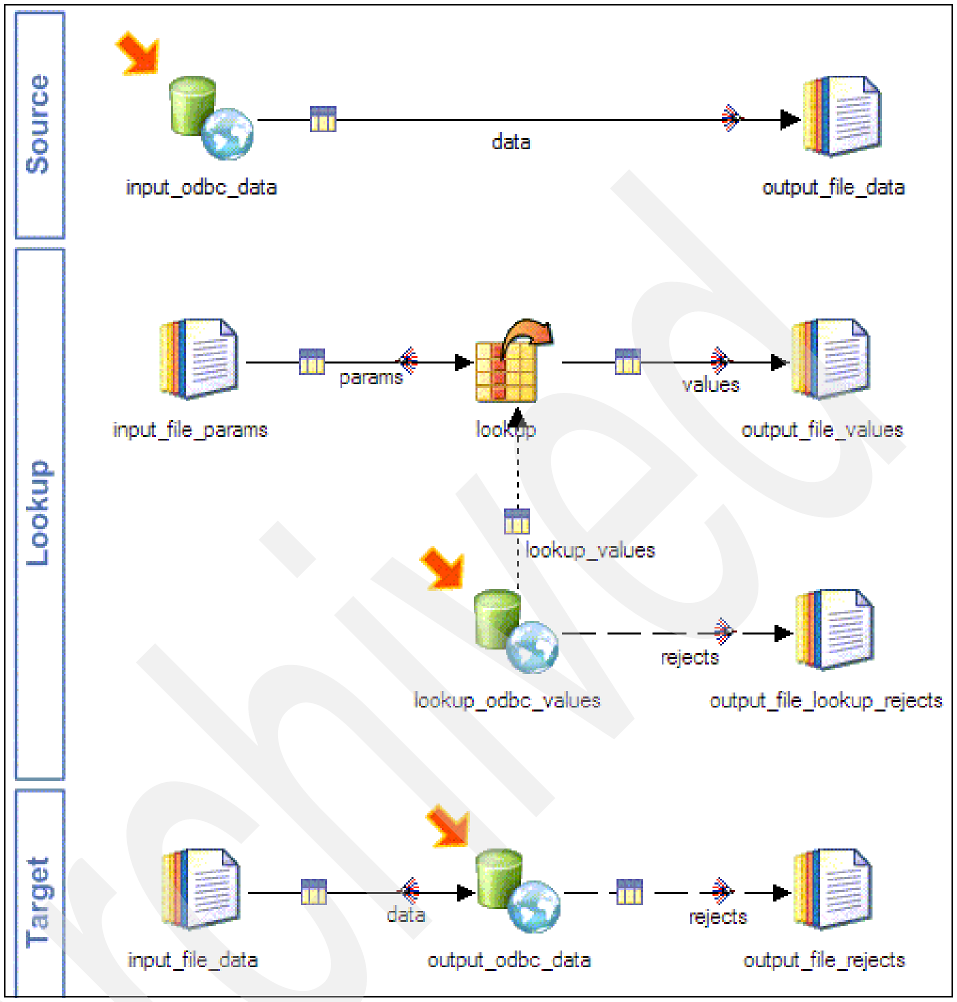


Figure 14-10 ODBC connector usage patterns

The Connector allows the option of passing LOBs by reference, rather than by extracting the data and passing it inline into the job flow. When configured to pass LOB values by reference, the ODBC Connector assembles a special block of data, called a *locator* or *reference*, that it passes into the job dataflow. Other Connectors are placed at the end of the job dataflow. When the LOB locator arrives at the other Connector stage, the Connector framework initiates the retrieval of the actual ODBC data represented by the reference and provides the data to the target Connector so that it can be loaded into the represented resource. This way it is possible to move LOB data from one data resource to another where the size of the data is measured in megabytes and gigabytes, without having to move the actual data through the job. The drawback is that the data passed this way cannot be altered by the intermediate stages in the job.

The Connector supports array insert operations in target context. The Connector buffers the specified number of input rows before inserting them to the database in a single operation. This provides for better performance when inserting large numbers of rows.

The Connector user uses the SQL Builder tool to design the SQL statements. The SQL Builder tool is a graphical tool that enables construction of SQL statements in a drag and drop fashion.

The Connector provides a mechanism for executing special sets of SQL statements before processing the data. These statements are typically used to initialize the database objects (for example to create a new table or truncate the existing table before inserting data into the table).

The Connector supports metadata operations, such as the discovery of database objects and describing these objects.

The IBM Information Server comes with a set of branded ODBC drivers that are ready for use by the ODBC Connector. On Windows, the built-in driver manager is used. On UNIX, a driver manager is included with the IBM Information Server installation.

14.3 WebSphere MQ Connector

WebSphere MQ Connector provides access to message queues in the WebSphere MQ enterprise messaging system. It provides the following types of support:

- WebSphere MQ versions 5.3 and 6.0, and WebSphere Message Broker 6.0 for the publish/subscribe mode of work.
- MQ client and MQ server connection modes. The choice can be made dynamically through the special connection property in the Connector's stage editor.
- Filtering of messages based on various combinations of message header fields. Complex filtering conditions might be specified.
- Synchronous (request/reply) messaging. This scenario is configured by defining both input link (for request messages) and output link (for reply messages) for the Connector's stage.

- Publish/Subscribe mode of work. Both the WebSphere MQ broker (with MQRFH command messages) and WebSphere Message Broker (with MQRFH2 command messages) are supported. The Connector stage can be configured to run as a publisher or as a subscriber. Dynamic registration and deregistration of publisher/subscriber is supported.
- MQ dynamic queues, name lists, transmission queues and shared cluster queues for remote queue messaging
- Designated error queues and standard reject links.

The MQ Connector usage patterns are illustrated in Figure 14-11.

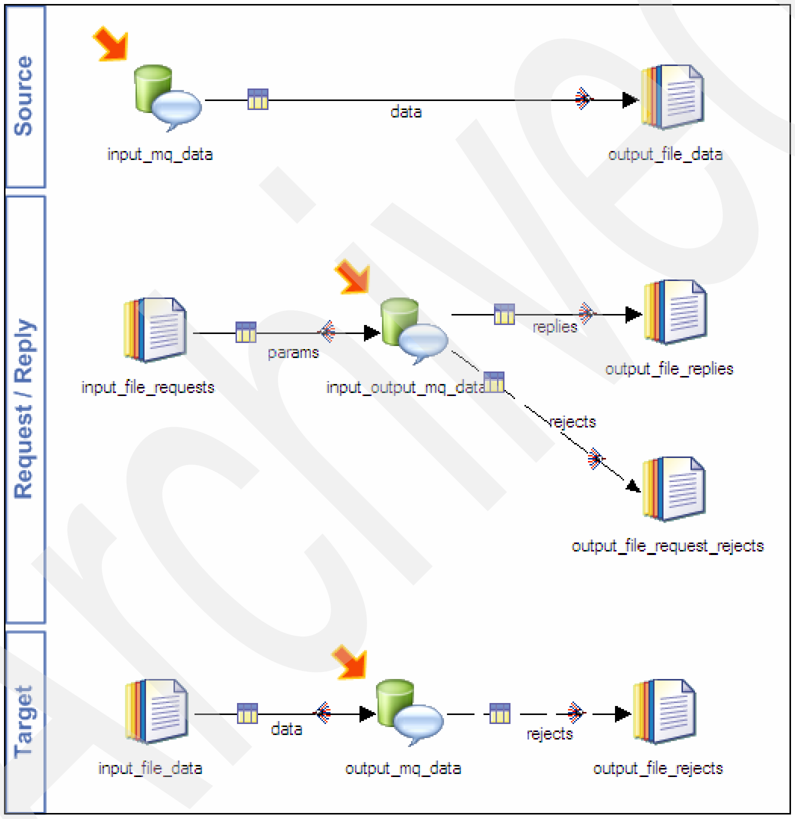


Figure 14-11 The MQConnector usage patterns

14.4 Teradata Connector

The Teradata Connector includes the following features of the ODBC Connector:

- Source, target and lookup context
- Reject links
- Passing LOBs by reference
- Arrays
- SQL Builder
- Pre/post run statements
- Metadata import

It supports Teradata server versions V2R6.1 and V2R6.2 and Teradata client TTU versions V8.1 and V8.2. The Connector uses CLIv2 API for immediate operations (SELECT, INSERT, UPDATE, DELETE) and Parallel Transporter Direct API (formerly TEL-API) for bulk load and bulk extract operations.

Parallel bulk load is supported through LOAD, UPDATE, and STREAM operators in Parallel Transporter. This corresponds to the functionality provided by the FastLoad, MultiLoad, and TPump Teradata utilities, respectively. When the UPDATE operator is used it supports the option for deleting rows of data (MultiLoad delete task).

Parallel bulk export is supported through the EXPORT operator in Parallel Transporter. This corresponds to the functionality provided by the FastExport Teradata utility. The Connector persists the bulk-load progress state and provides sophisticated support for restarting the failed bulk-load operations. The Connector uses a designated database table for synchronization of distributed Connector instances in the parallel bulk-load.

A limited support for stored procedures and macros is also available.

The Teradata usage patterns are illustrated in Figure 14-12 on page 237.

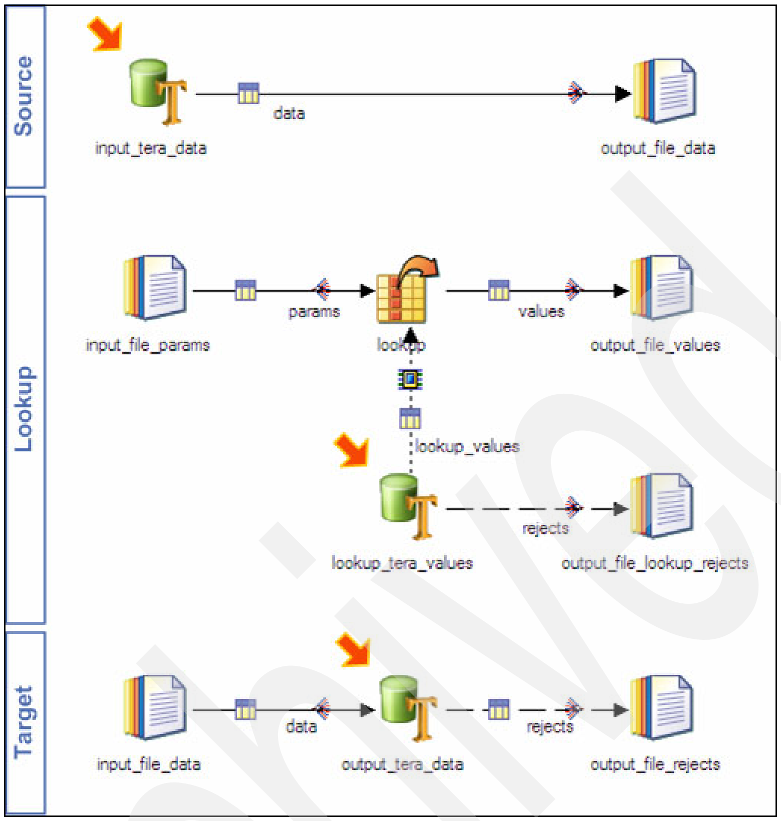


Figure 14-12 Teradata Connector usage patterns

14.4.1 Teradata Connector advantages

The following list details the Teradata Connector advantages:

- Parallel MultiLoad capability including MPP configurations
- Parallel immediate lookups and writes
- Array support for better performance of immediate writes
- Reject link support for lookups and writes on DS Enterprise Edition
- Reject link support for bulk loads
- Cursor lookups (lookups that return more than one row)
- Restart capability for parallel bulk loads
- MultiLoad delete task support
- Support for BLOB and CLOB data types
- Reject link support for missing UPDATE or DELETE rows
- Error message and row count feedback for immediate lookups/writes
- Parallel synchronization table

14.4.2 Parallel Synchronization Table

The following list details the Parallel Synchronization Table properties:

- Used for coordination of player processes in parallel mode
- Now optional, connector runs sequentially if not specified
- Can be used for logging of execution statistics
- Connector stage can use its own sync table or share it
- Primary key is SyncID, PartitionNo, StartTime
- Each player updates its own row, no lock contention
- Management properties for dropping, deleting rows

14.4.3 Parallel Transport operators

Table 14-1 shows a list of advantages and disadvantages of the Parallel Transporter operators.

Table 14-1 Parallel Transporter operators

Operator	Equivalent Utility	Advantages	Disadvantages
Export	FastExport	Fastest export method.	Uses utility slot, No single-AMP SELECTs.
Load	FastLoad	Fastest load method.	Uses utility slot, INSERT only, Locks table, No views, No secondary indexes.
Update	MultiLoad	INSERT, UPDATE, DELETE, Views, Non-unique secondary indexes.	Uses utility slot, Locks table, No unique secondary indexes, Table inaccessible on abort.
Stream	TPump	INSERT, UPDATE, DELETE, Views, Secondary indexes, No utility slot, No table lock.	Slower than UPDATE operator.

14.4.4 Cleanup after an aborted load or update

Take the following steps for cleanup after an aborted load or update operation.

1. DROP TABLE TableName_LOG;
2. DROP TABLE TableName_ET;
3. DROP TABLE TableName_UV;
4. DROP TABLE TableName_WT;
5. RELEASE MLOAD TableName; (Update operator only)

14.4.5 Environment variables for debugging job execution

Table 14-2 shows the environment variables for debugging job execution.

Table 14-2 Environment variables for debugging

CC_MSG_LEVEL=2	Turns on debugging output in the Director job log, displays property values, generated SQL
CC_MSG_LEVEL=1	Full trace of the connector method calls

CC_TERA_DEBUG=1
CC_MSG_LEVEL=2
	Dump of the CLIv2 structures after each call to the Teradata client

CC_TERA_DEBUG=4	Turns on the Parallel Transporter's trace output which is written to a filename beginning with "TD_TRACE_OUTPUT" in the project directory

14.4.6 Comparison with existing Teradata stages

The following list details comparisons with Teradata stages:

- Limited support for stored procedures and macros, but the Stored Procedure plug-in is still better suited for it.
- No more utilities, named pipes, control scripts, or report files.
- Interface with the Parallel Transporter is through a direct call-level API.
- Error messages are reported in the DataStage Director log.
- MultiLoad plug-in jobs that use advanced custom script features cannot be migrated to use the Teradata Connector.
- Number of players is determined by the PX Engine config file.

In Figure 14-13 we illustrate the Teradata stages and their relation to Teradata Client APIs.

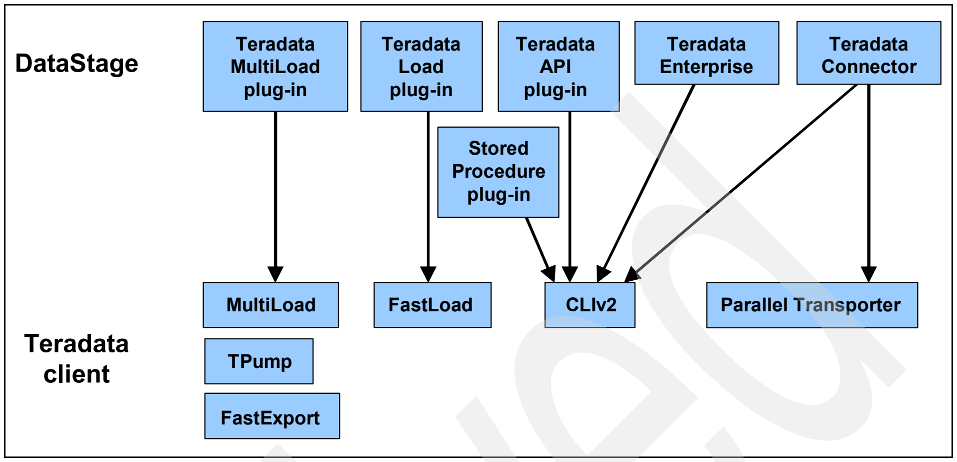


Figure 14-13 Teradata stages and their relation to Teradata client APIs

In Table 14-3, we present a feature comparison among all Teradata stage types.

Table 14-3 Teradata features for stage types

Feature / Requirement	**Teradata Connector**	**Teradata Operator**	**TDMLoad Plug-in**	**Teradata Load Plug-in**	**Teradata API Plug-in**
Read, bulk Export	YES	Export only	Export only	No	Read only
Write, bulk Load	YES	Load only	Load only	Load only	Write only
Update/Upsert	YES	No	YES	No	Yes
Sparse Lookup	YES	No	No	No	YES
Array Support	YES	Limited support (Fixed sizes: 32/64 K)	Limited support (Fixed sizes: 32/64 K)	Limited support (Fixed sizes: 32/64 K)	NO
Checkpoint support	YES	No	Yes	Yes	No
EOW support	YES	No	Yes	No	No
Reject Link support	YES	No	No	No	No
Multiple-input-Links	YES	No	No	No	No
DTS-nonXA support	YES	No	No	No	No

New TD Data-Types: Decimal(38), BIGINT	Supported	Not supported	Not supported	Not supported	Not supported
Teradata basic client (TTU)	Required	Required	Required	Required	Required
Teradata PT client (TPT)	Required	Not Required	Not Required	Not Required	Not Required
stage Execution Mode (Sequential / Parallel)	Both	Both	Sequential	Sequential	Sequential
Canvas (PX / Server)	Both	PX	Both	Both	Both

In Table 14-4 we present a mapping between the Teradata Enterprise environment variables and Connector functionality.

Table 14-4 Mapping of variables to functionality

Environment Variable	**Setting**	**Description**	**Equivalent in TDCC**
$APT_TERA_SYNC_DATABASE	[name]	Starting with v7, specifies the database used for the terasync table.	stage Property: “sync database”
$APT_TERA_SYNC_USER	[user]	Starting with v7, specifies the user that creates and writes to the terasync table.	stage Property: “sync user”
$APT_TER_SYNC_PASSWORD	[password]	Specifies the password for the user identified by $APT_TERA_SYNC_USER.	stage Property: “sync password”
$APT_TERA_64K_BUFFERS		Enables 64K buffer transfers (32K is the default). Might improve performance depending on network configuration.	Automatic (checks supported values from DBS configuration and sets the buffer-size to (32K/64K or 1M).
$APT_TERA_NO_ERR_CLEANUP		This environment variable is not recommended for general use. When set, this environment variable might assist in job debugging by preventing the removal of error tables and partially written target table.	stage Property (under “Bulk Access” category): “Cleanup Mode” (with drop-down options: keep/drop).

$APT_TERA_NO_PERM_CHECKS		Disables permission checking on Teradata system tables that must be readable during the TeraData Enterprise load process. This can be used to improve the startup time of the load.	Default behavior

14.5 DB2 Connector

The DB2 Connector includes the following features of the ODBC Connector:

- Source
- Target and lookup context
- Reject links
- Passing LOBs by reference
- Arrays
- SQL Builder
- Pre/post run statements
- Metadata import
- Supports DB2 version V9.1

The Connector is based on the CLI client interface. It can connect to any database cataloged on the DB2 client. The DB2 client must be collocated with the Connector, but the actual database might be local or remote to the Connector.

Separate the sets of connection properties for the job setup phase (conductor) and execution phase (player nodes), so the same database might be cataloged differently on conductor and player nodes.

The Connector provides support for the following tasks:

- Specifying DB2 instance dynamically (through connection properties), which overrides the default environment settings (DB2INSTANCE environment variable).
- XML data type in DB2 V9.1.
- DB2 DPF. A job with a DB2 Connector target stage might be configured to assign on execution player node with each DB2 partition, and to write data to the partitioned database in parallel, providing dramatic performance improvement over sending the data to the same partition node and forcing DB2 to redirect data to corresponding partitions.
- DB2 bulk load functionality. The invocation of bulk load is done through the CLI interface. Parallel bulk load is also supported for DB2 with DPF.

In Figure 14-14 we illustrate the DB2 Connector usage patterns.

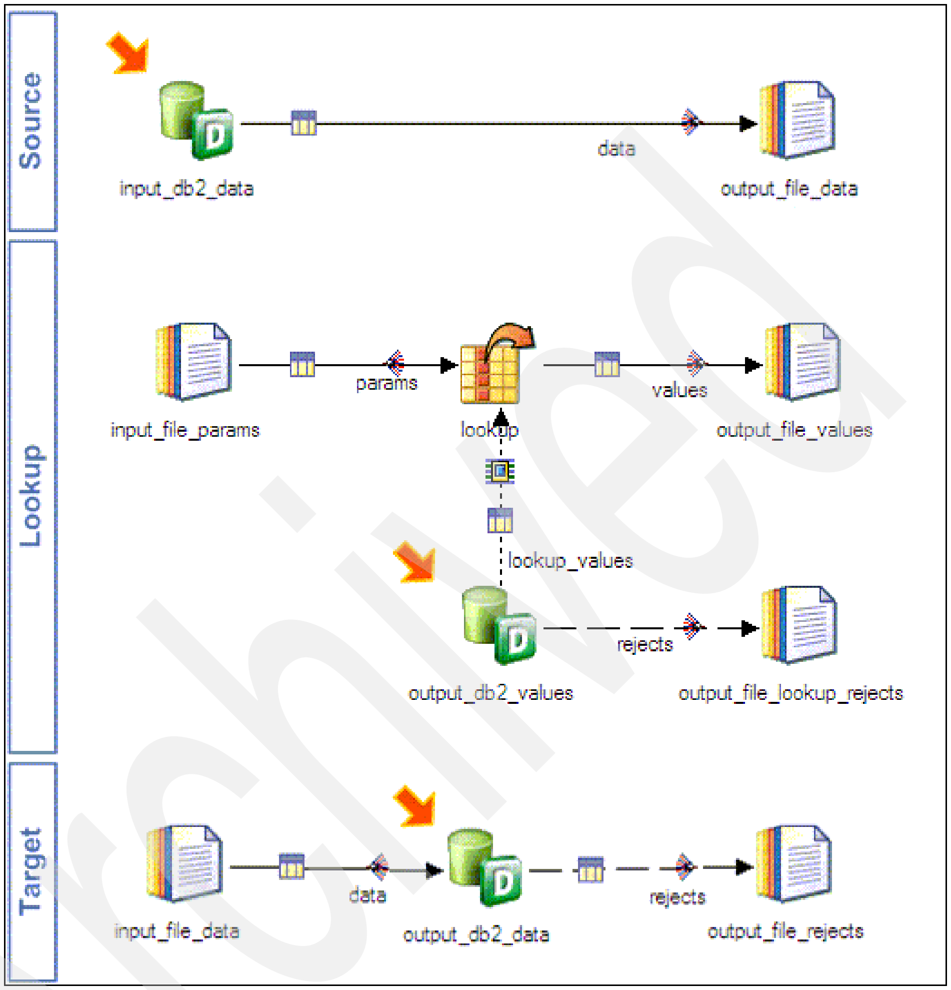


Figure 14-14 DB2 connector usage patterns

14.5.1 New features

In terms of functionality, the DB2 Connector offers more capabilities than all of the existing stages. Specifically it provides support for the following elements:

- XML data type
- LOBs, with data passed either inline, or by a reference mechanism. The latter allows for LOB data of any size to be moved from a source to a target.
- Client-server access to DB2 server. This overcomes a limitation with the EE stage that can only be used in homogenous environments where DataStage and DB2 are on identical (or the same) servers.
- Options to control bulk load than EE Operator.
- Design time capabilities, such as metadata import to the common model, and enumeration of server-side information.

14.5.2 Using rejects with user-defined SQL

In this section we describe using rejects with user-defined SQL (UserSQL).

UserSQL without the reject link

The following list describes the use of UserSQL without the reject link:

- All statements in the UserSQL property are either passed for all of the input records in the current batch (as specified by the Array size property) or none. In other words events in previous statements do not control the number of records passed to the statements that follow.
- If FailOnError=Yes, the first statement that fails causes the job to fail and the current transaction, as depicted by the Record Count property, is rolled back. No more statements from the UserSQL property are executed after that. For example, if there are three statements in the property, and the second one fails, it means that the first one has already executed, and the third one is not executed. None of the statements have its work committed because of the error.
- If FailOnError=No, all statements still get all the records but any statement errors are ignored and statements continue to be executed. For example, if there are three statements in the UserSQL property and the second one fails, all three are executed and any successful rows are committed. The failed rows are ignored.

UserSQL with the reject link

The following list describes the use of UserSQL with the reject link:

- All statements in the UserSQL property are either passed all of the input records in the current batch (as specified by the Array size property) or none. In other words, events in previous statements do not control the number of records passed to the statements that follow.
- All the rows in each batch (as specified by the Array size property) are either successfully consumed by all statements in the UserSQL property, or are rejected as a whole. This is important to preserve the integrity of the records processed by multiple statements that are expected to be atomic and committed in the same transaction. In other words, the connector tries to eliminate the possibility of having each statement successfully consume a set of rows.
- If any of the rows in any statement in the UserSQL property are not consumed successfully, the processing of the current batch is aborted and the whole batch of records is sent to the reject link. The statements that follow the failed statement are not executed with the current batch. The processing resumes with the next batch.
- To preserve the consistency of records in each batch, the connector forces a commit after every successful batch and forces a rollback after every failed batch. This means the connector overrides the transactional behavior specified by the Record Count property.

14.5.3 Using alternate conductor setting

When the connector is used on the parallel canvas, the engine spawns several processes when the job runs. The main process is called the conductor. The other processes are called players. These processes can run on the same machine or on their own separate machines as defined in the PX configuration file (`default.apt`). In the conductor process the connector performs metadata discovery (describes the table that is later read or written to) and negotiates the execution options with the engine. Later, in each player, the data is actually read or written according to the setup performed in the conductor.

The following list details important facts and restrictions: in using alternate conductor settings:

- There is always only one conductor process per job execution regardless of the number of stages used in the job or the number of nodes defined in the configuration file.
- Only one DB2 client library can be loaded in a process for the life of the process. This means that the conductor process loads the client library specified by the first DB2 stage in the job that it loads.
- Only one DB2 instance can be accessed from one process for the duration of the process.

The following list describes common scenarios when the alternate conductor setting might be needed:

- Scenario 1: All nodes run on the same physical machine

  If there are DB2 client versions installed on the system or if there multiple instances defined, it is important that all DB2 stages in the job specify the same conductor connection parameters, otherwise the job fails because of the aforementioned facts/restrictions. Using the alternate conductor setting helps achieve this.

- Scenario 2: Each node runs on a separate machine

  In this case the conductor might not run on the same machine as the player processes. The same remote database that the connector is trying to access might be cataloged differently on each node, and it might appear under a different instance or database than it does to the players. Using the alternate conductor setting allows you to use a separate instance/database for the conductor process.

14.5.4 Comparison with existing DB2 stages

In determining the placement of the DB2 Connector in DataStage when compared to the three other DB2 stages (DB2 EE Operator, DB2 API plug-in, and DB2 Load plug-in) you must consider both functionality and performance.

In terms of functionality, the DB2 Connector offers more capabilities than all of the existing stages, with new features as listed in 14.5.1, “New features” on page 244. There are four DB2 stages available to DataStage users, as listed in Table 14-5 on page 247.

Table 14-5 DB2 stages available to users

Name	Technology	Where it Runs	Capabilities
DB2 API	Plug-in	Server, Parallel	Query, Upsert, Lookup
DB2 Load	Plug-in	Server, Parallel	Load
DB2 EE	Operator	Parallel	Query, Upsert, Lookup, Load
DB2 Connector	Connector	Server, Parallel	Query, Upsert, Lookup, Load

The plug-in stages collectively offer approximately the same functionality as both the operator and connector. However, on the parallel canvas the two plug-in stages perform significantly worse than the operator and connector. Because these stages offer poor performance and do not provide any functionality that is not offered by the connector or operator, the emphasis of this document shall be on a comparison of the DB2 Connector and DB2 EE Operator.

In terms of performance, the DB2 Connector performs significantly better than the DB2 API and DB2 Load plug-ins (between three and six times the throughput of these stages). When compared to the EE stage, the connector performs better for INSERT. SELECT performance of the DB2 Connector can vary when compared to the EE stage, depending on the number and type of columns, and the number of parallel nodes.

The DB2 Connector must be considered the stage of choice in almost every case. Because it offers more functionality and significantly better performance than the DB2 API and DB2 Load stages, the connector is the choice, rather than these stages. The DB2 Connector must be used instead of the EE stage in most cases.

The EE stage has an advantage in the following cases:

- Bulk load performance
- Select performance, in certain situations, especially with a large number of columns

DB2 Enterprise configuration is significantly more complex than the configuration of the DB2 Connector. If obtaining absolutely the best performance is paramount, and the added configuration complexity is acceptable, then DB2 Enterprise must be used.

Architectural differences

The primary difference between the connector and EE stage is that the former uses the CLI interface for all of its interactions to enable client-server communications for all functions. By contrast, the EE stage only uses a client-server interface on the conductor node for querying table definitions and partitioning information. The player nodes have to run on the DB2 server. The EE stage can only be used with a partitioned DB2 database. That is, the DB2 Database Partitioning Feature (DPF) must be enabled.

The deployment of the EE stage is illustrated in Figure 14-15.

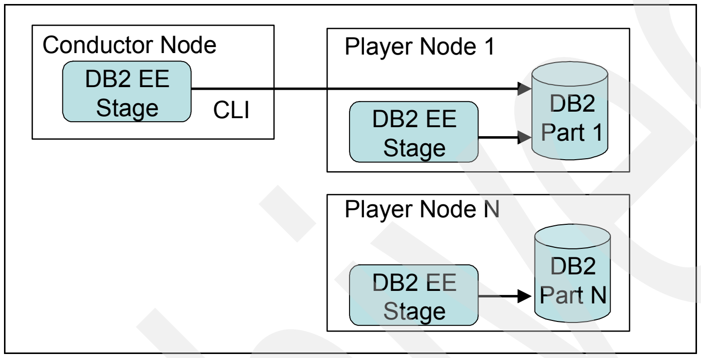


Figure 14-15 DB2 enterprise deployment architecture

In the case of the connector, the player nodes might correspond to DB2 partitions (as is the requirement for the EE stage), but this is not a requirement. Because the connector communicates with DB2 through the remote client interface, the deployment options are much more flexible. This is depicted in Figure 14-16.

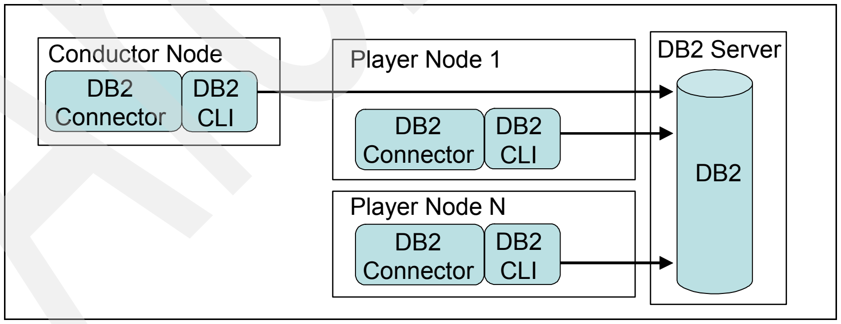


Figure 14-16 DB2 connector deployment architecture

There are advantages and disadvantages to both approaches:

- The EE stage requires that the DataStage engine and libraries be installed on the DB2 nodes. This can be achieved by running a script that is provided as part of the DataStage install. This script copies only PX core libraries, but does not copy any of the operators, including DB2 EE stage. These additional libraries have to manually copied to the DB2 partitions.

  Advantage: connector

- The EE stage requires a homogeneous environment. The DataStage server nodes and DB2 server nodes must be running the same hardware with the same operating system. The connector has no such requirement. This means there are customer situations where the EE stage cannot be used.

  Advantage: connector

- The EE stage maintains a direct local connection between the operator and the DB2 server (because it is always co-resident). This might offer performance advantages compared to the connector, which always communicates to the DB2 server through TCIP/IP, even when it is installed locally to the DB2 server.

  Advantage: operator

14.6 Oracle Connector

The Oracle Connector includes these features of the ODBC Connector:

- Source
- Target and lookup context
- Reject links
- Passing LOBs by reference
- Arrays
- SQL Builder
- Pre/post run statements
- Metadata import

In addition, it supports bulk loads and Oracle partitioning.

The Connector works with Oracle versions 10g and 11g. It supports connecting to an Oracle database through Oracle Full Client or Oracle Instant Client (Basic or Basic Lite). Oracle Connector usage patterns are depicted in Figure 14-17.

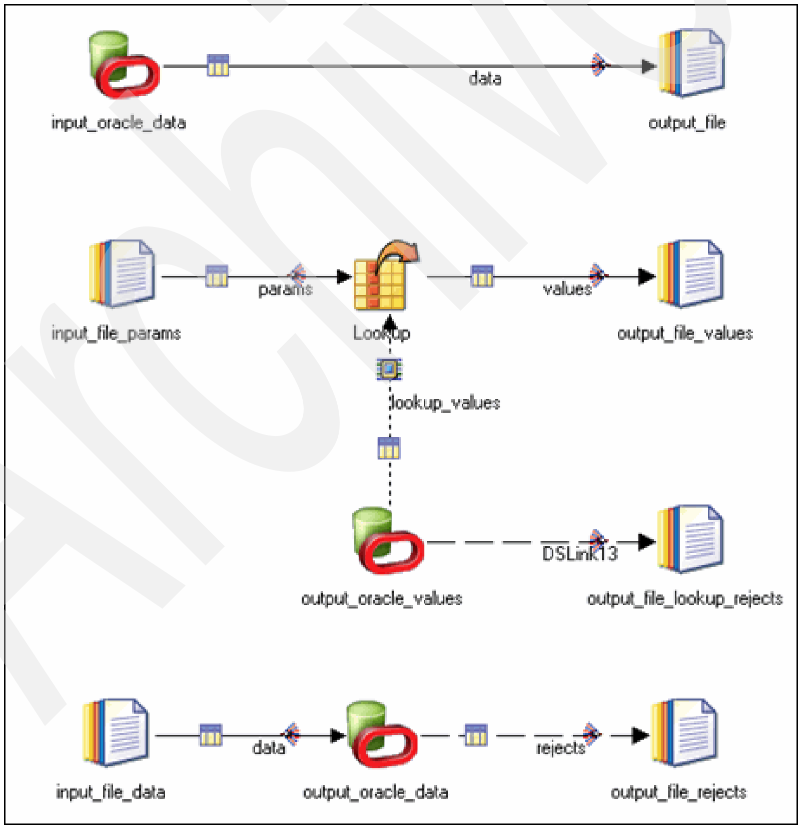


Figure 14-17 Oracle connector usage patterns

14.6.1 New features and improvements

The following list details improvements made to the Oracle Connector:

- Distributed transactions
  - Support for guaranteed delivery of transactions arriving in form of MQ messages. In case of success, the messages are processed by the job and the data is written to the target Oracle database. In case of a failure the messages are rolled back to the queue.
  - To use the Distributed Transaction stage, you need MQ 6.0 and Oracle 10g R2.
- Built-in Oracle scalar data types are supported, including BLOB, CLOB, NCLOB, BFILE, LONG, and LONG RAW data types.
- XMLType columns and object tables are supported.
- PL/SQL anonymous blocks with bind parameters are supported.
- Support for a rich set of options for configuring reject links. Reject links are also supported in bulk load mode
- Pre- and Post- SQL operations are supported:
  - SQL statement, multiple SQL statements or PL/SQL anonymous block might be specified.
  - The statements might be configured to run at the job or at the node level.
  - The statements might be specified in the stage UI or in external files.
- Rich metadata import functionality:
  - Table selection based on table type (table, view, IOT, materialized view, external table, synonym), the table owner or the table name pattern.
  - Supports importing PK, FK, and index information.
- Table action:
  - Performing Create, Replace, or Truncate table actions in the job is supported before writing data to the table.
  - Input link column definitions automatically used to define target table columns.
- Fast read, write, and bulk load operations:

  Internally conducted testing showed performance improvements over Oracle Enterprise stage.
- Rich set of options for configuring parallel read and write operations.

- Provides control over the Transparent Application Failover (TAF) mechanism in environments such as Oracle RAC.

  Includes storing TAF notifications in the job log to inform the user about the failover progress.
- Oracle or OS authentication is supported for the connector at job runtime.

14.6.2 Comparison with Oracle Enterprise

Table 14-6 shows a mapping between Oracle-related environment variables used by existing stages to the Oracle connector. Behavior of Oracle Connectors is no longer affected by any environment variables, with the exception of the Oracle client vars such as ORACLE_HOME and ORACLE_SID.

Table 14-6 Equivalence of Oracle-related environment variables

Existing Oracle-Related Env Vars	**Connector Equivalent**
$ORACLE_HOME and $ORACLE_SID	These are Oracle environment variables. They apply to the connector the same way they apply to the Enterprise stage. They are used by the Oracle client library to resolve Oracle service name to which to connect.
$APT_ORAUPSERT_COMMIT_ROW_INTERVAL	No environment variable. The connector has a record count property that can be used to control this.
$APT_ORAUPSERT_COMMIT_TIME_INTERVAL	No environment variable and no property.
$APT_ORACLE_LOAD_OPTIONS	Not applicable to the connector. The connector does not use SQL*Loader for bulk-load hence the loader control and data files are not applicable to the connector. The connector uses OCI Direct Path Load API. The settings to control the load process are provided through connector properties.
$APT_ORACLE_LOAD_DELIMITED	None, because the connector does not use SQL*Loader.
$APT_ORA_IGNORE_CONFIG_FILE_PARALLELISM	None, because the connector does not use SQL*Loader.
$APT_ORA_WRITE_FILES	None. The connector does not use SQL*Loader for bulk-load operations.
DS_ENABLE_RESERVED_CHAR_CONVERT	None. The connector relies on the framework to take care of this. The connector itself supports importing, reading or writing to Oracle tables and columns that contain # and $ characters. No extra setting is needed for this.

14.7 DT stage

The Connector framework provides support for distributed two-phase XA transactions in DataStage Enterprise jobs. The transactional data is carried by MQ messages that arrive at the source queue. Each message might encompass multiple database operations, and multiple messages might be grouped in a single physical transaction.

The MQ Connector stage provides properties for the configuration of transaction boundaries. It is possible to specify the number of source messages included in each transaction or the time interval for collecting source messages in each transaction. The MQ Connector uses a specially designated work queue as temporary buffer storage for source messages that participate in transactions.

The retrieved messages are processed by any number and combination of transformation stages, chosen from a rich palette of stage types provided by DataStage. The processed messages result in rows of data that arrive on the Distributed Transaction (DT) stage on one or more input links. Each input link on the DT stage is associated with one external resource. The rows on each link are sent to the designated resource (as insert, update, or delete operations on the resource).

The DT stage reads messages from the work queue and updates external resources with the data rows corresponding to those work queue message. The reading of messages and writing to external resources is done in atomic distributed transactions using the two-phase XA protocol.

The DT stage can also be used outside the context of a queuing application. Transactions are committed upon EOW markers. This has application in ISD jobs that need to guarantee transactional consistency, but do not involve the processing of messages.

The DT stage is depicted in Figure 14-18 and described in detail in Chapter 16, “Real-time data flow design” on page 293.

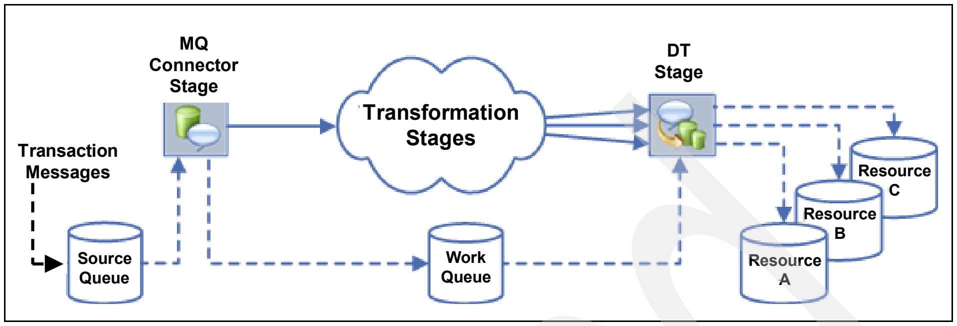
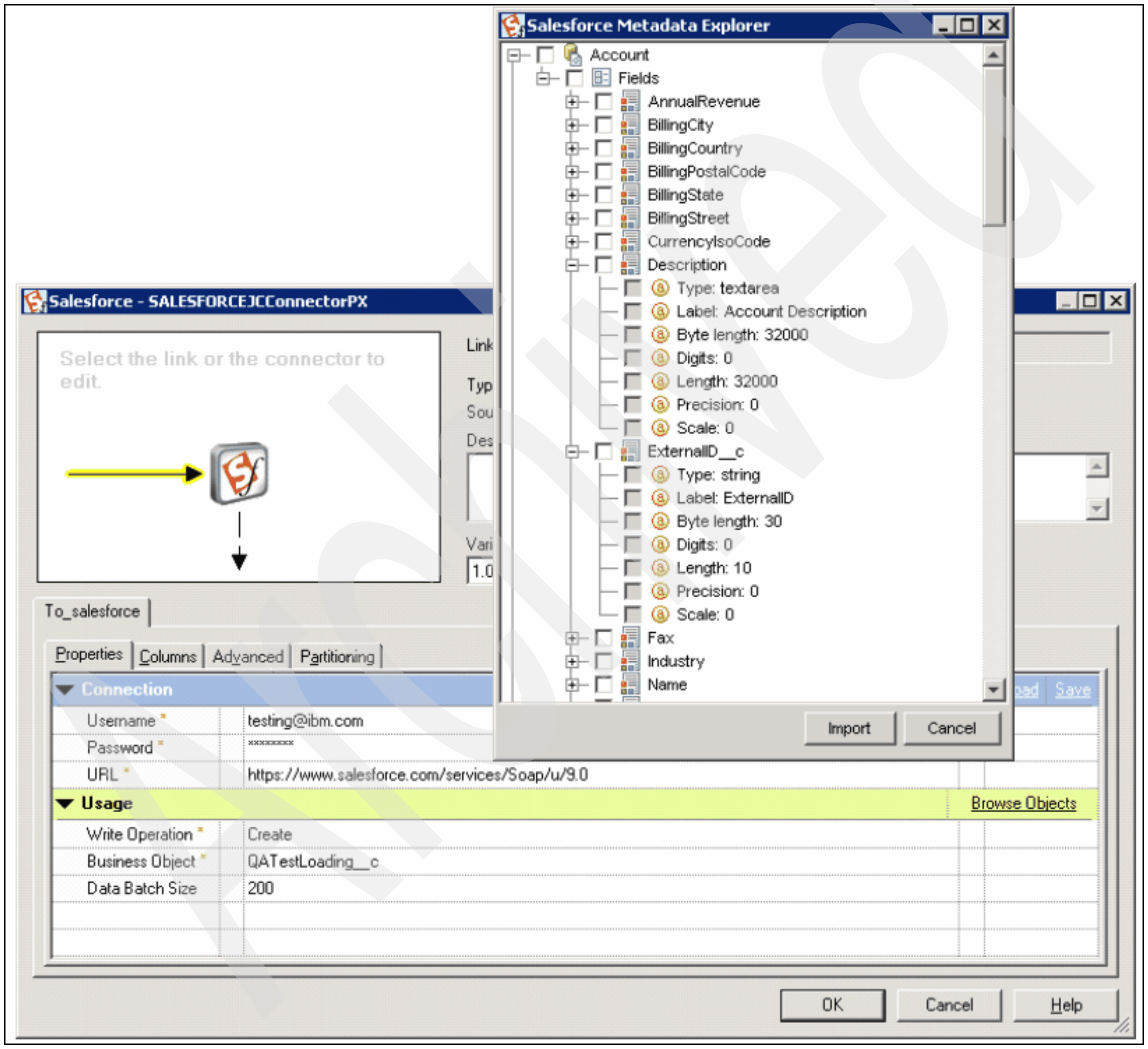


Figure 14-18   The DT stage

14.8 SalesForce Connector

The SalesForce Connector, depicted in Figure 14-19, supports load, extraction, and delta extraction. It generates SQL statements based on user selection. Load mode supports create and updates, as well as reject links. The connector uses a Web service interface through an Axis library, and is implemented as a Java connector using DMDI.



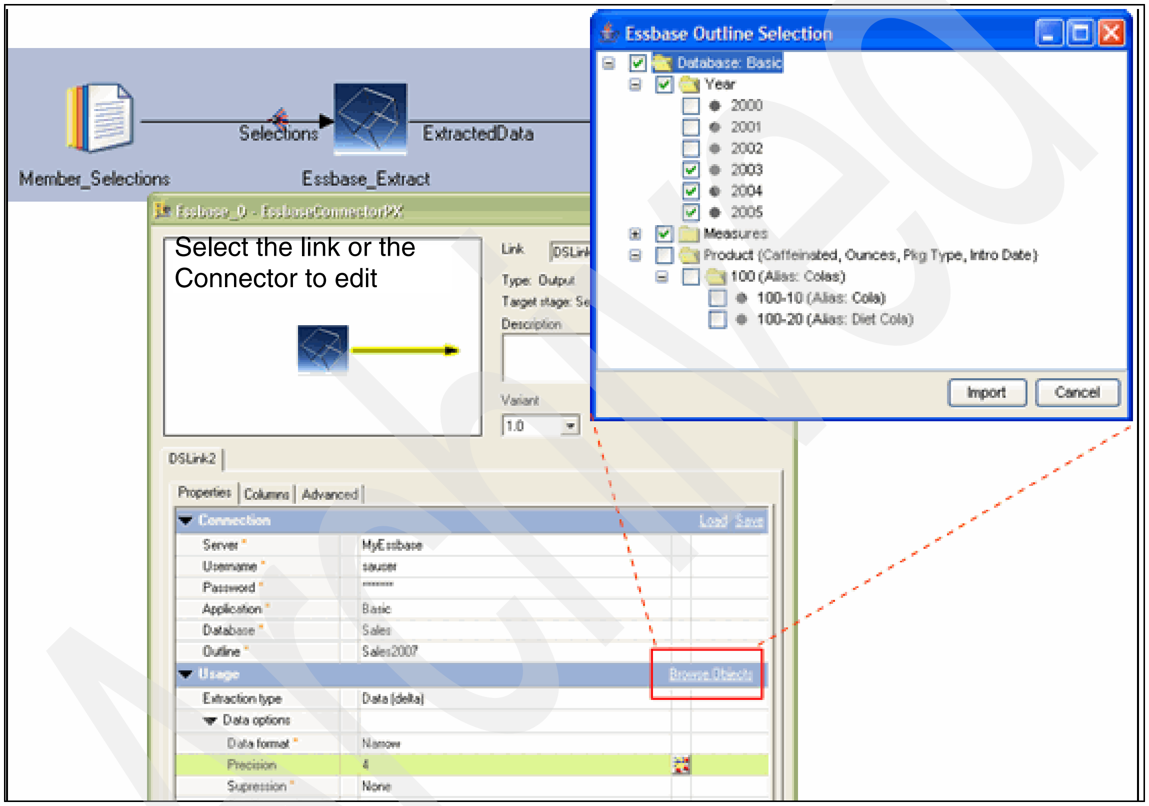
Connection	
Username *	testing@ibm.com
Password *	********
URL *	https://www.salesforce.com/services/Soap/u/9.0
Usage	
Write Operation *	Create
Business Object *	QATestLoading__c
Data Batch Size	200

Figure 14-19 SalesForce connector

14.9 Essbase connector

The Essbase connector supports the extraction, delta extraction, and load of data to and from Essbase databases. It performs hierarchical to relational mapping of cube data. The connector supports parallel read and writes.

It is implemented in C++ using a 3rd party interface library from a partner, and uses DMDI to allow selection from cube data. See Figure 14-20.


Member_Selections
Selections
Essbase_Extract
ExtractedData
Essbase_0 - EssbaseConnectorPX
Select the link or the Connector to edit
Essbase Outline Selection
Import
Cancel

Link: DSLink
Type: Output
Target stage: Se
Description
Variant: 1.0

DSLink2

Properties | Columns | Advanced

Connection	Load Save
Server *	MyEssbase
Username *	sauser
Password *	*******
Application *	Basic
Database *	Sales
Outline *	Sales2007
Usage	Browse Objects
Extraction type	Data (delta)
Data options	
Data format *	Narrow
Precision	4
Supression *	None

- [✓] Database: Basic
  - [✓] Year
    - [ ] 2000
    - [ ] 2001
    - [ ] 2002
    - [✓] 2003
    - [✓] 2004
    - [✓] 2005
  - [✓] Measures
  - [ ] Product (Caffeinated, Ounces, Pkg Type, Intro Date)
    - [ ] 100 (Alias: Colas)
      - [ ] 100-10 (Alias: Cola)
      - [ ] 100-20 (Alias: Diet Cola)

Figure 14-20 The Essbase Connector

14.10 SWG Connector

The SWG Connector, depicted in Figure 14-21, provides a bridge to SWG Adapters. The first release is to support SAP. The SWGA Connector is Adapter-neutral. Any adapter should plug in without a need to change the connector. A DMDI implementation provides the means to configure the stage and import metadata.

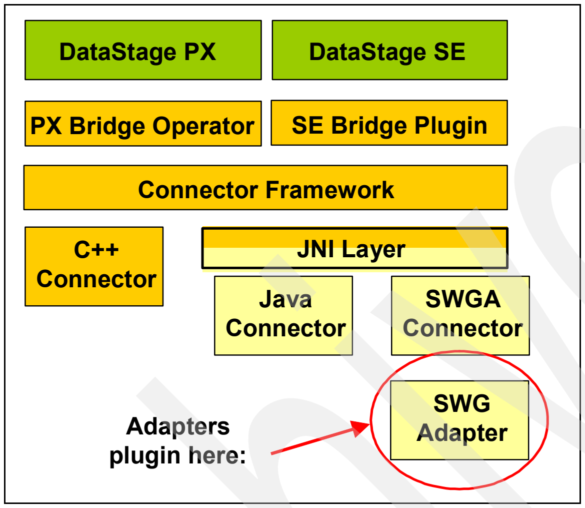


Figure 14-21 SWG Connector

15

Batch data flow design

Batch applications are always constrained by execution time windows. They must complete the extraction, transformation, and load of large amounts of data in limited time windows. If the execution fits in a certain expected amount of time, performance requirements are met. However, it is not enough to meet initial expectations narrowly. Jobs must be able to scale, as data volumes grow.

Parallel batch jobs must be implemented with parallel techniques that yield the necessary scalability, which is the focus of this chapter. The basic assumption throughout this discussion is that no applications other than DataStage (DS) update the target databases during batch windows.

This chapter does not focus on individual stage parameters, but rather on data flow design patterns. Individual stage types are the subject of other chapters in this document. For instance, when dealing with database interfaces, there are several options for each database type, along with their tuning parameters.

15.1 High performance batch data flow design goals

In this section we describe the main goals a DS application developer must keep in mind when designing batch applications.

15.1.1 Minimize time required to complete batch processing

The first and foremost goal is to complete work in the available time, be it a daily, weekly, or monthly process.

15.1.2 Build scalable jobs

It is not enough to meet initial expectations. As more data is processed, jobs should complete in approximately the same amount of time as more hardware is added (processors, memory, disks). Conversely, efficient designs make it possible to reduce the execution time for the same amount of data as more hardware resources are added. There are things that money can buy, such as more hardware. However, bad job designs do not scale even with more hardware, in most circumstances.

15.1.3 Minimize the impact of startup time

Jobs have a startup up time, which is the time spent by a job during its preparation phase. During this phase the conductor process parses the OSH script (the script created by the compilation), builds a job score (similar to a query execution plan in database terms) and spawns child processes such as section leaders and players. These are steps that cannot be parallelized. They are run sequentially by the conductor process and they take a certain amount of time.

In most cases, the startup times tend to be small but they grow as the jobs grow in size. They can take considerable time for a large number of jobs.

Jobs must be designed in a way that allows them to process large amounts of data as part of a single run. This means input files must be accumulated and submitted as input to a single job run. This way, instead of starting up each and every job in a long sequence several times (such as once for each input file), the sequences and their jobs are started up only once.

Once up and running, well-designed jobs are able to cope with large amounts of data. The use of parallel techniques leads to more streamlined and efficient designs that help mitigate the impact of startup times.

15.1.4 Optimize network, I/O and memory usage

The following list details the hardware resources in descending order of their cost (this order assumes storage subsystems comprised of parallel disks):

1. Network
2. Disk
3. Memory
4. CPU

Network and disk I/O are the two slowest hardware resources in most systems. Therefore, they must be used judiciously. For small amounts of data, they are not a concern. But when there is a need to meet the demands of large environments, these two resources are no longer negligible. There are a few aspects to keep in mind that have impact on one or more of the resources listed:

- Minimize the landing of data to disk.
  - Instead of writing and reading data to and from disk after each and every small job, create jobs that do more processing as part of the same execution.
  - This was the typical approach for DS Server applications, which must be avoided in the parallel world.
- Minimize interaction with the database Server.
  - Extract/Load data in bulk.
  - Avoid sparse lookups.
  - Avoid extracting unnecessary data for lookups

    For instance, avoid full table extracts when the number of input rows is significantly smaller. One does not need to extract 100 million rows from a database as a lookup table to process only a few thousand incoming records.
- Avoid unnecessary re-partitioning.

  Re-partitioning cost is not pronounced in SMP systems, but it adds to private network usage in DataStage grid environments.
- Limit the amount of reference data for normal lookups.
  - Single large normal lookup reference tables can consume a huge amount of memory.
  - Avoid full table extracts when the number of rows being processed is significantly smaller.
  - If the number of rows cannot be restricted, use a sorted merge instead of a normal lookup.

- Implement Efficient Transformations

  Must implement efficient expressions and logic after data is in the job's address space. This has a direct impact on CPU usage.

  For instance, instead of using a long "IF-THEN-ELSE IF-…" sequence in a Transformer for conversions, use a small normal lookup to implement the same logic.

15.1.5 Plan job concurrency and degrees of parallelism

Once application coding is finished and ready for production, the applications must be able to be executed with varying degrees of parallelism, depending on their complexity and amount of data to be processed:

- Contain the number of concurrent jobs.
  - The goal is to avoid swamping the hardware resources with too many concurrent jobs.
  - Job sequencing and scheduling must be planned carefully.
- Maximize the degree of parallelism for larger, complex jobs.
- Reduce degree of parallelism for smaller, simpler jobs.

15.2 Common bad patterns

Common bad design patterns are a result of experience with the existing DS Server technology. Other bad design patterns result from inexperience in high volume batch processing.

Without going in to detail about the origins of bad practices, the following sections discuss a few recurring patterns that unfortunately are common in the field, and lead to less than optimal results to say the least. Bad practices includes, but are not limited to, the topics discussed in the following subsections.

15.2.1 DS server mentality for parallel jobs

DS server and parallel jobs are similar from a Graphical User Interface perspective providing the same basic paradigm: stages and links. However, they implement fundamentally different execution frameworks.

DS Server provides a limited degree of intra-job concurrency: an active stage and its surrounding stages are mapped to processes, and the number of active stages determines the number of processes that are executed as part of a

Server job run. There is no support for partitioning parallelism, unless done manually with multiple invocations of the same job.

The active/passive nature of DS Server job stages limits the types of stages and prevents a data flow execution model. This ultimately prevents joins and merges from being made available in DS Server. As a result, DS Server jobs follow design patterns that include the following attributes:

- Singleton lookups.

  A SQL statement is executed for each and every incoming row.
- Small jobs with frequent landing of data to disk.

  The main reason being checkpoint and job restartability.
- All correlation must be done using hash files and Transformers.

  Results from lack of join/merge functionality.
- Manual partitioning with multiple instances of a same job.

Unfortunately there is a tendency for seasoned DS Server professionals to adopt the same techniques for parallel jobs. This must be avoided. Developers must put effort into learning new techniques to make the most of the parallel framework.

15.2.2 Database sparse lookups

A database sparse lookup is one of the most inefficient constructs in parallel jobs. That is because it involves the invocation of an SQL statement for each and every row flowing through the lookup stage.

This invocation involves preparing and sending one or more packets to the database server, the database server must execute the statement with the given parameters and results are returned to the lookup stage.

Even if the database server is collocated with DataStage, it is still a bad solution. There is at least one round-trip communication between the Lookup stage and the database server, as well as a minimum set of instructions on both sides (DataStage and database) that is executed for each and every row.

By using bulk database loads and unloads, larger sets of data are transferred between DataStage and the database. The usage of the network transports as well as the code paths on each side are optimized. The best solution is then achieved, at least in terms of DataStage interfacing.

This has implications in the way DataStage jobs are designed. Parallel jobs must make extensive use of the following elements:

- Bulk database stage types;
- Other lookup techniques, such as joins and normal lookups.

In rare occasions, sparse lookups might be tolerated. However, that is only when it is guaranteed that the number of incoming rows, for which the Lookup must be performed, is limited to at most a few thousand records.

15.2.3 Processing full source database refreshes

There are cases when source systems (such as mainframes) are unable to restrict the data transferred to data warehouses on a daily basis.

Those systems are not equipped with the right tools to only extract the net difference since the last refresh. As a result, incremental transfers are not possible and the entire content of the source database must be copied to the target environment in each processing cycle (frequently on a daily basis).

That puts a tremendous amount of load across the board:

- Processor, memory and disk:

  On the source system and especially on the target DataStage and DW servers;
- Network

  Networks are limited to a certain bandwidth and unless the infrastructure is upgraded (assuming there is faster technology available), they just cannot go any faster.

Source systems must implement a form of change data capture to avoid this unnecessary burden.

15.2.4 Extracting much and using little (reference datasets)

Beware of extracting huge reference datasets but only using a small fraction of them. If an input dataset to a batch process is significantly smaller than the target database table, it is a poor use of time and resources to download the entire table contents to a lookup file set or dataset. It is a waste of network, disk, memory and processor resources. Unfortunately, this is a common, recurring pattern.

The size of reference datasets and lookup file sets must be restricted using the techniques presented in this chapter.

15.2.5 Reference data is too large to fit into physical memory

Beware of insisting on normal lookups when the reference data is too large to fit into physical memory. There are frequent cases when, although it is clear the reference data consumes all memory, developers insist on using normal lookups instead of resorting to joins.

It might be the case of not only a single lookup, but the combination of all normal lookups across multiple concurrent jobs that eat up all available memory and leave little or no space in physical memory for other processes and applications.

Even worse, projects continue to consume not only the physical memory, but all swap space as well. In these severe cases, DataStage throws fatal exceptions related to fork() errors. The application must be changed to use the right mechanisms:

- Restrict the size of the reference data
- Use sorted joins

15.2.6 Loading and re-extracting the same data

Instead of caching rows in the form of persistent datasets or file sets, applications load records into a target database and re-extract the same information over and over again. This is a bad design practice.

The place to store data that might be re-referenced in the same batch cycle is in persistent datasets or file sets. As jobs process incoming data, looking up data, resolving references and applying transformations, the data must be kept local to DataStage. Only at the last phase of the Extract, Transform, and Load (ETL) process can data be loaded to the target database by provisioning jobs.

15.2.7 One sequence run per input/output file

Another common pattern is the re-execution of entire job sequences for each and every input or output file. There are cases when the application generates large numbers of output files. But the file names are derived from values set as job and sequence parameters. In this case, the entire sequence must be executed for each and every output file.

Data values cannot be passed as job parameters. The application must be structured in a way that it can process large amounts of data in a single job or sequence run. The application is then able to scale across multiple processors (and blades in a grid environment) by increasing the degree of parallelism in the APT_CONFIG_FILE and by adding more hardware resources.

There are cases when having multiple job invocations is unavoidable, such as when input files are received at separate times during the execution time window. For example, files from a mainframe might be transferred to the ETL environment at separate times during the 12:00am to 6:00am window. When this happens, it is not a good option to wait until 6:00am to have all files ready and then start the entire batch sequence. However, it is important to accumulate input files to an extent, to limit the total number of sequence executions.

15.3 Optimal number of stages per job

It is a good practice to combine more logic inside a single job. The main purpose in combining more logic inside a single job is to gain the following advantages:

- Avoid small jobs
- Limit the frequency in which data is landed to disk

We understand that combining too much logic inside a single job might saturate the hardware with an excessive number of processors and used memory. That is why it is important to know how much memory and processing power is available. However, considering the difference between memory and disk access speeds, we tend to favor larger jobs.

The number of stages for each job must be limited by the following factors:

- The amount of hardware resources available: processors and memory
- The amount of data in the form of normal lookup tables
  - One or more large lookup tables might end up consuming most of available memory.
  - Excessively large normal lookups must be replaced by joins.
  - The amount of data on the reference link must be restricted to what is minimally necessary.
- Natural logic boundaries
  - Such as when producing a reusable dataset or lookup fileset. The creation of such reusable set must be implemented as a separate job.
  - Database extraction and provisioning jobs must be isolated to separate jobs.
  - There are times when an entire dataset must be written out to disk before the next step can proceed.

    This is the case, for instance, when we need to upload a set of natural keys to extract relevant reference data (see 15.6.1, “Restricting incoming data from the source” on page 270).

There is not a one-size-fits-all. If there is enough hardware, it is better to do perform more tasks in a single job:

- Let the framework orchestrate the execution of components in the parallel flow
- It might require a well-tuned buffer space (scratch/buffer pools) possibly on local disks or SAN (with multiple physically independent spindles)

  This is the case for jobs that contain fork/join patterns.

15.4 Checkpoint/Restart

The parallel framework does not support checkpoints inside jobs. The checkpoint/restart unit is the job.

DataStage sequences must be designed in a way they can catch exceptions and avoid re-executing jobs already completed. DataStage sequences provide for the ability to design restart-able sequences. The designer, however, must explicitly use constructs to enable this behavior.

The DataStage Server approach has always been to land data frequently, after small steps, as pointed out in previous sections. It is not the same with parallel jobs. Again, the question is: where is the biggest cost? The answer is, in descending order:

1. Network access
2. Disk
3. Memory
4. Processor

As a result, it is better to restart a bigger scalable, optimized job than to land data after each and every small step.

15.5 Balanced optimization

In this section we discuss when it is best to use a database or DataStage for transformations. DataStage and databases work in conjunction to achieve a certain goal: loading clean, consistent data into useful warehouses.

There are certain tasks that ETL tools do better, and tasks at which databases excel. It is not only a matter of capabilities, but also how many resources are available to each, and their financial, operational, and licensing costs.

As stated before, network and database interfacing have the highest costs (That is, loading data into and extracting from databases are costly operations).

15.5.1 Transformations inside the database

The following considerations are for performing transformations inside the DB.

The following list details the conditions that, if true, indicate that you should consider performing transformations inside the DB:

- The database server is massively parallel and scalable
- There is enough HW capacity and licensing on the database side for the extra processing cost for transformations
- All the data is contained inside the DB
- Data is already cleansed
- There are guarantees that set-oriented SQL statements does not fail
- For example, you are certain that a “SELECT .. INTO…” always runs successfully.
- The database is one of the database types supported by the Balanced Optimizer

If you decide to perform the transformations inside the DB, then you can delegate transformation logic to the database server. That is what the Balanced Optimizer (Teradata) was built for (and is a new component of the InfoSphere product suite in Information Server 8.1). There is no need to extract and load data back into the database because it is already there.

The key is the use of Balanced Optimizer, so the logic is designed inside the DS paradigm, keeping the ability to draw data lineage reports. The use of pure SQL statements defeats the purpose, as there is no support for data lineage and the application becomes increasingly complex to maintain.

15.5.2 Transformations with DataStage

These are circumstances that favor performing transformations with DataStage:

- DataStage can scale in a separate environment, meaning:
  - It should result in lower cost
  - A grid environment might be set up so it scales way beyond the database capabilities
  - It does not interfere with complex database queries

- When there are limitations on the database side:
  - The database does not scale well or becomes saturated with extra transformation load
    - Not a truly parallel database
    - Not enough HW resources
  - The database is not supported by the InfoSphere Balanced Optimizer
- Application needs functionality not available inside DBs, such as:
  - QualityStage
  - Custom plug-ins
- Must combine/correlate data from other sources
  - Data not completely clean yet

    Must be able to reject rows on a row-by-row basis
  - Date from existing systems (SAP, other database flavors, non-relational sources)

15.6 Batch data flow patterns

In this section we present design patterns that address important aspects that have major impact in high performance batch applications. Some might say the proposed patterns are overkill, they are too complicated, and the results do not really pay off. Our experience demonstrates the opposite. Advice presented here is based on a solid track record. Projects that follow these recommendations and patterns are typically successful, and projects that do not follow them fail.

There is a multitude of topics covered in other chapters that are all still valid and relevant. For example, consider the type of database stage for extracting and loading data from and to databases. You need to use bulk database stages, or loads and unloads take too long. This is dependent upon how efficiently DS interoperates with target database systems.

The specifics on databases and other stage types and their properties are described in the other chapters of this IBM Redbooks publication.

In terms of design patterns (the focus of this chapter) the four considerations that have the most impact in high performance batch applications are as follows:

- Extracting reference data from source/target databases efficiently
- Correlating various datasets efficiently, such as incoming files and reference data (once this reference data is already in DataStage format)
  - Matching
  - Joins, Lookups
- Transforming efficiently based on business rules (transformation expressions, data quality
  - Custom business rules
  - Field mappings
  - Standardization, survivorship
- Loading data into the target databases efficiently.

We mentioned in the list the subject of data quality (Matching, Standardization, Survivorship). In this document we focus on efficient ways to bring candidate records into DataStage. QualityStage practices are a subject of a separate CEDI document and are not discussed here.

15.6.1 Restricting incoming data from the source

The first step in the process of implementing highly efficient batch processes is to restrict the amount of data sent from a source system (for example, a mainframe) to DataStage. The data must be restricted at the source so that only modifications to data are transmitted and not the entire contents of the source database.

If the entire source database is sent to DS for processing, it is an inefficient use of network and processing resources. Processing windows are unnecessarily long and the network proves to be the most significant bottleneck.

Networks have a certain bandwidth and do not yield a higher throughput unless the entire network infrastructure is upgraded (assuming there is a faster network technology available).

This means using Change Data Capture on the mainframe, for instance, and not resending the entire source database. If it is not possible to restrict the source data sent to DataStage, the application has to resort to full target table extracts.

Under those circumstances, you must resort to sorted joins instead of normal lookups, as there is a high probability that the reference data does not fit in memory.

15.6.2 A fundamental problem: Reference lookup resolution

The fundamental problem that pervades any type of database batch load is how to correlate the input data to database rows in the most efficient way. This includes the following common tasks:

- Finding out, for a given input record, if a corresponding record already exists in the target DB

  This helps determine if a new or existing SK must be created, or if an existing record needs to be updated.

  This applies to any Slowly Changing Dimension (SCD) Type (1 or 2), or any relational database load scenario.

- Fetching candidate records for de-duplication/matching with QualityStage

By reference lookup we mean the process of looking up any type of information (either fact or dimension tables, or any other type of table, for that matter) in the target database, on behalf of input records.

There are two possible ways of accessing reference data:

- Export the entire contents of the target tables to datasets or file sets;
- Use database sparse lookups (that is, one singleton select for each incoming row).

The first solution is inefficient if the size of the input data is small when compared to the database table size. The second solution involves a method that must be used for really small input files.

In case of SCD type 2, doing a full extract implies restricting the set to rows with Current Flag set to 1, or a filter on Expiration Date. Even restricting the extracted set to the most current records might represent an amount of data that is too large to hold in memory, or that contains way more records than are needed for a given processing cycle. Instead, we propose an approach that is based on the optimization of the interaction between DataStage and the target database, which involves the following elements:

- Restriction of the reference dataset to only what is minimally relevant for a given run.

  Avoid extracting reference data that is not referenced in the input files.

- Use of the optimal correlation strategy inside DataStage.

  Use elements such as the join, normal lookup, change capture, and SCD.

- Use data already stored locally to DataStage in the form of datasets or filesets.

  Avoid repeated and redundant database extracts.

15.6.3 A sample database model

Figure 15-1 depicts an example that is used throughout this section. It consists of three tables that are loaded from three separate input files.

Each input file maps directly to a table. The Source System Key (SSK) fields are marked as *_SSK. Surrogate key (SK) fields follow the pattern *_SK.

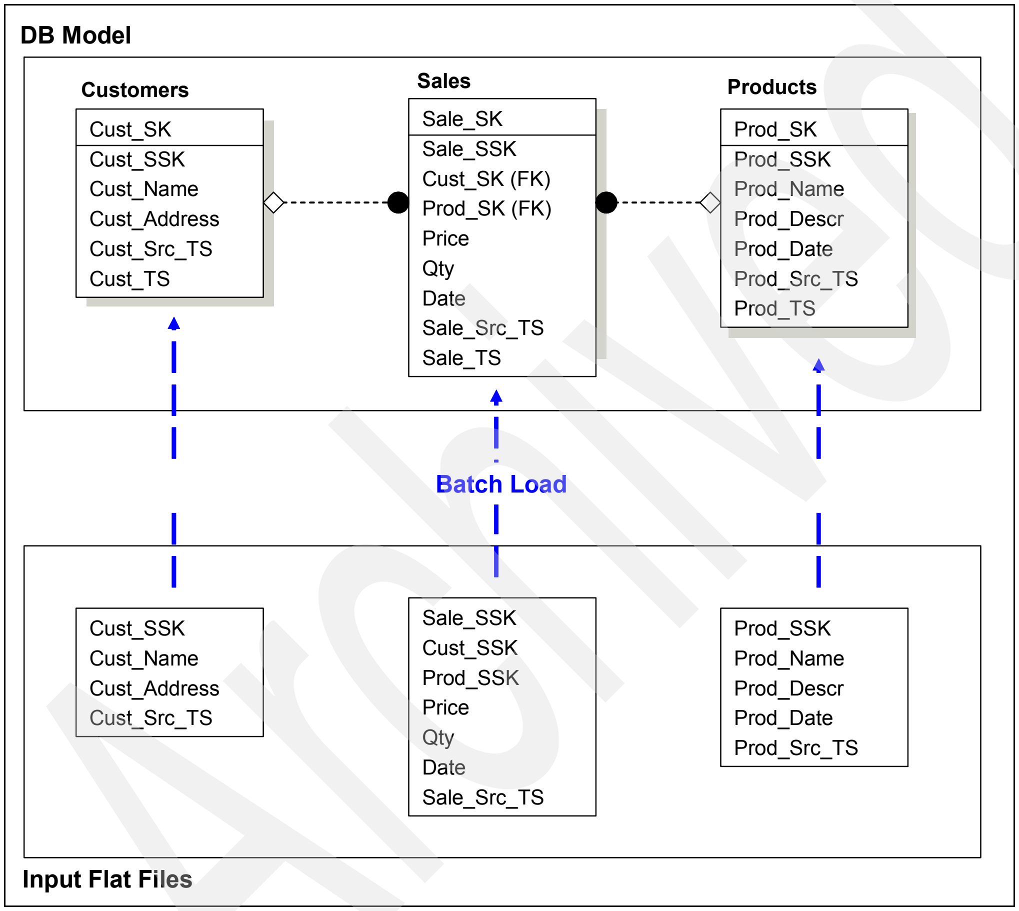


Figure 15-1 Sample database model and input files

Input files contain a Source Timestamp (*_Src_TS). Database records carry those values into the target tables. However, each record is assigned a timestamp for the time the record was created or updated by the batch process.

In this chapter we abstract the type of slowly changing dimension. However, the techniques described in the following sections apply to any SCD types.

15.6.4 Restricting the reference lookup dataset

The following process must be following when the number of input records is significantly smaller than the number of records in the target table. It consists of three basic steps:

1. Collect Unique SSKs across all input files.
2. Load those unique SSKs into a temporary table in the target database.
3. Extract the relevant set of reference records with a database join between the target and temporary tables.

Collecting unique SSKs for same entity from multiple input files

Follow these steps to collect unique SSKs for the same entity from multiple files:

1. Isolate unique sets of incoming SSKs. These unique SSK datasets help restrict the set of relevant reference records for a given run.

   As shown in Figure 15-2, each type of SSK is present in more than one file. For instance, the customer SSK (Cust_SSK) is present in the Customers and Sales flat files. All Cust_SSK values must be mapped to valid surrogate keys Cust_SK. Instead of extracting reference data for each input file separately, combine all the SSK types into separate datasets, one dataset per SSK type.

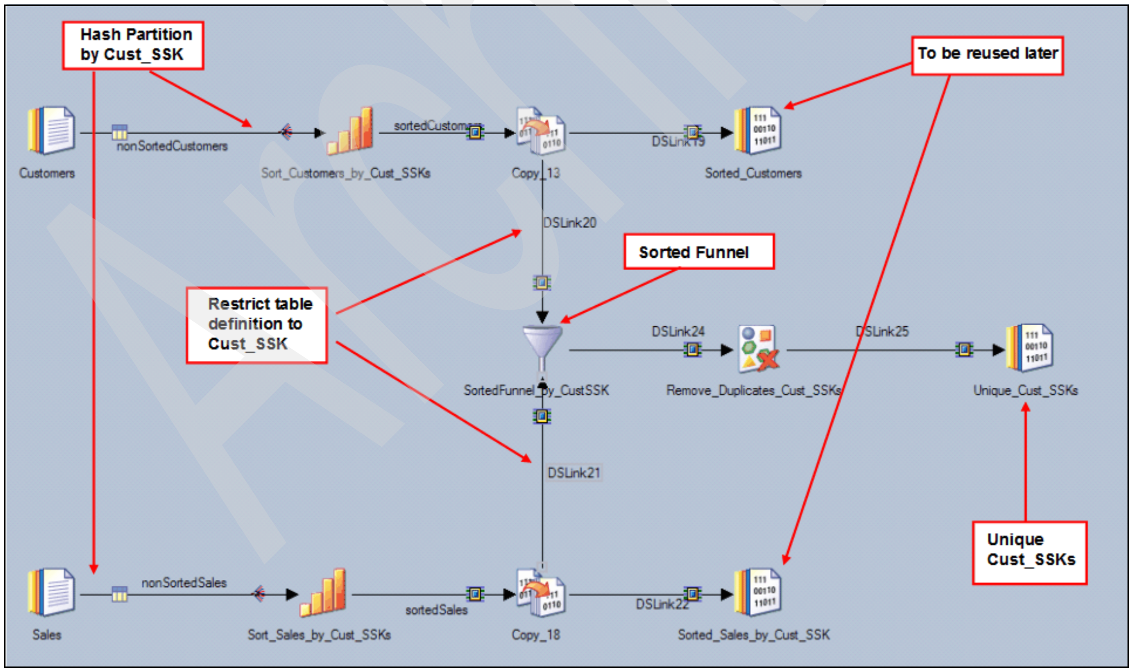


Figure 15-2 Collecting unique customer SSKs

SSKs are funneled and duplicates removed. The inputs to the sort stages are hash partitioned by Cust_SSK. We recommend the use of explicit sort stages.

2. Load the unique SSKs into a temporary table as in Figure 15-3.

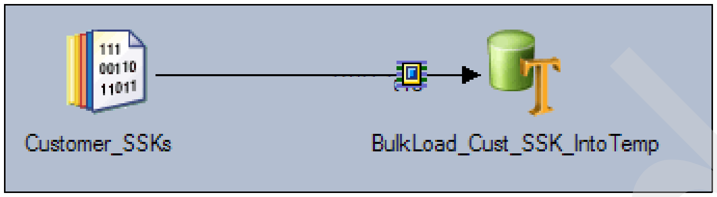


Figure 15-3 Loading Cust_SSKs into a temp table

Another pair of jobs, similar to the ones depicted in Figure 15-2 on page 273 and Figure 15-3 must be implemented for Product SSKs.

The net result of this phase is that each temp table (one for each SSK type) contains all the unique SSKs for that type across all input files.

QS matching

QualityStage matching requires candidate records based on blocking keys. There is no exact key match that can be used to extract candidate records from the database. The load process needs to create a dataset containing unique blocking keys, instead of unique SSKs. This set of unique blocking keys is loaded into the aforementioned temporary table.

SCD Type 2

For SCD Type 2, you still need to upload a set of unique SSKs. The record version to be extracted is determined by the SQL statement that implements the database join, as discussed in the next section.

Extracting the reference lookup dataset

Once the unique SSKs are uploaded to the temp table, the next job in the sequence executes a database join between the temp table and the corresponding target table (For example, a join between the Customers table and the temp table containing unique Cust_SSK).

Figure 15-4 on page 275 shows an example where the target table contains 100 million records. The uploaded temp table contains less than a million records. The result from the join is at most the same as the number of records in the temporary table.

The number of records coming out of the join is certainly less than the number in the temp table, because uploaded SSKs are not present in the target table yet.

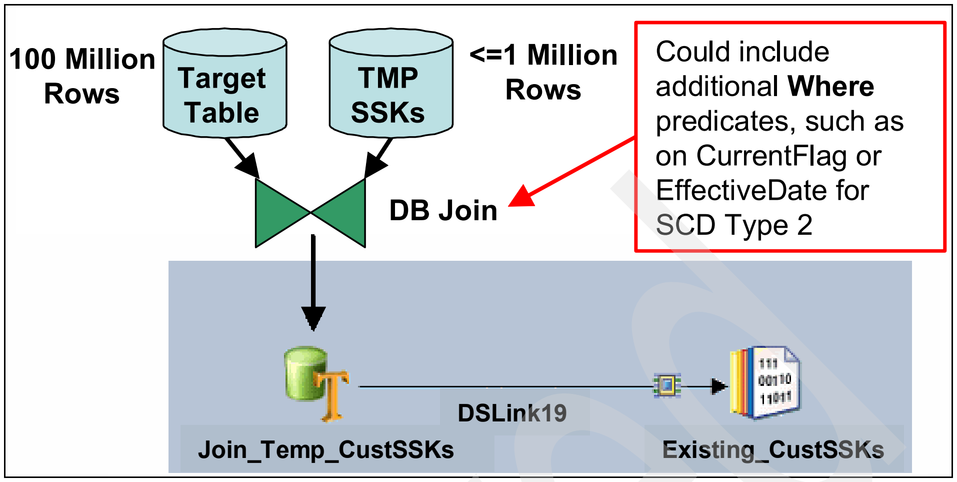


Figure 15-4 Extracting existing customer SKs with a database join

The join in Figure 15-4 guarantees that no useless data is extracted for the given run. Only what is necessary (that is, referenced by any of the input files) is extracted.

QS matching

As described in the previous section, for QS matching the temp table would have been loaded with a set of unique blocking keys.

For QS Matching, the database join extracts the set of candidate records for all input records, which serves as reference input for a QS Matching stage.

SCD Type 2

For SCD Type 2, the database join implements a filter to restrict records based on one of the following elements:

- Current flag
- Expiration Date
- Effective Date

When to use full table extracts

For extreme situations when it is unavoidable to deal with large refresh datasets, you might have to resort to full table extracts instead of the techniques described so far.

That is the case in the following circumstances:

- The amount of input data is too large when compared with the size of the target table.

  For instance, the target table contains 100 million records, and the input file has 90 million.
- The database join does not resort to an index-based plan, it does full table scans, and the overhead of uploading unique keys into a temp table + plus running a join inside the database is greater than extracting the whole table.

This makes it even more important to resort to Joins. The assumption is we are dealing with large amounts of data, so performing a full table scan is much more efficient.

15.6.5 Correlating data

Once the input and reference datasets are prepared (that is, the reference dataset is restricted to a minimally relevant subset), one of the following options can be used to do the actual record correlation:

- ChangeCapture
- Compare
- Difference
- Join plus Transformer
  - Field-by-field comparisons in Transformer expressions
  - Checksum comparison
- Normal Lookups
- SCD

Chapter 10, “Combining data” on page 149, reviews stages such as ChangeCapture, Difference Compare, Checksum and SCD. They are described in detail in the *Parallel Job Developers Guide*.

The SCD stage embeds logic specific to the processing of slowly changing dimensions. When using the SCD stage, the data for the reference link must still be restricted only to what is relevant for a given run. The SCD stage receives in its reference input whatever the corresponding source stage produces.

The preferred, scalable options are as follows:

- Joins

  The Join brings rows together and a Transformer implements the comparison logic.
- Comparison stages

  ChangeCapture, Difference, Compare

In Figure 15-5 on page 278 we show an example of using a Join to bring input and reference records together based on the SSK.

The join type is left outer join. If there are incoming SSKs that were not found in the target database table (that is, they are not present in the Existing_Cust_SSKs dataset), the join still writes those records out. The subsequent Transformer directs matching and non-matching records to separate branches.

Non-matched records are directed to a surrogate key generator. All records, with new and existing SKs are saved to the same dataset (Resolved_Cust_SSKs).

The top part of the example figure shows using a join to bring input and reference records together based on the SSK.

The dataset containing the result from the database join (Existing_Cust_SSKs) might have to include not only the existing SK values, but other fields as well. This would be the situation, for example, where business requirements dictate that other data values be compared to determine whether or not a given record should be updated.

The Transformer after the join could compare checksum values. When doing comparisons on checksums, the following tasks must be performed:

- A checksum value must be stored in the database table.
- The input dataset must be augmented with a newly created checksum value on the same set of columns used originally to create the checksums stored in the target table.

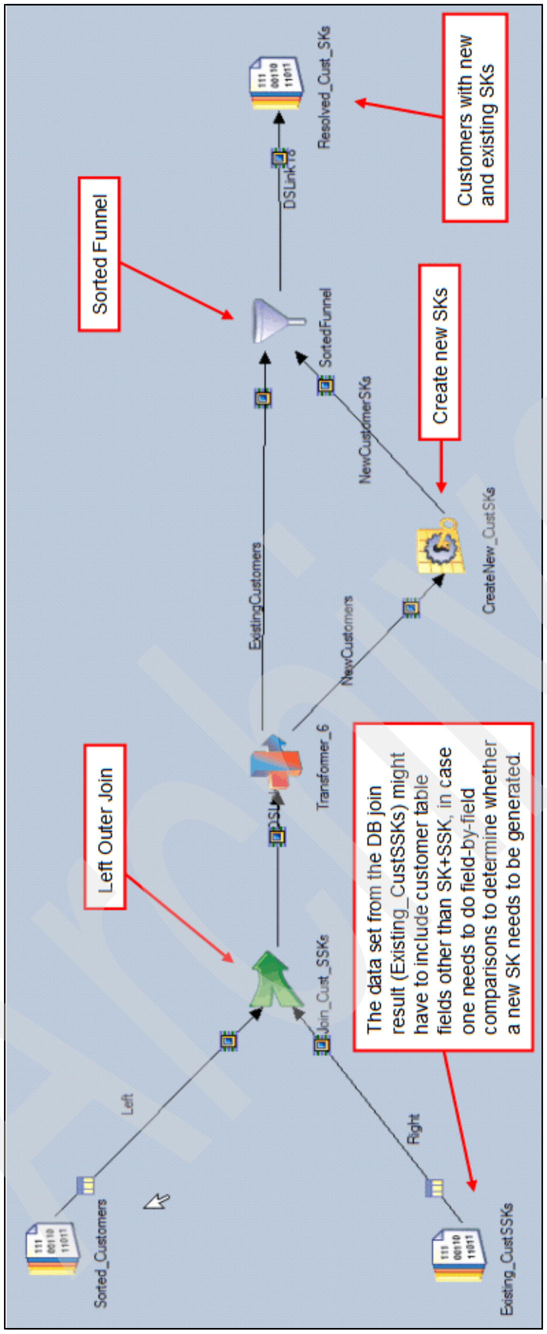

Sorted_Customers
Left
Existing_CustSSKs
Right
Join_Cust_SSKs
Left Outer Join
DSLink
Transformer_6
ExistingCustomers
NewCustomers
CreateNew_CustSKs
NewCustomerSKs
SortedFunnel
Sorted Funnel
DSLink10
Resolved_Cust_SKs
Create new SKs
Customers with new and existing SKs
The data set from the DB join result (Existing_CustSSKs) might have to include customer table fields other than SK+SSK, in case one needs to do field-by-field comparisons to determine whether a new SK needs to be generated.

Figure 15-5 Resolving customers with a Sorted Join

The following options must be used only when the reference data always fits comfortably in memory:

- Normal Lookups
- SCD stage

Figure 15-6 depicts an example of a job using a normal lookup.

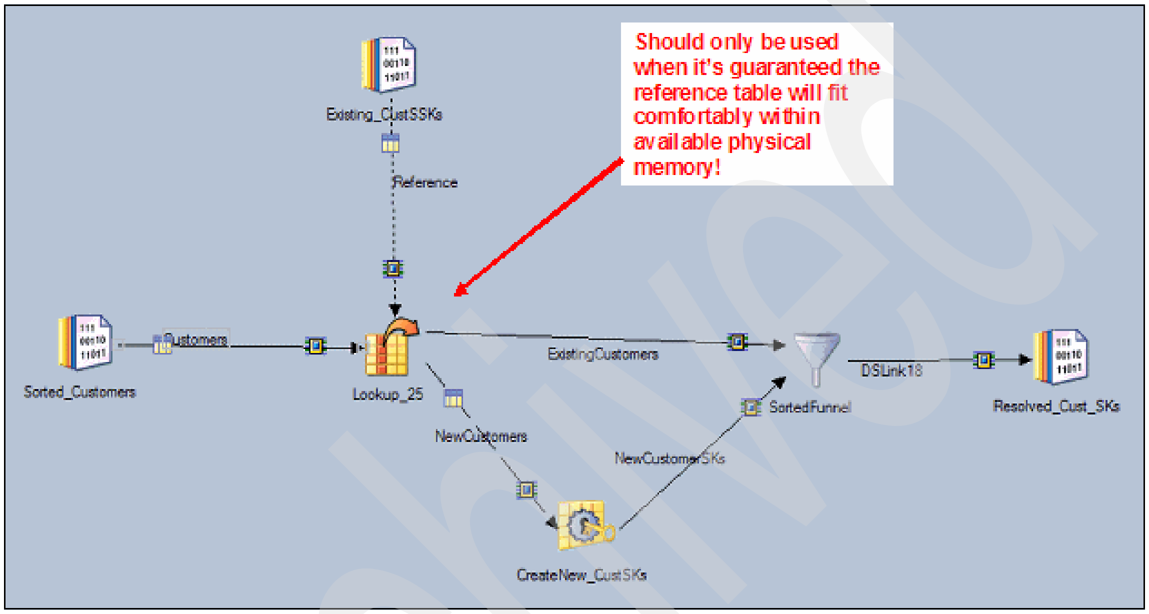


Figure 15-6 Resolving customers with a normal Lookup

Jobs similar to the ones described would have to be implemented to resolve Product SKs.

In Figure 15-7 on page 280, we describe how to use the resolved surrogate key datasets for Customers and Products to resolve foreign key references in the fact table (Sales).

For simplicity reasons, we are assuming that sales records are never updated. Records are always added to the target table. The update process follows the same approach as outlined for the Customer table.

The reference datasets are already sorted and partitioned. The input Sales file needs to be re-partitioned twice: first in the input to the first sort (for Customer lookup) and then in the input to the second sort (for Product lookup).

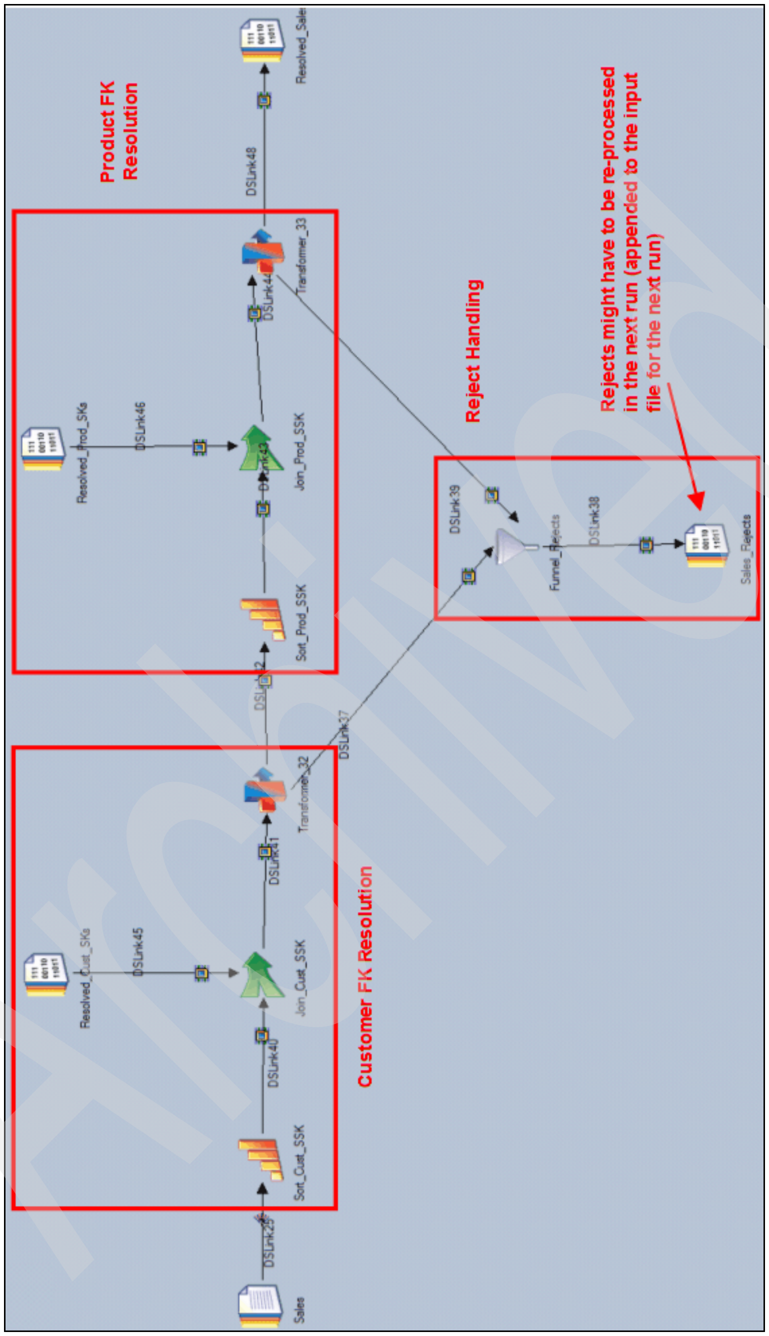


Figure 15-7 Resolving sales foreign keys

Sales records that do not find a corresponding Customer or Product SK are directed to a reject dataset. This reject dataset might be accumulated to the input to the next processing cycle (by then, hopefully, the missing customers and products have arrived).

15.6.6 Keeping information server as the transformation hub

Assume an example where input data must be first loaded into an system of record (SOR) database. This is a normalized database.

After the SOR is loaded with the day's incremental load, the DW database must be updated.

One might think of adopting a path as depicted in Figure 15-8. That diagram represents a scenario in which the second database is updated with information extracted from the first database.

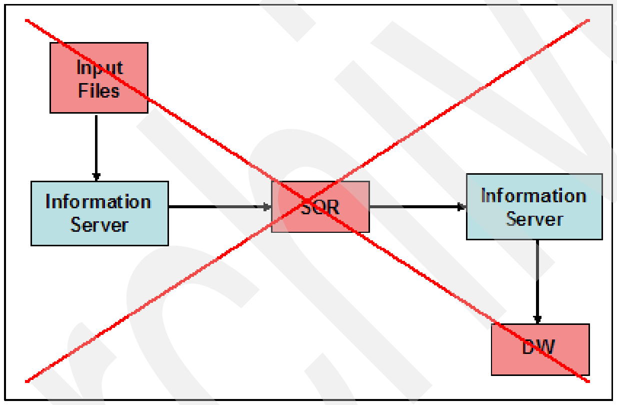


Figure 15-8 Avoiding cascading database loads

That is what must be avoided, but unfortunately is something that has been implemented in the real world.

Instead, Information Server must be kept as the transformation hub. Any data that is fed into the second and subsequent databases is derived from datasets stored locally by Information Server as persistent datasets or filesets. This is depicted in Figure 15-9.

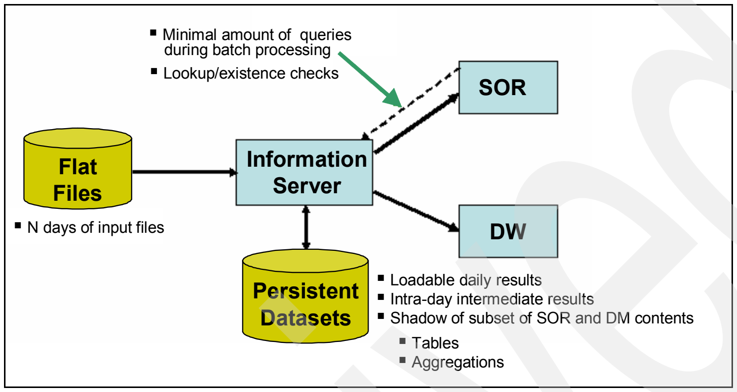


Figure 15-9 Information Server as the transformation hub

There is minimal interaction with the database. The SOR database is only queried to the extent of obtaining restricted reference datasets as outlined in the previous section.

Information Server should keep as datasets, for instance, rolling aggregations for a limited number of days, weeks and months. Those datasets act as shadows to content in the second and subsequent DBs in the chain.

15.6.7 Accumulating reference data in local datasets

When there is a high chance that resolved SKs are referenced again in the same batch window, the reference datasets containing those resolved SKs must be accumulated locally for the same processing window. This is the case when the reception of inter-related input files is spread throughout the processing window.

One good example is banking. Most likely there is separate feeds during the same processing window that relate to each other. For instance, new customers are received at one time, and then later the account movements are received as well. In this case, why first load and then re-extract the same data? Save the new customer SKs in a local dataset or file set, and use this one later on for subsequent files.

In Figure 15-10 on page 284 we show the collection of unique SSKs from multiple files. It can be expanded to include a join against a locally stored dataset containing the resolved Customer SKs for the previous run. This is characterized by the segment highlighted inside the rectangle.

This approach avoids uploading to the temp table SSKs that were already resolved by the previous run.

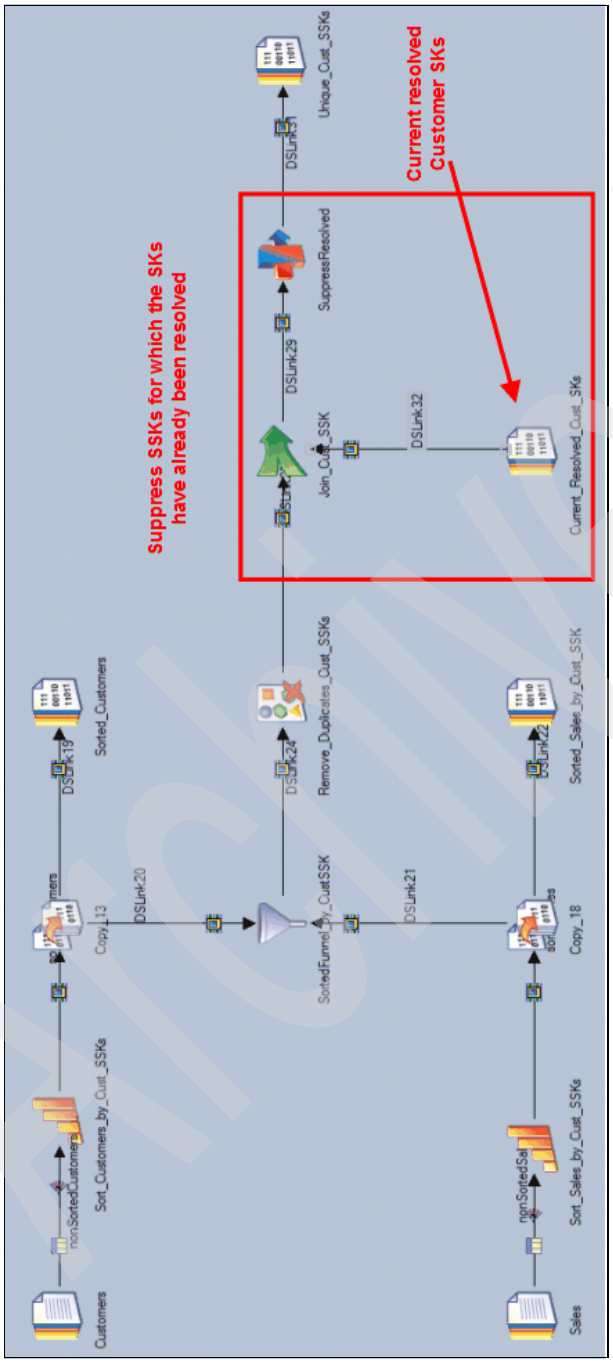


Figure 15-10 Isolating incoming SSKs, suppressing already resolved ones

Figure 15-11 shows how the existing SKs dataset is funneled with the resolved dataset from the previous run (CURRENT_Resolved_Cust_SKs). The job produces a new dataset (NEW_Cust_SKs) which, after the entire batch sequence finishes, is appended to CURRENT_Resolved_SKs along with Existing_SKs.

This way, we avoid reloading and re-extracting SSKs that were already resolved by the previous run. We combine newly created SKs and newly extracted SKs into a single dataset that serve as input to the next run.

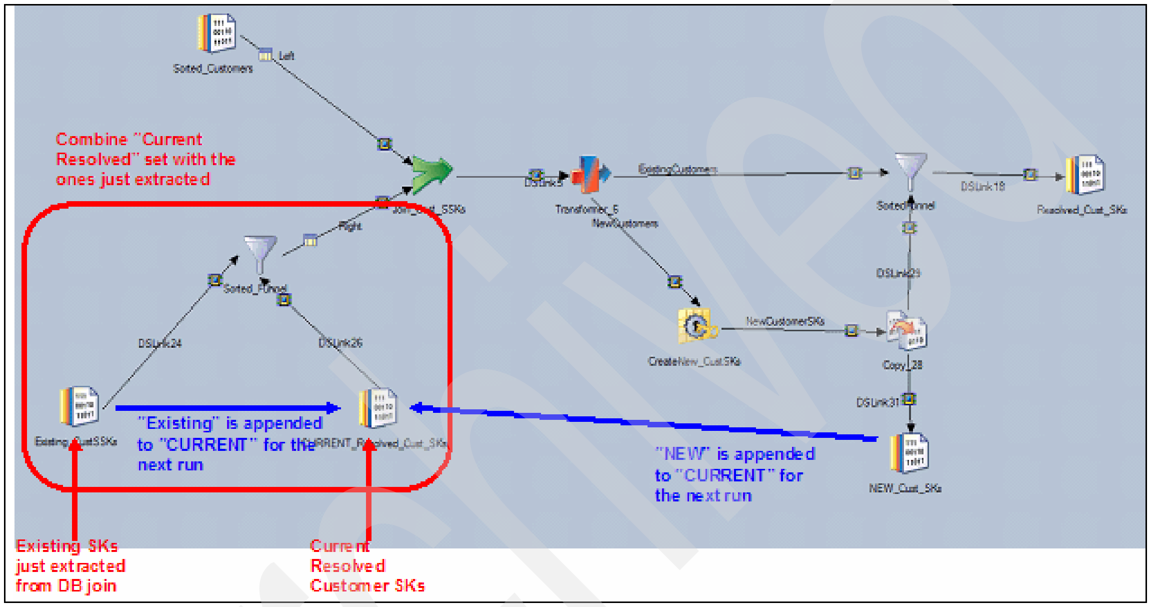

Sorted_Customers
Left
Combine "Current Resolved" set with the ones just extracted
Join_Cust_SSKs
Right
Sorted_Funnel
DSLink24
DSLink26
Existing_CustSSKs
"Existing" is appended to "CURRENT" for the next run
CURRENT_Resolved_Cust_SKs
Existing SKs just extracted from DB join
Current Resolved Customer SKs
DSLink5
Transformer_5
ExistingCustomers
NewCustomers
CreateNew_CustSKs
NewCustomerSKs
Copy_28
DSLink29
DSLink31
SortedFunnel
DSLink18
Resolved_Cust_SKs
NEW_Cust_SKs
"NEW" is appended to "CURRENT" for the next run

Figure 15-11 Resolving customer SKs, taking into account already resolved ones

15.6.8 Minimize number of sequence runs per processing window

Jobs and sequences can be executed only once per processing cycle. If that is not possible (that is, the transmission of input files is scattered throughout the execution window) the DS sequence should wait for as many input files to be received as possible before relaunching the entire sequence.

The goal is to minimize the impact of the startup time for all jobs. You must strike a balance between the urgency to process files as they arrive, and the need to minimize the aggregate startup time for all parallel jobs.

15.6.9 Separating database interfacing and transformation jobs

Provisioning and extraction stages must be contained in separate jobs. This reduces the length of time database connections are kept open, and prevents database loads or extract queries from failing in the event transformations fail for other reasons. All loads to target tables must be delayed towards the end of the processing window.

15.6.10 Extracting data efficiently

In this section we discuss techniques to consider to speed up the extraction of data in bulk mode.

Use parallel queries

Whenever possible, use parallel queries. The availability of this mode depends on the type of database stage and the partitioning of the source database table.

The ODBC stage can be set to run a query in parallel, depending on the number of partitions on which the stage is run. The stage modifies the SQL query with an extra where clause predicate based on a partition column.

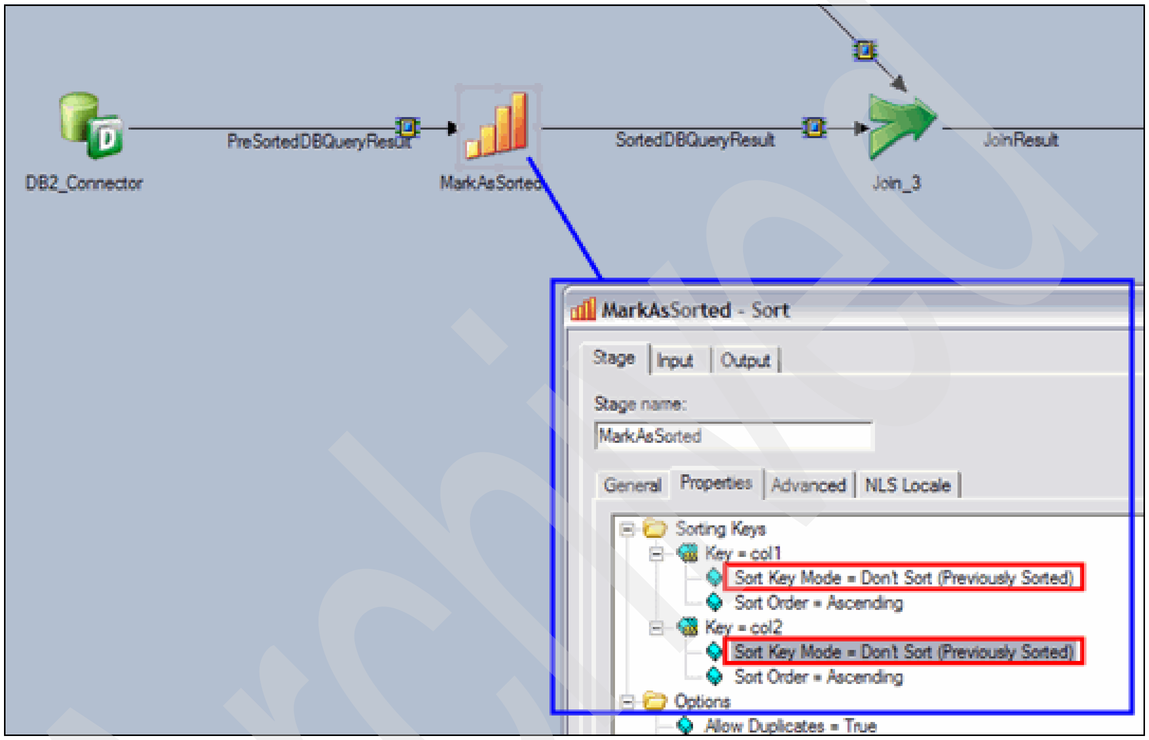
The Oracle Enterprise stage requires one to set the partition table property.

The DB2 UDB Enterprise stage requires DB2 DPF to be installed on the target database, to take advantage of direct connections to separate DB2 partitions. DB2 connectors do not need DPF installed on the target database, but the degree of parallelism of the query is still determined by the partitioning scheme of the source table.

See Chapter 13, “Database stage guidelines” on page 189 for details on how to do parallel reads with the various database stage types supported by DataStage.

Making use of database sort order

It is a good practice to make use of the sort order of the result set from a database query. This is illustrated in Figure 15-12. The developer must work closely with the database administrator to understand whether the query returns a result set that conforms to the sort order needed in the job flow. In this example, the query result is fed into one of the inputs to a Join stage.



Stage name: MarkAsSorted

- Sorting Keys
  - Key = col1
    - Sort Key Mode = Don't Sort (Previously Sorted)
    - Sort Order = Ascending
  - Key = col2
    - Sort Key Mode = Don't Sort (Previously Sorted)
    - Sort Order = Ascending
- Options
  - Allow Duplicates = True

Figure 15-12 Making use of the sort order of a database query result

If the output from the query is naturally sorted according to the criteria necessary to perform the join, one can set properties as marked in red in Figure 15-12.

By *naturally sorted* we mean the query itself does not include an ORDER BY clause, but still returns a properly sorted result set. This might happen, for instance, if the database query plan involves a scan on index pages and the index definition matches the sort order required in the DS job flow.

However, if the result set is not naturally sorted, a sort must be performed in one of two places:

- In the DS parallel job
- Inside the database by means of an ORDER BY clause

Using DataStage is the natural choice.

However, depending on the nature of the source database (whether it is a parallel database with enough spare hardware resources), you can think of adding an ORDER BY clause to the query. Whenever doing so, the technique depicted in Figure 15-12 on page 287 is used.

15.6.11 Uploading data efficiently

Parallel database stages invariably support the two major write methods:

- Bulk load
- Upsert

The first method uses highly optimized APIs to load large volumes of data the fastest possible way. The underlying mechanisms tend to be specific to each database, such as Oracle, DB2, and Teradata. With bulk loads, no SQL statements are specified and they tend to be much faster than Upserts.

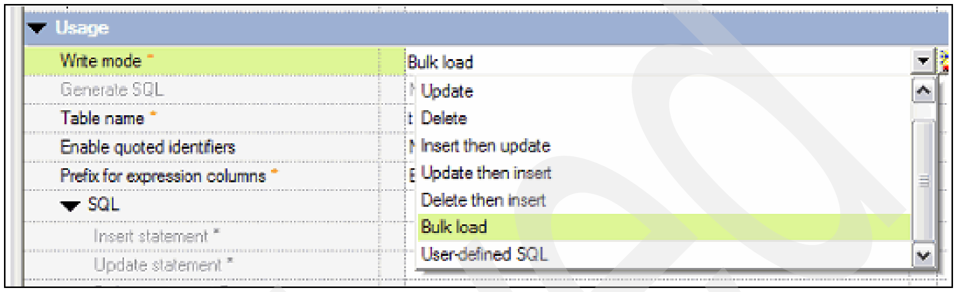
Bulk loads have the option of turning off or temporarily disabling indices and constraints. These can be rebuilt and re-enabled at the end of the load process.

The Upsert method implies the execution of SQL statements, either generated automatically or specified by the user as a custom SQL. Upserts rely on database call-level interfaces and follow a record-at-a-time processing model, as opposed to bulk loads. Most database CLIs support the execution of SQL statements by sending arrays of rows to the DB.

Upserts invariably require indices to be present and enabled, otherwise, the cost of executing an update or delete statement with a where clause is too inefficient, requiring a full table scan per row.

Figure 15-13 shows the Write Mode options of a DB2 Connector stage:

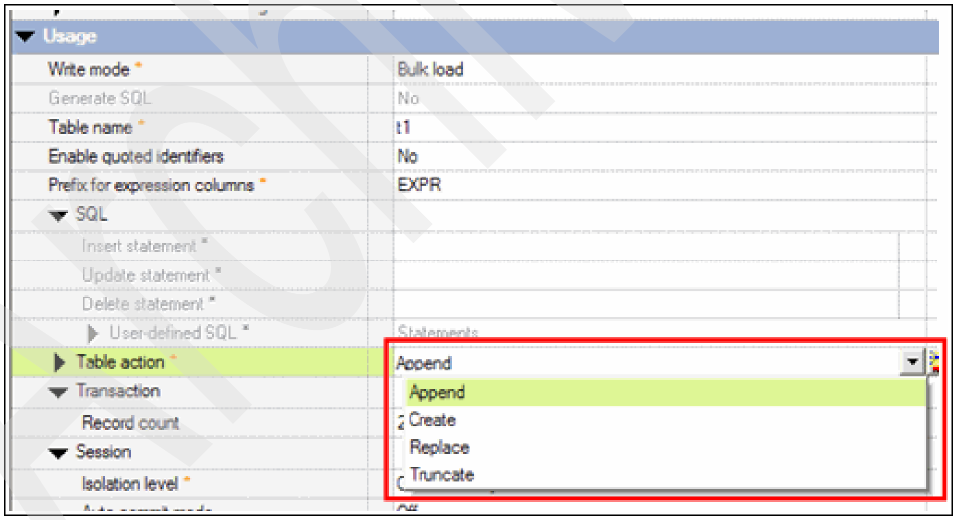
- Update
- Delete
- Insert then Update
- Update then Insert
- Delete then Insert
- Bulk Load



Usage	
Write mode *	Bulk load
Generate SQL	Update
Table name *	Delete
Enable quoted identifiers	Insert then update
Prefix for expression columns *	Update then insert
SQL	Delete then insert
Insert statement *	Bulk load
Update statement *	User-defined SQL

Figure 15-13 DB2 Connector write model

Figure 15-14 presents the table action options for the DB2 Connector, when the Write Mode is set to Bulk Load.



Usage	
Write mode *	Bulk load
Generate SQL	No
Table name *	t1
Enable quoted identifiers	No
Prefix for expression columns *	EXPR
SQL	
Insert statement *	
Update statement *	
Delete statement *	
User-defined SQL *	Statements
Table action *	Append
Transaction	Append
Record count	Create
Session	Replace
Isolation level *	Truncate

Figure 15-14 DB2 Connector table actions

Although DataStage presents a single stage type with different options, these options map to underlying Orchestrate operators and database load utilities at runtime.

Separate stages types have small variants, but drop-down options tend to be mostly similar. See Chapter 13, “Database stage guidelines” on page 189 for details on uploading with the database stage types supported by DataStage.

The question we address in this section is how to decide which database load method to use for a given task at hand? The answer depends on a number of factors:

- Size of the input DS versus size of the target table
- How clean the data to be uploaded is and if there might be any rejects
- Ratio of inserts X updates

To help determine the database load method to use, we provide the following key selection criteria:

- Bulk Loads
  - All rows of the input dataset are new to the target DB
  - There is an option of disabling indices and constraints:
    - Used when the cost of re-enabling indices and constraints is less then the cost of updating indices and evaluating constraints for each and every input row.
    - Used when the data is guaranteed to be thoroughly cleansed by the transformation phase, so when indices and constraints are re-enabled, they do not fail.
- Upserts
  - Required whenever the following is needed:
    - An existing record must be removed (Delete mode);
    - An existing record must be replaced (Delete then Insert mode);
    - An existing record must be updated.
  - For new records:

    The following options are available:

    - Insert
    - Update then Insert
    - Insert then Update

    Must be used in the event the data is not sufficiently clean (for instance, there is a chance new records might already be present in the target table). Duplicate and violating records can be caught through **reject links**;

    You must consider doing Bulk Loads for new records instead

– For updates to existing records:

  The following options are available:

  - Insert then Update mode

    #inserts > #updates (the number of records to be inserted is larger then the number of records to be updated);

  - Update then Insert mode

    #updates > #inserts (the number of records to be updated is larger than the number of records to be inserted).

  In the event the data is not sufficiently clean (that is, there is a chance the new records might already be present in the target table), invalid records can be caught through reject links.

The use of Upsert mode for updates is the natural way with DataStage database stages.

Bulk updates

There is one additional alternative that can be explored for applying bulk updates to target tables, assuming the incoming update records are guaranteed to be clean. Perform the following steps:

1. Bulk load the update records to a database temporary (or scratch) table
   a. This table should have neither indices nor constraints;
   b. The assumption is the input update records are cleansed and are guaranteed not to throw any SQL exceptions when processed against the target table. This cleansing must have been accomplished in the transformation phase of the ETL process;
2. Execute a single Update SQL statement that updates the target table from the records contained in the temporary table.

This approach is illustrated in Figure 15-15.

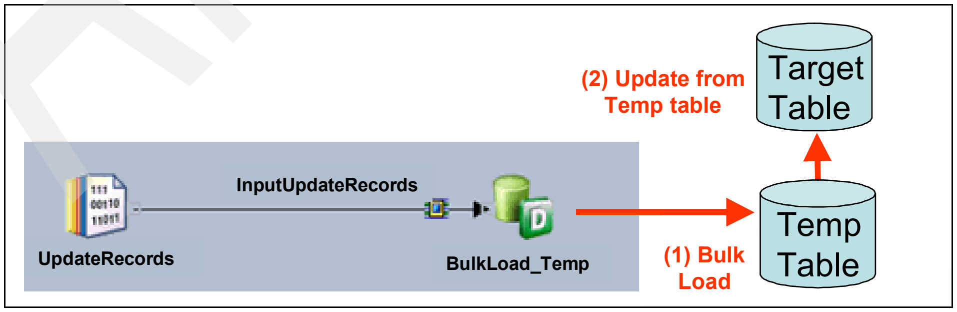


Figure 15-15 Combination of bulk load and SQL statement for processing bulk updates

The advantage of this approach is that it would bypass the inherent overhead of database call-level interfaces when submitting updates (which includes setting up and sending to the database server incoming rows in the form of arrays, and checking the return status for each and every array element).

The Bulk Update technique avoids that overhead because it assumes the data does not raise any SQL exceptions. There is no need for checking the return status on a row-by-row basis. This technique requires a closer cooperation with DBAs, who have to set up the necessary table spaces for the creation of the temp/scratch tables. Hopefully, the target database can do the two steps in parallel.

Real-time data flow design

Much of the DataStage history has focused on batch-oriented applications. Many mechanisms and techniques have been 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 there are concerns related 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 are related to bulk data processing.

Batch applications tend to be bound to time constraints, such as batch execution windows, which are typically require at least a few hours for daily processing.

However, there are several applications that do not involve the processing of huge amounts of data at once, but rather deal with several small, individual requests. We are not saying the number and size of requests are minimal. They might actually be relatively large, but not as much as in high volume batch applications.

For high volume scenarios, users typically resort to batch-oriented techniques. So, what is discussed in this chapter must not be used to implement the processing of daily incremental DW loads, for instance.

The types of applications in this chapter are focused on scenarios where there is large number of requests, each of varying size, spanning a long period of time.

The following list details these types of applications:

- Real-time transactions
  - B2C applications

    There is a person waiting for a response, such as in a web-based or online application.
  - B2B applications

    The caller is another application.
- Real-time reporting/decision making
  - There are few or no updates to database tables.
  - Involves fetching and massaging large amounts of data.
  - Generates support of a specific decision, such as insurance approval.
- Real-time data quality
  - Other applications invoke QualityStage for cleansing/de-duplication.
  - An example is the use of MDM.
- Integration with third party business process control systems
  - There is a business process coordinator that 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 JMS, as an example. A third party example is Tibco.
- Near-Real-time
  - Message-oriented for instant access and update.
  - Data trickled from a source mainframe to populate active data warehouses and reporting systems

There are two capabilities that have been added to DataStage over time, to address the needs of real-time:

- MQ/DTS (Distributed Transaction stage)

  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 UOW (Unit-of-Work) stage. The original target in DS 7.5 was Oracle. In InfoSphere DataStage 8.X, this solution has been substantially upgraded (incorporating the new database Connector technology for various database flavors) and re-branded as the Distributed Transaction stage.

- ISD (Information Services Director)

  ISD enables DataStage to expose DS jobs as services for service-oriented applications (SOA). ISD supports the following types of bindings:

  – SOAP
  – EJB
  – JMS

DTS and ISD work in different ways and serve different purposes. In this section we discuss the specifics of each. However there are common aspects that relate to both:

- 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 and every 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. They are a key optimization mechanism for high volume processing.

  – Real-time applications cannot afford to have records sitting in buffers waiting to be flushed.

  – The framework was adapted to support End-of-Waves, which force the flushing of memory buffers so responses are generated or transactions committed.

- Payload processing

  – Frequently batch applications have to 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 can be circumvented.

We do not go into the details of the deployment or installation of DTS and ISD. For that, the reader must see the product documentation.

16.1 Definition of real-time

The notion of real-time is not related to that which is typically used for embedded systems, such as avionics, traffic control, and industrial control. DS is not for embedded applications or for any type of scenario in which there are stringent timing constraints.

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

- Request/Response
  - As an example, there is a caller waiting for a response. They expect to receive a response quickly, but there is no physical tragedy, such as a transportation accident, that would occur if the response takes too long. It instead might be that the caller receives a timeout.
  - Typical examples:
    - B2C, such as Web applications
    - Online data entry applications
    - B2B
  - 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* has been 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).
  - There is no person waiting for a response.
  - Examples:
    - 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 be enhanced with additional capabilities to support these types of scenarios. This is because their implementation with batch applications is not feasible.

16.2 Mini-batch approach

A common scenario in the field is the attempt to use batch applications for near real-time scenarios (as defined in the previous section). In other words, instead of having perpetual jobs waiting for requests or messages to arrive, systems integrators and developers propose the use of batch jobs.

The idea is to accumulate input messages or requests to a certain extent, and fire a job sequence. The hope is that such job sequences can finish in a short amount of time. A common job duration of this type is 15 minutes, for instance.

These are the main problems with this approach:

- Extremely short execution windows

  Application can be complex and as a result jobs tend to be huge. The final result is that the total amount of time to start up and shut down all jobs in the mini-batch sequences does not fit in the planned time window.

- Target database Inconsistencies
  - The transactional consistency of the target database might be compromised. Batch jobs might resort to multiple database stages spread across multiple jobs. If the DS application is the only one updating the target database, impact might be minimized. However, if other applications update the same DB, there is a chance mini-batch updates conflict with other real-time applications.
  - Users see inconsistent views of the target database as the various mini-batch provisioning jobs apply updates to target tables. These updates are not part of a single unit-of-work and therefore the database state is not as consistent (even though database constraints such as parent-child relationships are maintained). For instance, a user might see an order header without its details. This can only be guaranteed by an effective unit-of-work.

In summary, we strongly advise against the use of mini-batches. Instead, use either DTS or ISD, depending on the nature of the interaction with the sources and targets.

16.3 Parallel framework in real-time applications

The advantages of the parallel framework are alive in the real-time world.

Service-oriented applications (SOAs), represented in their vast majority by application server-based implementations such as J2EE, consist of a number of services that are executed in a single thread of execution. Application servers can spawn lots of threads on behalf of incoming requests, but each request is serviced by a set of instructions that are executed one at a time in a strict order.

This execution model fits well for cases when there are absolutely no interdependencies between the requests, and the methods that implement the services are simple. As the picture becomes more complicated, this model reaches its limits, and that is where the parallel framework has an advantage.

Pipeline and partitioning parallelism allow for the breaking up of the application logic into several concurrent steps, and execution paths. This is depicted in Figure 16-1.

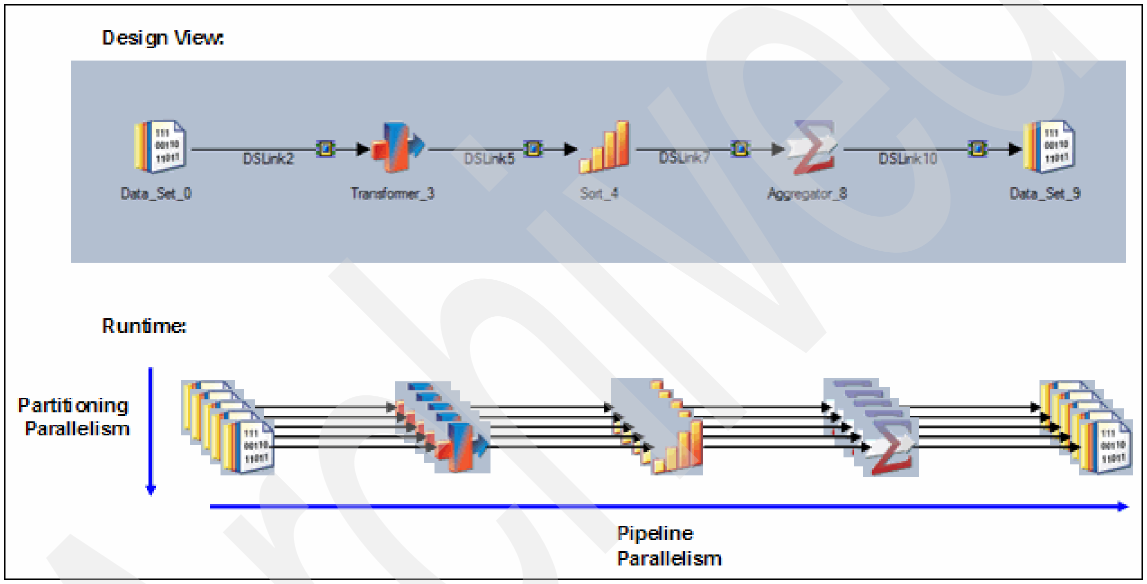


Figure 16-1 Pipeline and partitioning parallelism

There are at least two scenarios in which the parallel framework helps break through the limitations of a single-threaded execution model:

- Services of increased complexity

  In real-time decision-support applications (such in the sales of insurance policies), the amount of input and output data is small, but the service itself must fetch, correlate, and apply rules on relatively large amounts of data from several tables across data sources. This might involve doing lots of database queries, sorts/aggregations, and so forth. All this processing takes a considerable amount of time to be processed by a single threaded enterprise JavaBean method. With the parallel framework, the several steps cannot only be spread across multiple steps in a pipeline, but also have segments of the logic executing across multiple partitions.

- Interdependencies and strict ordering in message-oriented applications

  There are cases when a strict order must be imposed for the processing of incoming messages, for example when bank account information is transferred over to a decision support database. Account transactions must be applied in the exact order in which they are executed in the source system. In this scenario, the maximum throughput is determined by the efficiency of a single thread of execution. In these types of scenarios, partitioning parallelism is not applicable. However, these types of scenarios can greatly benefit from breaking up the logic into multiple pipelined steps. Instead of having a single thread of execution, there are multiple threads, each one executing on a separate OS process.

Interesting challenges are introduced by pipeline parallelism in the real-time world. However, the advantages exceed the challenges by far.

16.4 DataStage extensions for real-time applications

The parallel framework was extended for real-time applications. Three extended aspects are:

- Real-time stage types that keep jobs always up and running,
- End-of-wave
- Support for transactions in target database stages.

16.4.1 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 EOF to the next stage in the pipeline. In the Figure 16-1 on page 298, the source dataset passes on the EOF marker to the Transformer stage, which in turn passes it on to the sort, until it reaches the final output dataset. The job terminates when all target stages run to completion.

In real-time applications, this model is not efficient, as discussed in 16.2, “Mini-batch approach” on page 297. 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.

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

- ISD Input

  This is the input stage for Information Services Director applications. It is the same source stage, for a number of supported bindings (such as EJBs, SOAP, JMS)

- MQConnector
  - MQ reader stage in Information Server 8.X
  - For pre-8.X Information Server, the corresponding stage was named MQRead
- Custom plug-ins

  New stage types can be created for special purposes, using 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 un-deployed.
- 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.

16.4.2 End-of-wave

The ability for jobs to remain always-on introduces the need to flush records across process boundaries.

The parallel framework was originally optimized for batch workloads, and as such, implements the concept of buffers in the form of virtual datasets. These buffers are an important mechanism even for real-time jobs. However, if for a given request (either an MQ message or an ISD SOAP request) there are not enough records to fill virtual dataset buffers, they can wait until more requests arrive so the buffers are flushed downstream.

An End-of-wave (EOW) marker is a sort of 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 16-2 shows an example of how EOW markers are propagated through a parallel job. The source stage can be an MQConnector for the purpose of this illustration (in this case, the target stage is a DTS/Distributed Transaction stage).

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.

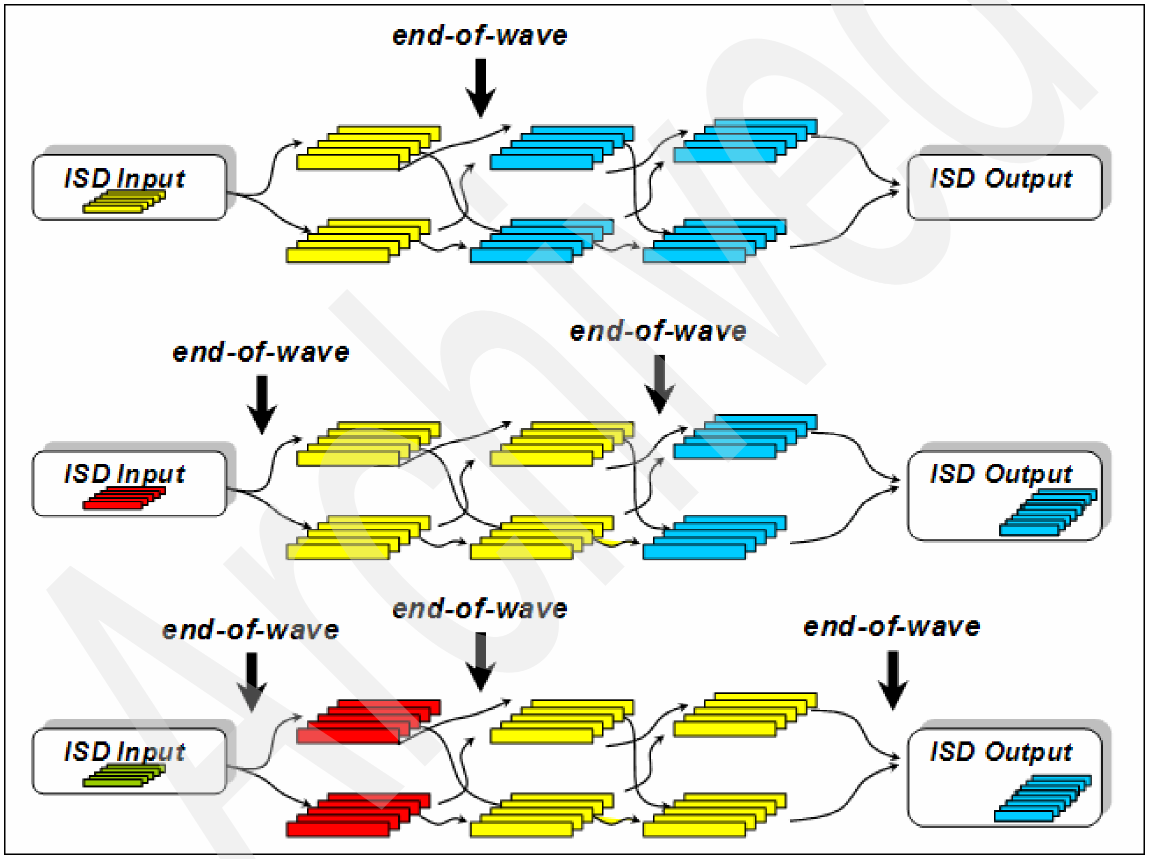


Figure 16-2 EOWs flowing through a parallel job

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

- ISD Input
  - It issues an EOW for each and every incoming request (SOAP, EJB, 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 big 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 means of the Information Services Console.
- MQConnector
  - It issues an EOW for one or more incoming messages.

    This is determined by a parameter named Record Count.
  - It issues an EOW after a certain amount of time has elapsed.

    This 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 stage's internal state must be reset, so a new execution context begins. For most stages, (Parallel Transformers, Modify, Lookups) this does not have any 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 re-connecting after each and every wave. The performance is poor.

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

- Sorts
- Aggregations

For these two stages, the corresponding logic is restricted to the set of records belonging to a certain wave. Instead of consuming all records during the entire execution of the job, the stage produces a partial result, just for the records that belong to a certain wave. This means that 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.

16.4.3 Transaction support

Support for real-time applications is not complete without 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 totally oblivious to what is going on in other database stages.

This is depicted in Figure 16-3.

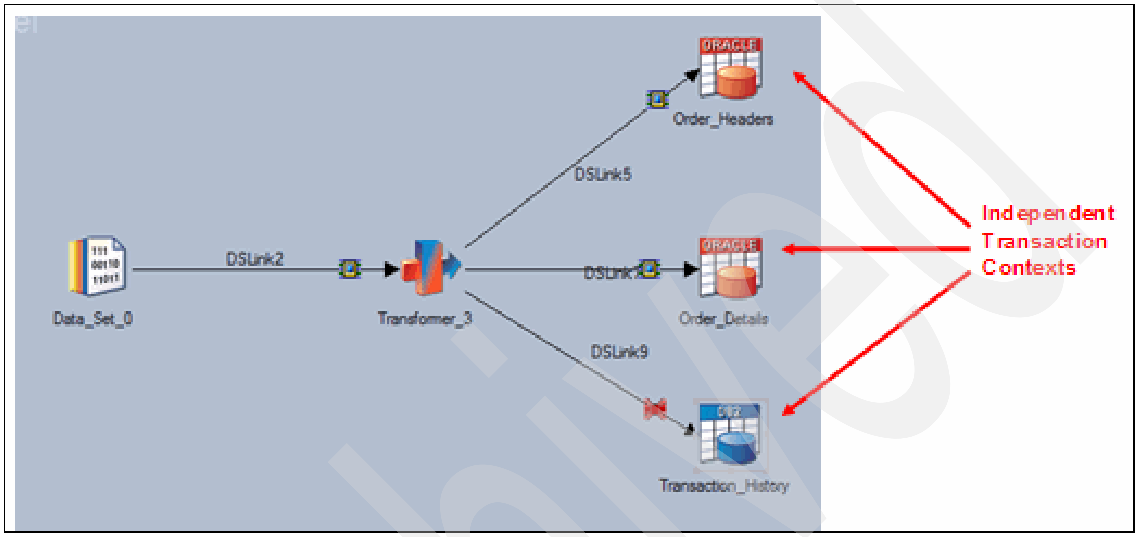


Figure 16-3 Transaction contexts with standard database stage types

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.

Pre-8.1, Information Services Director (ISD) and 7.X RTI jobs had no option other than resorting to multiple separate database stages when applying changes to target databases as part of a single job flow. The best they could do was to synchronize the reject output of the database stages before sending the final response to the ISD Output stage (See 16.7.4, “Synchronizing database stages with ISD output ” on page 353 for a discussion on 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, as well as when dealing with message-oriented and heterogeneous targets (DTS).

The Connector and DTS stages are discussed in the following subsections.

Connector stages

Information Server 8.5 has Connector stages for the following targets:

- DB2
- Oracle
- Teradata
- ODBC
- MQSeries

IS 8.5 Connectors support multiple input links, instead of a single input link in pre-8.5 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. This is depicted in Figure 16-4 on page 305.

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.

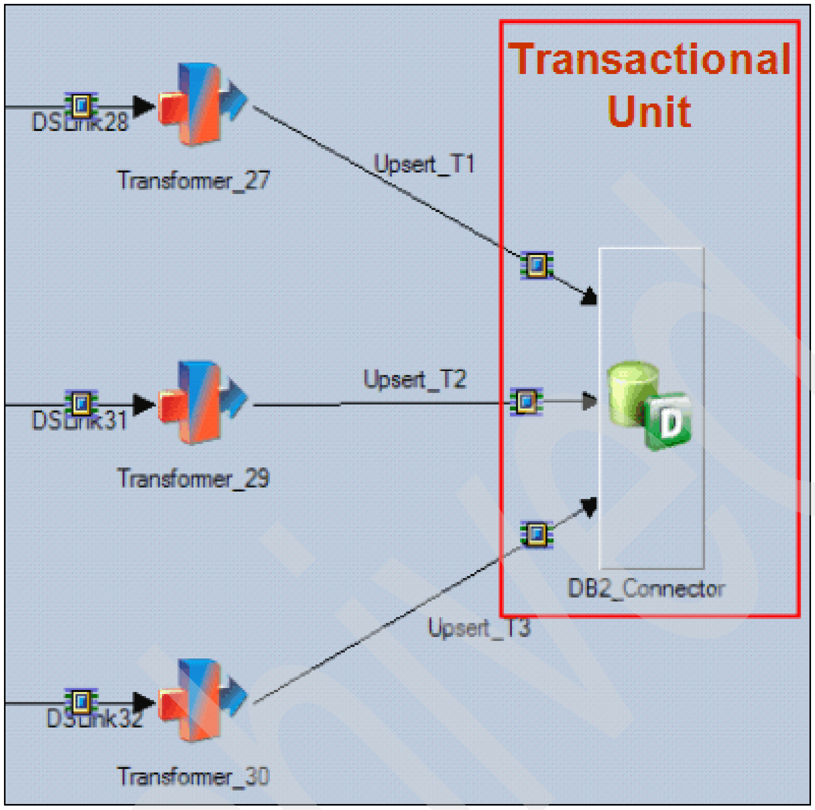


Figure 16-4 Multi-statement transactional unit with a connector for a single database type

With database Connectors, ISD jobs no longer have to cope with potential database inconsistencies in the event of failure. ISD requests might still have 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 across multiple database types and those that include guaranteed delivery of MQ messages, one 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 DTS.

This stage was originally introduced in DataStage 7.5, with the name of *UnitOfWork*. It was re-engineered for Information Server 8, and is now called DTS. It is depicted in Figure 16-5.

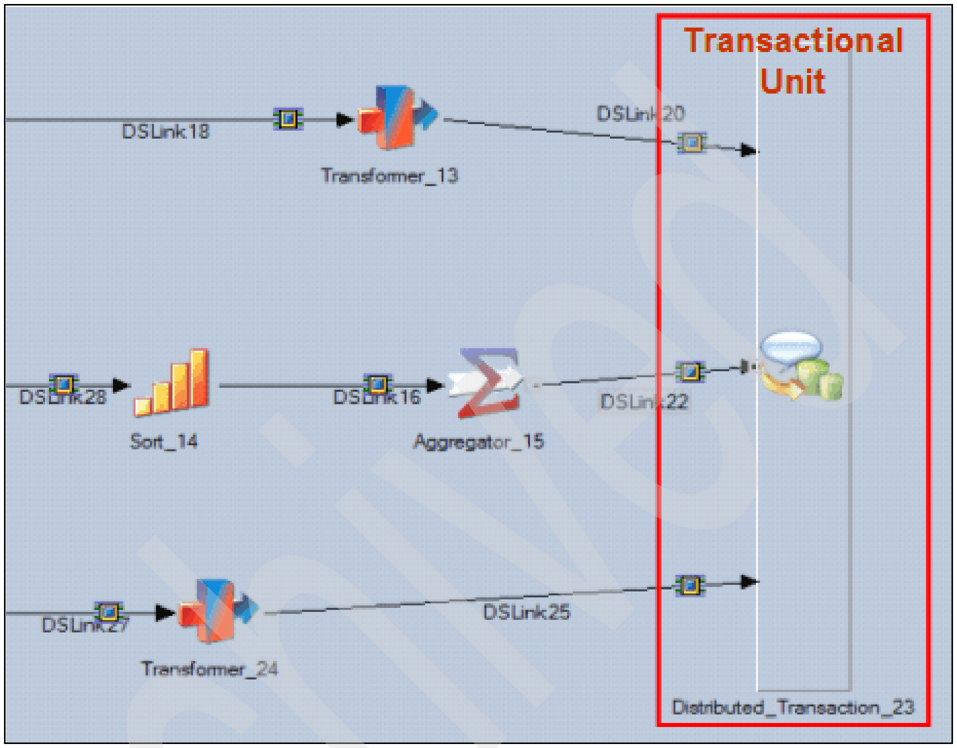


Figure 16-5 Distributed Transaction stage

The DTS is discussed in detail in 16.6, "MQConnector/DTS" on page 313. It applies to target databases all the SQL statements for incoming records for all input links for a given wave, as a single unit-of-work. Each wave is independent from the other.

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

- Guaranteed delivery from a source queue to a target database

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

- Once-and-only-once 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

DTS can also be used without a source MQConnector, such as in Information Services Director jobs. One still needs to install and configure MQ, because it is required as the underlying transaction manager.

16.5 Job topologies

In Chapter 15, “Batch data flow design” on page 259, we describe a series of techniques to be adopted to optimize batch-oriented applications. Unfortunately, one cannot have the exact same implementation applied to both real-time and batch scenarios. A real-time application must obey certain constraints (elaborated upon in this section).

Figure 16-6 on page 308 illustrates how a real-time job must be structured. The examples in this chapter use WISD stages, but they might be replaced with the MQConnector as the source and the DTS as the target. The same constraints apply to both message-oriented and SOA jobs.

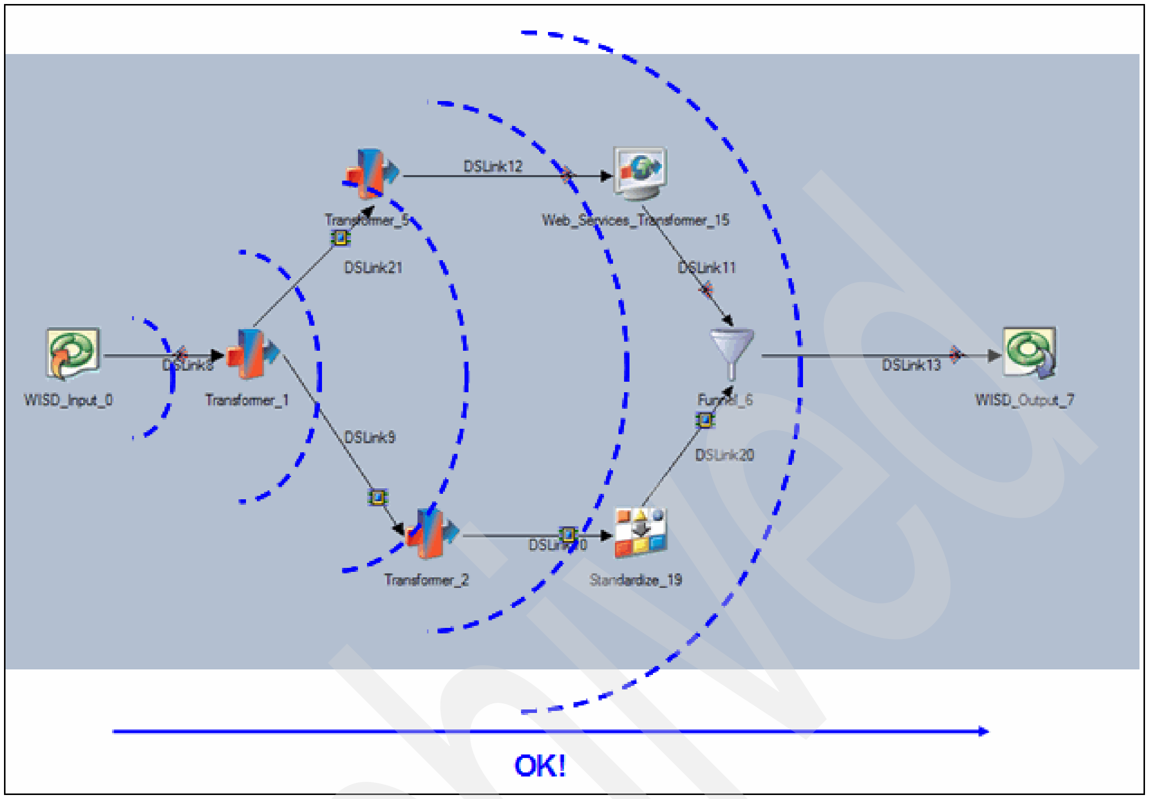


Figure 16-6   Valid real-time data flow with all paths originating from the same source

All the data paths must originate from the same source, the source real-time stage (either WISD Input or MQConnector). Figure 16-6 uses concentric blue dotted semicircles to illustrate the waves originating from the same source.

Real-time jobs might split the source into multiple branches, but they must converge back to the same target: either a DTS or WISD Output stage.

There can be multiple waves flowing through the job at the same time. You cannot easily determine the number of such waves. It is dependent on the number of stages and their combinability. The larger the job, the higher the number of possible waves flowing through the job simultaneously.

Although originating from the exact same source, EOW markers might flow at various speeds through different branches. This is dependent on the nature of the stages along the data paths. In the example of Figure 16-6, the upper branch might be slower because it involves a remote Web Service invocation, which tends to be significantly slower than the standardization on the lower branch.

However, the framework makes sure EOW markers are synchronized whenever they reach join stages such as the Funnel. The term “Join” is being used loosely and refers to any stages that accept multiple inputs and support a single output stream link, such as Join, Merge, Funnel, ChangeCapture, and Compare.

The need to restrict flow designs so that all data paths originate from the same source restrict the types of jobs and stages that can be made use of. There is one exception to this rule that is discussed later (consisting of data paths that lead to the reference link of Normal Lookups).

Figure 16-7 presents one example of an invalid flow. The Funnel stage has two inputs, one of them originating from the WISD input stage. The second input (marked in red waves) originates from an Oracle Enterprise stage. The semantics of this layout remain undefined and as a result it cannot be adopted as a valid real-time construct. All data paths entering the Funnel stage must originate from the same source. This example also applies to other Join stages, such as Join, Merge, and ChangeCapture.

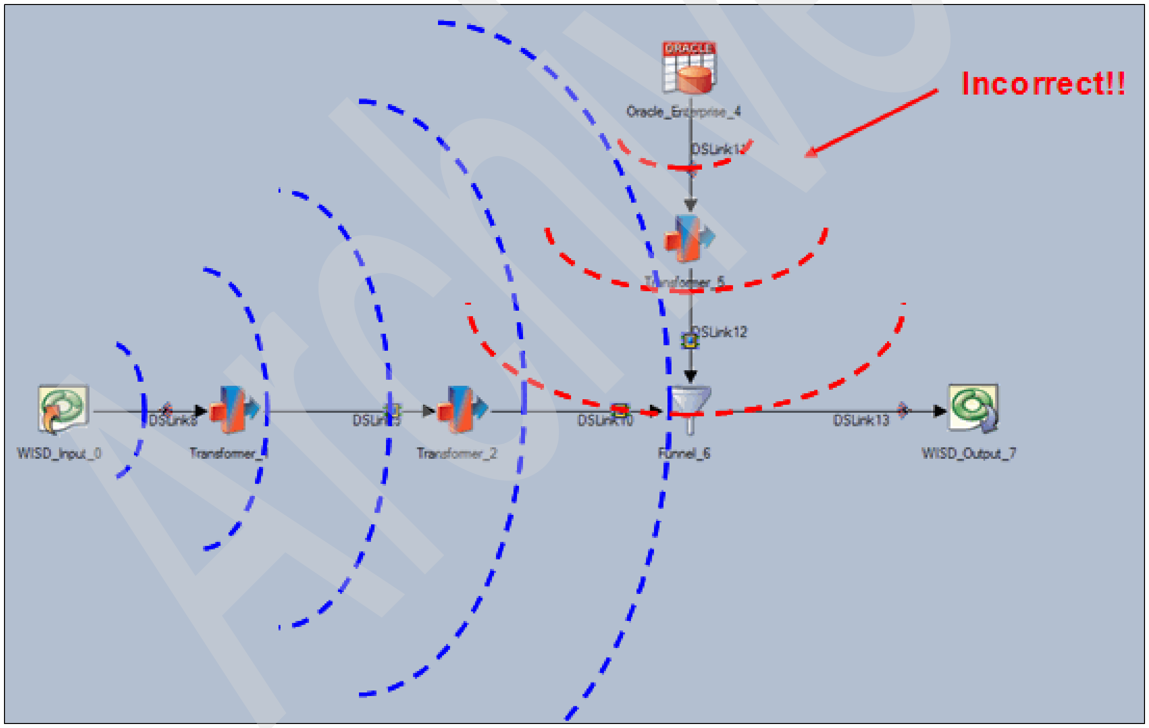


Figure 16-7 Invalid real-time data flow with multiple Source stages

The query in the Oracle Enterprise stage of Figure 16-7 on page 309 is executed only once for the duration of the entire job life. The Oracle EE stage never issues an EOW marker, and no matter how many EOWs are issued by the WISD input stage, the Oracle EE stage is totally oblivious to what is going on the blue branch. The WISD EOWs in no way affect the behavior of the Oracle EE stage.

Figure 16-8 presents another scenario, similar to the one in Figure 16-7 on page 309. The Merge stage has an input that originates from an Oracle Enterprise stage. As stated before, Merge and Join stages are perfectly legal stages in real-time job flows. However, this is the case in which records from two separate sources are being correlated by a common key. In other words, this Merge stage is acting as a lookup.

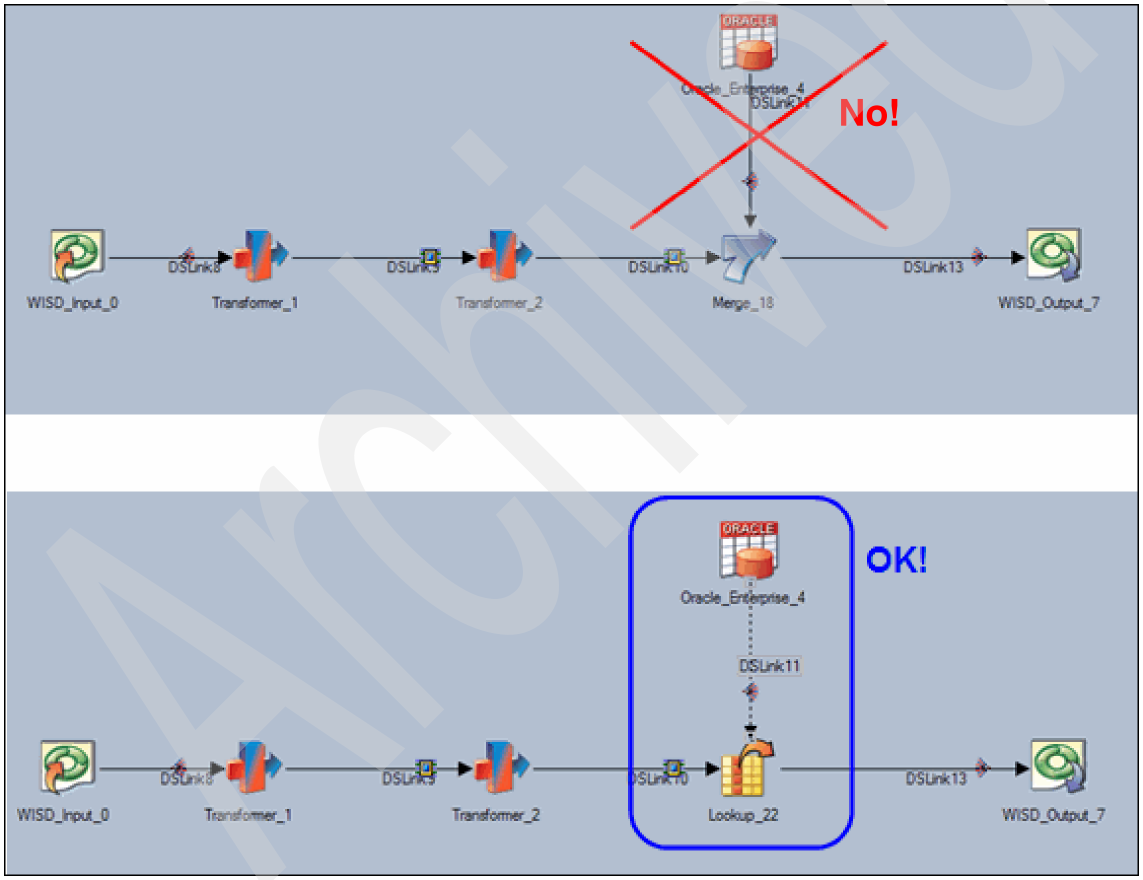


Figure 16-8 Using a Sparse or Normal Lookup instead of a Join

To make the job of Figure 16-8 on page 310 real-time compliant, we need to replace the Merge with a Normal Lookup stage. A Normal Lookup stage is a two-phase operator, which means in its first phase, immediately after job start up, the Lookup stage consumes all records from its reference link and builds a lookup table that is kept in memory for the duration of the job. In the second phase, the Lookup stage processes incoming records that arrive through its stream input, performing the actual lookup against the in-memory table according to the lookup criteria.

The flow in Figure 16-7 on page 309, although containing a separate branch that does not originate from the same WISD input stage, is valid, because the output from the Oracle Enterprise stage is read only once during the startup phase. For the duration of the entire job, the lookup table remains unchanged in memory.

Therefore, one can have branches originating from independent sources, as long as those branches lead to reference links of Normal Lookups. Such branches can have multiple stages along the way and do not have to necessarily attach the source stage directly to the reference link.

It is important to note that for Normal Lookups the in-memory reference table does not change for the entire duration of the job. This means Normal Lookups must only be used for static tables that fit comfortably in memory.

The same layout of Figure 16-8 on page 310 can be used for a Sparse Lookup, in case the reference data changes frequently and the incoming records need to do a Lookup against the always-on data in the target database.

For reference links of Normal Lookups, the opposite of the previously stated rule applies. Figure 16-9 illustrates a case where the reference link receives records from the a branch that originates from the source WISD stage.

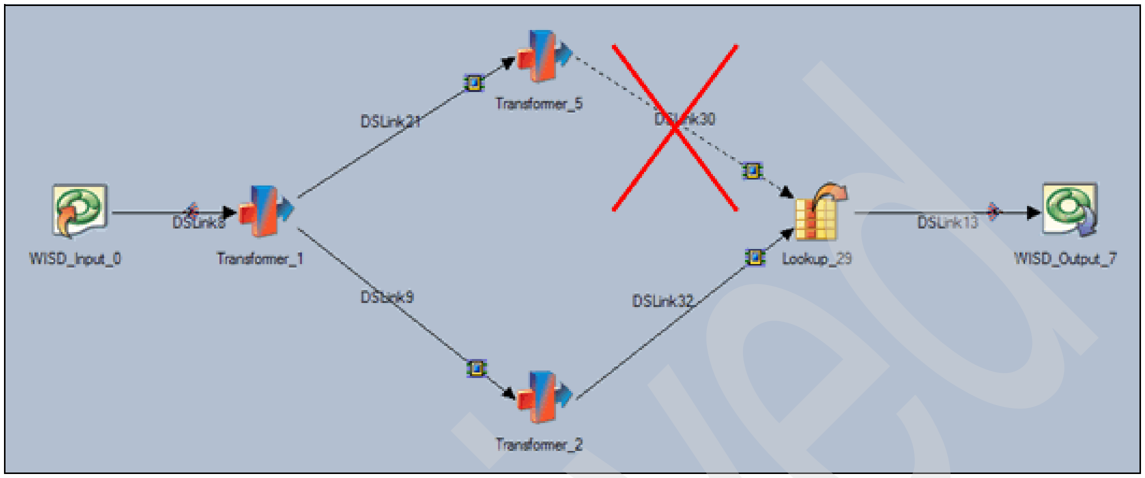


Figure 16-9 Invalid use of a Lookup stage in a real-time job

The two-phase nature of the Normal Lookup stage renders this design invalid. The lookup table is built only once, at stage startup. However, the flow requires EOWs to flow throughout all branches. For the second EOW, the upper branch remains stuck, as the Lookup stage does not consume any further records. Therefore, this flow does not work.

16.5.1 Summary of stage usage guidelines

In summary, the following general stage usage guidelines apply to always-on, real-time data flows:

- Normal Lookups
  - Must only be used for lookups against static reference datasets.
  - The reference link must always originate from an independent branch.
- Join stages
  - Stages included in this list are Funnel, Join, Merge, Compare, ChangeCapture.
  - All input links for Join stages must originate from the source real-time stage (WISD Input or MQConnector).

- Sparse Lookups
  - Must be used whenever there is a need to lookup against non-static, live reference data repositories;
  - Sparse Lookups are not limited to returning a single record. They might return multiple records, which is a valid construct for real-time jobs. The returned records might be further processed by transformations, aggregations, sorts, joins, and so forth.
  - Sparse Lookups are one of the predominant stage types in real-time, always-on jobs, along with Parallel Transformers.

16.5.2 ISD batch topologies

ISD jobs might also conform to two batch topologies, which are not the focus of this chapter:

- A job with no ISD Input, and with one ISD Output;
- A job with no ISD Input and no ISD Output.

These ISD job types are re-started for each request and might contain any data flow patterns, including the ones that are okay for batch scenarios. The only difference between them is the first requires data flow branches to be synchronized before reaching the ISD Output. However, they can both have branches that originate independently of each other.

16.6 MQConnector/DTS

The solution for transactional processing of MQ messages was originally implemented in DataStage 7.X as a combination of two stages:

- MQRead
- Unit-Of-Work

These stages provided the guaranteed delivery from source MQ queues to target Oracle databases, using MQSeries as the transaction manager. Those stages were re-engineered in Information Server 8, and are now known as the MQConnector and DTS stages.

The main difference is that they now implement a plug-able solution, taking advantage of DataStage Connectors and supporting multiple types of database targets, such as DB2, Oracle, Teradata, ODBC and MQ.

This plug-ability is implemented behind the scenes and is not a design feature available to users. Users cannot build new components by plugging in existing connectors to build new components. Instead, it is a feature that eases the engineering effort to extend the capabilities of database interfacing. This is visible through the DTS stage that, starting in IS 8.5, allows for multiple target stage types as part of a single transaction.

DataStage historically had a set of database stages, some of them from the existing DS Server (referred to as *Plug-in stages*) as well as a set of native parallel database operators (such as Oracle EE, DB2 EE, Teradata EE, and so forth). IBM defined a new common database interface framework, known as Connector stages, which is the foundation for a new set of database stages.

The following types of Connectors are available:

- Oracle
- DB2
- Teradata
- MQSeries
- ODBC

Information Server 8.1 supports a single database type as the target for DTS. Information Server 8.5 supports different target types as part of the same unit of work.

16.6.1 Aspects of DTS application development

There are three main aspects involved in the development and deployment of DTS jobs:

- The installation and configuration of the MQConnector and DTS stages

  This involves identifying any fixpacks or updates that need to be downloaded and installed so the Connector stages become available, both on the client and the server sides.

  – Requires the installation of corresponding database clients and MQSeries software.

  – Requires configuring XA resource managers and building an xaswit file.

- The design of the MQ/DTS job flows

  The rules for design topology in 16.6.4, “Design topology rules for DTS jobs” on page 316 must be obeyed.

- The configuration of the runtime environment

  This is based on the ordering and the relationships between incoming messages. It determines the scalability of the implementation.

  – Determining the underlying configuration of work queues and number of concurrent partitions.

  – Setting the maximum number of messages that can flow through a job simultaneously.

The job design must focus on obtaining a job flow that addresses business requirements and, at the same time, complies with the constraints laid out in 16.5, “Job topologies” on page 307 and 16.6.4, “Design topology rules for DTS jobs” on page 316.

There might be cases when there are concurrency conflicts that prevent multiple messages from flowing through the job at the same time. In other words, the job must somehow hold back messages from the source queue until the message that is currently being processed is committed to the database. The third bullet point is further elaborated in 16.6.11, “Database contention ” on page 339 and 16.10, “Pipeline Parallelism challenges” on page 366. The job design tends to be independent from the runtime configuration.

16.6.2 Reference documentation

In this document, we do not intend to fully reproduce and replace the information contained in the following two documents:

- *The DTS Functional Specification*

  This document describes the DTS functionality and a detailed description of the work queue topologies.

- *Setting Up and Running DTS Jobs*

  This document provides information about the following topics:

  – Installing and configuring MQConnector and DTS.
  – Configuring XA resource managers.
  – Shows sample DTS jobs.

16.6.3 A sample basic DTS job

Figure 16-10 presents a sample DTS job, with a basic pattern that consists of four parts:

- The source MQConnector
- The parsing of the message payload into a DS record format
- The transformation of the message content according to business rules
- The execution of the transaction against the target database with the DTS stage

Real-time jobs tend to be quite large, but that is the basic pattern to which they tend to adhere.

The Column Import stage is sometimes replaced with an XMLInput stage, but the result of the second step is always a set of records with which the rest of the DS stages in the job can work.

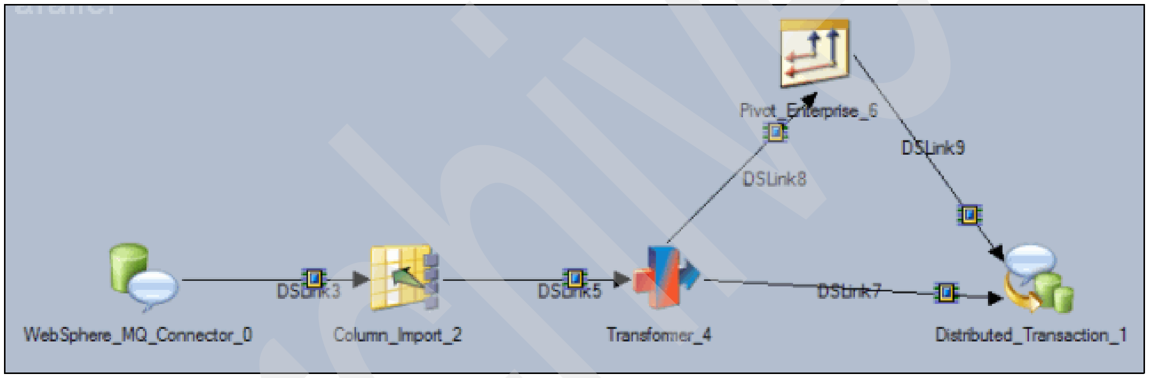


Figure 16-10 Sample DTS job

16.6.4 Design topology rules for DTS jobs

There must be one source MQConnector stage and one target DTS stage.

The following rules from 16.5, “Job topologies” on page 307 must be obeyed:

- All data paths must originate from the same source stage (the DTS Connector).

  The exceptions are reference links to Normal Lookups. These must originate from independent branches.

- All data paths must lead to the same target DTS stage

  There might be dangling branches leading to Sequential File stages, for instance, but these 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.

16.6.5 Transactional processing

The MQ/DTS solution provides the following benefits:

- Guaranteed delivery of messages

  No messages are discarded unless the corresponding transactions are successfully committed to the target database.
- Once-and-only-once semantics
  - Transactions on behalf of incoming messages are executed only once.
  - If a job fails in the middle of a transaction, the entire transaction is rolled back, and the original message remains either in the source queue or a so-called work queue.
  - Once-and-only-once semantics are only available for XA-Compliant resource managers:
    - Oracle
    - DB2
    - MQ
- Non-XA-compliant resource managers
  - There is guaranteed delivery, but transactions might have to be replayed in the event of job restart after failure.
    - Non-XA-Compliant RMs: Teradata and ODBC
    - Job logic must be implemented so transactions are IDEMPOTENT

      That is, if replayed once or more with the same input parameters, they must yield the same result in the target DB.

What makes this possible is that the MQConnector and DTS stages interact with each other by means of work queues.

Work queues are not visible on the DS canvas. They are referenced by means of stage properties.

The diagram in Figure 16-11 shows the relation between the source and target stages through the work queue.

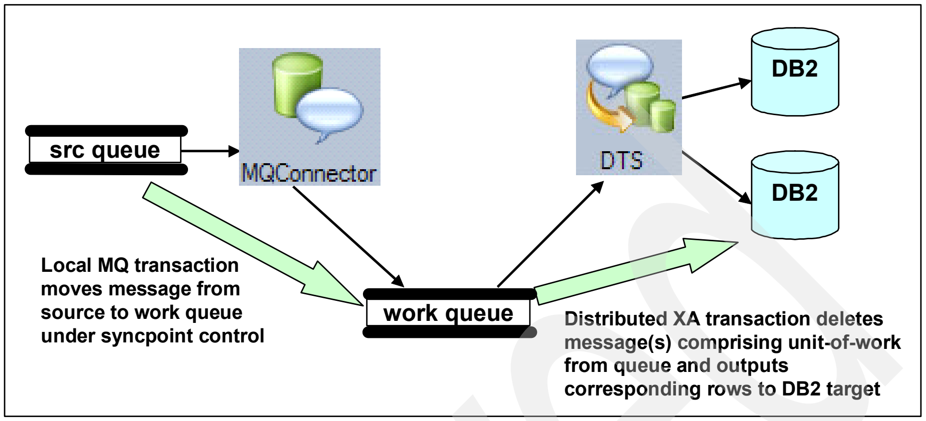

src queue
MQConnector
DTS
DB2
DB2
work queue
Local MQ transaction moves message from source to work queue under syncpoint control
Distributed XA transaction deletes message(s) comprising unit-of-work from queue and outputs corresponding rows to DB2 target

Figure 16-11   Transaction support with MQConnector and DTS stages

The MQConnector transfers messages from the source to a work queue through a local MQ transaction. It also forwards the payload to the next stage through its output link. Typically, the next stage is a Column Import or XMLInput stage, to convert the payload to DS records.

Once forwarded downstream, the message is guaranteed to be safely stored in the work queue. Depending on the setting in the source MQConnector, this stage might issue an EOW for each message, or for a group of messages.

On the target side, the DTS receives records from transformation stages along multiple data paths. There might have been Parallel Transformers, Sparse Lookups, Joins, Merges, and so forth. Those rows are stored by DTS in internal arrays.

Upon receiving an EOW marker (which was, again, originated from the source MQConnector), the target DTS stage performs the following tasks:

1. Invokes the transaction manager (MQSeries) to begin a global transaction.
2. Executes the various SQL statements for the rows that arrived as part of that wave according to a certain order.
3. Removes from the work queue one or more messages that were part of this wave.
4. Requests the Transaction Manager to commit the transaction

   This is a global transaction that follows the XA protocol, involving MQSeries and the target DBs.

Figure 16-11 on page 318 illustrates the fact that there is a close cooperation between the source and target stages to ultimately achieve the guaranteed delivery and once-and-only-once semantics.

16.6.6 MQ/DTS and the Information Server Framework

It might not be clear at first how the Information Server layers and parallel jobs fit together, so the following explanation is helpful.

Figure 16-12 illustrates how an MQ/DTS job fits in the Information Server Framework.

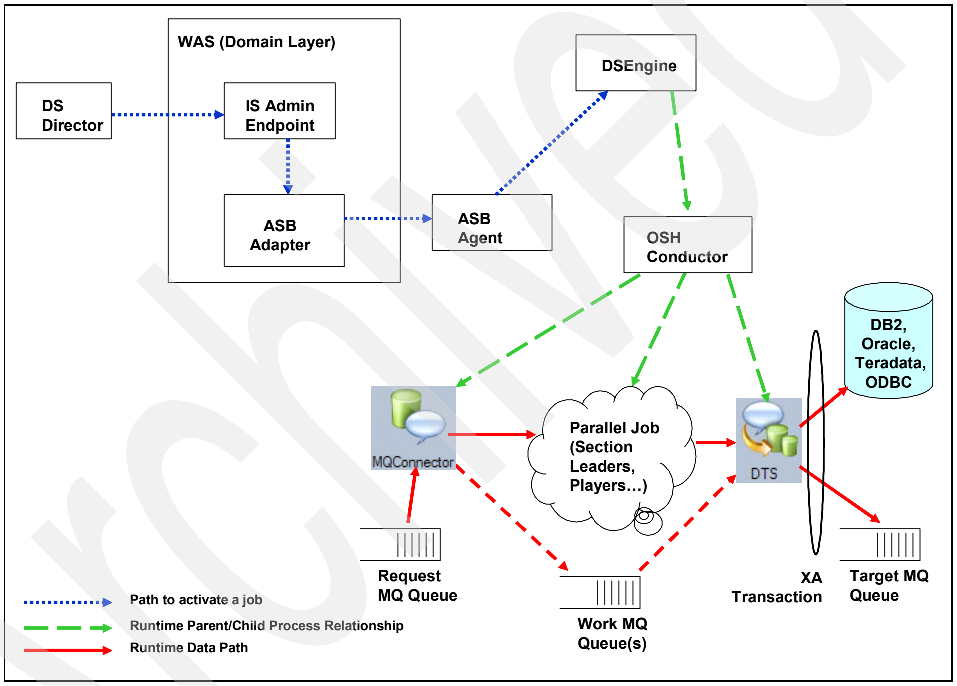


Figure 16-12 MQ/DTS and the Information Server framework

There are four major layers:

- The DS client

  Represented by DS Director
- The domain (or services) layer

  The WebSphere Application Server
- The engine

  Composed by the DS Engine and the ASB Agent
- The parallel framework processes

  These are the processes that are executed on behalf of a parallel job, doing the actual data extractions, transformations and load.

The illustration does not include the XMeta layer, which is not relevant for this discussion. The XMeta database is accessed directly by the domain layer, mostly at design time.

There are three major paths depicted in the figure:

- Job activation

  Denoted by dotted (blue) arrows.
- Parent/child relationship of runtime processes

  Denoted by large dashes (green) arrows.
- Data paths

  Denoted by solid (red) arrows.

There are several other processes and modules, so the figure and this discussion could go down to a finely grained level of detail. We chose the elements that are relevant for the level of abstraction that is appropriate for this discussion.

Job activation

To activate a job, the operator user opens DS Director, selects a job or sequence for execution, and issues the execute command. The DS Director client sends the request to the domain layer. Inside the domain layer, an administration endpoint forwards the request to the Application Service Backbone (ASB) adapter, which is sent to the ASB agent and finally reaches the DSEngine (at this moment, there is a dsapi_slave process, which is the DSEngine process that is activated on behalf of a client connection). Note that we do not provide any discussion on how DSEngine connections are established.

Parallel job activation

Upon receiving the `execute` command, the DSEngine spawns an OSH process. This process acts as the conductor and parent of all parallel framework processes that are launched on behalf of the job being started.

The dsapi_slave becomes the parent of the OSH conductor that, in turn, is the parent of all other parallel processes, including the MQConnector and DTS operators.

The parent/child relationship between the DSEngine and the OSH processes is denoted by the dotted green line.

Job execution

The activated processes establish communication channels among the virtual datasets, namely the virtual datasets.

The path followed by the actual data being processed is denoted by the small dashes (red) line. The data flows entirely in the realm of parallel framework processes. There is no participation by the domain layer (as opposed to Information Services Director).

16.6.7 Sample job and basic properties

In this section we discuss basic properties for the MQConnector and DTS stages that one would set to develop a working job. We use Figure 16-11 on page 318 as the example.

The reader should see the product documentation for a detailed description of all stage and link properties for these two stages.

In our discussion, the pictures combine aspects of the job design, with the job flow towards the top of the picture and stage and link properties at the bottom. Associations between the job flow and stage/link properties are marked as blue arrows. Comments on individual properties are marked in red. We believe this provides for a more concise characterization of the properties that are otherwise scattered throughout separate windows.

Linking the source and target stages

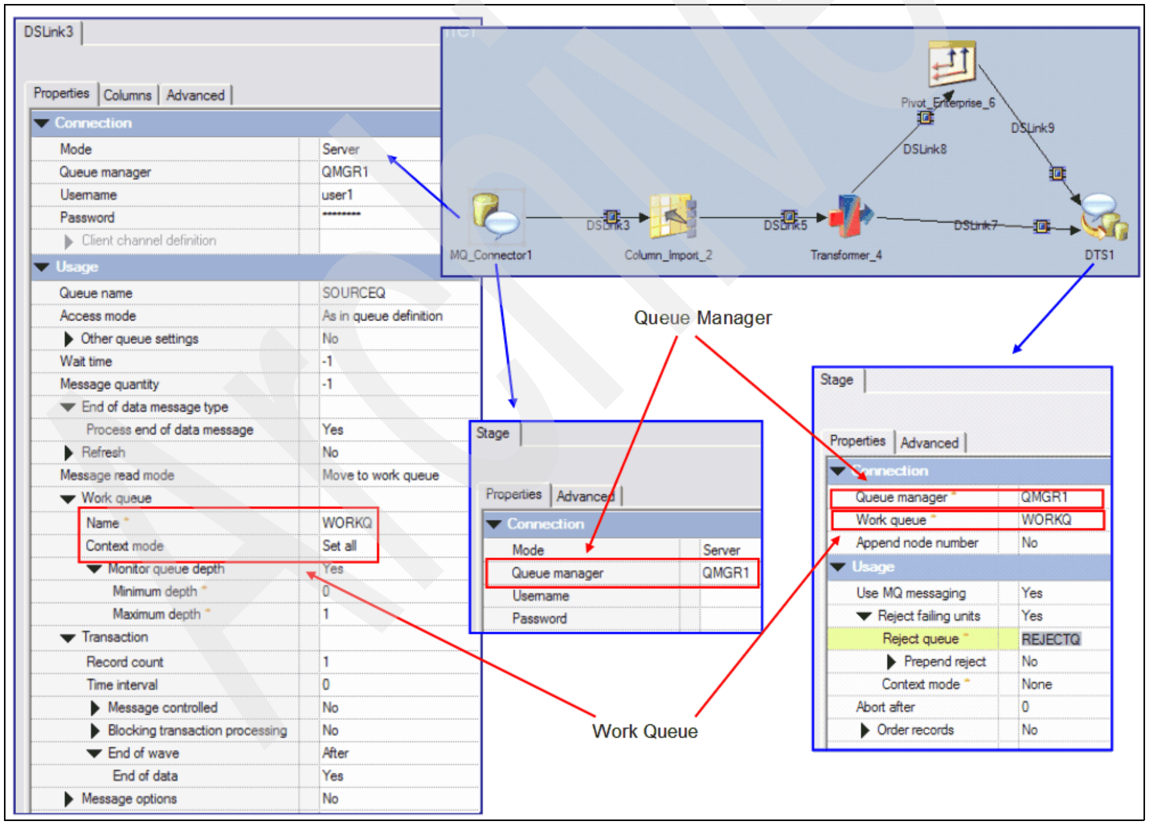
Figure 16-13 details properties that provide for the linking between the source and target stages. Both stages need to be assigned the same values for the following elements:

- ► Queue Manager Name
- ► Work Queue Name

Linking by means of a queue manager and a work queue is not apparent on the design canvas. It is, instead, expressed as stage and link properties.

For the DTS connector, the Queue Manager Name can be set either on the stage or link properties (it supports a single output link).

At runtime, the work queue name might be appended with a partition number, depending on the chosen runtime topology. This is discussed in 16.6.8, "Runtime Topologies for DTS jobs" on page 326.



DSLink3

Properties | Columns | Advanced

Connection	
Mode	Server
Queue manager	QMGR1
Username	user1
Password	********
Client channel definition	
Usage	
Queue name	SOURCEQ
Access mode	As in queue definition
Other queue settings	No
Wait time	-1
Message quantity	-1
End of data message type	
Process end of data message	Yes
Refresh	No
Message read mode	Move to work queue
Work queue	
Name *	WORKQ
Context mode	Set all
Monitor queue depth	Yes
Minimum depth *	0
Maximum depth *	1
Transaction	
Record count	1
Time interval	0
Message controlled	No
Blocking transaction processing	No
End of wave	After
End of data	Yes
Message options	No

Stage

Properties | Advanced

Connection	
Mode	Server
Queue manager	QMGR1
Username	
Password	

Stage

Properties | Advanced

Connection	
Queue manager *	QMGR1
Work queue *	WORKQ
Append node number	No
Usage	
Use MQ messaging	Yes
Reject failing units	Yes
Reject queue *	REJECTQ
Prepend reject	No
Context mode *	None
Abort after	0
Order records	No

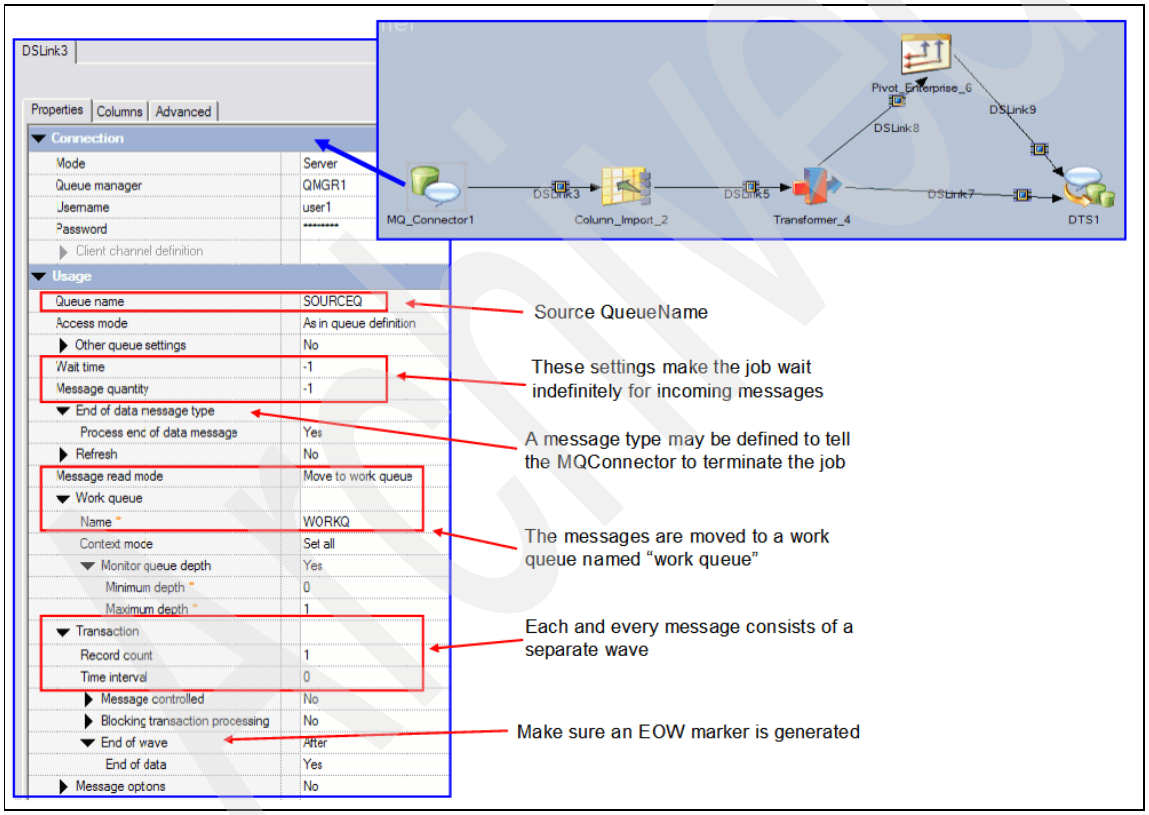
Figure 16-13 Queue manager and work queue information in Source and Target stages

MQConnector output link properties

Figure 16-14 indicates the basic properties one would have to set on the output link for the MQConnector:

- Source queue name
- Message wait options
- Transaction size
- Generation of EOW markers

The options in this example make the job wait indefinitely for messages to arrive in a source queue SOURCEQ. Every message is written to the work queue as a separate transaction. For each transaction, an EOW marker is sent downstream through its output link.



Connection	
Mode	Server
Queue manager	QMGR1
Username	user1
Password	********
Client channel definition	
Usage	
Queue name	SOURCEQ
Access mode	As in queue definition
Other queue settings	No
Wait time	-1
Message quantity	-1
End of data message type	
Process end of data message	Yes
Refresh	No
Message read mode	Move to work queue
Work queue	
Name *	WORKQ
Context mode	Set all
Monitor queue depth	Yes
Minimum depth *	0
Maximum depth *	1
Transaction	
Record count	1
Time interval	0
Message controlled	No
Blocking transaction processing	No
End of wave	After
End of data	Yes
Message options	No

Source QueueName

These settings make the job wait indefinitely for incoming messages

A message type may be defined to tell the MQConnector to terminate the job

The messages are moved to a work queue named "work queue"

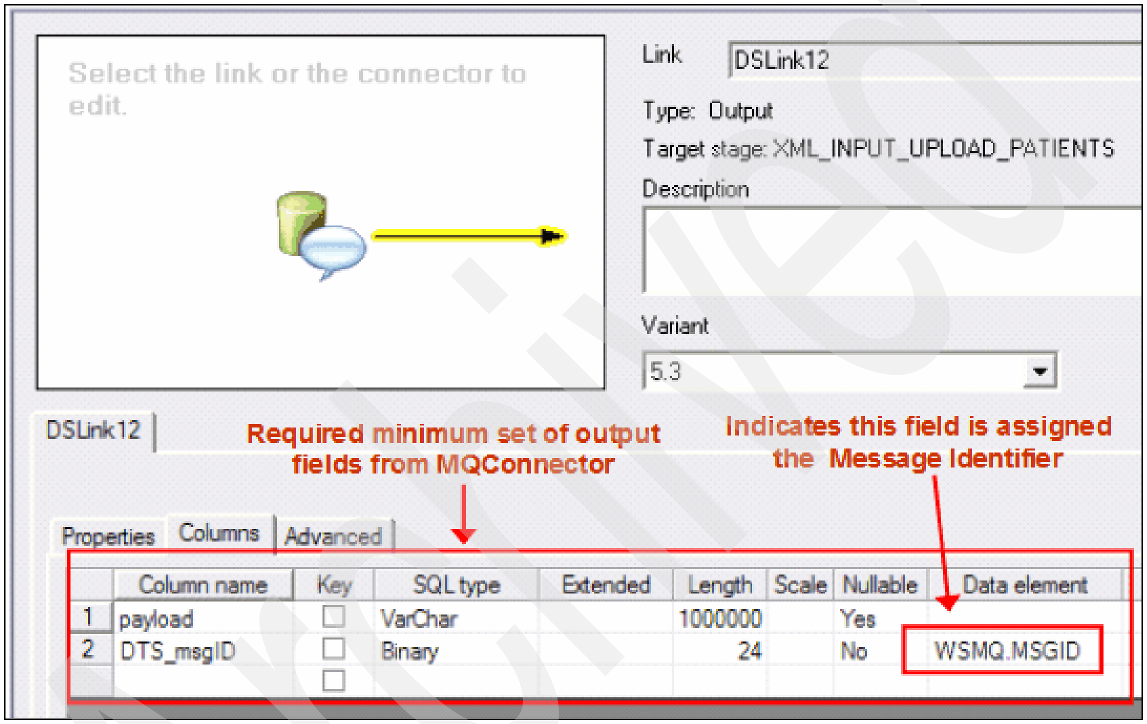
Each and every message consists of a separate wave

Make sure an EOW marker is generated

Figure 16-14 Basic MQConnector output link properties

You can also set an end of data message type that causes the MQConnector to terminate upon receiving a message of a certain type. The message type is a value that is set as an MQ message header property. This message can be either discarded or sent downstream and written downstream depending on the "Process end of data message" option.

Figure 16-15 presents a minimal set of columns that must be present in the output link for the MQConnector stage.


Select the link or the connector to edit.
DSLink12
Required minimum set of output fields from MQConnector
Indicates this field is assigned the Message Identifier
Properties
Columns
Advanced

Link: DSLink12

Type: Output

Target stage: XML_INPUT_UPLOAD_PATIENTS

Description

Variant: 5.3

	Column name	Key	SQL type	Extended	Length	Scale	Nullable	Data element
1	payload	☐	VarChar		1000000		Yes	
2	DTS_msgID	☐	Binary		24		No	WSMQ.MSGID
		☐						

Figure 16-15 Output link metadata for the MQConnector stage

The message payload must be sufficiently large to contain the largest expected message. The APT_DEFAULT_TRANSPORT_BLOCK_SIZE environment variable might have to be adjusted accordingly.

Each and every output row must have a field containing the MQ message ID (DTS_msgID). For the MQConnector, this field can assume any name. MQConnector knows it must assign the message ID to this field by means of the data element property, which must be set to WSMQ.MSGID.

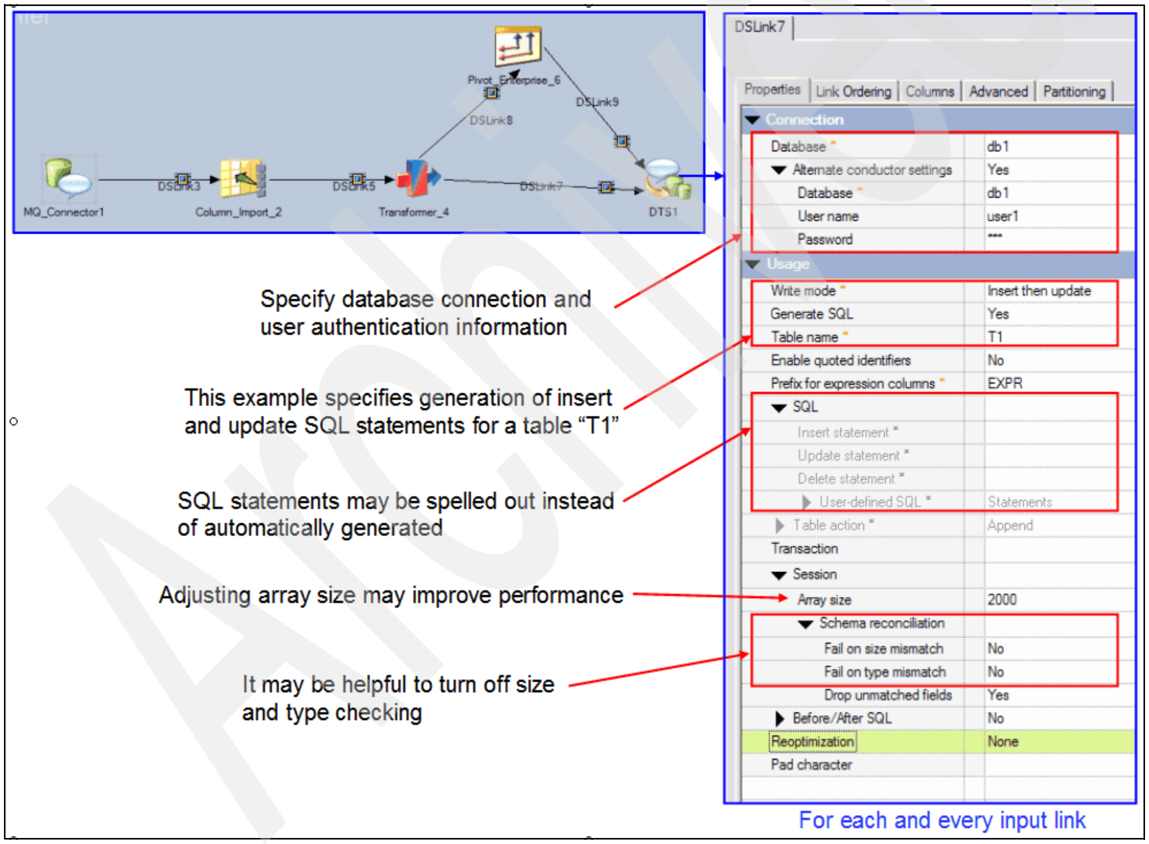
However, the DTS stage requires the message ID to be present in each of its input links with the name DTS_msgID, so it is better to name the field this way from the start.

There are other MQ header field values that can be added to the record definition, by selecting the appropriate type from the list of Data Element values.

Input link properties for the DTS stage

On the target side, Figure 16-16 highlights the minimal set of properties that you would have to set for each and every input link to obtain a working DTS job:

- Database Connection
- Write Mode
- SQL Generation or User-Defined Statements

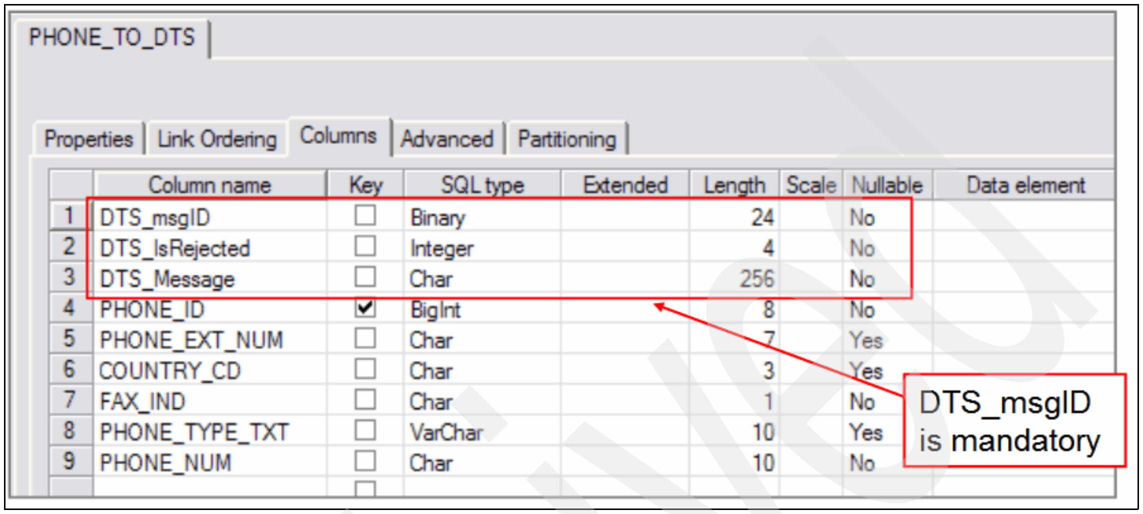

MQ_Connector1
DSLink3
Column_Import_2
DSLink5
Transformer_4
DSLink8
Pivot_Enterprise_6
DSLink9
DSLink7
DTS1
DSLink7
Properties
Link Ordering
Columns
Advanced
Partitioning
Specify database connection and user authentication information
This example specifies generation of insert and update SQL statements for a table "T1"
SQL statements may be spelled out instead of automatically generated
Adjusting array size may improve performance
It may be helpful to turn off size and type checking
For each and every input link

Connection	
Database *	db1
Alternate conductor settings	Yes
Database *	db1
User name	user1
Password	***
Usage	
Write mode *	Insert then update
Generate SQL	Yes
Table name *	T1
Enable quoted identifiers	No
Prefix for expression columns *	EXPR
SQL	
Insert statement *	
Update statement *	
Delete statement *	
User-defined SQL *	Statements
Table action *	Append
Transaction	
Session	
Array size	2000
Schema reconciliation	
Fail on size mismatch	No
Fail on type mismatch	No
Drop unmatched fields	Yes
Before/After SQL	No
Reoptimization	None
Pad character	

Figure 16-16 Basic DTS properties for input links

The Connector stages support strict checking of field type and length. The user has the option to turn this checking off.

Figure 16-17 depicts an example of input link metadata for the DTS.



	Column name	Key	SQL type	Extended	Length	Scale	Nullable	Data element
1	DTS_msgID	☐	Binary		24		No	
2	DTS_IsRejected	☐	Integer		4		No	
3	DTS_Message	☐	Char		256		No	
4	PHONE_ID	✓	BigInt		8		No	
5	PHONE_EXT_NUM	☐	Char		7		Yes	
6	COUNTRY_CD	☐	Char		3		Yes	
7	FAX_IND	☐	Char		1		No	
8	PHONE_TYPE_TXT	☐	VarChar		10		Yes	
9	PHONE_NUM	☐	Char		10		No	

Figure 16-17 Input link metadata for the DTS stage

The rules for input link columns are as follows:

- There must be a column called DTS_msgID, which carries the MQ message ID of the source message.
- Any column that does not correspond to a column in the target database must have a name prefixed with DTS_. For example, DTS_MySortKey. This typically applies to sort key columns which are not database columns.
- If your job makes use of job-generated failures, then one or more links should have columns called DTS_IsRejected, defined as an Integer, and DTS_Message, defined as a Char or Varchar.

16.6.8 Runtime Topologies for DTS jobs

The Runtime Topology refers to the following aspects:

- The number of possible partitions
- The interaction between the source MQConnector and target DTS stages through one of two ways:
  - Work queues
  - Solely relying on the source queue, instead of using work queues

As stated in 16.6.1, “Aspects of DTS application development” on page 314, these are runtime aspects that do not really affect the way jobs are designed. The design of real-time jobs must focus on the business requirements, making sure the job design follows the design rules outlined in 16.6.4, “Design topology rules for DTS jobs” on page 316.

However, the scalability of the job represented in terms of whether it is possible to run just one or multiple partitions is determined by the following characteristics of the incoming messages:

- Ordering

  Messages might have to be processed in the exact same order they were placed in the source queue, and there is nothing you can do about it from a design standpoint. This might be the case of banking feeds from a mainframe operational database to a reporting system. The ordering of the messages must be enforced.

- Relationships

  There might be an indication as to whether messages are correlated.

Appendix A, “Runtime topologies for distributed transaction jobs” on page 375, presents a detailed discussion on runtime topologies. In this section we discuss a few of them that make use of work queues. There is a possibility of bypassing work queues, in the sense that the DTS removes messages directly from the source queues. There are challenges with that. Use of work queues avoid those potential issues.

For each topology, we discuss the corresponding stage and link properties.

No ordering, no relationships

Appendix A, “Runtime topologies for distributed transaction jobs” on page 375 presents the ideal case, from a scalability standpoint, which is illustrated in Figure 16-18 on page 328. For as long as there are no ordering constraints and no relationships between the incoming messages, the job can scale across multiple partitions. There are multiple MQConnectors and multiple DTS instances.

All MQConnectors read messages off the same source queue. However, they transfer messages to work queues that are dedicated to each partition.

Upon job restart after failure, the job must be restarted with the exact same number of nodes in the config file. Messages are either reprocessed out of the private work queues, or read from the source queue.

All source MQConnectors transfer messages into individual work queues by means of MQ local transactions.

Each target DTS stages applies a separate global transaction, involving the target resource managers and the partition-specific private work queue.

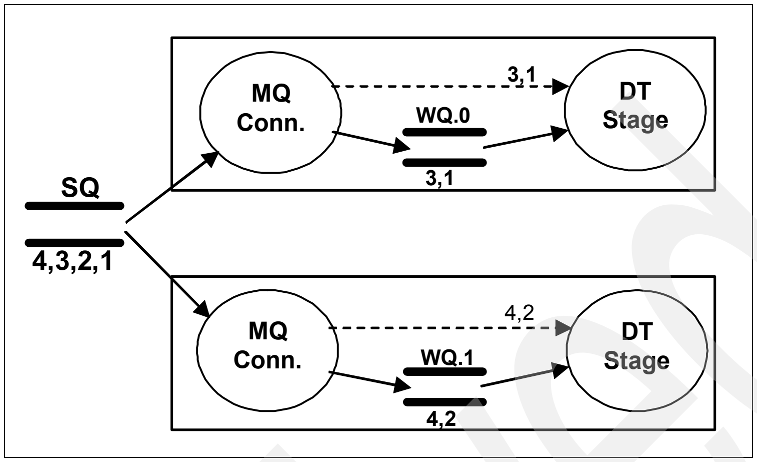


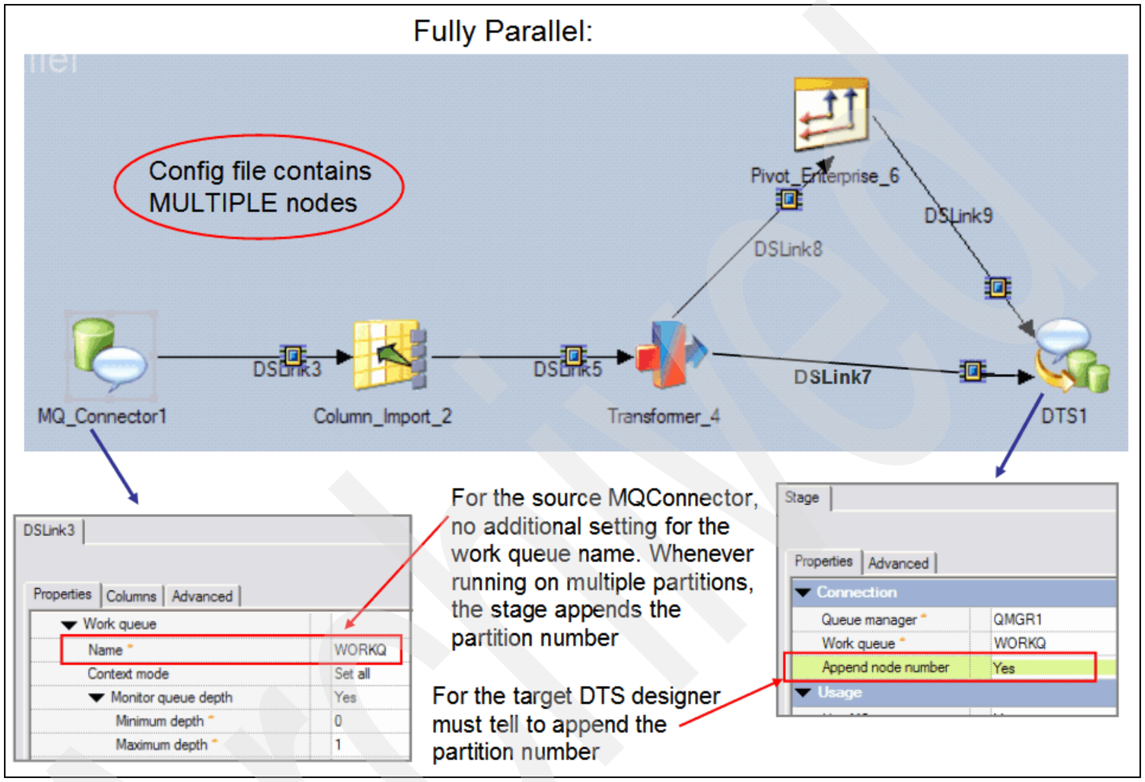
Figure 16-18 A scalable topology, with no ordering and no relationships between messages

Figure 16-18 indicates the properties that must be set in a parallel job to enable the topology of Figure 16-19 on page 329:

- APT_CONFIG_FILE must point to a config file containing mode than one node in the default node pool.
- The MQConnector and DTS stages must point to the same work queue name (WORKQ).
- The target DTS stage must have the "Append node number" option set to **Yes**.

The MQConnector stage, knowing it is running on multiple partitions, appends the partition number to the work queue name. There is no additional setting on this stage to make this happen.

The DTS, on the other hand, does not append the partition number automatically when it runs in parallel. That is because it is also intended to support the topology described in “No strict ordering with inter-message relationships” on page 332. For the DTS to interact with multiple work queues (that is, one per partition) the user must set the property “Append node number” to **Yes**.



DSLink3

Properties | Columns | Advanced

▼ Work queue	
Name *	WORKQ
Context mode	Set all
▼ Monitor queue depth	Yes
Minimum depth *	0
Maximum depth *	1

For the source MQConnector, no additional setting for the work queue name. Whenever running on multiple partitions, the stage appends the partition number

For the target DTS designer must tell to append the partition number

Stage

Properties | Advanced

▼ Connection	
Queue manager *	QMGR1
Work queue *	WORKQ
Append node number	Yes
▼ Usage	

Figure 16-19 MQ/DTS job properties for fully parallel topology

Strict ordering

Unfortunately, the more common scenario is the one that requires the strict ordering of messages, possibly without indication of how to group incoming messages based on any sort of criteria.

This is the typical case when transferring data from operational mainframe-based banking applications to reporting databases. Database transactions must be executed in a strict order. Otherwise, the result in the target reporting database does not make any sense when compared to the original source.

When that is the case, adopt the topology of Figure 16-20. There is a single node in the configuration file and as a consequence, there is a single work queue.

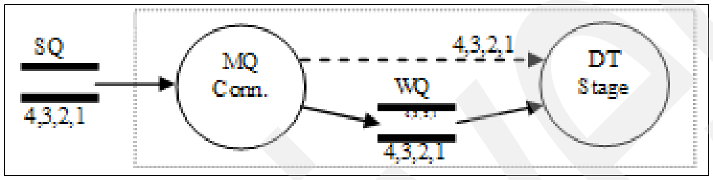


Figure 16-20 Strict ordering for message processing

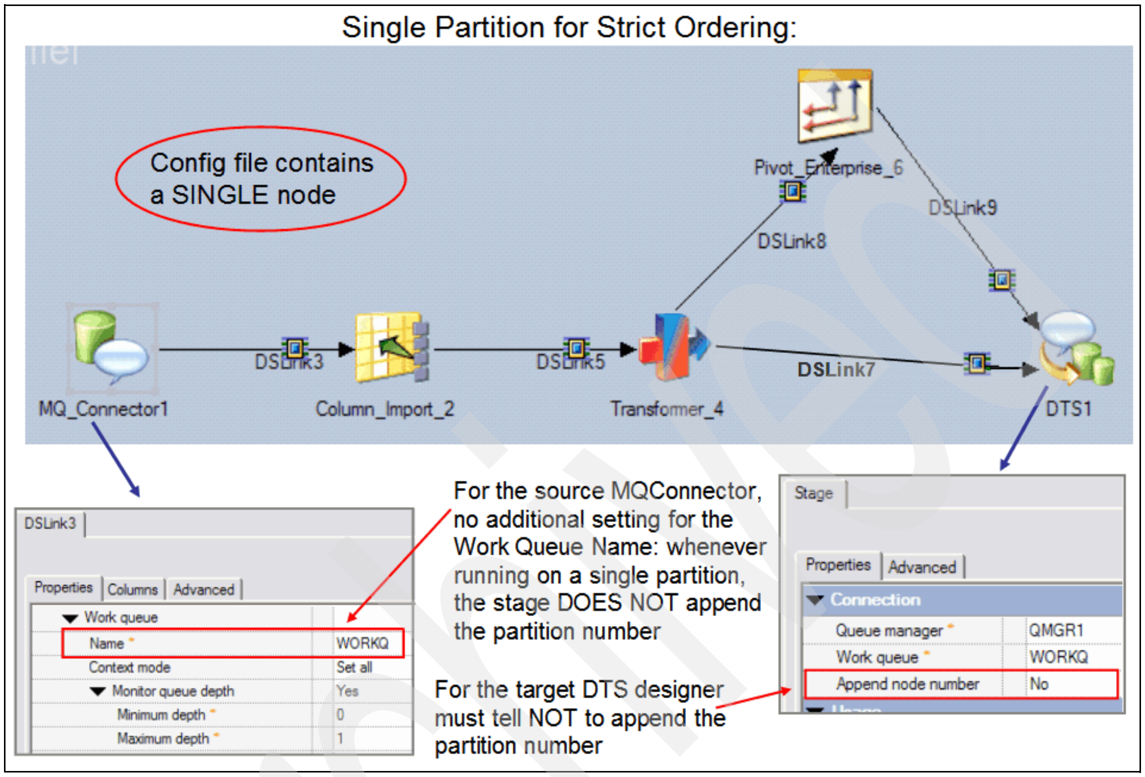
Figure 16-21 on page 331 presents the properties to enable the topology of Figure 16-20:

- APT_CONFIG_FILE must point to a config file containing a single node in the default node pool;
- The MQConnector and DTS stages must point to the same work queue name (WORKQ)
- The target DTS stage must have the option Append node number set to *No*.

Even though the job execution is restricted to a single partition, such applications can still benefit from pipeline parallelism. That is because the processing is broken down into separate processes. That is one of the major advantages of the parallel framework.

The job designers use the same DS constructs to express logic that meets business rules regardless of partitioning. At runtime, the framework still makes seamless use of multiple processes, scaling much better than an equivalent single-threaded implementation, which is typically the case with J2EE based applications.

J2EE application servers are multi-threaded. However, unless a great deal of effort is put into producing a sort of parallelism, individual Java methods that implement SOA services in J2EE applications tend to be single-threaded.



Config file contains a SINGLE node

For the source MQConnector, no additional setting for the Work Queue Name: whenever running on a single partition, the stage DOES NOT append the partition number

For the target DTS designer must tell NOT to append the partition number

Work queue	
Name	WORKQ
Context mode	Set all
Monitor queue depth	Yes
Minimum depth	0
Maximum depth	1

Queue manager	QMGR1
Work queue	WORKQ
Append node number	No

Figure 16-21 MQ/DTS job properties for strict ordering

No strict ordering with inter-message relationships

There is one intermediate topology, for cases when there is no strict global ordering, but there is a relationship between messages. Messages can be grouped by a certain criteria specified in the form of a hash key.

Messages with the same hash key are still processed in the same order in which they are posted on the source queue.

Figure 16-22 depicts a topology of no ordering, with relationships.

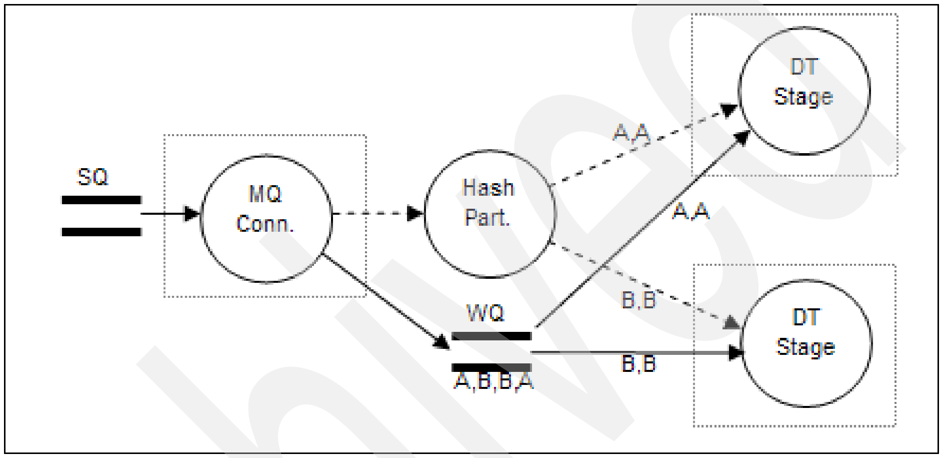
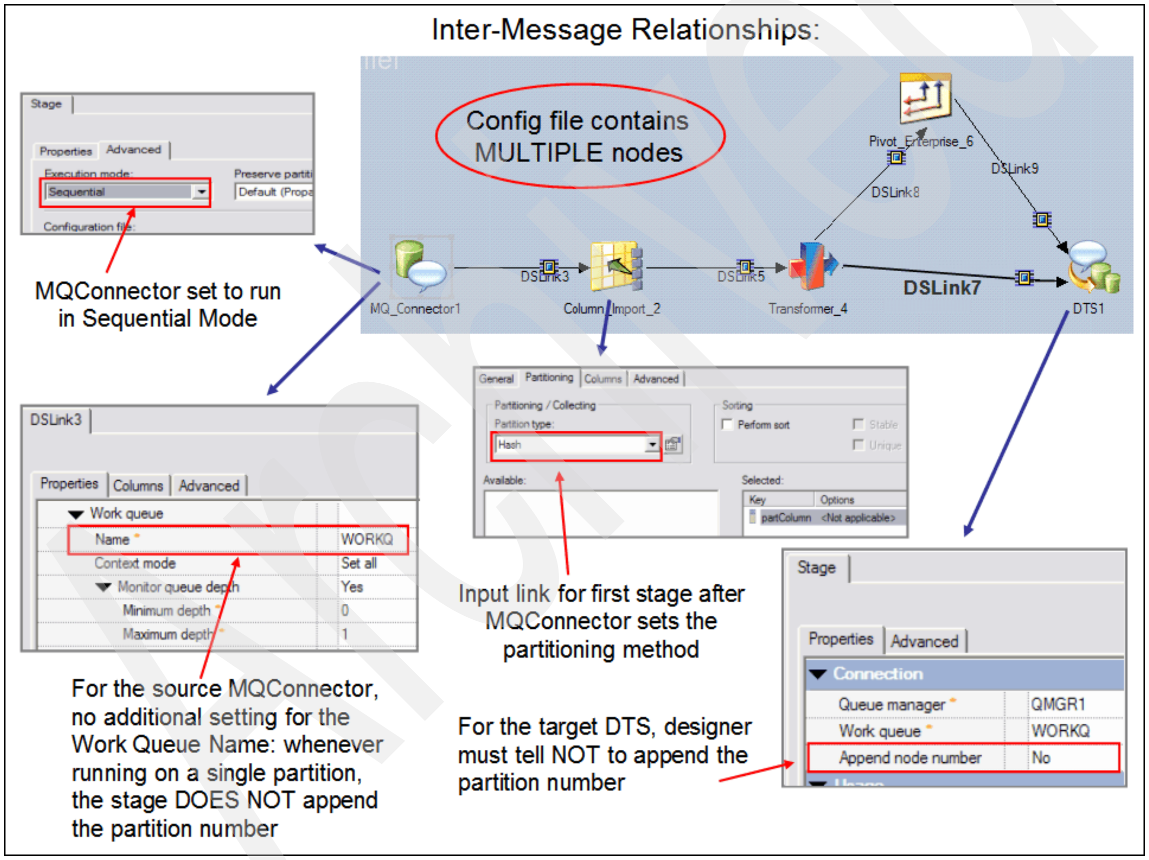


Figure 16-22 No ordering, with relationships

Figure 16-23 shows properties that must be set to enable the topology of Figure 16-22 on page 332:

- APT_CONFIG_FILE must point to a config file containing a multiple nodes in the default node pool;
- The MQConnector and DTS stages must point to the same work queue name (WORKQ);
- The target DTS stage must have the option Append node number set to No;
- The source MQConnector stage must execute in sequential mode;
- The input link for the stage immediately after the MQConnector (the column import) must set hash partitioning.



Execution mode: Sequential

Preserve partitioning: Default (Propa

Configuration file:

MQConnector set to run in Sequential Mode

DSLink3	
Work queue	
Name *	WORKQ
Context mode	Set all
Monitor queue depth	Yes
Minimum depth *	0
Maximum depth *	1

For the source MQConnector, no additional setting for the Work Queue Name: whenever running on a single partition, the stage DOES NOT append the partition number

Partitioning / Collecting

Partition type: Hash

partColumn <Not applicable>

Input link for first stage after MQConnector sets the partitioning method

Connection	
Queue manager *	QMGR1
Work queue *	WORKQ
Append node number	No

For the target DTS, designer must tell NOT to append the partition number

Figure 16-23 MQ/DTS job properties for inter-message relationships

Bypassing work queues

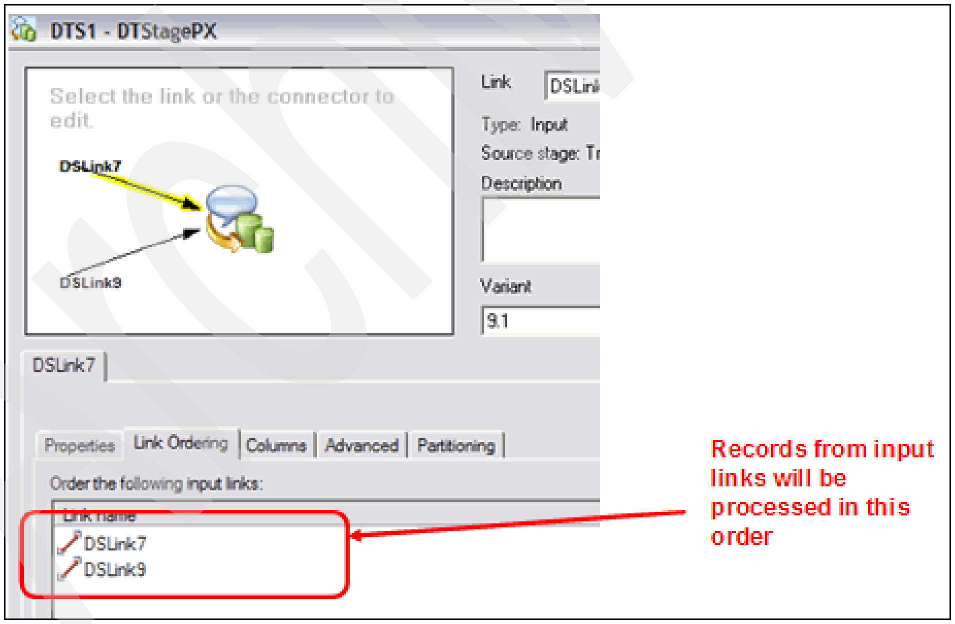
There are other topologies discussed in Appendix A, “Runtime topologies for distributed transaction jobs” on page 375 that bypass work queues. There are potential issues, so we suggest using work queues.

16.6.9 Processing order of input links

The default behavior of the DTS stage is to process the rows in the order the input links are defined in the DTS stage. Figure 16-24 presents the order of input links to the DTS stage, which can be accessed from the Link Ordering tab.

The SQL statements for all rows sent to the first input link are processed, and the SQL statements for all rows to the second input link are processed, and so on. The order of processing of the records from input links are highlighted in the box in Figure 16-24, which is pointed to with an arrow.

All SQL statements are processed as part of the same transaction. Input links for parent tables must be listed first, followed by the links for children, or dependent tables, so the referential integrity in the target database is maintained.


DTS1 - DTStagePX

Select the link or the connector to edit.

Link: DSLink

Type: Input

Source stage: T

Description

Variant: 9.1

Order the following input links:

Link name
DSLink7
DSLink9

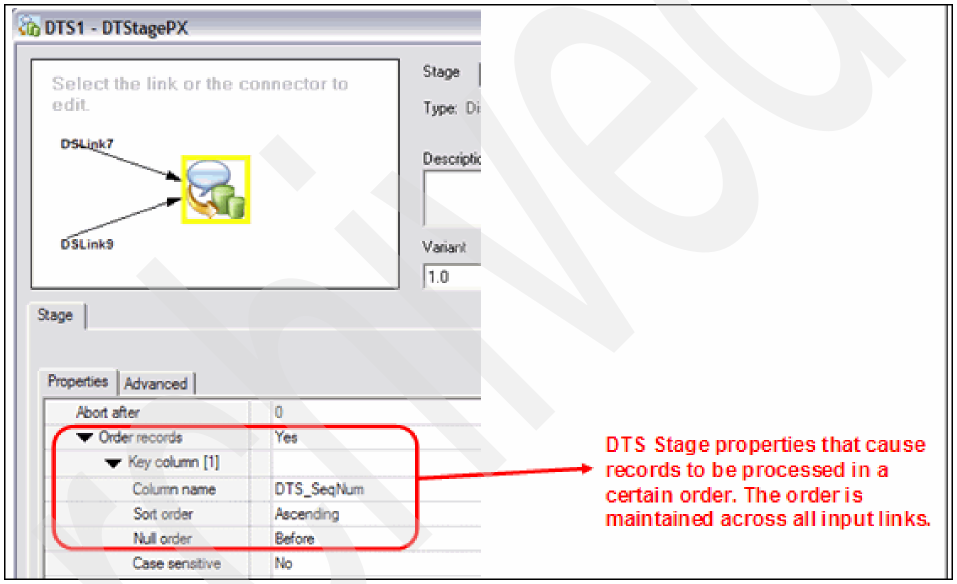
Records from input links will be processed in this order

Figure 16-24 Default input processing order

There might be cases when this processing order is not adequate. The designer might choose to adopt a different order: instead of processing all records on a link-at-a-time basis, the DTS stage processes records ordered across all input links.

Figure 16-25 shows an example in which there is a field named DTS_SeqNum. This field must be present on all input links. Remember, because this field is not present in the target tables, it must contain the prefix DTS_ according to the rules described in “Input link properties for the DTS stage” on page 325.

The DTS stage properties that cause records to be processed in a certain order are highlighted in the box shown in Figure 16-25. The order is maintained across all input links. The DTS stage then processes records from different input links, according to the order of the DTS_SeqNum values.


DTS1 - DTStagePX
Select the link or the connector to edit.
DSLink7
DSLink9
Stage
Type: Di
Descriptio
Stage
Properties
Advanced
DTS Stage properties that cause records to be processed in a certain order. The order is maintained across all input links.

Variant: 1.0

Abort after	0
Order records	Yes
Key column [1]	
Column name	DTS_SeqNum
Sort order	Ascending
Null order	Before
Case sensitive	No

Figure 16-25 Cross-link ordering

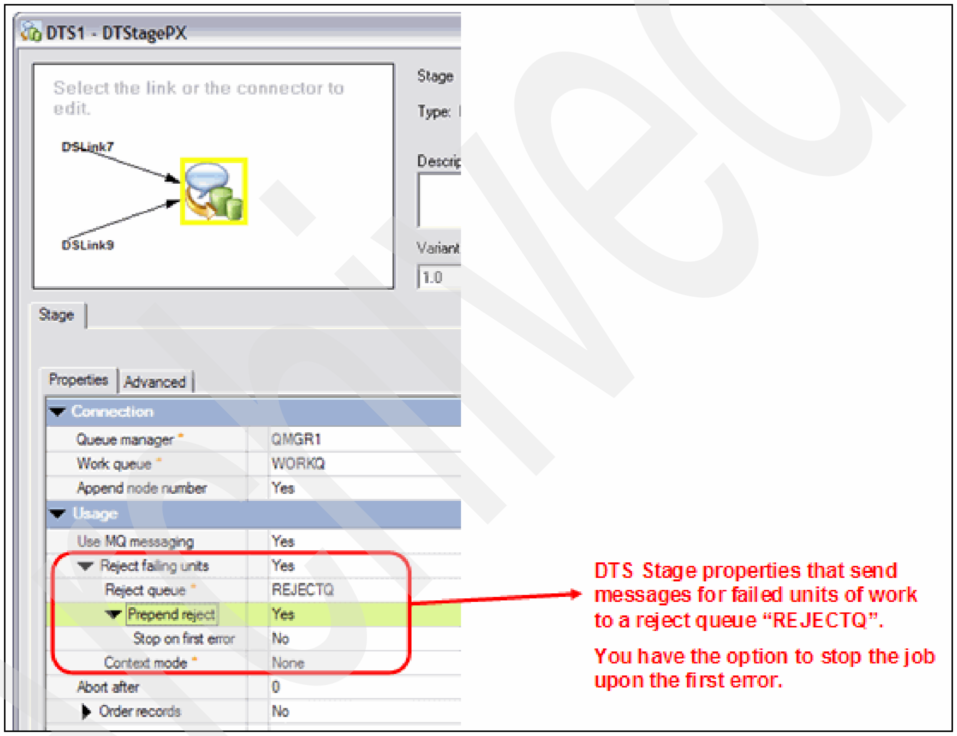
16.6.10 Rejecting messages

There are two types of failure that can occur during the processing of work items:

- A work item is rejected if a single record in any of the rows comprising that unit of work are flagged with the DTS_IsRejected column set to **True**. In this event, there might be a job-provided error message in the DTS_Message column. Any stage in the job (typically a transformation stage), might set the DTS_IsRejected column to **True** if it detects any condition that identifies the work item as warranting rejection.

- ► A failure occurs in outputting a record to a target. An example of this is an error being returned by the database due to a constraint violation.

Figure 16-26 shows how you can enable the use of a work queue in the DTS stage. The DTS stage properties that send messages for failed units of work to a reject queue REJECTQ are highlighted by the box in Figure 16-26. You have the option to stop the job upon the first error.


DTS1 - DTStagePX
Select the link or the connector to edit.
DSLink7
DSLink9
Stage
Type:
Descrip
Variant
1.0
Stage
Properties
Advanced
DTS Stage properties that send messages for failed units of work to a reject queue "REJECTQ".
You have the option to stop the job upon the first error.

Connection	
Queue manager *	QMGR1
Work queue *	WORKQ
Append node number	Yes
Usage	
Use MQ messaging	Yes
Reject failing units	Yes
Reject queue *	REJECTQ
Prepend reject	Yes
Stop on first error	No
Context mode *	None
Abort after	0
Order records	No

Figure 16-26 Enabling a reject queue in the DTS stage

In the event of a failure or rejection of a unit of work, the unit is rolled back so that no data is written to the targets. Additionally:

- ► The source messages might be as follows:
  - – Moved to a reject queue (the default behavior)
  - – Left on the source queue
- ► The job might be aborted after a specified number of units of work have been rejected.

- An option is provided in the DT stage to preface the rejected message data with information about the failure. This information is provided in a 256 byte record defined in Table 16-1.

Table 16-1 Failure information

Offset	Length	Type	Description
		int32	The link number containing the failure. If there are multiple, only the link number for the first occurrence is listed.
		int32	Status. Has one of the following values: 0 – Success: this particular message succeeded, but other messages in this transaction failed. 1 – Unknown: the records in this message were not executed because of an earlier failure (see StopOnFirst option) 2 – Target error: an error occurred writing to a target 3 – Commit error: the transaction cannot be committed 4 – Rejected: the transaction was rejected by logic in the job (the DTS_IsRejected flag was set to true)
		int32	Count of the number of rows in error or rejected in this message.
12	12	string	The error code from the resource (if the status of the unit of work was 2)
24	236	string	The error code from the resource (if the Status is 2), or the text message from the DTS_Message column (if the Status is 4). In the latter case, this contains the text from the first DTS_Message column encountered for this message that is not null or an empty string.
270	242	string	The text message from the DTS_Message column. This contains the text from the first DTS_Message column encountered for this message that is not null or an empty string.

If there are multiple failures, then the status reflects the status of the first detected error. The count field provides a count for the number of rows in the message that failed.

A property of the DT stage determines whether the processing of a unit-of-work stops upon the first detection of an error. This results in the status of subsequent messages in the unit-of-work being set to **Unknown**. If it is preferable to determine whether these subsequent messages are unknown or not, the stage can be configured to process all rows in the unit-of-work to obtain accurate message status. Doing this has a negative impact on performance, because all rows must be processed and subsequently rolled back.

Job-induced reject

The job RejectedByJobLogic demonstrates one way to reject a transaction by detection of an error condition, such as an error in the data. The job looks like Figure 16-27.

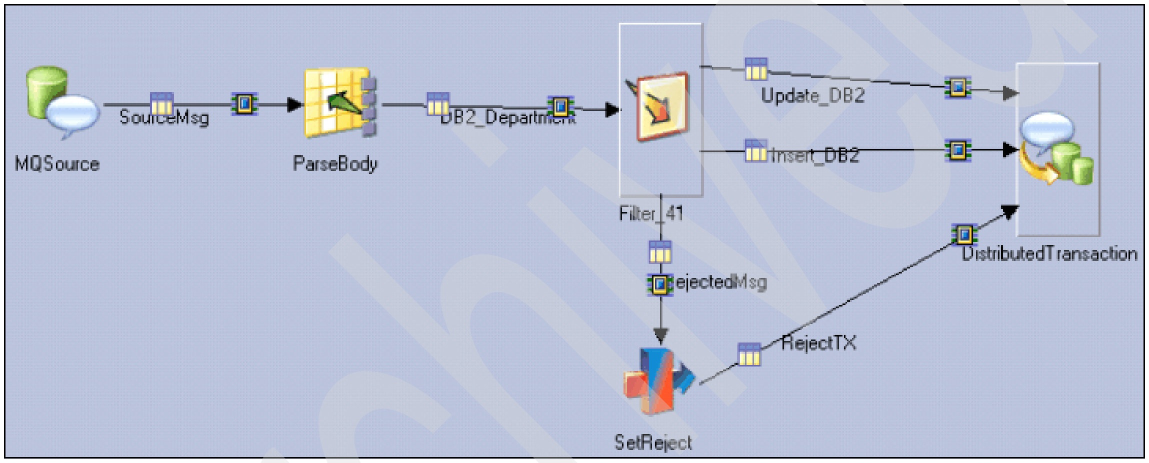
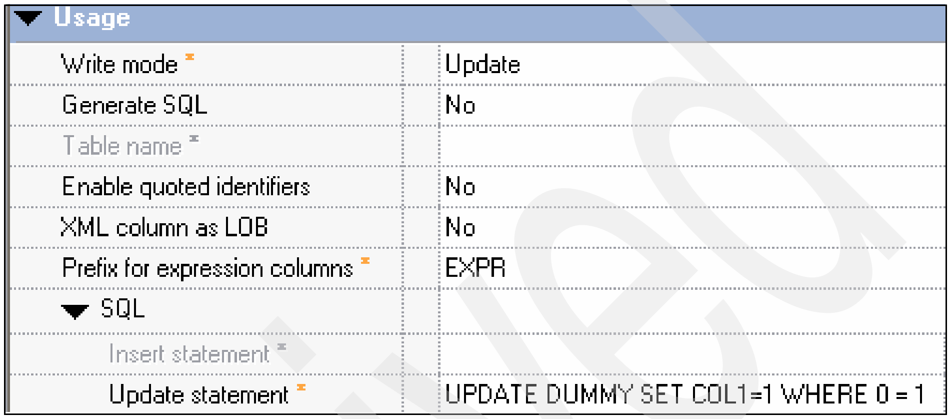


Figure 16-27 Reject by detection

There are three links to the DTS. Two of the links are to insert or update DB2 tables, and the third is purely for sending reject signals to the stage.

Though the DTS_isRejected column can be specified on any input link, it is often convenient to have a dedicated reject link containing this column. This is the approach used by this example. It is necessary for the link to execute a benign statement that has no impact on the target. In this example, this is accomplished by using a dummy table and a statement that has no effect by specifying the where clause as WHERE 0=1. That table is depicted in Figure 16-28.



Usage	
Write mode *	Update
Generate SQL	No
Table name *	
Enable quoted identifiers	No
XML column as LOB	No
Prefix for expression columns *	EXPR
SQL	
Insert statement *	
Update statement *	UPDATE DUMMY SET COL1=1 WHERE 0 = 1

Figure 16-28 Reject by detection dummy table

16.6.11 Database contention

Batch jobs tend to eliminate database contention by using various techniques to minimize the interaction with target and source databases. Batch applications must use bulk load and unload techniques, decoupling the transformation phases from extraction and provisioning jobs.

Real-time jobs cannot afford the luxury of minimizing the interface with target databases. Lookups, for instance, tend to be Sparse (the exception being lookups against static tables). Transactions must be successfully committed for every wave.

DTS jobs are tightly coupled with databases. There are multiple queries and update/insert/delete statements being executed concurrently, for separate waves, and for records that are part of the same wave.

As a result of the multitude of SQL statements being executed for multiple records in multiple waves across multiple partitions, there can be multiple database exceptions, such as lock timeouts and deadlocks.

Though the parallel framework enables scalability and parallelism, it introduces challenges that must be addressed for real-time database interoperability:

- Key collisions
- Deadlock and lock timeouts

These two challenges pertain to both ISD and MQ/DTS jobs. Key collisions are discussed in 16.10, “Pipeline Parallelism challenges” on page 366.

Although lock-related exceptions can occur in both ISD and MQ/DTS jobs, they are more pronounced in the latter.

Figure 16-29 presents an example of a DTS job that upserts records into three separate tables. One might think that one upsert does not affect the others and the job shall work perfectly well.

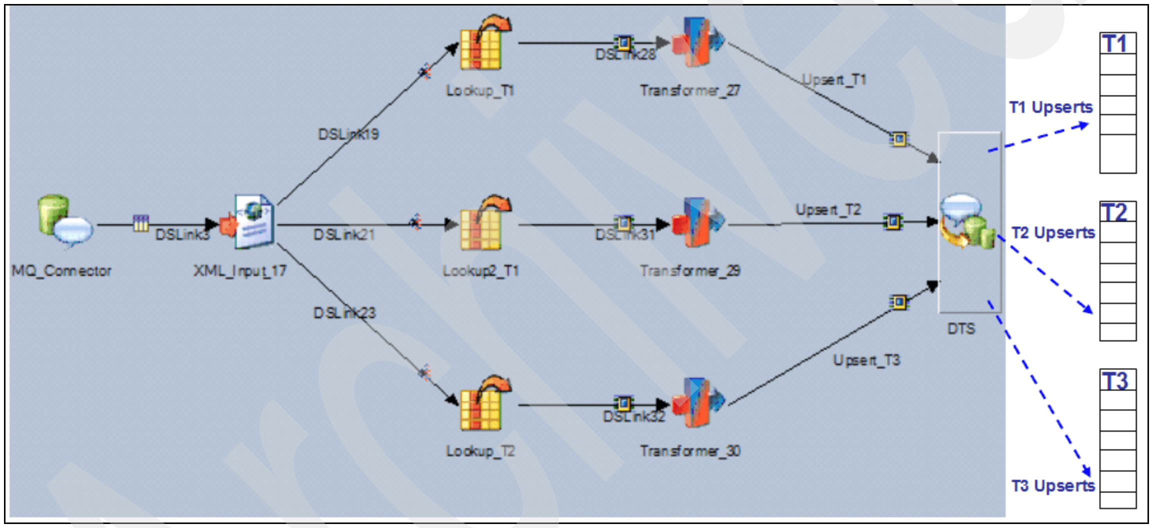


Figure 16-29 A simplistic view of database interactions

This might be true in a development and test environment, with only a few messages submitted to the source queue at a time. Things go pretty well, up until several more messages are submitted to that same source queue.

Upon submitting several messages, the developer might see deadlock and lock timeout exceptions in the DS Director log, thrown by both the DTS and Sparse Lookups. A more detailed view of what might happen at the database level can be seen in Figure 16-30. Inserting records into dependent/children tables involves not only placing locks on the table itself, but also checking foreign keys against the parent table primary key index.

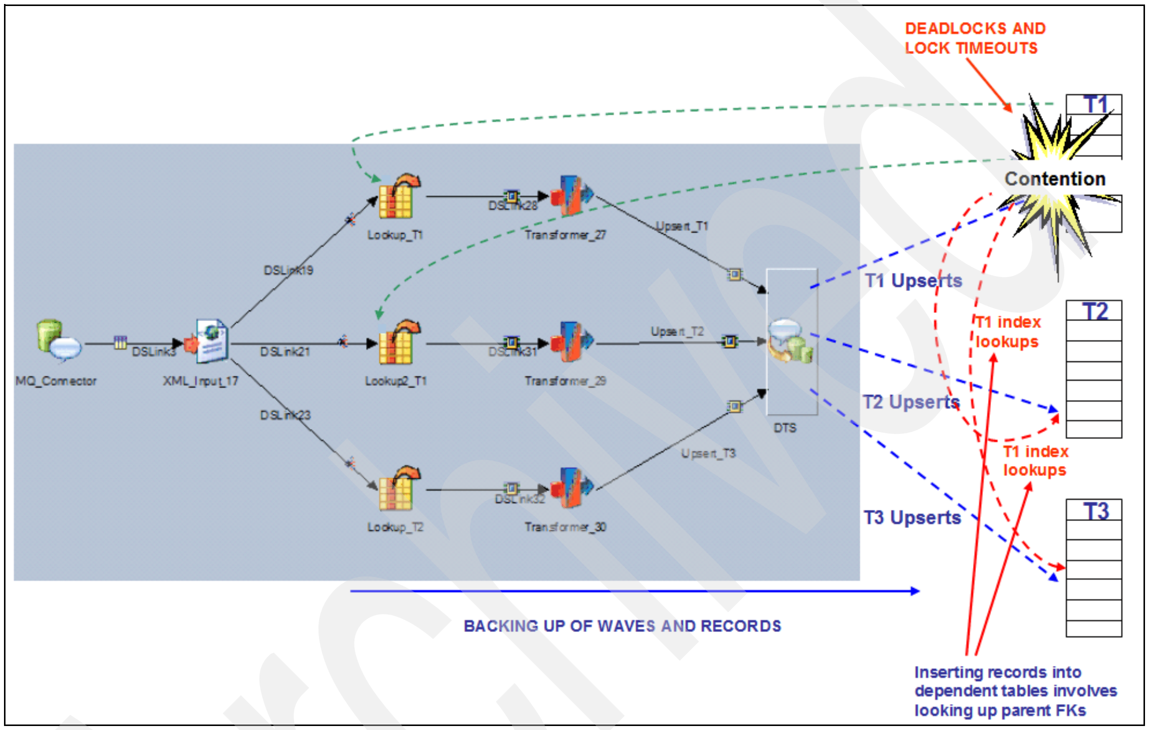


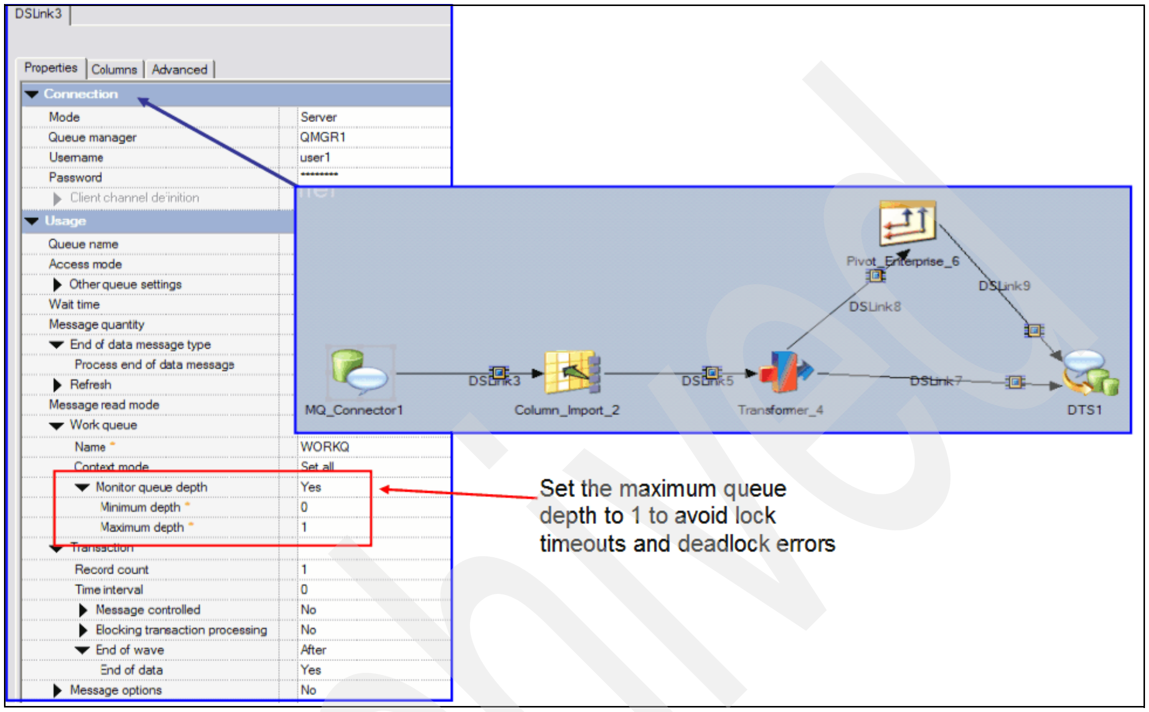
Figure 16-30 Database contention

This type of problem tends to be more pronounced when testing real-time jobs against empty or small tables. In this case, most of the activity is going against the same table and index pages, increasing the chance of contention. As there are more records being fed into the pipeline by the source MQConnector, these exceptions increase in number.

Having lock-related exceptions during real-time jobs is something we must to avoid at all costs.

These exceptions are further aggravated when running the job with multiple partitions. They even occur with a Strict Ordering topology (See “Strict ordering” on page 330).

The solution is to limit the depth of the work queue. Set the Monitor Queue Depth property to **Yes** and the Maximum Depth property to **1**, as in Figure 16-31. In addition, start with a single node in the config file.



Connection	
Mode	Server
Queue manager	QMGR1
Username	user1
Password	********
Client channel definition	
Usage	
Queue name	
Access mode	
Other queue settings	
Wait time	
Message quantity	
End of data message type	
Process end of data message	
Refresh	
Message read mode	
Work queue	
Name *	WORKQ
Context mode	Set all
Monitor queue depth	Yes
Minimum depth *	0
Maximum depth *	1
Transaction	
Record count	1
Time interval	0
Message controlled	No
Blocking transaction processing	No
End of wave	After
End of data	Yes
Message options	No

Set the maximum queue depth to 1 to avoid lock timeouts and deadlock errors

Figure 16-31 Limiting the work queue depth

Setting the work queue depth to 1 and running the job on a single partition might be done as a starting point. The queue depth and the number of partitions can then be increased, as long as the lock-related exceptions do not occur.

There are number of aspects related to the database that require close attention by the DBAs:

- Make sure the SQL plans are fully optimized and take advantage of indices.
- Avoid sequential scans in SQL plans.
- Make sure the locking is set at the record-level, not at page-level.
- Enable periodic execution of update statistics.
- DBAs must closely monitor database statistics and query plans.

In summary, real-time jobs tend to be closely coupled with databases. Therefore, there must be a close cooperation with DBAs, to make sure the database is fully optimized.

16.6.12 Scalability

The scalability of real-time MQ/DTS jobs depends on a number of factors, some of them previously discussed:

- Whether jobs can run in parallel or just sequentially
  - Ordering
  - Relationships between messages
- Length of wave

  How many messages can be grouped into a single wave/unit-of-work.
- Size of arrays

  An input link property for the DTS stage. Each link might have a different setting.

The one that has most impact is the length of the waves. Having more than one message in a single UOW is most beneficial because each 2-Phase Commit has a significant overhead. A significant improvement can be seen as the length of the wave is increased by adding more messages.

At some point, adding more messages will no longer provide a noticeable performance increase. However, that is not possible in most circumstances. UOWs must remain limited to a single incoming message in the majority of cases. The only resort is to adjust the size of the arrays.

Also, one might use multiple partitions whenever possible, depending on ordering constraints (See the discussion on runtime topologies in 16.6.8, "Runtime Topologies for DTS jobs" on page 326 and Appendix A, "Runtime topologies for distributed transaction jobs" on page 375).

16.6.13 Design patterns to avoid

In this section we discuss design patterns that, although valid, are to be avoided.

So-called mini-batches are to be avoided as well. They are not even considered a valid real-time pattern, which is one of the reasons why they are not listed in this section. The topic on mini-batches deserves a special topic on its own, and is presented in 16.2, "Mini-batch approach" on page 297.

Using intermediate queues

Starting with IS 8.5, it is possible to implement MQ/DTS jobs with MQConnector as a target. This would make it possible to write jobs that implement guaranteed delivery from source to target MQ queues.

Adopt this type of technique whenever the source is in the form of MQ messages, and the target is supposed to be consumed by an existing or third-party application.

One example is the case of a retail bank that implemented a DS job to transfer data from a source mainframe to a target system, using MQ Series as the standard middleware interface. This is depicted in Figure 16-32. DataStage jobs pick incoming messages, apply transformations, and deliver results to the target queue.

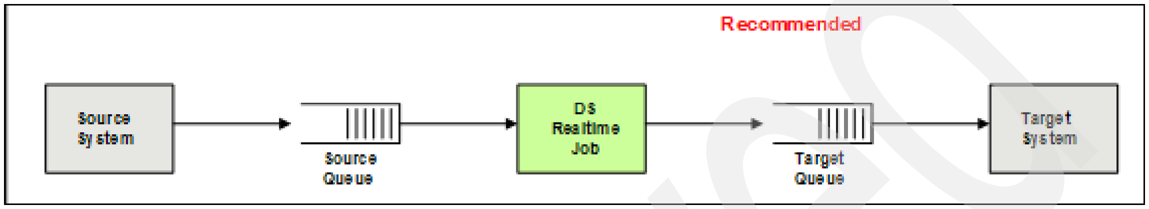


Figure 16-32 MQ as Source and Target

We consider this an acceptable pattern. However, as real-time jobs tend to grow quite large, developers might start thinking about using intermediate queues to break up larger logic into smaller steps, using intermediate queues as a way of safely landing data, guaranteeing against any loss of messages.

This is the scenario depicted in Figure 16-33 on page 345. The real-time DS logic is broken into multiple smaller steps.

We might trace a parallel to this approach with the DS Server-based mentality of breaking batch processing into multiple steps and frequently landing data to disk. In the case of batch processes, breaking larger jobs into smaller ones leads to higher disk usage, higher overhead, and longer execution times.

In the case of real-time processes, the net effect, in addition to higher overhead, is a longer lag time until the DataStage results are delivered to the target queue for consumption by the target system.

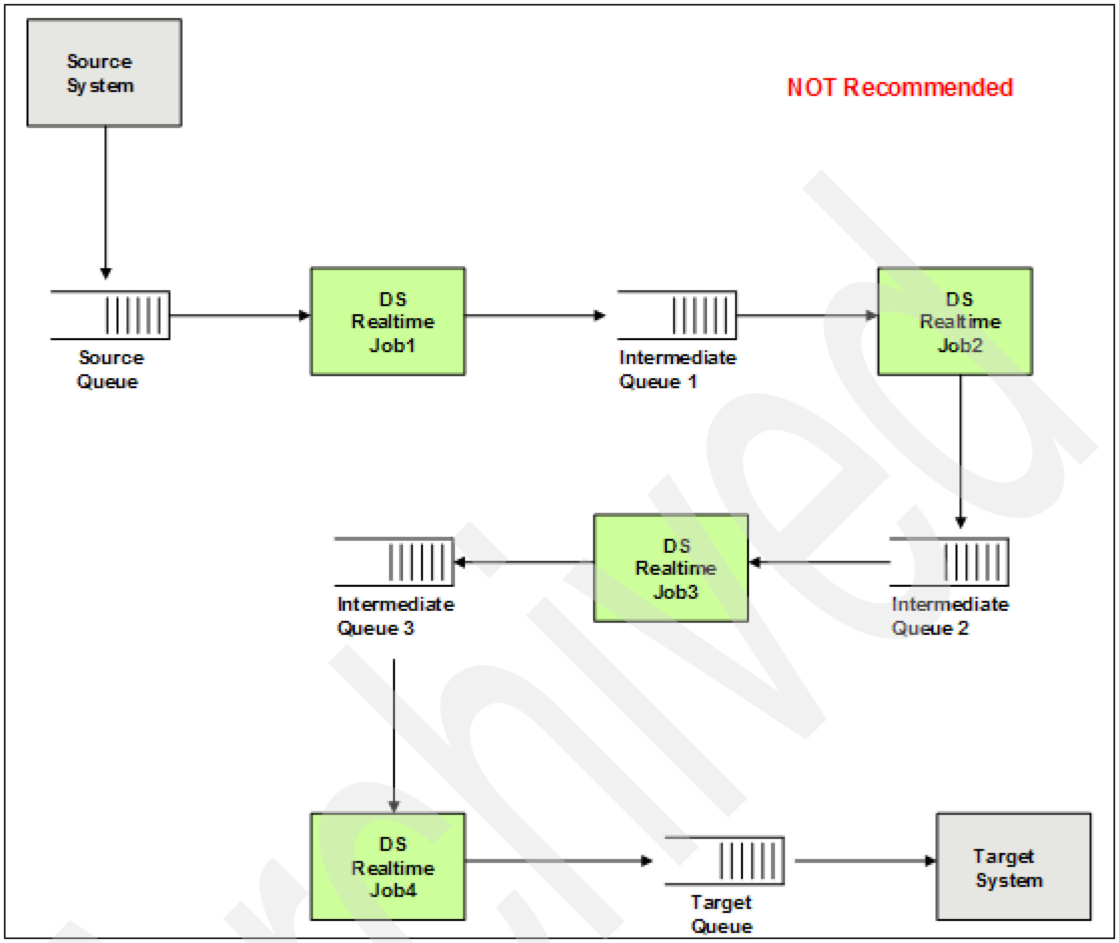


Figure 16-33 Using intermediate queues for real-time processing (not recommended)

Real-time jobs have a different profile of use of OS resources than batch jobs. Batch jobs move and transform large amounts of data at once, incurring much higher memory and disk usage.

Although real-time jobs normally have a larger number of stages (it is common to see hundreds of them in a single MQ/DTS or ISD flow), they tend to deal with smaller input data volumes. The overall memory usage for the data and heap segment of parallel operators, as well as disk use, tends to be smaller.

The goal of real-time jobs from a performance standpoint must be delivering results to the targets (result queues, target database tables, ISD outputs) as quickly as possible with minimal lag time.

The approach depicted in Figure 16-33 works against that goal.

16.7 InfoSphere Information Services Director

The second way of creating live, real-time jobs is with the InfoSphere Information Services Director (ISD). 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 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 can retain a single dataflow design. DS jobs exposed as ISD services are referred to as “ISD Jobs” throughout this section.

ISD is described in detail in the document *IBM InfoSphere Information Services Director*, which is included in the standard Information Server documentation installed on client workstations as part of the IS Client install. This document focuses on design patterns and practices for development of live ISD Jobs.

Figure 16-34 on page 347 is reproduced from the ISD manual and depicts its major components. The top half of the diagram shows 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 16-34 on page 347 presents components that belong to the Engine Layer, which can reside either on the same host as the Domain, or on separate hosts.

Each 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 by means of the Information Server Console.

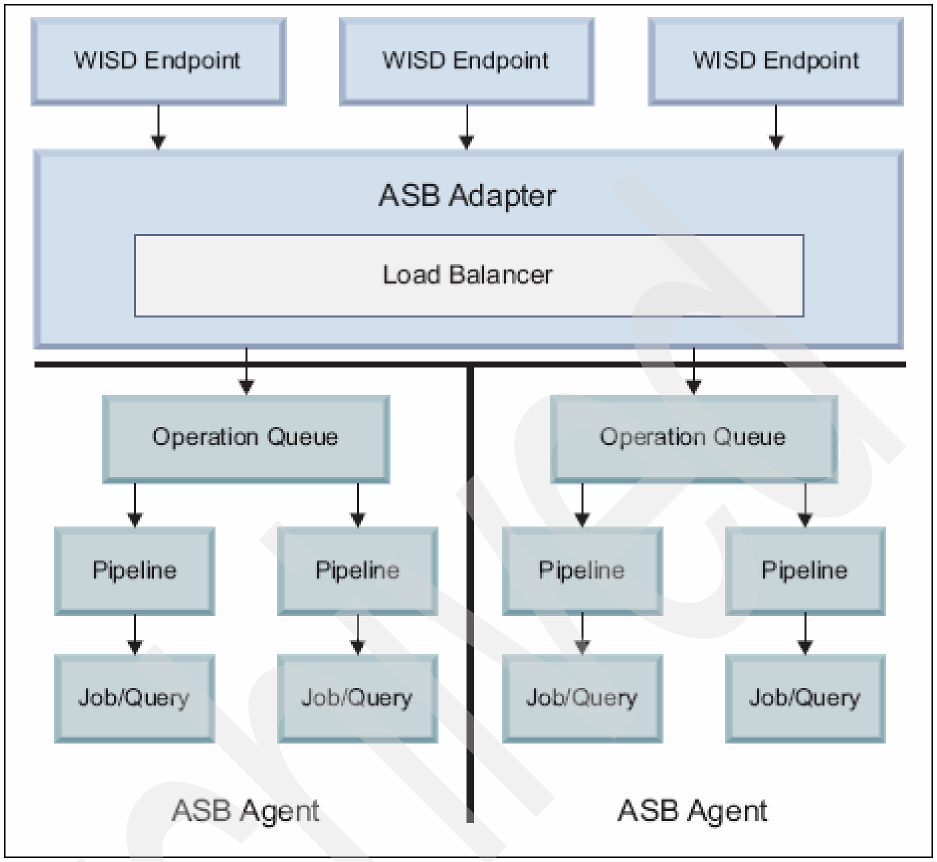


Figure 16-34 Major components of Information Services Director

The endpoints forward incoming requests to the ASB adapter, which provides for load balancing and interfacing with multiple services providers. Load balancing is another important concept in SOA applications. In this context it means the spreading of 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. This is a standalone Java application, meaning it does not run inside a J2EE container. Because it is a Java application, it is multi-threaded and supports the servicing of multiple incoming requests.

An ASB agent implements queuing and load balancing of incoming requests. Inside a given DS engine, there might be multiple pre-started, always-on job instances of the same type, ready to service requests. That is where the always-on nature comes into play. Instead of incurring in the cost of reactivating jobs whenever a new request arrives, the ASB agent controls the life cycle 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. This means multiple requests are forwarded to an ISD job instance, before responses are produced by the job and sent back to the ASB agent. There is a correlation between this concept and the pipeline parallelism of DS parallel jobs. There can be multiple concurrent processes executing steps of a parallel job in tandem. The pipelining of requests allows for multiple requests to flow through a job at the same time.

For ISD applications, there is a direct mapping between a service request and a wave. For each and every service request, an end-of-wave marker is generated (See 16.4.2, “End-of-wave” on page 300).

In this section we include an explanation on how the components in Figure 16-34 on page 347 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 16-35 on page 349 illustrates these relationships.

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

Once the ISD operations, services, and application are ready, the Information Server Console can be used to deploy that application, which results in 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, namely the DS engines that participate in this deployment.

This 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.

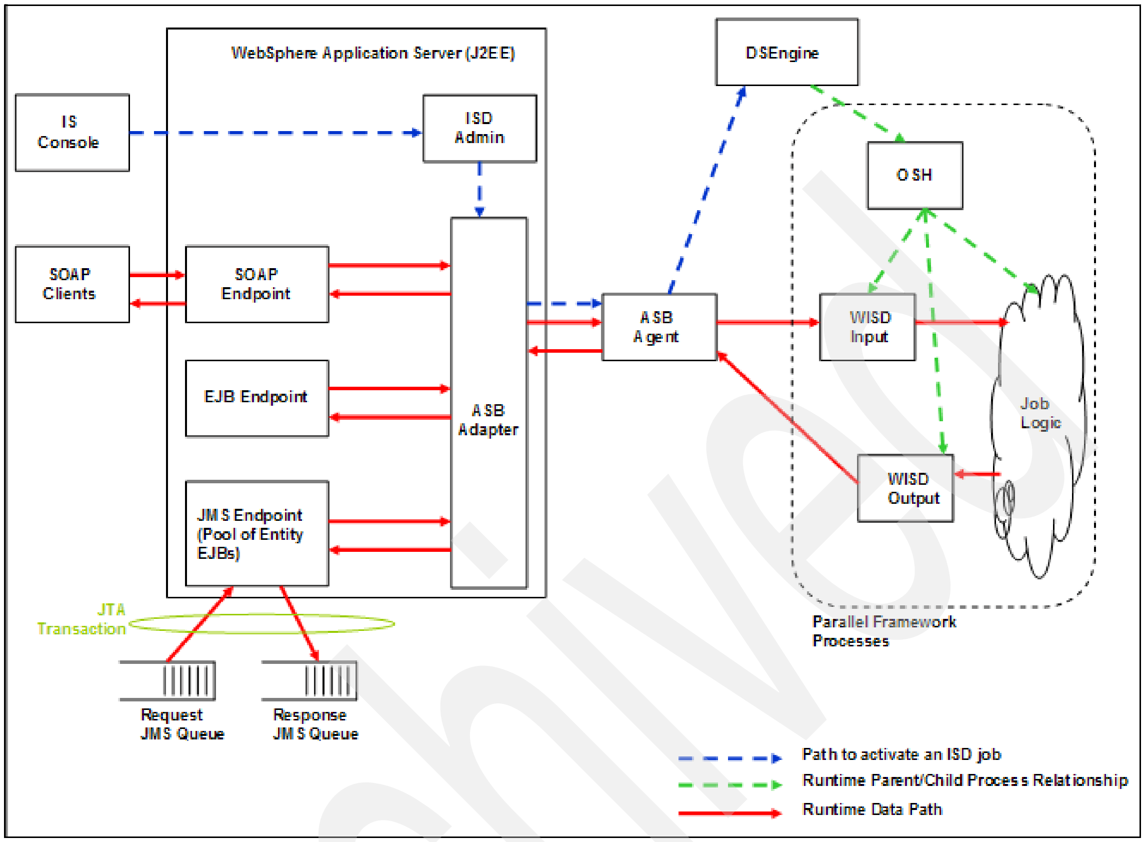


Figure 16-35 Data and control paths in ISD applications

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

Incoming requests follow the path 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. There is one endpoint 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 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) and 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 they came originally from. As opposed to the MQ/DTS solution, the WebSphere Application Server is actively participating in the processing of requests.

16.7.1 The scope of this section

In this section we focus on design patterns and practices for development of always-on ISD jobs.

The following aspects are outside of the scope of this document:

- A tutorial on SOA, Web Services, and J2EE development
- Federation, Classic Federation, and Databases (DB2, Oracle) as service providers
- The steps related to the definition of applications, services, and operations using the Information Server Console
- Batch jobs exposed as ISD services (that is, ISD jobs that do not have a WISD Input stage).

For these aspects, the reader should see the Information Services Director product manual.

For tutorials on SOA, Web Services and J2EE there are countless resources available in books and on the web.

The reason we exclude ISD job topologies that are not of always-on type is because those types of jobs should follow the recommendations outlined in Chapter 15, “Batch data flow design” on page 259.

16.7.2 Design topology rules for always-on ISD jobs

These 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 from 16.5, “Job topologies” on page 307 must be obeyed:
  - All data paths must originate from the same source stage (the WISD Input). The exception is reference links to Normal Lookups. These must originate from independent branches.
  - All data paths must lead to the same target WISD Output stage.
  - There might be dangling branches leading to Sequential File stages, for instance, but these 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.

16.7.3 Scalability

There are two layers of scalability supported by ISD:

- Multiple job instances.
- Deploy across multiple service providers, that is, multiple DS Engines.

These are illustrated in Figure 16-36.

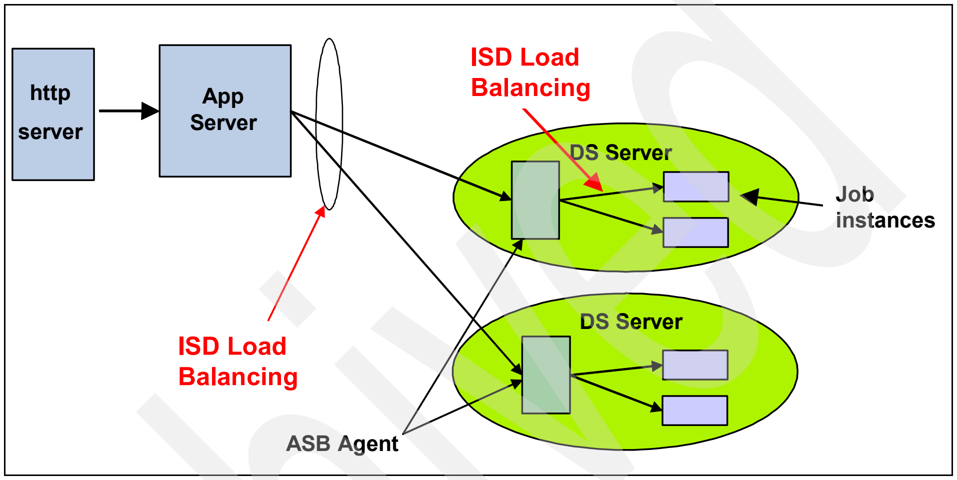


Figure 16-36 ISD load balancing

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 make sure the logic is efficient, which includes, among other things, making sure database transactions, transformations and the entire job flow are optimally designed.

Once the job design is tuned, assess the maximum number of requests that a single job instance can handle. This 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 SLA requirements.

However, there might be cases when one reaches the maximum number of requests a single ASB agent can handle. This means the limit of a single DS engine has been reached. This can be verified when no matter how many job instances are instantiated the engine cannot handle more simultaneous requests. If this is the case, add more DS Engines either on the same host (if there is enough spare capacity) or on separate hosts.

Keep in mind that throughout this tuning exercise, the assumption is that there are enough hardware resources. Increasing the number of job instances and DS Engines does not help if the CPUs, disks, and network are already saturated.

16.7.4 Synchronizing database stages with ISD output

Prior to Information Server 8.5, ISD jobs had to rely on multiple database stages when updating more than one table in a single job flow, similar to what was discussed in 16.4.3, “Transaction support” on page 302.

Figure 16-3 on page 303 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, illustrated in Figure 16-37, that involves using a sequence of stages connected to the reject link of a standard database stage.

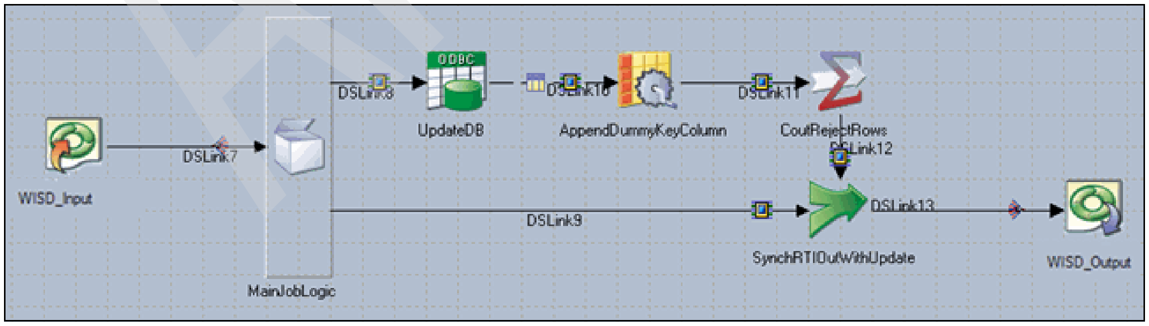


Figure 16-37 Synchronizing database and WISD output

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 along the lines of what is described in 16.7.6, “ISD with connectors” on page 357. However, prior to 8.5, the Teradata did not allow more than a single input link.

There is a Column Generator that created a new 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 is sent only after the database statements for the wave complete (either successfully or with errors).

The logic implemented by means of the ColumnGenerator, Aggregator, and Join stages were repeated for each and every standard database stage present in the flow. Synchronized results made sure there was no other database activity going on for a request that has already been answered.

Again, that is what had to be done 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.

There are two alternatives for always-on 8.5 jobs when it comes to database operations:

- DTS
- Database connector stages

These are discussed in the subsequent sections.

16.7.5 ISD with DTS

The DTS stage has been enhanced as part of the 8.5 release to support transactions without the presence of source and work queues. This enables a DTS stage to respond and commit transactions solely as a result of EOW markers. Using the mode 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:

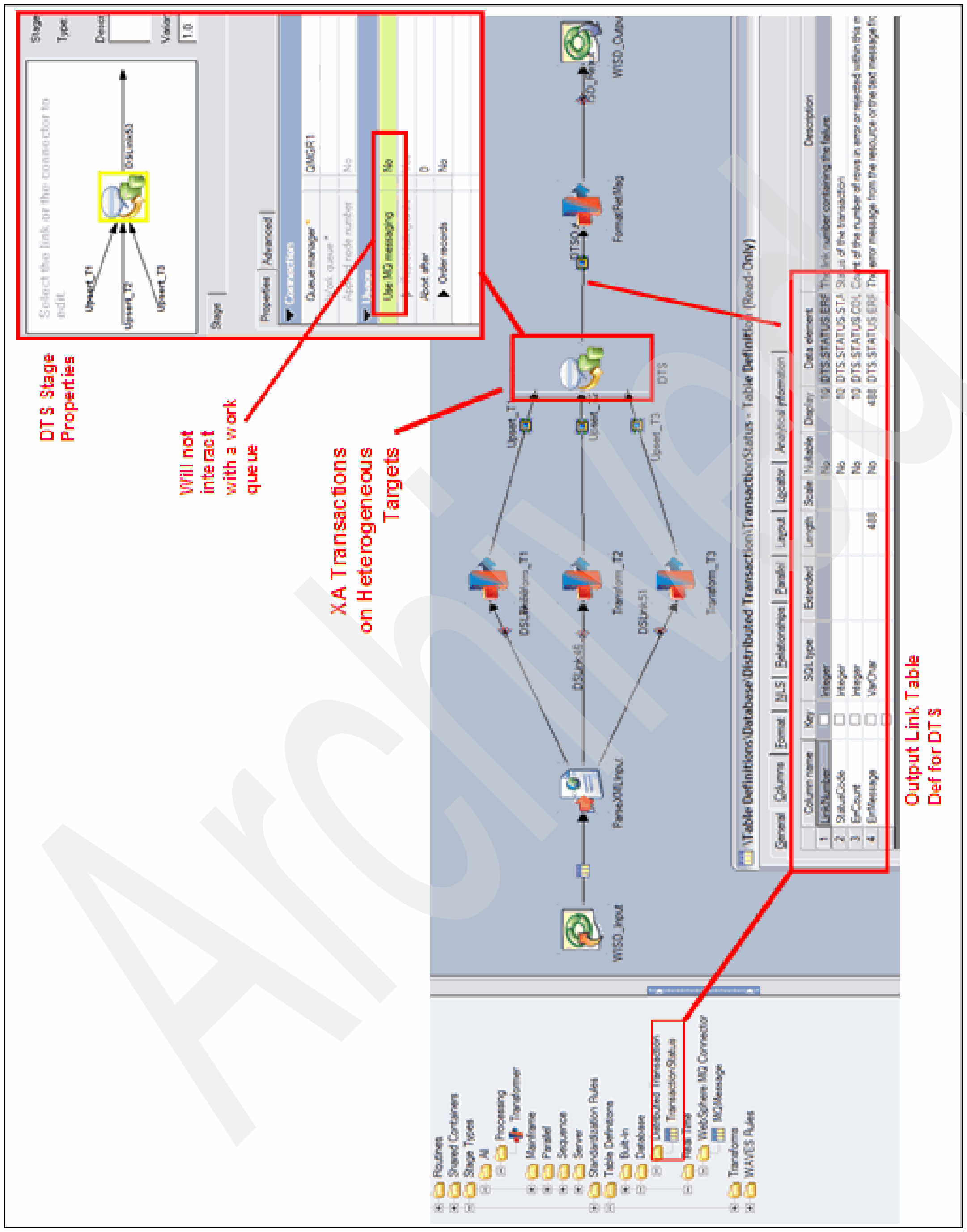
- Multiple SQL statements
- Multiple target types
  - DB2
  - Oracle
  - Teradata
  - Oracle
  - MQ Series

Figure 16-38 on page 356 illustrates a DTS stage used in an ISD job. It significantly reduces the clutter associated with the synchronization of the standard database stages as discussed in the previous section. This illustration is provided to give 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**. Note that although source and work queues are not present, MQ Series 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. This is discussed in the following section.



	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 16-38 DTS in an Information Services Director job

16.7.6 ISD 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 were already supported by the Teradata Connector in IS 8.0.1, although that connector was still restricted to a single input link.

Figure 16-39 on page 358 shows an example in which there are 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 and every 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), then the DTS stage must be used instead.

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

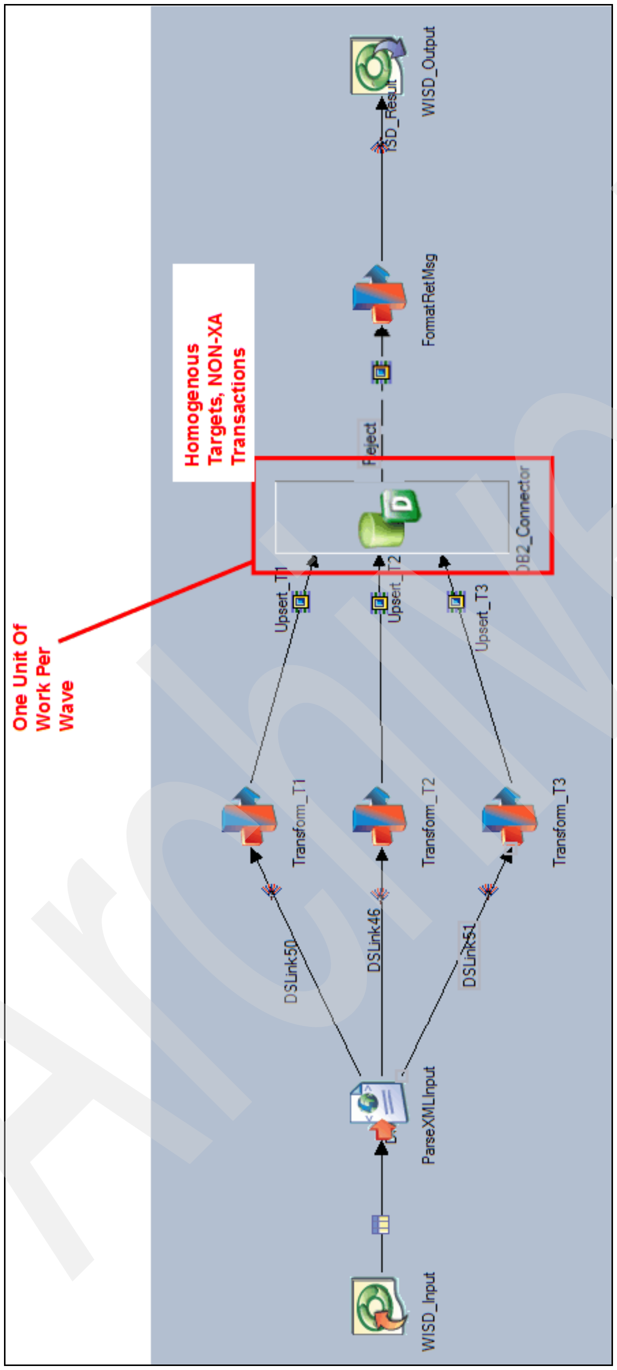


Figure 16-39 Units of work with Connector stage

16.7.7 Re-partitioning in ISD jobs

Always-on ISD jobs can take advantage of multiple partitions, as long as all partitions are collected back to a single partition prior to sending the result to the ISD output stage.

Multiple partitions provide for a way of scaling a single job instance across multiple partitions. This is a key advantage for complex services that receive request payloads of an arbitrary size and, as part of the processing of those requests, require the correlation of large amounts of data originating from sparse lookups.

One such example is an application developed for an insurance company for the approval of insurance policies. Jobs are of great complexity, with several Lookups and lots of transformation logic. That type of job design benefits from intra-job partitioning, instead of executing the entire job logic in a single partition.

With multiple partitions, there are multiple instances of stages along the way that cut significantly the response time for such complex services.

As stated, all partitions must be collected back to a single partition, which conveys the final response to the WISD Output stage.

As discussed in 16.7.3, “Scalability” on page 352, there can be multiple job instance servicing requests, as determined at deployment time through the IS Console.

As always, proper monitoring must be in place to make sure hardware resources are not saturated and there is enough spare capacity to accommodate additional partitions and job instances.

16.7.8 General considerations for using ISD jobs

In this section we provide some general considerations for when and how to use ISD with always-on jobs.

- Generally speaking, the best job candidates for service publication with Information Services Director are those with 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 water mark traffic requirements that are 300 requests per second or less

- For parallel jobs always use a single node config. The quantities of data used in a service are typically small (industry guidelines), and do not need high degrees of partitioning. Jobs are not easy to debug when deployed as services. Having multiple nodes un-necessarily complicates things. The only exception to this rule, which must be well qualified, is if the service takes little input and returns little output, but generates large scale amounts of data for intermediate processing. For instance, an internal set of lookups that generate vast amounts of rows through Cartesian product would result in the need for multiple nodes and parallelism in the midst of a service.

  Helpful hint: Use node pools in the configuration file to restrict the default node pool to one node, and have other node pools available so that stages can run in parallel for the conditions described.

- Test your jobs fully without having them deployed as services. Use Flat File Source to Flat File Target to perform QA on the logic and processing of the job, or alternatively, RowGenerator (or in server, a leading Transformer with Variable and Constraint stages). Replace the row generation stage and flat files with WISDInput and WISDOutput once tested and verified.
- Consider using shared containers for critical logic that is shared among classic style batch jobs and always on. Though not always possible because of logic constraints, this offers the opportunity to keep the core logic in one place for simpler maintenance, and only alter the sources and targets.
- Beware of binary data in character columns. Binary data, especially binary 00's, is incompatible with SOAP processing and might cause problems for the ISD Server, the eventual client, or both.
- Beware of DS Parallel settings that pad character fields with NULLs. This usually only happens with fixed length columns. Varchar is a solution, as is setting $APT_STRING_PADCHAR to a single blank (no quotes around the blank).
- When performing INSERTs to relational databases, be safe and set array and transaction sizes to 1 (one). Otherwise you might not see the new key values immediately, or until the job is complete.
- Do not expect values written to flat file and other targets to be immediately available or visible. Use a transaction size of 1 for such things.
- The text bindings (Text over JMS and Text over HTTP) exist primarily for always on jobs, and because they have no formal metadata definition on their payload, and work best for single-column input and output (XML is a common

content used for the buffers). These bindings are able to start jobs through the other two topologies. However, their payloads (the message in the requesting or input queue) cannot populate the job parameter input.

To populate individual columns and job parameters using one of the text bindings you can use values in the JMS header columns (JMS headers, custom headers, and HTTP header fields). Jobs with multi-column output can also populate the payload and header information for a message in the output or response queue (one column to the payload and the others populating header properties).

- Job sequences cannot be published as services. Alternatively, publish a job with WISD Input and WISD Output stages that contains a BASIC Transformer (or use a server job) and invokes the utility sdk routine (found in the Transformer category in the pull-down list of the Transformer) called UtilityRunJob. This allows you to call another DS job from in your WISD Job. It has the ability to wait, or immediately return after starting the remote job sequence. Another approach is to create a server job with only (manually written) job control logic. This is not considered a sequencer and can be published as a service.

16.7.9 Selecting server or EE jobs for publication through ISD

When selecting server or EE jobs for publication through ISD, parallel jobs are the preference because of the following reasons:

- Support for pipeline and partitioning parallelism.
- Must be the preferred engine for any new development.
- They can scale on cluster and grid environments, and are not restricted to run on the head node only.

  Reduce the strain on the head node and therefore it is less likely to require the load balancing across multiple service providers (DSEngines);
- Transaction support enhancements described in 16.7.5, “ISD with DTS” on page 354 and 16.7.6, “ISD with connectors” on page 357 overcome 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;
- Invocation of DS Sequences

- Sequences fall into the Batch category, and are therefore 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 by means of the utility sdk routine (find it in the Transformer category in the pull-down list of the Transformer) called UtilityRunJob.

16.8 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 obvious difference is that the first is for interoperability with JMS, and the second is for MQ.

One might use of ISD with JMS for the processing of MQ Series messages and MQ/DTS for the processing of JMS messages by setting up a bridging between MQ and JMS by means of WebSphere ESB capabilities. However, there is a relatively high complexity in the setup of a bridging between JMS and MQ.

We put both solutions side-by-side in a diagram, Figure 16-40 on page 363. The goal of this illustration is to draw a comparison of how transactions are handled and the path the data flows.

Both ISD and MQ/DTS jobs are parallel jobs, 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 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, whereas 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. This means a given transaction is not committed twice against the target database, and each and every transaction is guaranteed to be delivered to the target database. Transaction contexts are entirely managed by DS job stages. There is a local transaction managed by the MQConnector (for the transfer of messages from the source queue to the work queue) and an XA transaction involving multiple database targets and the work queue.

In an ISD job, there are transaction contexts managed by DS parallel processes, depending on how many database stages are present in the flow. See 16.7.4, “Synchronizing database stages with ISD output ” on page 353 through 16.7.6, “ISD with connectors” on page 357.

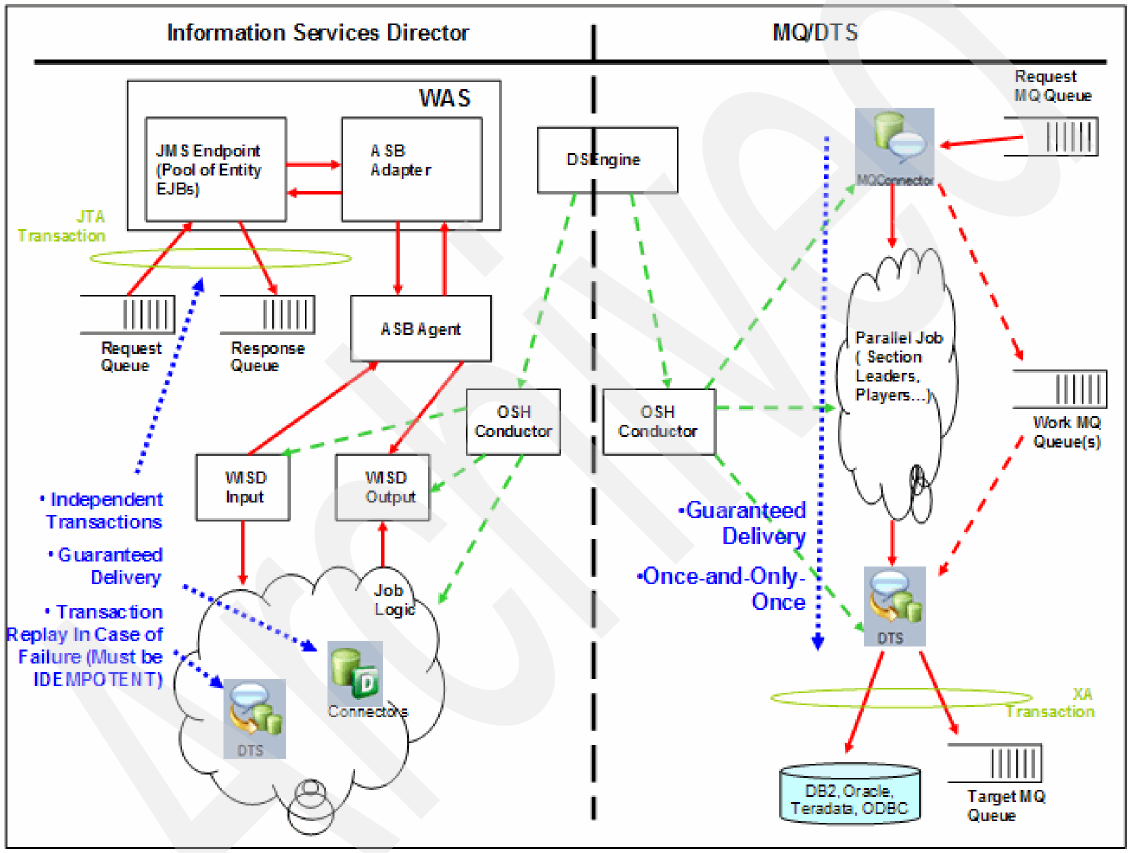


Figure 16-40 Transactional Contexts in ISD and MQ/DTS jobs

However, there is one additional transaction context one the JMS side, managed by EJBs in the WAS J2EE container as JTA transactions.

JTA transactions make sure no messages are lost. If any components along the way (WAS, 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 re-processed. The JTA transaction that picked the incoming message from the source queue is rolled back and the message remains on the source queue. This message is then re-processed upon job restart.

This means database transactions in ISD jobs exposed as JMS services must be idempotent. For the same input data, they must yield the same result on the target DB.

In summary, the strengths of the MQ/DTS and ISD/JMS solutions are as follows:

- 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.

As discussed in 16.6, “MQConnector/DTS” on page 313, the DTS stage has been 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 implies a non-trivial effort in setting up the necessary bridging between JMS and MQ on the WAS instance.

There are a few important aspects that need attention when taking advantage of ISD with JMS bindings:

- The retry attempt when JTA transactions are enabled is largely dependent upon 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.
- One problem is that there are subtle differences in all of this from provider to provider. It becomes a question of exactly how things operate when MQ is the provider, or when the embedded JMS in WAS is the provider.
- The JMS binding also creates a spectrum of other issues because of the pool of EJBs. Multiple queue listeners can result in such confusions as messages ending up out of order, and a flood of concurrent clients going into ISD, and overwhelm the number of instances that you have established for DataStage.

16.9 Payload processing

There are a number of ways of processing incoming payloads:

- Regular records
  - Possible with ISD.
  - The ISD operation must be defined in such a way its signature is mapped to records written out by the WISD Input stage.
  - With regular records, no extra parsing is needed.
- XML
  - The input MQ message, JMS message, or SOAP payload contains an XML document as a varchar or binary field.
  - The XMLInput stage must be used to parse the incoming message into multiple records of different types in different output links.
  - One might have to use a cascade of XMLInput stages depending on the complexity of the input XML.
  - XMLInput stages might have to be complemented with XSLT transformations with XMLTransformer stages.

    DS Parallel jobs can handle any type of payload, it is a matter of using the right components for the tasks at hand.
- COBOL/Complex Records
  - The input MQ message, JMS message, or SOAP payload contains a complex record as a varchar or binary field.
  - Either the Column Import or Complex Flat File stage must be used to parse the payload.
  - One might have to use a cascade of such stages, depending on the complexity of the input record format.

If the input payload is too large, you might run into memory issues. The environment variable APT_DEFAULT_TRANSPORT_BLOCK_SIZE must be tuned accordingly.

16.10 Pipeline Parallelism challenges

As discussed in Chapter 15, “Batch data flow design” on page 259, DS batch applications must be optimized to minimize the interaction with databases. Data must be loaded and extracted in bulk mode, and the correlation of data should avoid the use of sparse lookups.

As opposed to batch, real-time applications are tightly coupled with databases. These types of jobs cannot use the same load/unload and correlation techniques. The result is that they make heavy use of Sparse Lookups, so they have immediate access to the latest information stored in the target DBs. However, using Sparse Lookups is not enough to address real-time database access needs.

The nature of the parallel framework introduces interesting challenges, that are the focus of this section. This discussion applies to both MQ/DTS and ISD jobs. The challenges result from pipeline parallelism, in which there are multiple processes acting on a dataflow pipeline concurrently. The most notable challenge is key collision, for which a solution is described in the next section.

16.10.1 Key collisions

Key collision refers to a condition that involves an attempt to create two or more surrogate keys when processing multiple contiguous records that contain the same business key.

Figure 16-41 illustrates two contiguous records (R1 and R2), containing the same business key (“AA”) for which either an existing surrogate (SK) must be returned from the database, or a new SK must be created if such SK does not exist yet.

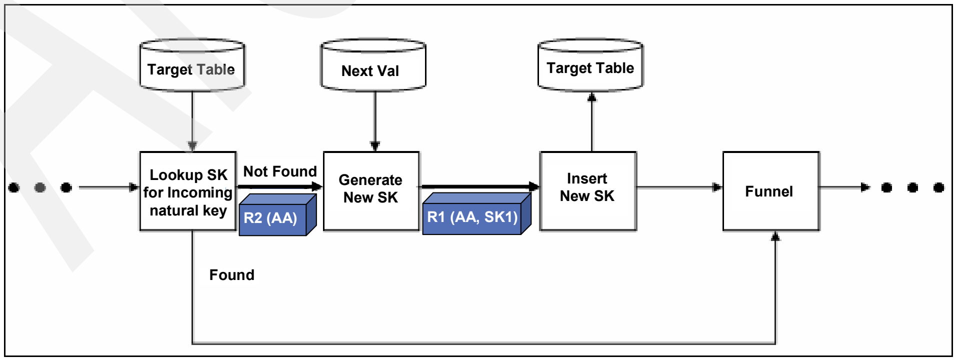


Figure 16-41 Back-to-back records leading to key collisions

The stages are represented by means of squares. They all execute as separate processes.

R1 arrives and goes through the first lookup first. An SK is not found, so it is forwarded to the Generate New SK stage. This one invokes a NextVal function in the target database to obtain a new SK value that is from now on associated to business key AA. The Generate New SK stage returns a new value, SK1, which is now propagated downstream. This value, though, is not saved in the target database, so for all purposes, AA's surrogate is not SK1 in the database yet.

R2 arrives immediately next. It also goes through the same initial lookup, and because AA is not saved yet in the target database, it is also forwarded to the Generate New SK stage.

In Figure 16-42, you can see that both records were sent downstream to the Insert New SK stage. R2 has been assigned a surrogate key value of SK2. However, only the first one is inserted successfully. The second one violates a unique key constraint and therefore from now on, R2 becomes invalid.

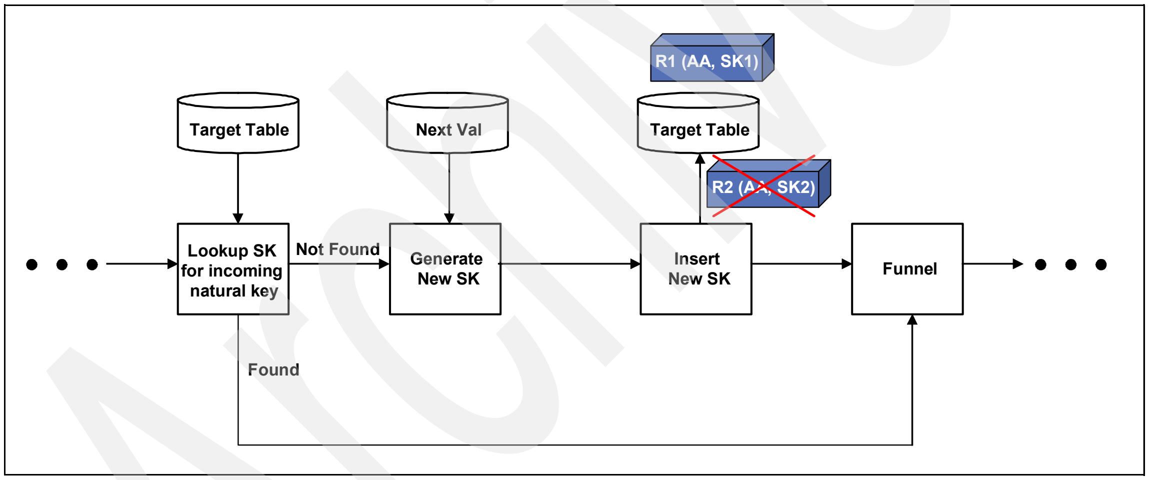


Figure 16-42 Key collision occurs

There might also be situations in which there are inter-branch dependencies, such as in Figure 16-43. One branch generates an SK value that is queried by a different branch.

This inter-branch dependency can also be extrapolated to include different jobs and even other types of applications, with all of them querying and updating surrogate keys for the same target table. So, the question is how to solve this key collision problem in a straightforward way.

Any solutions involving any type of caching or global lookup table in DataStage would not solve this problem; it would only further complicate things. The Lookup must always be done against the target database. The least a DataStage cache would do is introduce the problem of keeping that cache in synch with the target database, so this approach is not suitable.

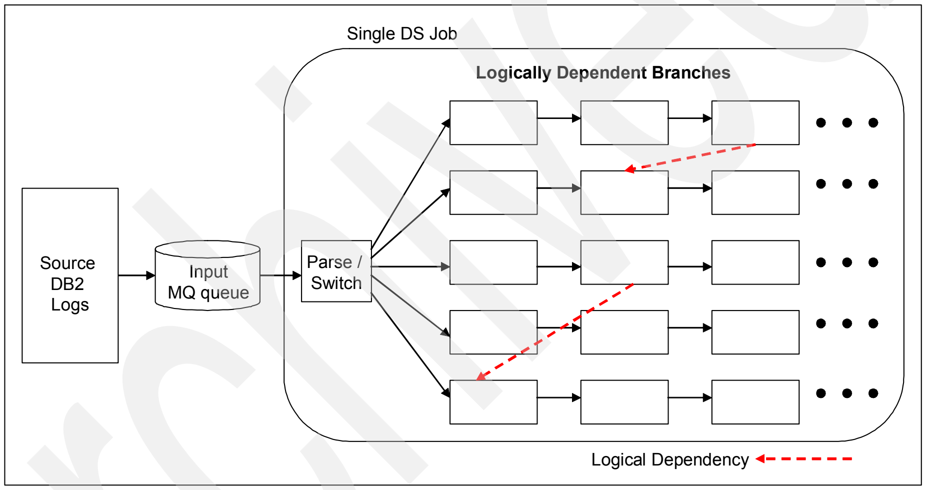


Figure 16-43 Inter-branch dependencies

16.10.2 Data stubbing

The technique that allows for circumventing the key collision issue is called *data stubbing*.

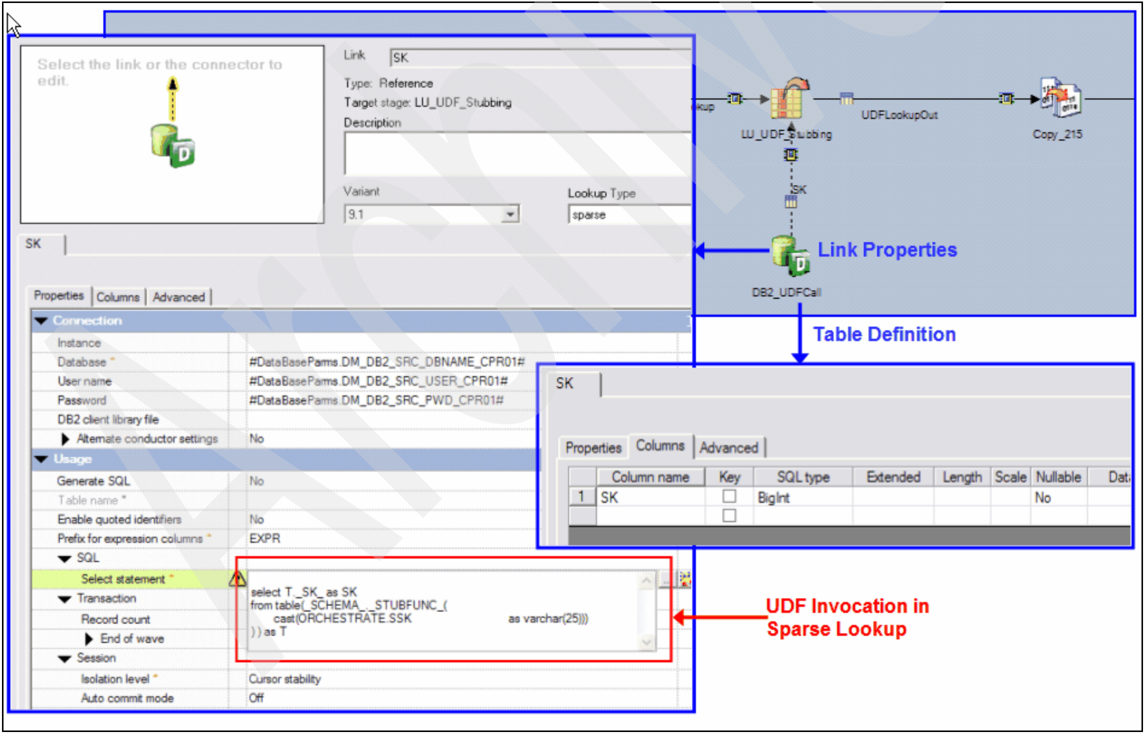
This technique involves creating a skeleton record in the target database which, for a short period of time, holds the association between a business key and its surrogate key. Upon full processing of a record through the application flow, the record and its relationships to other records are fully populated.

The implementation of this technique comprises the following:

- A stored function
  - This function implements the logic to look up an SK for an incoming business key. If such an SK is not found, it creates a new SK and stores that value in the target table, establishing the link between the business key and the newly-created surrogate key.
  - The logic is implemented as a single transaction.
  - This function becomes the access point for all applications.
- A Sparse Lookup to invoke the stored function

  All records that need to go through the lookup/SK generation process, invoke the stored function by means of a sparse lookup.

This is illustrated in Figure 16-44. There is a DB2Connector stage attached to the reference link of a Sparse Lookup stage. Every incoming record flows through this stage and result in the invocation of this function, in this particular branch of the job flow.



Link: SK

Type: Reference

Target stage: LU_UDF_Stubbing

Description:

Variant: 9.1

Lookup Type: sparse

SK

Connection	
Instance	
Database *	#DataBaseParms.DM_DB2_SRC_DBNAME_CPR01#
User name	#DataBaseParms.DM_DB2_SRC_USER_CPR01#
Password	#DataBaseParms.DM_DB2_SRC_PWD_CPR01#
DB2 client library file	
Alternate conductor settings	No
Usage	
Generate SQL	No
Table name *	
Enable quoted identifiers	No
Prefix for expression columns *	EXPR
SQL	
Select statement *	select T._SK_ as SK from table(_SCHEMA_._STUBFUNC_(cast(ORCHESTRATE.SSK as varchar(25)))) as T
Transaction	
Record count	
End of wave	
Session	
Isolation level *	Cursor stability
Auto commit mode	Off

```
select T._SK_ as SK
from table(_SCHEMA_._STUBFUNC_(
     cast(ORCHESTRATE.SSK                    as varchar(25)))
) ) as T
```

SK

	Column name	Key	SQL type	Extended	Length	Scale	Nullable	Dat
1	SK	☐	BigInt				No	
		☐						

Figure 16-44 Invoking a stubbing UDF with a sparse lookup

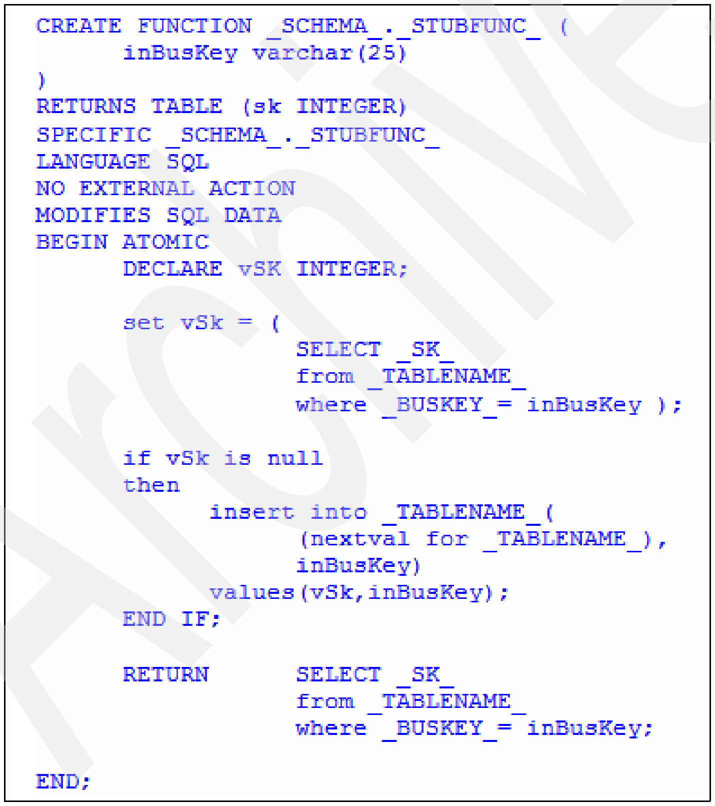
The red square in Figure 16-44 on page 369 shows how a hypothetical function _SUBFUNC_ is invoked. This example is for DB2 and it uses a special syntax as illustrated in Figure 16-44 on page 369.

The UDF returns a column named *SK*, which is propagated downstream. This column is defined in the output link for the DB2 Connector stage.

DB2 functions

The code for the sample DB2 UDF is listed in Figure 16-45. The function is defined as ATOMIC.

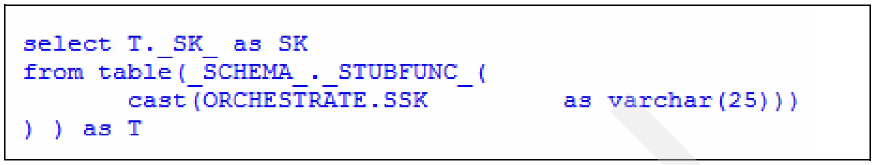
It first queries the target table _TABLENAME_. If no value is found, it inserts a new record, invoking the Nextval function on a database sequence for this table in the insert's value list. The final statement returns whatever value is present in the table for the incoming business key at that point.



```
CREATE FUNCTION _SCHEMA_._STUBFUNC_ (
      inBusKey varchar(25)
)
RETURNS TABLE (sk INTEGER)
SPECIFIC _SCHEMA_._STUBFUNC_
LANGUAGE SQL
NO EXTERNAL ACTION
MODIFIES SQL DATA
BEGIN ATOMIC
      DECLARE vSK INTEGER;

      set vSk = (
                  SELECT _SK_
                  from _TABLENAME_
                  where _BUSKEY_= inBusKey );

      if vSk is null
      then
            insert into _TABLENAME_(
                  (nextval for _TABLENAME_),
                  inBusKey)
            values(vSk,inBusKey);
      END IF;

      RETURN      SELECT _SK_
                  from _TABLENAME_
                  where _BUSKEY_= inBusKey;

END;
```

Figure 16-45 A DB2 UDF that implements a data stubbing function

The UDF is invoked by means of a select statement as illustrated in Figure 16-46. It requires the use of the Cast function to pass in the arguments.

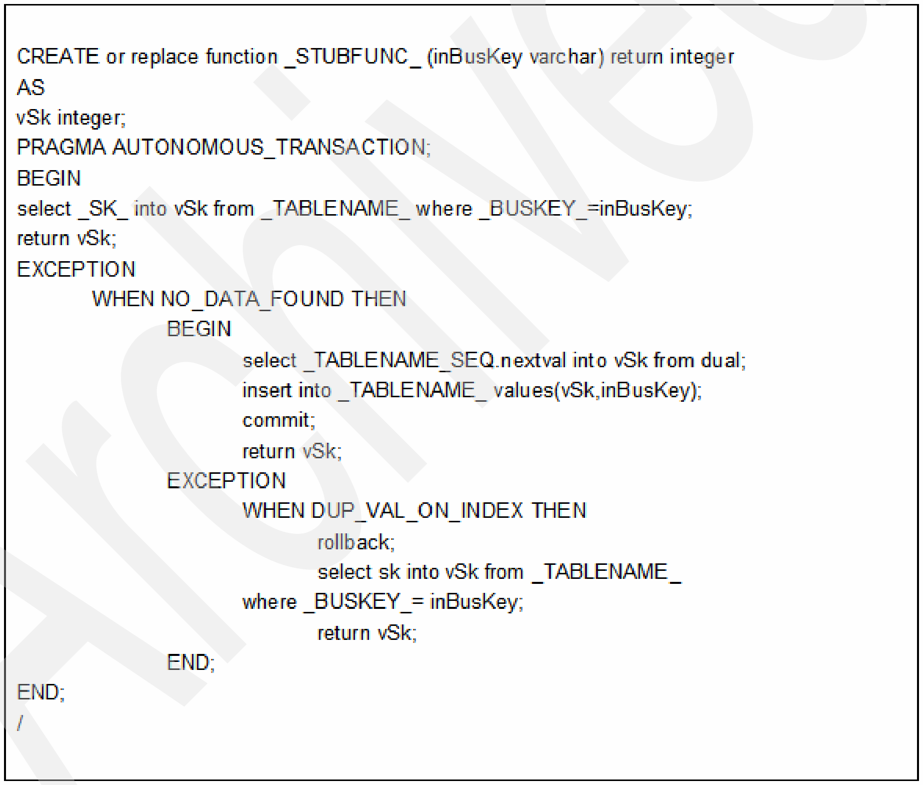


```
select T._SK_ as SK
from table(_SCHEMA_._STUBFUNC_(
       cast(ORCHESTRATE.SSK          as varchar(25)))
) ) as T
```

Figure 16-46 Sample query to invoke a stubbing DB2 UDF

Oracle stored functions

Figure 16-47 shows an equivalent Oracle stored function. The syntax is different, but the logic is the same as the DB2 version.

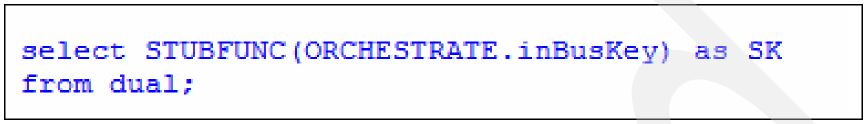


```
CREATE or replace function _STUBFUNC_ (inBusKey varchar) return integer
AS
vSk integer;
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
select _SK_ into vSk from _TABLENAME_ where _BUSKEY_=inBusKey;
return vSk;
EXCEPTION
        WHEN NO_DATA_FOUND THEN
                BEGIN
                        select _TABLENAME_SEQ.nextval into vSk from dual;
                        insert into _TABLENAME_ values(vSk,inBusKey);
                        commit;
                        return vSk;
                EXCEPTION
                        WHEN DUP_VAL_ON_INDEX THEN
                                rollback;
                                select sk into vSk from _TABLENAME_
                        where _BUSKEY_= inBusKey;
                                return vSk;
                END;
END;
/
```

Figure 16-47 An Oracle stored function that implements data stubbing

The Oracle stored function needs to be created with the PRAGMA AUTONOMOUS_TRANSACTION option, so that it can be invoked in the SELECT clause of a query. By default, select statements cannot invoke UDFs that contain DML statements. That PRAGMA directive makes that possible. The only catch is that the transaction inside the function is independent from the transaction in which the calling SQL is being executed. That is acceptable, because the sparse lookups run independently from the target DTS or database Connectors anyway.

The syntax to invoke this Oracle stored function is shown in Figure 16-48.



```
select STUBFUNC(ORCHESTRATE.inBusKey) as SK
from dual;
```

Figure 16-48 Sample query to invoke a stubbing Oracle function

16.10.3 Parent/Child processing

Now that we have a notion of the parallelism challenges and solutions, we can discuss the specific problem of handing parent/child relationships. The solution builds upon the use of the previous technique.

The assumption is that the SKs are not generated automatically at the time of insert. Instead, the SK can be obtained by invoking a stubbing UDF to obtain the surrogate key for the parent table. This SK value is propagated downstream, in both parent and children links. The DTS or target database Connector stage then commits the records as a single unit of work.

16.11 Special custom plug-ins

A custom solution was implemented to support the integration of DataStage into a Tibco application. The advantage of this solution is that it integrates DataStage directly with a third-party JMS provider, without the need to set up bridges between WAS JMS and the external JMS provider.

Also, the JMSPlug-in runs entirely outside of a J2EE container, making the interoperability with the third-party JMS provider straightforward. This is illustrated in Figure 16-49 on page 373.

This solution guarantees the delivery of a message to a target queue. It uses techniques similar to the work queues in the MQ/DTS solution.

However, because the source and target JMSPlug-in stages run on standalone JVMs, outside of J2EE containers, there is no support for JTA transactions. Messages are never lost, but they might be re-processed in the event of restart after failure. Therefore, database transactions along the job flow must be idempotent.

This solution is described in the document *Integrating JMS and DataStage Jobs with the JMSPlug-in* available from IBM Information Management Professional Services.

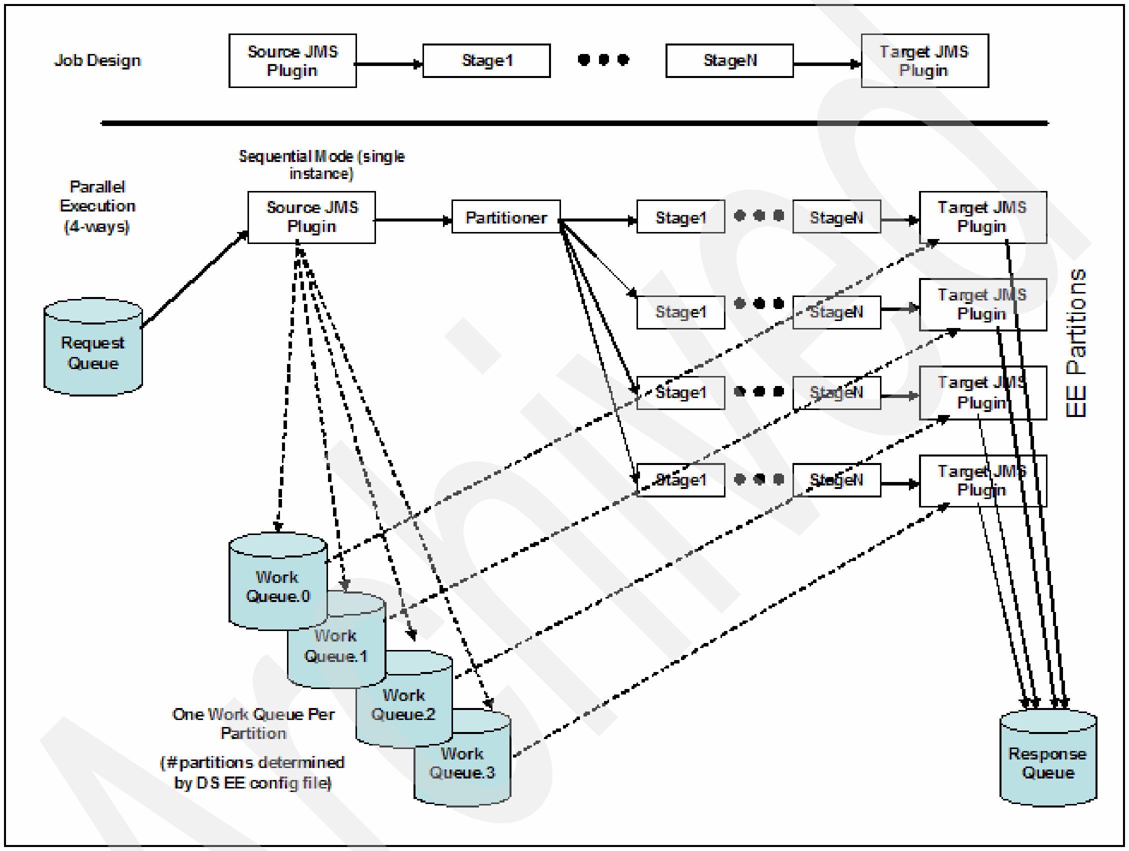


Figure 16-49 The JMSPlug-in

16.12 Special considerations for QualityStage

QualityStage jobs developed for use under WISD need to be conscious of the concept described in the previous section, especially when feeding data into a reference match.

An often desired pattern is to feed the incoming service-based data as the primary link for the reference match, and bring in reference data from a fixed data source. This violates the rules described in 16.7.2, “Design topology rules for always-on ISD jobs” on page 351.

Instead of having a fixed data source attached to the reference link, perform a Lookup based upon your high level blocking factors in the incoming WISD request.

This involves using a Copy stage that splits the incoming row, sending a row to the primary input as before when sending the other row to a Lookup where multiple reference candidates can be dynamically retrieved.

Be sure to make this a Sparse Lookup if you expect that the source data could change when the WISD job is on (enabled), and check it carefully to be sure you have set it to return multiple rows.

A

Runtime topologies for distributed transaction jobs

There are several ways in which DT jobs can be deployed. The MQ Connector and DT stage components need to offer flexibility so that the users can select the appropriate topology for their particular use case. The choice of topology is dependent upon the nature of the source data. Functional requirements that influence the appropriate topology are as follows:

- Order

  This requirement governs whether the messages have to be processed in the order they are written to the source queue. If the order must be maintained, then it is not possible to execute the job in parallel, because there is no co-ordination between player processes on multiple nodes.

- Relationships

  This requirement governs whether source messages are related, that is, they have a key field that indicates they need to be processed as a unit. A hash partitioner is used to ensure that all messages with a given key are processed by the same node.

In the diagrams in this section the numbers under the queues represent message sequence numbers and illustrate how messages might be distributed across queues. Letters represent hash partitioning key fields. The solid arrows show the movement of MQ messages to and from queues. The dashed lines represent the job links. For clarity, only MQ and DT stages are shown in these jobs, but in reality there are other stages in between to implement the business logic of the extract, transform, and load (ETL) job.

A.1 No ordering, no relationships

If the order of processing of source messages is unimportant, and there is no relationship between messages, then it is possible to run the jobs fully in parallel.

Each node contains an MQ stage, a work queue, and a DT stage. The MQ stages access a single source queue and distribute these to the nodes. Because the MQ stage instances are reading the messages destructively off the source queue, there is no contention for messages. The work queues are required, because the instances of the MQ Connector source stages must remove the message from the source queue. These stages cannot browse the queue, because all instances would then see and process the same messages. This is depicted in Figure A-1.

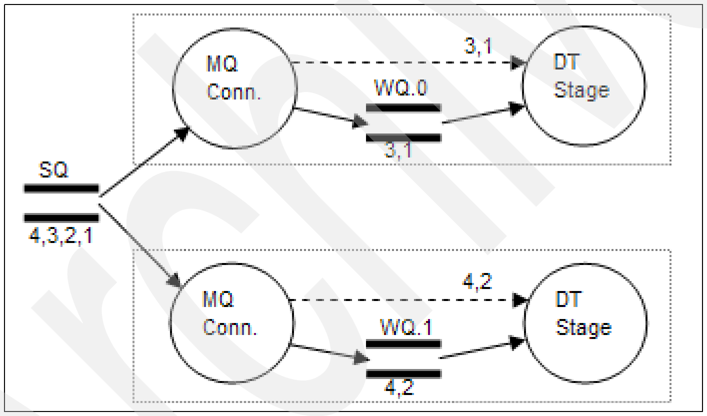


Figure A-1 No order, no relationships

The reason to have multiple work queues is to restart jobs upon catastrophic failure. Consider the case of a single work queue, depicted in Figure A-2.

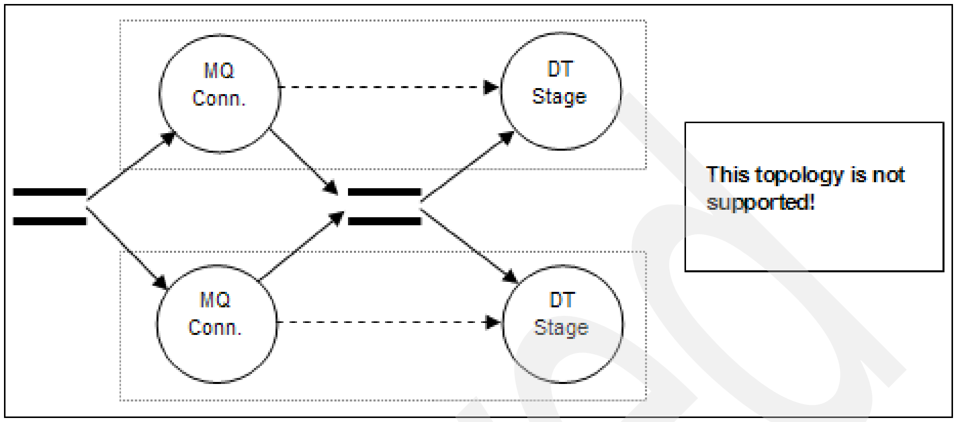


Figure A-2 Unsupported topology

If a restart is required, each MQ Connector stage player process attempts to browse messages from the single work queue. The MQ stages have to browse the queue because they must leave the messages on the work queue. This cannot be supported. It is not possible to have multiple processes browse the same queue because they all read the same messages. With multiple work queues, this problem does not arise, because each MQ process has exclusive access to browse the queue in the event of a restart.

Multiple work queues also aid performance, because there is no contention for work queues, and the DTStage is more likely to find its message from the head of the queue.

A.2 No ordering, with relationships

If there is a need to ensure that all messages that are related to each other by a shared key value are sent to the same node, a single MQ stage combined with the use of a hash partitioner is used. This is depicted in Figure A-3.

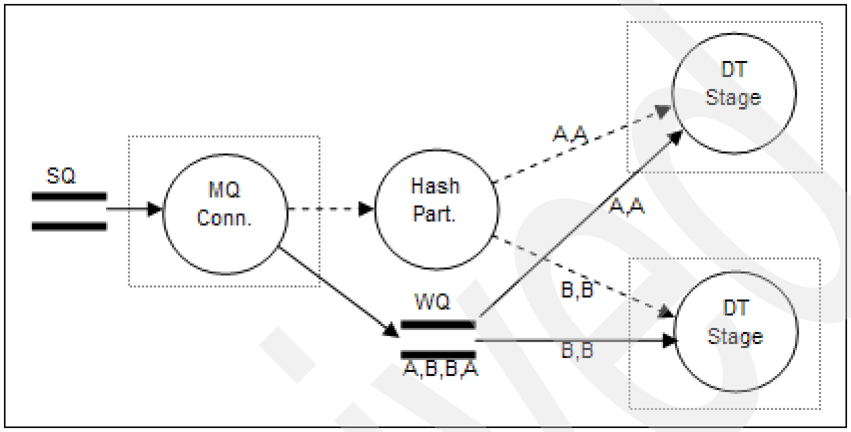


Figure A-3 No ordering, with relationships

In this scenario, there is a single work queue, because the MQ Connector cannot determine which node is targeted for a specific message. By contrast, the MQRead operator is implemented as a combinable operator, and can be combined with the hash partitioner. This permits MQRead to learn to which partition the message is sent, and can therefore direct the message to the appropriate work queue if multiple work queues are used. MQRead can therefore use multiple work queues, but the MQ Connector cannot.

Whether there is a significant performance hit by using a single work queue rather than multiple queues needs to be determined. Any impact is due to locking mechanisms on the queue, and potentially having to search further in the queue to match messages by ID.

Ordering

If there is a need to process the messages in the order they arrive on the source queue it is necessary to execute the entire job sequentially. The reason for this is that there is no synchronization between nodes, and so distributing messages to multiple nodes cannot guarantee any ordering of messages. The topology would appear as shown in Figure A-4.

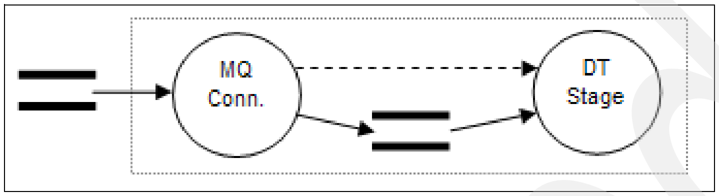


Figure A-4 Strict ordering

A.3 Bypassing work queues

The scenarios depicted in section A.2, "No ordering, with relationships" on page 378 and section "Ordering" on page 379 can be modified. It is possible to configure both the MQ and DT stage to operate in absence of a work queue. When running with no work queue, the MQ connector would browse the source queue rather than destructively getting messages from the source queue. The DT stage then accesses the same source queue and destructively get the messages.

The revised topologies are depicted in Figure A-5 and Figure A-6 on page 380. When no ordering is required, it is as depicted in Figure A-5.

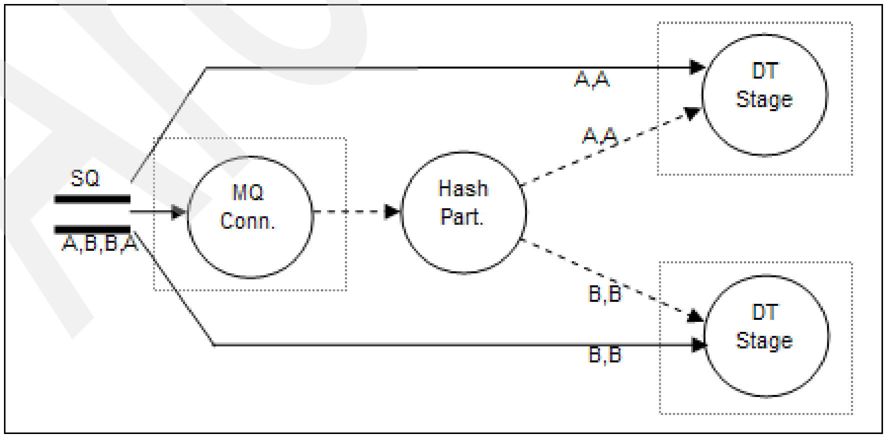


Figure A-5 No work queues, no ordering

When ordering is required, it is as depicted in Figure A-6.

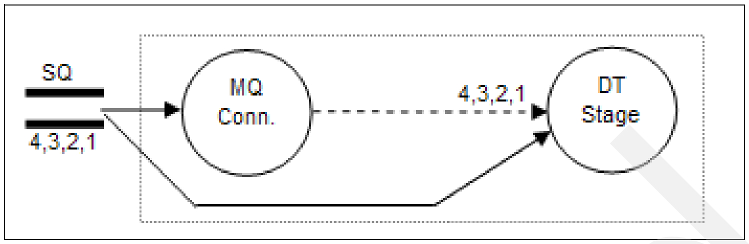


Figure A-6 No work queues, with ordering

One advantage of running without a work queue is that restart after failure is simpler. The job can be restarted, and it continues from where it was aborted. Additionally, evidence with the MQRead operator indicates that reading from a source queue and writing to a work queue under sync point control for a small transaction size (small number of messages) is an expensive operation. By omitting the need to write to a work queue, the overall performance is improved.

There are dangers in this approach however. Prior work with MQ and WebSphere TX determined two scenarios where source messages can be missed due to the message cursor not detecting messages:

If multiple processes are writing to the source queue, the queue browser might miss a message if the PUT and COMMIT calls from these processes are interspersed in a certain order.

If the writing processes use message priorities, the queue browser does not see messages of a higher priority, as they jump ahead of the current cursor position.

The solution offers support for all of these scenarios. It is the responsibility of the job designer to select the appropriate settings to configure the stages to enable a particular scenario.

B

Standard practices summary

In this appendix we summarize suggestions that have been outlined in this document, along with cross-references for more detail.

B.1 Standards

It is important to establish and follow consistent standards in directory structures for install and application support directories. An example directory naming structure is given in 3.1, “Directory structures” on page 22.

Directory Structures

IBM InfoSphere DataStage requires file systems to be available for the following elements:

- Software Install Directory

  IBM InfoSphere DataStage executables, libraries, and pre-built components
- DataStage Project Directory
- Runtime information (compiled jobs, OSH scripts, generated BuildOps and Transformers, logging info);
- Data Storage
  - DataStage temporary storage: Scratch, temp, buffer
  - DataStage parallel dataset segment files
  - Staging and Archival storage for any source files

By default, these directories (except for file staging) are created during installation as subdirectories under the base InfoSphere DataStage installation directory.

In addition to the file systems listed, a DataStage project also requires a proper amount of space in the Metadata layer (which is a relational database system). As opposed to the Project directory, the Metadata layer stores the design time information, including job and sequence designs, and table definitions.

This section does not include requirements and recommendations for other Information Server layers (Metadata and Services) The discussion here is strictly in terms of the Engine layer.

A single Information Server instance (services and metadata) can manage multiple engines. The following discussion pertains to the setup of a single engine instance.

Data, install, and project directory structure

This directory structure contains naming conventions, especially for DataStage Project categories, stage names, and links. An example DataStage naming structure is given in 3.2, “Naming conventions” on page 32.

All DataStage jobs must be documented with a short description field, as well as with annotation fields. See 3.3, “Documentation and annotation” on page 47.

It is the DataStage developer’s responsibility to make personal backups of work on local workstations, using the DS Designer DSX export capability. This can also be used for integration with source code control systems. See 3.4, “Working with source code control systems” on page 50.

B.2 Development guidelines

Modular development techniques must be used to maximize re-use of DataStage jobs and components, as outlined in Chapter 5, “Development guidelines” on page 69.

The following list is a summary of those development guidelines:

- Job parameterization allows a single job design to process similar logic instead of creating multiple copies of the same job. The Multiple-Instance job property allows multiple invocations of the same job to run simultaneously.
- A set of standard job parameters must be used in DataStage jobs for source and target database parameters (DSN, user, password, and so forth) and directories where files are stored. To ease re-use, these standard parameters and settings must be made part of a designer job template.
- Create a standard directory structure outside of the DataStage project directory for source and target files, intermediate work files, and so forth.
- Where possible, create re-usable components such as parallel shared containers to encapsulate frequently-used logic.

DataStage Template jobs must be created with the following elements:

- Standard parameters (as examples, source and target file paths, and database login properties).
- Environment variables and their default settings
- Annotation blocks

Job parameters must be used for file paths, file names, and database login settings.

Parallel shared containers must be used to encapsulate frequently-used logic, using RCP to maximize re-use.

Standardized error handling routines must be followed to capture errors and rejects. Further details are provided in 5.6, “Error and reject record handling” on page 74.

B.3 Component usage

As discussed in 5.7, “Component usage” on page 85, the following guidelines must be used when constructing parallel jobs with DataStage:

- Never use Server Edition components (BASIC Transformer, Server Shared Containers) in a parallel job. BASIC routines are appropriate only for job control sequences.
- Always use parallel datasets for intermediate storage between jobs.
- Use the Copy stage as a placeholder for iterative design, and to facilitate default type conversions.
- Use the parallel Transformer stage (not the BASIC Transformer) instead of the Filter or Switch stages.
- Use BuildOp stages only when logic cannot be implemented in the parallel Transformer.

B.4 DataStage data types

Be aware of the mapping between DataStage (SQL) data types and the internal DS parallel data types, as outlined in Appendix H, “DataStage data types” on page 423.

Use default type conversions using the Copy stage or across the Output mapping tab of other stages.

B.5 Partitioning data

Given the numerous options for keyless and keyed partitioning, the following objectives help to form a methodology for assigning partitioning:

- Objective 1: Choose a partitioning method that gives close to an equal number of rows in each partition, and minimizes overhead. This ensures that the processing workload is evenly balanced, minimizing overall run time.
- Objective 2: The partition method must match the business requirements and stage functional requirements, assigning related records to the same partition, if required.

  Any stage that processes groups of related records (generally using one or more key columns) must be partitioned using a keyed partition method.

  This includes, but is not limited to the following stages: Aggregator, Change Capture, Change Apply, Join, Merge, Remove Duplicates, and Sort. It might also be necessary for Transformers and BuildOps that process groups of related records.

  In satisfying the requirements of this second objective, it might not be possible to choose a partitioning method that gives close to an equal number of rows in each partition.
- Objective 3: Unless partition distribution is highly skewed, minimize repartitioning, especially in cluster or Grid configurations.

  Repartitioning data in a cluster or grid configuration incurs the overhead of network transport.
- Objective 4: Partition method must not be overly complex.

  The simplest method that meets these objectives is generally the most efficient and yield the best performance.

Using the above objectives as a guide, the following methodology can be applied:

- Start with Auto partitioning (the default).
- Specify Hash partitioning for stages that require groups of related records.
  - Specify only the key columns that are necessary for correct grouping as long as the number of unique values is sufficient.
  - Use Modulus partitioning if the grouping is on a single integer key column.
  - Use Range partitioning if the data is highly skewed and the key column values and distribution do not change significantly over time (Range Map can be reused).

- If grouping is not required, use Round-robin partitioning to redistribute data equally across all partitions.

  This is especially useful if the input dataset is highly skewed or sequential.
- Use Same partitioning to optimize end-to-end partitioning and to minimize repartitioning:
  - Being mindful that Same partitioning retains the degree of parallelism of the upstream stage.
  - In a flow, examine up-stream partitioning and sort order and attempt to preserve for down-stream processing. This might require re-examining key column usage in stages and re-ordering stages in a flow (if business requirements permit).

Across jobs, persistent datasets can be used to retain the partitioning and sort order. This is particularly useful if downstream jobs are run with the same degree of parallelism (configuration file) and require the same partition and sort order.

Further details about partitioning methods can be found in Chapter 6, “Partitioning and collecting” on page 91.

B.6 Collecting data

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 Auto partitioning (the default).
- When the input dataset has been sorted in parallel, use Sort Merge collector to produce a single, globally sorted stream of rows.

  When the input dataset has been 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 datasets, as long as the dataset has not been repartitioned or reduced.

Further details on partitioning methods can be found in Chapter 6, “Partitioning and collecting” on page 91.

B.7 Sorting

Using the rules and behavior outlined in Chapter 7, “Sorting” on page 115, apply the following methodology when sorting in a DataStage parallel data flow:

- Start with a link sort.
- Specify only necessary key columns.
- Do not use Stable Sort unless needed.
- Use a stand-alone Sort stage instead of a Link sort for options that not available on a Link sort:
  - Sort Key Mode, Create Cluster Key Change Column, Create Key Change Column, Output Statistics
  - Always specify DataStage Sort Utility for standalone Sort stages
  - Use the “Sort Key Mode=Do not Sort (Previously Sorted)” to resort a sub-grouping of a previously-sorted input dataset
- Be aware of automatically-inserted sorts.

  Set $APT_SORT_INSERTION_CHECK_ONLY to verify but not establish required sort order
- Minimize the use of sorts in a job flow.
- To generate a single, sequential ordered result set use a parallel Sort and a Sort Merge collector.

B.8 Stage-specific guidelines

As discussed in Section 9.1.1, “Transformer NULL handling and reject link” on page 140, precautions must be taken when using expressions or derivations on nullable columns in the parallel Transformer:

- Always convert nullable columns to in-band values before using them in an expression or derivation.
- Always place a reject link on a parallel Transformer to capture / audit possible rejects.

The Lookup stage is most appropriate when reference data is small enough to fit into available memory. If the datasets are larger than available memory resources, use the Join or Merge stage. See 10.1, “Lookup versus Join versus Merge” on page 150.

Limit the use of database Sparse Lookups to scenarios where the number of input rows is significantly smaller (for example 1:100 or more) than the number of reference rows, or when exception processing.

Be particularly careful to observe the nullability properties for input links to any form of Outer Join. Even if the source data is not nullable, the non-key columns must be defined as nullable in the Join stage input to identify unmatched records. See 10.2, “Capturing unmatched records from a Join” on page 150.

Use Hash method Aggregators only when the number of distinct key column values is small. A Sort method Aggregator must be used when the number of distinct key values is large or unknown.

B.9 Database stage guidelines

Where possible, use the Native Parallel Database stages for maximum performance and scalability, as discussed in section 13.1.1, “Existing database stage types” on page 190.

Native Parallel Database stages are as follows:

- DB2/UDB Enterprise
- Informix Enterprise
- ODBC Enterprise
- Oracle Enterprise
- Netezza Enterprise
- SQL Server Enterprise
- Teradata Enterprise

The ODBC Enterprise stage can only be used when a native parallel stage is not available for the given source or target database.

When using Oracle, DB2, or Informix databases, use **`orchdbutil`** to import design metadata.

Care must be taken to observe the data type mappings documented in Chapter 13, “Database stage guidelines” on page 189, when designing a parallel job with DataStage.

If possible, use a SQL where clause to limit the number of rows sent to a DataStage job.

Avoid the use of database stored procedures on a per-row basis in a high-volume data flow. For maximum scalability and parallel performance, it is best to implement business rules natively using DataStage parallel components.

B.10 Troubleshooting and monitoring

Always test DS parallel jobs with a parallel configuration file ($APT_CONFIG_FILE) that has two or more nodes in its default pool.

Check the Director log for warnings, which might indicate an underlying problem or data type conversion issue. All warnings and failures must be addressed (and removed if possible) before deploying a DataStage job.

The environment variable $DS_PX_DEBUG can be used to capture all generated OSH, error and warning messages from a running DS parallel job.

Set the environment variable $OSH_PRINT_SCHEMAS to capture actual runtime schema to the Director log. Set $DS_PX_DEBUG if the schema record is too large to capture in a Director log entry.

Enable $APT_DUMP_SCORE by default, and examine the job score by following the guidelines outlined in Appendix E, “Understanding the parallel job score” on page 401.

C

DataStage naming reference

Every name must be based on a three-part concept: Subject, Subject Modifier, Class Word where the following frequently-used class words describe the object type, or the function the object performs. In this appendix we provide a number of tables, each for a specific category for easier access and understanding, with the DataStage naming references. The first is Table C-1.

Table C-1 Project repository and components

Project Repository and Components	
Development	Dev_<proj>
Integration Test	IT_<proj>
User Acceptance Test	UAT_<proj>
Production	Prod_<proj>
BuildOp	BdOp<name>
Parallel External Function	XFn<name>
Wrapper	Wrap<name>

The next is in Table C-2.

Table C-2 Job names and properties

Job Names and Properties	
Extract Job	Src<job>
Load	Load<job>
Sequence	<job>_Seq
Parallel Shared Container	<job>Psc
Server Shared Container	<job>Ssc
Parameter	<name>_parm

The next is in Table C-3.

Table C-3 Sequencer

Sequencer	
Job Activity	Job
Routine Activity	Rtn
Sequencer (All)	SeqAll
Sequencer (Any)	SeqAny
Notify	Notify
Sequencer Links (messages)	msg_

The next is in Table C-4.

Table C-4 Links

Links (prefix with “lnk_”)	
Reference (Lookup)	Ref
Reject (Lookup, File, DB)	Rej
Get (Shared Container)	Get
Put (Shared Container)	Put
Input	In
Output	Out
Delete	Del
Insert	Ins
Update	Upd

The next is in Table C-5.

Table C-5 Data store

Data Store	
Database	DB
Stored Procedure	SP
Table	Tbl
View	View
Dimension	Dim
Fact	Fact
Source	Src
Target	Tgt

The next is in Table C-6.

Table C-6 Development and debug stages

Development / Debug stages	
Column Generator	CGen
Head	Head
Peek	Peek
Row Generator	RGen
Sample	Smpl
Tail	Tail

The next is in Table C-7.

Table C-7 File stages

File stages	
Sequential File	SF
Complex Flat File	CFF
File Set	FS
Parallel dataset	DS
Lookup File Set	LFS
External Source	XSrc
External Target	XTgt
Parallel SAS dataset	SASd

The next is in Table C-8.

Table C-8 Processing stages

Processing stages	
Aggregator	Agg
Change Apply	ChAp
Change Capture	ChCp
Copy	Cp
Filter	Filt
Funnel	Funl
Join (Inner)	InJn
Join (Left Outer)	LOJn
Join (Right Outer)	ROJn
Join (Full Outer)	FOJn
Lookup	Lkp
Merge	Mrg
Modify	Mod
Pivot	Pivt
Remove Duplicates	RmDp
SAS processing	SASp
Sort	Srt
Surrogate Key Generator	SKey
Switch	Swch

The next is in Table C-9.

Table C-9 Transformer stage

Transformer stage	
Transformer (native parallel)	Tfm
BASIC Transformer (Server)	BTfm
stage Variable	SV

The next is in Table C-10.

Table C-10 Real-time stages

Real Time stages	
RTI Input	RTIi
RTI Output	RTIo
XML Input	XMLi
XML Output	XMLo
XML Transformer	XMLt

The next is in Table C-11.

Table C-11 Restructure stages

Restructure stages	
Column Export	CExp
Column Import	CImp

D

Example job template

This section summarizes the suggested job parameters for all DataStage jobs, and presents them in the following tables. These might be defined in a job template, which can be used by all developers for creating new parallel jobs.

Table D-1 Suggested environment variables for all jobs

Environment Variable	Setting	Description
$APT_CONFIG_FILE	filepath	Specifies the full path name to the DS parallel configuration file. This variable must be included in all job parameters so that it can be easily changed at runtime.
$APT_DUMP_SCORE		Outputs parallel score dump to the DataStage job log, providing detailed information about actual job flow including operators, processes, and datasets. Extremely useful for understanding how a job actually ran in the environment.
$OSH_ECHO		Includes a copy of the generated osh in the job's DataStage log
$APT_RECORD_COUNTS		Outputs record counts to the DataStage job log as each operator completes processing. The count is per operator per partition. This setting must be disabled by default, but part of every job design so that it can be easily enabled for debugging purposes.
$APT_PERFORMANCE_DATA	$UNSET	If set, specifies the directory to capture advanced job runtime performance statistics.
$OSH_PRINT_SCHEMAS		Outputs actual runtime metadata (schema) to DataStage job log. This setting must be disabled by default, but part of every job design so that it can be easily enabled for debugging purposes.
$APT_PM_SHOW_PIDS		Places entries in DataStage job log showing UNIX process ID (PID) for each process started by a job. Does not report PIDs of DataStage "phantom" processes started by Server shared containers.
$APT_BUFFER_MAXIMUM_TIMEOUT		Maximum buffer delay in seconds

Table D-2 Project_Plus environment variables

Name	Type	Prompt	Default Value
$PROJECT_PLUS_DATASETS	String	Project + Dataset descriptor dir	$PROJDEF
$PROJECT_PLUS_LOGS	String	Project + Log dir	$PROJDEF
$PROJECT_PLUS_PARAMS	String	Project + Parameter file dir	$PROJDEF
$PROJECT_PLUS_SCHEMAS	String	Project + Schema dir	$PROJDEF
$PROJECT_PLUS_SCRIPTS	String	Project + Scripts dir	$PROJDEF

Table D-3 Staging environment variables

Name	Type	Prompt	Default Value
$STAGING_DIR	String	Staging directory	$PROJDEF
$PROJECT_NAME	String	Project name	$PROJDEF
$DEPLOY_PHASE	String	Deployment phase	$PROJDEF

Table D-4 Job control parameters

Name	Type	Prompt	Default Value
JOB_NAME_parm	String	Job Name	
RUN_ID_parm	String	Run ID	

E

Understanding the parallel job score

DataStage parallel jobs are independent of the actual hardware and degree of parallelism used to run them. The parallel configuration file provides a mapping at runtime between the compiled job and the actual runtime infrastructure and resources by defining logical processing nodes.

At runtime, the DS parallel framework uses the given job design and configuration file to compose a job score that details the processes created, degree of parallelism and node (server) assignments, and interconnects (datasets) between them. Similar to the way a parallel database optimizer builds a query plan, the parallel job score performs the following tasks:

- Identifies degree of parallelism and node assignments for each operator
- Details mappings between functional (stage/operator) and actual operating system processes
- Includes operators automatically inserted at runtime:
  - Buffer operators to prevent deadlocks and optimize data flow rates between stages
  - Sorts and Partitioners that have been automatically inserted to ensure correct results

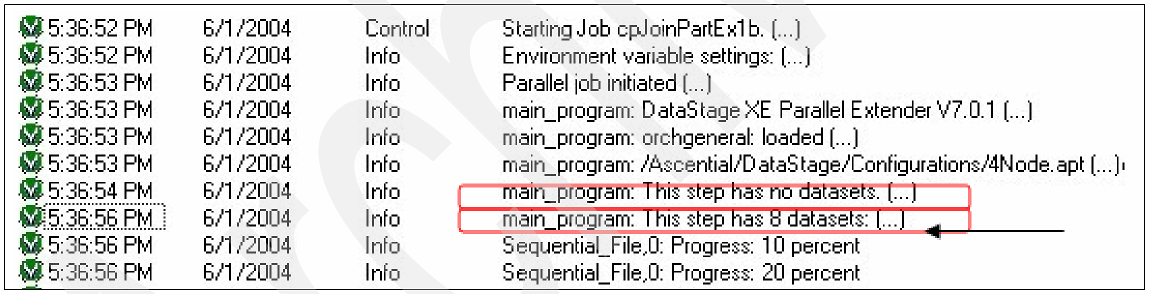
- Outlines connection topology (datasets) between adjacent operators and persistent datasets
- Defines number of actual operating system processes

Where possible, multiple operators are combined in a single operating system process to improve performance and optimize resource requirements.

E.1 Viewing the job score

When the environment variable APT_DUMP_SCORE is set, the job score is output to the DataStage Director log. It is recommended that this setting be enabled by default at the project level, as the job score offers invaluable data for debugging and performance tuning, and the overhead to capture the score is negligible.

As shown in Figure E-1, job score entries start with the phrase `main_program: This step has` *n* `datasets` Two separate scores are written to the log for each job run. The first score is from the license operator, not the actual job, and can be ignored. The second score entry is the actual job score.



```
5:36:52 PM   6/1/2004   Control   Starting Job cpJoinPartEx1b. (...)
5:36:52 PM   6/1/2004   Info      Environment variable settings: (...)
5:36:53 PM   6/1/2004   Info      Parallel job initiated (...)
5:36:53 PM   6/1/2004   Info      main_program: DataStage XE Parallel Extender V7.0.1 (...)
5:36:53 PM   6/1/2004   Info      main_program: orchgeneral: loaded (...)
5:36:53 PM   6/1/2004   Info      main_program: /Ascential/DataStage/Configurations/4Node.apt (...)
5:36:54 PM   6/1/2004   Info      main_program: This step has no datasets: (...)
5:36:56 PM   6/1/2004   Info      main_program: This step has 8 datasets: (...)
5:36:56 PM   6/1/2004   Info      Sequential_File,0: Progress: 10 percent
5:36:56 PM   6/1/2004   Info      Sequential_File,0: Progress: 20 percent
```

Figure E-1 Job score sample

E.2 Parallel job score components

The parallel job score is divided into two sections, as shown in Figure E-2 on page 404:

- Datasets

  Starts with the words `main_program: This step has n datasets:`

  The first section details all datasets, including persistent (on disk) and virtual (in memory, links between stages). Terminology in this section can be used to identify the type of partitioning or collecting that was used between operators. In this example, there are two virtual datasets.

- Operators:

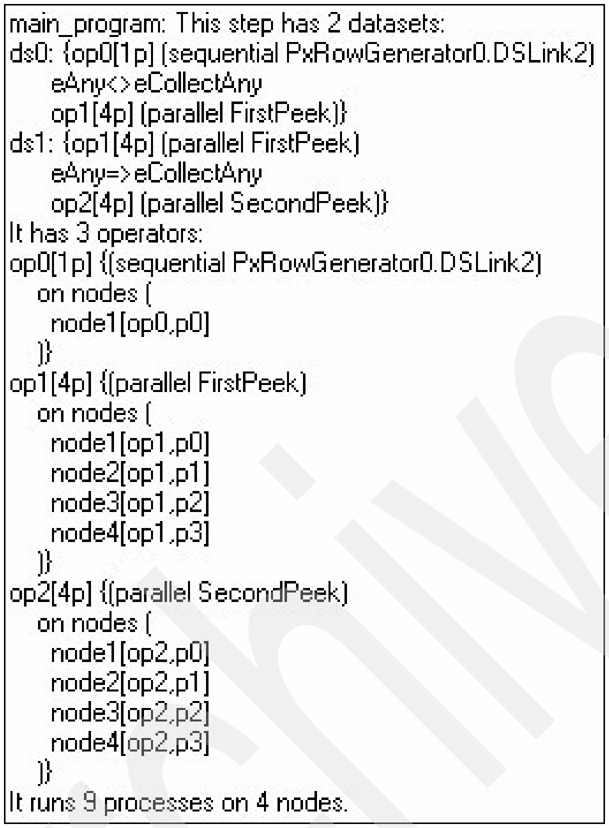
  Starts with the words `It has` *n* `operators:`

  The second section details actual operators created to execute the job flow. This includes:

  – Sequential or Parallel operation, and the degree of parallelism per operator.

  – Node assignment for each operator. The actual node names correspond to node names in the parallel configuration file. (in this example: “node1”, “node2”, “node3”, “node4”).

In Figure E-2, there are three operators, one running sequentially, two running in parallel across four nodes, for a total of nine operating system process.



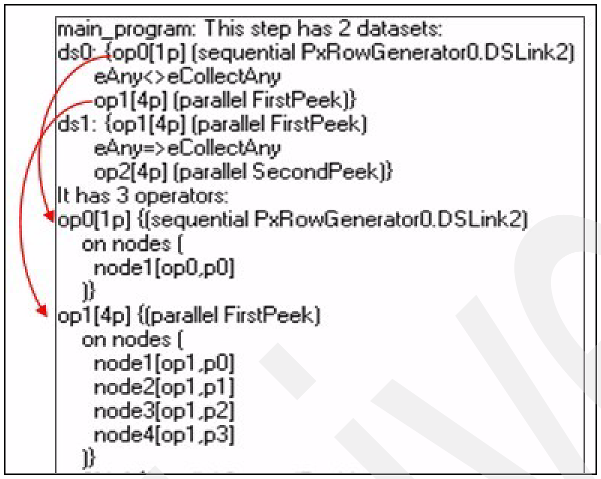
```
main_program: This step has 2 datasets:
ds0: {op0[1p] (sequential PxRowGenerator0.DSLink2)
      eAny<>eCollectAny
      op1[4p] (parallel FirstPeek)}
ds1: {op1[4p] (parallel FirstPeek)
      eAny=>eCollectAny
      op2[4p] (parallel SecondPeek)}
It has 3 operators:
op0[1p] {(sequential PxRowGenerator0.DSLink2)
    on nodes (
      node1[op0,p0]
    )}
op1[4p] {(parallel FirstPeek)
    on nodes (
      node1[op1,p0]
      node2[op1,p1]
      node3[op1,p2]
      node4[op1,p3]
    )}
op2[4p] {(parallel SecondPeek)
    on nodes (
      node1[op2,p0]
      node2[op2,p1]
      node3[op2,p2]
      node4[op2,p3]
    )}
It runs 9 processes on 4 nodes.
```

Figure E-2 Operators

The number of virtual datasets and the degree of parallelism determines the amount of memory used by the inter-operator transport buffers. The memory used by deadlock-prevention BufferOps can be calculated based on the number of inserted BufferOps.

E.2.1 Job Score: Datasets

The parallel pipeline architecture passes data from upstream producers to downstream consumers through in-memory virtual data sets. Figure E-3 depicts an example.



```
main_program: This step has 2 datasets:
ds0: {op0[1p] (sequential PxRowGenerator0.DSLink2)
   eAny<>eCollectAny
   op1[4p] (parallel FirstPeek)}
ds1: {op1[4p] (parallel FirstPeek)
   eAny=>eCollectAny
   op2[4p] (parallel SecondPeek)}
It has 3 operators:
op0[1p] {(sequential PxRowGenerator0.DSLink2)
  on nodes (
   node1[op0,p0]
  )}
op1[4p] {(parallel FirstPeek)
  on nodes (
   node1[op1,p0]
   node2[op1,p1]
   node3[op1,p2]
   node4[op1,p3]
  )}
```

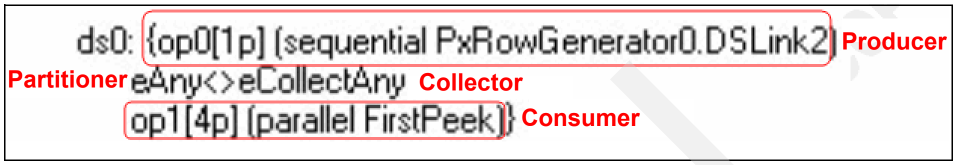
Figure E-3 Data sets and operators in job score

Datasets are identified in the first section of the parallel job score, with each dataset identified by its number (starting at zero). In this example, the first dataset is identified as “ds0”, and the next “ds1”.

Producers and consumers might be either persistent (on disk) datasets or parallel operators. Persistent datasets are identified by their dataset name. Operators are identified by their operator number and name (see the lines starting with "op0" and "op1"), , corresponding to the lower section of the job score.

The degree of parallelism is identified in brackets after the operator name. For example, operator zero (op0) is running sequentially, with one degree of parallelism [1p]. Operator 1 (op1) is running in parallel with four degrees of parallelism [4p].

In the dataset definition, the upstream producer is identified first, followed by a notation to indicate the type of partitioning or collecting (if any), followed by the downstream consumer. This is depicted in Figure E-4.



```
ds0: {op0[1p] (sequential PxRowGenerator0.DSLink2)
eAny<>eCollectAny
op1[4p] (parallel FirstPeek)}
```

Figure E-4 Producer and consumer

The notation between producer and consumer is used to report the type of partitioning or collecting (if any) that is applied. The partition type is associated with the first term, collector type with the second. The symbol between the partition name and collector name indicates the partition type and consumer. A list of the symbols and their description is shown in Table E-1.

Table E-1 Partition types and consumers

>	**Sequential producer to Sequential consumer**
<>	Sequential producer to Parallel consumer
=>	Parallel producer to Parallel consumer (SAME partitioning)
#>	Parallel producer to Parallel consumer (repartitioned; not SAME)
>>	Parallel producer to Sequential consumer
	No producer or no consumer (typically, for persistent datasets)

Finally, if the Preserve Partitioning flag has been set for a particular dataset, the notation “[pp]” appears in this section of the job score.

E.2.2 Job Score: Operators

In Figure E-5 we depict job score operators.

```
op0[1p] {(sequential APT_CombinedOperatorController:
     (Row_Generator_0)
     (inserted tsort operator {key={value=LastName},
   key={value=FirstName}})
    ) on nodes (
     node1[op0,p0]
    )}
op1[4p] {(parallel inserted tsort operator {key={value=LastName}
   key={value=FirstName}}(0))
    on nodes (
     node1[op2,p0]
     node2[op2,p1]
     node3[op2,p2]
     node4[op2,p3]
    )}
op2[4p] {(parallel buffer(0))
    on nodes (
     node1[op2,p0]
     node2[op2,p1]
     node3[op2,p2]
     node4[op2,p3]
    )}
```

Figure E-5 Job score operators

The lower portion of the parallel job score details the mapping between stages and actual processes generated at runtime. For each operator, this includes (as illustrated in the job score fragment) the following elements:

- Operator name (op*n*) numbered sequentially from zero (example “op0”)
- Degree of parallelism in brackets (example “[4p]”)
- Sequential or parallel execution mode
- Components of the operator, which have the following characteristics:
  - Typically correspond to the user-specified stage name in the Designer canvas
  - Can include combined operators (APT_CombinedOperatorController), which include logic from multiple stages in a single operator
  - Can include framework-inserted operators such as Buffers, Sorts
  - Can include composite operators (for example, Lookup)

Certain stages are composite operators. To the DataStage developer, a composite operator appears to be a single stage on the design canvas. But internally, a composite operator includes more than one function. This is depicted in Figure E-6.

```
op2[1p] {(parallel APT_LUTCreateImpl in Lookup_3)
    on nodes (
      ecc3671[op2,p0]
    )}
op3[4p] {(parallel buffer(0))
    on nodes (
      ecc3671[op3,p0]
      ecc3672[op3,p1]
      ecc3673[op3,p2]
      ecc3674[op3,p3]
    )}
op4[4p] {(parallel APT_CombinedOperatorController:
      (APT_LUTProcessImpl in Lookup_3)

        (APT_TransformOperatorImplV0S7_cpLookupTes
        t1_Transformer_7 in Transformer_7)
      (PeekNull)
    ) on nodes (
      ecc3671[op4,p0]
      ecc3672[op4,p1]
      ecc3673[op4,p2]
      ecc3674[op4,p3]
    )}
```

Figure E-6 Composite operators

For example, Lookup is a composite operator. It is composite of the following internal operators:

- APT_LUTCreateImpl

  This operator reads the reference data into memory

- APT_LUTProcessImpl

  This operator performs actual lookup processing after reference data has been loaded

At runtime, each individual component of a composite operator is represented as an individual operator in the job score, as shown in the following score fragment, as depicted in Figure E-6.

Using this information together with the output from the $APT_PM_SHOW_PIDS environment variable, you can evaluate the memory used by a lookup. Because the entire structure needs to be loaded before actual lookup processing can begin, you can also determine the delay associated with loading the lookup structure.

In a similar way, a persistent dataset (shown in Figure E-7) defined to overwrite an existing dataset of the same name has multiple entries in the job score to perform the following tasks:

- Delete Data Files
- Delete Descriptor File

```
main_program: This step has 2 datasets:
ds0: {op1[1p] (parallel delete data files in delete temp.ds)
      ->eCollectAny
      op2[1p] (sequential delete descriptor file in delete
temp.ds)}
ds1: {op0[1p] (sequential Row_Generator_0)
      ->
      temp.ds}
It has 3 operators:
op0[1p] {(sequential Row_Generator_0)
    on nodes (
      node1[op0,p0]
    )}
op1[1p] {(parallel delete data files in delete temp.ds)
    on nodes (
      node1[op1,p0]
    )}
op2[1p] {(sequential delete descriptor file in delete temp.ds)
    on nodes (
      node1[op2,p0]
    )}
It runs 3 processes on 1 node.
```

Figure E-7 Persistent dataset

F

Estimating the size of a parallel dataset

For the advanced user, this Appendix provides a more accurate and detailed way to estimate the size of a parallel dataset based on the internal storage requirements for each data type. We have listed the data types and their sizes in Table F-1 on page 412.

Table F-1 Data types and sizes

Data Type	Size
Integers	4 bytes
Small Integer	2 bytes
Tiny Integer	1 byte
Big Integer	8 bytes
Decimal	(precision+1)/2, rounded up
Float	8 bytes
VarChar(*n*)	*n* + 4 bytes for non-NLS data 2*n* + 4 bytes for NLS data (internally stored as UTF-16)
Char(*n*)	*n* bytes for non-NLS data *2n* bytes for NLS data
Time	4 bytes 8 bytes with microsecond resolution
Date	4 bytes
Timestamp	8 bytes 12 bytes with microsecond resolution

For the overall record width, calculate and add the following values:

- (# nullable fields)/8 for null indicators
- one byte per column for field alignment (worst case is 3.5 bytes per field)

Using the internal DataStage parallel C++ libraries, the method APT_Record::estimateFinalOutputSize() can give you an estimate for a given record schema. APT_Transfer::getTransferBufferSize() can do this as well, if you have a transfer that transfers all fields from input to output.

G

Environment variables reference

In this Appendix we summarize the environment variables mentioned throughout this document and have them listed in tables. These variables can be used on an as-needed basis to tune the performance of a particular job flow, to assist in debugging, or to change the default behavior of specific DataStage parallel stages. An extensive list of environment variables is documented in the *DataStage Parallel Job Advanced Developers Guide*.

The environment variable settings in this Appendix are only examples. Set values that are optimal to your environment.

Table G-1 Job design environment variables

Environment Variable	**Setting**	**Description**
$APT_STRING_PADCHAR	[char]	Overrides the default pad character of 0x0 (ASCII null) used when DS extends, or pads, a variable-length string field to a fixed length (or a fixed-length to a longer fixed-length).

Table G-2 Sequential File stage environment variables

Environment Variable	Setting	Description
$APT_EXPORT_FLUSH_ COUNT	[nrows]	Specifies how frequently (in rows) that the Sequential File stage (export operator) flushes its internal buffer to disk. Setting this value to a low number (such as 1) is useful for real-time applications, but there is a small performance penalty from increased I/O.
$APT_IMPORT_REJECT_ STRING_FIELD_OVERRUNS (DataStage v7.01 and later)		Setting this environment variable directs DataStage to reject Sequential File records with strings longer than their declared maximum column length. By default, imported string fields that exceed their maximum declared length are truncated.
$APT_IMPEXP_ALLOW_ ZERO_LENGTH_FIXED_ NULL	[set]	When set, allows zero length null_field value with fixed length fields. Use this with care as poorly formatted data causes incorrect results. By default, a zero length null_field value causes an error.
$APT_IMPORT_BUFFER_ SIZE $APT_EXPORT_BUFFER_ SIZE	[Kbytes]	Defines size of I/O buffer for Sequential File reads (imports) and writes (exports) respectively. Default is 128 (128K), with a minimum of 8. Increasing these values on heavily-loaded file servers can improve performance.
$APT_CONSISTENT_ BUFFERIO_SIZE	[bytes]	In certain disk array configurations, setting this variable to a value equal to the read/write size in bytes can improve performance of Sequential File import/export operations.
$APT_DELIMITED_READ _SIZE	[bytes]	Specifies the number of bytes the Sequential File (import) stage reads-ahead to get the next delimiter. The default is 500 bytes, but this can be set as low as 2 bytes. This setting must be set to a lower value when reading from streaming inputs (for example, socket or FIFO) to avoid blocking.
$APT_MAX_DELIMITED_ READ_SIZE	[bytes]	By default, Sequential File (import) reads ahead 500 bytes to get the next delimiter. If it is not found the importer looks ahead 4*500=2000 (1500 more) bytes, and so on (4X) up to 100,000 bytes. This variable controls the upper bound ,which is, by default, 100,000 bytes. When more than 500 bytes read-ahead is desired, use this variable instead of APT_DELIMITED_READ_SIZE.
$APT_IMPORT_PATTERN_ USES_FILESET	[set]	When this environment variable is set (present in the environment) file pattern reads are done in parallel by dynamically building a File Set header based on the list of files that match the given expression. For disk configurations with multiple controllers and disk, this significantly improves file pattern reads.

Table G-3 Data set environment variables

Environment Variable	Setting	Description
$APT_PHYSICAL_DATASET _BLOCK_SIZE	[bytes]	Specifies the size, in bytes, of the unit of data set I/O. Dataset segment files are written in chunks of this size. The default is 128 KB (131,072)
$APT_OLD_BOUNDED_ LENGTH	[set]	When this environment variable is set (present in the environment), Varchar columns is only stored using the actual data length. This might improve I/O performance (and reduce disk use) when processing a large number of varchar columns with a large maximum length and highly-variable data lengths.

Table G-4 DB2 environment variables

Environment Variable	Setting	Description
$INSTHOME	[path]	Specifies the DB2 install directory. This variable is usually set in a user's environment from `.db2profile`.
$APT_DB2INSTANCE_ HOME	[path]	Used as a backup for specifying the DB2 installation directory (if $INSTHOME is undefined).
$APT_DBNAME	[database]	Specifies the name of the DB2 database for DB2/UDB Enterprise stages if the Use Database Environment Variable option is **True**. If $APT_DBNAME is not defined, $DB2DBDFT is used to find the database name.
$APT_RDBMS_COMMIT_ ROWS Can also be specified with the "Row Commit Interval" stage input property.	[rows]	Specifies the number of records to insert between commits. The default value is 2000 per partition.
$DS_ENABLE_RESERVED_ CHAR_CONVERT		Allows DataStage plug-in stages to handle DB2 databases that use the special characters # and $ in column names.

Table G-5 Informix environment variables

Environment Variable	Setting	Description
$INFORMIXDIR	[path]	Specifies the Informix install directory.
$INFORMIXSQLHOSTS	[filepath]	Specifies the path to the Informix sqlhosts file.
$INFORMIXSERVER	[name]	Specifies the name of the Informix server matching an entry in the sqlhosts file.
$APT_COMMIT_INTERVAL	[rows]	Specifies the commit interval in rows for Informix HPL Loads. The default is 10000 per partiton.

Table G-6 Oracle environment variables

Environment Variable	Setting	Description
$ORACLE_HOME	[path]	Specifies installation directory for current Oracle instance. Normally set in a user's environment by Oracle scripts.
$ORACLE_SID	[sid]	Specifies the Oracle service name, corresponding to a TNSNAMES entry.
$APT_ORAUPSERT_ COMMIT_ROW_INTERVAL $APT_ORAUPSERT_ COMMIT_TIME_INTERVAL	[num] [seconds]	These two environment variables work together to specify how often target rows are committed for target Oracle stages with Upsert method. Commits are made whenever the time interval period has passed or the row interval is reached, whichever comes first. By default, commits are made every two seconds or 5000 rows per partition.
$APT_ORACLE_LOAD_ OPTIONS	[SQL* Loader options]	Specifies Oracle SQL*Loader options used in a target Oracle stage with Load method. By default, this is set to OPTIONS(DIRECT=TRUE, PARALLEL=TRUE)
$APT_ORACLE_LOAD_ DELIMITED *(DataStage 7.01 and later)*	[char]	Specifies a field delimiter for target Oracle stages using the Load method. Setting this variable makes it possible to load fields with trailing or leading blank characters.
$APT_ORA_IGNORE_ CONFIG_FILE_ PARALLELISM		When set, a target Oracle stage with Load method limits the number of players to the number of datafiles in the table's table space.
$APT_ORA_WRITE_FILES	[filepath]	Useful in debugging Oracle SQL*Loader issues. When set, the output of a Target Oracle stage with Load method is written to files instead of invoking the Oracle SQL*Loader. The filepath specified by this environment variable specifies the file with the SQL*Loader commands.
$DS_ENABLE_RESERVED _CHAR_CONVERT		Allows DataStage plug-in stages to handle Oracle databases that use the special characters # and $ in column names.

Table G-7 Teradata environment variables

Environment Variable	Setting	Description
$APT_TERA_SYNC_ DATABASE	[name]	Starting with V7, specifies the database used for the terasync table.
$APT_TERA_SYNC_USER	[user]	Starting with V7, specifies the user that creates and writes to the terasync table.
$APT_TER_SYNC_ PASSWORD	[password]	Specifies the password for the user identified by $APT_TERA_SYNC_USER.
$APT_TERA_64K_BUFFERS		Enables 64 K buffer transfers (32 K is the default). Might improve performance depending on network configuration.
$APT_TERA_NO_ERR_ CLEANUP		When set, this environment variable might assist in job debugging by preventing the removal of error tables and partially written target table. This environment variable is not recommended for general use.
$APT_TERA_NO_PERM_ CHECKS		Disables permission checking on Teradata system tables that must be readable during the TeraData Enterprise load process. This can be used to improve the startup time of the load.

Table G-8 Netezza environment variables

Environment Variable	Setting	Description
$NETEZZA	[path]	Specifies the Nezza home directory
$NZ_ODBC_INI_PATH	[filepath]	Points to the location of the .odbc.ini file. This is required for ODBC connectivity on UNIX systems.
$APT_DEBUG_MODULE_ NAMES	odbcstmt, odbcenv, nzetwriteop, nzutils, nzwriterep, nzetsubop	Prints debug messages from a specific DataStage module. Useful for debugging Netezza errors

Table G-9 Job monitoring environment variables

Environment Variable	Setting	Description
$APT_MONITOR_TIME	[seconds]	In V7 and later, specifies the time interval (in seconds) for generating job monitor information at runtime. To enable size-based job monitoring, unset this environment variable, and set $APT_MONITOR_SIZE.
$APT_MONITOR_SIZE	[rows]	Determines the minimum number of records the job monitor reports. The default of 5000 records is usually too small. To minimize the number of messages during large job runs, set this to a higher value (for example, 1000000).
$APT_NO_JOBMON		Disables job monitoring completely. In rare instances, this might improve performance. In general, this can only be set on a per-job basis when attempting to resolve performance bottlenecks.
$APT_RECORD_COUNTS		Prints record counts in the job log as each operator completes processing. The count is per operator per partition.

Table G-10 Performance-tuning environment variables

Environment Variable	Setting	Description
$APT_BUFFER_MAXIMUM_ MEMORY	41903040 (example)	Specifies the maximum amount of virtual memory in bytes used per buffer per partition. If not set, the default is 3 MB (3145728). Setting this value higher uses more memory, depending on the job flow, but might improve performance.
$APT_BUFFER_FREE_RUN	1000 (example)	Specifies how much of the available in-memory buffer to consume before the buffer offers resistance to any new data being written to it. If not set, the default is 0.5 (50% of $APT_BUFFER_MAXIMUM_MEMORY). If this value is greater than 1, the buffer operator reads $APT_BUFFER_FREE_RUN * $APT_BUFFER_MAXIMIMUM_MEMORY before offering resistance to new data. When this setting is greater than 1, buffer operators spool data to disk (by default scratch disk) after the $APT_BUFFER_MAXIMUM_MEMORY threshold. The maximum disk required is $APT_BUFFER_FREE_RUN * # of buffers * $APT_BUFFER_MAXIMUM_MEMORY
$APT_PERFORMANCE_ DATA	directory	Enables capture of detailed, per-process performance data in an XML file in the specified directory. Unset this environment variable to disable.
$TMPDIR	[path]	Defaults to /tmp. Used for miscellaneous internal temporary data including FIFO queues and Transformer temporary storage. As a minor optimization, might be best set to a filesystem outside of the DataStage install directory.

Table G-11 Job flow debugging environment variables

Environment Variable	Setting	Description
$OSH_PRINT_SCHEMAS		Outputs the actual schema definitions used by the DataStage parallel framework at runtime in the DataStage log. This can be useful when determining if the actual runtime schema matches the expected job design table definitions.
$APT_DISABLE_ COMBINATION The Advanced stage Properties editor in DataStage Designer v7.1 and later allows combination to be enabled and disabled for on a per-stage basis.		Disables operator combination for all stages in a job, forcing each parallel operator into a separate process. Though not normally needed in a job flow, this setting might help when debugging a job flow or investigating performance by isolating individual operators to separate processes. Note that disabling operator combination generates more UNIX processes, and requires more system resources (and memory). Disabling operator combination also disables internal optimizations for job efficiency and run-times.
$APT_PM_PLAYER_TIMING		Prints detailed information in the job log for each operator, including CPU use and elapsed processing time.
$APT_PM_PLAYER_ MEMORY		Prints detailed information in the job log for each operator when allocating additional heap memory.
$APT_BUFFERING_POLICY *Setting* $APT_BUFFERING_POLICY =FORCE *is not recommended for production job runs.*	FORCE	Forces an internal buffer operator to be placed between every operator. Normally, the DataStage framework inserts buffer operators into a job flow at runtime to avoid deadlocks and improve performance. Using $APT_BUFFERING_POLICY=FORCE in combination with $APT_BUFFER_FREE_RUN effectively isolates each operator from slowing upstream production. Using the job monitor performance statistics, this can identify which part of a job flow is impacting overall performance.
$DS_PX_DEBUG		Set this environment variable to capture copies of the job score, generated osh, and internal parallel framework log messages in a directory corresponding to the job name. This directory is created in the Debugging sub-directory of the Project home directory on the DataStage server.

$APT_PM_STARTUP_ CONCURRENCY		This environment variable should not normally need to be set. When trying to start large jobs on heavily-loaded servers, lowering this number limits the number of processes that are simultaneously created when a job is started.
$APT_PM_NODE_TIMEOUT	[seconds]	For heavily -loaded MPP or clustered environments, this variable determines the number of seconds the conductor node waits for a successful startup from each section leader. The default is 30 seconds.

H

DataStage data types

The DataStage Designer represents column data types using SQL notation. Each SQL data type maps to an underlying data type in the DS parallel framework. The internal parallel data types are used in schema files and are displayed when viewing generated OSH or viewing the output from $OSH_PRINT_SCHEMAS.

H.1 Parallel data types

In Table H-1 we summarize the underlying parallel data types of DataStage.

Table H-1 DataStage parallel data types

SQL Type	Internal Type	Size	Description
Date	date	4 bytes	Date with month, day, and year
Decimal, Numeric	decimal	(roundup(p)+ 1)/2	Packed decimal, compatible with IBM packed decimal format.
Float, Real	sfloat	4 bytes	IEEE single-precision (32-bit) floating point value
Double	dfloat	8 bytes	IEEE double-precision (64-bit) floating point value
TinyInt	int8, uint8	1 byte	Signed or unsigned integer of 8 bits *(Specify **unsigned** Extended option for unsigned)*
SmallInt	int16, uint16	2 bytes	Signed or unsigned integer of 16 bits (Specify **unsigned** Extended option for unsigned)
Integer	int32, unit32	4 bytes	Signed or unsigned integer of 32 bits (Specify **unsigned** Extended option for unsigned)
BigInt[a]	int64, unit64	8 bytes	Signed or unsigned integer of 64 bits (Specify **unsigned** Extended option for unsigned)
Binary, Bit, LongVarBinary, VarBinary	raw	1 byte per character	Untyped collection, consisting of a fixed or variable number of contiguous bytes and an optional alignment value
Unknown, Char, LongVarChar, VarChar	string	1 byte per character	ASCII character string of fixed or variable length *(**Unicode** Extended option NOT selected)*
NChar, NVarChar, LongNVarChar	ustring	multiple bytes per character	ASCII character string of fixed or variable length *(**Unicode** Extended option NOT selected)*
Char, LongVarChar, VarChar	ustring	multiple bytes per character	ASCII character string of fixed or variable length *(**Unicode** Extended option IS selected)*
Time	time	5 bytes	Time of day, with resolution to seconds
Time	time (microseconds)	5 bytes	Time of day, with resolution of microseconds *(Specify **microseconds** Extended option)*

Timestamp	timestamp	9 bytes	Single field containing both date and time value with resolution to seconds.
Timestamp	timestamp (microseconds)	9 bytes	Single field containing both date and time value with resolution to microseconds. *(Specify* ***microseconds*** *Extended option)*

a. BigInt values map to long long integers on all supported platforms except Tru64 where they map to longer integer values.

Strings and Ustrings

If NLS is enabled on your DataStage server, parallel jobs support two types of underlying character data types:

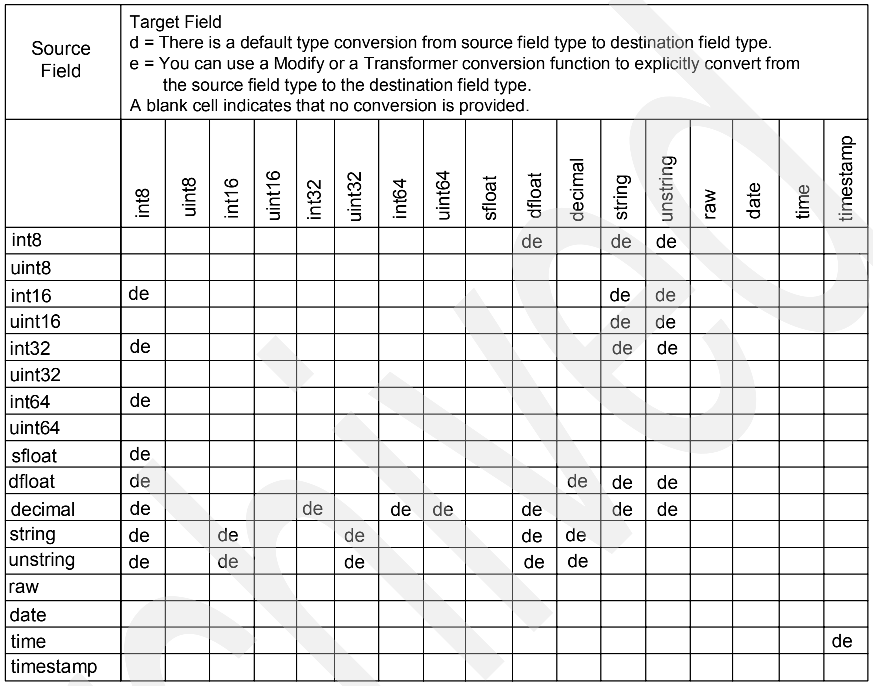
- Strings
- Ustrings

String data represents unmapped bytes, ustring data represents full Unicode (UTF-16) data.

The Char, VarChar, and LongVarChar SQL types relate to underlying string types where each character is 8-bits and does not require mapping because it represents an ASCII character. You can, however, specify that these data types are extended, in which case they are taken as ustrings and require mapping. (They are specified as such by selecting the “Extended” check box for the column in the Edit Meta Data dialog box.) An Extended field appears in the columns grid, and extended Char, VarChar, or LongVarChar columns have `Unicode` in this field. The NChar, NVarChar, and LongNVarChar types relate to underlying ustring types, so do not need to be explicitly extended.

Default and explicit type conversions

DataStage provides a number of default conversions and conversion functions when mapping from a source to a target parallel data type. Default type conversions take place across the stage output mappings of any parallel stage. Figure H-1 summarizes Data Type conversions.



Source Field	Target Field d = There is a default type conversion from source field type to destination field type. e = You can use a Modify or a Transformer conversion function to explicitly convert from the source field type to the destination field type. A blank cell indicates that no conversion is provided.																
	int8	uint8	int16	uint16	int32	uint32	int64	uint64	sfloat	dfloat	decimal	string	unstring	raw	date	time	timestamp
int8										de		de	de				
uint8																	
int16	de											de	de				
uint16												de	de				
int32	de											de	de				
uint32																	
int64	de																
uint64																	
sfloat	de																
dfloat	de										de	de	de				
decimal	de				de		de	de		de		de	de				
string	de		de			de				de	de						
unstring	de		de			de				de	de						
raw																	
date																	
time																	de
timestamp																	

Figure H-1 Data type conversions

The conversion of numeric data types can result in a loss of precision and cause incorrect results, depending on the source and result data types. In these instances, the parallel framework displays a warning message in the job log.

When converting from variable-length to fixed-length strings, the parallel framework pads the remaining length with NULL (ASCII zero) characters by default.

- The environment variable APT_STRING_PADCHAR can be used to change the default pad character from an ASCII NULL (0x0) to another character; for example, an ASCII space (0x20) or a Unicode space (U+0020). When entering a space for the value of APT_STRING_PADCHAR do note enclose the space character in quotes.
- As an alternate solution, the PadString Transformer function can be used to pad a variable-length (Varchar) string to a specified length using a specified pad character. Note that PadString does not work with fixed-length (CHAR) string types. You must first convert a Char string type to a Varchar type before using PadString.
- Certain stages (for example, Sequential File and DB2/UDB Enterprise targets) allow the pad character to be specified in their stage or column definition properties. When used in these stages, the specified pad character overrides the default for that stage only.

H.2 Null handling

The DataStage parallel framework represents nulls in two ways:

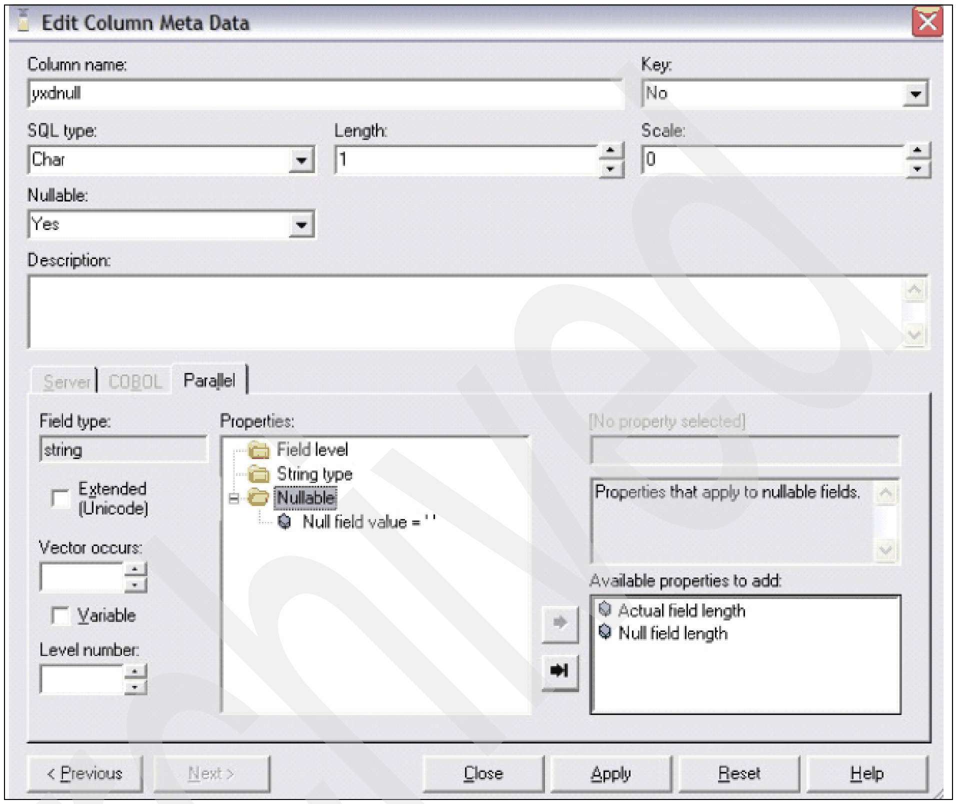
- It allocates a single bit to mark a field as null. This type of representation is called an *out-of-band null*.
- It designates a specific field value to indicate a null (for example a numeric field's most negative possible value). This type of representation is called an *in-band null*. In-band null representation can be disadvantageous because you must reserve a field value for nulls, and this value cannot be treated as valid data elsewhere.

The Transformer and Modify stages can change a null representation from an out-of-band null to an in-band null and from an in-band null to an out-of-band null.

Note: When processing nullable columns in a Transformer stage, care must be taken to avoid data rejects. See 9.1.1, "Transformer NULL handling and reject link" on page 140.

When reading from dataset and database sources with nullable columns, the DataStage parallel framework uses the internal, out-of-band null representation for NULL values.

When reading from or writing to Sequential Files or File Sets, the in-band (value) must be explicitly defined in the extended column attributes for each Nullable column, as shown in Figure H-2.


Edit Column Meta Data

Column name: yxdnull
Key: No
SQL type: Char
Length: 1
Scale: 0
Nullable: Yes
Description:

Server | COBOL | Parallel

Field type: string
Extended (Unicode)
Vector occurs:
Variable
Level number:

Properties:
- Field level
- String type
- Nullable
  - Null field value = ' '

[No property selected]

Properties that apply to nullable fields.

Available properties to add:
- Actual field length
- Null field length

< Previous | Next > | Close | Apply | Reset | Help

Figure H-2 Extended column metadata (Nullable properties)

The Table Definition of a stage's input or output data set can contain columns defined to support out-of-band nulls (Nullable attribute is checked). In Table H-2 we list the rules for handling nullable fields when a stage takes a dataset as input or writes to a dataset as output.

Table H-2 Table definitions

Source Field	Destination Field	Result
not Nullable	not Nullable	Source value propagates to destination.
Nullable	Nullable	Source value or null propagates.
not Nullable	Nullable	Source value propagates;. Destination value is never null.
Nullable	not Nullable	If the source value is not null, the source value propagates. If the source value is null, a fatal error occurs.

H.3 Runtime column propagation

Runtime column propagation (RCP) allows job designs to accommodate additional columns beyond those defined by the job developer. Before a DataStage developer can use RCP, it must be enabled at the project level through the administrator client.

Using RCP judiciously in a job design facilitates re-usable job designs based on input metadata, rather than using a large number of jobs with hard-coded table definitions to perform the same tasks. Certain stages, for example the Sequential File stage, allow their runtime schema to be parameterized, further extending re-use through RCP.

Furthermore, RCP facilitates re-use through parallel shared containers. Using RCP, only the columns explicitly referenced in the shared container logic need to be defined, the remaining columns pass through at runtime, as long as each stage in the shared container has RCP enabled on their stage Output properties.

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

For information about ordering these publications, see “How to get Redbooks” on page 432. Note that some of the documents referenced here might be available in softcopy only.

- *IBM WebSphere QualityStage Methodologies, Standardization, and Matching*, SG24-7546

Other publications

These publications are also relevant as further information sources:

- *IBM Information Server 8.1 Planning, Installation and Configuration Guide, GC19-1048*
- *IBM Information Server Introduction,* GC19-1049

Online resources

These Web sites are also relevant as further information sources:

- IBM Information Server information center

  `http://publib.boulder.ibm.com/infocenter/iisinfsv/v8r0/index.jsp`
- IBM Information Server Quick Start Guide

  `http://www-01.ibm.com/support/docview.wss?uid=swg27009391&aid=1`

How to get Redbooks

You can search for, view, or download Redbooks, Redpapers, Technotes, draft publications and Additional materials, as well as order hardcopy Redbooks publications, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

IBM

Redbooks

InfoSphere DataStage: Parallel Framework Standard Practices

(0.5” spine)
0.475”<->0.875”
250 <-> 459 pages

InfoSphere DataStage Parallel Framework Standard Practices

Develop highly efficient and scalable information integration applications

Investigate, design, and develop data flow jobs

Get guidelines for cost effective performance

In this IBM Redbooks publication, we present guidelines for the development of highly efficient and scalable information integration applications with InfoSphere DataStage (DS) parallel jobs.

InfoSphere DataStage is at the core of IBM Information Server, providing components that yield a high degree of freedom. For any particular problem there might be multiple solutions, which tend to be influenced by personal preferences, background, and previous experience. All too often, those solutions yield less than optimal, and non-scalable, implementations.

This book includes a comprehensive detailed description of the components available, and descriptions on how to use them to obtain scalable and efficient solutions, for both batch and real-time scenarios.

The advice provided in this document is the result of the combined proven experience from a number of expert practitioners in the field of high performance information integration, evolved over several years.

This book is intended for IT architects, Information Management specialists, and Information Integration specialists responsible for delivering cost-effective IBM InfoSphere DataStage performance on all platforms.

SG24-7830-00 0738434477