

# ORCHESTRATE

## VISUAL ORCHESTRATE USER'S GUIDE FOR ORCHESTRATE VERSION 4.5

TORRENT SYSTEMS, INC.



# ORCHESTRATE

VISUAL ORCHESTRATE

USER'S GUIDE

FOR ORCHESTRATE VERSION 4.5

This document, and the software described or referenced in it, are confidential and proprietary to Torrent Systems, Inc. They are provided under, and are subject to, the terms and conditions of a written license agreement between Torrent Systems and the licensee, and may not be transferred, disclosed, or otherwise provided to third parties, unless otherwise permitted by that agreement.

No portion of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Torrent Systems, Inc.

The specifications and other information contained in this document for some purposes may not be complete, current, or correct, and are subject to change without notice. The reader should consult Torrent Systems for more detailed and current information.

NO REPRESENTATION OR OTHER AFFIRMATION OF FACT CONTAINED IN THIS DOCUMENT, INCLUDING WITHOUT LIMITATION STATEMENTS REGARDING CAPACITY, PERFORMANCE, OR SUITABILITY FOR USE OF PRODUCTS OR SOFTWARE DESCRIBED HEREIN, SHALL BE DEEMED TO BE A WARRANTY BY TORRENT SYSTEMS FOR ANY PURPOSE OR GIVE RISE TO ANY LIABILITY OF TORRENT SYSTEMS WHATSOEVER. TORRENT SYSTEMS MAKES NO WARRANTY OF ANY KIND OR WITH REGARD TO THIS DOCUMENT OR THE INFORMATION CONTAINED IN IT, EVEN IF TORRENT SYSTEMS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Torrent is a registered trademark of Torrent Systems, Inc. Orchestrate and Orchestrate Hybrid Neural Network are trademarks of Torrent Systems, Inc.

AIX, DB2, SP2, Scalable POWERparallel Systems, and IBM are trademarks of IBM Corporation.

BYNET is a registered trademark of Teradata Corporation.

INFORMIX is a trademark of Informix Software, Inc.

Linux is a registered trademark of Linus Torvalds.

Oracle is a registered trademark of Oracle Corporation.

Sun and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc.

Teradata is a registered trademark of Teradata Corporation.

UNIX is a registered trademark of the Open Group.

Windows and Windows NT are U.S. registered trademarks of Microsoft Corporation.

The "X" device is a trademark of X/Open Company Ltd. in the UK and other countries.

All other product and brand names are trademarks or registered trademarks of their respective companies or organizations.

Copyright © 2000 Torrent Systems, Inc. All rights reserved. All patents pending.

Torrent Systems, Inc.  
Five Cambridge Center  
Cambridge, MA 02142  
617 354-8484  
617 354-6767 FAX

For technical support, send e-mail to: [tech-support@torrent.com](mailto:tech-support@torrent.com).

visorchug.4.5 11.00

# Table of Contents

## 1. Introduction to Orchestra

<b>Parallelism and Orchestra Applications</b> .....	1-1
<b>Introduction to Parallelism</b> .....	1-1
<b>Pipeline Parallelism</b> .....	1-2
<b>Partition Parallelism</b> .....	1-2
<b>Parallel-Processing Environments: SMP and Cluster/MPP</b> .....	1-3
<b>The Orchestra Configuration File</b> .....	1-3
<b>Orchestra Application Components</b> .....	1-4
<b>Data-Flow Modeling</b> .....	1-5
<b>Orchestra Data Sets</b> .....	1-5
<b>The Orchestra Schema</b> .....	1-6
<b>Virtual and Persistent Data Sets</b> .....	1-7
<b>Partitioning Data Sets</b> .....	1-9
<b>Orchestra Operators</b> .....	1-9
<b>Operator Execution</b> .....	1-10
<b>Prebuilt and Custom Operators</b> .....	1-10
<b>Orchestra Steps</b> .....	1-11
<b>The Orchestra Performance Monitor</b> .....	1-13
<b>Creating Orchestra Applications</b> .....	1-14
<b>Orchestra Installation and Administration</b> .....	1-14

## 2. Creating Applications with Visual Orchestra

<b>The Orchestra Development Environment</b> .....	2-1
<b>Creating an Orchestra Application</b> .....	2-3
<b>Deploying the Application on Your UNIX System</b> .....	2-6
<b>Deploying Your Application with <code>job-manager</code></b> .....	2-7
<b>Summary of Deployment Commands</b> .....	2-8
<b>Setting User Preferences</b> .....	2-9

<b>Setting Program Directory Paths</b> .....	2-13
<b>Visual Orchestrate Utilities</b> .....	2-14
<b>Checking an Orchestrate Configuration</b> .....	2-14
<b>Using the Orchestrate Shell</b> .....	2-15
<b>Generating an osh Script to Configure and Run a Program</b> .....	2-15
<b>Using the Lock Manager</b> .....	2-15
<b>3. Orchestrate Data Types</b>	
<b>Introduction to Orchestrate Data Types</b> .....	3-1
<b>Vectors</b> .....	3-2
<b>Support for Nullable Fields</b> .....	3-2
<b>Orchestrate Data Types in Detail</b> .....	3-2
<b>Date</b> .....	3-2
<b>Decimal</b> .....	3-4
<b>Floating-Point</b> .....	3-5
<b>Integers</b> .....	3-6
<b>Raw</b> .....	3-6
<b>String</b> .....	3-6
<b>Subrecord</b> .....	3-6
<b>Tagged</b> .....	3-6
<b>Time</b> .....	3-6
<b>Timestamp</b> .....	3-7
<b>Performing Data Type Conversions</b> .....	3-8
<b>Rules for Orchestrate Data Type Conversions</b> .....	3-8
<b>Summary of Orchestrate Data Type Conversions</b> .....	3-9
<b>Example of Default Type Conversion</b> .....	3-10
<b>Example of Type Conversion with modify</b> .....	3-10
<b>Data Type Conversion Errors</b> .....	3-11
<b>4. Orchestrate Data Sets</b>	
<b>Orchestrate Data Sets</b> .....	4-1
<b>Data Set Structure</b> .....	4-1
<b>Record Schemas</b> .....	4-2
<b>Using Data Sets with Operators</b> .....	4-3

<b>Using Virtual Data Sets</b> .....	4-4
<b>Using Persistent Data Sets</b> .....	4-6
<b>Importing Data into a Data Set</b> .....	4-7
<b>Partitioning a Data Set</b> .....	4-7
<b>Copying and Deleting Persistent Data Sets</b> .....	4-7
<b>Using Visual Orchestrate with Data Sets</b> .....	4-7
<b>Working with Persistent Data Sets</b> .....	4-8
<b>Working with Virtual Data Sets</b> .....	4-12
<b>Using the Data Set Viewer</b> .....	4-14
<b>Obtaining the Record Count from a Persistent Data Set</b> .....	4-16
<b>Defining a Record Schema</b> .....	4-16
<b>Schema Definition Files</b> .....	4-17
<b>Field Accessors</b> .....	4-17
<b>How a Data Set Acquires Its Record Schema</b> .....	4-17
<b>Using Complete or Partial Schema Definitions</b> .....	4-18
<b>Naming Record Fields</b> .....	4-19
<b>Defining Field Nullability</b> .....	4-19
<b>Using Value Data Types in Schema Definitions</b> .....	4-20
<b>Vectors and Aggregates in Schema Definitions</b> .....	4-24
<b>Default Values for Fields in Output Data Sets</b> .....	4-27
<b>Using the Visual Orchestrate Schema Editor</b> .....	4-27
<b>Representation of Disk Data Sets</b> .....	4-32
<b>Setting the Data Set Version Format</b> .....	4-33
<b>Data Set Files</b> .....	4-34
<b>5. Orchestrate Operators</b>	
<b>Operator Overview</b> .....	5-1
<b>Operator Execution Modes</b> .....	5-2
<b>Persistent Data Sets and Steps</b> .....	5-2
<b>Using Visual Orchestrate with Operators</b> .....	5-3
<b>Operator Interface Schemas</b> .....	5-6
<b>Example of Input and Output Interface Schema</b> .....	5-6
<b>Input Data Sets and Operators</b> .....	5-7
<b>Output Data Sets and Operators</b> .....	5-8

<b>Operator Interface Schema Summary</b> .....	5-10
<b>Record Transfers and Schema Variables</b> .....	5-11
<b>Flexibly Defined Interface Fields</b> .....	5-15
<b>Using Operators with Data Sets That Have Partial Schemas</b> .....	5-15
<b>Data Set and Operator Data Type Compatibility</b> .....	5-17
<b>Data Type Conversion Errors and Warnings</b> .....	5-17
<b>String and Numeric Data Type Compatibility</b> .....	5-18
<b>Decimal Compatibility</b> .....	5-19
<b>Date, Time, and Timestamp Compatibility</b> .....	5-20
<b>Vector Data Type Compatibility</b> .....	5-21
<b>Aggregate Field Compatibility</b> .....	5-21
<b>Null Compatibility</b> .....	5-21
<b>6. Orchestrate Steps</b>	
<b>Using Steps in Your Application</b> .....	6-1
<b>The Flow of Data in a Step</b> .....	6-2
<b>Designing a Single-Step Application</b> .....	6-3
<b>Designing a Multiple-Step Application</b> .....	6-4
<b>Working with Steps in Visual Orchestrate</b> .....	6-4
<b>Creating Steps</b> .....	6-5
<b>Executing a Step</b> .....	6-7
<b>Setting Server Properties for a Step</b> .....	6-8
<b>Setting Environment Variables</b> .....	6-10
<b>Setting Step Execution Modes</b> .....	6-10
<b>Using Pre and Post Scripts</b> .....	6-12
<b>7. The Performance Monitor</b>	
<b>The Performance Monitor Window</b> .....	7-1
<b>How the Performance Monitor Represents Your Program Steps</b> .....	7-3
<b>Configuring the Performance Monitor</b> .....	7-4
<b>Controlling the Performance Monitor Display</b> .....	7-5
<b>General Display Control</b> .....	7-5
<b>Operator Display Control</b> .....	7-6
<b>Data Set Display Control</b> .....	7-7



<b>Generating a Results Spreadsheet</b> .....	<b>7-8</b>
<b>Creating Movie Files</b> .....	<b>7-10</b>
<b>8. Partitioning in Orchestrate</b>	
<b>Partitioning Data Sets</b> .....	<b>8-1</b>
<b>Partitioning and a Single-Input Operator</b> .....	<b>8-2</b>
<b>Partitioning and a Multiple-Input Operator</b> .....	<b>8-2</b>
<b>Partitioning Methods</b> .....	<b>8-3</b>
<b>The Benefit of Similar-Size Partitions</b> .....	<b>8-3</b>
<b>Partitioning Method Overview</b> .....	<b>8-4</b>
<b>Partitioning Method Examples</b> .....	<b>8-5</b>
<b>Using the Partitioning Operators</b> .....	<b>8-7</b>
<b>Choosing a Partitioning Operator</b> .....	<b>8-8</b>
<b>The Preserve-Partitioning Flag</b> .....	<b>8-11</b>
<b>Example of the Preserve-Partitioning Flag's Effect</b> .....	<b>8-11</b>
<b>Preserve-Partitioning Flag with Sequential Operators</b> .....	<b>8-13</b>
<b>Manipulating the Preserve-Partitioning Flag</b> .....	<b>8-13</b>
<b>Example: Using the Preserve-Partitioning Flag</b> .....	<b>8-14</b>
<b>9. Collectors in Orchestrate</b>	
<b>Sequential Operators and Collectors</b> .....	<b>9-1</b>
<b>Sequential Operators and the Preserve-Partitioning Flag</b> .....	<b>9-2</b>
<b>Collection Methods</b> .....	<b>9-3</b>
<b>Choosing a Collection Method</b> .....	<b>9-3</b>
<b>Setting a Collection Method</b> .....	<b>9-4</b>
<b>Collection Operator and Sequential Operator with Any Method</b> .....	<b>9-5</b>
<b>Collection Operator before Write to Persistent Data Set</b> .....	<b>9-5</b>
<b>10. Constraints</b>	
<b>Using Constraints</b> .....	<b>10-1</b>
<b>Controlling Where Your Code Executes on a Parallel System</b> .....	<b>10-2</b>
<b>Controlling Where Your Data Is Stored</b> .....	<b>10-4</b>
<b>Using Constraints with Operators and Steps</b> .....	<b>10-5</b>
<b>Configuring Orchestrate Logical Nodes</b> .....	<b>10-5</b>
<b>Using Node Pool Constraints</b> .....	<b>10-6</b>

<b>Using Resource Constraints</b> .....	10-7
<b>Combining Node and Resource Constraints</b> .....	10-8
<b>Using Node Maps</b> .....	10-8
<b>Data Set Constraints</b> .....	10-9
<b>11. Run-Time Error and Warning Messages</b>	
<b>How Orchestrate Detects and Reports Errors</b> .....	11-1
<b>Error and Warning Message Format</b> .....	11-2
<b>Messages from Subprocesses</b> .....	11-3
<b>Controlling the Format of Message Display</b> .....	11-4
<b>12. Creating Custom Operators</b>	
<b>Custom Orchestrate Operators</b> .....	12-1
<b>Kinds of Operators You Can Create</b> .....	12-2
<b>How a Generated Operator Processes Data</b> .....	12-3
<b>Configuring Orchestrate For Creating Operators</b> .....	12-4
<b>Using Visual Orchestrate to Create an Operator</b> .....	12-5
<b>How Your Code Is Executed</b> .....	12-8
<b>Specifying Operator Input and Output Interfaces</b> .....	12-8
<b>Adding and Editing Definitions of Input and Output Ports</b> .....	12-8
<b>Reordering the Input Ports or Output Ports</b> .....	12-10
<b>Deleting an Input or Output Port</b> .....	12-10
<b>Specifying the Interface Schema</b> .....	12-10
<b>Defining Transfers</b> .....	12-13
<b>Referencing Operator Interface Fields in Operator Code</b> .....	12-13
<b>Examples of Custom Operators</b> .....	12-14
<b>Convention for Property Settings in Examples</b> .....	12-14
<b>Example: Sum Operator</b> .....	12-15
<b>Example: Sum Operator Using a Transfer</b> .....	12-16
<b>Example: Operator That Recodes a Field</b> .....	12-17
<b>Example: Adding a User-Settable Option to the Recoding Operator</b> .....	12-17
<b>Using Orchestrate Data Types in Your Operator</b> .....	12-20
<b>Using Numeric Fields</b> .....	12-21
<b>Using Date, Time, and Timestamp Fields</b> .....	12-21

<b>Using Decimal Fields</b> .....	12-23
<b>Using String Fields</b> .....	12-24
<b>Using Raw Fields</b> .....	12-25
<b>Using Nullable Fields</b> .....	12-25
<b>Using Vector Fields</b> .....	12-26
<b>Using the Custom Operator Macros</b> .....	12-27
<b>Informational Macros</b> .....	12-27
<b>Flow-Control Macros</b> .....	12-27
<b>Input and Output Macros</b> .....	12-28
<b>Transfer Macros</b> .....	12-29
<b>How Visual Orchestra Executes Generated Code</b> .....	12-31
<b>Designing Operators with Multiple Inputs</b> .....	12-31
<b>Requirements for Coding for Multiple Inputs</b> .....	12-32
<b>Strategies for Using Multiple Inputs and Outputs</b> .....	12-32
<b>13. Creating UNIX Operators</b>	
<b>Introduction to UNIX Command Operators</b> .....	13-1
<b>Characteristics of a UNIX Command Operator</b> .....	13-2
<b>UNIX Shell Commands</b> .....	13-3
<b>Execution of a UNIX Command Operator</b> .....	13-5
<b>Handling Operator Inputs and Outputs</b> .....	13-7
<b>Using Data Sets for Inputs and Outputs</b> .....	13-8
<b>Example: Operator Using Standard Input and Output</b> .....	13-9
<b>Example: Operator Using Files for Input and Output</b> .....	13-13
<b>Example: Specifying Input and Output Record Schemas</b> .....	13-19
<b>Passing Arguments to and Configuring UNIX Commands</b> .....	13-22
<b>Using a Shell Script to Call the UNIX Command</b> .....	13-22
<b>Handling Message and Information Output Files</b> .....	13-24
<b>Handling Configuration and Parameter Input Files</b> .....	13-25
<b>Using Environment Variables to Configure UNIX Commands</b> .....	13-26
<b>Example: Passing File Names Using Environment Variables</b> .....	13-26
<b>Example: Defining an Environment Variable for a UNIX Command</b> .....	13-27
<b>Example: Defining User-Settable Options for a UNIX Command</b> .....	13-28
<b>Handling Command Exit Codes</b> .....	13-35

- How Orchestrate Optimizes Command Operators ..... 13-36**
  - Cascading UNIX Command Operators ..... 13-37**
  - Using Files as Inputs to UNIX Command Operators ..... 13-38**
  - Using FileSets as Command Operator Inputs and Outputs ..... 13-39**
  - Using Partial Record Schemas ..... 13-39**

**Index**

# 1: Introduction to Orchestra

With the Orchestra Development Environment, you create parallel applications without becoming bogged down in the low-level issues usually associated with parallel programming. Orchestra allows you to develop parallel applications using standard sequential programming models, while Orchestra handles the underlying parallelism.

Orchestra is designed to handle record-based data, much like the data stored in an RDBMS such as DB2, INFORMIX, Teradata, or Oracle. In fact, Orchestra can read data directly from an RDBMS for parallel processing and then store its results in the RDBMS for further analysis.

Orchestra provides a graphical user interface, Visual Orchestra, to enable you to create a complete parallel application in a Microsoft Windows development environment.

This chapter introduces the fundamental capabilities of Orchestra, in the following sections:

- “Parallelism and Orchestra Applications” on page 1-1
- “Orchestra Application Components” on page 1-4
- “Orchestra Data Sets” on page 1-5
- “Orchestra Operators” on page 1-9
- “Orchestra Steps” on page 1-11
- “Creating Orchestra Applications” on page 1-14

The next chapter describes in more depth how to use Visual Orchestra to create parallel applications. The rest of this book explains in detail how to use the three Orchestra application components: data sets, operators, and steps.

## Parallelism and Orchestra Applications

This section first describes the two basic kinds of parallelism that can be used in Orchestra applications. This section then describes the main categories of parallel-processing environments in which Orchestra applications can be run. The section concludes with a description of the Orchestra configuration file.

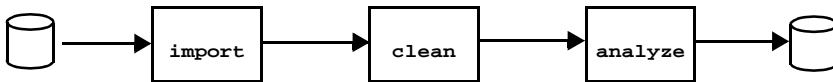
### Introduction to Parallelism

There are two basic kinds of parallelism, both of which you can use in your Orchestra applications:

- Pipeline parallelism
- Partition parallelism

## Pipeline Parallelism

In *pipeline parallelism*, each operation runs when it has input data available to process, and all processes are running simultaneously, except at the beginning of the job as the pipeline fills, and at the end as it empties. In a sequential application, operations execute strictly in sequence. The following figure depicts a sample application that imports data, then performs a clean operation on the data (perhaps removing duplicate records), and then performs some kind of analysis:



Use of Orchestrate lets an application concurrently run each operation in a separate operating-system process, using shared memory to pass data among the processes. Each operation runs when it has input data available to process.

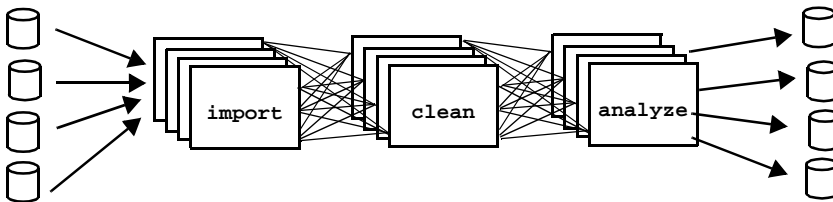
The theoretical limit to the efficiency gain of the pipeline is a factor of the number of operations that your application uses. This gain in efficiency is achievable independently of and in addition to partition parallelism, described below.

## Partition Parallelism

The more powerful kind of parallelism relies on data partitions. *Partition parallelism* distributes an operation over multiple processing nodes in the system, allowing multiple CPUs to work simultaneously on one operation.

Partitioning divides a data set into multiple *partitions* on the processing nodes of your system. Partitioning implements the “divide and conquer” aspect of parallel processing. Because each node in the parallel system processes a partition of a data set rather than all of it, your system can produce much higher throughput than with a single-processor system.

The following figure is a data-flow diagram for the same application, as executed in parallel on four processing nodes.



With enough processors, the model shown above can use both pipeline and partition parallelism, further improving performance.

## Parallel-Processing Environments: SMP and Cluster/MPP

The environment in which you run your Orchestra applications is defined by your system's architecture and hardware resources. All parallel-processing environments are categorized as one of the following:

- SMP (symmetric multiprocessing), in which some hardware resources may be shared among processors
- Cluster or MPP (massively parallel processing), also known as *shared-nothing*, in which each processor has exclusive access to hardware resources

SMP systems allow you to scale up the number of CPUs, which may improve application performance. The performance improvement depends on whether your application is CPU-, memory-, or I/O-limited. In CPU-limited applications, the memory, memory bus, and disk I/O spend a disproportionate amount of time waiting for the CPU to finish its work. Running a CPU-limited application on more processing units can shorten the waiting time of other resources and thereby speed up overall performance.

Some SMP systems allow scalability of disk I/O, so that throughput improves as the number of processors increases. A number of factors contribute to the I/O scalability of an SMP, including the number of disk spindles, the presence or absence of RAID, and the number of I/O controllers

In a cluster or MPP environment, you can use the multiple CPUs and their associated memory and disk resources in concert to tackle a single application. In this environment, each CPU has its own dedicated memory, memory bus, disk, and disk access. In a shared-nothing environment, parallelization of your application is likely to improve the performance of CPU-limited, memory-limited, or disk I/O-limited applications.

## The Orchestra Configuration File

Every MPP or SMP environment has characteristics that define the system overall as well as the individual processing nodes. These characteristics include node names, disk storage locations, and other distinguishing attributes. For example, certain processing nodes might have a direct connection to a mainframe for performing high-speed data transfers, while other nodes have access to a tape drive, and still others are dedicated to running an RDBMS application.

To optimize Orchestra for your system, you edit and modify the Orchestra *configuration file*. The configuration file describes every processing node that Orchestra will use to run your application. When you invoke an Orchestra application, Orchestra first reads the configuration file to determine the available system resources.

When you modify your system by adding or removing processing nodes or by reconfiguring nodes, you do not need to recode or even to recompile your Orchestra application. Instead, you need only edit the configuration file.

Another benefit of the configuration file is the control it gives you over parallelization of your application during the development cycle. For example, by editing the configuration file, you can first execute your application on a single processing node, then on two nodes, then four, then eight, and so forth. The configuration file lets you measure system performance and scalability without modifying your application code.

For complete information on configuration files, see the *Orchestrate Installation and Administration Manual*.

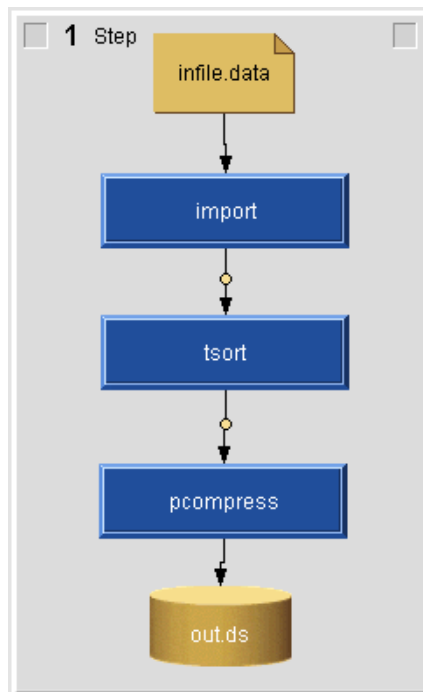
## Orchestrate Application Components

You create Orchestrate applications with three basic components:

- Data sets: Sets of data processed by the Orchestrate application
- Operators: Basic functional units of an Orchestrate application
- Steps: Groups of Orchestrate operators that process the application's data

The Visual Orchestrate graphical user interface lets you easily create and run an application. For instructions on using Visual Orchestrate, see the chapter “Creating Applications with Visual Orchestrate”.

The following figure shows a Visual Orchestrate Program window with a sample application, demonstrating all three Orchestrate components—data sets, operators, and steps:

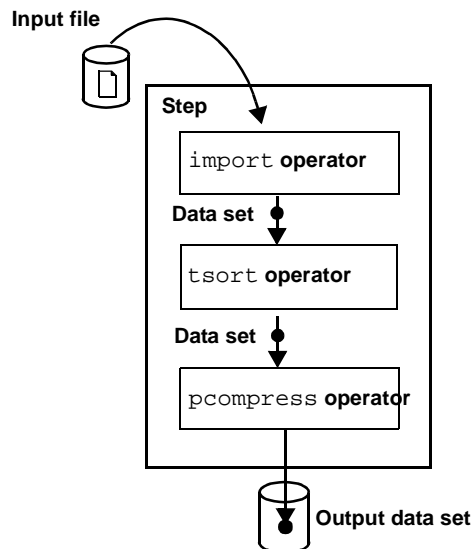




This sample application consists of one step. The step first uses the Orchestrate `import` operator to create an input data set from external data in file `inFile.data`. It then pipes the imported data to the `tsort` operator, which performs a sort. The application then compresses the data using the `pcompress` operator. Note that the sort and compress operations are performed in parallel. After data has been sorted and compressed, the application stores it to disk as an output data set named `out.ds`. (Note that output data can also be exported to flat files.)

## Data-Flow Modeling

A *data-flow model* can help you plan and analyze your Orchestrate application. The data-flow model lets you conveniently represent I/O behavior and the operations performed on the data. The following data-flow diagram models this sample application:



The following sections describe data sets, operators, and steps in more detail.

## Orchestrate Data Sets

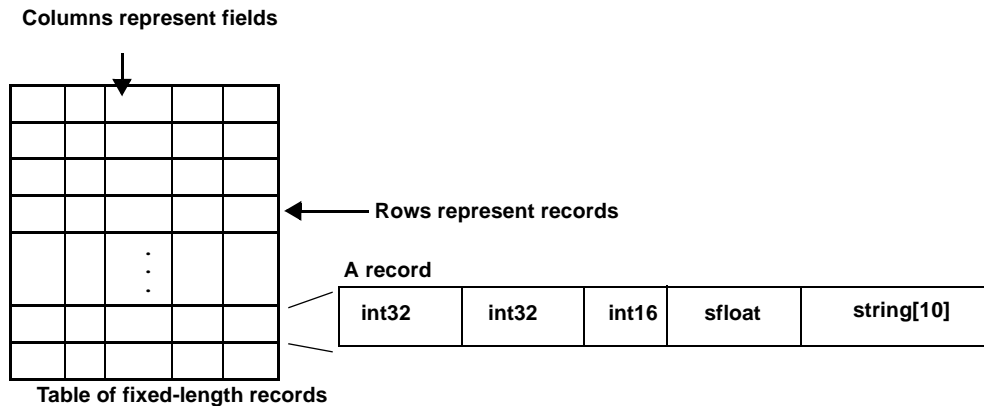
Orchestrate applications process three basic kinds of data, all of which have a structured format:

- Data stored in flat files: A file can contain data stored as newline-delimited records, fixed-length or variable-length records, binary streams, or a custom format. You use an Orchestrate *schema* to describe the layout of imported data (described in the section “The Orchestrate Schema” on page 1-6).
- RDBMS tables: Orchestrate supports direct access of Oracle, DB2, Teradata, and INFORMIX tables for both reading and writing.
- Orchestrate data sets

A *data set* is the body of data that is input to or output from an Orchestra application. Orchestra processes record-based data in parallel, so the data set that is input to an Orchestra application always has a *record* format. Orchestra's record-based processing is similar to the processing performed by an RDBMS (relational database management system), such as DB2, Oracle, Teradata, or INFORMIX.

Record-based data is structured as rows, each of which represents a record. Records are further divided into *fields*, where a field is defined by a field identifier, or *name*, and a field *data type*.

As an example, the following figure shows a record-based data set:



In the figure above, a data set is represented by a table with multiple rows, representing records. Each record has five data fields, represented by columns. All fields are of fixed length, so that the records are all the same length.

A record format can also include one or more variable-length fields, so that the record is also of variable length. A variable-length field indicates its length either by marking the end of the field with a delimiter character or by including information indicating the length.

## The Orchestra Schema

In Orchestra, the record structure of a data set is defined by an Orchestra record *schema*, which is a form of metadata. Many data processing applications include metadata support. For example, an RDBMS uses metadata to define the layout of a database table, including the name, data type, and other attributes of every record field. Also, COBOL programs can contain an FD (File Description) section to describe the layout of a COBOL data file. For details on Orchestra schema capabilities and usage, see the section “Defining a Record Schema” on page 4-16.

An Orchestra record schema allows you to reference an individual field by its name, without knowing the field's exact location within the record. You use the Orchestra data definition language to define a schema for a data set only once. The following is a sample Orchestra record schema:

```
record (  
  a:int32;  
  b:int32;  
  c:int16;  
  d:sfloat;  
  e:string[10] )
```

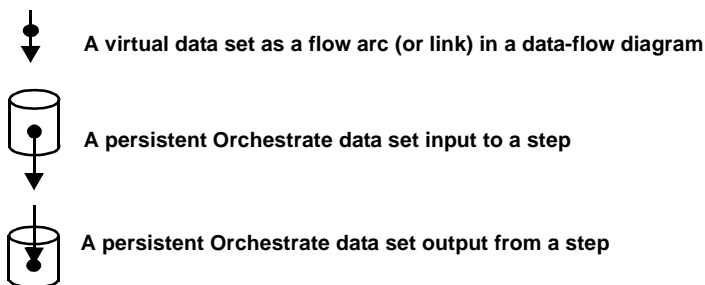
An Orchestrate record definition consists of the keyword `record`, followed by a parenthesized list of semicolon-separated field definitions. You can optionally include a terminating semicolon after the last field definition.

Each field definition consists of the field's name, a colon, and the field's data type. For a variable-length data type, such as a string, you can include an optional length specifier; in the example above, string `e` is specified to have the length 10.

A central feature of the Orchestrate record schema facility is *flexibility*. With schemas, you can create a range of record layouts as well as support existing database table standards. You can use schemas to represent existing data formats for RDBMS data tables, COBOL data files, and UNIX data files. Schema flexibility is also very advantageous when you read external data into an Orchestrate application (*import*), or write Orchestrate data to an external format (*export*).

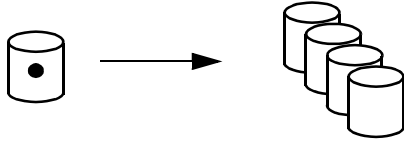
## Virtual and Persistent Data Sets

As a data is passed from one operator to another in a step, Orchestrate handles it as a *virtual* data set, which exists only during the step's processing. A data set input to or output from a step must be *persistent*, or saved to disk. In data-flow diagrams, Orchestrate data sets are represented by the following symbols.



The virtual data set and data set output symbols are shown in the data-flow diagram example in the preceding section.

Data set icons can also show the physical storage of the data set, in files on multiple disks in your system. In the following figure, to the right of the arrow an Orchestra data set is shown as files stored on four separate disks:



## Required Naming Convention for Data Sets

For Orchestra to correctly process persistent data sets, their file names must have the extension `.ds`. For example, `inData.ds` is a valid name for a persistent data set.

In some steps, such as those with branches (see the chapter “Orchestra Steps”), it is necessary to name virtual data sets. For Orchestra to correctly process named virtual data sets, they must be named with the extension `.v`. For example, `tempData.v` is a valid name for a virtual data set.

## Data in Flat Files

Orchestra can read and write data from a flat file (sometimes referred to as a UNIX file), represented by the following symbol:



A flat file

In reading from and writing to flat files, Orchestra performs implicit import and export operations.

## Data in RDBMS Tables

Orchestra can also read and write an RDBMS table from DB2, Oracle, Teradata, or INFORMIX. In an Orchestra data-flow diagram, an RDBMS table is represented by the following symbol:



An RDBMS table

When it reads an RDBMS table, Orchestra translates the table into an Orchestra data set. When it writes a data set to an RDBMS, Orchestra translates the data set to the table format of the destination RDBMS. See the *Orchestra User's Guide: Operators* for information on reading and writing tables.

Managing a data set distributed over an MPP that may contain hundreds of individual processing nodes, disk drives, and data files is a complex task. However, Orchestra handles all the underlying communications necessary to route each record of a data set to the appropriate node for processing, even if the data set represents an RDBMS table. When you design and create an Orchestra application, you do not need to be concerned with the location of individual data set records or the means by which records will be transmitted to processing nodes.

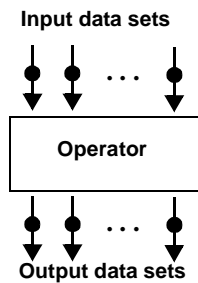
For more information about data sets, see the chapter “Orchestrate Data Sets”.

## Partitioning Data Sets

The benefits of partitioning your data sets were introduced in the section “Partition Parallelism” on page 1-2. Orchestrate allows you to control how your data is partitioned. For example, you may want to partition your data in a particular way to perform an operation such as a sort. On the other hand, you may have an application that partitions data solely to optimize the speed of your application. See the chapter “Partitioning in Orchestrate” for more information.

## Orchestrate Operators

Orchestrate operators, which process or analyze data, are the basic functional units of an Orchestrate application. An operator can take data sets, RDBMS tables, or data files as input, and can produce data sets, RDBMS tables, or data files as output. The following figure represents an Orchestrate operator in a data-flow diagram:



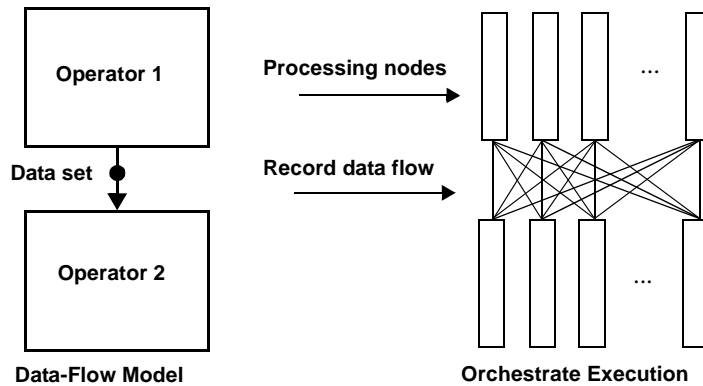
The operators in your Orchestrate application pass data records from one operator to the next, in pipeline fashion. For example, the operators in an application step might start with an import operator, which reads data from a file and converts it to an Orchestrate data set. Subsequent operators in the sequence could perform various processing and analysis tasks. In the section “Data-Flow Modeling” on page 1-5, you saw a more detailed data-flow diagram of such an Orchestrate application.

The processing power of Orchestrate derives largely from its ability to execute operators in parallel on multiple processing nodes. You will likely use parallel operators for most processing in your Orchestrate applications. Orchestrate also supports sequential operators, which execute on a single processing node. Orchestrate provides libraries of general-purpose operators, and it also lets you create custom operators (see the section “Prebuilt and Custom Operators” on page 1-10).

## Operator Execution

By default, Orchestrate operators execute on all processing nodes in your system. Orchestrate dynamically scales your application up or down in response to system configuration changes, without requiring you to modify your application. This capability means that if you develop parallel applications for a small system and later increase your system's processing power, Orchestrate will automatically scale up those applications to take advantage of your new system configuration.

The following figure shows two Orchestrate operators connected by a single data set:



The left side of this figure shows the operators in an Orchestrate data-flow model. The right side of the figure shows the operators as executed by Orchestrate. Records from any node that executes Operator 1 may be processed by any node that executes Operator 2. Orchestrate coordinates the multiple nodes that execute one operator, and Orchestrate also manages the data flow among nodes executing different operators.

Orchestrate allows you to limit, or *constrain*, execution of an operator to particular nodes on your system. For example, an operator may use system resources, such as a tape drive, not available to all nodes. Another case is a memory-intensive operation, which you want to run only on nodes with ample memory.

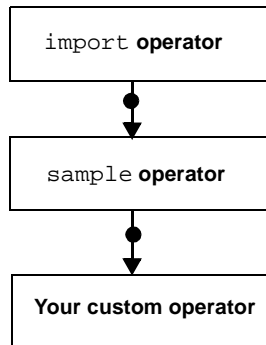
## Prebuilt and Custom Operators

Orchestrate supplies libraries of operators that perform general-purpose tasks in parallel, including the following:

- Import and export data
- Copy, merge, sort, and split data sets
- Summarize, encode, and calculate statistics on a data set
- Perform data mining operations using the Orchestrate analytic tools

See the *Orchestrate User's Guide: Operators* for information on these prebuilt operators.

In addition to the Orchestrate operators, your application may require other operators for specific data-processing tasks. Orchestrate allows you to develop custom operators and execute them in parallel or sequentially, as you execute the prebuilt operators. For example, the step shown below first processes the data with two Orchestrate operators, import and sample. Then, it passes the data to a custom operator that you have created:



You can create custom operators in the following three ways:

- Create an operator from UNIX commands or utilities, such as `grep` or `awk`. Visual Orchestrate lets you conveniently create UNIX operators with the UNIX Command Custom Operator (UNIX Command) feature; see the chapter “Creating UNIX Operators” for details.
- Create an operator from a few lines of your C or C++ code with the Custom Operator (Native) feature. For details on using this feature to conveniently implement logic specific to your application, see the chapter “Creating Custom Operators”.

You can also use the `cbuildop` command utility to create operators from your own C functions; see the chapter “Building Operators in C” in the *Orchestrate Shell User's Guide*.

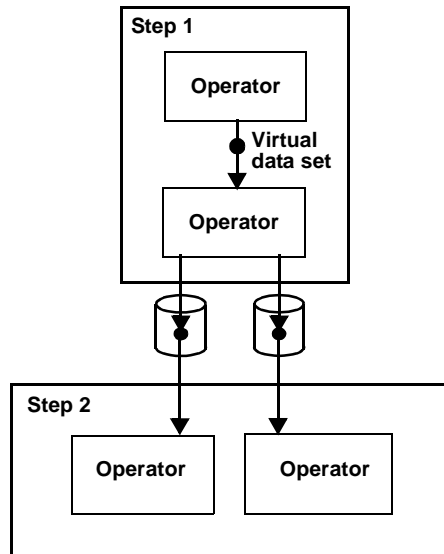
- Derive an operator from the Orchestrate C++ class library. The operator can execute in parallel or sequentially. See the *Orchestrate/APT Developer's Guide* for more information.

## Orchestrate Steps

An Orchestrate application consists of at least one step, in which one or more Orchestrate operators process the application's data. A step is a data flow, with its input consisting of data files, RDBMS data, or persistent data sets. As output, a step produces data files, RDBMS data, or persistent data sets. Steps act as structural units for Orchestrate application development, because each step executes as a discrete unit. Often, the operators in a step execute simultaneously, but a step cannot begin execution until the preceding step is complete.

Within a step, data is passed from one operator to next in virtual data sets. Steps pass data to other steps via Orchestrate persistent data sets, RDBMS tables, or disk files. Virtual and persistent data sets are described in the section “Virtual and Persistent Data Sets” on page 1-7.

In the figure below, the final operator in Step 1 writes its resulting data to two persistent data sets. Operators in Step 2 read these data sets as input.



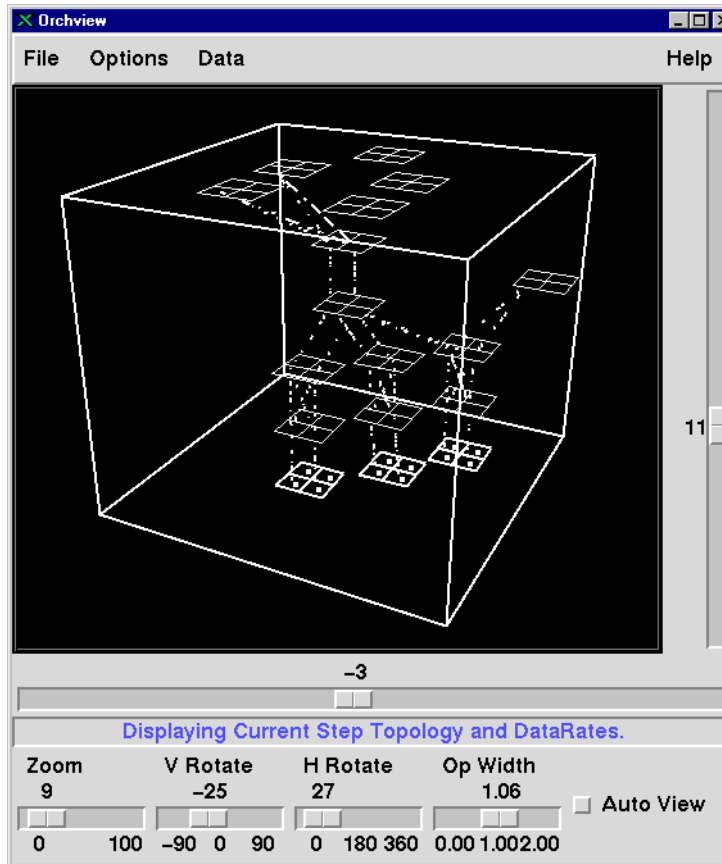
A step is also the unit of error handling in an Orchestrate application. All operators within a step succeed or fail as a unit, allowing you to conditionalize application execution based on the results of a step. In the event of a failure during step execution, the Orchestrate framework performs all necessary clean up of your system. This includes deleting any files used for temporary storage and the freeing of any system resources used by the step.

For more information about steps, see the chapter “Orchestrate Steps”.



## The Orchestrate Performance Monitor

You can direct information about an executing step to the Performance Monitor, Orchestrate's dynamic performance visualization tool. The figure below shows the Performance Monitor window, in which a rectangular grid corresponds to an Orchestrate operator in a data-flow diagram and lines connecting the grids correspond to records flowing between operators.



The Performance Monitor produces a graphical, 3-D representation of an Orchestrate application step as it executes. The Performance Monitor allows you to track the progress of an application step and to display and save statistical information about it, both during and after completion of execution.

See the chapter “The Performance Monitor” for more information.

# Creating Orchestrate Applications

The following is a general procedure for developing an Orchestrate application:

1. Create a data-flow model of your application. Data-flow models are introduced in the section “Data-Flow Modeling” on page 1-5.
2. Create any custom operators required by your application. Custom operators are introduced in the section “Prebuilt and Custom Operators” on page 1-10 and described in detail in the chapter “Orchestrate Operators”.
3. Develop your application using Visual Orchestrate (see the chapter “Creating Applications with Visual Orchestrate”).
4. Create a test data set. As many Orchestrate applications process huge amounts of data, to simplify and speed debugging you will probably want to test your application first on a subset of your data.
5. Create or edit your configuration file(s). You might want to create different configuration files, for use at different stages of application development. For example, one configuration file could define a single node, a second could define a few nodes, and a third could define all nodes in your system. Then, as testing progressed, you could increase the number of nodes by changing the environment variable `APT_CONFIG_FILE` to point to the appropriate configuration file. The *Orchestrate Installation and Administration Manual* describes configuration files and environment variables in detail.
6. Run and debug your application in sequential execution mode. Sequential mode executes your application on a single processing node; the configuration file is used only to determine the number of partitions into which data sets are divided for parallel operators. You can start by using only a single partition, while you concentrate on testing and debugging your main program and operators. Later, you can use a different configuration file (or edit your original configuration file) to increase the number of partitions and, if applicable, to test your custom partitioning. Partitioning is described in the chapter “Partitioning in Orchestrate”, and the debugging process is described in the section “Setting Step Execution Modes” on page 6-10.
7. Run and debug your application in parallel execution mode. Parallel execution mode enables the full functionality of Orchestrate. You can start by running in parallel on a single node, then on a few nodes, and complete testing by running on the full parallel system.

## Orchestrate Installation and Administration

The *Orchestrate Installation and Administration Manual* thoroughly describes installation and administration tasks, such as setting environment variables and maintaining a configuration file.

## 2: Creating Applications with Visual Orchestrate

The Orchestrate graphical user interface, Visual Orchestrate, lets you create Orchestrate applications from data-flow components (operators, data sets, and steps) in a Microsoft Windows development environment. After you have created the data-flow diagram, you can deploy the application on your UNIX system.

This chapter describes how to use Visual Orchestrate to create and deploy an Orchestrate application, in the following sections:

- “The Orchestrate Development Environment” on page 2-1
- “Creating an Orchestrate Application” on page 2-3
- “Deploying the Application on Your UNIX System” on page 2-6
- “Setting User Preferences” on page 2-9
- “Setting Program Directory Paths” on page 2-13
- “Visual Orchestrate Utilities” on page 2-14

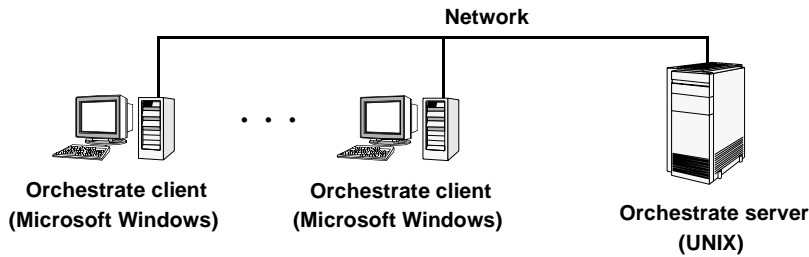
### The Orchestrate Development Environment

At the most general level, developing an application for Orchestrate has two phases:

1. Developing and testing the application on PC running Microsoft Windows.
2. Deploying the application on a target UNIX machine.

Orchestrate uses a client-server development environment, in which you develop applications on a PC running Microsoft Windows (95, 98, or NT). The PC is then connected over a network to the target UNIX machine. This development environment allows you to use Visual Orchestrate, Orchestrate's graphical user interface, to develop your application, and then run and debug the application on the target machine.

The following figure shows the Orchestrate development environment:

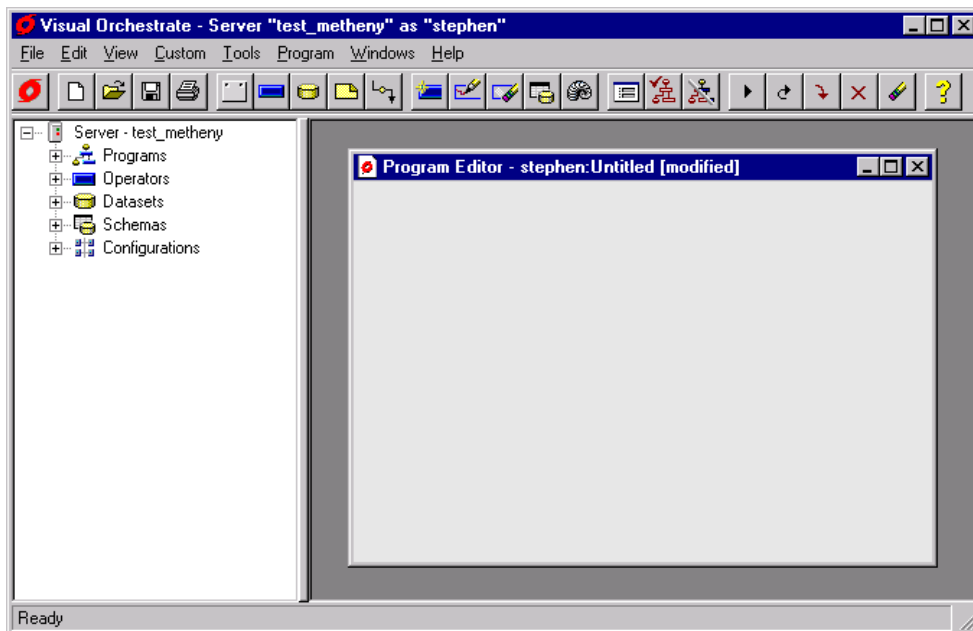


The Orchestrate server performs three basic tasks:

1. Stores your application.
2. Stores configuration information about the UNIX target system.
3. Controls and coordinates the execution of Orchestrate applications.

For Orchestration application development to take place, the Orchestrate server must be running on the target machine. You must designate an *Orchestrate server administrator*. The Orchestrate server administrator could be your system administrator or another person responsible for managing your target UNIX system. The Orchestrate server administrator is responsible for configuring and managing the Orchestrate server. For detailed information on Orchestrate server administration, see the *Orchestrate Installation and Administration Manual*.

The Orchestrate client, Visual Orchestrate, is a graphical development tool that you use on a PC to create Orchestrate applications. Shown below is the main window of Visual Orchestrate:



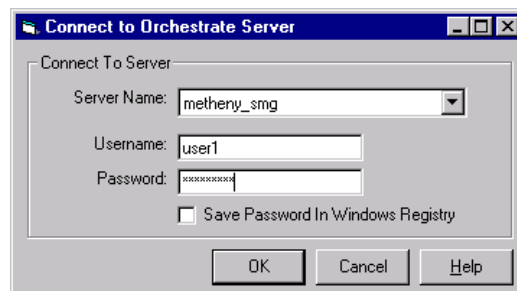
You develop your Orchestrate application by creating and configuring Orchestrate data sets, operators, and steps. Using Visual Orchestrate, you can also execute and debug your application on the target UNIX system.

When your application is complete and ready for deployment, you run the deployed application through either Visual Orchestrate or the Orchestrate UNIX-based deployment tools. These tools allow you to incorporate your Orchestrate application into a larger application that may run as part of an overnight batch job or run under a UNIX job control application.

## Creating an Orchestrate Application

This section describes the basic procedure that you follow to create an Orchestrate application using Visual Orchestrate.

1. Start Visual Orchestrate by clicking on the Windows **Start** button, then choosing **Programs -> Visual Orchestrate -> Visual Orchestrate**.
2. Connect to an Orchestrate server, either by clicking on the Torrent logo button on the toolbar or by using the Visual Orchestrate menu command **File -> Connect**. Either action opens the following dialog box:



3. Choose the **Server Name** from the drop-down list in the dialog box. You may have a choice of Orchestrate servers if your UNIX system has multiple Orchestrate installations or your network has multiple UNIX target systems.

---

**Note:** The Orchestrate server administrator must define and configure the server on each Orchestrate client PC. In addition, the Orchestrate server administrator must start your Orchestrate server before you can connect to it. These tasks are described in the chapter on client-server environment installation in the *Orchestrate Installation and Administration Manual*.

---

4. Enter your UNIX **Username** on the UNIX machine hosting the Orchestrate server. The UNIX account for **Username** must have the correct configuration settings to run Orchestrate, as described in the *Orchestrate Installation and Administration Manual*.
5. Enter the **Password** for **Username**.

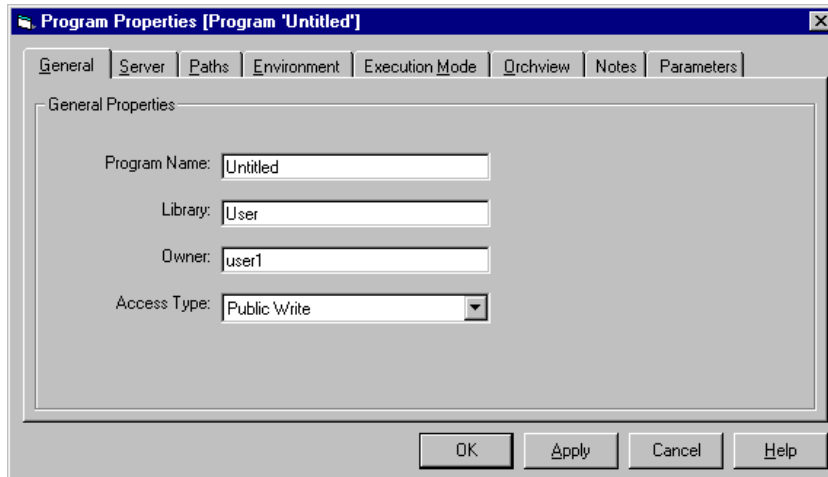
By default, Visual Orchestrate does not save the **Password**. Check **Save Password in Windows Registry** if you want Visual Orchestrate to save the **Password**.

- If you are creating a new program, choose **File -> New** from the Visual Orchestrate menu. This opens an empty **Program Editor** window. You develop a complete application (containing data sets, operators, and steps) in a single **Program Editor** window.

If you are editing an existing program, choose **File -> Open** and select the program from the list of stored programs.

- Use the **Program Properties** dialog box to configure your program.

The **Program -> Properties** menu entry opens the **Program Properties** dialog box:



Specify the **Program Name**. This is the name you use to invoke the application when deploying it on your UNIX target machine.

Specify the **Library**. This defines the library name of the program under the **Programs** entry in the display area of Visual Orchestrate.

Specify the **Owner** of the program. By default, the program owner is the same as the user name of the person logged in to Visual Orchestrate.

Specify the **Access Type** of the program. Options are:

**Public Write** (default): Anyone can read, write, or execute the program.

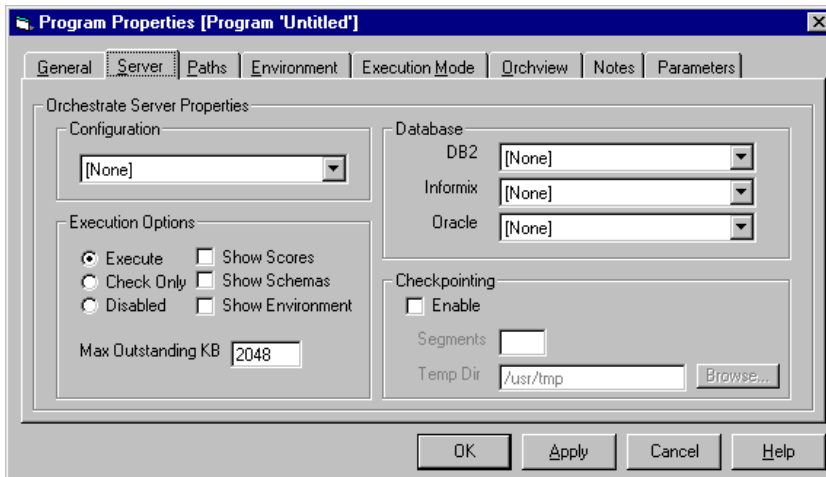
**Public Read**: Anyone can read and execute the program; only the program owner can modify it.

**Private**: Only the program owner can read, write, or execute the program.

- Choose the configuration information for your application. This information usually includes the processing nodes on the target machine that you want to use and, if applicable, the RDBMS that you want to access.

You can specify the configuration information for the entire program, or you can separately configure each step in the program. Use the Visual Orchestrate **Program -> Properties** menu entry to set the global properties for the application. Later, you can configure each step, as necessary.

In the **Program Properties** dialog box, choose the **Server** tab to open the following form:



**Configuration:** Select the Orchestrate configuration used to execute the application. The configuration defines the processing nodes and disk drives available for use by your program. Often, the server configuration is the only program property that you need to set.

The Orchestrate server administrator is required to set up at least a default configuration before you can create an Orchestrate program. If no configuration is available, see the Orchestrate server administrator.

You may have several different configurations available. For example, one configuration may be for testing and another for production.

You can validate a configuration using the **Tools->Check Config** menu entry. See the section “Checking an Orchestrate Configuration” on page 2-14 for more information.

**Database:** (Optional) Specify the database configuration used by the application. The database configuration defines the database type (DB2, Informix, or Oracle), as well as the specific database configuration to use.

If you are accessing a database, the Orchestrate server administrator must set up at least a default database configuration before you can create an Orchestrate program. If no configuration is available, see the Orchestrate server administrator.

You may have several different configurations available, depending on the database and database data that you want to access.

**Execution Options:** Select **Check Only** to validate the step but not to run it, and **Execute** to execute the step.

Leave the remaining settings in their default state. See the chapter “Orchestrate Steps” for more information.

---

**Note:** For information on the **Paths** settings, see the section “Setting Program Directory Paths” on page 2-13. For information on the **Environment** and **Execution Mode** tabs, see the chapter “Orchestrate Steps”. For information on the **Orchview** tab, See the chapter “The Performance Monitor”. For information on the **Parameters** tab, see the section “Using the Program Properties Dialog Box to Set Program Parameters” on page 2-15.

---

9. Develop your application by creating a data-flow diagram containing data sets, operators, and steps, as described in the chapter “Orchestrate Steps”.
10. Optionally, check for programming errors in your application without running it. (Note that Orchestrate automatically checks your application for errors when you select **Run**, as described in the next step.) To check your application, click the **Validate** button on the Visual Orchestrate tool bar:



Orchestrate checks your application for programming errors, such as invalid links in data flows. Orchestrate displays invalid data-flow elements in red, with an explanatory message.

After fixing any reported errors, you can run validate again and continue the process until you have corrected all errors that Orchestrate detects.

11. Run your application, by clicking the **Run** button on the tool bar:



If you have not already validated your application and corrected all reported errors, Orchestrate now checks it. After successful validation, Visual Orchestrate runs your application.

During execution, an **Execution Window** shows the output of your application. If any run-time errors occur, Visual Orchestrate reports them in the **Execution Window**. You can use this information to correct the error in your application.

12. Optionally, deploy your application on the target UNIX system. The next section describes how to deploy your application, using the Orchestrate deployment tools.

## Deploying the Application on Your UNIX System

Once your application executes correctly under the control of Visual Orchestrate, you can optionally deploy the application on your UNIX target system. A deployed application executes under the control of Orchestrate's UNIX deployment tools, not under Visual Orchestrate. The Orchestrate deployment tools are UNIX commands that you use to invoke and monitor the execution of your application.

You do not have to deploy your application on the same Orchestrate server that you used to develop it. In fact, it is common to have two or more Orchestrate servers installed. This allows you to use one server for application development and testing and another for application deployment.



Before deploying your application, you can use the **File** menu command (**File -> Copy To**) to copy a program, Orchestrate configuration, custom operator, or schema from the server to which you are currently connected, to another server.

## Deploying Your Application with `job-manager`

To deploy and manage your application on your target UNIX system, you use the Orchestrate utility `job-manager`. The `job-manager` utility is located in `install_dir/apt/bin`, where `install_dir` is the path to the Orchestrate installation. You must either include this directory in your UNIX PATH or provide the complete path name to the utility.

You run `job-manager` with the following command:

```
$ job-manager command
```

where `command` is the command option to `job-manager`. The `command` options are:

- `run jobname`
- `abandon jobinstance`
- `restart jobinstance`
- `kill jobinstance`
- `errors jobinstance`

---

**Note:** Before you run your application with `job-manager`, you must execute it at least once using Visual Orchestrate.

---

## Running your Application

To invoke an Orchestrate application developed using Visual Orchestrate, you use the `run` command with `job-manager`. This command takes the name of the Orchestrate application to invoke. An Orchestrate application name has the form:

```
libname:programe
```

To find the `libname` and `programe`, see the **Program Properties** dialog box, **General** tab, fields **Library** and **Program Name**.

The application executes using the current server configuration. You can examine and, if necessary, modify this server configuration using the `set-server-parameters` command. See the *Orchestrate Installation and Administration Manual* for information on using this command.

The following is an example of the `run` command:

```
$ job-manager run User:myApp > jobInstance.txt
```

## Checking the Job Instance Number and Exit Status

In the example above, `job-manager` executes the application named `User:myApp`. If the application completes successfully, `job-manager` writes the job instance number to the file `jobinstance.txt`. The Orchestrate server assigns an instance number to each run of your application. Therefore, if you invoke multiple instances of `User:myApp`, you can identify each instance by its job instance number. All `job-manager` commands (other than `run`), take as an argument a job instance number.

As your application executes, the Orchestrate server also writes error messages to `jobinstance.txt`. It is recommended that immediately after your application run completes, you do the following:

1. Check the exit status of the `job-manager` command that you used to run the application, as you check the exit status of any other UNIX shell command.
2. Execute the following command to display any run-time error messages:

```
$ job-manager errors 'cat jobInstance.txt'
```

## Terminating an Application Run

To terminate an Orchestrate application, you must first terminate the `job-manager` command. If you used the keyboard to invoke the `job-manager` command, terminate it by pressing `<Ctrl-C>`.

If the `job-manager` command was invoked from a script, you must first halt the script. Then, terminate the Orchestrate application by using the following command:

```
$ job-manager kill 'cat jobInstance.txt'
```

## Summary of Deployment Commands

The following table lists the commands, and command options, to `job-manager`:

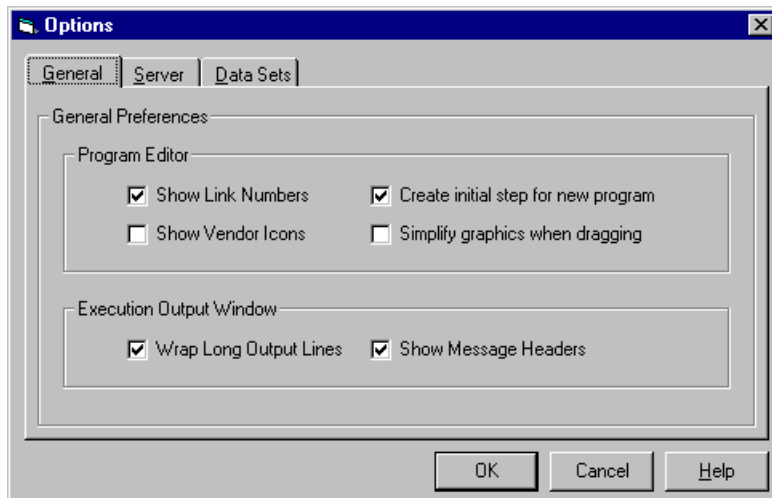
Command	Use
<code>run</code>	<p><code>run jobname</code></p> <p>Executes the application identified by <code>jobname</code>, which has the form:</p> <pre>libname:progname</pre> <p>The <b>Library</b> and <b>Program Name</b> entries in the <b>General</b> tab are of the <b>Program Properties</b> dialog box define this information.</p>
<code>abandon</code>	<p><code>abandon jobinstance</code></p> <p>If your application terminates during a restartable step, this command deletes all data added to any output persistent data sets by all iterations of the abandoned step. Specifying this option means you cannot resume the application.</p> <p><code>jobinstance</code> specifies the Orchestrate job number as returned by the <code>run</code> command.</p>

Command	Use
kill	kill <i>jobinstance</i> Terminates an executing application. <i>jobinstance</i> specifies the Orchestrate job number as returned by the run command.
errors	errors <i>jobinstance</i> Displays on the screen any error messages generated by an application. <i>jobinstance</i> specifies the Orchestrate job number as returned by the run command.

Refer to the *Orchestrate Installation and Administration Manual* for a further discussion of deployment.

## Setting User Preferences

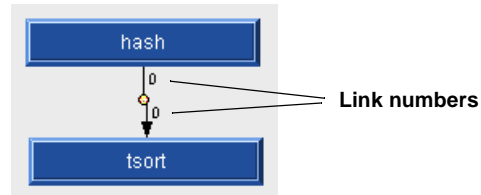
You set preferences for Visual Orchestrate through the **Tools** menu. Select **Tools->Options** to open the following dialog box:



From the **General** tab:

- Click **Show Link Numbers** to configure Visual Orchestrate to show the number of the operator output and operator input for each end of a link between two operators. By default, this box

is unchecked. Checking it enables link numbers, as shown below:



- Click **Show Vendor Icons** to cause the vendor name to appear in the operator icon in the **Program Editor** window. By default, this option is unchecked.
- Click **Create initial step for new program** to cause Visual Orchestrate to create an empty step when you create a new program. This is the default action of Visual Orchestrate.
- Click **Wrap Long Output Lines** to configure Visual Orchestrate to wrap long text lines in the **Execution Window**. Otherwise, you must scroll the window to view the entire line.
- Click **Show Message Headers** to enable the message headers in the Execution Window. By default, message headers are suppressed. Your setting of this option takes effect the next time you run your application.

Shown below is an example **Execution Window** display from an application run, with message headers disabled:

```

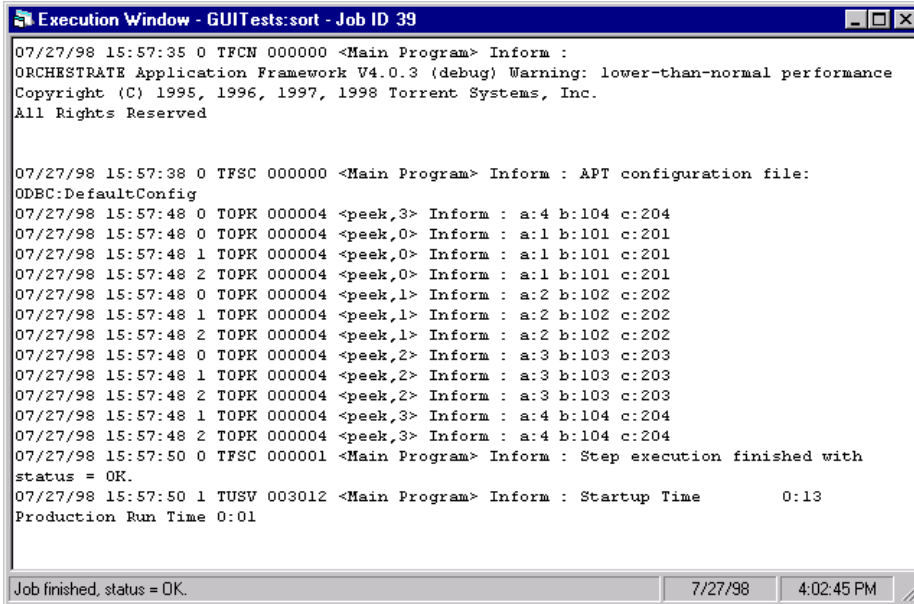
Execution Window - GUI Tests: sort - Job ID 38
-----
ORCHESTRATE Application Framework V4.0.3 (debug) Warning: lower-than-normal performance
Copyright (C) 1995, 1996, 1997, 1998 Torrent Systems, Inc.
All Rights Reserved

APT configuration file: ODBC:DefaultConfig
a:3 b:103 c:203
a:1 b:101 c:201
a:1 b:101 c:201
a:1 b:101 c:201
a:2 b:102 c:202
a:2 b:102 c:202
a:2 b:102 c:202
a:3 b:103 c:203
a:3 b:103 c:203
a:4 b:104 c:204
a:4 b:104 c:204
a:4 b:104 c:204
Step execution finished with status = OK.
Startup Time      0:13
Production Run Time 0:01

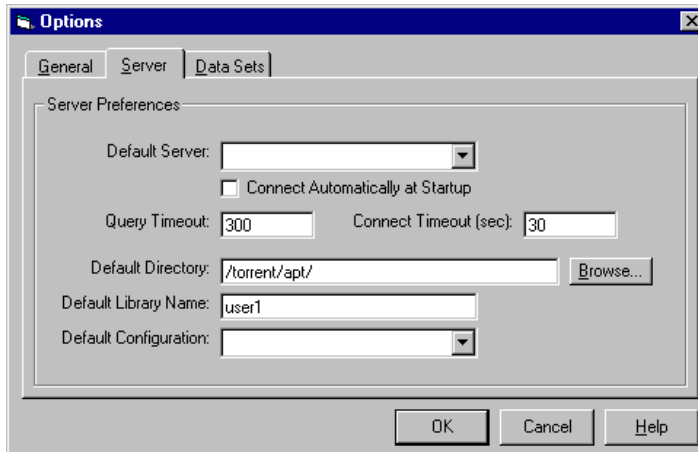
Job finished, status = OK.          7/27/98      4:02:14 PM

```

Shown below is the output from a run of the same application, after you have enabled display of messages headers:



From the **Server** tab, you set the server characteristics:



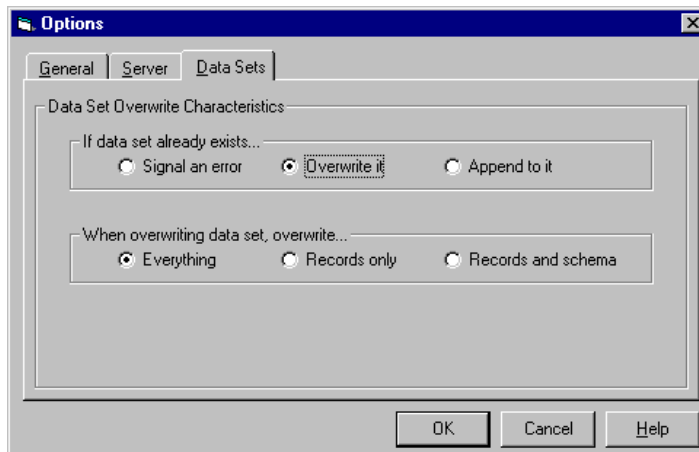
- Set the **Default Server** used by Visual Orchestrate.

Set **Connect Timeout** to number of seconds that Visual Orchestrate will wait before signalling a server-not-present error.

- Click **Connect Automatically at Startup** to cause Visual Orchestrate to connect to the **Default Server** whenever you start Visual Orchestrate.
- Use **Default Directory** to set the server working directory for your applications, as well as setting the default path for the file browser and shell tool (see “Using the Orchestrate Shell” on page 2-15).
- Use **Default Library** to set the default library name for all programs, operators, and schemas that you create.
- Set the default Orchestrate configuration for all programs created in Visual Orchestrate using the **Default Configuration** pull-down list.

You can override the default configuration for a program (using the **Program -> Properties** menu command) or for a step (by double clicking on the step to open the **Step Properties** dialog box).

From the **Data Sets** tab, choose the data set overwrite characteristics:

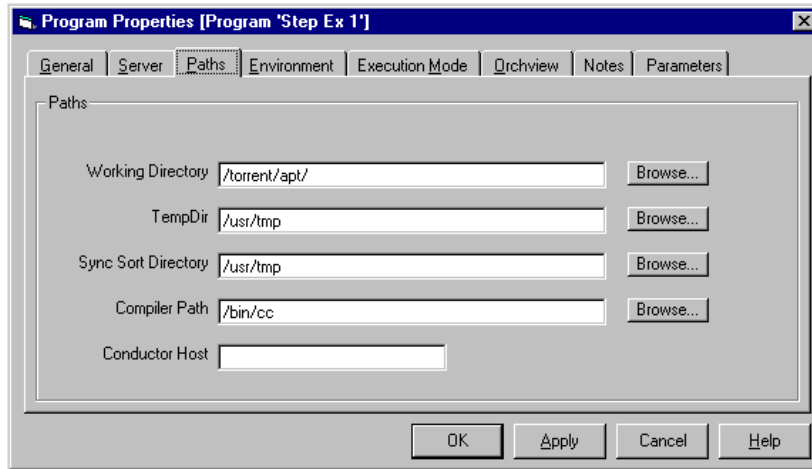


- In the panel **If data set already exists**, select the action that your application will take if an overwrite is attempted:
  - **Signal an error** to cause the step to fail with a message.
  - **Overwrite it** to allow the overwrite to occur.
  - **Append to it** to append the data while keeping the current data, schema, and partitioning information.
- If in the first panel you selected **Overwrite it**, select the extent of the overwrite:
  - **Everything** to overwrite the records, schema, and partitioning information.
  - **Records only** to overwrite only the records.
  - **Records and schema** to overwrite the records and schema, but not the partitioning information.

For information on data sets and record schemas, see the chapter “Orchestrate Data Sets”. For information on partitioning, see the chapter “Partitioning in Orchestrate”.

## Setting Program Directory Paths

To optionally change the directory paths used by your program, you use the **Paths** tab in the **Program Properties** dialog box, shown below:



The **Paths** tab lets you set any of the following properties:

Use **Working Directory** to set the working directory for the step. This setting overrides the **Default Directory** setting in the **Tools -> Options** dialog box.

**TempDir**: By default, Orchestrate uses the directory `/tmp` for some temporary file storage. If you do not want to use this directory, you can set the parameter **TempDir** to a path name to a different directory.

**SortDirectory**: Optionally sets the location of SyncSort on your processing nodes. Usually, SyncSort will be installed in the same location on every node. The Orchestrate `psort` operator can use SyncSort to sort a data set. If both **SortDirectory** and the Orchestrate server parameter **SYNCSORTDIR** are undefined, the `psort` operator uses UNIX sort.

**SortDirectory** overrides the Orchestrate server parameter **SYNCSORTDIR**.

**Compiler Path**: Sets the path to the C++ compiler used by Orchestrate when you create native operators. See the section “Configuring Orchestrate For Creating Operators” on page 12-4 for more information.

**Conductor Host**: The network name of the processing node from which you invoke an Orchestrate application should be included in the configuration file using either `node` or `fastname`. If the network name of the node is not included in the configuration file, Orchestrate users must set by **Conductor Host** to the fastname of the node invoking the Orchestrate application.

---

**Note:** The maximum length of any path name that you enter in Visual Orchestrate is 254 characters.

---

## Visual Orchestrate Utilities

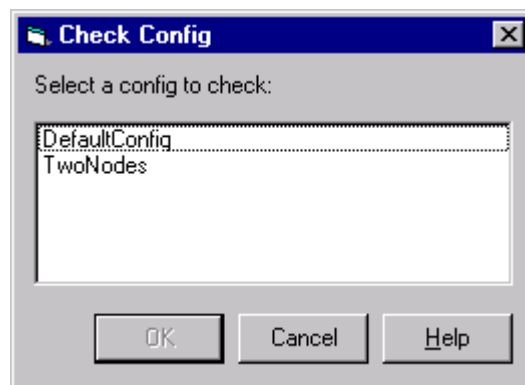
This section describes Visual Orchestrate built-in utilities, for performing the following tasks:

- “Checking an Orchestrate Configuration” on page 2-14
- “Using the Orchestrate Shell” on page 2-15
- “Generating an osh Script to Configure and Run a Program” on page 2-15
- “Using the Lock Manager” on page 2-15

### Checking an Orchestrate Configuration

An Orchestrate configuration describes the processing nodes on the target machine for your application. The **Tools -> Check Config** feature lets you to test the validity of an Orchestrate configuration that the Orchestrate server administrator has made available on your server. To check a configuration:

1. Select **Tools -> Check Config** to open the **Check Config** dialog box, which displays a list of all available Orchestrate configurations. The following sample **Check Config** dialog box shows two available configurations:



2. Select a configuration name from the dialog box, and then click **OK**.

An execution window opens containing the results of the test.

Refer to the *Orchestrate Installation and Administration Manual* for a detailed discussion of configuring your Orchestrate system.

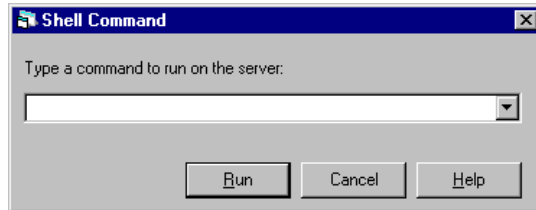


## Using the Orchestrate Shell

You can issue commands from the Orchestrate shell (`osh`) by means of the Orchestrate shell tool. To access the shell tool, on the tool bar click the shell icon:



The following dialog box is displayed:



You can now issue `osh` commands from the default directory that is defined in **Tools** —> **Options** ... —> **Server** —> **Default Directory**.

## Generating an `osh` Script to Configure and Run a Program

In addition to Visual Orchestrate, Orchestrate has a command interface, called the Orchestrate shell or `osh`. When using `osh`, you build your Orchestrate application from UNIX command lines. You can invoke simple Orchestrate applications using a single `osh` command line, and you can create shell scripts containing multiple `osh` commands.

The **Tools** menu (**Tools** -> **Generate Script**) and the **Program** menu (**Program** -> **Generate Script**) let you generate a file containing an `osh` script from a Visual Orchestrate program. You can then run the generated script (which you may need to edit) from the UNIX command line. The script will create the necessary configuration file, set environment parameters necessary to run the application, and run the application. For information on creating `osh` commands and scripts, see the chapter on creating applications in the *Orchestrate Shell User's Guide*.

## Using the Program Properties Dialog Box to Set Program Parameters

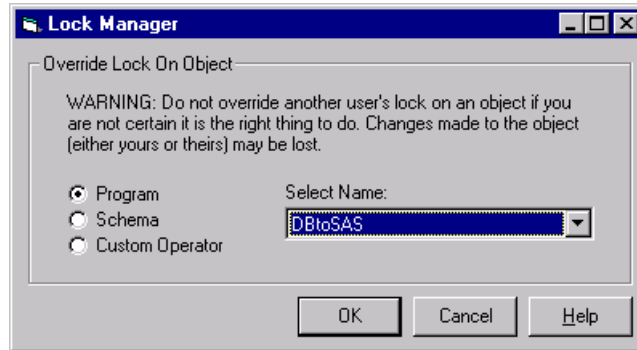
You also have the option of setting initial (or default) values for parameters used by your application, by using the **Parameters** tab of the **Program Properties** dialog box. This tab lists the names of all variables used in your program. You can also set values of parameters in pre and post scripts, as described in the section “Using Pre and Post Scripts” on page 6-12.

## Using the Lock Manager

When a Visual Orchestrate user opens a program, schema definition, or custom operator, the Orchestrate server places a *lock* on the object. This lock prevents a user from opening and writing to an object after another user has already opened the object.

A user can open a locked object for reading only. If the user wants to modify the locked object, the user must first save the object with a different name.

You may occasionally need to explicitly clear a lock. For example, an object may remain locked as a result of a system problem, such as a crash of the PC running Visual Orchestrate. To explicitly clear a lock, use the Lock Manager menu command, **Tools -> Lock Manager**. This menu command opens the following dialog box:



1. Choose the object type whose lock you want to clear: **Program**, **Schema**, or **Custom Operator**.
2. Use **Select Name** to choose the name of the object.

---

**Warning:** Do *not* use the Lock Manager to reset a lock in order to gain write access to a locked object. Doing so would give write access both to you *and* to the user who originally opened the object, and you would overwrite each other's work.

---

## 3: Orchestrate Data Types

This chapter covers fundamental information about the Orchestrate data types, through the following topics:

- “Introduction to Orchestrate Data Types” on page 3-1
- “Orchestrate Data Types in Detail” on page 3-2
- “Performing Data Type Conversions” on page 3-8

### Introduction to Orchestrate Data Types

Orchestrate supports all value (scalar) data types and two aggregate data types. Orchestrate data types are listed in the table below.

Orchestrate Data Type	Size	Description
date	4 bytes	Date, with month, day, and year.
decimal	(Roundup(p)+1)/2	Packed decimal, compatible with IBM packed decimal format.
sfloat	4 bytes	IEEE single-precision (32-bit) floating-point value.
dfloat	8 bytes	IEEE double-precision (64 bits) floating-point value.
int8 uint8	1 byte	Signed or unsigned integer of 8 bits.
int16 uint16	2 bytes	Signed or unsigned integer of 16 bits.
int32 uint32	4 bytes	Signed or unsigned integer of 32 bits.
int64 uint64	8 bytes	Signed or unsigned integer of 64 bits.
raw	1 byte per character	Untyped collection, consisting of a fixed or variable number of contiguous bytes and an optional alignment value.
string	1 byte per character	ASCII character string of fixed or variable length.

Orchestra Data Type	Size	Description
subrec	Sum of lengths of aggregate fields	Aggregate consisting of nested fields.
tagged	Sum of lengths of aggregate fields	Aggregate consisting of tagged fields, of which one can be referenced per record.
time	5 bytes	Time of day, with resolution in seconds or microseconds.
timestamp	9 bytes	Single field containing both a date and a time value.

## Vectors

Orchestra supports vectors, which are one-dimensional arrays of any type except tagged. For details on vectors, see the section “Vector Fields” on page 4-24.

## Support for Nullable Fields

If a field is *nullable*, it can contain a valid representation of null. When an application detects a null value in a nullable field, it can take an action such as omitting the null field from a calculation or signaling an error condition. You can specify nullability for an Orchestra record field of any Orchestra value data type. For fields of aggregate data types, you can specify nullability for each element in the aggregate. For details on field nullability, see the section “Defining Field Nullability” on page 4-19.

## Orchestra Data Types in Detail

This section describes the characteristics of each Orchestra data type listed in the section “Introduction to Orchestra Data Types” on page 3-1. For applicable data types, the descriptions include data conversion details, such as the data conversion format string for values passed to certain operators, such as `import` and `export`. For information on conversion between types, see the section “Performing Data Type Conversions” on page 3-8.

## Date

The Orchestra `date` data type is compatible with the RDBMS representations of date supported by DB2, INFORMIX, Oracle, and Teradata.

An Orchestra date contains the following information:

- *year*: between 1 and 9999, inclusive
- *month*: between 1 and 12, inclusive

- *day of month*: between 1 and 31, inclusive

You can also specify a date using two forms of the Julian representation:

- Julian *date* uses two components to define a date: a year and the day of the year in the range 1 to 366, inclusive.
- Julian *day* contains a single component specifying the date as the number of days from 4713 BCE January 1, 12:00 hours (noon) GMT. For example, January 1, 1998 is Julian day count 2,450,815.

## Data Conversion Format for a Date

By default, an Orchestrate operator interprets a string containing a date value `yyyy-mm-dd`. If the date argument does not include a day, the operator sets it to the first of the month in the destination field. If the date does not include either the month or the day, they default to January 1. Note that if the date includes a day of the month, it *must* also include a month.

If you wish to specify a non-default format, you can pass the operator an optional *format string* describing the format of the date argument. In a format string for a source string converted to a date, you must zero-pad the date components (date, month, and year) to the component length you specify. For a destination string that receives a date, Orchestrate zero-pads the date components to the specified length.

The following list describes the components that you can use in the format string. In addition to the required portions of the `date` components described below, you can also include non-numeric characters as separators or delimiters.

- `%dd`: A two-digit day of the month (range of 1 - 31).
- `%ddd`: Day of year in three-digit form (range of 1 - 366).
- `%mm`: A two-digit month (range of 1 - 12).
- `%<year_cutoff>yy`: A two-digit year derived from `yy` and a four-digit year cutoff.  
`<year_cutoff>` specifies the starting year of the century in which the specified `yy` falls. You can specify any four-digit year as `<year_cutoff>`. Then, `yy` is interpreted as the low-order two digits of the year that is the same as or greater than the year cutoff. For example, to indicate that the year passed as 31 represents 1931, you could specify the `<year_cutoff>` of 1930. If you pass a format string with a `<year_cutoff>` of 1930 and the corresponding year in the date string argument is 29, Orchestrate interprets that year as 2029, which is the next year ending in 29 that is after the `<year_cutoff>`.
- `%yy`: A two-digit year derived from a default year cutoff of 1900. For example, using the `yy` format for a year of 42 results in its interpretation as 1942.
- `%yyyy`: A four-digit year.

---

**Note:** Each component of the `date` format string must start with the percent symbol (%).

---

Following are examples of complete date format strings using one or more of the possible components:

- `%mm/%dd/%yyyy` for dates in the form 03/24/1995 (interpreted as March 24, 1995)
- `%mm-%dd-%1800yy` for dates in the form 03-24-95 where 1800 is the year cutoff (interpreted as March 24, 1895)
- `%ddd-%yy` for dates in the form 056-1998 (interpreted as February 25, 1998)
- `%dd/%mm/%yy` for dates in the form 25/12/96 where 1900 is the default year cutoff (interpreted as December 25, 1996).

## Decimal

Orchestra provides the `decimal` data type for representing decimal data. The Orchestra `decimal` format is compatible with the IBM packed decimal data format and with the DB2, Informix, Oracle, and Teradata `DECIMAL` data types.

The Orchestra `decimal` format is characterized by two components: precision (P) and scale (S). Precision is the total number of digits in the decimal. Precision must be at least 1, and the maximum precision possible is 255. Scale is the number of digits to the right of the decimal point, comprising the fractional part of the `decimal`. The ranges of permissible values for precision and scale are the following:

$$1 \leq P \leq 255, 0 \leq S \leq P$$

A `decimal` with a scale of 0 represents integer values (no fractional part).

Decimal values are always signed. The `decimal`'s sign nibble represents the sign by one of six numeric values, shown below:

Sign Nibble Value	Sign	Notes
0xA	+	
0xB	-	
0xC	+	(Preferred) Always generated when Orchestra writes to a decimal with a positive value.
0xD	-	(Preferred) Always generated when Orchestra writes to a decimal with a negative value.
0xE	+	
0xF	+	

The number of bytes occupied by a `decimal` value is  $(P/2)+1$ . This packed decimal representation uses one nibble for each decimal digit, plus a sign nibble. If the number of decimal digits (that is, the precision) is even, Orchestra prepends a zero-valued leading nibble in order to make the total number of nibbles even.

By default, a `decimal` with zero in all its nibbles is invalid. Many operations performed on a `decimal` detect this condition and either fail or return a flag signifying an invalid `decimal` value. You can, however, specify that an operation treat a `decimal` containing all zeros as a valid

representation of the value 0. In that case, Orchestrate treats the `decimal` as valid and performs the operation.

A `decimal`'s available *range*, or its maximum and minimum possible values, is based on its precision and scale. The equation for determining a `decimal`'s upper and lower bounds is the following:

$$\text{range} = \pm 10^{(P-S)}$$

Note that this is an exclusive range, so that the `decimal` will always be less than the maximum and greater than the minimum. Thus, if  $P = 4$  and  $S = 2$ , the range is  $-99.99 < \text{decimal} < 99.99$  (it is not  $-100.00 < \text{decimal} < 100.00$ ).

## String Assignment and Conversion for Decimals

Orchestrate lets you assign a `string` to a `decimal` and a `decimal` to a `string`. The two most likely situations for such assignments are performing an import or export and using a data set as an input to an operator.

When a `string` is assigned to a `decimal`, the `string` is interpreted as a `decimal` value. Strings assigned to `decimals` must be in the form:

```
[ +/- ]ddd[ .ddd ]
```

where items in brackets are optional.

By default, Orchestrate treats the `string` as null-terminated. However, you can also specify a string length. Orchestrate ignores leading or trailing white space in the string. Range checking is performed during the assignment, and a requirement failure occurs if the assigned value does not fit within the `decimal`'s available range.

You can also assign a `decimal` to a `string`. The destination string represents the `decimal` in the following format:

```
[ +/- ]ddd.[ddd]
```

A leading space or minus sign is written, followed by the decimal digits and a decimal point. Leading and trailing zeros are not suppressed. A fixed-length string is padded with spaces to the full length of the string. A fixed-length string must be precision + 2 bytes long (one byte for the leading sign indicator and one byte for the decimal point). A range failure occurs if a fixed-length string field is not large enough to hold the `decimal`.

## Floating-Point

Orchestrate defines single- and double-precision floating-point data types. All standard arithmetic and conditional operations are supported for these floating-point data types.

## Integers

Orchestrate defines signed and unsigned, 8-, 16-, 32-, and 64-bit integer data types. All standard arithmetic and conditional operations are supported by these data types.

## Raw

The Orchestrate `raw` data type is a collection of untyped bytes, similar to the `void` data type in the C programming language. A `raw` in Orchestrate always contains a length, rather than a null terminator.

## String

An Orchestrate `string` field always specifies a length and does not include a null terminator.

## Subrecord

You define nested field definitions, or subrecords, with the aggregate data type `subrec`. A subrecord itself does not define any storage; instead, the fields of the subrecord define storage. The fields in a subrecord can be of any data type, including `tagged`.

## Tagged

You define tagged aggregate fields (similar to C unions) with the aggregate data type `tagged`. Defining a record with a tagged aggregate allows each record of a data set to have a different data type for the tagged field. When your application writes to a field in a tagged aggregate field, Orchestrate updates the tag, which identifies it as having type of the field that is referenced.

The data type of a tagged aggregate subfields can be of any Orchestrate data type except `tagged` or `subrec`.

## Time

The `time` data type uses a 24-hour representation, with either a one-second resolution or a microsecond resolution. The Orchestrate `time` data type is compatible with most RDBMS representations of time.

Valid times are in the range 00:00:00.000000 to 23:59:59.999999. Incrementing a time value of 23:59:59.999999 wraps the time around to 00:00:00.000000.



The `time` data type contains no information regarding time zone. In Orchestrate operations, all time values are treated as if they are in the same time zone.

## Data Conversion Format for a Time Value

By default, an Orchestrate operator interprets a string containing a time value as `hh:nn:ss`, where `hh` is the hour, `nn` is the minute (so indicated to avoid confusion with the month value in the `date` format), and `ss` is the second.

If you wish to specify a non-default format, you can pass the operator an optional *format string* describing the format of the time argument. In a format string for a source string converted to a time, the time components (hour, minutes, and seconds) must be zero-padded to the character length specified by the format string. For a destination string that receives a time, Orchestrate zero-pads the time components to the specified length.

The possible components of the format string are:

- `%hh`: A two-digit hours component.
- `%nn`: A two-digit minute component (`nn` represents minutes because `mm` is used for the month of date).
- `%ss`: A two-digit seconds component.
- `%ss.N`: A two-digit seconds plus fractional part where *N* is the number of fractional digits with a maximum value of 6. If *N* is 0, no decimal point is printed as part of the seconds component. Trailing zeros are not suppressed.

---

**Note:** Each component of the `time` format string must start with the percent symbol (%).

---

For example, you could specify a format string of `%hh:%nn` to specify that the string contains only an hour and minutes component. You could also specify the format as `%hh:nn:ss.4` to specify that the string also contains the seconds to four decimal places.

## Timestamp

A `timestamp` includes both a date, as defined by the Orchestrate `date` data type, and a time, as defined by the `time` data type. The Orchestrate `timestamp` data type is compatible with most RDBMS representations of a timestamp.

## Data Conversion Format for a Timestamp Value

By default, an Orchestrate operator interprets a string containing a `timestamp` value as the following:

```
yyyy-mm-dd hh:nn:ss
```

Note that the month is represented by `mm`, while minutes are represented by `nn`. Also note the required space between the date and time parts of the `timestamp`.

To specify a non-default conversion format for a `timestamp`, you use a format string that combines the format strings for `date` (see the section “Data Conversion Format for a Date” on page 3-3) and `time` (see the section “Data Conversion Format for a Time Value” on page 3-7). The conversion format you specify must be valid for both the date and the time segments, as described in the applicable sections.

## Performing Data Type Conversions

This section describes the following topics:

- “Rules for Orchestrate Data Type Conversions” on page 3-8
- “Summary of Orchestrate Data Type Conversions” on page 3-9
- “Example of Default Type Conversion” on page 3-10
- “Example of Type Conversion with `modify`” on page 3-10
- “Data Type Conversion Errors” on page 3-11

### Rules for Orchestrate Data Type Conversions

For a data set to be used as input to or output from an Orchestrate operator, its record schema must be compatible with the interface for that operator, as follows:

- The names of the data set's fields must be identical to the names of the corresponding fields in the operator interface.
- The data type of each field in the data set must be compatible with that of the corresponding field in the operator interface. Data types are compatible if Orchestrate can perform a default data type conversion, translating a value in a source field to the data type of a destination field.

If there are any discrepancies in field names, you must use the `modify` operator to change the field names for your data set. If your data set has any fields with incompatible data types, you must use the `modify` operator to convert those types so they are compatible.

---

**Note:** For all built-in Orchestrate operators (except `import/export` of flat files), the internal, physical representation of Orchestrate data types is handled transparently by Orchestrate. For details on using the `import` and `export` operators, see the *Orchestrate User's Guide: Operators*.

---

## Summary of Orchestrate Data Type Conversions

Orchestrate performs default type conversions on Orchestrate built-in numeric types (integer and floating point), as defined in the book *C: A Reference Manual* (Third Edition), by Harbison and Steele. Orchestrate also performs default data conversions involving `decimal`, `date`, `time`, and `timestamp` fields. In addition, you can perform a number of data type conversions with the `modify` operator, as described in the *Orchestrate User's Guide: Operators*. Data type conversions are described in more detail in the section “Data Set and Operator Data Type Compatibility” on page 5-17.

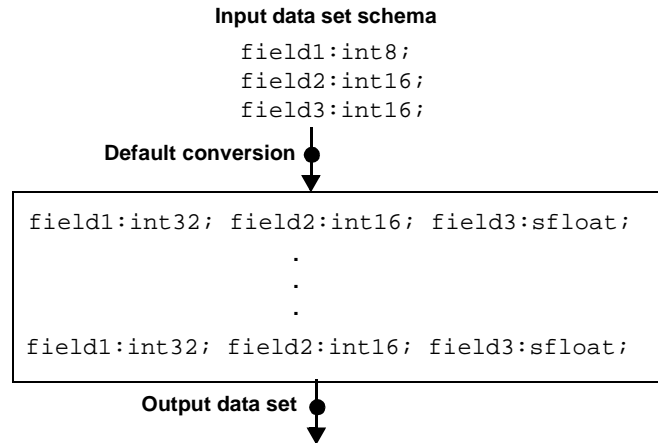
The table below shows the default data type conversions performed by Orchestrate and the conversions that you can perform with the `modify` operator, as follows:

- `d` indicates that Orchestrate performs a default type conversion from source field type to destination field type.
- `m` indicates that you can use a conversion specification with `modify` to convert from source field to destination field.
- A blank cell indicates that Orchestrate does not provide any conversion.

Source Field	Destination Field															
	int8	uint8	int16	uint16	int32	uint32	int64	uint64	sfloat	dfloat	decimal	string	raw	date	time	timestamp
int8	d,m	d	d	d	d	d	d	d	d	d,m	d	d,m		m	m	m
uint8	d		d	d	d	d	d	d	d	d	d	d				
int16	d,m	d		d	d	d	d	d	d	d	d	d,m				
uint16	d	d	d		d	d	d	d	d	d	d	d				
int32	d,m	d	d	d		d	d	d	d	d	d	d,m		m		m
uint32	d	d	d	d	d		d	d	d	d	d	m		m		
int64	d,m	d	d	d	d	d		d	d	d	d	d				
uint64	d	d	d	d	d	d	d		d	d	d	d				
sfloat	d,m	d	d	d	d	d	d	d		d	d	d				
dfloat	d,m	d	d	d	d	d	d	d	d	d,m	d,m	d,m			m	m
decimal	d,m	d	d	d	d,m	d	d,m	d,m	d	d,m	d,m	d,m				
string	d,m	d	d,m	d	d	d,m	d	d	d	d,m	d,m	d,m		m	m	m
raw	m				m								d			
date	m		m		m	m						m		m		m
time	m				m					m		m			d	d,m
timestamp	m				m					m		m		m	m	d

## Example of Default Type Conversion

The following figure shows an input data set schema in which the data types of fields `field1` and `field3` do not match, but as shown in the conversion table above, they are compatible with the types of the operator's corresponding fields:



As shown, Orchestra performs the following two default conversions:

- The data type of `field1` is converted from `int8` to `int32`.
- The data type of `field3` is converted from `int16` to `sfloat`.

## Example of Type Conversion with `modify`

The table in the section “Summary of Orchestrate Data Type Conversions” on page 3-9 shows that you can use `modify` to convert a field of type `time` to type `int32`. For example, to convert field `t` from type `time` to type `int32`, perform the following steps:

1. Right-click the `modify` operator to open its **Operator Properties** dialog box. Press **Add**, to open the **Option Editor** dialog box.
2. In the **Option Editor dialog box**, press **Edit** to open the **Modify Adapter Editor** dialog box.
3. In the **Rename/Convert** area of the **Modify Adapter Editor** dialog box, press **Add** to open the **Rename/Conversion** dialog box.
4. In the **Rename/Conversion** dialog box, do the following:
  - Enter the **Source Field Name**, `t`.
  - Enter the **Dest(ination) Field Name**, `t` (or another name of your choice).
  - Check **Set Dest Type To**, and select the data type of the destination field, `int32`.
  - Check **Convert Source Type**. Select the **Source Type**, `time`. Select the **Conversion to perform**, `int32_from_date`. Click **OK** to perform the conversion.

As shown in the conversion table above, there is no default conversion from `time` to `uint64`. However, there is a default conversion from `int32` to `uint64`, and there is an explicit (with `modify`) conversion from `int32` to `uint64`. Therefore, you can effect a default conversion to convert field `t` from type `time` to type `uint64`. To do so, use the **Rename/Conversion** dialog box, described in Step 4. above. Enter the **Source Field Name**, `t`. Enter the **Dest Field Name** of your choice. In the **Set Dest Type To** field, enter `uint64`. Do not check **Convert Source Type**. Press **OK**. Orchestrate converts the type from `time` to `int32`, and then from `int32` to `uint64`.

## Data Type Conversion Errors

A error results when any Orchestrate operator is unable to perform a default data type conversion. See “Data Type Conversion Errors and Warnings” on page 5-17 for details on data type conversion errors and warnings and on how to prevent them.



## 4: Orchestrate Data Sets

Orchestrate data sets contain the data processed by an Orchestrate application. Orchestrate operators take data sets as input, process all records of the input data set(s), and write their results to output data sets.

This chapter introduces data sets by defining their structure and the two types of data sets used by Orchestrate. It also explains how to use data sets with operators.

This chapter contains the following sections:

- “Orchestrate Data Sets” on page 4-1
- “Using Visual Orchestrate with Data Sets” on page 4-7
- “Defining a Record Schema” on page 4-16
- “Representation of Disk Data Sets” on page 4-32

---

**Note:** To manage Orchestrate persistent data sets, use the Orchestrate administration utility `orchadmin`, which is described in the *Orchestrate Installation and Administration Manual*.

---

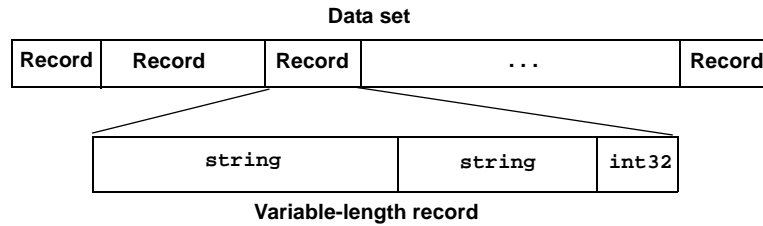
### Orchestrate Data Sets

This section covers data set structure, record schemas, field data types, use of data sets with operators, and the different types of data sets.

#### Data Set Structure

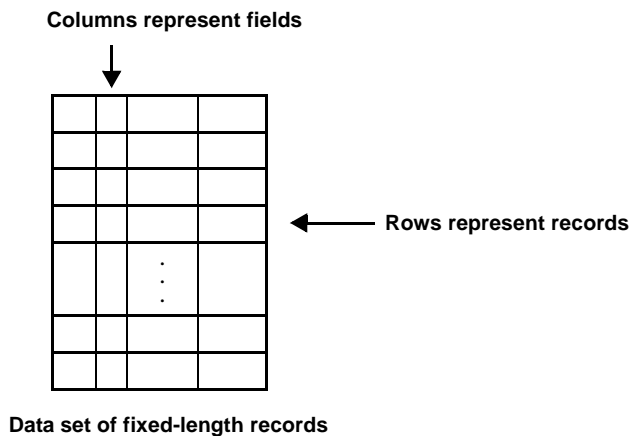
A data set consists of a one-dimensional array (vector) of records. The fundamentals of record-based data are described in the section “Orchestrate Data Sets” on page 1-5. As described in that chapter, fields of some types, such as `int8`, are of fixed length. Fields of other types, such as `string`, can be of variable length. A record that defines one or more variable-length fields is a *variable-length record*.

The following figure shows a sample data set and the format of its variable-length record definition, which contains two variable-length fields of type `string`:



An example of an appropriate use of this variable-length record format is a mailing list. In that case, each record in the data set would hold the data for one person. One variable-length string field could contain the person's name, and the second field the address. A 32-bit integer field could contain a key for sorting the records.

Another kind of data set has a fixed-length record layout, similar to a normalized table in a Relational Database Management System (RDBMS). Normalized tables have a regular row and column layout. In the figure below, each row of the table corresponds to a single record, and each column corresponds to the same fixed-length field in every record:



## Record Schemas

This section describes the data set schema, introduced in the section “The Orchestrate Schema” on page 1-6. See the chapter “Orchestrate Data Types” for fundamental information on Orchestrate data types.

A schema describes the prototypical record in a data set or operator interface. A schema consists of a record-level property list and a description of each field in the record. Specific record-level properties are described in the chapter on import/export properties in the *Orchestrate User's Guide: Operators*.

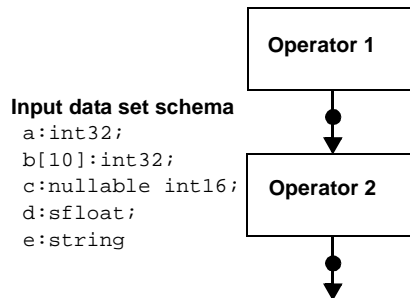


The schema describes each field with the following information:

- An identifier (field name)
- A data type (for some types, parameterized)
- For string and raw fields, an optional length specification
- For a vector field, an optional length
- Field-level properties, such as the nullability specification (see the section “Defining Field Nullability” on page 4-19)

A fundamental use of a record schema is to describe the data for import into an Orchestrate data set. Orchestrate's record schema can describe the data layout of any RDBMS table, and it supports most COBOL records formats, including repeating substructures. See the chapter on the import/export utility in the *Orchestrate User's Guide: Operators* for more information on import and export.

In the following figure, a data set's record schema is shown in a fragment of a data-flow diagram:



In this figure, the record schema for the input data set consists of five fields:

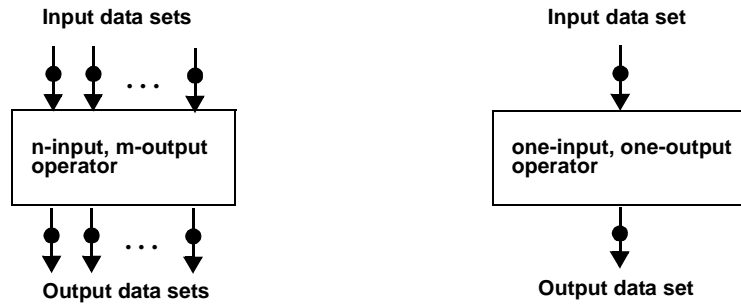
- a: A 32-bit integer
- b: A 10-element vector of 32-bit integers
- c: A nullable 16-bit integer
- d: A single-precision floating point value
- e: A variable-length string

See the section “Defining a Record Schema” on page 4-16 for more information on defining schemas.

## Using Data Sets with Operators

Orchestrate operators, introduced in the section “Orchestrate Operators” on page 1-9, can take data sets or data files as input, and can produce data sets or data files as output. Some operators can also use RDBMS tables as input or output. For extensive information on using Orchestrate operators, see the chapter “Orchestrate Operators”.

Some operators can take multiple input and output data sets (*n-input, m-output* operators), as shown in the left-hand data-flow diagram below. Other operators are limited in the number of input and/or output data sets they handle; the right-hand diagram below shows a *one-input, one-output* operator:



---

**Note:** Within a single step, you cannot use a single data set (either virtual or persistent) as both input and output for the same operator.

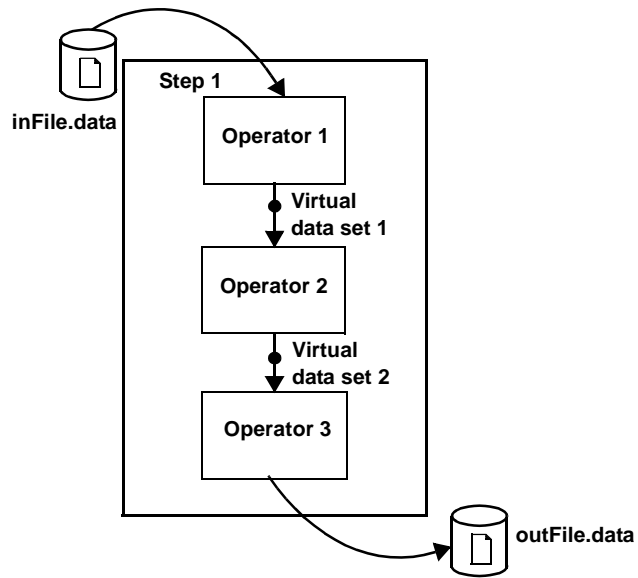
---

For the number of input and output data sets allowed for each Orchestrate operator, see the appropriate chapter in the *Orchestrate User's Guide: Operators*.

## Using Virtual Data Sets

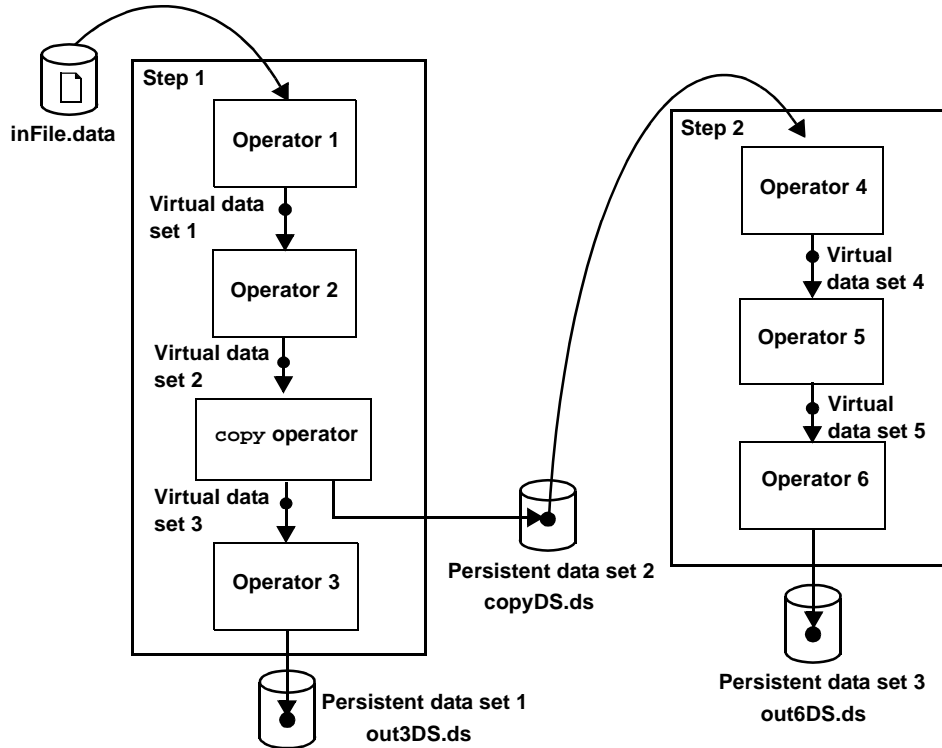
Orchestrate uses virtual data sets to temporarily store data that is output by an operator and input by another operator. Virtual data sets exist only within a step, to connect the output of one operator to the input of another operator in the step. You cannot use virtual data sets to connect operators in different steps. A virtual data set does not permanently buffer or store the data to disk. Virtual data sets are created and processed by one step and then destroyed when that step terminates.

The following data-flow model shows a step that uses two virtual data sets to connect three operators:



## Using Persistent Data Sets

Persistent data sets are stored to a disk file, so that the data processed by a step is preserved after the step terminates. You can use persistent data sets to share data between two or more steps, as shown below:



Step 1 saves the data set output from Operator 2 and uses it as the input to Step 2. These two steps could be part of a single executable file, or each could be part of a separate executable file.

The example above uses the Orchestrate `copy` operator to create two copies of the data set output of Operator 2: a virtual data set passed to Operator 3 and a persistent data set used as input to Operator 4. The `copy` operator takes a single data set as input and produces any number of copies of the input data set as output.

---

**Note:** A persistent data set cannot serve as both an input and an output in a single step. The reason for this restriction is that a file cannot be open simultaneously for reading and for writing.

---

## Importing Data into a Data Set

In many Orchestrate applications, the first operation is to read data from a disk file and to convert the data to an Orchestrate data set. Then, Orchestrate can begin processing the data in parallel, including partitioning the data set. When processing is complete, your application can export the data set to a disk file in the same format as the input file for the application.

The Orchestrate import/export utility lets you import a data file into Orchestrate as a data set and export a data set to a file. See the chapters on the `import` and `export` operators in the *Orchestrate User's Guide: Operators* for more information.

## Partitioning a Data Set

Partitioning divides a data set into multiple pieces, or *partitions*. Each processing node in your system then performs an operation in parallel on an individual partition of the data set rather than on the entire data set, resulting in much higher throughput than a using a single-processor system.

To implement partitioning, Orchestrate divides a data set by records. See the chapter “Partitioning in Orchestrate” for more information on how Orchestrate partitions data sets.

## Copying and Deleting Persistent Data Sets

Orchestrate represents a single data set as multiple files on multiple processing nodes. Therefore, you cannot use the standard UNIX commands `rm` to delete or `cp` to copy a persistent data set. See the section “Representation of Disk Data Sets” on page 4-32 for more information on data set representation.

Orchestrate provides the `orchadmin` utility to manipulate data sets. This utility recognizes the layout of a persistent data set and accesses all data files to copy or delete the persistent data set. See the *Orchestrate Installation and Administration Manual* for a detailed discussion of `orchadmin`.

## Using Visual Orchestrate with Data Sets

This section describes how to use Visual Orchestrate to manipulate both persistent and virtual data sets. Included below are the following sections:

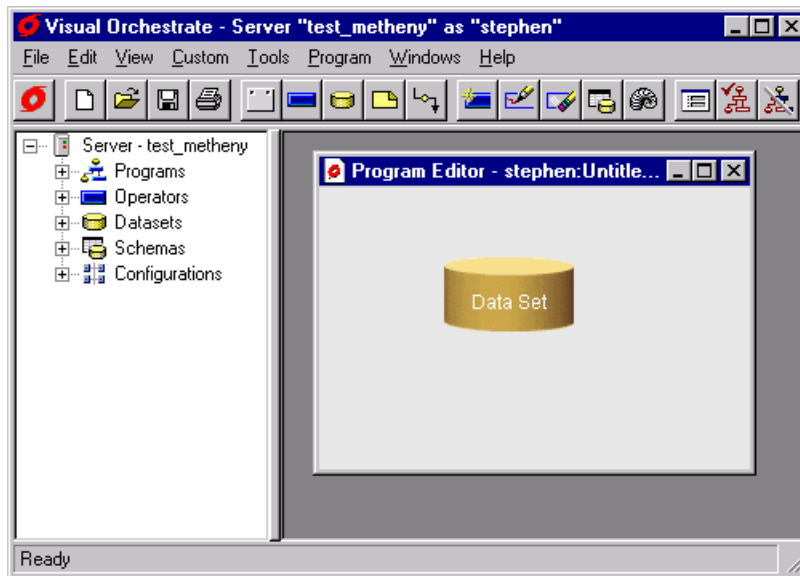
- “Working with Persistent Data Sets” on page 4-8
- “Working with Virtual Data Sets” on page 4-12

## Working with Persistent Data Sets

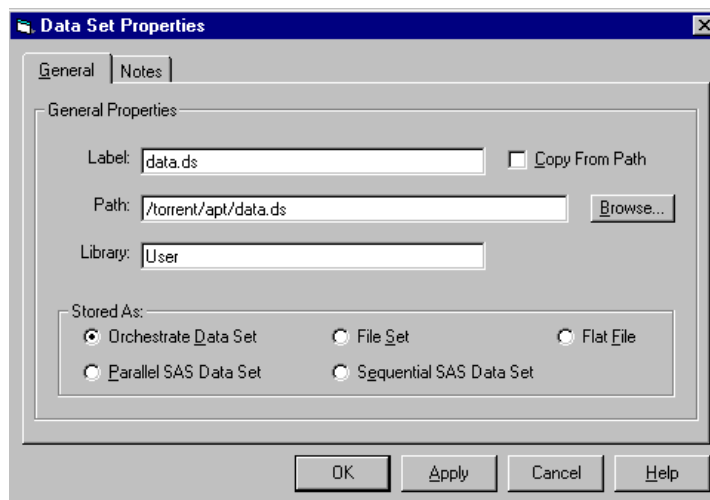
To create a persistent data set in the **Program Editor**, you can either:

- Choose **Program -> Add Data Set** from the menu.
- Click the data set icon in the tool bar.
- With the cursor in a **Program Editor** window, click the right mouse button and select **Add Data Set** from the popup menu.

The data set icon appears in the **Program Editor** window, as shown below:



Double click the data set icon to open the **Data Set Properties** dialog box, as shown below:



Use this dialog box to specify the following:

The **Label** of the data set in the **Program Editor** window.

The **Pathname** of the data set's descriptor file. The descriptor file contains a list of all the individual data files, stored across your parallel machine, that contain the data set records.

See the section “Representation of Disk Data Sets” on page 4-32 for more information on data set layout.

The **Library** name for the data set. The library corresponds to the entry in the **View Window**, under **Data Set**, containing the data set name.

**Stored As.** Choose the representation of the data set as either an **Orchestrate Data Set** (default), a **File Set**, a **Flat File**, or either of the two SAS representations. See the section on the `import` and `export` operators in the *Orchestrate User's Guide: Operators* for more information on representations of data sets.

After you create the data set, you create a link to connect the data set to an operator as either an input or output data set. This procedure is described in the following two sections:

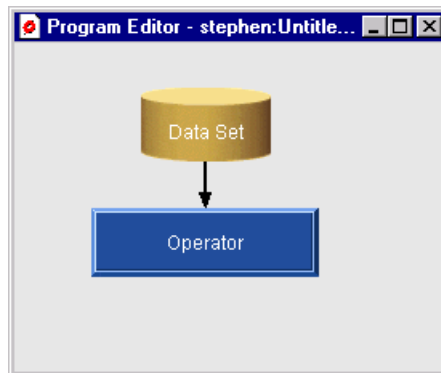
- “Connecting a Persistent Data Set as Input” on page 4-9
- “Connecting a Persistent Data Set as Output” on page 4-11

## Connecting a Persistent Data Set as Input

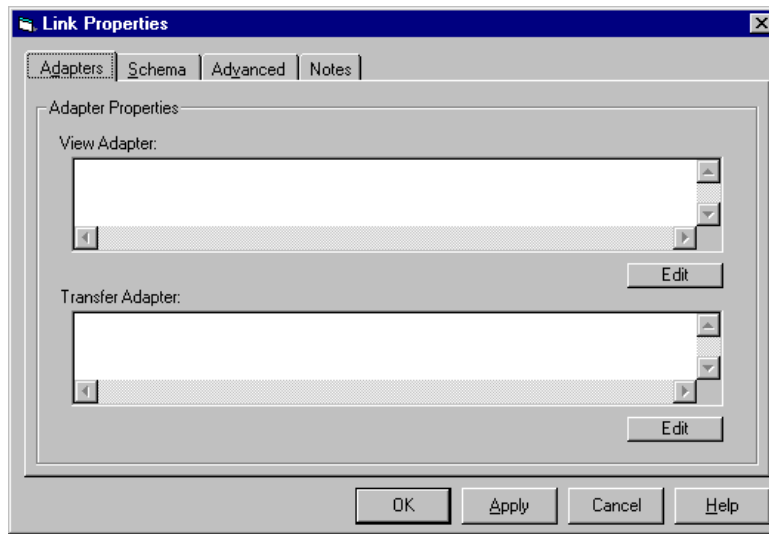
To connect a persistent data set as an input to an operator, create a link and attach one end of the link to the data set and the other end of the link to the input of an operator. To create a link, you can do any one of the following:

- Choose **Program -> AddLink** from the menu.
- Click the link icon in the tool bar.
- With the cursor in a **Program Editor** window, click the right mouse button and select **Add Link** from the popup menu.

The following figure shows a data set connected to an operator by a link:



Double click on the link to perform any optional configuration for the link. The **Link Properties** dialog box for an input data set is shown below:



Use this dialog box to specify the following:

**Adapters** allows you to specify a view adapter or transfer adapter on the data set. See the chapter on the `modify` operator in the *Orchestrate User's Guide: Operators* for more information.

**Schema** allows you to define the record schema when the input data set represents a **Flat File**. See the chapter on the import/export utility in the *Orchestrate User's Guide: Operators* for more information on data representation.

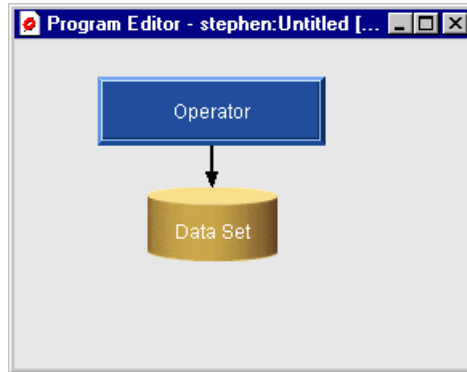
**Advanced** allows you to enter options for buffering. See the *Orchestrate Installation and Administration Manual* for more information on these options.

**Notes** allows you to enter optional text that describes the link. The text is saved with the link. This tab provides standard text editing functions, such as selection, cutting, and pasting; right-click to display a context menu of these functions.

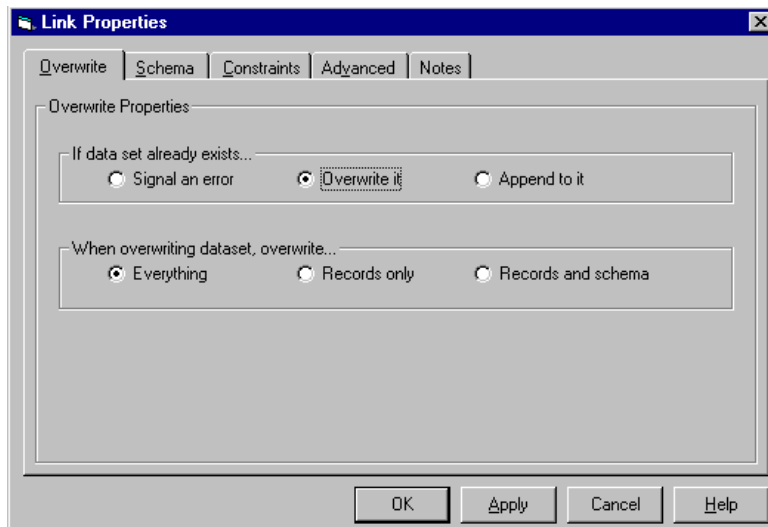


## Connecting a Persistent Data Set as Output

To connect the data set as an output, create a link and attach one end of the link to the data set and the other end of the link to the output from an operator.



Double click on the link to perform any optional configuration for the link. The **Link Properties** dialog box for an output data set is shown below:



This dialog box contains the following tabs:

**Overwrite** tab to control Orchestrate's action when the output data set already exists. By default, Orchestrate signals an error and aborts your application when writing to an existing data set. This prevents you from accidentally overwriting data.

You can accept the default of **Signal an error**, choose to **Overwrite it**, or choose to **Append** data to it. See the next section for more information on append.

**Schema** tab to explicitly define the record schema for the output data set.

**Constraints** tab to control where the data set is written on your system. See the chapter "Constraints" for more information.

**Advanced** tab for setting preserve-partitioning flag and buffering options. See the section “The Preserve-Partitioning Flag” on page 8-11 for more information on the preserve-partitioning flag.

**Notes** tab for entering optional text that describes the link. The text is saved with the link. This tab provides standard text editing functions, such as selection, cutting, and pasting; right-click to display a context menu of these functions.

### Appending Data to an Existing Data Set

Orchestrate allows you to append data to an existing data set. The appending operator must execute in a separate step from the operator performing the write that first created the data set. In addition, records appended to an existing data set must have an identical record schemas as the existing data set.

When a persistent data set is created and first written to disk, the number of partitions it contains is equal to the number of instances of the operator that creates it. When you later append data to the data set, the number of processing nodes on which the appending operator executes may be different from the number of partitions in the existing data set. Also, the data files containing the appended records can reside on disk drives other than those holding the original data.

For example, if a data set was originally created with eight partitions, and is appended to by an operator executing on four processing nodes, Orchestrate repartitions the appending data to store it as eight partitions.

The state of a data set's repartitioning flag is saved to disk along with the records of the data set. To prevent Orchestrate from repartitioning the data set, you can set the repartitioning flag to *preserve*. Then, when you append data to that data set, Orchestrate creates one instance of the appending operator for each partition of the data set.

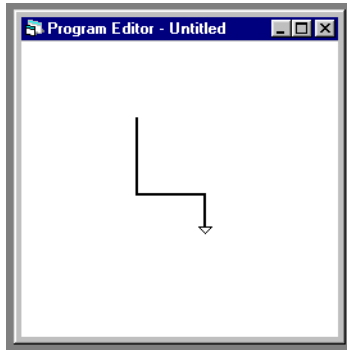
## Working with Virtual Data Sets

Virtual data sets connect the output of one operator to the input of another operator. In Visual Orchestrate, you use a link to implement a virtual data set.

To create a link, you can either:

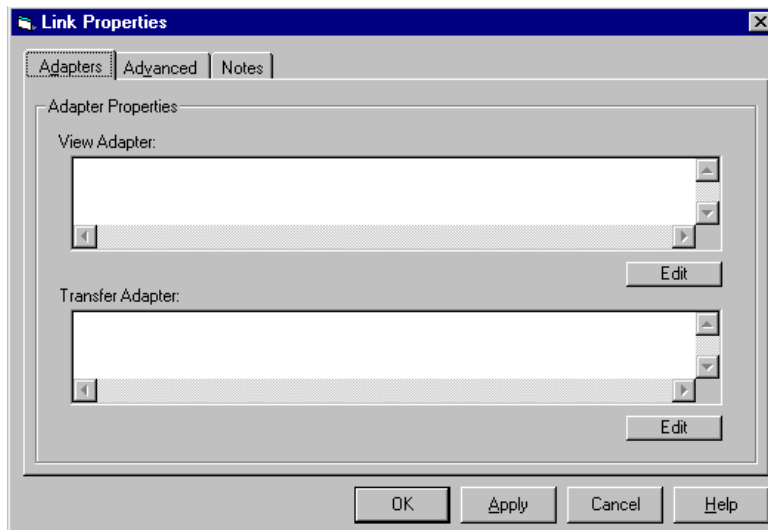
- Choose **Program -> Add Link** from the menu.
- Click the link icon in the tool bar.
- With the cursor in a **Program Editor** window, click the right mouse button and select **Add Link** from the popup menu.

The link icon for the virtual data set appears in the **Program Editor** window, as shown below:



Before you configure the link, connect it to the output of one operator and to the input of another operator. You can then perform optional configuration on the link.

Double click on the link icon to open the **Link Properties** dialog box. This dialog box is shown below:



This dialog box contains the following tabs:

**Adapters** tab for specifying a view adapter or transfer adapter on the data set. See the chapter on the `modify` operator in the *Orchestrate User's Guide: Operators* for more information.

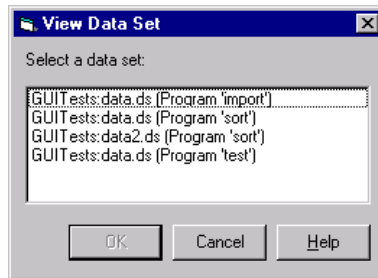
**Advanced** tab for setting preserve-partitioning flag and buffering options. See the section "The Preserve-Partitioning Flag" on page 8-11 for more information on the preserve-partitioning flag.

**Notes** tab for entering optional text that describes the link. The text is saved with the link. This tab provides standard text editing functions, such as selection, cutting, and pasting; right-click to display a context menu of these functions.

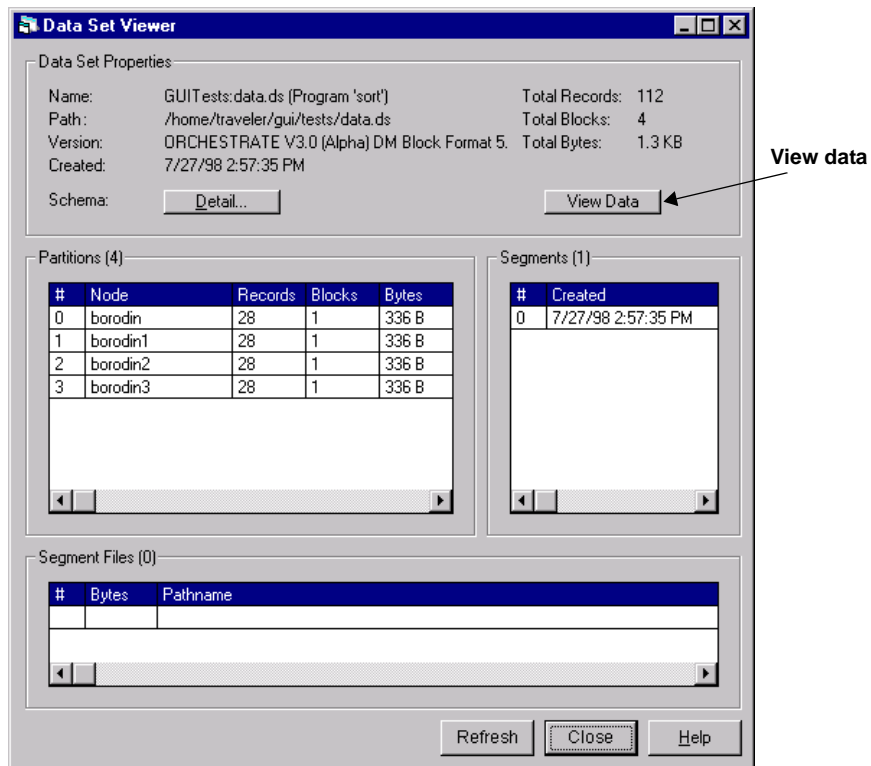
## Using the Data Set Viewer

The Visual Orchestrate **Data Set Viewer** lets you display information about an Orchestrate persistent data set. You access the utility by choosing the menu command **Tools -> Data Set Viewer**.

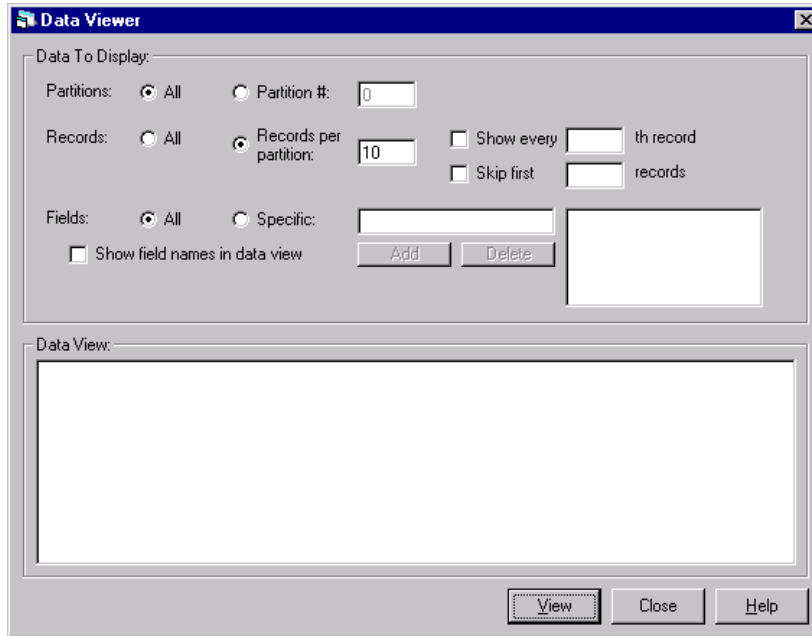
1. Choose **Tools -> Data Set Viewer** from the Visual Orchestrate menu. This opens the following dialog box that you use to select a persistent data set you want information on:



2. Choose the data set and click **OK** to open the following dialog box displaying information about the data set:

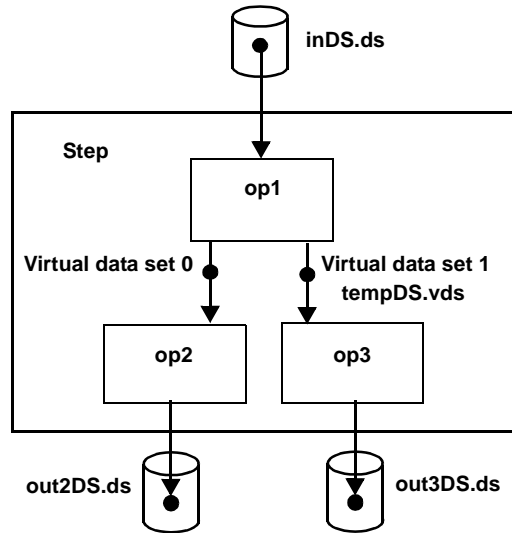


3. You can dump data set records by clicking on the **View Data** button. This opens the following dialog box:



Specify the records you want to display, and then click the **View** button.

## Obtaining the Record Count from a Persistent Data Set



You may on occasion need to determine the count of the number of records in a persistent data set. Orchestrate provides the UNIX command-line utility `dsrecords` that returns this count.

You execute `dsrecords` from the UNIX command line using the Visual Orchestrate **Shell Command**. Shown below is the syntax for `dsrecords`:

```
dsrecords ds_name
```

where `ds_name` specifies the pathname of a persistent data set.

This output of `dsrecords` appears in the **Execution Window** is shown below:

```
Job started.
4/9/98 6:14:36 PM 0 TUSV 001013 Inform : 24 records

Job finished, status = OK.
```

## Defining a Record Schema

An Orchestrate record schema definition is an ASCII string beginning with the key word `record` and consisting of a sequence of field definition statements enclosed in parentheses and separated by semicolons. For example:

```
record (aField:int32; bField:sfloat; cField:string[]; )
```

This record schema defines three-record fields:

- `afield`: A 32-bit integer field

- `bField`: A single-precision (32-bit) floating-point field
- `cField`: A variable-length character string

## Schema Definition Files

For use with the `import`, `export`, and `generator` operators, you can create a file containing your schema definition. See the chapters for those operators in the *Orchestrate User's Guide: Operators*.

You can include comments in schema definition files. The starting delimiter for a comment is a double slash `//`, and the ending delimiter is a new-line. Note, however, that you cannot use comments in schema definitions that you specify on the command line.

## Field Accessors

You can retrieve information about a field, such as whether it is nullable, by calling functions on its field accessor. For information on declaring and using field accessors, see the *Orchestrate/APT Developer's Guide* for information.

## How a Data Set Acquires Its Record Schema

A data set acquires its record schema in one of the following three ways:

1. Import of the data set from an external representation. Often, the first action of an Orchestrate application is to import data and convert it to a data set. You define the record schema of the resultant data set at the time of the import. See the chapter on `import` in the *Orchestrate User's Guide: Operators* for more information.
2. A write to the data set by an Orchestrate operator. A data set with no schema inherits the output interface schema of the operator that writes to it. This is the most common way for an output data set to obtain a schema. See the section “Output Data Sets and Operators” on page 5-8 for more information.
3. A read from an RDBMS table by an RDBMS operator. On a read operation, Orchestrate converts the table into an Orchestrate data set by reading both the data and the record layout of the table. Orchestrate automatically creates a record schema for the new data set, based on the information from the RDBMS table.

Note that on a write operation to an RDBMS table, Orchestrate converts the record schema of a data set to the table layout of an RDBMS data set. See the *Orchestrate User's Guide: Operators* for specific information on how your RDBMS works with Orchestrate.

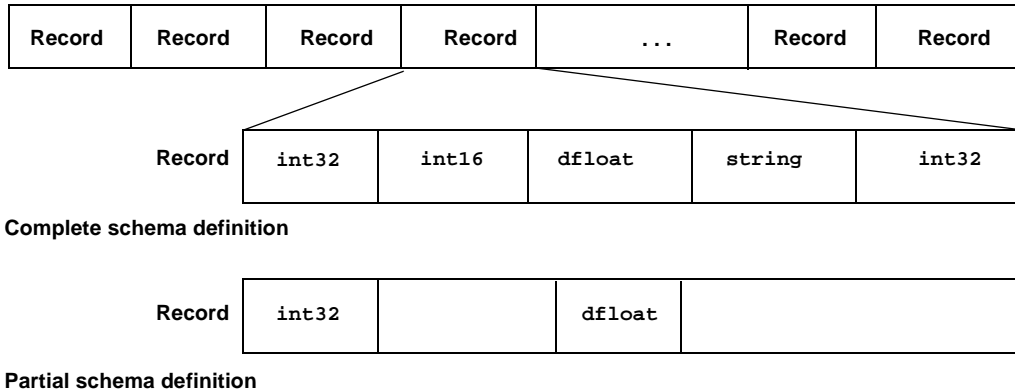
The following sections describe how to create record schemas, name record fields, specify nullability, and set record schema properties for each Orchestrate data type.

## Using Complete or Partial Schema Definitions

When you define the record schema of a data set, you do either of the following:

- Define complete record schemas: You describe every field in the record, including the field's name and data type.
- Partial record schemas: You describe only the components of the record schema needed for processing.

The following figure shows both options:



For a complete schema definition, you define all fields in the record, including the field's name, data type, and nullability. You must define the complete schema if your application needs to access all the fields in the record, where *access* includes the following:

- Reading a field value
- Writing a field value
- Dropping a field and its associated data storage from the record
- Modifying a field's name, data type, or other characteristics

You can use partial schemas if your application will access only some of the data in a record. For example, the partial schema definition in the figure above defines only two fields (the first and third) of a five-field record. An Orchestrate application would be able to access only those defined fields, where *access* includes the following:

- Reading a field value
- Dropping a field definition (but not its associated data storage) from the record
- Modifying the field's name, data type, or other characteristics

Note that a partial schema definition allows you to drop access to a defined field, but does *not* remove its data storage.

The main advantage of a complete record schema is the flexibility to manipulate the records during processing. However, to define a complete schema you must define all record fields in your data, regardless of whether your application will reference all the fields.



Partial schemas allow you to define only the fields that you are interested in processing and to ignore all other data in the records. However, Orchestrate cannot perform some types of manipulations on partial schema definitions, such as dropping unused fields from the record.

See the chapter on the import/export utility in the *Orchestrate User's Guide: Operators* for more information.

## Naming Record Fields

In your Orchestrate record schema definition, all field names must conform to the following conventions:

- The name must start with a letter or underscore (`_`) character.
- The name can contain only alphanumeric and underscore characters.
- The name is case insensitive.

Field names can be any length.

## Defining Field Nullability

If a field is *nullable*, it can hold *null*, which indicates that the field contains no data. In processing a record, an operator can detect a null value and take the appropriate action, such as omitting the null field from a calculation or signalling an error condition.

This section describes two different methods for representing nulls, and how to specify nullability in an Orchestrate record schema.

### Orchestrate Null Representation

Orchestrate uses a single-bit flag to mark a field as null. If the field is a vector, Orchestrate uses a single bit for each element of the vector, so that a nullable vector can contain both valid data elements and null elements. This type of null support is called an *indicated* null representation.

Some other software packages implement null support with an *in-band* representation, which designates as null a specific value, such a numeric field's most negative value. The disadvantage of in-band null representation is that requires the user to treat an in-range value as a null. However, Orchestrate does allow you to process data that uses in-band null representation, by means of the `modify` operator (see the *Orchestrate User's Guide: Operators*).

### Defining Nullability in a Record Schema

In creating a record schema, you mark a field as nullable by inserting the keyword `nullable` immediately before the field's data type. Any field defined without the `nullable` keyword is by default *not* nullable. The following sample record schema defines a nullable `int32` and a nullable `dfloat` vector:

```
record (n:nullable int32; s[10]:nullable dfloat;)
```

Even though a field is by default non-nullable, you can use the `not_nullable` keyword to be explicit, as shown below:

```
record (n:not_nullable int32;)
```

To make every field in a record schema nullable, you can specify `nullable` at the record level, with the following syntax:

```
record nullable (n:int32; m:int16; f:dfloat;)
```

Specifying nullability at the record level affects top-level fields only and does not affect nested fields.

Nullability can be overridden for any field in the record by explicit use of `not_nullable`, as shown below:

```
record nullable (n:int32; m:not_nullable int16; f:dfloat;)
```

## Nullability of Vectors and Aggregates

Vectors are nullable, and you can check for a null in each element of a nullable vector.

An aggregate (tagged or subrecord) is not itself nullable. However, you can specify nullability for the fields of an aggregate that have types that are nullable. Nullability and aggregate is further discussed in the section “Vectors and Aggregates in Schema Definitions” on page 4-24.

## Checking Null and Nullability

To determine whether a field is nullable, you must check its nullability property. To check whether a field is nullable, or whether a nullable field currently contains null, you must first declare an accessor for the field; see the *Orchestrate/APT Developer's Guide* for information. Field accessors can also provide other information, such as the length of a variable-length vector.

## Using Value Data Types in Schema Definitions

This section describes how to use the Orchestrate value data types in record schema definitions, with a section for each data type. Each section includes examples of field definitions, and some sections describe required and optional properties for fields of the data type. Vectors are further described in the section “Vectors and Aggregates in Schema Definitions” on page 4-24.

### Date Fields

You can include date fields, in a record schema, as shown in the following examples:

```
record (dateField1:date; )           // single date
record (dateField2[10]:date; )     // 10-element date vector
```

```
record (dateField3[:date; ]           // variable-length date vector
record (dateField4:nullable date;) // nullable date
```

## Decimal Fields

You can include decimal fields in a record schema. To define a record field with data type `decimal`, you must specify the field's precision, and you may optionally specify its scale, as follows:

```
fieldname:decimal[precision, scale];
```

where:

- $1 \leq \textit{precision}$  (no maximum)
- $0 \leq \textit{scale} < \textit{precision}$

If the scale is not specified, it defaults to zero, indicating an integer value.

Examples of decimal field definitions:

```
record (dField1:decimal[12]; )           // 12-digit integer
record (dField2[10]:decimal[15,3]; ) // 10-element decimal vector
record (dField3:nullable decimal[15,3];) // nullable decimal
```

## Floating-Point Fields

You can include floating-point fields in a record schema. To define floating-point fields, you use the `sfloat` (single-precision) or `dfloat` (double-precision) data type, as in the following examples:

```
record (aSingle:sfloat; aDouble:dfloat; ) // float definitions
record (aSingle: nullable sfloat;) // nullable sfloat
record (doubles[5]:dfloat;) // fixed-length vector of dfloats
record (singles[:sfloat;]) // variable-length vector sfloats
```

## Integer Fields

You can include integer fields in a record schema. To define integer fields, you use an 8-, 16-, 32-, or 64-bit integer data type (signed or unsigned), as shown in the following examples:

```
record (n:int32;) // 32-bit signed integer
record (n:nullable int64;) // nullable, 64-bit signed integer
record (n[10]:int16;) // fixed-length vector of 16-bit signed integer
record (n[:uint8;]) // variable-length vector of 8-bit unsigned int
```

## Raw Fields

You can define a record field that is a collection of untyped bytes, of fixed or variable length. You give the field data type `raw`. You can also specify a byte alignment value on a raw field, to satisfy requirements of your system architecture or to optimize the speed of data access.

The definition for a `raw` field is similar to that of a string field, as shown in the following examples:

```

record (var1:raw[];) // variable-length raw field
record (var2:raw;) // variable-length raw field; same as raw[]
record (var3:raw[40];) // fixed-length raw field
record (var4[5]:raw[40];) // fixed-length vector of raw fields
// variable-length raw aligned on 8-byte boundary:
record (var5:raw[align = 8];)
// vector of fixed-length raw fields aligned on 4-byte boundary:
record (var6[5]:raw[align = 4, length = 20];)

```

### Variable-Length Raw Fields

When an Orchestrate operator writes to a variable-length raw field, it determines the field length and updates the field's length prefix. When an operator reads from a variable-length raw field, it first reads the length prefix to determine the field's length.

You can specify the maximum number of bytes allowed in the raw field with the optional property `max`, as shown in the example below:

```
record (var7:raw[max=80];)
```

If an application attempts to write more than `max` bytes to a raw field, Orchestrate writes only the first `max` bytes.

### Fixed-Length Raw Fields

The length of a fixed-length raw field must be at least 1.

## String Fields

You can define string fields of fixed or variable length. For variable-length strings, the string length is stored as part of the string as a hidden integer. The storage used to hold the string length is not included in the length of the string.

In a data set with a variable-length string field, each record of the data set can contain a different length string. When an operator writes to the field, Orchestrate determines the string length and updates the field's hidden length integer. When an operator reads from a variable-length field, Orchestrate first reads the length integer to determine the field's length.

The following examples show `string` field definitions:

```

record (var1:string[];) // variable-length string
record (var2:string;) // variable-length string; same as string[]
record (var3:string[80];) // fixed-length string of 80 bytes
record (var4:nullable string[80];) // nullable string
record (var5[10]:string;) // fixed-length vector of strings
record (var6[]:string[80];) // variable-length vector of strings

```

### Variable-Length String Fields

For variable-length string fields, you can include the parameter `max` to specify the maximum length of the field in bytes. Shown below is an example using this parameter:

```
record (var7:string[max=80];)
```

When a record containing a string field with a specified maximum length is created, the length of the string is zero, as it is for normal variable-length strings. Writing data to the string field with fewer bytes than the maximum sets the length of the string to the number of bytes written. Writing a string longer than the maximum length truncates the string to the maximum length.

### Fixed-Length String Fields

The length of a fixed-length string field must be at least 1.

You can use the optional property `padchar` to specify the character for unassigned elements in fixed-length string fields. The `padchar` property has the following syntax:

```
padchar = int_val | ASCII_char | null
```

where:

- `int_val` is an integer in the range 0 - 255, that is the ASCII value of the pad character.
- `ASCII_char` is the pad character as a single ASCII character.
- `null` (default) specifies a value of 0 as the pad character (same as specifying `padchar = 0`).

The following example shows use of `padchar` in a field definition:

```
record (var8:string[80, padchar = ' '];) // ASCII space padchar (0x20)
```

If an application wrote fewer than 80 characters to the `var8` field defined in this example, `var8` will be padded with the space character to the full length of the string.

Note that the Orchestrate `export` operator uses the specified `padchar` to pad a fixed-length string that is exported. See the section on the `import/export` utility in the *Orchestrate User's Guide: Operators* for more information.

### Time Fields

You can include time fields in a record schema. By default, the smallest unit of measure for a time value is seconds, but you can instead use microseconds with the `[microseconds]` option. The following are examples of time field definitions:

```
record (tField1:time; ) // single time field in seconds
record (tField2:time[microseconds];)// time field in microseconds
record (tField3[:time; ) // variable-length time vector
record (tField4:nullable time;) // nullable time
```

### Timestamp Fields

Timestamp fields contain both time and date information. In the time portion, you can use seconds (the default) or microseconds for the smallest unit of measure. For example:

```
record (tsField1:timestamp;)// single timestamp field in seconds
record (tsField2:timestamp[microseconds];)// timestamp in microseconds
record (tsField3[15]:timestamp;)// fixed-length timestamp vector
record (tsField4:nullable timestamp;)// nullable timestamp
```

## Vectors and Aggregates in Schema Definitions

This section describes how to use the Orchestrate vectors and the two aggregate data types, subrecords and tagged aggregates, in record schema definitions. It also describes how to reference vector elements and fields in aggregates.

### Vector Fields

Orchestrate records can contain one-dimensional arrays, or *vectors*, of fixed or variable length. You define a vector field by following the field name with brackets `[ ]`. For a variable-length vector, you leave the brackets empty, and for a fixed-length vector you put the number of vector elements in the brackets. For example, to define a variable-length vector of `int32`, you would use a field definition such as the following:

```
intVec[]:int32;
```

To define a fixed-length vector of 10 elements of type `sfloat`, you would use a definition such as:

```
sfloatVec[10]:sfloat;
```

### Data Types for Vectors

You can define a vector of any Orchestrate value data type, including `string` and `raw`. You cannot define a vector of a vector or tagged aggregate type. You can, however, define a vector of type subrecord, and you can define that subrecord to include a tagged aggregate field or a vector. For more information on defining subrecords, see the section “Subrecord Fields” on page 4-25.

### Numbering of Elements

Orchestrate numbers vector elements from 0. For example, the third element of a vector has the index number 2. This numbering scheme applies to variable-length and fixed-length vectors.

### Referencing Vector Elements

To reference an element of a vector field of fixed or variable length, you use indexed addressing of the form `vField[eleNum]`. Remember that element numbering starts with 0. For example, suppose that you have defined the following a vector field:

```
vInt[10]:int32;
```

To reference the third element of that field, you use `vInt[2]`.

### Nullability of Vectors

You can make vector elements nullable, as shown in the following record definition:

```
record (vInt[]:nullable int32;
       vDate[6]:nullable date; )
```

In the example above, every element of the variable-length vector `vInt` will be nullable, as will every element of fixed-length vector `vDate`.

To test whether a vector of nullable elements contains no data, you must check each element for null.

## Subrecord Fields

Record schemas let you define nested field definitions, or *subrecords*, by specifying the type `subrec`. A subrecord itself does not define any storage; instead, the fields of the subrecord define storage. The fields in a subrecord can be of any data type, including `tagged`.

The following example defines a record that contains a subrecord:

```
record ( intField:int16;
        aSubrec:subrec (
            aField:int16;
            bField:sfloat; );
        )
```

In this example, the record contains a 16-bit integer field, `intField`, and a subrecord field, `aSubrec`. The subrecord includes two fields: a 16-bit integer and a single-precision float.

### Referencing Subrecord Fields

To reference fields of a subrecord, you use *dot* addressing, following the subrecord name with a period (`.`) and the subrecord field name. For example to refer to the first field of the `aSubrec` example shown above, you use `aSubrec.aField`.

### Nullability of Subrecords

Subrecord fields of value data types (including `string` and `raw`) can be nullable, and subrecord fields of aggregate types can have nullable elements or fields. A subrecord itself cannot be nullable.

### Vectors of Subrecords

You can define vectors (fixed-length or variable-length) of subrecords. The following example shows a definition of a fixed-length vector of subrecords:

```
record (aSubrec[10]:subrec (
        aField:int16;
        bField:sfloat; );
        )
```

### Nested Subrecords

You can also nest subrecords and vectors of subrecords, to any depth of nesting. The following example defines a fixed-length vector of subrecords, `aSubrec`, that contains a nested variable-length vector of subrecords, `cSubrec`:

```
record (aSubrec[10]:subrec (
        aField:int16;
        bField:sfloat;
        cSubrec[:subrec (
            cAField:uint8;
            cBField:dfloat; );
        );
        )
```

To reference a field of a nested subrecord or vector of subrecords, you use the dot-addressing syntax `<subrec>.<nSubrec>.<srField>`. To reference `cAField` in the sample subrecord definition above, you would use `aSubrec.cSubrec.cAField`.

**Subrecords Containing Tagged Aggregates**

Subrecords can include tagged aggregate fields, as shown in the following sample definition:

```
record (aSubrec:subrec (
    aField:string;
    bField:int32;
    cField:tagged (
        dField:int16;
        eField:sfloat;
    );
);
)
```

In this example, `aSubrec` has a `string` field, an `int32` field, and a tagged aggregate field. The tagged aggregate field `cField` can have either of two data types, `int16` or `sfloat`.

To reference a field of a tagged aggregate field of a subrecord, you use the dot-addressing syntax `<subrec>.<tagged>.<tfield>`. To reference `dField` in the sample subrecord definition above, you would use `aSubrec.cField.dField`.

**Tagged Aggregate Fields**

You can use schemas to define tagged aggregate fields (similar to C unions), with the data type `tagged`. Defining a record with a tagged aggregate allows each record of a data set to have a different data type for the tagged field. When your application writes to a field in a tagged aggregate field, Orchestrate updates the tag, which identifies it as having type of the field that is referenced.

The data type of a tagged aggregate subfields can be of any Orchestrate data type except `tagged` or `subrec`. For example, the following record defines a tagged aggregate field:

```
record ( tagField:tagged (
    aField:string;
    bField:int32;
    cField:sfloat;
);
)
```

In the example above, the data type of `tagField` can be one of following: a variable-length string, an `int32`, or an `sfloat`.

**Referencing Subfields in Tagged Aggregates**

In referring to subfields of a tagged aggregate, you use the dot-addressing syntax, `<tagged>.<tagField>`. For example to refer to the first field of the `tagField` example shown above, you use `tagField.aField`.

Internally, Orchestrate assigns integer values to the aggregate fields, starting with 0. In the example above, the tag value of `aField` is 0; `bField`, 1; and `cField`, 2.



### Nullability of Tagged Aggregates

Tagged aggregate fields of value data types (including `string` and `raw`) can be nullable, and subrecord fields of aggregate types can have nullable elements or fields. A tagged aggregate field itself cannot be nullable.

## Default Values for Fields in Output Data Sets

Input data sets are read-only, and Orchestrate treats the record fields of an input data set as if they contain valid information. Record fields in an output data set have read/write access, and Orchestrate gives them default values upon creation.

If the field is nullable, Orchestrate sets it to null. Otherwise, Orchestrate assigns a default value based on the field's data type, as follows:

- All integers = 0
- `dfloat` or `sfloat` = 0
- `date` = January 1, 0001
- `decimal` = 0
- `time` = 00:00:00 (midnight)
- `timestamp` = 00:00:00 (midnight) on January 1, 0001
- Length of a variable-length `string` or `raw` field = 0
- Length of a variable-length vector = 0
- Characters of a fixed-length string = null (0x00), or if specified, the `padchar` value
- Bytes of a fixed-length `raw` = 0
- Elements of a fixed length vector are assigned a default value in accordance with their nullability and data type.
- Tag on a tagged aggregate = 0, indicating the first field of the aggregate. The field is assigned a default value in accordance with its nullability and data type.

## Using the Visual Orchestrate Schema Editor

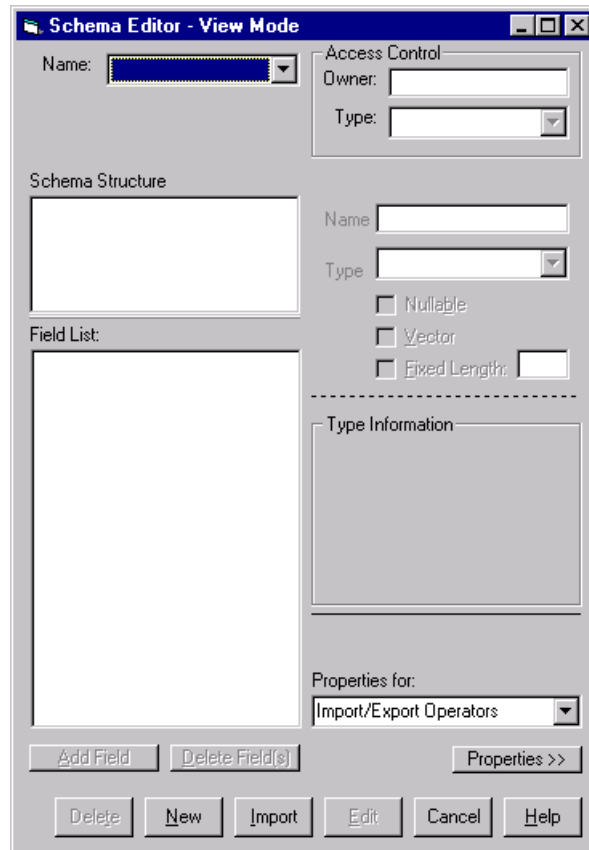
You create record schemas for importing or exporting data and for defining custom Orchestrate operators. To create record schemas, you use the Orchestrate Schema Editor.

To access the Schema Editor, do any of the following:

- Choose **Tools->Schema Editor** from the Visual Orchestrate menu.
- Click the Schema Editor icon in the Visual Orchestrate tool bar.
- From the **Custom Operator** dialog box, do either of the following:
  - For a native operator, open the Schema Editor from the **Interfaces** tab (see the section “Example: Specifying Input and Output Record Schemas” on page 13-19).

- For a UNIX command operator, open the Schema Editor from the **Input** or **Output** tab (see the section “Specifying the Interface Schema” on page 12-10).

Shown below is the Schema Editor dialog box as it appears when you open it from the **Tools** menu or from the tool bar. If you open it from the **Custom Operator** dialog box, the **Schema Structure** area is instead labeled **Interface Structure**.



This dialog box contains the following areas:

**Name** (at top): To view or edit an existing schema, select a schema name from this pull-down list.

**Access Control:** Specify the schema **Owner** and the access **Type** for other users. Options for **Type** are the following:

**Public Write** (default): Anyone can read or write the schema.

**Public Read:** Anyone can read the schema; only the owner can modify it.

**Private:** Only the program owner can read or write the schema.

**Schema Structure** (or **Interface Structure**): Shows the record level properties of the schema. All aggregate fields (tagged and subrecord) are shown here as well. To access an aggregate field, click it in the **Schema Structure** window. The aggregate's component fields appears below in the **Field List** window.

**Field List:** Shows all fields defined for the record or aggregate selected in the **Schema Structure** window. The display includes the field's name, data type, and optional properties.

**Name:** Specify the name of a field.

**Type:** Select the field's data type from the pull-down list.

**Type Information:** Set the parameters for the field's type; the applicable parameters appear below the type name. See the section "Defining a Record Schema" on page 4-16 for details on type parameters for schemas.

**Properties:** Select **Import/Export** or **Generator** from the **Properties for** list, and click the **Properties>>** button. This action opens an additional Schema Editor area, entitled either **External Data Layout Properties** (for import/export) or **Generator Operator Properties**. To view or specify properties at the record level, click **Record** in the **Schema Structure** area. To view or specify properties for a individual field, click the field in the **Field List**. For more information on import/export and on the generator operator, see the *Orchestrate User's Guide: Operators*.

After opening the Schema Editor, you can create a new record schema or edit an existing one, as described in the following sections:

- "Creating a New Record Schema" on page 4-29
- "Creating a New Record Schema from an Existing Schema" on page 4-30
- "Editing a Record Schema" on page 4-30
- "Creating an Aggregate Field" on page 4-30

## Creating a New Record Schema

To create a new record schema:

1. Open the Schema Editor.
2. Click the **New** button.
3. Choose **Named** or **Unnamed**. Unnamed (labeled `local`) schemas are available only when you use the Schema Editor to configure the interface schema for a custom operator. When you invoke the Schema Editor from the Visual Orchestrate menu or toolbar, you can view and create only named schemas.

A **Named** schema is accessible to anyone connected to the server. To access a named schema, double-click its name in the **Server View Area** of Visual Orchestrate.

An **Unnamed** schema is available only to the schema creator, for configuring a custom operator. If you want reference the schema by name and make it available to other users, you must name the schema.

4. (Optional) Specify a **Library** name used to store the record schema. When you store a record schema, it will be saved under **Schema** -> `library_name`. You can edit the schema by double-clicking it in the **Server View Area** of Visual Orchestrate.
5. Select **Record** in the **Schema Structure** area of the dialog box, to specify record-level properties for the schema. You view and enter these properties in the **External Data Layout Properties** or **Generator Operator Properties** area.
6. Click in the **Field List** area to add fields to the schema.

7. Click **Add Field** to add a new field.  
**Add Field** adds a field before the currently selected record field. If no field is selected, the new field is added to the end of the schema. By default, the data type of the field is `int32`.
8. Define the fields of the record schema, including:
  - **Name** (required)
  - **Vector** (by default, vector is variable length) and optional **Fixed Length**
  - **Nullability**
  - **Type**
  - **Type Information**
  - **Properties** for import or export
9. Continue to add all fields
10. Click **Save** to save the record schema.

### Creating a New Record Schema from an Existing Schema

To create a new schema from an existing one:

1. Open the Schema Editor.
2. Select an existing record schema from the **Name** drop down list.
3. Press **New** to create the new schema based on the selected schema from Step 2.
4. Edit the record schema.
5. Click **Save** to save the record schema.

### Editing a Record Schema

To edit an existing record schema:

1. Open the Schema Editor.
2. Select an existing record schema from the **Name** drop down list.
3. Press **Edit** to edit the schema or **New** to create a new schema based on the selected schema from Step 2.
4. Edit the record schema.
5. Click **Save** to save the record schema.

### Creating an Aggregate Field

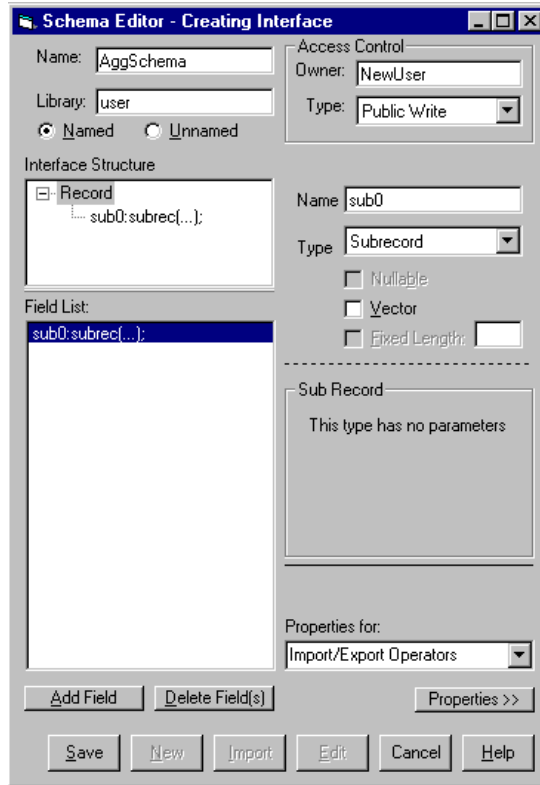
An aggregate field, either a subrecord or a tagged, contains nested field definitions. This section describes how to create an aggregate field as part of a record schema.

Use the following procedure to create either a subrecord or tagged field:

1. Start the Schema Editor and define the name of the schema.
2. Click **Add Field** to add an aggregate field

- Specify the **Name** and **Type** of the aggregate. The type of an aggregate is either **Subrecord** or **Tagged Subrecord**.

Once you have defined the **Name** and **Type**, it appears in the **Schema Structure** window under the word **Record**. Shown below is an example of a subrecord named `sub0`:

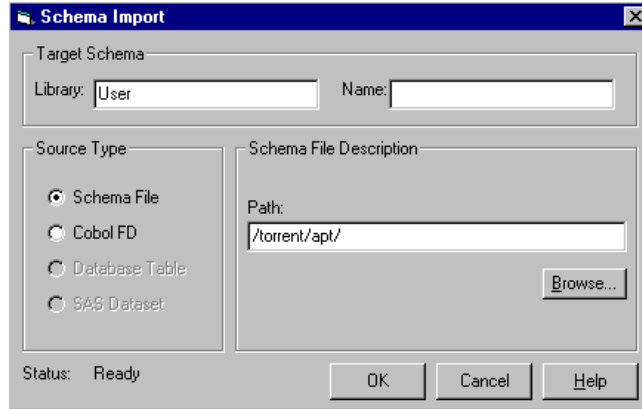


- To add fields to the aggregate, select the aggregate in the **Schema Structure** window. The **Field List** window lists all fields defined within the aggregate. Initially, the **Field List** window is blank.
- Add new fields to the subrecord as described above for adding fields to a record schema.
- Click **Save** to save the record schema.

## Importing a Record Schema

Visual Orchestra lets you create a schema by importing the schema definition from a text file or from a COBOL FD definition. To import a schema definition, perform following steps:

1. In the Schema Editor, click the **Import** button, or from the **Tools** menu, select **Import Schema**, to open the following dialog box:



2. Specify the **Library** name (default is `User`) and schema **Name**.
3. Select the **Source Type**, meaning the format of the imported schema.
4. Enter the path of the file containing the imported schema.

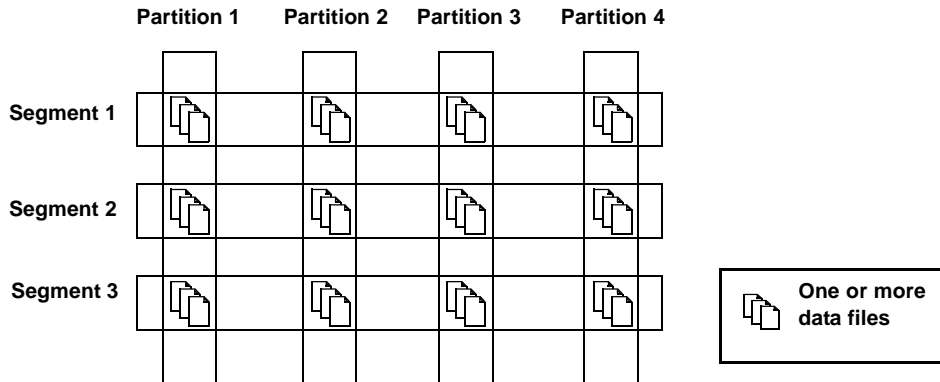
**For a Schema File**, the file may contain only a single schema definition.

**For a COBOL FD**, the file may contain only a single FD description. This import is equivalent to running the Orchestra `readcobol` utility with the `-f` option (specifying free format COBOL files). See the chapter on COBOL Schema Conversion in the *Orchestra User's Guide: Operators* for more information.

## Representation of Disk Data Sets

To use Orchestra well, you need to be familiar with Orchestra's representation of persistent data sets in a UNIX file system. Remember that virtual data sets are not stored to disk; only persistent data sets are saved.

Persistent data sets are stored in multiple data files on multiple disks in your system. The following figure shows the equivalent representation of an Orchestrate persistent data set represented as four partitions:



Each partition of a data set is stored on a single processing node. In this example, the data set has four partitions stored on four processing nodes.

A data segment contains all the records written to a data set by a single Orchestrate step. For example, if a step creates a data set and then writes its results to the data set, the data set will contain a single data segment.

You can select one of several write modes when your step writes its output to a data set. The default write mode is *create*, which means that Orchestrate creates the data set if it does not already exist. After the step writing to the data set completes, the data set will contain a single data segment. This mode causes an error if the data set already exists, in order to prevent you from accidentally overwriting your data.

*Replace* mode allows you to replace the contents of an existing data set. In this case, all data segments in the data set are deleted, and a single data segment is added to hold the new data records. In this case, the data set also contains a single segment after the write.

You use *append* mode to add records to a data set that already contains data. In this case, a new segment is added to the existing data set, and all records written to the data set are written to the new segment. Append mode does not modify the records in any other data segment.

## Setting the Data Set Version Format

By default, Orchestrate saves data sets in the Orchestrate Version 4.1 format. However, Orchestrate lets you save data sets in formats compatible with previous versions of Orchestrate. For example, to save data sets using the Version 3 format, set the `APT_WRITE_DS_VERSION` environment variable, as shown below:

```
export APT_WRITE_DS_VERSION=v3_0
```

After this statement takes effect, all data sets written by Orchestra are saved using the Version 3 format.

The *Orchestra Installation and Administration Manual* discusses in detail how to set and use environment variables.

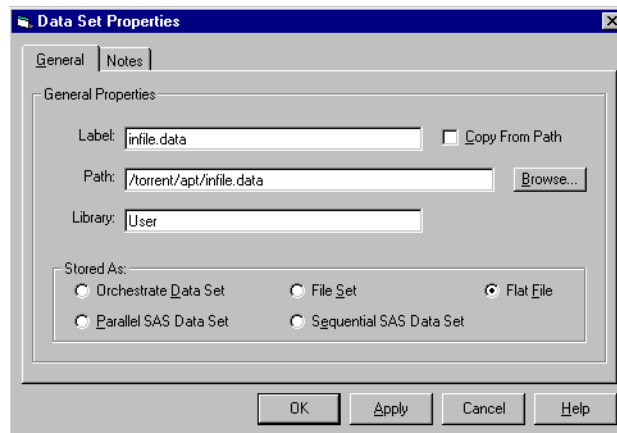
## Data Set Files

A persistent data set is physically represented on disk by:

- A single *descriptor* file
- One or more *data* files

The descriptor file contains the record schema of the data set, as well as the location of all the data files. The descriptor file does not contain any data. To access a persistent data set from your Orchestra application, you reference the descriptor file path name. Orchestra uses the information in this file to open and access the data set.

For example, the following **Data Set Properties** dialog box is for an input persistent data set whose descriptor file is named `/torrent/apt/infile.data`:



The data of a parallel data set is contained in one or more data files. The number of data files depends on the number of segments in the data set and the size of each partition. See the section “File Allocation for Persistent Data Sets” on page 4-35 for information on determining the number of data files for a data set.

## Descriptor File Contents

The data set descriptor file contains the following information about the data set:

- Data set header information identifying the file as a data set descriptor.
- Creation time and date of the data set.
- Data set record schema.



- A copy of the Orchestrate configuration file at the time the data set was created. By storing the configuration file within the data set, Orchestrate can access all data files of the data set even if you change the Orchestrate configuration file.

For each segment, the descriptor file contains:

- The time and date the segment was added to the data set.
- A flag marking the segment as valid or invalid.
- Statistical information such as number of records in the segment and the number of bytes. You can access this information using `orchadmin`.
- Path names of all data files, on all processing nodes, containing the records of the segment.

As stated above, the descriptor file contains a flag marking each segment as valid or invalid. When a new segment is added to a data set, the corresponding flag initially marks the segment as invalid. The flag is not set to valid until the step writing the data to the segment successfully completes. If the step fails for any reason, all information about the segment is deleted from the descriptor file, and all data in the segment is discarded.

In the case of a severe system failure, the data set may be stored with a flag marking the last segment in the data set as invalid. If you then read the data set as input, the invalid segment is ignored. Writing or appending data to the data set deletes the invalid segment.

You can also use the `cleanup` command with `orchadmin` to delete any invalid data segments within a data set, as shown below:

```
orchadmin cleanup myDataSet.ds
```

where `myDataSet.ds` is the name of the data set descriptor file.

See the *Orchestrate Installation and Administration Manual* for a detailed discussion of `orchadmin`.

## File Allocation for Persistent Data Sets

Persistent data sets are stored in multiple data files distributed throughout your system. This section describes how Orchestrate determines the location of these data files.

You can create persistent data sets in two ways:

- Use the `orchadmin` utility to create empty data sets. This data set contains no data segments or data files. See the *Orchestrate Installation and Administration Manual* for a detailed discussion of `orchadmin`.
- Use an Orchestrate operator to write to an output persistent data set. This data set will contain a single data segment and associated data files when it is first created. Each time you append data to the data set, a new segment is created to hold the new records. This data set can be read as input by another Orchestrate operator.

Orchestrate uses the configuration file to identify the processing nodes, and the disk drives connected to those nodes, available for use by Orchestrate applications. Additionally, you can define groups of nodes, called node pools, and groups of disk drives, called disk pools, to *constrain*

operations to those elements within the pool. See the section “Using Constraints with Operators and Steps” on page 10-5 for more information.

Several factors influence the number and location of the data files used to hold a persistent data set:

- The number of processing nodes in the default node pool
- Any node constraints applied to the operator writing to an output data set
- The number of disk drives connected to each processing node included within the default disk pool
- Any disk pool constraints applied to the output data set

Listed below are the rules Orchestrate uses to allocate data files for storing a persistent data set:

**Rule 1.** By default, Orchestrate executes an operator on all processing nodes defined in the default node pool. When an operator writes to an output data set, Orchestrate creates one partition of the data set on each processing node executing the operator.

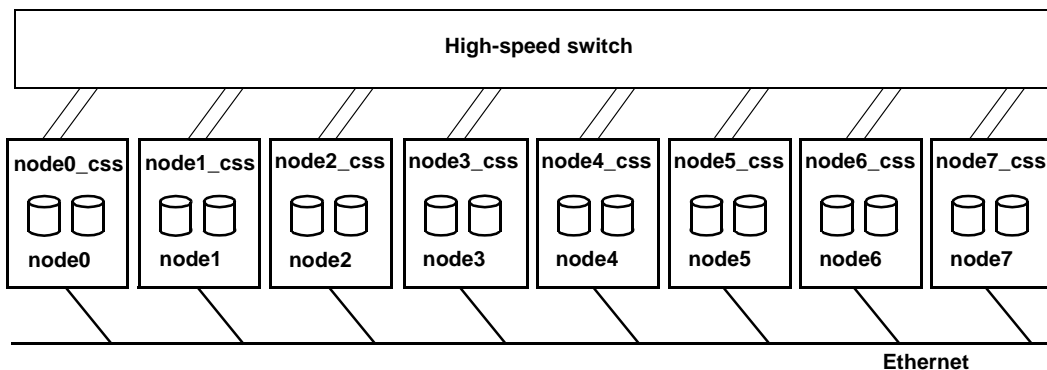
**Rule 2.** If you impose a node constraint on the operator, Orchestrate creates a partition of the data set only on those processing nodes executing the operator. See the section “Using Node Pool Constraints” on page 10-6 for information on specifying a node pool constraint.

**Rule 3.** For each processing node storing a partition of a data set, Orchestrate creates data files on all the disk drives included in the default disk pool connected to the processing node. See the section “Using Resource Constraints” on page 10-7 for information on specifying a disk pool constraint.

**Rule 4.** If you impose a disk pool constraint, Orchestrate creates data files on only those disk drives in the specified disk pool.

For example, suppose your application uses data set create mode to write 16 GB to a persistent data set. Since the data set did not already exist, it will be created with a single data segment. For this example, the data set's descriptor file is named `/home/user1/myDataSet.ds`.

This example executes on an eight-node system in which each processing node is connected to two disk drives. The following figure shows the system for this example:



In this example, all processing nodes are contained in the default node pool, and both disk drives are contained in the default disk pool. The operator writing to the output data set executes on all processing nodes.

Since all eight processing nodes execute the writing operator, Orchestrate creates eight partitions for the data set, one on each processing node. Orchestrate further divides each partition among the two disks in the default disk pool connected to each node to create 16 data files. Because each node receives approximately a 2-GB partition, the total amount of free space in all disks in the default pool on each processing node must be at least 2 GB.

If the data set held 64 GB, each of the 16 disks in the system would be required to hold 4 GB. Since many operating systems limit file size to 2 GB, each disk would hold two data files of 2 GB each.

Each data segment uses its own data files for storing the records of the segment. If you append data to an existing data set, a new segment descriptor is created, and new data files are created to hold the records in the new data segment.

In some circumstances, you may need your application to execute its operators on one set of processing nodes and store its data on another. The default operation of Orchestrate is to store a partition of a data set on each node executing the writing operator. When you want to store the data somewhere else, you insert a copy operator at the end of your step and use node constraints to execute the copy operator only on the processing nodes on which you want to store your data set. See the chapter “Constraints” for more information on node pools.

## File Naming for Persistent Data Sets

Orchestrate uses the following naming scheme for the data files that make up each partition of a parallel data set:

```
disk/descriptor.user.host.ssss.pppp.nnnn.pid.time.index.random
```

where:

- `disk`: Path for the disk resource storing the data file as defined in the Orchestrate configuration file.
- `descriptor`: Name of the data set descriptor file.
- `user`: Your user name.
- `host`: Hostname from which you invoked the Orchestrate application creating the data set.
- `ssss`: 4 digit segment identifier (0000-9999)
- `pppp`: 4 digit partition identifier (0000-9999)
- `nnnn`: 4 digit file identifier (0000-9999) within the partition
- `pid`: Process ID of the Orchestrate application on the host from which you invoked the Orchestrate application that creates the data set.
- `time`: 8-digit hexadecimal time stamp in seconds.
- `index`: 4-digit number incremented for each file.
- `random`: 8 hexadecimal digits containing a random number to insure unique file names.

For example, suppose that your configuration file contains the following node definitions:

```
{
  node node0 {
```

```
    fastname "node0_css"
    pool "" "node0" "node0_css"
    resource disk "/orch/s0" {}
    resource scratchdisk "/scratch" {}
  }
  node node1 {
    fastname "node1_css"
    pool "" "node1" "node1_css"
    resource disk "/orch/s0" {}
    resource scratchdisk "/scratch" {}
  }
}
```

For this example, your application creates a persistent data set with a descriptor file named `/data/mydata.ds`. In this case, Orchestrate creates two partitions for the data set: one on each processing node defined in the configuration file. Because each processing node contains only a single *disk* specification, each partition of `mydata.ds` would be stored in a single file on each processing node. The data file for partition 0 on the disk `/orch/s0` on `node0` is named:

```
/orch/s0/mydata.ds.user1.host1.0000.0000.0000.1fa98.b61345a4.0000.88dc5aef
```

and the data file for partition 1 on `node1` is named:

```
/orch/s0/mydata.ds.user1.host1.0000.0001.0000.1fa98.b61345a4.0001.8b3cb144
```

# 5: Orchestrate Operators

Orchestrate operators are the basic functional units of every Orchestrate application. An operator takes in data sets, RDBMS tables, or data files, and produces data sets, RDBMS tables, or data files. An Orchestrate step consists of one or more Orchestrate operators that process the data, according to the data-flow model for the step.

Orchestrate provides libraries of predefined operators for essential functions, such as import/export and sorting. For descriptions of operator interfaces and other details about individual operators in the Orchestrate libraries, see the *Orchestrate User's Guide: Operators*.

Orchestrate also lets you create your own operators, with either of the following methods:

- Creating an operator from a UNIX command, script, or program, as described in the chapter “Creating UNIX Operators”.
- Creating an operator from a few lines of your C or C++ code, as described in the chapter “Creating Custom Operators”.

In addition, you can use operators that you obtain from third-party developers in your Orchestrate application.

This chapter describes how to use predefined, user-defined, and third-party-developed operators in Orchestrate applications, through the following sections:

- “Operator Overview” on page 5-1
- “Using Visual Orchestrate with Operators” on page 5-3
- “Operator Interface Schemas” on page 5-6
- “Data Set and Operator Data Type Compatibility” on page 5-17

## Operator Overview

In general, an Orchestrate operator takes zero or more data sets as input, performs an operation on all records of the input data sets, and writes its results to zero or more output data sets. Data sets are described in detail in the chapter “Orchestrate Data Sets”.

Some operators limit the number of input or output data sets they handle, while others can handle any number of data sets (*n-input, m-output*).

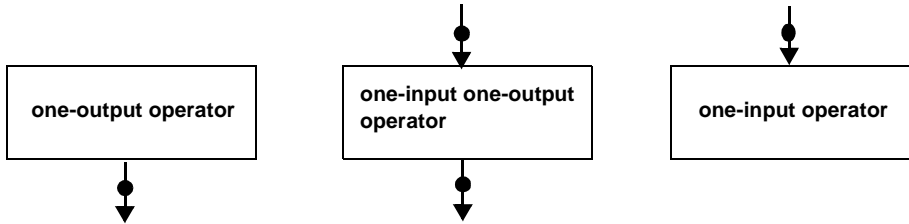
Most Orchestrate steps include one or more of the following kinds of operators:

- A *one-output* operator is usually the first operator in a step, to import data from a disk file or RDBMS and convert it into a data set.
- A *one-input, one-output* operator takes a single input data set, performs a processing operation

on it, and creates a single output data set.

- A *one-input* operator is usually used to export a data set to a disk file or an RDBMS. This type of operator is often the final operator in a step.

The following figure shows these three kinds of operators:



## Operator Execution Modes

Orchestrate operators execute in either parallel mode, on multiple processing nodes, or in sequential mode, on a single processing node.

Every Orchestrate operator has a default execution mode, either parallel or sequential. Many Orchestrate operators allow you to override the default execution mode. For example, the default execution mode of the `copy` operator is parallel, but you can configure it to run sequentially. Setting the operator execution mode is described in the section “Using Visual Orchestrate with Operators” on page 5-3.

In some circumstances, you may want to run an operator in parallel but limit the processing nodes that it uses. The reason might be that the operator requires system resources, such as a disk or a large amount of memory, that is not available to all nodes. Orchestrate allows you to limit, or *constrain*, an operator to a particular set of nodes to meet your application's requirements. See the section “Using Constraints with Operators and Steps” on page 10-5 for more information on controlling the processing nodes used by an operator.

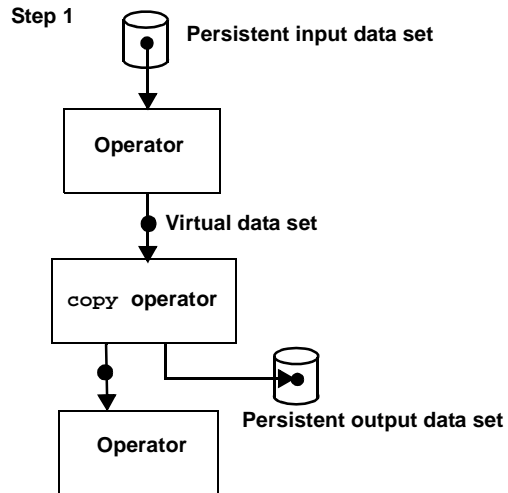
## Persistent Data Sets and Steps

Orchestrate steps consist of one or more operators, connected by data sets, as discussed throughout the preceding chapters. Creating and using steps to build Orchestrate applications is covered in detail in the chapter “Orchestrate Steps”.

As mentioned in the section “Using Data Sets with Operators” on page 4-3, a data set file cannot be both read from and written to in a single step.

If you want to write to a persistent data set and also use the data set as input in the same step, you need to use the Orchestrate `copy` operator to make a copy of the data set. For example, the following data-flow diagram shows the `copy` operator outputting both a persistent data set saved to

disk and a virtual data set that is input to the next operator in the step. See the *Orchestrate User's Guide: Operators* for more information on the `copy` operator.

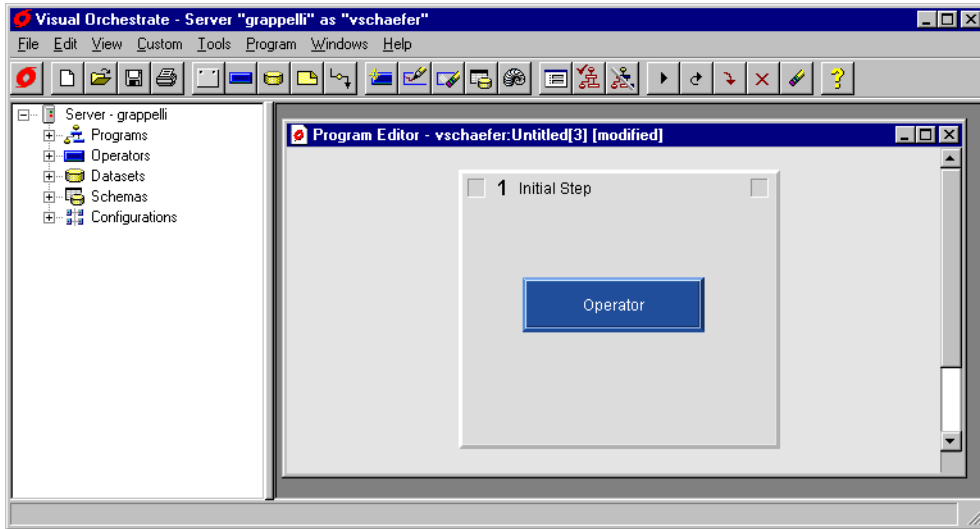


## Using Visual Orchestrate with Operators

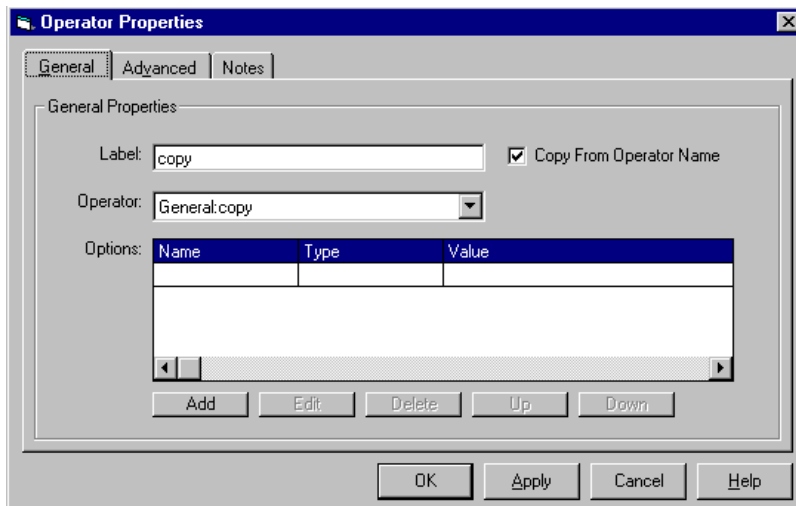
This section describes how to use Visual Orchestrate to configure operators.

1. Open an existing program, or create a new one.
2. To create an operator, do one of the following:
  - Select **Program -> Add Operator** from the menu.
  - Click the operator icon in the tool bar.
  - With the cursor in a **Program Editor** window, click the right mouse button and select **Add Operator** from the popup menu.
  - Click the operator name in the **Server View** and drag it into the **Program Editor**.

The operator icon appears in the **Program Editor** window, as shown below:



3. Double click the operator icon to open the **Operator Properties** dialog box. This dialog box is shown below:





The **General** tab contains the following:

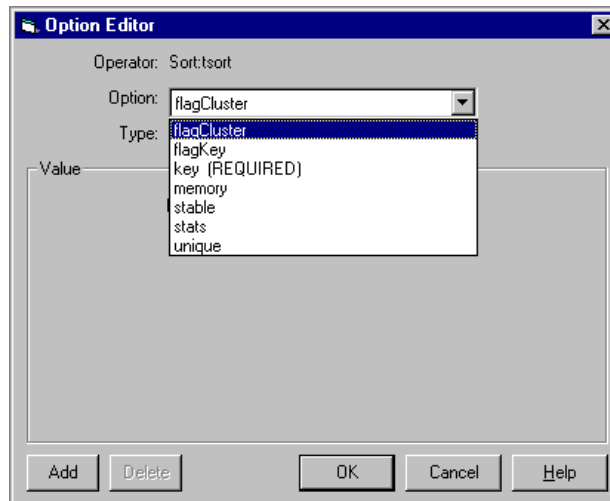
- **Label** is the operator label that appears in the **Program Editor** dialog box. It defaults to the operator name.
- **Operator** lets you select another operator. This pull-down list contains all Orchestrate built-in operators and all operators that you have created with the Orchestrate operator builders.
- **Options** lets you set the options that you can use to control and configure the currently selected operator. Use the buttons to work on options, as follows:

**Add** opens the **Option Editor** dialog box to set a new option for the operator.

**Edit** opens the **Option Editor** dialog box to edit the selected option. Use **Up** and **Down** to select the option to edit.

**Delete** removes an option from the list.

For example, if you select the `tsort` operator from the list of available operators, selecting **Add** opens the following **Option Editor** dialog box:



In the **Option Editor** dialog box, use the **Option** pull-down list to select the option you want to set for the operator. The remaining area of the dialog box lists the available values for the selected option.

Each Orchestrate operator has its own options accessible through the **Option Editor** dialog box. See the *Orchestrate User's Guide: Operators* for information on each operator.

4. Use the **Advanced** tab of the **Operator Properties** dialog box, as follows:
  - Set **Execute operator** options. This allows you to specify whether to execute an operator in parallel or sequentially. All operators have a default execution mode of parallel or sequential if you do not set this option.
  - Set **Constraints** on the operator. Constraints allow you to control the processing nodes used to execute the operator. See the chapter “Constraints” for more information.

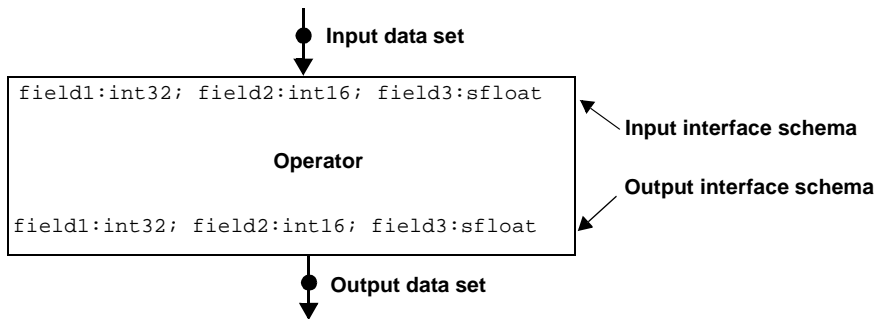
## Operator Interface Schemas

Orchestrate *schemas* are described in the section “Record Schemas” on page 4-2. To be used in an Orchestrate application, an operator has an *interface schema* for each of its input and output data sets. An output interface schema is propagated from an operator to the output data set, to the input interface of the next operator downstream, to that operator's output interface, and so forth through the operators in the step. Because an output data set schema would be overridden by the upstream operator's output interface schema, specifying an output data set schema is unnecessary.

The interface schemas of all the Orchestrate built-in operators are described in the *Orchestrate User's Guide: Operators*. Defining an interface schema is part of creating a user-defined operator; as described in the chapter “Creating Custom Operators”.

### Example of Input and Output Interface Schema

The following figure shows an operator that takes a single data set as input and writes its results to a single output data set.

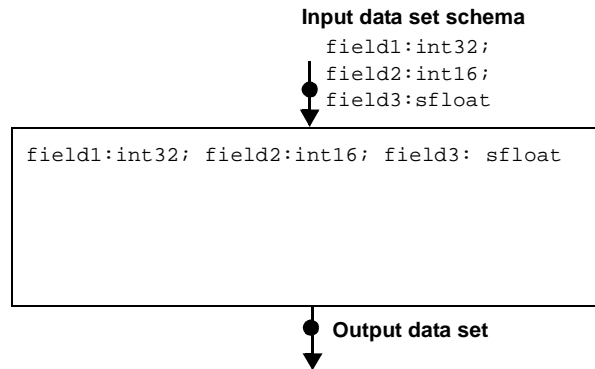


The figure shows the operator's input and output interface schemas, which in this case are the same. This interface schema specifies that the data set (both the input and the output) must have two integer fields named `field1` and `field2` and a floating-point field named `field3`.

The following sections describe using input and output data sets with operators. They also describe data type conversions between a data set's record schema and the interface schema of an operator.

## Input Data Sets and Operators

The following figure shows an input data set used with an operator:

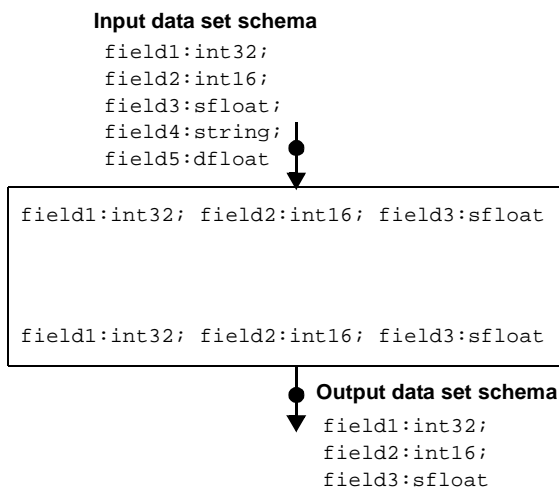


The operator's input interface schema requires the input data set to have three fields named `field1`, `field2`, and `field3`, with compatible data types. In this example, the input data set's record schema exactly matches the input interface schema and is therefore accepted by the operator.

Input data sets can contain aggregate fields (subrecords and tagged fields), as well as vector fields. To be compatible, the operator input interface must contain a corresponding vector or aggregate field in its input interface schema.

### Operators that Ignore Extra Input Fields

Some operators ignore any extra fields in an input data set, so that you can use the operator with any data set that has *at least* the fields that are compatible with those of the operator's input interface schema. The following example shows such an operator, with an input interface that defines three fields, taking as input a data set that defines five fields:



The first three fields of the data set are compatible with the three fields of the operator input interface, and the operator accepts the input. The operator ignores the input data set's `field4` and `field5` and does not propagate the extra fields to the output data set.

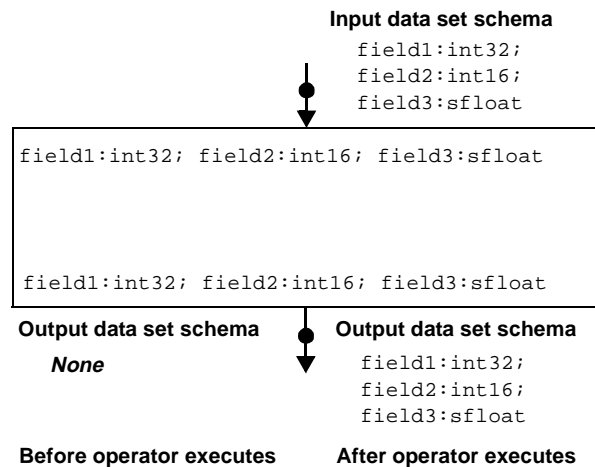
## Output Data Sets and Operators

This section describes how an operator writes to an output data set, as dictated by the relationship between the operator's output interface schema and, if present, the schema of the output data set. The following table gives a summary:

Output Data Set Schema In Relation to Operator Output Interface Schema	Operator Behavior
Output data set has no record schema (the usual case).	The output data set adopts the schema of the operator's output interface.
Output data set schema defines the same number of fields as the operator output interface.	The operator writes to all fields in the output data set.
Output data set schema defines more fields than the operator's output interface schema.	The operator sets extra fields to default value (according to nullability and type), and Orchestrate issues a warning.
Output data set schema defines fewer fields than the operator's output interface.	The operator drops the output fields that are not present in the output data set, and Orchestrate issues a warning.

In many cases, an output data set has no record schema. When written to by an operator, the output data set takes the schema of the operator's output interface. Therefore, an output data set with no record schema is compatible with all operators.

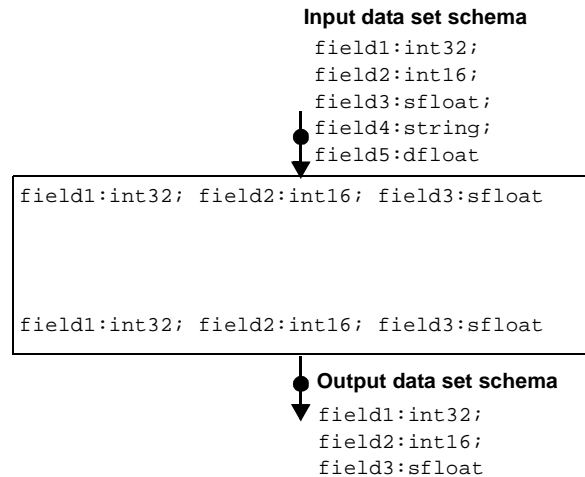
The following figure shows an output data set for which a record schema was not defined:



An output data set may optionally have a record schema. If an output data set has a record schema,

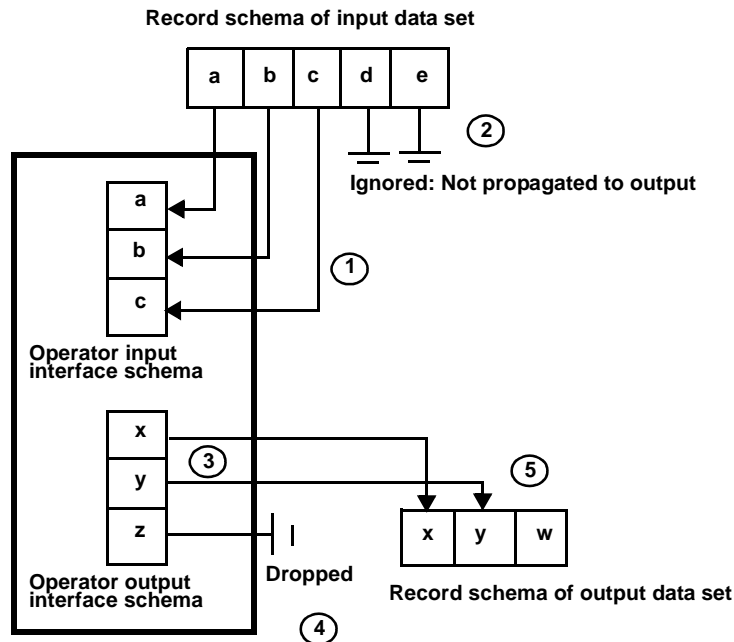
it must be compatible with the output interface schema of the operator. As shown in the second figure in the section “Input Data Sets and Operators” on page 5-7, if the output data set has the same number of fields and they are compatible with the output interface schema, the operator writes to all the fields.

An output data set can define fewer fields than the operator's output interface schema. In that case, the operator drops the fields not defined in the output data set, and it issues a warning. In the example shown in the figure below, the operator drops `field4` and `field5` from the output interface schema:



## Operator Interface Schema Summary

The following figure and keyed text summarize the relationship between the record schema of a data set and the interface schema of an operator:



1. Fields of an input data set are matched by name and compatible data type with fields of the input interface schema of an operator. The input data set must contain at least the number of fields defined by the input interface schema of the operator.

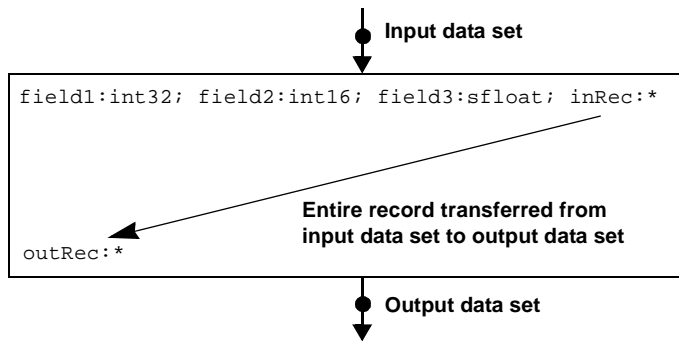
You can use the `modify` operator to perform certain changes, such as renaming, to enumerated fields in the input data set. See the chapter on the `modify` operator in the *Orchestrate User's Guide: Operators* for more information.

2. The operator ignores extra fields in the input data set.
3. If the output data set has no record schema (as recommended), the data set adopts the record schema of the operator's output interface schema, and the operator writes all fields to the data set.
4. If the output data set has a record schema, the operator writes to fields of the output data set that match fields in the operator's output interface schema. The operator drops any unmatched fields from the operator's output interface schema, and Orchestrate issues a warning.
5. If the output data set has a record schema, the operator sets to default values any fields in the data set that are not matched in the operator output interface schema, and Orchestrate issues a warning message.

## Record Transfers and Schema Variables

Some operators take or write an entire record, regardless of its size or the number and types of its fields. In an operator interface schema, an entire record is represented by a *schema variable*. A field that is a schema variable has an asterisk (\*) in place of a data type; for example, `inRec:*`. Schema variables give flexibility and efficiency to operators that input and/or output data on the record level without regard to the record schema.

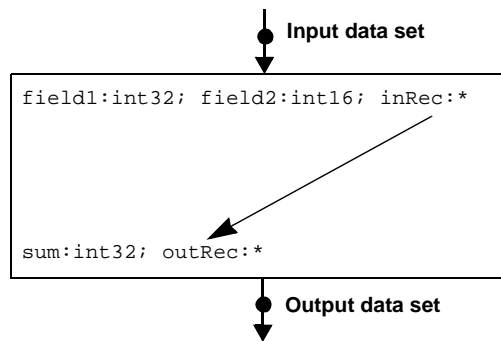
By default, an operator with a schema variable in its interface schema *transfers* (copies) an entire record from an input data set to an output data set, regardless of the other elements of the input and output interface schema. The following figure shows this default behavior for an operator that includes a schema variable in its input and output interface schema:



Transfers are used by some operators that modify the input data set and by others that do not. For example, the operator `lookup` modifies the input data set, and the operator `peek` performs a transfer without modifying the input data set.

Suppose that the operator in the figure above calculates the mean and standard deviation of the three fields identified in the input interface schema, across the entire input data set. In calculating these statistics, the operator does not have to modify the records of the input data set. This operator reads the records, makes calculations, and transfers each record to the output data set without change.

An operator can combine a schema variable in the output interface schema with additional, *enumerated* fields, as shown in the following figure:



In this example, the operator transfers the entire input record to the output data set and adds an additional field, which holds the sum of `field1` and `field2`.

### Determining the Record Schema of a Schema Variable

A schema variable refers to an entire input or output record, regardless of any other fields in the interface schema. This section describes how to determine the record schema associated with a schema variable.

The following figure shows an operator with schema variables in its interface. Below the figure are the record schemas represented by the input and output schema variables:

#### Input data set schema

```
field1: int32;
field2: int16;
field3: sfloat;
field4: int8
```



#### Output data set

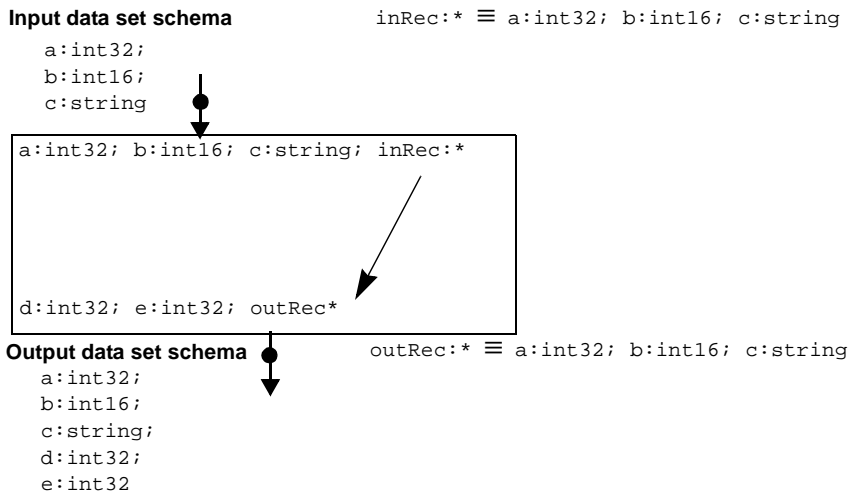
```
inRec:* ≡ field1:int32; field2:int16; field3:sfloat; field4:int8
```

```
outRec:* ≡ field1:int32; field2:int16; field3:sfloat; field4:int8
```



## Output Interface with Schema Variable and Enumerated Fields

In the following example, the output interface includes two enumerated fields, whose values are calculated by the operator, plus a schema variable:



The total output interface schema of the operator above comprises the schema variable `outRec` and the two new fields:

```

d:int32; e:int32; outRec:*
      /      \
d:int32; e:int32; a:int32; b:int16; c:string

```

The order of fields in the interface schema determines the order of fields in the records of the output data set. In the example, the two new fields were added at the *beginning* of the record, as listed in the output interface schema. The two new fields would be added to the *end* of the record if the output interface schema listed `outRec` first, as follows:

```

outRec:*; d:int32; e:int32
 /      \
a:int32; b:int16; c:string; d:int32; e:int32

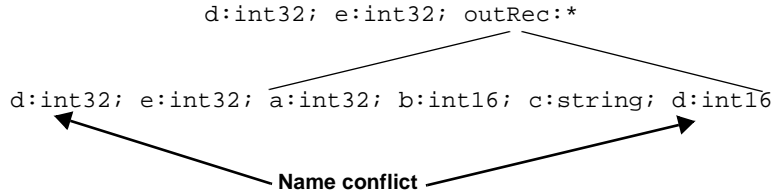
```

## Handling Duplicate Field Names in an Output Schema

In an operator output interface, a schema variable can have one or more fields with the same names as individually listed fields. This situation introduces a potential name conflict. For example, suppose in the example above, the record in the input data set that corresponds to `inRec` in the input interface schema, contained a field named `d`:

```
a: int32; b:int16; c:string; d:int16
```

If that record were transferred to `outRec` and both additional fields defined by the output interface schema, `d` and `e`, were added to the output data set schema, there would be a conflict between the `d` field in `outRec` and the extra `d` field, as shown below:



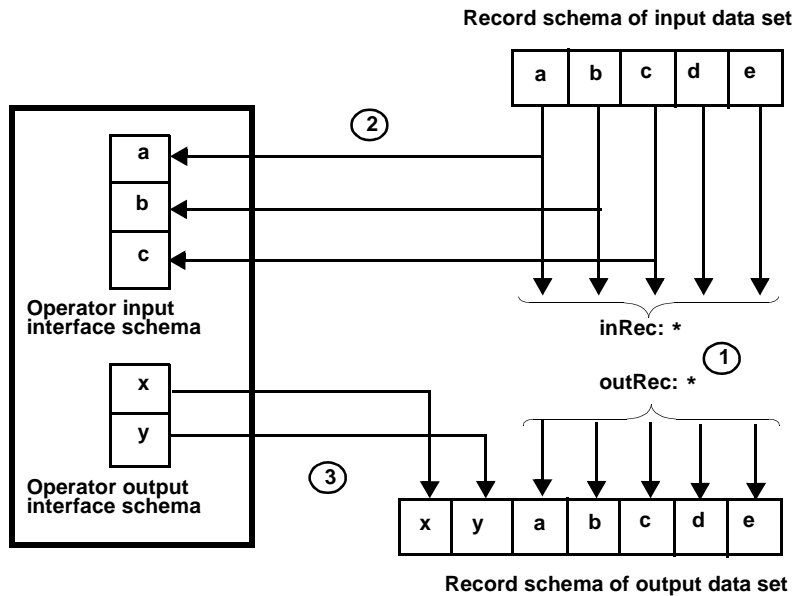
You can use the `modify` operator to explicitly drop or rename duplicate fields, as described in the `modify` chapter of the *Orchestrate User's Guide: Operators*.

**How Orchestrate Handles Duplicate Field Names**

If you do not use `modify` to handle duplicate field names, Orchestrate resolves name conflicts by dropping from the output data set schema any field with the same name as a preceding field (to its left) in the output interface schema, and Orchestrate also issues a warning message. In the example above, Orchestrate drops field `d` of the schema variable and issues a warning message.

**Summary of Schema Variable Usage**

This section summarize this section's discussion of schema variables, with the figure below, followed by a description keyed to the circled numbers:



1. The input interface schema can include a schema variable.
2. Operators use schema variables to transfer an entire record of an input data set to the output data set. By default, an operator with a schema variables in its interface schema transfers an

entire record from an input data set to an output data set, regardless of the other elements of input and output interface schemas of the operator.

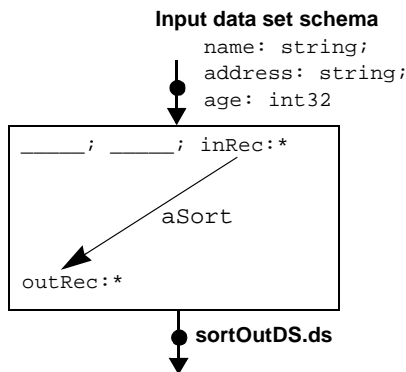
Operators that do not use schema variables drop unused fields from the records of an input data set. See the section “Input Data Sets and Operators” on page 5-7 for more information.

3. The output interface schema can include enumerated fields and/or a schema variable. The enumerated fields are added to the output data set. If an enumerated field has the same name as a field in the record assigned to the output schema variable, the duplicate field (reading left to right in the output interface schema) is dropped from the output data set schema.

## Flexibly Defined Interface Fields

Many Orchestrate operators (for example, `compare`) allow flexibility in the fields that they accept. For example, one operator's dynamic interface might take any number of double-precision floating point fields as input. Another's might accept either a single 8-bit integer or a string as input. A third operator's dynamic interface could take fields of any data type supported by Orchestrate.

For example, the following figure shows a sample operator, `aSort`, that has a dynamic interface schema:



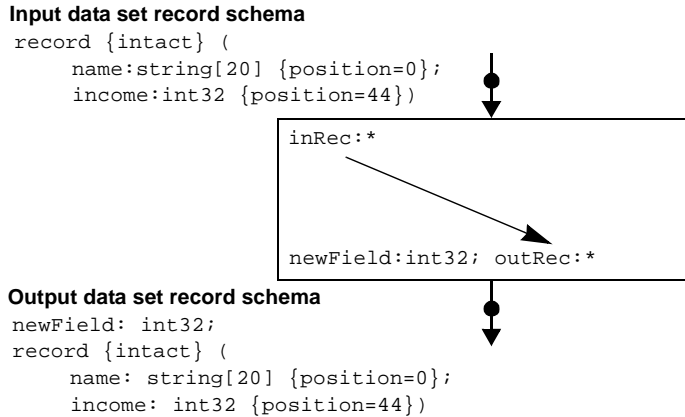
The input interface accepts two fields and a schema variable. When you invoke this operator, you pass the names of two fields in the input data set that you want `aSort` to use as sort keys.

The Orchestrate operator `tsort` has a more advanced input interface, which lets you specify any number of fields as sort keys. The `tsort` operator interface provides a `-key` control that takes one or more field names. With this interface you can, for example, specify `age` as the sort key. The `aSort` operator determines the data type of `age` from the input data set.

## Using Operators with Data Sets That Have Partial Schemas

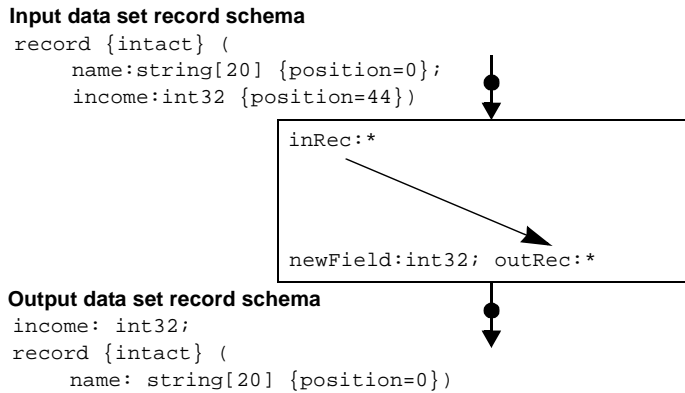
When an Orchestrate operator processes a data set with a partial schema (defined with the `intact` property), the operator cannot add, remove, or modify the data storage in the record. The operator can, however, add result fields to the beginning or end of the record.

For example, the following figure shows an operator adding a field to a data set that uses a partial schema definition:



In this example, the operator adds the extra field, `newField`, at the beginning of the record. If the operator output interface listed `newField` after `outRec`, the operator would add `newField` to the end of the output data set schema.

The name of an added field may be the same as a field name in the partial record schema. As with potential name conflicts involving complete record schemas, you can use the `modify` operator to handle the conflict (see the section “Handling Duplicate Field Names in an Output Schema” on page 5-13). If you do not use `modify`, Orchestrate drops the duplicate name (considering the output interface left to right) from the output data set schema. For example, the following figure shows an operator adding a field with the same name, `income`, as a field in the partial record schema:

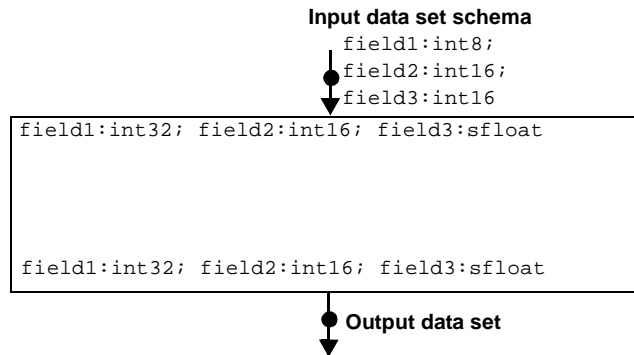


In this example, the new `income` field is added at the beginning of the data set schema, and the `income` field of the partial record is dropped. Note that dropping a field from a partial record schema eliminates only the field definition, and the contents of the record are not altered.

## Data Set and Operator Data Type Compatibility

For a data set to be used as input to or output from an operator, its record schema must be compatible with the operator's interface. For input data set record fields with types that differ from corresponding fields of the operator input interface, the types of those fields must be converted. The basics of Orchestrate data type conversion are introduced in the chapter "Orchestrate Data Types".

For example, the figure below shows an input data set with field data types that differ from those of the operator input interface schema:



The table in the section "Summary of Orchestrate Data Type Conversions" on page 3-9 shows the data type conversions that Orchestrate performs by default and those that you can perform with the modify operator.

The following sections contain additional information about default data compatibility and type conversions between source and destination data types:

- "Data Type Conversion Errors and Warnings" on page 5-17
- "String and Numeric Data Type Compatibility" on page 5-18
- "Decimal Compatibility" on page 5-19
- "Date, Time, and Timestamp Compatibility" on page 5-20
- "Vector Data Type Compatibility" on page 5-21
- "Aggregate Field Compatibility" on page 5-21
- "Null Compatibility" on page 5-21

## Data Type Conversion Errors and Warnings

During data type conversion, Orchestrate detects warning and error conditions. This section describes Orchestrate's actions in warning or error conditions arising from type conversions.

## Using `modify` to Prevent Errors and Warnings

To prevent many of the conditions that lead to warnings and errors caused by type incompatibility, use the `modify` operator. For example, you can use `modify` to configure a field to convert a null to a non-null value. Using the `modify` operator is described in the *Orchestrate User's Guide: Operators*.

You can also use `modify` to suppress warning messages.

## Orchestrate Handling of Warnings

A warning condition causes Orchestrate to write a message to the warning log. When a field causes a warning condition in consecutive records, the message appears for a maximum of five records. After the fifth warning message, Orchestrate suppresses the message. After a successful data conversion, Orchestrate's message counter resets to 0. If the same warning condition then recurs, Orchestrate begins issuing the message again, for up to five more records. However, Orchestrate will suppress a warning message after it is issued a total of 25 times during the execution of a step.

## Conversion Errors and Orchestrate Actions

An error occurs when Orchestrate cannot perform a data type conversion. For example, an error occurs if you attempt to convert a string field holding nonnumeric data, such as "April", to a numeric value.

When a data type conversion error occurs, Orchestrate's action depends on whether the destination field has been defined as nullable.

**Rule 1.** If the destination field has been defined as `nullable`, Orchestrate sets it to null.

**Rule 2.** If the destination field is not `nullable` but you have directed `modify` to convert a null to a value, Orchestrate sets the destination field to the value.

**Rule 3.** Otherwise, Orchestrate issues an error message and terminates the application. Unless you have disabled warnings, a warning is issued at step-check time.

## String and Numeric Data Type Compatibility

You can use the `modify` operator to perform conversions between type `string` and numeric types. For information, see the `modify` chapter in the *Orchestrate User's Guide: Operators*.

If you do not explicitly perform a conversion, Orchestrate automatically converts the integer or floating-point value to its string representation. For example, it converts the integer 34 to the character string "34", and the floating-point 1.23 to "1.23".

For converting string fields to numerics, Orchestrate attempts to interpret the string contents as a number that matches the data type of the target field. For example, Orchestrate converts the string "34" to the integer 34, and it converts the string "1.23" to the float 1.23. Orchestrate can convert a string-represented floating-point that includes an exponent, represented by "e" followed by the numbers comprising the exponent. For example, Orchestrate converts "1.23e4" to 12300.0.

Before first performing an automatic type conversion between a string and a numeric type, Orchestrate writes a warning to the error log. Orchestrate also issues a warning for any other type conversion that introduces:

- A possible loss in precision, such as a single-precision float converted to a 32-bit integer
- A possible loss in range, such as a 16-bit integer converted to an 8-bit integer

If the conversion between numeric and string data is not possible, a data type conversion error occurs.

For general information on Orchestrate warning and error handling, see the section “Data Type Conversion Errors and Warnings” on page 5-17.

## Decimal Compatibility

Orchestrate performs automatic conversions between `decimal` fields and integer, floating point, and string fields. As part of the conversion, Orchestrate checks for potential range violations and loss of precision.

Orchestrate checks for potential range violations before it runs the step that includes the conversion. If it detects a potential range violation, it issues a warning message and then proceeds to run the step. A range violation occurs when the magnitude of a source `decimal` field exceeds the capacity of the destination data type. The *magnitude* of a `decimal`, or number of digits to the left of the decimal point, is calculated by subtracting the scale of the `decimal` from the precision. For example, if the `decimal` has a precision of 15 and a scale of 5, there are ten digits to the left of the decimal point. Converting a decimal of this magnitude can result in a number that is too large for an `int16` field, for example.

Orchestrate checks for required rounding before it runs a step. If it detects a need for rounding, it issues a warning message and then proceeds to run the step.

Orchestrate performs the rounding necessary to convert the decimal. A decimal number with a scale greater than zero represents a real number with a fractional component. Rounding occurs when Orchestrate converts a source decimal with a scale greater than zero to an integer. Rounding also occurs when Orchestrate converts a source `decimal` to another `decimal` with a smaller scale. In both cases, Orchestrate must round the source decimal to fit the destination field.

The default rounding mode used by Orchestrate is called *truncate toward zero*. In converting a `decimal` to an integer field, Orchestrate truncates all fractional digits from the `decimal`. In converting a `decimal` to a `decimal`, Orchestrate truncates all fractional digits beyond the scale of the destination `decimal` field. For example, Orchestrate converts the `decimal` value -63.28 to the integer value -63.

If the source and destination decimals have the same precision and scale, Orchestrate does not perform a default conversion and thus does not check for a potential range violation or a need for rounding. You can, however, use the `modify` operator to perform a `decimal_from_decimal`

conversion, which may be helpful if the source field allows all zeros and the destination field does not (see the `modify` chapter in the *Orchestrate User's Guide: Operators*).

### String to Decimal Conversion

When converting a string to a decimal, Orchestrate interprets the string as a decimal value. To be converted to a decimal, the string must be in the form:

```
[+/-]ddd[.ddd]
```

Orchestrate ignores leading and trailing white space. Orchestrate performs range checking, and if the value does not fall into the destination decimal, a requirement failure occurs during setup check.

### Decimal to String Conversion

You can also convert a decimal to a string. Orchestrate represents the decimal in the destination string in the following format:

```
[+/-]ddd.[ddd]
```

Orchestrate does not suppress leading and trailing zeros. If the string is of fixed length, Orchestrate pads with spaces as needed.

A fixed-length string must be at least `precision+2` bytes long. If a fixed-length string field is not large enough to hold the decimal, a range failure occurs.

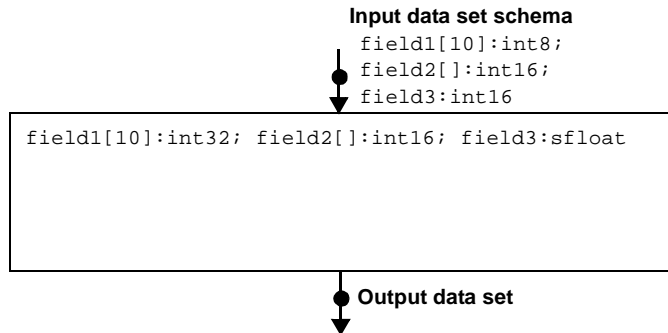
## Date, Time, and Timestamp Compatibility

Orchestrate does not perform any automatic conversions between `date`, `time`, or `timestamp` and other data types. However, you can use the `modify` operator to perform conversions between these three types and most other data types, as described in the `modify` chapter in the *Orchestrate User's Guide: Operators*.



## Vector Data Type Compatibility

Fixed-length vectors in the input data set must correspond to fixed-length vectors of the same length in the input interface. Likewise, input data set variable-length vectors must correspond to variable-length vectors in the input interface. The following figure illustrates these rules:



In this figure, `field1` of the input data set is a fixed-length vector of ten 8-bit integers; it is compatible with `field1`, a fixed-length vector of 32-bit integers, in the operator's input interface schema. Orchestrate automatically promotes the source field to a 32-bit integer representation.

The variable-length source field, `field2`, is compatible with the variable-length destination field, `field2`, in the operator input interface schema. As the data types are the same, no conversion is necessary.

## Aggregate Field Compatibility

For compatibility, subrecord fields in the input data set must correspond to subrecord fields in the operator input interface schema, and the same rule holds for tagged fields.

You can use the `modify` operator to match field names or perform data type conversions among the elements of an aggregate.

## Null Compatibility

If a record field's nullability attribute is set to `nullable`, the field can hold the null value. If a field's nullability attribute is set to `not_nullable` (default), it cannot hold a null. For more details on nullability of fields, see "Defining Field Nullability" on page 4-19.

The following table lists the rules followed for the nullability setting when an operator takes an input data set or writes to an output data set:

Source Field	Destination Field	Result
not_nullable	not_nullable	Source value propagates.
not_nullable	nullable	Source value propagates; destination value is never null.
nullable	not_nullable	If source value is not null, source value propagates. If source value is null, a fatal error occurs.
nullable	nullable	Source value (can be null) propagates.

An error occurs only if a source field holds a null value and the destination field is defined as `not_nullable`. In this case, Orchestrate issues a fatal error and terminates the application. You can use the `modify` operator to prevent the fatal error; see the chapter on the `modify` operator in *Orchestrate User's Guide: Operators*.

## 6: Orchestrate Steps

An Orchestrate application consists of at least one *step*, in which one or more Orchestrate operators process the application's data. A step is a data flow, with its input consisting of data files, RDBMS data, or persistent data sets. As output, a step produces data files, RDBMS data, or persistent data sets. Steps act as structural units for Orchestrate application development, because each step executes as a discrete unit.

As described in the chapter “Creating Applications with Visual Orchestrate”, you use Visual Orchestrate to create a step. The chapter describes in detail how to implement a multiple-step application and also provides information on debugging your application.

Because Orchestrate steps can perform complex processing tasks requiring considerable time to complete, you may want to check the step configuration before you run it. The last section of the chapter describes how to check for errors in a step before you run it.

This chapter contains the following sections:

- “Using Steps in Your Application” on page 6-1
- “Working with Steps in Visual Orchestrate” on page 6-4

### Using Steps in Your Application

In the data-flow model of your application, a step is bounded by the input and output of persistent data, which can be flat files or persistent Orchestrate data sets. Because disk I/O consumes resources and time, it is advantageous to use as few steps as possible to perform your application's processing.

Many applications, however, have conditions that require you to divide processing into two or more steps. The following two conditions require a multiple-step application design:

- The application's processing includes one operator that outputs a persistent data set and another operator that inputs that data set.

If two operators need to input from and output to the same persistent data set, they cannot run in the same step. For example, suppose in your application you need to use the transform operator, which always outputs a persistent data set, and then input that data to the statistics operator. The transform and statistics operators must run in separate steps.

- The application must reiterate an action, so that an operator processes the same data more than once.

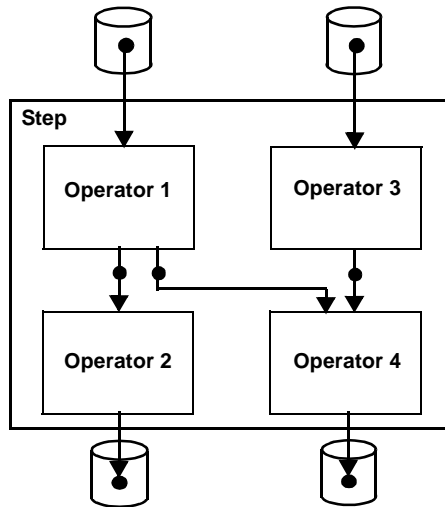
Looping (reversing the data flow) is not allowed in a step, as described on the section “Working with Steps in Visual Orchestrate” on page 6-4. Therefore, each iteration must be performed in a separate step.

The following section further describes the kinds of data flows that you can use in a step.

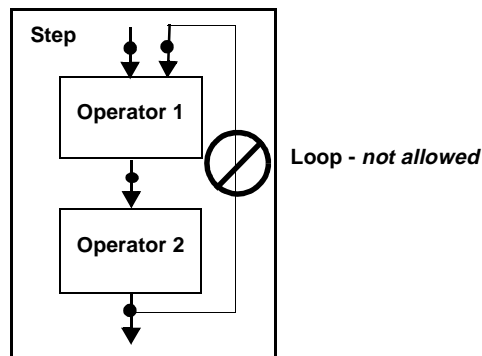
## The Flow of Data in a Step

To create a step, you connect multiple operators in a data-flow model. The data flow for a step must be a *directed acyclic graph*, which allows multiple inputs and outputs but in which data can flow in only one direction. However, you *cannot* use a loop (reverse-direction data flow) in a step.

The following example shows the data-flow model of a valid step that includes multiple outputs (from Operator 1) and multiple inputs (to Operator 4):

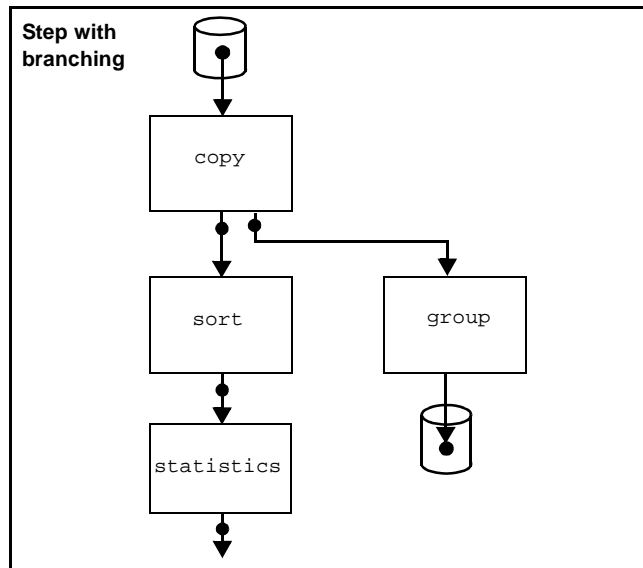


The following figure shows a data-flow model of an invalid step, containing a loop:



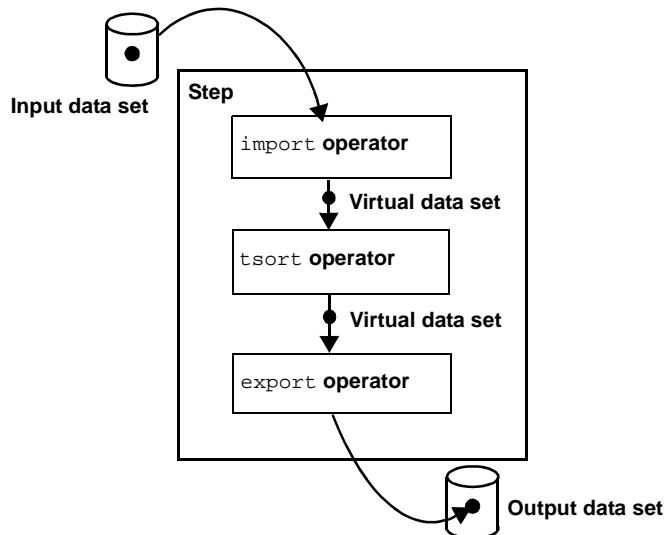
If you attempt to run a step that includes a loop, Orchestrate terminates execution and issues an error message. In order to reiterate an operation on a data set, you must create another step.

You can use branching operations within a step. The following figure shows an operator that produces two output data sets, each of which is used by a separate operator:



## Designing a Single-Step Application

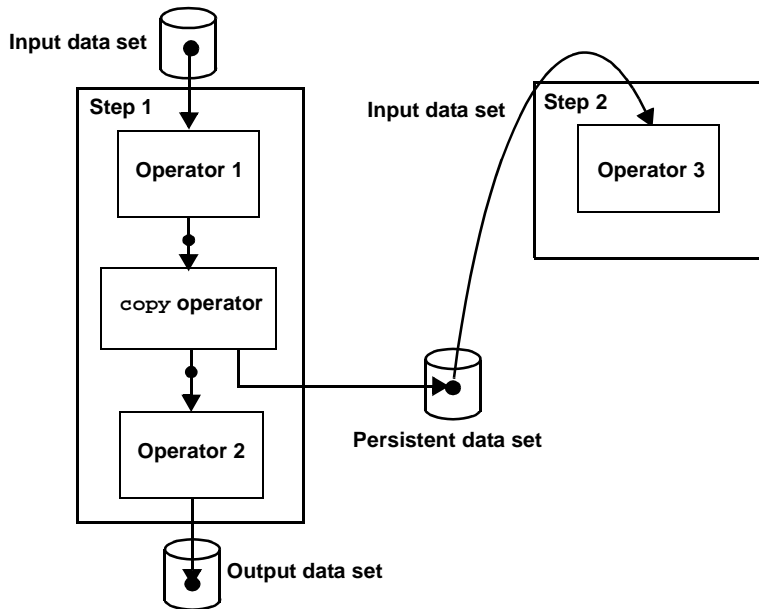
The following example shows the data-flow model of a single-step application, which sorts data. The step uses the predefined Orchestrate operators `import`, `tsort`, and `export`:



As described in the chapter "Orchestrate Operators", a virtual data set is a temporary buffer for data output from one operator and input to another operator in the step. The example figure shows a virtual data set connecting `import` to `tsort`, and another one connecting `tsort` to `export`.

## Designing a Multiple-Step Application

In the sample step above, the step takes an input file, performs operations on it, and stores the result in a file on disk, possibly to be used as input to another step. Storing the output of a step in a persistent data set allows you to create a data set in one step and process the data set in a second step. In the following figure, Step 1 performs a preprocessing action, such as sort, on a data set that Step 2 also requires. By performing the preprocessing in Step 1 and saving the result to a persistent data set, Step 2 can input the preprocessed data set.



You cannot use a persistent data set both for input and for output within one step. However, as described in the chapter “Orchestrate Data Sets”, you can use the Orchestrate `copy` operator to save a data set to disk and to output the same data as a virtual data set that can be input to the next operator in the step.

## Working with Steps in Visual Orchestrate

The following sections describe how to use Visual Orchestrate to manipulate steps:

- “Creating Steps” on page 6-5
- “Executing a Step” on page 6-7
- “Setting Server Properties for a Step” on page 6-8
- “Setting Environment Variables” on page 6-10
- “Using Pre and Post Scripts” on page 6-12

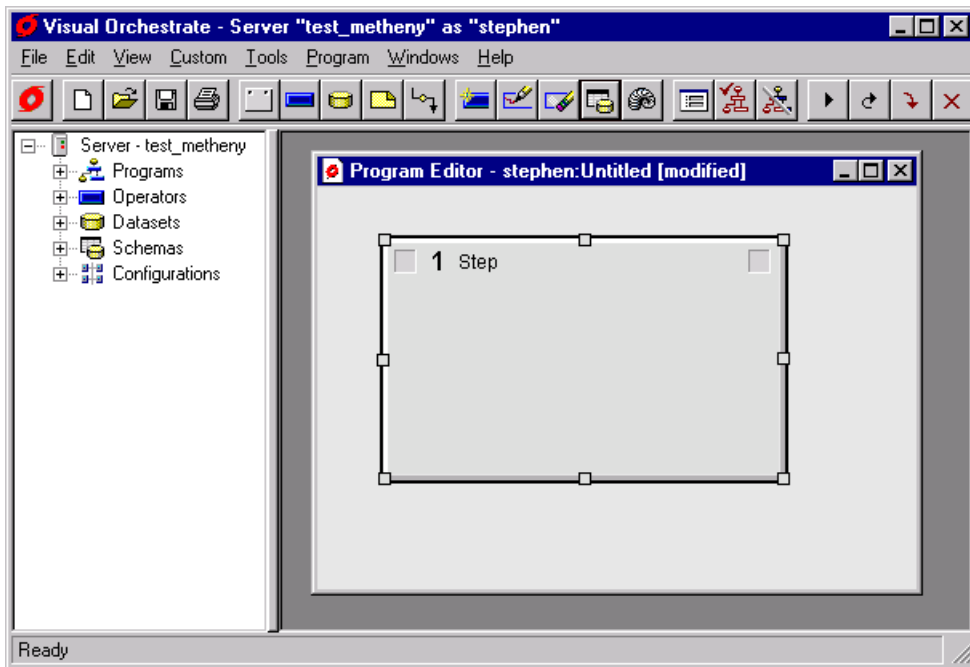
## Creating Steps

When you create a new program, Visual Orchestrate automatically creates an initial step for the program.

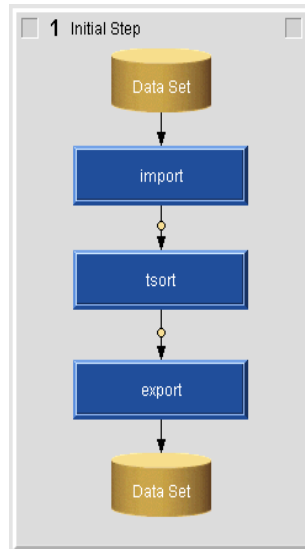
To add a step to a program, perform any one of the following actions:

- Select **Program -> Add Step** from the menu
- Click the step icon in the tool bar
- Right-click in the **Program Editor** window background area (outside any step). From the popup menu, select **Add Step**.

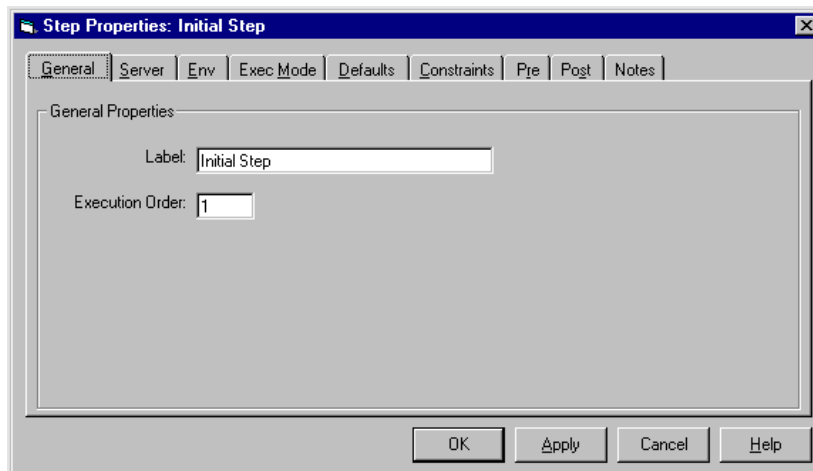
The step displays as a box in the **Program Editor** window, as shown below



You can add operators, data sets, and data-flow links to the step. For example, the figure below shows a step that implements the example in the section “Designing a Single-Step Application” on page 6-3:



To view and modify the properties of a step, use the **Step Properties** dialog box, shown below. To open the **Step Properties** dialog box, double-click in the step box (not on an operator, data set, or link).



This dialog box contains the following tabs:

- Use the **General** tab to set the step **Label** and **Execution Order** within your program.
- Use the **Server** tab to configure the execution environment of the step. See the section “Setting Server Properties for a Step” on page 6-8 for more information
- Use the **Env** tab to set the environment variables used by the step. See the section “Setting Environment Variables” on page 6-10 for more information.



- Use the **Execution Mode** to configure how your step uses processors when it executes. See the section “Setting Step Execution Modes” on page 6-10 for more information.
- Use the **Defaults** tab to set buffering parameters and default import schemas.  
See the chapter on the import/export utility in the *Orchestrate User's Guide: Operators* for more information on default schemas.
- Use the **Constraints** tab to set constraints on the step.  
See the chapter “Constraints” for more information.
- Use the **Pre** and **Post** tabs to add a UNIX shell script that executes before (**Pre**) or after (**Post**) the step. These scripts are executed using the UNIX Bourne shell (`/bin/sh`). See the section “Using Pre and Post Scripts” on page 6-12 for more information.
- Use the **Notes** tab to enter text that describes the step. The text is saved with the step. This tab provides standard text editing functions, such as selection, cutting, and pasting; right-click to display a context menu of these functions.

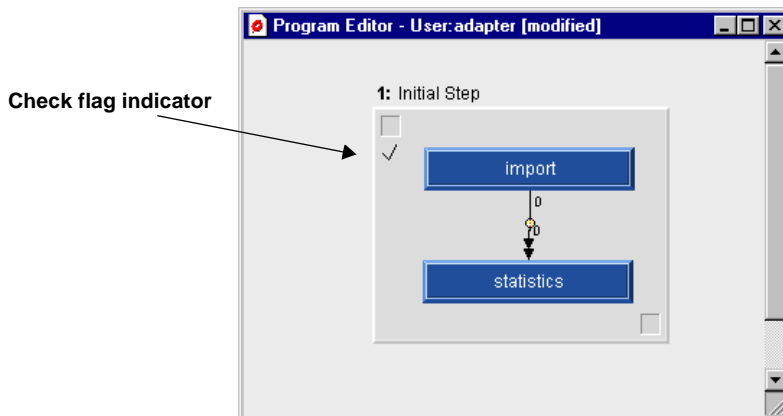
## Executing a Step

To execute a step, press the **Run** button on the Visual Orchestrate tool bar. Orchestrate first checks the step for errors (such as incorrectly connected links), and then runs the step.

To configure Visual Orchestrate to check a step but not to run it, use the **Program** menu. This menu entry has the following options:

- **Step Check All:** If you select this option, pressing the Run button will check all steps for errors, but will not execute any step. This option sets the check-only flag for all steps. You must explicitly clear the check-only flag using **Step Check None**, or by clicking **Enable Step** in the popup menu for the step.

Shown below is a step with the check flag set:



- **Step Check Selected:** Sets the check flag for the currently selected step. Pressing the Run button will check the step for errors, but will not execute it. You must explicitly clear the check-only flag using **Step Check None**, or by clicking **Enable Step** in the popup menu for the step.
- **Step Check None:** Clears the check-only flag for all steps in an application. After choosing

this menu command, pressing the Run button will both check and execute all steps in the application.

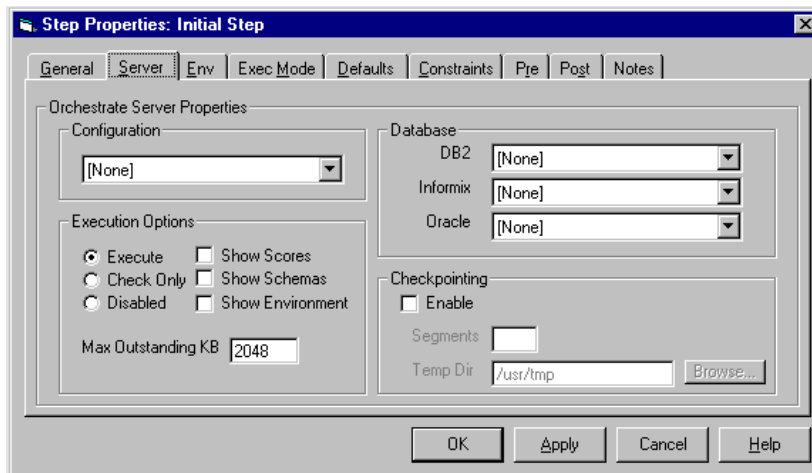
---

**Note:** Running an application that displays a great deal of text (more than a few thousand lines) in Visual Orchestrator can negatively affect its performance. When you run a display-intensive application in Visual Orchestrator, redirect its output from the screen to a file, which you can examine outside Orchestrator (in a text editor, for example) after the run.

---

## Setting Server Properties for a Step

An Orchestrator program consists of one or more Orchestrator steps. When you create an Orchestrator program, you use the **Program Properties** dialog box to set the global properties on the program. However, you can override many of these properties on individual steps using the **Server** tab of the **Step Properties** dialog box, shown below:




---

**Note:** Any properties left unchanged in the **Step Properties** dialog box default to the settings in the **Program Properties** dialog box.

---

You can optionally set any one of the following server properties:

**Configuration:** Select the Orchestrator configuration used to execute the step. The configuration defines the processing nodes and disk drives available for use by your program.

The Orchestrator server administrator is required to set up at least one default configuration before you can create an Orchestrator step or program. If no configuration is available, see the Orchestrator server administrator

You may have several different configurations available. For example, one configuration may be for testing and another for deployment.

**Execution Options:** Click **Check Only** to validate the step but not run it, and **Execute** to execute the step. Click **Disabled** to cause Visual Orchestrate to skip this step (that is, not to execute the step as part of the application).

If you change execution options with the menu (such as with **Program->Step Check All**) or with the step popup menu (such as with **Disable Step**), the **Execution Options** setting on the Server Properties tab will reflect the setting the next time you open the Step Properties dialog box.

**Show Scores** causes Visual Orchestrate to display extensive diagnostic information about the step as it executes.

**Shows Schemas** causes Visual Orchestrate to print the record schema of all data sets and the interface schema of all operators.

**Shows Environment** causes Visual Orchestrate to display the applicable environment variables during the program run.

**Max Outstanding KB:** Sets the amount of memory, in bytes, reserved for Orchestrate on every node to communicate over the network. The default value is 2 MB (2048 bytes).

---

**Note:** If you are working on a stand-alone workstation or stand-alone SMP, leave **Max Outstanding KB** at its default value.

---

If your system uses a network to connect multiple processing nodes, such as an IBM switch in an MPP or an Ethernet connection in a network cluster, set **Max Outstanding KB** as follows:

- If you are using the IBM HPS switch (also referred to as TB2), set a value less than or equal to  $(\text{thewall} * 1024)/2$ .
- If you are using the IBM SP switch (also referred to as TB3), set a value less than or equal to  $\text{MIN}(\text{spoolsize}, \text{rpoolsize})$ .
- For workstations connected by a network, set a value less than or equal to  $(\text{thewall} * 1024)/2$ .

Contact your system administrator for the correct setting for **Max Outstanding KB**. The following are some guidelines for determining the setting for your system.

Setting **Max Outstanding KB** to its maximum value means that Orchestrate reserves all available memory for communicating over the network, and that no other application will be able to communicate. Set this environment variable based on your understanding of the system load required by Orchestrate and by all other applications running on the system by all other users.

For example, if you are using DB2 and Orchestrate together, you could set **Max Outstanding KB** to reserve 8 MB for Orchestrate. If you are testing Orchestrate by running small applications, you could set **Max Outstanding KB** to reserve only 4 MB.

If you run multiple Orchestrate applications concurrently, set **Max Outstanding KB** to a fraction of the value that you would use for a single application. For example, if you normally set **Max Outstanding KB** to 8 MB for a single Orchestrate application, set it to 4 if you are going to run two concurrent Orchestrate applications.

If you use SMP nodes in your system, you must set **Max Outstanding KB** to the value determined above, divided by the number of Orchestrate nodes defined for the SMP node. For example, if you configure Orchestrate to recognize a single SMP node as two separate process-

ing nodes, set **Max Outstanding KB** to its normal value divided by two. If you have multiple SMP nodes, divide the value of **Max Outstanding KB** by the largest number of Orchestrate nodes defined for any single SMP node.

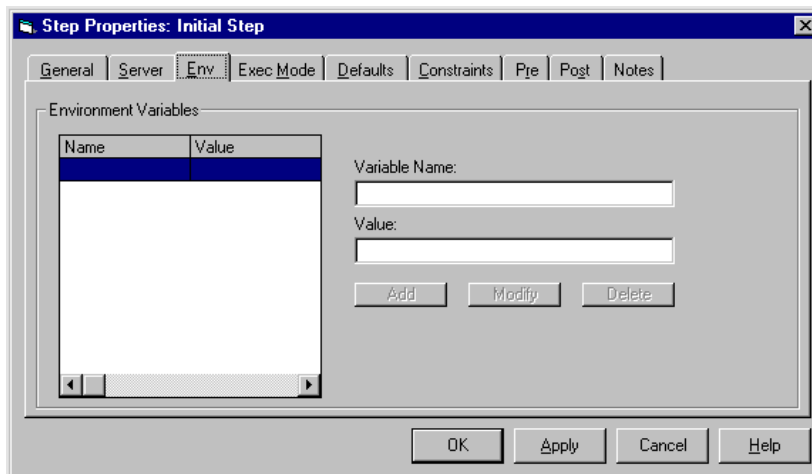
**Database:** Specifies the database configuration used by the step. The database configuration defines the database type (DB2, INFORMIX, Oracle) as well as the specific database configuration to use.

If you are accessing a database, the Orchestrate server administrator must set up at least a default database configuration before you can create an Orchestrate step or program. If no configuration is available, see the Orchestrate server administrator.

You may have several different configurations available depending on the database and database data that you want to access.

## Setting Environment Variables

You can use the **Env** tab in the **Step Properties** dialog box to set environment variables required by the step, or to override global environment variables set in the **Program Properties** dialog box. The **Env** tab is shown below:



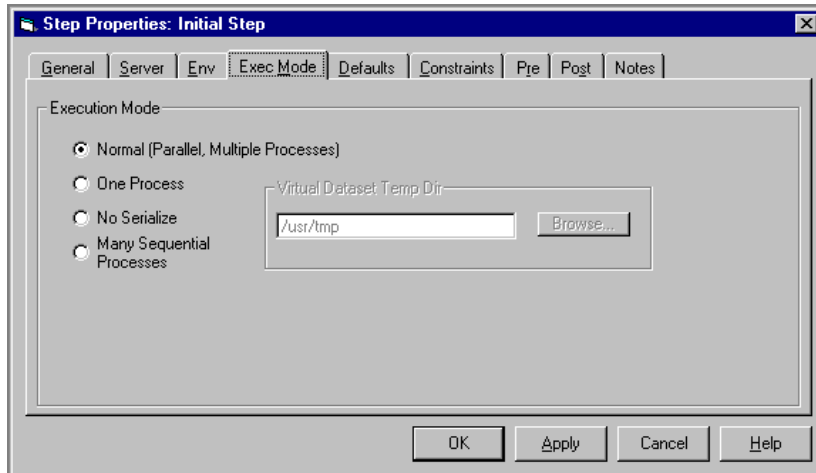
## Setting Step Execution Modes

Debugging a parallel Orchestrate application can be difficult when the application executes in multiple UNIX processes on multiple processing nodes. To simplify application development and debugging, you can execute your Orchestrate application in the sequential execution mode.

In sequential execution mode, a step executes on a single processing node and has access only to the resources (including disk storage) of that node. To use sequential execution mode, you need to construct a testing data set as a subset of your complete data set. The testing data set must be small enough to be handled easily on a single processing node with the available system resources.

When your application works properly in sequential execution mode, it is time to test the program in parallel execution mode. In parallel mode, your application executes in multiple UNIX processes on multiple processing nodes using the full capabilities of the Orchestrate framework.

The **Execution Mode** tab of the **Step Properties** dialog box, shown below, lets you override global execution mode properties set in the **Program Properties** dialog box.



You can optionally set the following properties:

**Normal:** Specifies standard execution mode. This means each operator executing on each processing node creates a separate UNIX process. The operators execute in parallel or sequentially based on how you configured the operator.

**One Process:** Sets the program to execute sequentially on a single processing mode. In addition, the application executes in a single UNIX process. You need to run only a single debugger session, and you can set breakpoints anywhere in your code. In addition, data is partitioned according to the number of nodes defined in the configuration file.

Orchestrate executes each operator as a subroutine. Each operator is called the number of times appropriate for the number of partitions on which it must operate.

**No Serialize:** The Orchestrate persistence mechanism is not used to load and save objects. Turning off persistence may be useful for tracking errors in derived C++ classes. That is, if turning off serialization in a program that previously crashed results in the program executing correctly, the problem may be located in your serialization code.

**Many Sequential Processes:** Sets the program to execute sequentially on a single processing node. Orchestrate forks a new UNIX process for each instance of each operator and waits for it to complete.

**Virtual Dataset Temp Dir:** During sequential program execution, virtual data sets are written to files in the specified directory which defaults to the current working directory. These files are named using the prefix `aptvds`. This allows you to examine a virtual data set as part of debugging your application. Virtual data sets are deleted by the framework after they are used.

## Using Pre and Post Scripts

This section describes how to create and use pre and post scripts and server variables, to modify the execution of your steps. This section includes a detailed example, in the section “Example: Passing a Value to an Operator from the Pre Shell Script” on page 6-13.

### Creating Pre and Post Scripts

Each Orchestrate step allows you to create and associate two scripts with the step: a **Pre** and a **Post** shell script. You can use these scripts to perform such operations as:

- Opening and closing UNIX pipes for use by the import/export operators
- Calculating run-time values passed to Orchestrate operators
- Performing any other pre or post processing actions required by the step

The order of execution of a step and its associated shell scripts:

1. Execute the **Pre** shell script, if any.
2. Execute the step.
3. Execute the **Post** shell script, if any.

Note that the **Pre** and **Post** shell scripts are independent of each other; that is, you can create either a **Pre** or a **Post** shell script, or both.

By default, Orchestrate uses the Bourne shell to execute the shell script. However, you can use any shell to execute the script. For example, the first line of the following script specifies the Korn shell:

```
#!/bin/ksh
# script goes here
```

Each step in an application, as well as each **Pre** and **Post** shell script, executes in its own shell environment. Any modifications to the UNIX shell environment made by the **Pre** shell script are not passed to the shell environment of the step or to that of the **Post** shell script. Therefore, you cannot use UNIX environment variables to pass information. Instead, you can use Orchestrate server variables, which exist only during execution of the application in which you create them.

### Using Orchestrate Server Variables

Orchestrate provides a function that you can use to store on the server the result of calculations in your **Pre** shell script. In your step, operators can reference that value. Orchestrate also provides a function that you can use to access the variable in a **Post** shell script.

To write a value to the server, use the function:

```
set_orchserver_variable var_name var_value
```

where:

- *var\_name* is a variable name. The server stores the variable and its value for the duration of your Orchestrate program, so that other steps in the program can access the variable. At the end of the program run, Orchestrate deletes the variable.
- *var\_value* is the variable's value and can be an integer (signed or unsigned), float, or string. All values are stored on the server as strings.

To get a value from the server, use the function:

```
value = get_orchserver_variable var_name
```

where:

- *value* is the variable value represented as a string
- *var\_name* is the name of the server variable

`get_orchserver_variable` returns an exit code of 0 if it succeeds and 1 if it fails. If *var\_name* is not stored in the server, the call succeeds but returns the empty string.

The following two sections describe examples of using pre scripts and server variables.

### Example: Passing a Value to an Operator from the Pre Shell Script

A common use of the **Pre** shell script is to calculate the value of an argument for an operator in the step. For example, the Orchestrate `sample` operator takes an argument specifying the percentage of an input data set written to each output data set. The percentage is a floating-point value in the range of 0.0, corresponding to 0.0%, to 100.0, corresponding to 100.0%.

Shown below is a shell script that you can use to calculate the percent sample size required by the `sample` operator to copy 1600 records from the input data 'set to the output data set. You enter this script in the **Pre** tab area of the **Step Properties** dialog box:

```
#!/bin/ksh
numSampled=1600 # Line 1
numRecs=`dsrecords inDS.ds | cut -f1 -d' '` # Line 2
percent=`echo "10 k $numSampled $numRecs / 100 * p q" | dc` # Line 3
set_orchserver_variable sample_percent `echo $percent` # Line 4
```

**Line 1.** In this example, you want the output data set to contain 1600 records.

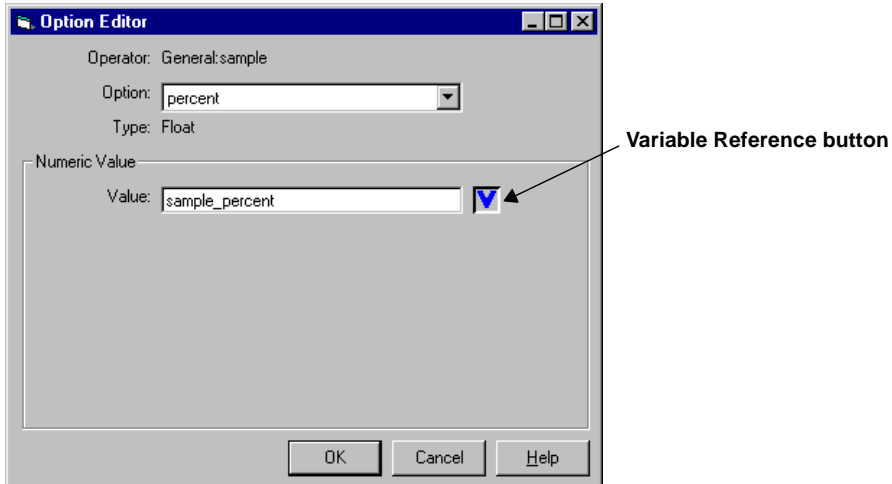
**Line 2.** Use the `dsrecords` utility to obtain the number of records in the input data set. The return value of `dsrecords` has the form `# records` where `#` is the number of records. This statement returns the record count and strips off the word `record` from the value.

**Line 3.** Calculate a floating point value for the sample percentage from the 1600 records required by the `sample` and the number of records in the data set. This example uses the UNIX `dc` command to calculate the percentage. In this command, the term `10 k` specifies that the result has 10 digits to the left of the decimal point. See the `man` page on `dc` for more information.

**Line 4.** Use `set_orchserver_variable` to write the variable `sample_percent` to the server.

You can now reference `sample_percent` in your step, to set the sampling percentage for the `sample` operator. To reference the `sample_percent`, double-click the operator to display its

**Operator Properties** dialog box. On the **General** tab, select **Add** to add a option or **Edit** to edit an existing one. The **Options Editor** dialog box for the `sample` operator appears. Click the Variable Reference button, marked with a blue **V**, to select it. In the text field to the left of the Variable Reference button, type the variable name, as shown in the following figure:



Each time you run the step, the **Pre** shell script calculates the required percentage to generate 1600 records in the output data set and passes this sampling percentage to the operator.



# 7: The Performance Monitor

The Orchestrate performance monitor (also called `orchview`) produces a graphical, 3-D representation of an Orchestrate step as it executes. The monitor allows you to track the execution of an Orchestrate step and display statistical information about Orchestrate operators during and after step execution.

This chapter describes the performance monitor, including the monitor's graphical user interface. This chapter also describes how to configure your system to use the performance monitor and how to run the performance monitor.

This chapter contains the following sections:

- “The Performance Monitor Window” on page 7-1
- “Controlling the Performance Monitor Display” on page 7-5

---

**Note:** Your system must have X Windows support in order to run the Orchestrate performance monitor.

---

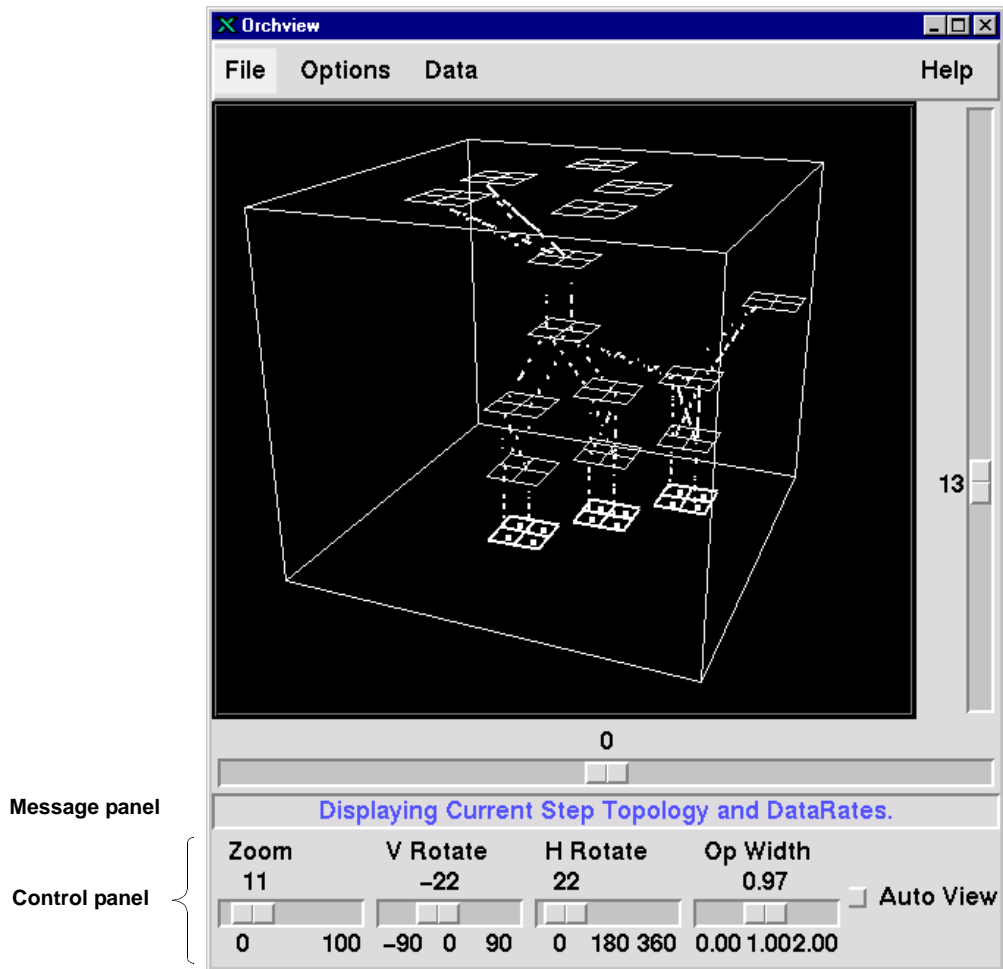
## The Performance Monitor Window

The performance monitor shows the execution of a step in a 3-D format that allows you to view the operators and the data flow among the operators that make up the step. You can use the monitor to display a step as it executes, or you can record the step and play it back later.

The performance monitor allows you to zoom the display, and rotate the display horizontally and vertically, to examine individual operators or operator connections. You can set a number of options that control how the performance monitor displays your program execution. For example, you can set the frequency at which the entire display is updated. You can also customize the display style for data flows, and the volume or rate cutoff at which the performance monitor shows a change in the data flow.

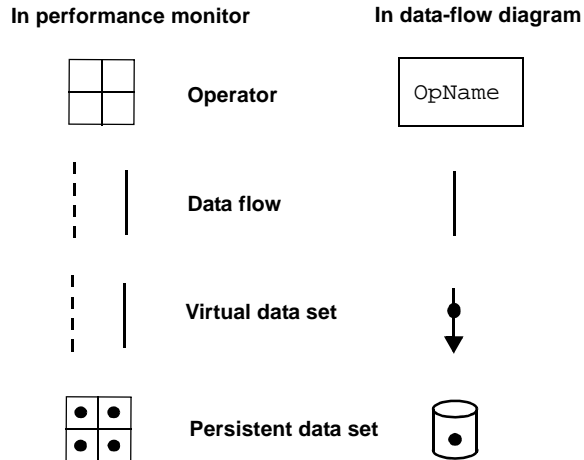
The performance monitor lets you display statistical information about the operators and the data sets in your program. For example, you can display a snapshot of the current record read and write rates. The performance monitor also has a feature that lets you create a spreadsheet from your program statistics.

Shown below is the performance monitor window with a run-time display of a sample application:



## How the Performance Monitor Represents Your Program Steps

The symbols the performance monitor uses to represent operators and data sets are similar to those used in data-flow diagrams, as shown below:



As shown in the figure above, the performance monitor displays *operators* as square grids. Each used cell in an operator grid represents an instance of the operator on an Orchestrate processing node. The performance monitor also displays *persistent data sets* as square grids. Each used cell in the grid represents an Orchestrate processing node.

An operator grid may also contain cells that are never used, as explained below.

### How the Performance Monitor Determines Grid Dimensions

All operator and persistent data contain a square number of cells (4, 9, 16, 25, and so forth), arranged as a square. All the operator and persistent data set grids in a step have the same number of cells: the largest number of Orchestrate processing nodes used by the step, rounded up to the nearest square number.

For example, if the largest number of processing nodes used by the step is 16, then all operator and data-set grids have 16 cells (4 rows by 4 columns). In another example, if the largest number of processing nodes used is 22, the number of cells is rounded up to the nearest square number, 25, and every grid in the step has 5x5 cells.

---

**Note:** Any cells in a grid that do not represent processing nodes, which the performance monitor has added to make the grid a square, are *never used* and therefore never show activity. Do not confuse these never-used cells with temporarily *inactive* cells, which represent Orchestrate nodes but show no activity in the current display.

---

The cells in grids for operators and persistent data sets are identified by row-column coordinates, with each row and column indexed 0 through  $n-1$ .

## How the Performance Monitor Represents Data Flows

Flows of data (including virtual data sets), among operators and data sets in the step, are represented by solid or dashed lines (*arcs*). The performance monitor treats any data connection with no flow of data within a sampling interval as currently *inactive*, and by default does not display it. You set the sampling interval when you configure the performance monitor (see the section “Configuring the Performance Monitor” on page 7-4).

During the step's processing, the performance monitor first represents a data flow as a dashed line. After a minimum number of records (by default, 100,000) have been written to one processing node in the virtual data set, the performance monitor changes the virtual data set representation to a solid line. As explained on the section “Data Set Display Control” on page 7-7, you can use the **Options** dialog box to enable or suppress the display of inactive data flows. You can also use this dialog box to change the minimum number of records at which the performance monitor represents the flow with a solid line.

## Configuring the Performance Monitor

To configure the performance monitor for an entire application, you use the **Program Properties** dialog box, as described below:

1. Choose **Program -> Properties** to open the **Program Properties** dialog box.
2. From the **Orchview** tab, select the **Debug Options** button. This opens the **Step Debug** dialog box.
3. Set the parameters that enable the performance monitor, as follows:

**Enable:** Click to enable the performance monitor.

**Host running Orchview performance monitor:** The name of the UNIX processing node running the performance monitor. On MPP systems, this is the node from which the server is started. On other systems, this is the name of the node as it appears in the Orchestrate configuration file, using the node parameter.

**X Windows DISPLAY on which Orchview window appears:** The name of the machine on which the performance monitor output window is displayed. This entry is usually the network name of your PC.

**TCP port by which Orchview receives performance data:** The logical port number used to communicate information from your application to the performance monitor.

The port number must be different from all other performance monitor processes, and from that of other applications that allocate specific port numbers.

Many standard applications port numbers are in the 0-1000 range. Recommended values for the port number are between 10,000 and 60,000.

**Temporary Score File:** The name and path of the file used by the performance monitor to hold data. The performance monitor creates this file if it does not already exist.

**Display update interval (seconds):** The sampling interval for the performance monitor in one-second increments.

Increasing the number reduces the network bandwidth used by the monitor. Decreasing the number creates a finer granularity for viewing the step.

Shown below are example settings for these environment variables:

**Enable:** Checked

**Host running Orchview performance monitor:** node0

**X Windows DISPLAY on which Orchview window appears:** myPC:0

**TCP port by which Orchview receives performance data:** 22222

**Temporary Score File:** /home/user1/working\_dir/myscorefile

**Display update interval (seconds):** 10

## Controlling the Performance Monitor Display

This section describes how you can control the display of the performance monitor. Included in this section is information on:

- “General Display Control” on page 7-5
- “Operator Display Control” on page 7-6
- “Data Set Display Control” on page 7-7
- “Generating a Results Spreadsheet” on page 7-8

### General Display Control

The performance monitor contains the following general controls:

- In the menu bar, the **Options** selection includes several display options, such as setting the background color of the display window to black or white.
- In the menu bar, the **Help** selection accesses online help.
- Below the graphical display area, the message panel displays output messages from the performance monitor.
- The control panel lets you:
  - Automatically pan and zoom the display with the **Auto View** button.
  - Zoom in and out on the display.
  - Rotate the step both horizontally and vertically, so you can focus in on a particular operator or connection.
  - Change the width of the operator grids. Reducing the size of the grids can make a complex step easier to view, and it gives you the option of displaying only the data flow of the step.

## Operator Display Control

Not all operators execute on all nodes. By observing the data connections to an operator's grid, you can determine the number of nodes used to execute it. For example, if you have a 16-node system, each grid contains 16 cells. However, constraints applied to an operator or to the step can restrict an operator to executing on a subset of the 16 Orchestrate processing nodes. Only the cells corresponding to nodes actually executing the operator have a data connection. Cells with no connection correspond to nodes not used by the operator.

The performance monitor provides the following controls for operator display:

- To move an operator in the window, you use the middle mouse button (or both buttons on a two-button mouse) to click and drag the operator grid in the window.
- To display a window with a snapshot of the statistics about the operator, click the right mouse button on the operator-grid cell that represents the operator instance on which you want to obtain statistics. The statistics display includes the operator number and instance number of the operator cell on which you clicked, as shown below:

Operator number

Op: 10, parallel funnel(2)			
Operator	Rate	Records	Runtime (sec)
In:	0	8073	1
Out:	0	8073	2
Instance ( 1 )	Rate	Records	Runtime (sec)
In:	0	4036	0
Out:	0	4036	2

Instance number

The statistics window remains open as long as you hold down the right mouse button. To refresh the statistics display, release the mouse button and then click again to display the updated statistics.

- To display statistics about an operator for which you know the operator number and operator instance number, you can also use the menu to display the operator statistics window shown above. Choose the **Data -> Operator Stats** menu selection to open the **Specify Operator** dialog box. Enter the operator number into the **Specify Operator Number** field, and enter the instance number into the **Specify Instance Number** field. Then, click the **stats...** button to open the statistics window for the operator.

When opened via the menu, the window remains open until you dismiss it, and it is automatically updated every 5 seconds.

## Data Set Display Control

The performance monitor displays virtual data sets as dashed or solid lines, and it displays persistent data sets as grids. The performance monitor lets you control some aspects of the display of data sets, and to display statistical information about your data.

Not all persistent data sets are transferred using all processing nodes. From the performance monitor window, you can see the data connections to a data set's grid and therefore determine the number of nodes used to store the a data set. For example, if you have a 16-node system, each data-set grid contains 16 cells, but constraints applied to an operator or to the step restrict an operator to transferring the data set on fewer than 16 nodes. Only the data-set grid cells corresponding to nodes actually used for the data set have a data connection. Cells with no connection correspond to nodes that are not used to transfer the data set.

The performance monitor provides the following options for controlling the display of information about a data set:

- To display a snapshot of statistics about a data set, control-click the right mouse button on the data flow (solid or dashed line). The statistics window remains open as long as you hold down the right mouse button. To refresh the statistics display, release the mouse button and then control-click again to display the updated statistics.

The window includes the data-set number and the reading and writing operators, as shown below:

Data-set number
Writing operator
Reading operator

	Rate	Records	Runtime (sec)
DataSet Total:	0	8073	1
Arc ( 0 , 0 ):	0	2019	1

Data-set arc information

At the bottom of the window is the information on  $Arc(N, M)$ , where  $N$  is the instance number of the writing operator and  $M$  is the instance number of the reading operator. This window displays information about the partition of the data set connecting the output of instance 0 of the writing operator to the input of instance 0 of the reading operator.

You can determine the data-set number from this window in order to open the statistics window for the operator, described next.

- To display statistical information about a data set and a specific arc in it, choose the **Data -> DataSet Stats** menu selection. The statistics displayed include the time running, total records passed, and data-flow rate. The performance monitor updates this window continuously.
- To show the volume of records processed, select **Options** from the menu to open the **Options** dialog box. In that dialog box, check the selection **DS Spectrum Coloring Marker (No. of Records)**. By default, the data-flow line color is determined by the node number of the pro-

cessing node writing the data, and the color remains constant throughout processing. Setting the **DS Spectrum Coloring Marker** option causes the performance monitor to indicate record volume by cycling through the color palette, from orange to violet, changing color when your specified number of records has been processed.

- To show the rate of data flow, you can use either of two selections in the **Options** dialog box:
  - **DS Spectrum Coloring Marker (Records per Second)**. Check this option, and enter the number of records per second at which you want the color to change. The performance monitor cycles through the color palette (from orange to violet), changing color at the number of records you specify.
  - **DS Binary Rate Coloring (Records per Second)**. Using this option causes the performance monitor to display in red the data flows that transfer data below the cutoff rate, and to display in green the data flow arcs with a rate at or above the cutoff. Check the option and set the cutoff rate in the **Records per Second** field, and then press Enter.
- To set the number of records processed after which a virtual data set is displayed as a solid line, check the **Options** dialog box selection **Set DS Solid Line Marker (No. of Records)**. The default number of records is 100,000.
- To cause the performance monitor to display inactive data connections, use the **Options** dialog box selection **Show Data Set By Blockage (Solid Line)**. The performance monitor treats a data connection as *inactive* if no records have been transferred over it during one sampling period (approximately five seconds).

## Generating a Results Spreadsheet

The performance monitor lets you save to a file, in a spreadsheet layout, the information it gathers on record flow in your data set. The information is saved as tab-delimited ASCII text, so that you can open the file in any ASCII text editor. The format is also compatible with Microsoft Excel, so you can use Excel to view your results spreadsheet.

Use the following procedure save your record-flow data to a spreadsheet:

1. In the performance monitor, choose the **File -> Save Spread Sheet** menu command to open a dialog box prompting you for the name of the file.
2. Enter the name of the file.
3. Click the **Save Spread Sheet** button to save the current data flow information to the file.

This action saves the number of records transferred by every data-flow arc in the step at the time you click the **Save Spread Sheet** button. Clicking on the button again overwrites the contents of the file with new information.

4. View the file using either an ASCII text editor or in Microsoft Excel.

When you read the file into Excel, Excel opens a text import dialog box allowing you to specify the layout of the file. Use the default value of **Delimited** text to read the file, then click on the **Finish** button.



The following is an example of the contents of the saved spreadsheet:

Orchestrate Performance Spreadsheet, Copyright (C) 1995 - 2000  
Torrent Systems, Inc. All Rights Reserved.

Sequence Number: 3  
Spreadsheet Version: 1  
Current Date 1997 10 28  
Current Time 12:35:50  
Unique Orchestrate Step ID: 60346-878059706  
Step Start Date 1997 10 28  
Step Start Time 12:28:26

Data Section:

data saved: detailed data set flow volumes.  
begin data:

	DS0	DS1	DS2	DS3	DS4	DS5	DS6
(0,0)	673	673	1346	1346	1346	337	337
(0,1)	673	673	0	0	0	337	337
(0,2)	673	673	0	0	0	336	336
(0,3)	672	672	0	0	0	336	336
(1,0)	0	0	0	0	0	337	337
(1,1)	0	0	1346	1346	1346	337	337
(1,2)	0	0	0	0	0	336	336
(1,3)	0	0	0	0	0	336	336
(2,0)	0	0	0	0	0	337	337
(2,1)	0	0	0	0	0	337	337
(2,2)	0	0	1346	1346	1346	336	336
(2,3)	0	0	0	0	0	336	336

End Data  
End Data Section  
Sequence Number: 3

In the spreadsheet, the row labels (such as 0,0) specify the partition numbers of the reading and writing operators for each data-flow arc, in the form.

(writing\_partition, reading\_partition)

The column labels identify the data sets. To determine the data set that corresponds to the label for a column (such as DS0), control-click the right mouse button on the data flow arc in the performance monitor display. See the section "Data Set Display Control" on page 7-7 for information about data set arcs.

Each cell in the spreadsheet body corresponds to a single arc in the display of the performance monitor. The cell contains the number of records transferred from the writing operator to the reading operator for a partition of the data set.

## Creating Movie Files

The performance monitor *movie* feature lets you record and play back the run-time display of your program. You save one or more steps to a *movie* file, and you play it back in the performance monitor. Once the display information is saved, you can repeatedly play back the file.

To save a step to a movie file, choose the **File -> Save Movie As** menu command before you run the step. This command prompts you for the file name for the movie.

To play a movie file back, choose the **File -> Play Movie** menu command, and enter the name of a file containing a movie.

You can also view a step's configuration stored in a movie file, without playing back the entire movie. The step configuration is a snapshot of all operators and data sets in a step. To display the step configuration stored in a movie file, choose the **File -> View Step** menu command.

# 8: Partitioning in Orchestrate

**Partitioning** is the action of dividing a data set into multiple segments or *partitions*. Partitioning implements the “divide and conquer” aspect of parallel processing, where each processing node in your system performs an operation on a portion of a data set rather than on the entire data set. Therefore, your system produces much higher throughput than it does using a single processor.

One of the goals of Orchestrate is to insulate application developers from the complexities of partitioning. Usually, a parallel operator defines its own partitioning algorithm, or *method*, so that you do not have to modify how the operator partitions data.

However, Orchestrate allows you to specify an explicit partitioning method in certain circumstances. The first section of this chapter describes how Orchestrate partitions data during normal program execution and how you can control the partitioning behavior of an application.

Note that sequential Orchestrate operators do not use a partitioning method. Instead, a sequential operator defines a *collection method*. A collection method defines how a sequential operator combines the partitions of an input data set for processing by a single node. See the chapter “Collectors in Orchestrate” for more information.

This chapter contains the following sections:

- “Partitioning Data Sets” on page 8-1
- “Partitioning Methods” on page 8-3
- “Using the Partitioning Operators” on page 8-7
- “The Preserve-Partitioning Flag” on page 8-11

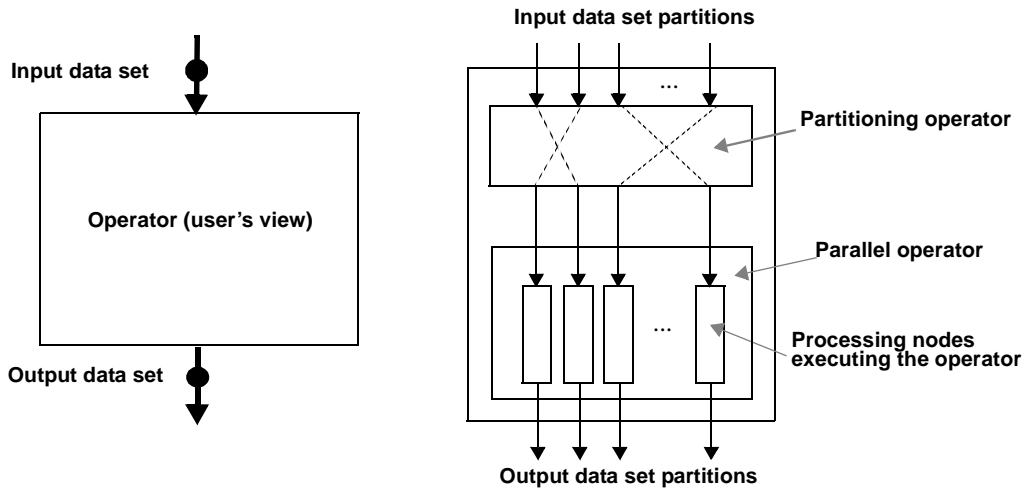
## Partitioning Data Sets

A record comprises one row of a data set and is the unit of partitioning in Orchestrate. An operator partitions an input data set by dividing it record by record to distribute the data set to all processing nodes executing the operator.

All records assigned to a single processing node are referred to as a data-set *partition*. An operator executing on a processing node performs an action only on those records in its input partition. All records on a processing node output by an operator are written to the same partition of the output data set. For each partitioning method, Orchestrate provides a *partitioning operator* (also called a partitioner), which you use to partition data for a parallel operator. Partitioning methods are described in the section “Partitioning Methods” on page 8-3, and the partitioning operators are introduced in the section “Using the Partitioning Operators” on page 8-7 and are described in detail in the *Orchestrate User's Guide: Operators*.

## Partitioning and a Single-Input Operator

Partitioning operators work closely with parallel operators, so that all partitioning and parallelism are hidden from the application user. In the following figure, the left-hand data-flow diagram is an application user's view of a parallel operator that takes one input data set and outputs one data set. The detailed diagram on the right shows the application developer's view: a partitioning operator partitions the data that the parallel operator then processes, and the parallel operator outputs a partitioned data set.



As shown in the right-hand diagram, the partitioning operator performs the following tasks:

- Takes as input a data set, which may have already been partitioned
- According to its partitioning method, determines the output partition for each record
- Writes each record to an output partition

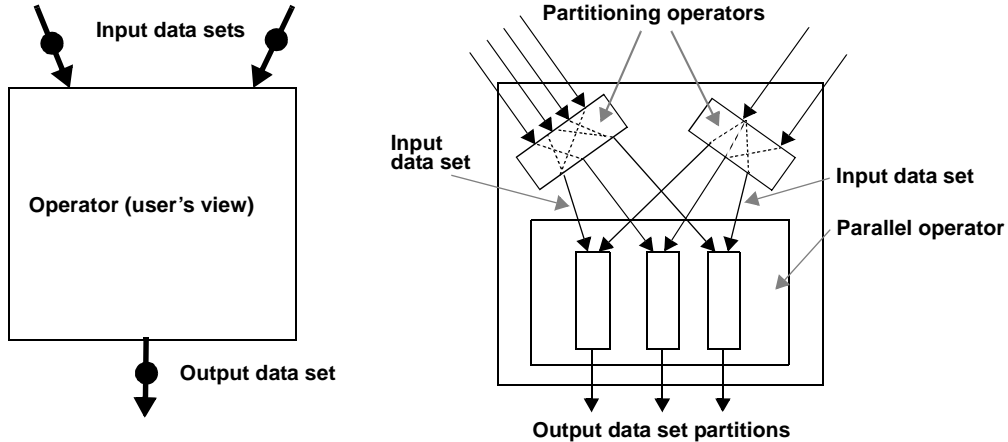
The parallel operator runs as a separate instance on each processing node in the system. The operator instance processes records from a single input partition, and it writes all its output to a single output partition.

## Partitioning and a Multiple-Input Operator

A multiple-input parallel operator can use a different partitioning operator to partition each input. However, the partitioning operators must create the same number of partitions for each input of the parallel operator.

In the figure below, the left-hand data-flow model is an application user's view of a two-input operator. The right-hand diagram shows the details: the operator takes each of its two inputs from a different partitioning operator. It also shows that each partitioning operator takes a different number

of partitions (four partitions to the left-hand partitioning operator and two partitions for the right-hand partitioning operator), as output by differently partitioned upstream operators.



The right-hand diagram shows that each partitioning operator creates three partitions for one of the two inputs to the parallel operator.

## Partitioning Methods

Each Orchestrate partitioning operator uses a different method to determine how to partition a data set. A partitioning method may be as simple as the random distribution of records, or it may involve complex analysis of the data.

### The Benefit of Similar-Size Partitions

In selecting a partitioning method, an important objective is to make all partitions similar in size, so that processing will be distributed fairly evenly among processing nodes. Greatly varied partition sizes can result in heavy record processing by some of your processing nodes and little processing by others.

For example, suppose that you need to partition a data set in which every record contains a zipcode field. You could select a partitioning method that partitions the data according to value of the zipcode field. If there are approximately the same number of records for each zipcode, your partitions will be of similar size. However, if most of your data set records have one zipcode value and few records have other zipcode values, your partitions will vary significantly in size.

## Partitioning Method Overview

Many Orchestrate operators specify a default partitioning method of *any*. The *any* method allows Orchestrate to partition the input data set in any way that it determines will optimize the performance of the operator. However, insertion of a partitioning operator in front of the operator overrides the *any* method.

Orchestrate supports a number of the following commonly used partitioning methods, briefly described below. The Orchestrate operators that implement these methods are described in the section “Using the Partitioning Operators” on page 8-7.

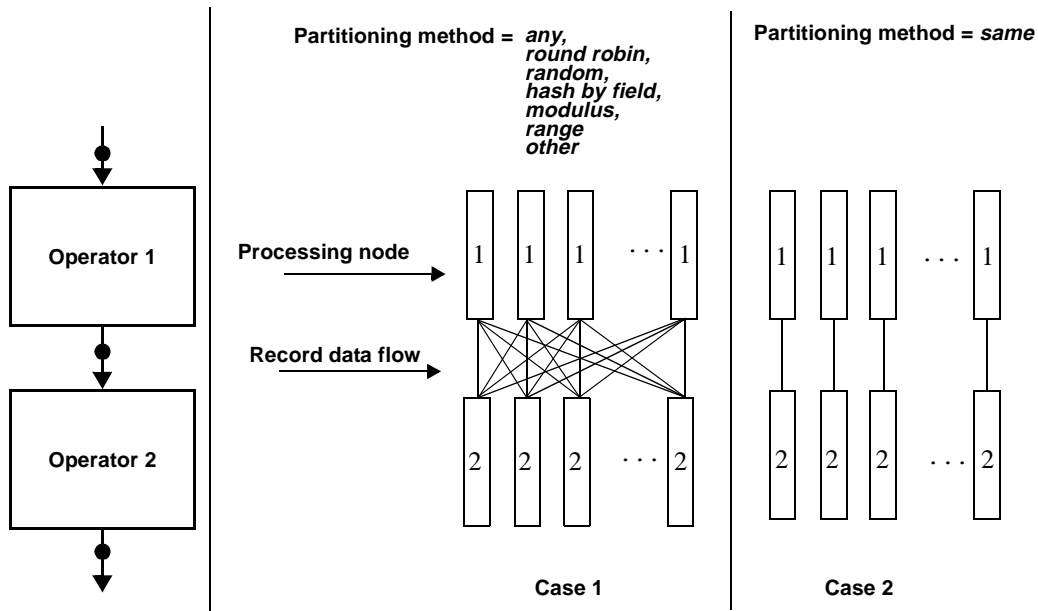
- *Round robin*: The first record goes to the first processing node, the second to the second processing node, and so on. When Orchestrate reaches the last processing node in the system, it starts over. This method is useful for resizing partitions of an input data set that are not equal in size. The *round robin* method always creates approximately equal-sized partitions.
- *Random*: Records are randomly distributed across all processing nodes. Like *round robin*, *random* partitioning can rebalance the partitions of an input data set to guarantee that each processing node receives an approximately equal-sized partition. The *random* partitioning has a slightly higher overhead than *round robin* because of the extra processing required to calculate a random value for each record.
- *Same*: The operator using the data set as input performs no repartitioning and takes as input the partitions output by the preceding operator. With this partitioning method, records stay on the same processing node; that is, they are not redistributed. *Same* is the fastest partitioning method.
- *Entire*: Every instance of an operator on every processing node receives the complete data set as input. It is useful when you want the benefits of parallel execution, but every instance of the operator needs access to the entire input data set. You are most likely to use this partitioning method with operators that create lookup tables from their input.
- *Hash by field*: Partitioning is based on a function of one or more fields (the hash partitioning keys) in each record. This method is useful for ensuring that related records are in the same partition.
- *Modulus*: Partitioning is based on a key field *modulo* the number of partitions. This method is similar to *hash by field*, but involves simpler computation.
- *Range*: Divides a data set into approximately equal-sized partitions, each of which contains records with key fields within a specified range. This method is also useful for ensuring that related records are in the same partition.
- *DB2*: Partitions an input data set in the same way that DB2 would partition it. For example, if you use this method to partition an input data set containing update information for an existing DB2 table, records are assigned to the processing node containing the corresponding DB2 record. Then, during the execution of the parallel operator, both the input record and the DB2 table record are local to the processing node. Any reads and writes of the DB2 table would entail no network activity. See the chapter on interfacing with DB2 in the *Orchestrate User's Guide: Operators* for more information.
- *Other*: You can define a custom partitioning operator by deriving a class from the C++ APT\_Partitioner class. *Other* is the partitioning method for operators that use custom partitioners. See the chapter on partitioning in the *Orchestrate/APT Developer's Guide* for more information.

mation.

## Partitioning Method Examples

This section describes examples of different partitioning methods used with several basic data-flows.

The following figure shows a data flow between two Orchestrate operators:



On the left, the operators appear in a data-flow diagram. The rest of the figure shows the internal data flow.

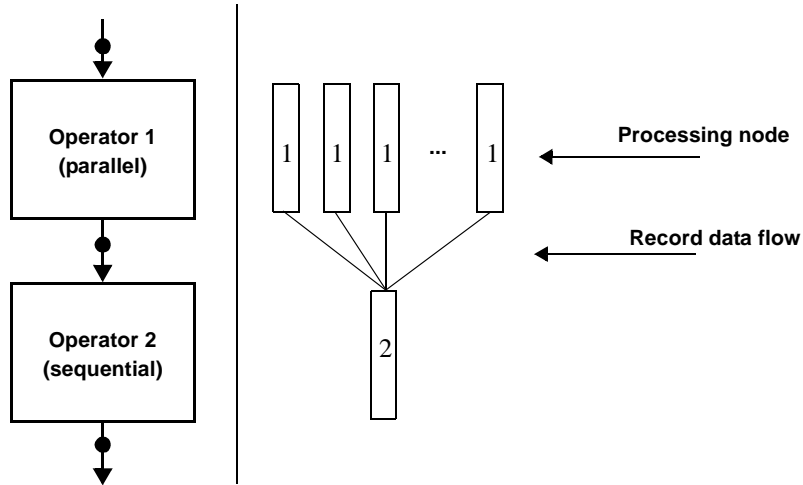
Operator 1 processes its input data set on multiple nodes. Each processing node receives a single partition of the input data set. As Operator 1 writes its results to its output data set, Operator 2 redistributes the records based on its partitioning method.

**Case 1:** If Operator 2 uses *any*, *round robin*, *random*, *hash by field*, *modulus*, *range*, or *other*, an output record from a node executing Operator 1 may be sent to any node executing Operator 2 because this second operation repartitions the data. Note that the number of partitions for Operator 1 and Operator 2 do not need to be equal.

**Case 2:** If Operator 2 uses *same*, each node executing Operator 2 inherits a complete partition of the data set as created by a node executing Operator 1. No repartitioning is performed. Note that the *any* partitioning method is treated as *same* if the input data set has its preserve-partitioning flag set. See the section “The Preserve-Partitioning Flag” on page 8-11 for more information.

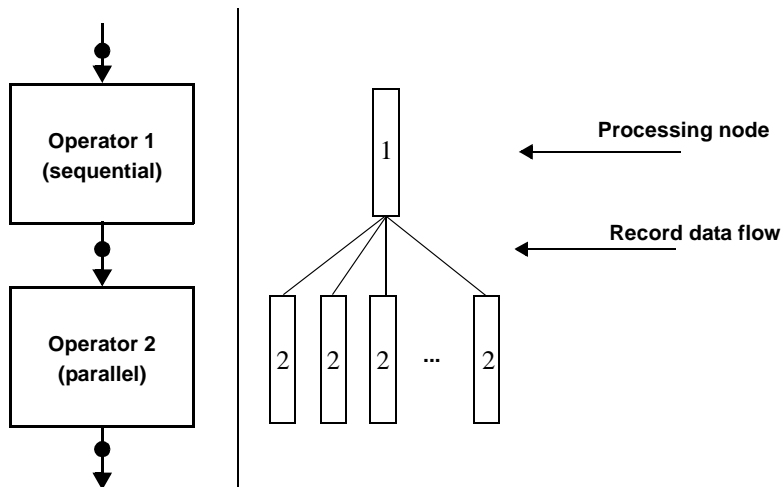
**Case 3** (not shown): If Operator 2 uses *entire*, each node executing Operator 2 receives a copy of the entire input data set.

**Case 4:** Orchestrate lets you include both parallel and sequential operators in a step. Sequential operators execute on a single processing node and therefore always take an entire data set as input. Sequential operators input the output of all processing nodes in the upstream, parallel operator. This *fan-in* operation is shown in the following figure:



You have the option of using a collector operator to control how the partitions are gathered for input to the sequential operator; see the chapter “Collectors in Orchestrate” for more information.

**Case 5:** This case is the converse of Case 4 above. When a sequential operator precedes a parallel operator, the parallel operator *fans out* the sequential operator's output to the processing nodes, as shown in the following figure:



This fan -out is controlled by the partitioning method of the parallel operator.



## Using the Partitioning Operators

Nearly all Orchestrate built-in operators have a predefined partitioning method. Only the `psort` operator and the `group` operator in hash mode. require that you specify a partitioning method.

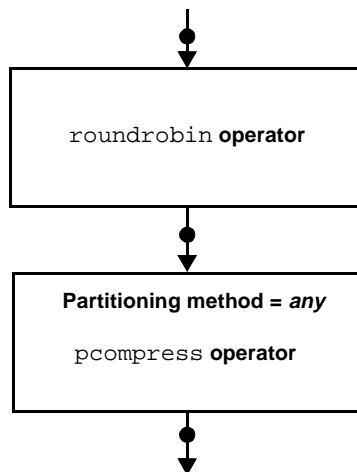
Many Orchestrate operators have the predefined partitioning method *any*. You can place any partitioning operator in front of an operator that specifies the *any* method. Overriding an operator's partitioning method is useful for controlling the distribution of records in your system. Note that inserting a partitioning operator before an operator with a partitioning method other than *any* causes an error.

You can also use a partitioning operator before you write a persistent data set to disk. This allows you to distribute your data set as a preprocessing operation before the write.

Orchestrate has a partitioning operator for each partitioning method described in the section “Partitioning Method Overview” on page 8-4, as follows:

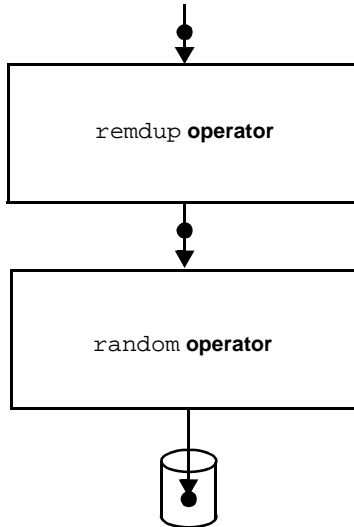
- `roundrobin` operator
- `random` operator
- `same` operator
- `entire` operator
- `hash` operator
- `range` operator
- `modulus` operator

For example, the following figure shows the `roundrobin` operator inserted before the Orchestrate `pcompress` operator:



A benefit of the `roundrobin` operator is that it creates relatively equal-sized partitions in its output data set. If processing previous to the `roundrobin` operator, such as a filter or duplicate-record removal, creates partitions that are unequal in size, you can use `roundrobin` to redistribute records before inputting them to the `pcompress` operator.

The following figure shows the `random` operator used to distribute the output partitions of the `remdup` operator before writing a data set to disk:



Like the `roundrobin` operator, the `random` operator also performs a load balancing form or repartitioning so that the partitions in its output data set are approximately equal in size.

---

**Note:** The optional Orchestrate Analytics Library includes the specialized operator `smartpartitioner`, which uses model-building techniques to partition a data set. The Analytics Library manual describes how to use `smartpartitioner` to support parallelizing modeling algorithms.

---

## Choosing a Partitioning Operator

This section contains an overview of how to use the partitioning operators in an Orchestrate application. See the *Orchestrate User's Guide: Operators* for a complete description of each operator.

### General Guidelines for Selecting a Partitioning Method

Following are some general guidelines for choosing a partitioning method and, therefore, an Orchestrate partitioning operator:

- Choose a partitioning method that creates a large number of partitions. For example, hashing by the first two digits of a zipcode produces a maximum of 100 partitions. This may not be a large enough number of partitions for a particular parallel processing system. To create a greater number of partitions, you could hash by five digits of the zipcode, to produce as many as 10,000 partitions. Or, you could combine a two-digit zipcode hash with a three-character name hash, to yield 1,757,600 possible partitions ( $100 * 26 * 26 * 26$ ).
- Choose a partitioning method that creates partitions that are roughly uniform in size, as dis-

cussed in the section “Partitioning Methods” on page 8-3.

- Make sure the partitioning method matches the action of the operator. For example, if you are performing a comparison, choose a partitioning scheme that assigns related records to the same partition, so the processing node can compare them.
- Do not use a complicated partitioning method with an operator that performs a uniform operation on all records, for example, extracting data. If the processing operation performed on each record is not related to any other records, use *any*. Remember that you are processing huge amounts of data. Any additional processing on an individual record is multiplied by the number of records in the data set.

## Keyed and Keyless Partitioning

*Keyed* partitioners examine one or more fields of a record, called the *partitioning key* fields, to determine the partition to which to assign the record. The keyed partitioning operators are `hash`, `modulus`, and `range`.

*Keyless* operators determine the partition for a record without regard to the record itself. The keyless partitioning operators are the `random`, `roundrobin`, and `same`.

The keyed partitioning operators `hash` and `range` are discussed in more detail below.

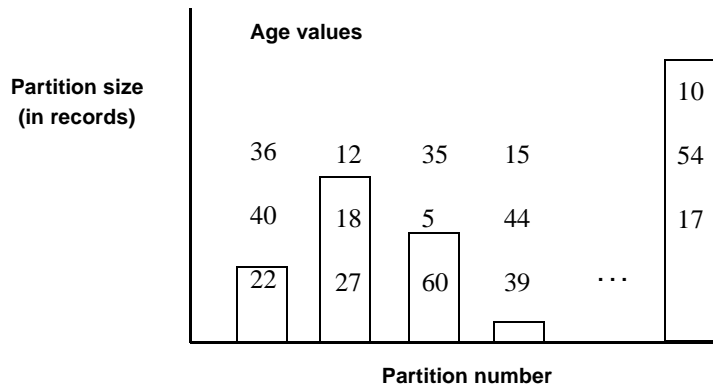
## Hash Partitioning

Hash partitioning is useful when you need to compare records, such as in a sort or join operation. You use `hash` to hash-partition the data set, to assign records with the same partitioning key values are to the same partition. Then, you sort or join the individual partitions of the data set. While you cannot control which partition receives which records, the hash partitioner guarantees that all records with the same hashing keys are assigned to the same partition.

You can also use hash partitioning when the processing operation relies on a particular ordering or relationship among the records of each partition. For example, suppose that you use the `remdup` operator to remove duplicate records from a data set, according to the first-name and last-name fields. If you randomly partition records, duplicate records might not be in the same partition and, therefore, would not be detected and removed.

While the `hash` partitioning operator guarantees that all records with the same hashing keys will be assigned to the same partition, it does not control the size of each partition. For example, if you hash partition a data set based on a `zipcode` field, where a large percentage of your records are from one or two zipcodes, a few partitions will contain most of your records. This behavior can lead to bottlenecks because some nodes will be required to process more records than other nodes.

For example, the figure below shows the possible results of hash partitioning a data set using the field `age` as the partitioning key:



The hash operator assigns records with the same `age` value to the same partition. As evident in this figure, the key values are randomly distributed among the partitions, but the number of keys per partition is the same.

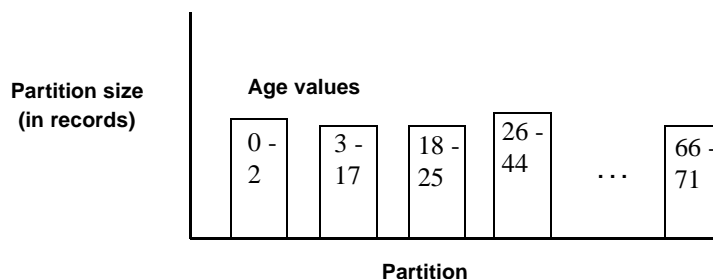
The partition sizes resulting from a hash partitioner depend on the distribution of records in the data set. The example distribution shown above could be created when the data set contains an unequal record distribution based on the `age` field. Note that you may be able to choose a set of partitioning keys for this example that creates equal-sized partitions. Or, your application may not care that the partitions are of different sizes.

See the chapter on the hash operator in the *Orchestrate User's Guide: Operators* for more information.

## Range Partitioning

The range partitioning operator guarantees that all records with the same partitioning key values are assigned to the same partition and that the partitions are approximately equal in size. This means that all nodes perform an equal amount of work when processing the data set.

Using the range operator always results in partitions with a size distribution similar to that in the figure below:



As shown in the figure, all partitions are approximately the same size. In an ideal distribution, every partition would be the same size. However, you will usually observe small differences in partition size, based on your choice of partitioning keys.

In order to size the partitions, the `range` operator orders the partitioning keys. The `range` operator then calculates partition boundaries based on the partitioning keys in order to evenly distribute records to the partitions. The above figure shows that the distribution of partitioning keys is not even; that is, some partitions contain many partitioning keys, and others contain relatively few. However, based on the calculated partition boundaries, the number of records in each partition is approximately the same.

The `range` operator offers the advantages of keyed partitioning, while guaranteeing similar-size partitions. The `random` and `round robin` partitioning method also guarantee that the partitions of a data set will be equivalent in size. However, these two partitioning methods are keyless and, therefore, do not allow you to control how the records of a data set are grouped within a partition.

See the chapter on the `range` operator in the *Orchestrate User's Guide: Operators* details on using that operator.

## The Preserve-Partitioning Flag

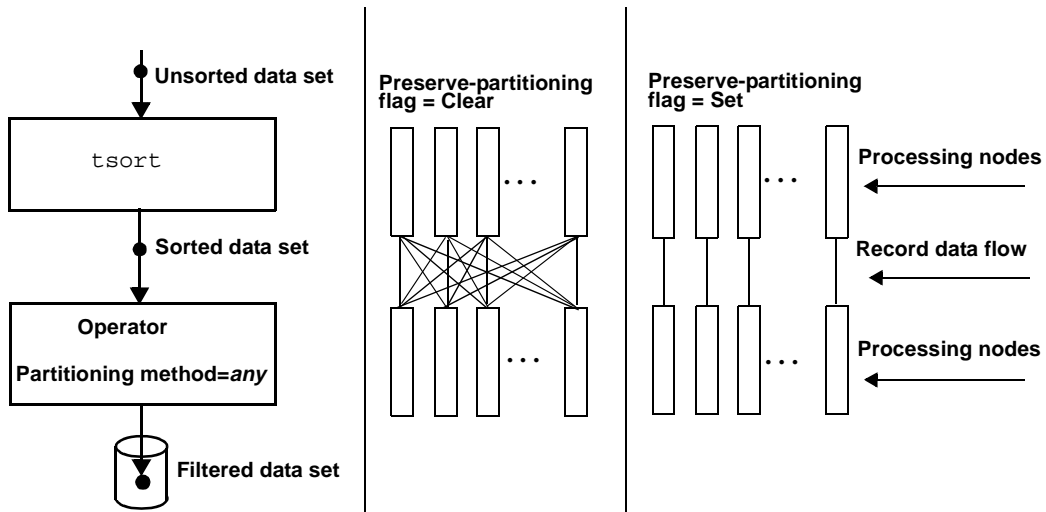
Some Orchestrate operators produce as output a data set with a specific partition layout. For example, the `tsort` operator produces a data set in which the records in each partition are sorted. If you then use the output data set from `tsort` as input to an operator with a partitioning method other than `same`, Orchestrate by default repartitions the data set and consequently destroys the sorted order of the records.

To let you control automatic repartitioning, Orchestrate provides the `preserve-partitioning` flag, which when set, prevents repartitioning of the data set. In some cases, Orchestrate automatically sets the preserve-partitioning flag, and in others, you specify the flag's setting when you create your step. Orchestrate's setting of the preserve-partitioning flag and the propagation of the setting, as well as how you specify the setting when you create a step, are described in the section "Manipulating the Preserve-Partitioning Flag" on page 8-13.

## Example of the Preserve-Partitioning Flag's Effect

In the figure below, the data-flow diagram on the left shows the `tsort` operator outputting data that is partitioned and sorted. The `tsort` output is then input to another operator, which has a partitioning method of `any`. The diagram in the middle shows that with the preserve-partitioning flag clear, the

data is automatically repartitioned before it is input to the operator. The right-hand diagram shows that with the preserve-partitioning flag set, the partitioning is preserved.



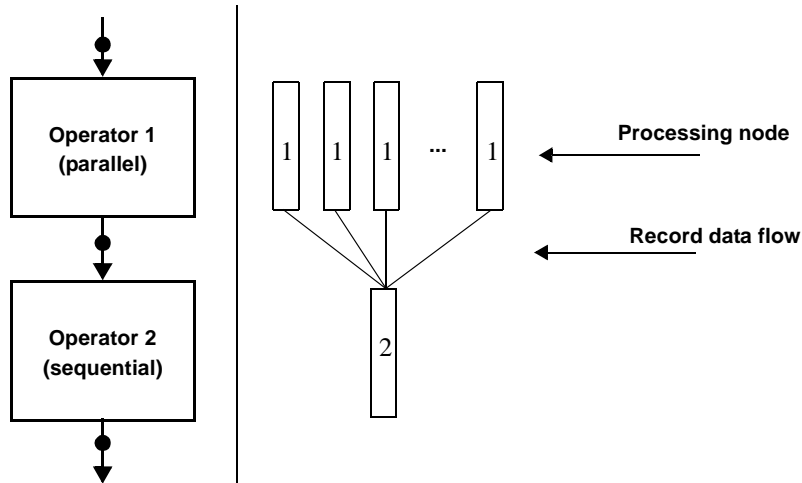
Only operators that allow a partitioning method of *any* or *same* allow you to set the preserve-partitioning flag for an input data set. In fact, operators that specify the *same* partitioning method issue a warning if the preserve-partitioning flag is not set for an input data set. If the flag is set for an input data set to a partitioning operator, Orchestrate issues a warning and repartitions the data set.

The state of the preserve-partitioning flag is stored to disk along with the records of a persistent data set. Therefore, when you read a persistent data set into Orchestrate, the saved state of the preserve-partitioning flag will control how the data set is partitioned.

You need to be concerned with the preserve-partitioning flag only if you include operators in your application that explicitly set the flag. Because most Orchestrate operators ignore the flag when it has not been set, you can create an entire application without manipulating the flag.

## Preserve-Partitioning Flag with Sequential Operators

Sequential operators use as input the output of all processing nodes from the preceding operator, as shown in the following figure:



In this example, the sequential operator has to repartition the input data set, regardless of the state of the preserve-partitioning flag, because all partitions must be collected into a single input stream. Orchestrate issues a warning if the preserve-partitioning flag is set and a sequential operator repartitions a data set. In the example above, if Operator 1 is sequential, that is, its output data set has only one partition, Orchestrate will not issue the warning.

See the section “Partitioning Method Examples” on page 8-5 for more information on how Orchestrate performs partitioning with parallel and sequential operators.

## Manipulating the Preserve-Partitioning Flag

A data set's preserve-partitioning flag may be set indirectly by Orchestrate as it executes your application or directly by you using the **Advanced** tab area of the **Link Properties** dialog box for the data set. The following rules define how the preserve-partitioning flag can be manipulated:

**Rule 1.** If any data set input to an operator has the preserve-partitioning flag set, Orchestrate sets the preserve-partitioning flag in all the operator's output data sets.

This means that Orchestrate automatically propagates the preserve-partitioning flag from an input data set to an output data set. This is necessary when you want to perform several actions on a carefully partitioned data set.

**Rule 2.** An operator can set or clear the preserve-partitioning flag of an output data set as part of writing its results to the data set.

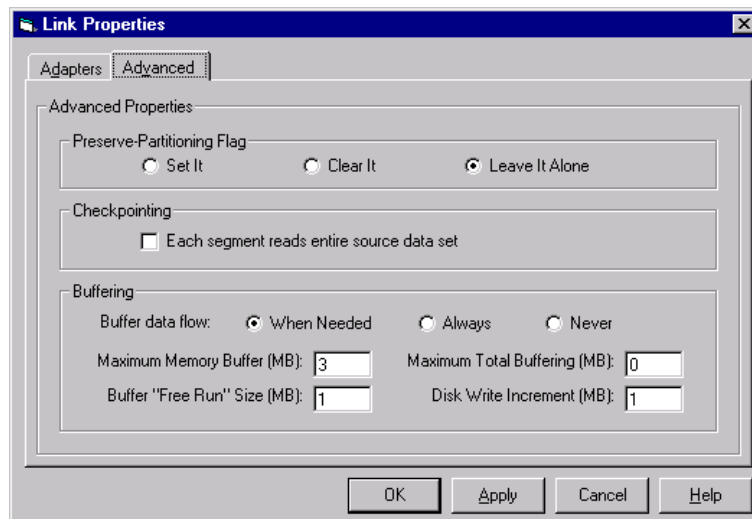
The ability to manipulate the preserve-partitioning flag is required by operators that create carefully partitioned data as output. For example, the `tsort` operator sets the preserve-partitioning flag in its output data set.

Several Orchestrate operators set the preserve-partitioning flag in the output data set as part of normal execution. These operators include:

- The `tsort` operator
- The `psort` operator
- The `pcompress` operator (compress mode)
- The `encode` operator (encode mode)
- The hash partitioning operator
- The range partitioning operator

**Rule 3.** You can directly set or clear the preserve-partitioning flag for any data set using the **Advanced** tab area of the **Link Properties** dialog box. An operator cannot modify the preserve-partitioning flag of a data set if the flag has been explicitly set or cleared using these arguments; an attempt by an operator to modify the flag is ignored. This rule means that an Orchestrate application programmer has final control of the state of the preserve-partitioning flag.

In order to set the flag, double click on the link to open the **Link Properties** dialog box, then select the **Advanced** tab, as shown below:



You can use this dialog box to set the flag, clear the flag, or do nothing (the default).

## Example: Using the Preserve-Partitioning Flag

This section presents a sample Orchestrate step that uses four operators and five data sets. The operators are the following:

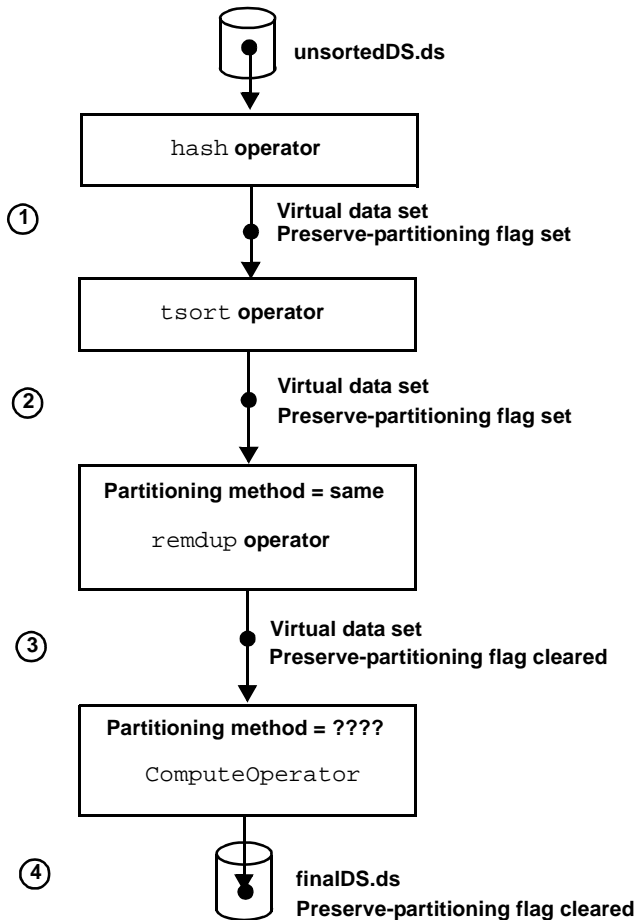
- The Orchestrate `hash` operator, which hash-partitions the input data set and sets the preserve-partitioning flag in its output data set.
- The Orchestrate `tsort` operator.



- The Orchestrate `remdup` operator, which specifies a partitioning method of *same*. This operator removes duplicate records from its input data set. The input data set is assumed to be sorted.
- `ComputeOperator`, a third-party operator which calculates values on an input data set. This operator assumes no order in its input data set and defines its own partitioning method.

In this example, you manipulate the preserve-partitioning flag as required by each operator.

The figure below shows the data-flow diagram for this example:



The state of the preserve-partitioning flag for each of the data sets is described below:

1. The `hash` operator sets the preserve-partitioning flag in its output data set (Rule 2).
2. The `tsort` operator sets the preserve-partitioning flag on the output data set (Rule 2).
3. Note that you can explicitly clear the preserve-partitioning flag in the output data set, overriding the default `tsort` setting (Rule 3). `remdup` creates a single output data set. Because the input data set has its preserve-partitioning flag set, the output data set also has its preserve-partitioning flag set (Rule 1).

4. However, the next operator, `ComputeOperator`, does not care about ordering to perform its action; therefore, `ComputeOperator` should be allowed to repartition its input data sets. Use the **Link Properties** dialog box to clear the flag. `ComputeOperator` writes its output to a persistent data set. Since `ComputeOperator` takes as input a data set with the preserve-partitioning flag cleared, and it does not modify the flag of its output data set (Rules 1 and 2), the flag is cleared in its output data set.

The state of the preserve-partitioning flag is stored to disk along with the records of the data set.

This example shows how you can manipulate the preserve-partitioning flag as part of an Orchestrate application. Note, however, that the partitioning rules are designed so that your step functions properly if you ignore the flag. However, because you limit the system ability to repartition data in this case, your application may not function as efficiently as possible.

## 9: Collectors in Orchestrate

**Partitioning** is the process by which a parallel operator divides a data set into multiple segments, or *partitions*. Each processing node in your system performs an operation on one partition of a data set, rather than on the entire data set. A *collector* defines how a sequential operator combines the partitions of an input data set for processing by a single node. Collecting for sequential operators is the inverse of partitioning for parallel operators.

Usually, a sequential operator defines its own collection algorithm or *method*, and you do not have to modify this method. However, in certain circumstances Orchestrate gives you the option of modifying an operator's collection method.

Building on information in the chapter “Partitioning in Orchestrate”, this chapter describes how sequential operators perform collection. It then describes how to select and use collection operators to modify the partitioning behavior of an operator. This chapter contains the following sections:

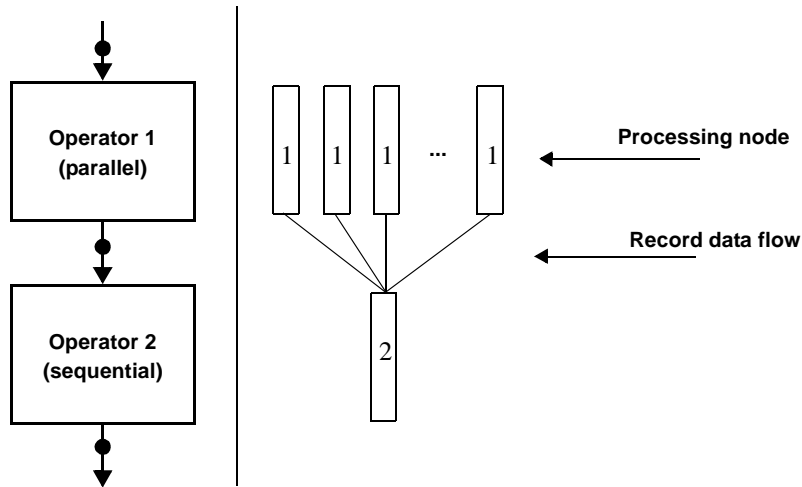
- “Sequential Operators and Collectors” on page 9-1
- “Choosing a Collection Method” on page 9-3
- “Setting a Collection Method” on page 9-4

For details on using the Orchestrate operators described in this chapter, see the *Orchestrate User's Guide: Operators*.

### Sequential Operators and Collectors

Orchestrate allows you to use both parallel and sequential operators in a step. Parallel operators execute on multiple processing nodes, where each node receives a partition of an input data set. Sequential operators execute on a single processing node, which receives all partitions of an input data set.

When a sequential operator takes as input a data set with multiple partitions, the operator must combine all partitions into a single input stream. This *fan-in* operation is shown in the following figure:



In this figure, Operator 2 is a sequential operator that combines the partitions of its input data set. Once these partitions have been combined, all partition boundaries are lost. This process of combining the input partitions by a sequential operator is called *collecting* the partitions. The mechanism used by a sequential operator to combine the partitions is called a *collector*.

A collector defines the way a sequential operator combines the partitions of an input data set for processing by a single node. Collectors are the inverse of partitioners, which define how a parallel operators distribute input data sets over multiple processing nodes.

Sequential operators offer various algorithms, called *collection methods*, that control the way an operator combines partitions. A sequential operator with multiple inputs can define one collection method for all input data sets, or it can use a different method for each input. The following section describes the collection methods available for sequential operators.

## Sequential Operators and the Preserve-Partitioning Flag

As described in the chapter “Partitioning in Orchestrate”, you can set the preserve-partitioning flag for a data set, to prevent its repartitioning by a parallel operator that uses a partitioning method of *any*. However, the preserve-partitioning flag applies only to parallel operators.

A *sequential* operator repartitions an input data set without regard to the state of its preserve-partitioning flag. Before a sequential operator repartitions an input data set with its preserve-partitioning flag set, Orchestrate issues a warning. If the input data set has only one partition, no warning is issued.

## Collection Methods

A collection method may be as simple as combining the partitions on a first-come first-served basis, in which the sequential operator processes records in the order in which they are received from the preceding operator. More complex collection methods may determine the collection order from information in the records of the data set.

Orchestrate also supports the following collection methods:

- *Any*: By default, Orchestrate built-in operators in sequential mode use the *any* collection method. (The one exception is the `group` operator in sort mode, for which you must specify a collection method.) With the *any* method, the operator reads records on the first-come first-served basis. Operators that use the *any* method allow you to override that collection method with another.
- *Round robin*: Read a record from the first input partition, then from the second partition, and so on. After reaching the last partition, start over. After reaching the final record in any partition, skip that partition in the remaining rounds.
- *Ordered*: Read all records from the first partition, then all records from the second partition, and so on. This collection method preserves the order of totally sorted input data sets. In a totally sorted data set, both the records in each partition and the partitions themselves are ordered. See the chapters on the Sorting Library in the *Orchestrate User's Guide: Operators* for more information on totally sorting a data set.
- *Sorted merge*: Read records in an order based on one or more fields of the record. The fields used to define record order are called *collecting keys*. You use the `sortmerge` collection operator to implement the sorted merge collection method.
- *Other*: You can define a custom collection method by deriving a class from `APT_Collector`. Operators that use custom collectors have a collection method of *other*. See the chapter on collectors in the *Orchestrate/APT Developer's Guide* for information on creating a collection method.

## Choosing a Collection Method

When you choose a collection method, take into account the particular action of its associated operator. The built-in collection methods *any*, *round robin*, and *ordered* are keyless. A keyless collector does not rely on information in the records to define the order of records read by the operator.

Unless your sequential operator requires a deterministic order for processing records, the *any* collection method is likely to serve your needs. For more control over the order of records processed by the operator, you can use the *ordered* method, the *sortmerge* collection operator, or a custom collector that you define.

Operators use *any* when the location of the record in the data set is irrelevant and the operator performs the same operation on every record. For example, in an unsorted data set, each record has no relationship to the record immediately before or after it. When you specify *any*, your sequential operator is never forced to wait for a record from a particular partition.

The *ordered* method requires that all records are read from partition 0 before any records from partition 1 are read. Even if all records of partition 1 are ready for processing before all records of partition 0, the sequential operator must wait in order to process the partitions in order, possibly creating a processing bottleneck in your application. However, that the *ordered* collection method is necessary if you want to preserve the sort order when you process a totally sorted data set with a sequential operator.

## Setting a Collection Method

To set an operator's collection method, you use one of the three Orchestrate collection operators:

- `roundrobin_coll`
- `ordered`
- `sortmerge`

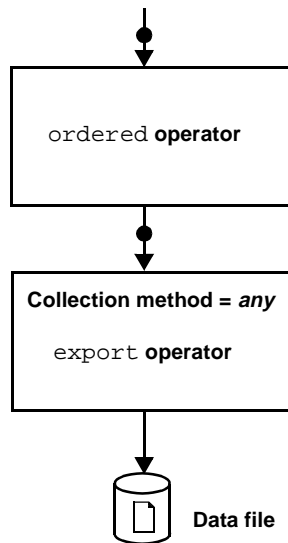
These operators implement respectively the *round robin*, *ordered*, and *sorted* merge methods (see the section "Collection Methods" on page 9-3).

The output of a collection operator must be one of the following:

- A virtual data set that is input to a sequential operator that uses the *any* collection method.
- A persistent data set. If the data set exists, it must contain only one partition.

## Collection Operator and Sequential Operator with Any Method

The output virtual data set from a collection operator overrides the collection method of an operator using the *any* collection method. For example, the following figure shows the `ordered` operator inserted before the Orchestrate `export` operator:



This example uses the `ordered` operator to read all records from the first partition, then all records from the second partition, and so on. This collection method preserves the order of the input data set that has been totally sorted. The `export` operator can then write the data set to a single data file that contains ordered partitions.

---

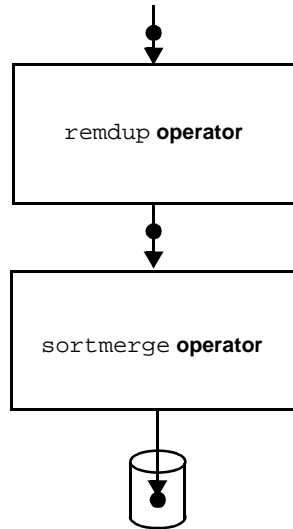
**Note:** You can insert a collection operator before an operator that uses the *any* collection method. Inserting a collection operator before an operator with another collection method causes an error.

---

## Collection Operator before Write to Persistent Data Set

You can insert a collection operator before the write operation to a persistent data set. Inserting a collection operator before the write operation allows you to control how the partitions of the data set are collected for writing to a single partition.

For example, the following figure shows the `sortmerge` operator used to collect the output partitions of the `remdup` operator:



The `sortmerge` operator collects records based on one or more collecting keys.



# 10: Constraints

A parallel processing system contains multiple processing nodes and multiple disk drives. Orchestrate's view of your system is controlled by the contents of the Orchestrate configuration file.

The configuration file includes definitions for the default node and disk pools. Most Orchestrate operators by default execute on all processing nodes that are members of the default node pool. In addition, Orchestrate data sets are stored on all disk drives in the default disk pool.

However, you can limit the processing nodes used by a specific operator or an entire step or the disks used to store a data set. For example, an operator may use system resources, such as a tape drive, not available to all nodes, or the action of the operator may be memory intensive and you want to execute the operator only on nodes with a large amount of memory.

To limit an operator or step to specific processing nodes, you impose a *constraint*. A constraint configures an operator or step to execute only on a particular set of processing nodes. You can also constrain a data set to a specific set of disks on a specific set of processing nodes.

This chapter first introduces constraints, then describes how to apply constraints to both your code and to your data. The chapter contains the following sections:

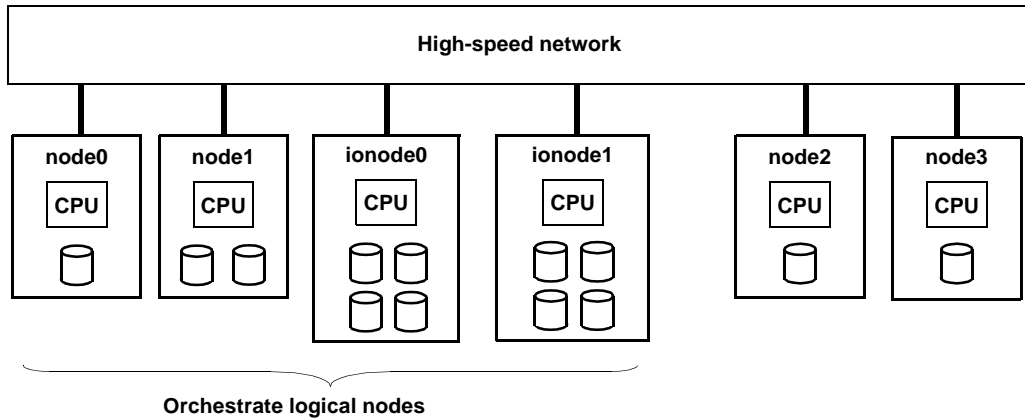
- “Using Constraints” on page 10-1
- “Using Constraints with Operators and Steps” on page 10-5
- “Data Set Constraints” on page 10-9

## Using Constraints

An Orchestrate application's view of your system is defined by the current Orchestrate *configuration* file. This file describes the processing nodes and disks drives connected to each node allocated for use by Orchestrate. Whenever it invokes an application, Orchestrate first reads the configuration file to determine allocated system resources, and it then distributes the application accordingly.

Whenever you modify your system by adding or removing nodes and disks, you need to make a corresponding modification to the Orchestrate configuration file. Then, the next time you start an application, Orchestrate reads the modified configuration file and automatically scales the application to fit the new system configuration, without requiring you to modify the application itself.

For example, the following figure shows a six-node MPP system, with four nodes configured as logical nodes for Orchestrate applications:



In this configuration, Orchestrate applications run on nodes 0 and 1 and I/O nodes 0 and 1, but not on nodes 2 and 3. Suppose your system gains a node, `node4` (with processor and disk), that you want Orchestrate to use in running your application. You modify your configuration file to allocate `node4` for Orchestrate processing and I/O. Then, on its next run your application would use `node4` according to your modified configuration file.

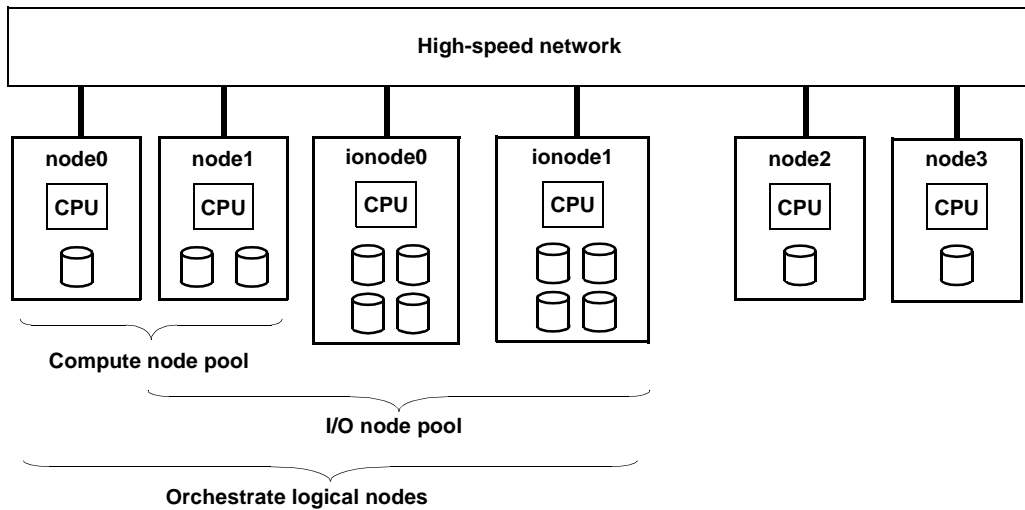
See the *Orchestrate Installation and Administration Manual* for detailed information on creating and administering configuration files.

## Controlling Where Your Code Executes on a Parallel System

The Orchestrate configuration file provides a great deal of flexibility in controlling the processing nodes that execute your application and the disk drives that store your data. All the processing nodes in your system may not be identically configured — some nodes may have a large amount of data storage, while other nodes have a large amount of physical memory that could be advantageous in performing complex calculations. Your application may use an operator requiring a particular system resource, such as a tape drive, that is not available to all nodes.

You can use the Orchestrate configuration file to define subgroups of nodes, called *node pools*, within your group of logical nodes. Using node pools, you select the specific nodes on which you want to perform certain actions.

The following figure shows an example of the nodes specified for use by Orchestrate applications:



In this example, the group of Orchestrate logical nodes is divided into two node pools: compute nodes and I/O nodes. A processing node may be a member of multiple pools; in this example, `node1` is part of both the compute node pool and the I/O node pool.

You can choose to execute an entire step, or any operator within the step, on all four processing nodes or on either node pool. Specifying the processing nodes that execute your application is called *constraining* the application. A *node pool* constraint limits execution (processing and I/O) to the nodes in the pool(s) that you allocate to the step or operator.

You can apply two other types of constraints to control the processing nodes that execute your code:

- *Resource* constraints limit the execution of an operator or step to nodes that have a specific resource, such as a disk or scratch disk (for temporary storage), or a resource in a specified resource pool.
- *Node map* constraints specify the list of processing nodes for a specific run of an operator. A node map constraint applies only to the particular operator invocation and while in effect, overrides any other constraint (node pool or other resource) previously specified for the step.

See the section “Using Constraints with Operators and Steps” on page 10-5 for more information on all three types of constraints.

## Using Node Constraints on a Stand-alone SMP

A stand-alone SMP (symmetric multiprocessing system) uses multiple CPUs that share system resources such as memory, disk drives, and network connection. This section describes how node constraints affect application execution on a stand-alone SMP.

For each operator in a step, Orchestrate creates one UNIX process for each Orchestrate processing node defined in the Orchestrate configuration file. On an SMP, each CPU executes a different process, allowing multiple processes to execute simultaneously.

The degree of parallelism in an Orchestrate application is determined by the number of Orchestrate processing nodes that you define in the configuration file. When applying node constraints on a stand-alone SMP, you can control the degree of parallelism (number of processes) that Orchestrate uses to execute an operator or step. Note, however, that you cannot select a particular CPU to execute a particular process.

Suppose, for example, that your SMP has four CPUs. You can define four Orchestrate processing nodes for the SMP, so that your application executes in a four-way parallel mode. However, you can also define only two Orchestrate processing nodes for the SMP, so that your application executes in a two-way parallel mode.

---

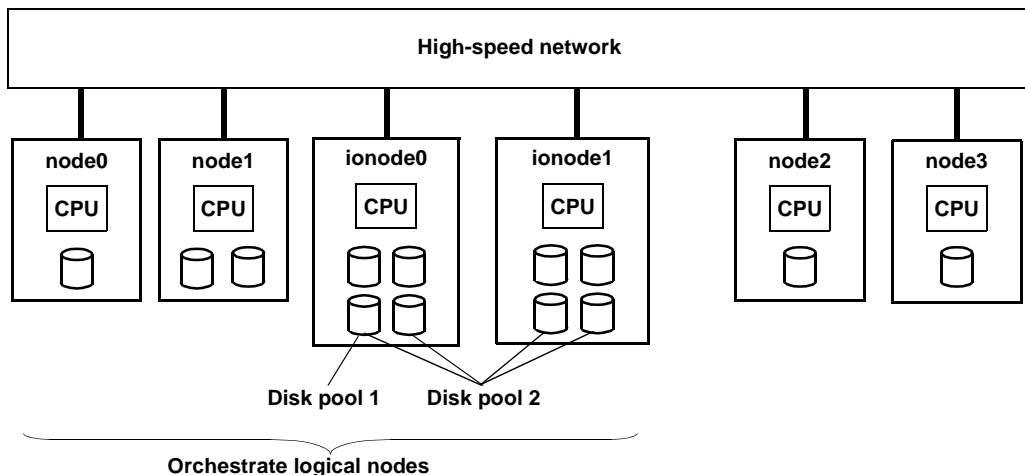
**Note:** Torrent recommends that you initially define one Orchestrate processing node for every two CPUs in an SMP. Later, during application testing and evaluation, you can modify this configuration to determine the optimal configuration for your system and application.

---

## Controlling Where Your Data Is Stored

Orchestrate lets you designate the logical nodes and the disk drives that store your application data. To control storage of an Orchestrate data set, you use the Orchestrate configuration file to define one or more *disk pools*, which are groups of disks that store your data. The disks in a disk pool can be all on one node or on multiple nodes.

For example, the following figure shows a system that defines two disk pools:



In this example, `ionode0` has one disk in disk pool 1 and two disks in disk pool 2. Also, `ionode1` has two disks in disk pool 2.

You can constrain an operator to use a particular disk pool; see the section “Data Set Constraints” on page 10-9 for more information.

## Using Constraints with Operators and Steps

You use the Orchestrate configuration file to set up node pools and disk pools. This section first briefly describes how to use your configuration file to set up node pools and disk pools. (For details on configuration file statements, see the *Orchestrate Installation and Administration Manual*.) It then describes how to use the constraints in execution of steps and individual operators and in storage of your data.

## Configuring Orchestrate Logical Nodes

The configuration file contains a node definition for each logical node that can be used by an Orchestrate application. The node definition can specify node pools and resource pools, as shown in the following example:

```
node "node0" {
    fastname "node0_css"
    pools "" "node0" "node0_css" "compute_node" /* node pools */
    resource disk "/orch/s0" {pools "" "pool1"}
    resource scratchdisk "/scratch" {}
}
```

In this example:

- The node argument, `node0`, is the node name.
- The `fastname` argument, `node0_css`, is the name used internally for data transfers.
- The `pools` option lists the node pools of which this node is a member.

The first argument `""` is the default node pool; the other arguments are node pools `node0`, `node0_css`, and `compute_node`. If you do not assign a node to any node pools, it is automatically a member of the default node pool.

- The `resource resource_type "directory" {resource_pools pl...pn}` option defines a disk or scratch disk connected to the node. You can optionally specify the disk's membership in non-default disk pools by including a `pools` clause inside the required braces `{}`; if you also want the disk or scratch disk to belong to the default disk pool, you must list it. Without a `pools` clause, the disk is automatically a member of the default disk pool.

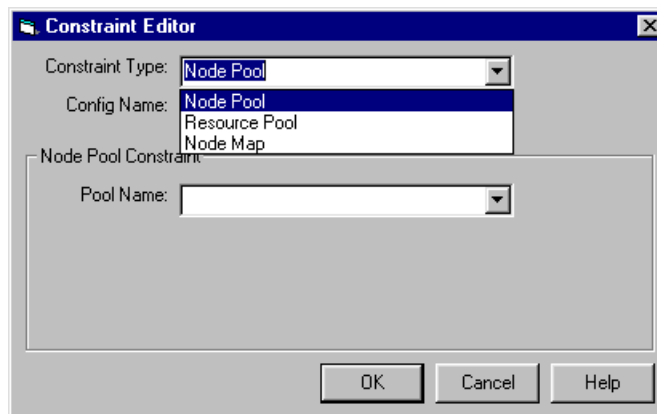
In the example, the `resource disk` clause assigns file directory `/orch/s0` to this node for permanent storage, and assigns the disk to the default pool and to `pool1`. The `resource scratchdisk` clause assigns to this node the file directory `/scratch`, for temporary storage; the empty braces `{}` indicate that the scratch disk is in the default scratch disk pool only.

## Using Node Pool Constraints

After you have defined a node pool, you can constrain an Orchestrate operator or an entire Orchestrate step to execute on only the processing nodes in the node pool.

To set a node pool constraint on an operator:

1. Double click the operator in the **Program Editor** to open the **Operator Properties** dialog box.
2. Click the **Advanced** tab.
3. Click the **Add** button under **Constraints** to add a new constraint, using the following dialog box:



4. Select **Node Pool** from the **Constraint Type** pull-down list.
5. Select the Orchestrate configuration from the **Config Name** list, containing the names of Orchestrate configuration files (created by the Orchestrate server administrator).
6. Enter the node pool name in the **Pool Name** area. You can enter multiple node pool names, separated by commas. For example, to specify the node pool `compute_node`, you enter:

**Pool Name:** `compute_node`

In addition, you can constrain an entire step to a node pool. To constrain a step, double click the step in the **Program Editor** to open the **Step Properties** dialog box. In that dialog box, choose the **Constraints** tab to set a node pool constraint for all the operators in a step.

Multiple node pool constraints are ANDed together, so that a node must meet all constraints in order to process the operator. For example, the following command constrains an operator to all nodes in both the `compute_node` and `io_node` node pools:

**Pool Name:** `compute_node, io_node`

In this case, only `node1` satisfies both constraints; therefore, the operator executes only on `node1`.

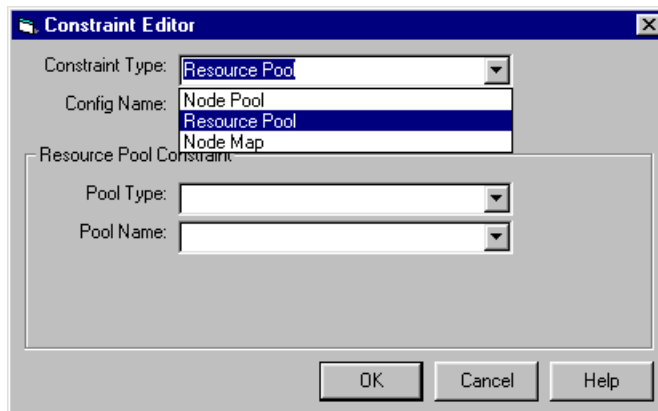
You can combine the `nodepool` and `resources` statements to combine constraints. See the section “Combining Node and Resource Constraints” on page 10-8 for more information.

## Using Resource Constraints

Each processing node in your system can have access to specific resources. Orchestrate allows you to impose constraints on operators based on node resources. As described in the section “Configuring Orchestrate Logical Nodes” on page 10-5, the `resource disk` and `resource scratchdisk` options allow you to specify pools for each type of disk storage.

To set a resource constraint on an operator:

1. Double click the operator in the **Program Editor** to open the **Operator Properties** dialog box.
2. Click the **Advanced** tab.
3. Click the **Add** button under **Constraints** to add a new constraint. This opens the following dialog box:



4. Select **Resource Pool** from the **Constraint Type** pull down list.
5. Select the Orchestrate configuration from the **Config Name** list. A configuration corresponds to a configuration file and is created by the Orchestrate server administrator.
6. Enter the resource pool type and name under **Resource Pool Constraint**. You can enter multiple pool names, separated by commas.

The following example constrains an operator to execute only on those nodes with a disk resource in the pool `pool1`:

**Pool Type:** `disk`

**Pool Name:** `pool1`

In addition, you can constrain an entire step to a resource pool. In order to constrain a step, double click a step in the **Program Editor** to open the **Step Properties** dialog box. Then choose the **Constraints** tab to set a resource pool constraint for all the operators in a step.

Orchestrate applies all resource constraints, so that a node must have a disk that satisfies both constraints in order to execute the operator. In this case, only `ionode0` has a disk in both `pool1` and `pool2` and is therefore the only node to execute the operator.

The following example constrains an operator to the nodes with a disk resource in `pool1` and `pool2`:

**Pool Type:** `disk`

**Pool Name:** `pool1, pool2`

## Combining Node and Resource Constraints

You can combine node and resource constraints. Node and resource constraints are ANDed together; a node must meet all constraints in order to execute the operator.

To combine constraints, you use the **Constraint Editor** for either a step or an operator to set both **Node Pool** and **Resource Pool** constraints. The following example sets both a node and a resource constraint:

**Node Pool:**

**Pool Name:** `compute_node`

**Resource Pool:**

**Pool Type:** `disk`

**Pool Name:** `pool1`

In this example, only `node1` is in the `compute_node` pool and has a disk in `pool1`.

## Using Node Maps

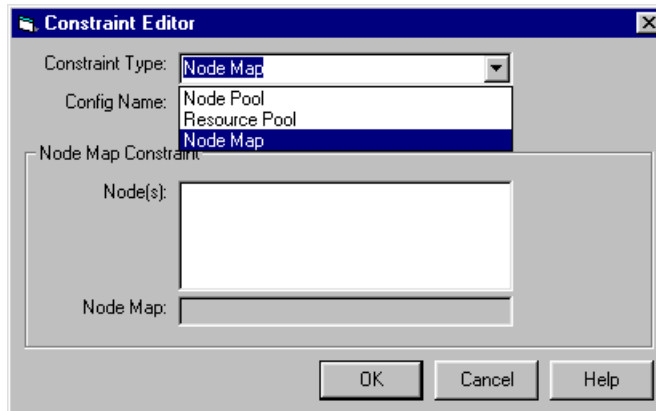
Node maps allow you to constrain a particular run of an operator to execute on a specific set of processing nodes. A node map applies to an operator invocation overrides any node pool or resource pool constraint applied to the operator or to its step. You cannot combine node maps with any other type of constraint.

To set a node map constraint on an operator:

1. Double click the operator in the **Program Editor** to open the **Operator Properties** dialog box.
2. Click the **Advanced** tab.



3. Click the **Add** button under **Constraints** to add a new constraint. This opens the following dialog box:



4. Select **Node Map** from the **Constraint Type** pull down list.
5. Select the Orchestrate configuration from the **Config Name** list. A configuration corresponds to a configuration file and is created by the Orchestrate server administrator.
6. Enter the node names, as defined by either the `node` or `fastname` parameter in the configuration file, in the **Node(s)** area. You can enter multiple node names, separated by commas.

For example, to specify that an operator executes only on `node1` and `node2`, you enter:

**Node(s):** `node1, node2`

Note that you can specify a node map only for an individual operator and not for an entire step.

## Data Set Constraints

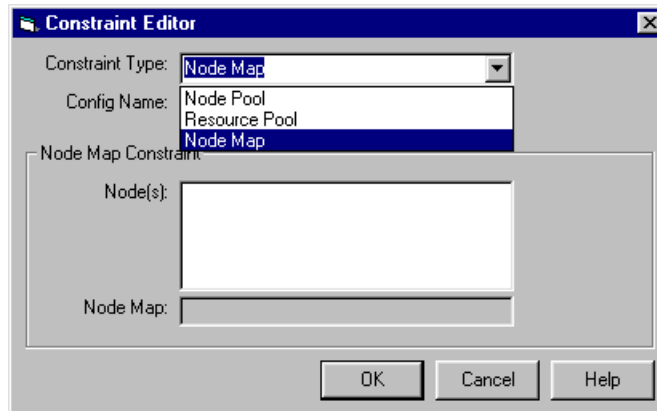
By default, Orchestrate writes a persistent data set to all disks assigned to the default disk pool. However, Orchestrate lets you to use disk pool constraints to specify the disk drives that store persistent and temporary data sets on your system. The data set constraints are based on the disk pool assignments in your Orchestrate configuration file, described in the section “Configuring Orchestrate Logical Nodes” on page 10-5.

In addition, you can use node pools and node maps to control the processing nodes used to store a data set. See the section “Using Node Pool Constraints” on page 10-6 or the section “Using Node Maps” on page 10-8 for more information on using these constraint types.

To set a resource constraint on a data set:

1. Double click the link for the data set in the **Program Editor** to open the **Link Properties** dialog box.
2. Click the **Constraints** tab.

3. Click the **Add** button under **Constraints** to add a new constraint. This opens the following dialog box:



4. Select **Node Pool**, **Resource Pool**, or **Node Map** from the **Constraint Type** pull down list.
5. Select the Orchestrate configuration from the **Config Name** list. A configuration corresponds to a configuration file and is created by the Orchestrate server administrator.
6. Enter the constraints.

For example, to specify that an output persistent data set is written only to the disks in the pool `pool1`, you set the following constraint:

**Resource Pool:**

**Pool Type:** `disk`

**Pool Name:** `pool1`

The number of partitions of the data set equals the number of nodes that have disks in the specified pool. All processing nodes executing the writing operator must contain at least one disk in the specified disk pool. See the *Orchestrate Installation and Administration Manual* for more information on disk pools.

# 11: Run-Time Error and Warning Messages

During execution of your application, Orchestrate detects and reports error and warning conditions. This chapter describes the format of Orchestrate warning and error messages and describes how to control the format of message display, in the following sections:

- “How Orchestrate Detects and Reports Errors” on page 11-1
- “Error and Warning Message Format” on page 11-2
- “Controlling the Format of Message Display” on page 11-4

## How Orchestrate Detects and Reports Errors

During execution of your application, Orchestrate detects error and warning conditions, which can be generated by the following:

- Orchestrate operators, used in your application steps. For details on Orchestrate operator error messages, see the *Orchestrate User's Guide: Operators*.
- Your application code, outside the Orchestrate steps. For information on using the Orchestrate error-handling class, see the *Orchestrate/APT Developer's Guide*.
- Subprocesses in your application, including wrappers and third-party applications.

When Orchestrate detects an error or warning condition, it writes the applicable information and message to the error log.

If the condition is not severe, after writing a warning message to the error log, Orchestrate allows the application to continue execution. At various points in running an application or utility, Orchestrate checks the error log for new entries. It writes new warning messages to the message window on the screen of the workstation from which you invoked the application. Right-clicking in the message window pops up a menu of standard commands that you can use to edit and copy the messages.

If the condition is so severe that application execution cannot continue, after writing an error message, Orchestrate terminates the application.

## Error and Warning Message Format

The table below lists the components of Orchestrate error and warning messages, as follows:

- The first column shows whether the default is on or off for display of the component.
- The second column is the keyword (case-insensitive) for changing the default display (see the section “Controlling the Format of Message Display” on page 11-4).
- The third column is the component length, which for some components is fixed and for others is variable. Orchestrate left-pads with zeros all fixed-length components. For any variable-length component, you can configure Orchestrate to display the component length.
- The last column describes the component.

Note that except where indicated, there is a one-space separator between message components. Every message is terminated with a newline. Only the last message component, the message text, may contain spaces.

Default Display	Keyword	Length	Description
On		2	The string "##". You cannot suppress display of this component.
		0	[No separator]
On	severity	1	Severity of condition: "F", "E", "W", or "I", for Fatal, Error, Warning, or Informational message.
Off	vseverity	7	Verbose severity indicator: "Fatal", "Error", "Warning", or "Inform".
Off	jobid	3	Job identifier of the Orchestrate application, to let you identify concurrently running Orchestrate applications. The default job ID is 0.
On	moduleId	4	Module identifier, which is one of the following: For Orchestrate-defined error messages, a four-character string beginning with "T". For user-defined error messages, the string "USER". For a message from a subprocess, the string "USBP".
		0	[No separator]
On	errorIndex	6	Index of the message at the time it was written.
On	timestamp	13	Message time stamp, consisting of the string "HH:MM:SS(msg_seq)", which is the hour, minute, second, and message sequence number at the time the message was written.  Note that error messages written within one second have ordered sequence numbers.

Default Display	Keyword	Length	Description
Off	<code>ipaddr</code>	15	IP address of the node generating the message. This 15-character string is in octet form, with octets zero-filled; e.g., 104.032.007.100.
Off	<code>lengthprefix</code>	2	Length in bytes of the following field, <code>nodeplayer</code> .
Off	<code>nodeplayer</code>	Variable	String " <i>(node,player)</i> ", containing the number of the section leader and player that generated the message.
Off	<code>lengthprefix</code>	2	Length in bytes of the following field, <code>nodename</code> .
Off	<code>nodename</code>	Variable	Name of the node generating the message.
Off	<code>lengthprefix</code>	2	Length in bytes of the following field, <code>opid</code> .
On	<code>opid</code>	Variable	Operator identifier, which is one of the following: <p>For messages originating in the main program (not in a step), the string "<code>&lt;main_program&gt;</code>".</p> <p>For system messages originating on a node, the string "<code>&lt;node_nodename&gt;</code>", where <i>node-name</i> is the name of the node.</p> <p>For messages originating in a step, the operator originator identifier, which identifies the instance of the operator that generated the message. This identifier is the string "<i>ident,partition_n</i>". <i>ident</i> is the operator name. If there is more than one instance of the operator, <i>ident</i> includes an operator index in parentheses. <i>partition_n</i> identifies the partition of the operator issuing the message. An example of an <code>opid</code> originating in a <code>osh</code> step is <code>&lt;myop,4&gt;</code></p>
Off	<code>lengthprefix</code>	5	Length in bytes of the following field, <code>message</code> .
On	<code>message</code>	Variable	Text of the message. Maximum message length is 15 KBytes.
		1	Newline

## Messages from Subprocesses

The message display configuration also controls display of error messages from subprocesses run by Orchestrate, including wrappers and third-party applications. Orchestrate catches subprocess messages written to the subprocess's standard output or standard error. Orchestrate displays the messages using the current message display configuration, on a per-line basis. The module

identifier for all subprocess output is "USBP". Orchestrate gives messages written to standard output Informational severity and a message index of 1. Orchestrate gives messages written to standard error Warning severity and a message index of 2.

## Controlling the Format of Message Display

The following is an example of a warning message with all its components displayed:

```
##I Inform 000 TOSH000010 10:46:15(001) 010.000.002.119 05 (0,0) 09
localhost 14 <main_program> 00016 orchsort: loaded
```

You can limit the message components that Orchestrate displays. Suppose, for example, that you limit the display of the sample warning message above to message severity, module and index, processing node, operator identifier, and message text. Orchestrate would then display the message as follows:

```
##I TOSH000010 localhost <main_program> orchsort: loaded
```

You use keywords to control message display. Specifying an unmodified keyword configures Orchestrate to display the associated message component. Preceding the keyword with an exclamation point (!) configures Orchestrate to suppress display of the associated component.

For example, shown below is the default message configuration for all messages originating from Orchestrate applications:

```
severity, !vseverity, !jobid, moduleid, errorIndex, timestamp,
!ipaddr, !nodeplayer, !nodename, opid, message, !lengthprefix
```

This example specifies suppression of the display of verbose severity, job identifier, IP address, node name, and length prefixes are all suppressed.

The display of messages from Orchestrate command utilities (such as `buildop` and `cbuildop`) has the following default:

```
severity, !vseverity, !jobid, moduleid, errorIndex, !timestamp,
!ipaddr, !nodeplayer, !nodename, !opid, message, !lengthprefix
```

For messages from command utilities, Orchestrate suppresses the display of verbose severity, job identifier, time, IP address, node player, node name, operator identifier, and length prefixes.

To control message display format, you use the `APT_ERROR_CONFIGURATION` environment variable.

For further details on `APT_ERROR_CONFIGURATION` and other environment variables, see the *Orchestrate Installation and Administration Manual*.

The environment variable `APT_ERROR_CONFIGURATION` lets you configure error and warning message display for all Orchestrate applications and utilities. To use the variable

APT\_ERROR\_CONFIGURATION, you issue a UNIX command to set it to a string containing the component keywords defined in the section “Error and Warning Message Format” on page 11-2. Any keywords omitted from your command to set APT\_ERROR\_CONFIGURATION remain unchanged from their previous state.

For example, the following commands set and export APT\_ERROR\_CONFIGURATION, in the syntax for the Korn and Bourne shells:

```
APT_ERROR_CONFIGURATION='! severity, ! timestamp, ipaddr, nodename'  
  
export APT_ERROR_CONFIGURATION
```

Following is the equivalent command to set and export APT\_ERROR\_CONFIGURATION, in the syntax for the C shell:

```
setenv APT_ERROR_CONFIGURATION "\! severity, \! timestamp, ipaddr,  
nodename'
```

In the C shell command, you must precede an exclamation point (!) with the escape character, backslash (\).

In both versions of the command to set APT\_ERROR\_CONFIGURATION, note the space between the exclamation point and the keyword. It is recommended that you insert this space, to make sure that the command shell does not interpret the exclamation point and keyword as a reference to the command history buffer.





# 12: Creating Custom Operators

Many Orchestrate applications require specialized operators to perform application-specific data processing, in parallel. You may need an operator to perform a simple operation such as adding two particular fields. Or, you may need an operator to carry out a much more complex task.

Visual Orchestrate gives you the ability to create two kinds of custom operators: native operators and UNIX command operators. This chapter describes how to create native operators, for which you supply a few C or C++ statements to perform the operator's action. (Creating UNIX command operators is described in the chapter "Creating UNIX Operators".)

This chapter includes the following sections:

- "Custom Orchestrate Operators" on page 12-1
- "Using Visual Orchestrate to Create an Operator" on page 12-5
- "Specifying Operator Input and Output Interfaces" on page 12-8
- "Examples of Custom Operators" on page 12-14
- "Using Orchestrate Data Types in Your Operator" on page 12-20
- "Using the Custom Operator Macros" on page 12-27
- "How Visual Orchestrate Executes Generated Code" on page 12-31
- "Designing Operators with Multiple Inputs" on page 12-31

## Custom Orchestrate Operators

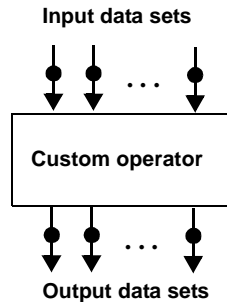
Visual Orchestrate lets you create custom (native) operators by supplying only a few lines of C or C++ code for the operator body, and some configuration information. From this information, Orchestrate creates the operator by automatically generating C/C++ code, and then compiling and linking the operator.

This section describes the characteristics of the custom operators that you can create. It then describes how custom operators perform input and output and process data.

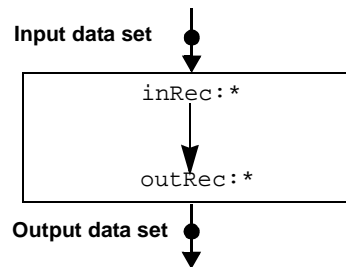
## Kinds of Operators You Can Create

Orchestrate allows you to create custom operators with the following characteristics:

- The operator has at least one input data set and one output data set. The operator can have multiple input data sets and multiple output data sets. The following data flow diagram shows a custom operator with multiple inputs and outputs:



- By default, the operator's execution mode is parallel. However, you can override this mode to specify sequential execution when you use the operator in a step.
- The default partitioning method of the operator is *any* (parallel mode), and the default collection method is *any* (sequential mode). You can use a partitioner or collector operator to modify these defaults.
- You can optionally define a transfer to copy an entire input record to an output data set, as shown in the data-flow diagram below:



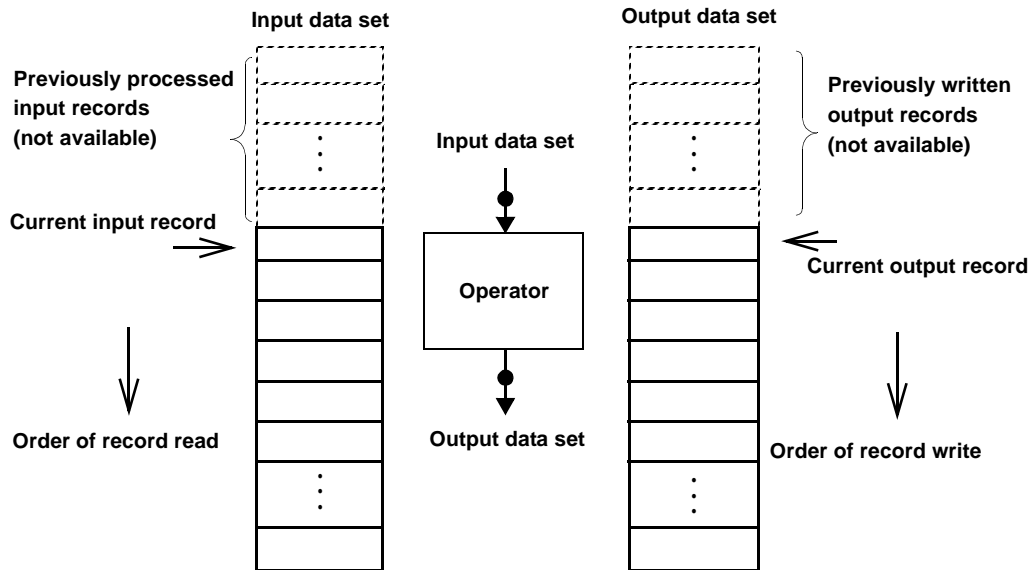
Note that the transferred record is represented in the operator input interface schema as `inRec: *`, and in the output interface schema as `outRec: *`. For a complete description of the Orchestrate transfer feature, see the section "Record Transfers and Schema Variables" on page 5-11.

- You can define user-settable options, or arguments, passed to an operator. Then, when users insert the operator into a step, they can set those control options by using the **Options Editor** dialog box.

## How a Generated Operator Processes Data

To plan and create an operator, you need to understand how Orchestrate operators process one or more input data sets and create one or more output data sets. This section describes this operation.

The following figure shows an operator with a single input and a single output data set:



An operator executes an I/O and processing loop, with iteration (time through the loop) for every input record. After all the required number of input records have been read from one or more inputs, the operator terminates.

### How the Operator Executes the Loop

For each input record, the operator executes the following steps in the I/O and processing loop:

1. Reads a record from the input data set (unless there are no remaining records or automatic read has been disabled).
2. Performs the action of the operator.
3. As needed, assigns results to the current output record.
4. If specified, performs a transfer of the entire input record to the output record (unless there are no more records to transfer or automatic transfer has been disabled).
5. Writes the current output record to the output data set (unless automatic write has been disabled).

Note that your operator does not have to produce one output record for each input record. Some operators read multiple input records before producing a single output record, such as an operator that removes duplicate records from the input data set. Other types of operators can create multiple output records from a single input record.

## How the Operator Processes the Input Data Sets

An operator reads each input data set record by record, to process its data. The operator continues to read and process records until it has read all records in the input data set. Once the operator has moved beyond a record of an input data set, it can no longer access that record.

Note that the fields of all input data sets are read-only, while the fields of all output data sets are readable and writable.

## How the Operator Writes to the Output Data Set

In creating your operator, you define the operator's input interface schema and output interface schema. See the section "Specifying Operator Input and Output Interfaces" on page 12-8 for more information.

After the operator assigns the processing results to the record fields, it writes the record to the output data set. Writing the output record commits the record to the data set, and it creates a new output record with default values for all fields. As with input data sets, once processing has advanced beyond a record of an output data set, the operator can no longer access the record.

The default value for an output record field is based on the field's data type and nullability setting. For a complete list of default values for fields of all types, see the description. For a list of default values for fields of all types, see the section the section "Default Values for Fields in Output Data Sets" on page 4-27.

## Configuring Orchestrate For Creating Operators

To allow creation of operators, the Orchestrate server must be able to access the necessary UNIX C++ compiler and other utilities. The appendix on supported operating systems, software packages, and databases in the *Orchestrate Installation and Administration Manual* lists these compilers and utilities. Your Orchestrate server administrator must ensure that you have the correct access privileges and path settings to run these utilities.

You can specify a non-default compiler for your custom operators, by using the Paths tab of the Program Properties dialog box. See the section "Setting Program Directory Paths" on page 2-13.

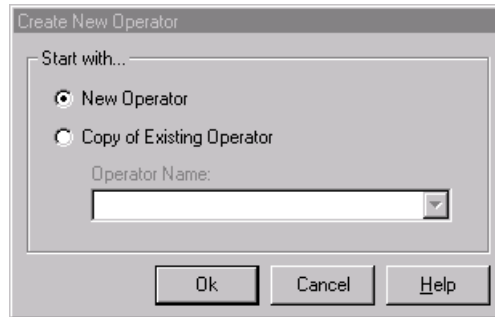
## Included Header Files

The custom operator utility lets you to use standard I/O, I/O stream, and math functions automatically. In order to do so, the utility includes the following header files:

- `stdio.h`
- `iostream.h`
- `math.h`

## Using Visual Orchestrate to Create an Operator

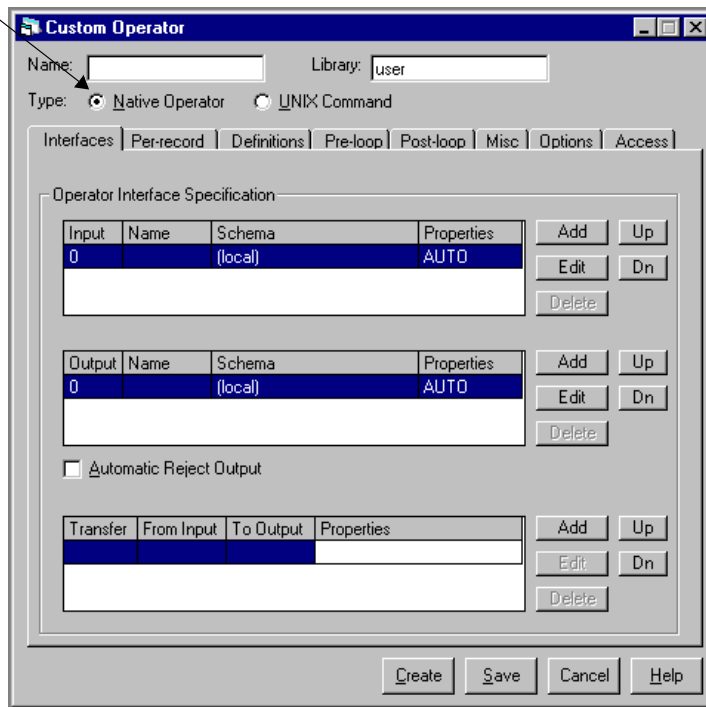
To create a custom operator, choose the **Custom -> Define Custom Operator** menu command. If there are no existing custom operators in your system, the **New Operator** dialog box (shown on the next page) opens. If there are existing custom operators, the following dialog box opens:



The **Create New Operator** dialog box allows you to select an existing custom operator to copy as a starting point for your new operator, or to create a completely new custom operator. If you choose **Copy of Existing Operator**, you then select an operator from the **Operator Name** list. When you click **OK**, the **Customer Operator** dialog box appears with the definition of the existing operator.

If in the **Create New Operator** dialog box you select **New Operator**, clicking **OK** opens an empty **Custom Operator** dialog box, as shown below

Native Operator selected



By default, the **Native Operator** radio button is selected. The **Library** text box defaults to your user name. You can type in a new library name or the name of any existing library.

In the **Name** text box (upper left of dialog box), type the name of this operator. A name can be up to 250 characters long. An operator name must be unique within its library, but it can be the same as the name of an operator in another library.

You define the operator by using the **Custom Operator** dialog box tabs, as follows:

**Interfaces** (shown above): Specify the names and schemas of the operator's inputs and outputs for the operator input and output interfaces.

Specifying your operator interface schemas is described in the section "Specifying the Interface Schema" on page 12-10. Also specify and configure record transfers, as described in the section "Defining Transfers" on page 12-13.

**Per-record:** Specify the code loop, which is executed once for every input record.

**Definitions:** Optionally, specify definitions to include before the operator's executable code. This option allows you to specify header files or other information needed to compile your code.

By default, Orchestrate always includes the header files `stdio.h`, `iostream.h`, and `math.h`.

**Pre-loop:** Optionally, specify code to execute before the operator executes the **Per-record** code to process the record. You can use this code section to initialize variables used in the **Per-record** section.

**Post-loop:** Optionally, specify code to execute after the operator has completed the **Per-record** section and before the operator terminates.

**Misc:** Optionally, specify the following options for operator execution:

**Operator Execution Mode:** You can select **Parallel**, **Sequential**, or **Operator Default** (default for this option). Specifying **Operator Default** allows the operator user to set the execution mode upon inserting the operator into a step.

**Compiler Options:** You can specify options passed to the C++ compiler (and linker) used to create the operator:

**Verbose:** Echoes commands and compiler messages to the message window.

**Debug:** Runs the compiler in debug mode.

**Optimize:** Optimizes the speed and efficiency of the compiled code.

**Additional Compiler Flags:** You can specify additional compiler command options.

**Additional Linker Flags:** You can specify additional linker command options.

**Operator Description:** Optionally, type your notes about this operator, to be saved with the operator definition. Right-clicking the text box opens a menu of standard text-editing commands, such as cut and paste.

**Options:** Optionally, define user-settable controls for the operator, with which the operator user can control the action of the operator.

**Access:** Set the access privileges of the operator, by selecting one of the following:

**Public Write** (default): Anyone can read, write, or execute the operator.

**Public Read:** Anyone can read and execute the operator; only the operator creator can modify it.

**Private:** Only the operator creator can read, write, or execute the operator.

At any time after you have specified a name and library for the operator, you select a dialog box buttons to do one of the following:

- **Save** the information about the operator, but do not create it. A saved operator appears under the specified library in the Server View area of Visual Orchestrate in parentheses, to indicate that the operator user cannot use it. (You must press **Create** to make it usable in an application.)
- **Create** the operator, so that an operator user can insert it into an Orchestrate application. The created operator appears under the specified library in the Server View area.
- **Cancel** operator definition.
- Open online **Help**.

The **Per-record**, **Pre-loop**, **Post-loop**, and **Definitions** tabs each have an **Import** button. That option lets you specify an ASCII text file on the PC, containing your code for the section, which Orchestrate will read into that section of your operator definition.

## How Your Code Is Executed

The following steps show how an operator executes to process its input data set and write to its output data set:

1. If specified, the operator executes the code in the **Pre-loop** section.
2. For each record in the input data set, performs the steps of the I/O and processing loop (introduced in the section “How the Operator Executes the Loop” on page 12-3):
  - a. Reads a record from the input data set (unless there are no records to input or automatic read has been disabled).
  - b. Executes the processing code specified in the **Per-record** section.
  - c. As needed, assigns results to the current output record.
  - d. As specified, performs a transfer of the entire input record to the output record (unless there are no more records to transfer or automatic transfer has been disabled).
  - e. Writes the current output record to the output data set (unless automatic write has been disabled). After the write, the operator creates a new, empty output record.
3. If specified, the operator executes the code in the **Post-loop** section.

## Specifying Operator Input and Output Interfaces

This section describes how to create an input and output interface for your operator, in the following sections:

- “Adding and Editing Definitions of Input and Output Ports” on page 12-8
- “Reordering the Input Ports or Output Ports” on page 12-10
- “Deleting an Input or Output Port” on page 12-10
- “Specifying the Interface Schema” on page 12-10
- “Defining Transfers” on page 12-13
- “Referencing Operator Interface Fields in Operator Code” on page 12-13

## Adding and Editing Definitions of Input and Output Ports

To view and modify the input and output interfaces, you use the **Interfaces** tab in the **Custom Operator** dialog box, shown in the section “Using Visual Orchestrate to Create an Operator” on



page 12-5. From the **Custom** menu, select **Define Custom Operator** or **Edit Custom Operator**, to open the **Custom Operator** dialog box. In the **Interfaces** tab, the top list shows the ports defined for the input interface, and the middle list shows the port defined for the output interface. The input and output interface ports are always indexed  $0 \dots numPorts-1$ . When you create the operator, Orchestrate automatically creates input port 0 for the input interface and output port 0 for the output interface.

To add an input port click the **Add** button to the right of the input interface list. The **Input Interface Editor** dialog box opens, with an **Input Name** text field and the input interface schema name. You can type a port name (see below for naming guidelines), or you can leave **Input Name** blank to accept the default name. If you want to disable automatic read, check the **Disable automatic record read** box. Click **OK** to create the port. The new port then appears in the input interface list.

To create an output port, click the **Add** button next to the output interface list to open the **Output Interface Editor**. Optionally type a name into the **Output Name** text field. If you want to disable automatic write, check the **Disable automatic record write** box. Click **OK** to add the port, which then appears in the output interface list.

---

**Note:** To save the changes that you make to the input and output interfaces, you *must* click **Update**, **Save**, or **Create**, in order to rebuild the operator with your changes. If you close the **Custom Operator** dialog box without updating the operator, any changes you have made to the operator are discarded.

---

Also note that the checkbox **Automatic reject output** on the **Interfaces** tab is to support applications created with earlier releases of Orchestrate. For a description of creating an operator with a reject data set, see the section “Example Operator: reject” on page 12-30.

## Naming Input and Output Ports

By default, Orchestrate names the input ports of your operator input interface `in0 \dots innum_ins-1`, and it names the output ports of the operator output interface `out0 \dots outnum_outs-1`. You can use these default names to refer to input and output ports in your operator code. You also have the option of specifying non-default names for any input or output port in your operator's interfaces. The only restriction is that you cannot specify a name of the form `inn` or `outn`, which is reserved for the default naming.

## Editing Port Definitions

You can create or modify a non-default name for an input or output port at any time, by selecting **Edit** from the **Interfaces** tab to open the **Input Interface Editor** or **Output Interface Editor** dialog box. You can type in a new name or modify an existing one. To accept the new name, click **OK**. To save the change to the operator interface, click **Create** or **Update** in the **Interfaces** tab.

## Reordering the Input Ports or Output Ports

To reorder the list of inputs or outputs, use the **Up** and **Dn** buttons for the list. Clicking **Up** exchanges the position and index of the selected port with the position and index of the port above it (at a lower index number) in the list. Clicking **Dn** makes the exchange between the selected port and the one below it.

## Deleting an Input or Output Port

To delete an input port, select the port and click the **Delete** button for the input interface. To delete an output port, select the port and click the **Delete** button for the output interface. When you delete an input or output port, Orchestrate immediately removes the port from the interface list and decrements the index of any input or output port that was below it in the interface list. To save the change to the operator interface, click **Update** in the **Interfaces** tab.

As an operator must have at least one input and one output, the **Delete** button is not active for an input or output interface with only one port.

## Port Deletions and Transfer Definitions

As described in the section “Defining Transfers” on page 12-13, transfer definitions identify the input and output by index only, so that changing a port's name or schema does not change the transfer definition, even though the port change may affect the result of performing the transfer. Likewise, If your operator defines a transfer that uses a port affected by a deletion, the change does not affect the transfer definition as long as there is still an input or output port defined with the index used in the transfer definition.

However, if your port deletion results in the loss of a port index that is used in a transfer, you cannot save the interface change. For example, suppose your operator input interface has two ports and defines a transfer from the input port at index 1. If you deleted an input port, leaving only input port 0, clicking Update to rebuild the operator would produce an error message regarding the undefined input port 1 in the transfer.

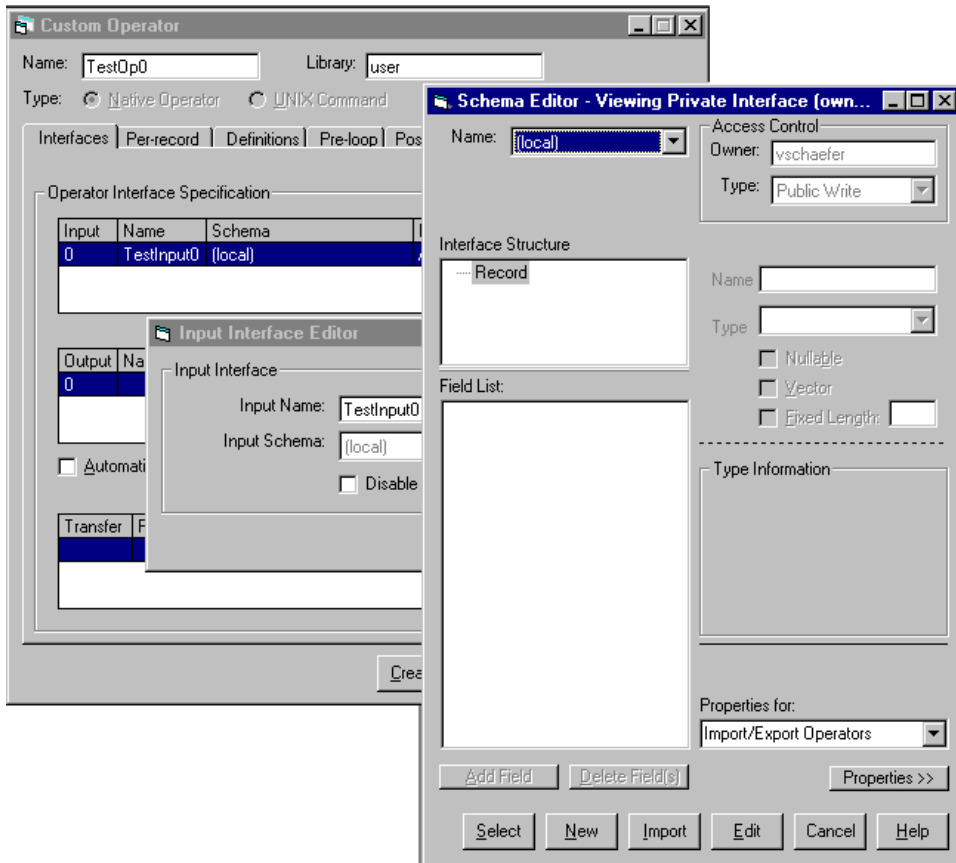
## Specifying the Interface Schema

To create and edit the input and output interface schemas, you use the Schema Editor. For a detailed description of the Schema Editor, see the chapter “Orchestrate Data Sets”.

## Creating or Editing an Interface Schema

To create an input or output interface schema, open the **Custom Operator** dialog box, as described in the section above. Click **Add** to add a new port, or select an existing port and click **Edit**. In the

**Input Interface Editor** or **Output Interface Editor** dialog box, click the **Details** button to open the Schema Editor, shown below:



In the Schema Editor, you can click **New** to create a new schema. The Schema Editor window displays the **Name** and **Library** text fields, with radio buttons **Named** and **Unnamed**. Naming an interface schema enables you to use the schema for more than one input or output port. If you select the **Named** radio button, you then enter a schema name in the **Name** field, and you can optionally modify the schema library name (the default is your user name). The schema name be unique within its library, but it can be the same as a name in another schema library.

In creating a new schema, you add and edit fields in the schema. You can optionally modify the owner and access control for the schema. To save the new schema, click **Save** to save the new schema. When you save a named schema, its name appears in the selected port's schema column in the **Interfaces** tab; however, to save the change to the port definition, you must click **Update** on the **Interfaces** tab.

You can create an unnamed schema, by selecting the **Unnamed** radio button in the Schema Editor window. When you save an unnamed schema, it appears as (local) in the definition for the selected port, and it is not available to be used in the definition of other ports.

To edit an existing schema, click **Edit** in the Schema Editor window. You can access properties, and fields of the schema. If it is a named schema, you can also edit the schema name and library name. To save the changes to the schema, click **Save** in the Schema Editor window.

### Selecting an Existing Named Schema for a Port

After you have saved a named schema, you can select it for an input or output port other than the one for which you created it. When you add or edit an input port, you can define it with any named input schema defined for the interface, and likewise, you can define an output port with any named output interface schema. To change the schema for a port, select the schema name from the list in the **Input Interface Editor** or **Output Interface Editor** dialog box, and click **OK**. To save the change to the operator interface, be sure to click **Update** in the **Interfaces** tab.

### Special Restrictions on Interface Schema Field Definitions

Note that every input and output interface schema must be a legal record schema. In addition, the following elements are *not* allowed in operator interface schemas:

- Subrecord or tagged aggregate elements
- Import/export or generator properties
- Schema variables

If you attempt to save or create an operator interface schema that contains any of these elements, the action will fail with an error message.

### Naming Fields in the Interface Schemas

You cannot name a record schema field with any of the C++ reserved words, listed below:

asm	class	double	friend	new	return	switch	union
auto	const	else	goto	operator	short	template	unsigned
break	continue	enum	if	private	signed	this	virtual
case	default	extern	inline	protected	sizeof	throw	void
catch	delete	float	int	public	static	try	volatile
char	do	for	long	register	struct	typedef	while

In addition, you cannot use either of the following as field names:

- `_r_`
- `__record__` (two trailing underscores)

If you create an operator used in an application that accesses an RDBMS (such as DB2, INFORMIX, Teradata, or Oracle), do not create a field with the same name as an SQL reserved word. See the SQL manual for your RDBMS for a list of these reserved words.

## Defining Transfers

You can define one or more transfers for your operator. As described in the section “How a Generated Operator Processes Data” on page 12-3, Orchestra performs a transfer as part of executing the code loop of the operator.

To define, modify, or delete a record transfer, open the **Custom Operator** dialog box, by clicking **Define Custom Operator** or **Edit Custom Operator** in the **Custom** menu. On the **Interfaces** tab, the bottom list shows all transfers defined for your operator. The transfer column is the index of the transfer ( $0-n-1$ ), From Index is the index of the input port, To Index is the index of the output port, and Properties is the automatic transfer setting (AUTO or NOAUTO) followed by the combine-transfer setting (COMBINE or SEPARATE). When you create an operator, by default there are no transfers defined, and transfer definition is optional.

To define a transfer, click the **Add** button to the right of the transfer list to open the **Transfer Editor** dialog box. From the From Input list, select the index of the input port for the transfer, and from the To Output list, select the output port index. If you want to disable automatic transfer, check the **Disable Automatic Transfer** box. If you want to prevent Orchestra from combining this transfer with others to the same output, check the **Transfer Separately** box. Click **OK** to create the transfer. The transfer then appears in the transfer list on the Interfaces tab.

To reorder the list of transfers, use the **Up** and **Dn** buttons for the list. Clicking **Up** exchanges the position and index of the selected transfer with the position and index of the transfer above it (at a lower index number) in the list. Clicking **Dn** makes the exchange between the selected transfer and the one below it.

To delete a transfer, select it in the transfer list and click **Delete**. Orchestra decrements the index of any transfers below it in the list. For example, if there are two transfers defined and you delete the first transfer in the list (index 0), the index of the second transfer is changed from 1 to 0.

## Referencing Operator Interface Fields in Operator Code

In your **Pre-loop**, **Post-loop**, and **Per-record** code, you refer to schema components with the field name, as defined in the input or output interface schema. If the field name is not unique among all interfaces, both input and output, for the operator, you must prefix the field name with the default or explicitly defined port name followed by a dot (.).

If the field name is not unique among all the operator's interface schema, then you must reference it with *portname.field\_name*, where *portname* is either the name you have given the port (see the section “Naming Input and Output Ports” on page 12-9) or, for a port that you have not named, the default name. For example, for a field price in your input schema for the third input port (index 2), which you have not explicitly named, you use the reference `in2.price`.

## Processing Fields According to Data Type and Properties

To let you process fields according to Orchestrate data type and properties, including nullable fields and vector fields, Orchestrate supplies functions that you can use in your code. These functions are described in the section “Using Orchestrate Data Types in Your Operator” on page 12-20.

## Examples of Custom Operators

This section describes how to define the following sample operators:

- “Example: Sum Operator” on page 12-15
- “Example: Sum Operator Using a Transfer” on page 12-16
- “Example: Operator That Recodes a Field” on page 12-17
- “Example: Adding a User-Settable Option to the Recoding Operator” on page 12-17

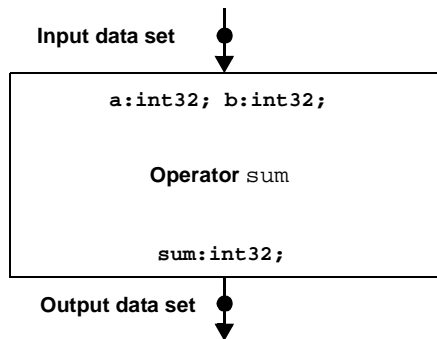
## Convention for Property Settings in Examples

In all custom operator examples, the property settings for inputs, outputs, and transfers are abbreviated, as follows:

- **Inputs:** `Auto` stands for *automatic input* (**Disable automatic read** is not checked), and `Noauto` stands for *no automatic input* (**Disable automatic read** is checked).
- **Outputs:** `Auto` stands for *automatic output* (**Disable automatic write** is not checked), and `Noauto` stands for *no automatic output* (**Disable automatic write** is checked).
- **Transfers:** `Auto` stands for *automatic transfer* (**Disable automatic transfer** is not checked), and `Noauto` stands for *no automatic transfer* (**Disable automatic transfer** is checked). `Combine` stands for *combine transfer* (**Transfer separately (don't combine with other transfers to the same output)** is not checked), and `Separate` stands for *separate transfer* (**Transfer separately...** is checked).

## Example: Sum Operator

This section describes how to define a sample operator, `sum`, shown in the following diagram:



This operator sums two fields of an input record to create a new field in the output record. Each record of the output data set contains a single field containing the sum.

Shown below is the definition for this operator.

**Name:** `Sum`

**Input 0 Name:** `SumIn0`

**Input 0 Properties:** `Auto`

**Input 0 Interface Schema:** `record (a:int32; b:int32;)`

**Output 0 Name:** `SumOut0`

**Output 0 Properties:** `Auto`

**Output 0 Interface Schema:** `record (sum:int32;)`

**Per-record:** `sum = a + b;`

You can modify this operator to also copy the input fields to the output record. Shown below is a version of the operator to copy fields `a` and `b` to the output record:

**Name:** `Sum`

**Input 0 Name:** `SumIn0`

**Input 0 Properties:** `Auto`

**Input 0 Interface Schema:** `record (a:int32; b:int32;)`

**Output 0 Name:** `SumOut0`

**Output 0 Properties:** `Auto`

**Output 0 Interface Schema:** `record (a:int32; b:int32; sum:int32;)`

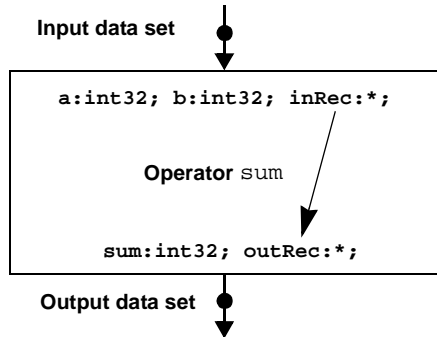
**Per-record:**

```
sum = SumIn0.a + SumIn0.b;
SumOut0.a = SumIn0.a;
SumOut0.b = SumIn0.b;
```

Note that field name references include the input and output port names. See the section “Referencing Operator Interface Fields in Operator Code” on page 12-13 for more information.

## Example: Sum Operator Using a Transfer

This section contains a modified version of the `sum` operator described in the section “Specifying Operator Input and Output Interfaces” on page 12-8. This version of the `sum` operator performs a transfer from input to output, as shown below:



For a description of defining a transfer, see the section “Defining Transfers” on page 12-13.

Shown below is the definition for this operator.

**Name:** `Sum`

**Input 0 Name:** `SumIn0`

**Input 0 Properties:** `Auto`

**Input 0 Interface Schema:** `record (a:int32; b:int32;)`

**Output 0 Name:** `SumOut0`

**Output 0 Properties:** `Auto`

**Output 0 Interface Schema:** `record (sum:int32;)`

**Transfer 0:** `Input From: 0; Output To: 0`

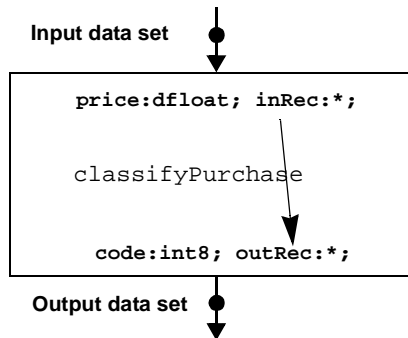
**Transfer 0 Properties:** `Auto, Combine`

**Per-record:** `sum = SumIn0.a + SumIn0.b;`



## Example: Operator That Recodes a Field

This section describes how to define another sample operator, `classifyPurchase`, shown in the following diagram:



In this example, operator `ClassifyPurchase` reads a record and transfers the entire input record to the output data set. The operator also defines the field, `code`, for the output data set. The operator assigns `code` the value 1 if the input field `price` is greater than or equal to 100, and 0 if `price` is below 100.

Shown below is the definition for this operator:

**Name:** `ClassifyPurchase`

**Input 0 Name:** `PurchIn0`

**Input 0 Properties:** `Auto`

**Input 0 Interface Schema:** `record (price:dfloat;)`

**Output 0 Name:** `PurchOut0`

**Output 0 Properties:** `Auto`

**Output 0 Interface Schema:** `record (code:int8;)`

**Transfer 0:** `Input From: 0; Output To: 0`

**Transfer 0 Properties:** `Auto, Combine`

**Per-record:**

```

if (PurchIn0.price >= 100) PurchOut0.code = 1;
else PurchOut0.code = 0;
  
```

## Example: Adding a User-Settable Option to the Recoding Operator

In the example above, operator `ClassifyPurchase` set the `code` field according to a set condition: whether the `price` field has the cutoff value of 100 or greater. This example alters the `ClassifyPurchase` operator, to define a parameter `cutoff` that operator users set. The operator then uses that cutoff to determine whether to set `code` to 1 or 0.

The following is the Per-record code for this modified `ClassifyPurchase` operator, in which `cutoff` is a floating-point variable holding a value set by the operator user:

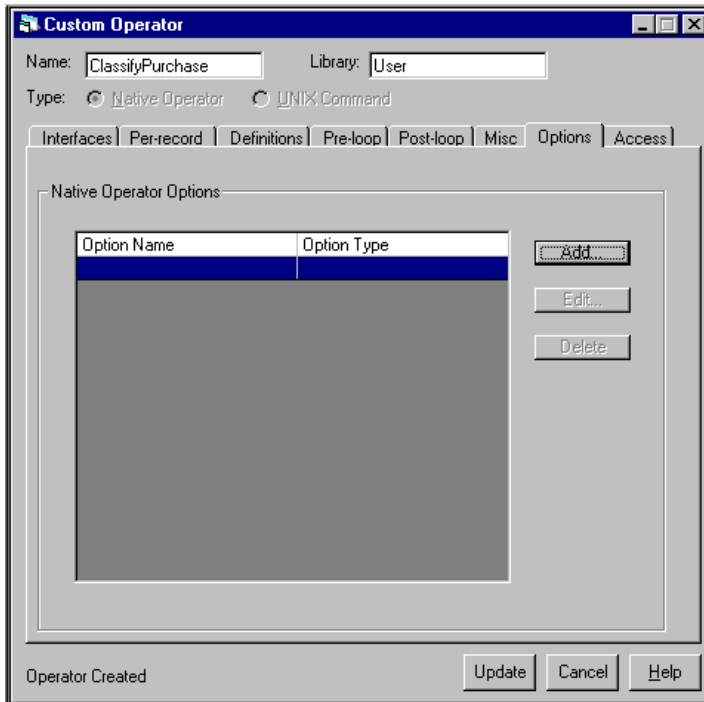
**Per-record:**

```
if (price >= cutoff) code = 1;
else code = 0;
```

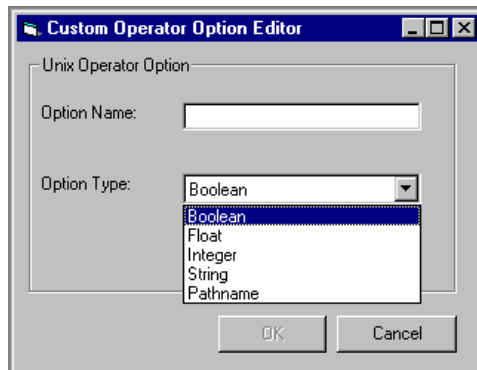
## Defining the User-Settable Option

To define the user-settable option `cutoff`, you perform the following steps:

1. Open the **Custom Operator** dialog box, and select the **Options** tab:



2. Press the **Add** button to open the **Custom Operator Option Editor** dialog box:



3. Specify the **Option Name**, which the operator user will select in the **Option Editor** dialog box. For this example, you enter the name `cutoff`.
4. Select the **Option Type** from the pull-down list. As shown in the figure above, available data types are `boolean`, `float`, `integer`, `string`, and `pathname`. (An option of type `pathname` takes a directory path, the validity of which must be determined by the operator.)  
For this example, select type `float`.
5. Set the **Option Name** to `cutoff`.
6. Set the **Option Type** to `float`.
7. Click **OK** to close the **Custom Operator Option Editor**.
8. Click **Update** to rebuild operator `ClassifyPurchase` with the `cutoff` option.

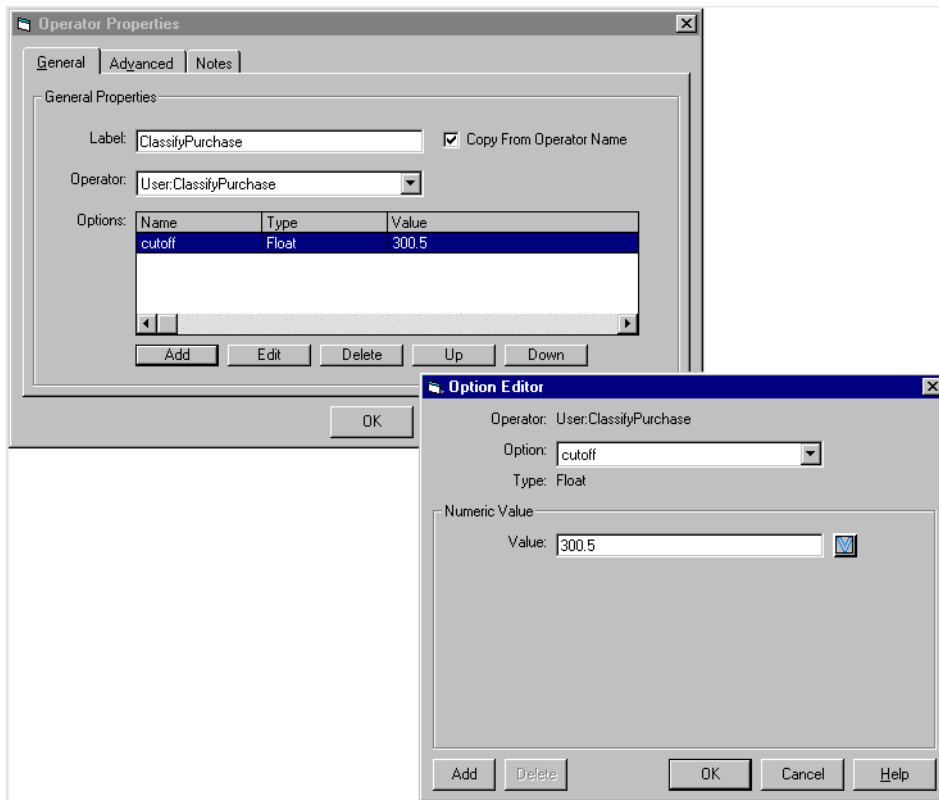
### How the Operator User Sets the Option

To set the `cutoff` option for `ClassifyPurchase`, the operator user opens the **Operator Properties** dialog box for `ClassifyPurchase` operator. The **Add** and **Edit** buttons open the **Options Editor** dialog box, with which the user adds or edits an option setting. The user can also use the **Delete** button to delete a setting, and the **Up** and **Down** buttons to rearrange the order of option settings.

To add a setting for option `cutoff`, click **Add** to open the **Option Editor**. In that dialog box, enter a value in the **Value** field. (Note that for a `boolean` option, the operator user simply selects or deselects it.)

To add the setting to the operator without closing the **Option Editor** dialog box, click **Add**. Or, the click **OK** to add the setting and close the **Option Editor** dialog box.

The following figure shows the **Option Editor** dialog box with the value 300.5 entered as a `cutoff` option setting, and the **Operator Properties** dialog box after that option has been added.



To save the setting, in the **Operator Properties** dialog box, you must either click the **Apply** button to apply the setting to the operator, or click **OK** to apply the setting and close the **Operator Properties** dialog box.

## Using Orchestrate Data Types in Your Operator

This section describes how to code your operators to use all the Orchestrate data types, in the following sections:

- “Using Numeric Fields” on page 12-21
- “Using Date, Time, and Timestamp Fields” on page 12-21
- “Using Decimal Fields” on page 12-23
- “Using String Fields” on page 12-24
- “Using Raw Fields” on page 12-25
- “Using Nullable Fields” on page 12-25
- “Using Vector Fields” on page 12-26

Orchestrate implements some of its data types using C++ classes. The *Orchestrate C++ Classes and Headers Reference Cards* provide a listing of the header (.h) files in which the Orchestrate C++ classes and macros are defined.

## Using Numeric Fields

This section contains a basic example using numeric fields. A numeric field can have one of the following data types:

- int8, int16, int32, int64
- uint8, uint16, uint32, uint64
- sfloat, dfloat

See the section “Specifying Operator Input and Output Interfaces” on page 12-8 for examples of declaring of integer fields in input and output interface schemas, and using integer fields in operator code.

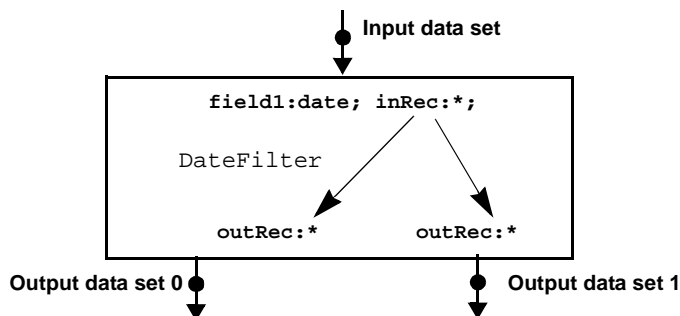
## Using Date, Time, and Timestamp Fields

Orchestrate allows you to access fields containing a date, time, or timestamp. Orchestrate uses the following C++ classes to represent these data types:

- APT\_Date
- APT\_Time
- APT\_TimeStamp

You can use any public member function of these classes in your operator. See the *Orchestrate C++ Classes and Headers Reference Cards* for a listing of the classes implementing these data types.

For example, the following figure shows an operator containing a date field in its interface schemas:



In this example, the operator copies all records with a year greater than 1990 to the output data set. All records with year before 1990 are copied to the reject data set.

**Name:** DateFilter

**Input 0 Name:** DateIn0

**Input 0 Properties:** Auto

**Input 0 Interface Schema:** record (field1:date;)

**Output 0 Name:** DateOut0

**Output 0 Properties:** Auto

**Output 0 Interface Schema:** record ()

**Output 1 Name:** DateOut1

**Output 1 Properties:** Noauto

**Output 1 Interface Schema:** record ()

**Transfer 0:** Input From: 0; Output To: 0

**Transfer 0 Properties:** Noauto, Combine

**Transfer 1:** Input From: 0; Output To: 1

**Transfer 1 Properties:** Noauto, Combine

**Per-record:**

```
// Check year component of date.
if (DateIn0.field11.year() < 1990)
// Send record to reject output
  transferAndWriteRecord(1);
else {
// Copy record to accepted output
  doTransfersTo(0);
// Write the record.
  writeRecord(0);
}
```

## Using Decimal Fields

Orchestrate implements decimal fields using the C++ class `APT_Decimal`. The most common functions of `APT_Decimal` that you may need to use in your operator are shown below:

### Class `APT_Decimal`

```

assignFromDecimal()
assignFromDFloat()
assignFromInt32()
assignFromString()
asDFloat()
asInteger()
asString()
compare()
scale()
stringLength()

```

You can use any public member function of these classes in your operator. See the *Orchestrate C++ Classes and Headers Reference Cards* to find the header file containing a complete description of `APT_Decimal`.

For example, Orchestrate decimals do not provide arithmetic functions. In order to use a decimal within an arithmetic expression, you must first convert it to an integer or float, perform your arithmetic operation, and convert it back to a decimal.

For example, the following operator adds two decimal fields and outputs a field containing the total:

**Name:** `sum`

**Input 0 Name:** `SumIn0`

**Input 0 Properties:** `Auto`

**Input 0 Interface Schema:**

```
record (field1:decimal[6,2]; field2:decimal[6,2];)
```

**Output 0 Name:** `SumOut0`

**Output 0 Properties:** `Auto`

**Output 0 Interface Schema:**

```
record (total:decimal[7,2];)
```

**Per-record:**

```

float var1 = SumIn0.field1.asDFloat();
float var2 = SumIn0.field2.asDFloat();

float total_temp = var1 + var2;
SumOut0.total.assignFromDFloat(total_temp);

```

## Using String Fields

One possible data type for a field is *string*—a field containing a fixed-length or variable-length character string. Orchestrate uses the class `APT_StringField` to define a string field. `APT_StringField` defines member functions that you use to manipulate a string field.

A string field has the following characteristics:

- The string can be fixed-length or variable-length.
- You can assign various kinds of strings to the field.
- A string field is not null-terminated; that is, it always has an explicit length.
- A string field can contain white-space characters and, if nullable, the null character (`\0`).

The following figure shows common functions in the public interface to `APT_StringField`:

### Class `APT_StringField`

```

assignFrom()
content()
isFixedLength()
isVariableLength()
length()
padChar()
setFixedLength()
setLength()
setPadChar()
setVariableLength()

```

You can use any public member function of these classes in your operator. See the *Orchestrate C++ Classes and Headers Reference Cards* to find the header file containing a complete description of `APT_StringField`.

For example, you can define an operator that takes as input a variable-length string field. To determine the runtime length of the field in an input record, you can use the string field function `length()`.

**Input 0 Name:** `LenIn0`

**Input 0 Properties:** `Auto`

**Input 0 Interface Schema:** `record (a:string;)`

**Output 0 Name:** `LenOut0`

**Output 0 Properties:** `Auto`

**Output 0 Interface Schema:** `record (a:string;)`

**Per-record:**

```

int length = LenIn0.a.length();
// If length is 0, discardRecord;
if ( length == 0 ) discardRecord();
// Process string...

```



```
// Copy string to output record
LenOut0.a = LenIn0.a;
```

## Using Raw Fields

Orchestrate lets you create raw fields, an untyped collection of contiguous bytes. You can also create an aligned raw field, a special type of raw field where the first byte is aligned to a specified address boundary. Raw and aligned raw fields may be either fixed- or variable-length.

Orchestrate uses the class `APT_RawField` to define both raw and aligned raw fields. `APT_RawField` includes member functions that you use to manipulate these fields. The following figure shows common member functions of `APT_RawField`:

### Class `APT_RawField`

<pre>assignFrom() content() isFixedLength() isVariableLength() length() setFixedLength() setLength() setVariableLength()</pre>
--

You can use any public member function of these classes in your operator. See the *Orchestrate C++ Classes and Headers Reference Cards* to find the header file containing a complete description of `APT_RawField`.

To process raw fields, you use the same techniques described above for string fields. However, you copy raw fields into a destination field of type `void *`, rather than of type `char *`.

## Using Nullable Fields

Orchestrate supports a null value representation for all field types. To define a field as nullable in an input or output interface schema, you use the `nullable` keyword, as shown in the following sample schema declarations:

**Input 0 Interface Schema:** `record (a:nullable int32; b:nullable int32; c:int8;)`

**Output 0 Interface Schema:** `record (d:nullable int32; e:int16;)`

You can specify nullability for any field in the schema. In this example, fields `a`, `b`, and `d` are declared as `nullable`, and fields `c` and `e` are declared as not `nullable` (the default). All fields in the output interface schema defined as `nullable` are initialized to null in each output record.

An expression using a `nullable` source field generates an error if the source field contains a null. For example, the following expression uses fields `a`, `b`, and `d`:

```
d = a / b;
```

In this case, both source fields (*a* and *b*) must not be null. If either field is null, the expression causes an error that terminates your application. Field *d* may contain a null, as it is the destination. The result of the expression is written to field *d*, replacing the null.

To process nullable fields, you can use the following functions:

- `fieldName_null()`: Returns the boolean value of *true* if *fieldName* contains a null and *false* if not.
- `fieldName_setnull()`: Sets an output field to null (input fields are read-only)
- `fieldName_clearnull()`: Sets an output field to non-null (input fields are read-only)

You can rewrite the example expression above to avoid an error caused by a nullable field, as shown below:

```
if (!a_null() && !b_null() )
    d = a / b;
```

## Using Vector Fields

Orchestrate record schemas allow you to define vector fields (one-dimensional arrays). Shown below is an example interface schema using vector fields:

**Input 0 Interface Schema:** `record (a[10]:int8;)`

**Output 0 Interface Schema:** `record (total:int32;)`

In this example, field *a* is a 10-element vector.

You can use the following functions with vector fields:

- `fieldName[index]`: Accesses the vector element at the specified index, where the first element in the vector is at index 0. Vector elements in an input record are read-only, and vector elements in an output record are read/write.
- `fieldName.vectorLength()`: Returns, as an integer, the number of elements in the vector
- `fieldName.setVectorLength(length)`: Sets the length of a variable-length vector field in an output record (input fields are read-only)
- `fieldName_null(index)`: Returns the boolean value *true* if the vector element at the specified index is null, and *false* if it is not
- `fieldName_setnull(index)`: Sets the vector element in an output record at the specified index to null (input fields are read-only)
- `fieldName_clearnull(index)`: Sets the vector element in an output record at the specified index to non-null (input fields are read-only)

The following code sums all the elements in the input vector *a* and stores the results in the output field *total*:

```
total = 0;
```

```
for (int i = 0; i < 10; i++)
{
    total = total + a[i];
}
```

## Using the Custom Operator Macros

Orchestrate provides a number of macros for you to use in **Pre-loop**, **Post-loop**, and **Per-record** code for your custom operator. This section describes these macros, which are grouped into the following four categories:

- Informational macros
- Flow-control macros
- Input and output macros
- Transfer macros

### Informational Macros

You can use the informational macros in your operator code, to determine the number of inputs, outputs, and transfers, as follows:

<code>inputs()</code>	Returns number of declared inputs.
<code>outputs()</code>	Returns number of declared outputs.
<code>transfers()</code>	Returns number of declared transfers.

### Flow-Control Macros

Flow-control macros let you override the default behavior of the input and processing loop in your operator's action section. Modifying the loop makes your operator code more complex and more difficult to debug. Before coding with these macros, be sure to carefully read the section "How Visual Orchestrate Executes Generated Code" on page 12-31 and the section "Designing Operators with Multiple Inputs" on page 12-31.

<code>endLoop()</code>	Causes operator to stop looping, following completion of the current loop and after writing any <code>auto</code> outputs for this loop.
<code>nextLoop()</code>	Causes operator to immediately skip to start of next loop, without writing any outputs.
<code>failStep()</code>	Causes operator to return a <code>failed</code> status and terminate the enclosing Orchestrate step.

For an example of using `endLoop()` in an `$action` section, see the section “Using Both `auto` and `noauto` Inputs” on page 12-34.

## Input and Output Macros

The input and output macros let you control read, write, and transfer of individual records.

Each of the input and output macros listed below takes an argument, as follows:

- An `input` argument is the index (0-*n*) of the declared input, and an `output` argument is the index (0-*n*) of the declared output. If you have defined a port name for an input or output, you can use `portname.portid_` in place of the index.
- An `index` argument is the index (0-*n*) of the declared transfer.

<code>readRecord(input)</code>	Immediately reads the next record from <code>input</code> , if there is one. If there is no record, the next call to <code>inputDone()</code> will return <code>false</code> .
<code>writeRecord(output)</code>	Immediately writes a record to <code>output</code> .
<code>inputDone(input)</code>	Returns <code>true</code> if the last call to <code>readRecord()</code> for the specified <code>input</code> failed to read a new record, because the <code>input</code> has no more records.
<code>holdRecord(input)</code>	Causes <code>auto</code> input to be suspended for the current record, so that the operator does not automatically read a new record at the start of the next loop. If <code>auto</code> is not set for the <code>input</code> , <code>holdRecord()</code> has no effect.
<code>discardRecord(output)</code>	Causes <code>auto</code> output to be suspended for the current record, so that the operator does not output the record at the end of the current loop. If <code>auto</code> is not set for the <code>output</code> , <code>discardRecord()</code> has no effect.
<code>discardTransfer(index)</code>	Causes <code>auto</code> transfer to be suspended, so that the operator does not perform the transfer at the end of the current loop. If <code>auto</code> is not set for the <code>transfer</code> , <code>discardTransfer()</code> has no effect.

For examples of using input and output macros, see the following:

- `readRecord()` in the section “Example Operator: reject” on page 12-30
- `writeRecord()` in several examples, including the section “Example Operator: reject” on page 12-30 and “Example Operator: noauto” on page 12-34
- `inputDone()` in “Example Operator: noauto” on page 12-34
- `discardRecord()` and `discardTransfer()` in the example below

### Example Using `discardRecord()` and `discardTransfer()`: Operator `divide`

The following example definition file is for operator `divide`, which calculates the quotient of two input fields. The operator checks to see if the divisor is zero. If it is, `divide` calls `discardTrans-`

`fer()` so that the record is not transferred, and then calls `discardRecord()`, which drops the record (does not copy it from the input data set to the output data set).

**Name:** `divide`

**Input 0 Name:** `[default]`

**Input 0 Properties:** `Auto`

**Input 0 Interface Schema:** `record(a:int32; b:int32;)`

**Output 0 Name:** `[default]`

**Output 0 Properties:** `Auto`

**Output 0 Interface Schema:** `record(quotient:int32; remainder:int32;)`

**Transfer 0:** `Input From: 0; Output To: 0`

**Transfer 0 Properties:** `Auto, Combine`

**Per-record:**

```
if (b == 0) {
    discardTransfer(0); // Don't perform transfer.
    discardRecord(0);  // Don't write output record.
}
else {
    quotient = a / b;
    remainder = a % b;
}
```

## Transfer Macros

This section describes how to use the transfer macros. Each of the transfer macros, listed below, takes an argument, as follows:

- An `input` argument is the index (0-*n*) of the declared input, and an `output` argument is the index (0-*n*) of the declared output. If you have defined a port name for an input or output, you can use `portname.portid_` in place of the index.
- An `index` argument is the index (0-*n*) of the declared transfer.

<code>doTransfer(<i>index</i>)</code>	Performs the transfer specified by <code>index</code> .
<code>doTransfersFrom(<i>input</i>)</code>	Performs all transfers from <code>input</code> .
<code>doTransfersTo(<i>output</i>)</code>	Performs all transfers to <code>output</code> .
<code>transferAndWriteRecord(<i>output</i>)</code>	Performs all transfers and writes a record for the specified output. Calling this macro is equivalent to calling the macros <code>doTransfersTo(<i>output</i>)</code> and <code>writeRecord(<i>output</i>);</code>

A transfer copies the input record to the output buffer. If your definition file specifies `auto` transfer (the default setting), immediately after execution of the **Per-record** code, the operator transfers the input record to the output record.

For an example of using `doTransfer()`, see the section “Example Operator: score” on page 12-32.

The code example below shows how you can use `doTransfersTo()` and `transferAndWriteRecord()` in a sample operator, `reject`. Following the code description is an example of using `reject` in an `osh` command.

### Example Operator: `reject`

This example operator, `reject`, has one automatic input. It has two non-automatic outputs. It also declares two non-automatic transfers, the first to output 0 and the second to output 1.

The input record holds a dividend in field `a` and a divisor in field `b`. The operator checks the divisor, and if it is zero, calls `transferAndWriteRecord(1)` to perform a transfer to output 1 and write the record to output 1. If the divisor is not zero, the operator calls `doTransfersTo(0)` to perform the transfer to output 0, assigns the division results to the fields `quotient` and `remainder`, and finally to call `writeRecord(0)` to write the record to output 0.

**Name:** `reject`

**Input 0 Name:** [default]

**Input 0 Properties:** `Auto`

**Input 0 Interface Schema:** `record(a:int32; b:int32;)`

**Output 0 Name:** [default]

**Output 0 Properties:** `Noauto`

**Output 0 Interface Schema:** `record(quotient:int32; remainder:int32;)`

**Transfer 0:** `Input From: 0; Output To: 0`

**Transfer 0 Properties:** `Noauto, Combine`

**Output 1 Name:** [default]

**Output 1 Properties:** `Noauto`

**Output 1 Interface Schema:** `record()`

**Transfer 0:** `Input From: 0; Output To: 1`

**Transfer 0 Properties:** `Noauto, Combine`

#### Per-record:

```
if (b == 0) {
// Send record to reject output
  transferAndWriteRecord(1);
}
else {
  // Copy record to normal output
  doTransfersTo(0);
  // Set additional data fields.
  quotient = a / b;
  remainder = a % b;
  // Write the record.
  writeRecord(0);
}
```

## How Visual Orchestrate Executes Generated Code

This section describes how Visual Orchestrate executes the code generated from your custom operator definition. If you are writing simple operators and your code uses only the default, automatic I/O handling, you do not need to be concerned with the details of this section. If you are writing more complex operators with non-default I/O handling, these details can be helpful.

Visual Orchestrate executes your custom operator code, as follows:

1. Handles any definitions that you have entered in the **Definitions** tab of the **Custom Operator** dialog box.
2. If you have entered any code in the **Pre-loop** section, executes it.
3. Loops repeatedly until either all inputs have run out of records, or the **Per-record** code has explicitly invoked `endLoop()`. In the loop, performs the following steps:
  - a. Reads one record for each input, except where any of the following is true:
    - The input has no more records left.
    - The input has been declared with `noauto`.
    - The `holdRecord()` macro was called for the input last time around the loop.
  - b. Executes the **Per-record** code, which can explicitly read and write records, perform transfers, and invoke loop-control macros such as `endLoop()`.
  - c. Performs each specified transfer, except where any of the following is true:
    - The input of the transfer has no more records.
    - The transfer has been declared with `noauto`.
    - The `discardTransfer()` macro was called for the transfer during the current loop iteration.
  - d. Writes one record for each output, except where any of the following is true:
    - The output was declared with `noauto`.
    - The `discardRecord()` macro was called for the output during the current loop iteration.
4. If you have specified **Post-loop** code, executes it.
5. Returns a status value, which is one of the following:
  - `APT_StatusOk` (value 0). The default.
  - `APT_StatusFailed` (value 1). Returned only if your code has invoked `failStep()`.

## Designing Operators with Multiple Inputs

Orchestrate supports creation of operators that perform complex data handling, through use of multiple inputs and outputs, flow control, and transfers. This section describes some suggested approaches for designing operators that successfully handle the challenge of using multiple inputs.

---

**Note:** Whenever possible, use the standard, automatic read, write, and record transfer features described in this chapter. In using non-automatic inputs, outputs, or record transfers in your operator definition, you introduce complexity and a greater possibility of programming errors.

---

## Requirements for Coding for Multiple Inputs

To use multiple inputs effectively, your operator code needs to meet the following two requirements:

- Perform a read of a record's fields only when the record is available on the specified input.  
Make sure that your operator code will not attempt to read a field from an input with no more records. It also must not attempt to read a field on a non-automatic input before it has performed a read on that input. Failure to prevent either of these situations causes the operator to fail with an error message similar to the following:

```
isUsable() false on accessor interfacing to field "fieldname"
```

- Terminate the reading of records from each input immediately after (but not before) all needed records have been read from it.

If you declare any inputs as non-automatic, your operator must determine when all needed records have been read from all inputs, and at that point to terminate the operator by calling the `endLoop()` macro. Remember that the operator continues to loop as long as there are records remaining on any of its inputs.

## Strategies for Using Multiple Inputs and Outputs

In general, the best approach to coding for multiple inputs is the simplest, using the automated features as much as possible. Below are three simple strategies for designing an operator with multiple inputs:

### Using Automatic Read for All Inputs

In this approach to multiple inputs, your definition file defines all inputs as automatic, so all record reads are handled automatically. You code your operator so that each time it accesses a field on any input, it first invokes the `inputDone()` macro to determine whether there are any more records on the input.

Note that this strategy is appropriate only if you want your operator to read a record from every input, every time around the loop.

#### **Example Operator:** `score`

This example operator, `score`, uses two automatic inputs. It also uses non-automatic outputs and record transfers, so that it writes record only when the operator code has determined that its score field has the highest value among all records of that `type`.



**Name:** score

**Input 0 Name:** [default]

**Input 0 Properties:** Auto

**Input 0 Interface Schema:** record(type:int32; score:int32;)

**Output 0 Name:** [default]

**Output 0 Properties:** Noauto

**Output 0 Interface Schema:** record ( )

**Transfer 0:** Input From: 0; Output To: 0

**Transfer 0 Properties:** Noauto, Combine

**Pre-loop:**

```
int current_type = -1;
int current_score = -1;
```

**Per-record:**

```
// Operator score uses the output record as a buffer for
// the highest-scoring record of each type processed.
// Assumptions about input:
// The input data is hash-partitioned on the type field.
// All records are guaranteed to have a score value not equal to -1.
// If the input has passed the final record of a type,
// output the buffered high-scoring record for that type,
// and reset the score.

if (type != current_type && current_type != -1) {
    // write highest scored record from previous group.
    writeRecord(0);
    // start tracking new type
    current_type = type;
    current_score = -1;
}

// If current record beats previous score, transfer it
// to the output buffer, but don't write it yet.
if (score > current_score) {
    doTransfersTo(0);
    current_score = score;
    current_type = type;
}

$post

// If there's a remaining record, write it to output.
if (current_type != -1) {
    writeRecord(0);
}
```

## Using Both `auto` and `noauto` Inputs

Your definition file declares one automatic input (or possibly more than one), and the remaining inputs as non-automatic. You code your operator to let the processing of records from the automatic input drive the handling of the other inputs. Each time around the loop, your operator calls `inputDone()` on the automatic input. When the automatic input has no more records, your code calls `exitLoop()` to complete the operator action.

For an example of an operator that uses this coding strategy, see the section “Example Operator: reject” on page 12-30.

## Using `noauto` for All Inputs

Your definition file declares all inputs as non-automatic (`noauto`), so your code must perform all record reads. You declare a **Pre-loop** section, which you code to call `readRecord()` once for each input. You code the **Per-record** section to invoke `inputDone()` for every input, on every iteration of the loop, to determine whether it obtained a record on the most recent `readRecord()`. If it did, process the record, and then call `readRecord()` on that input. When all inputs run out of records, the operator automatically exits the **Per-record** section.

### Example Operator: `noauto`

The following example operator, `noauto`, illustrates this strategy for reading multiple inputs. This example uses non-automatic inputs, outputs, and transfers, so that the operator code performs all reading and writing explicitly. This operator takes three inputs, with similar schemas, and merges them into a single output. It also has a separate output for records that have been rejected.

Unlike the Orchestrate `merge` operator, this operator filters and modifies some of the input. This operator modifies the `id` field to reflect the input source (0, 1, or 2). It also filters records from inputs 1 and 2, according to the values of fields `a` and `b`.

**Name:** `noauto`

**Input 0 Name:** [default]

**Input 0 Properties:** `Noauto`

**Input 0 Interface Schema:** `record (id:int32;)`

**Input 1 Name:** [default]

**Input 1 Properties:** `Noauto`

**Input 1 Interface Schema:** `record (id:int32; a:int32;)`

**Input 2 Name:** [default]

**Input 2 Properties:** `Noauto`

**Input 2 Interface Schema:** `record (id:int32; a:int32; b:int32;)`

**Output 0 Name:** [default]

**Output 0 Properties:** `Noauto`

**Output 0 Interface Schema:** `record (id:int32;)`

**Output 1 Name:** [default]

**Output 1 Properties:** `Noauto`

**Output 1 Interface Schema:** record ()**Transfer 0:** Input From: 0; Output To: 0**Transfer 0 Properties:** Noauto, Combine**Transfer 1:** Input From: 1; Output To: 0**Transfer 1 Properties:** Noauto, Combine**Transfer 2:** Input From: 2; Output To: 0**Transfer 2 Properties:** Noauto, Combine**Transfer 3:** Input From: 0; Output To: 1**Transfer 3 Properties:** Noauto, Combine**Transfer 4:** Input From: 1; Output To: 1**Transfer 4 Properties:** Noauto, Combine**Transfer 5:** Input From: 2; Output To: 1**Transfer 5 Properties:** Noauto, Combine**Pre-loop:**

```
// Before looping, it initially calls readRecord() once for each input,
// either to get the first record or to detect that input is empty.
int i;
for (i=0; i<inputs(); i++) {
  readRecord(i);
}
```

**Per-record:**

```
// Each time around the loop, look for any input that is
// not done, i.e. has a record, and process that input.
for (i=0; i<inputs(); i++) {
  if (!inputDone(i)) {
    // Process current record for this input
    switch (i) {
      case 0:
        // Input 0 needs no processing, so just copy to output.
        out0.id = in0.id;
        doTransfer(0); // from input 0 to output 0
        writeRecord(0);
        break;
      case 1:
        // Input 1 needs some filtering/rejection of records.
        if (in1.a > 50) {
          // Reject the record.
          doTransfer(4); // from input 1 to output 1
          writeRecord(1);
        }
        else {
          // Modify the id, and copy record to output 0.

```

```
        out0.id = in1.id + 10000;
        doTransfer(1); // from input 1 to output 0
        writeRecord(0);
    }
    break;
case 2:
    // Input 2 needs different filtering/rejection.
    if (in2.a > 50 || in2.b > 50) {
        // reject the record
        doTransfer(5); // from input 2 to output 1
        writeRecord(1);
    }
    else { // Modify the id appropriately.
        if (in2.a > in2.b)
            out0.id = in2.id + 20000;
        else
            out0.id = in2.id + 30000;
        // Copy the record to output 0.
        doTransfer(2); // from input 2 to output 0
        writeRecord(0);
    }
    break;
}
// Get next record for this input (or detect end of input).
readRecord(i);
}
}

// When all inputs are exhausted, operator automatically terminates.
```

# 13: Creating UNIX Operators

Orchestrate lets you use UNIX programs and commands in your Orchestrate application, just as you use predefined Orchestrate operators. This capability lets you run existing UNIX programs in parallel within an Orchestrate application, without having to rewrite them.

To execute your existing UNIX programs in Orchestrate, you create UNIX command operators. UNIX command operators execute your UNIX application in parallel or sequentially as part of an Orchestrate step. You can use UNIX command operators in the same step that you use Orchestrate operators.

This chapter describes how to create UNIX command operators. It describes how to handle various kinds of operator inputs and outputs and how to define operators that accept user-settable options. The final section of this chapter describes optimizations that Orchestrate performs with UNIX command operators.

This chapter contains the following sections:

- “Introduction to UNIX Command Operators” on page 13-1
- “Handling Operator Inputs and Outputs” on page 13-7
- “Passing Arguments to and Configuring UNIX Commands” on page 13-22
- “Handling Command Exit Codes” on page 13-35
- “How Orchestrate Optimizes Command Operators” on page 13-36

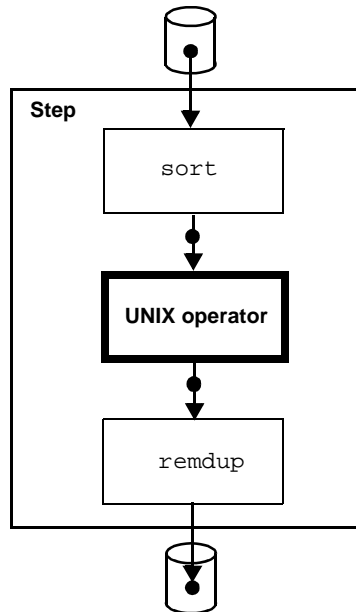
## Introduction to UNIX Command Operators

The Orchestrate framework includes many operators, which you use to perform basic data processing tasks such as copy, sample, and merge. Orchestrate also lets you create custom operators from existing or new code, to run in parallel in your Orchestrate application. This chapter describes how to use Visual Orchestrate to create custom operators from existing UNIX shell commands. UNIX shell commands can be invoked from a UNIX shell prompt, and include UNIX built-in commands such as `grep`, UNIX utilities such as `SyncSort`, and your own UNIX applications. (You can also create custom operators from your own code in C or C++; see the chapter “Creating Custom Operators”.)

Executing an existing UNIX application as a UNIX command operator has two main advantages:

1. Orchestrate's parallel execution improves the performance of your UNIX application, as it can process larger amounts of data than it can in sequential execution.
2. You can use your existing UNIX application, without any reimplementations, in your Orchestrate application.

You insert a UNIX command operator into a step just as you would any other operator. The following figure shows a UNIX command operator used in an Orchestrate step:



UNIX command operators can be executed sequentially or in parallel. When executed sequentially, the UNIX command operator adds the functionality of your existing application code to your Orchestrate application.

When executed in parallel, your UNIX command operator also takes advantage of Orchestrate's parallel execution performance. Orchestrate handles all the underlying tasks required to run parallel applications, including partitioning your data on the multiple nodes in a parallel system.

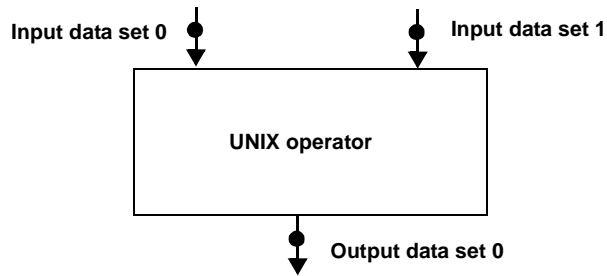
## Characteristics of a UNIX Command Operator

A UNIX command operator has the following characteristics:

- Parallel or sequential execution mode (selectable by the creator or user of the operator)
- Partitioning method of *any* (parallel execution mode) and collection method of *any* (sequential execution mode)
- Multiple input and output data sets
- Execution environment (environment variable definitions, exit code handling) controlled by the operator creator

The UNIX operator in the figure above takes a single data set as input and produces a single data set as output. You can also create UNIX command operators that take multiple inputs and outputs.

The following figure shows an operator that takes two input data sets and produces one output data set:



In addition, you can build node and resource constraints into the operator.

## UNIX Shell Commands

The command executed by a UNIX command operator is any UNIX application, file, or command that can be executed from a UNIX shell prompt. UNIX shell commands include:

- A UNIX-executable program, compiled from code written in COBOL, C, C++, or another programming language
- A built-in UNIX command, such as `grep`, or a UNIX utility, such as `SyncSort`
- A UNIX shell script

### Overview of UNIX Commands

This section contains an overview of UNIX shell commands, terms, and syntax. You will find this information useful when designing and developing your UNIX command operators.

A UNIX command has the following general form:

```
command_name arg_list input_list output_list
```

where:

- *command\_name* is the name of the UNIX shell command. Example shell commands are `grep`, `sort`, or any executable file or shell script.
- *arg\_list* is a list of arguments to the command, required and optional. For example, the `SyncSort` sorting package takes arguments that define the sorting characteristics. Some commands use environment variables to set arguments.
- *input\_list* is a list of inputs to the command. Inputs include data sets and other kinds of data, such as parameter files.
- *output\_list* is a list of outputs from the command. Outputs can be any output data, error or log messages, and result summaries.

For example, the UNIX `grep` command has the following form:

```
grep [ -bchilnsvw ] limited-regular-expression [ filename ... ]
```

The `grep` command searches one or more input files for a pattern and outputs all lines that contain that pattern. If the command line specifies no input files, `grep` takes its input from the standard input. By default, `grep` writes output to standard output.

UNIX commands take their input from one of the following sources:

- Standard input (`stdin`): Each UNIX command can have a single connection to standard input. By default, standard input is the keyboard.
- Files: Data files specified as arguments to the UNIX command.

UNIX commands write their output to one of the following:

- Standard output (`stdout`): A UNIX command can have a single connection to standard output. By default, standard output is the screen.
- Standard error (`stderr`): A UNIX command can have a single connection to standard error.
- Files: Data files are specified as arguments to the UNIX command.

For example, the following command line causes `grep` to read its input from the file `custs.txt` and to write all lines of the file containing the string `MA` to standard output (the screen):

```
grep MA custs.txt
```

You can also use standard input and output to connect UNIX commands, as shown below:

```
cat options | grep options
```

where *options* are any arguments passed to the commands. The pipe symbol (`|`) indicates a connection between two UNIX commands. In this example, `cat` writes its output to its standard output, and `grep` reads its input from `cat` on the standard input of `grep`.

## UNIX Commands Must Be Pipe-Safe

Orchestrate imposes one requirement on the UNIX command executed by a UNIX command operator: the command must be *pipe-safe*. To be pipe-safe, a UNIX command always reads its input sequentially, from beginning to end. More specifically, the command does not use the UNIX random-access command, `seek`, on any of its inputs.

UNIX commands that are pipe-safe can be connected with a UNIX pipe, as shown below:

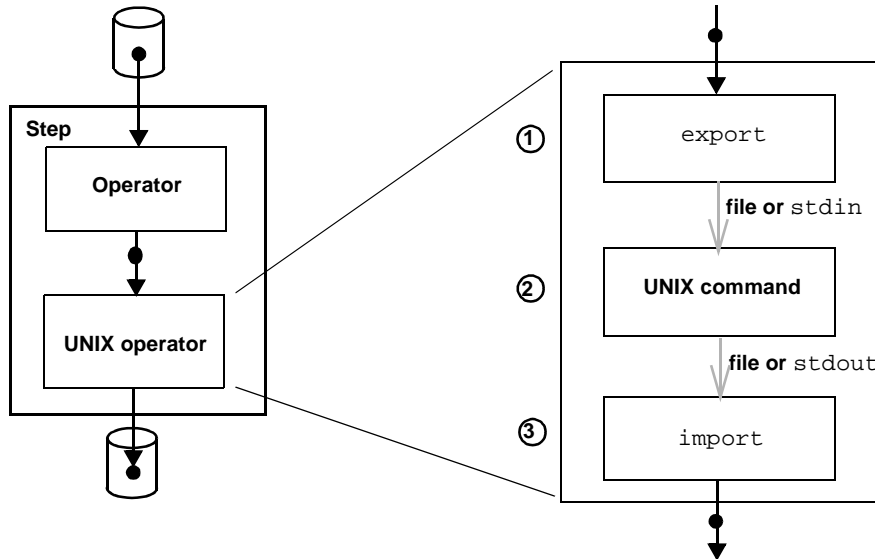
```
$ cat options | grep options | sort options
```

UNIX commands that read from standard input or write to standard output are pipe-safe and can be included in a UNIX command operator. UNIX commands that read from or write to files are usually pipe-safe.



## Execution of a UNIX Command Operator

A UNIX command operator takes as input a data set containing data in the Orchestrate record format. The embedded shell command then processes Orchestrate records one by one. The left-hand side of the figure below shows an Orchestrate application step that uses a UNIX command operator, and the right-hand side shows an expanded representation of the operator as it processes data.



In the figure above, the UNIX command operator performs both an export and an import operation. It performs the export to convert the input data set to the data format required by the UNIX command, and it performs the import to convert the output of the UNIX command to the Orchestrate data set format. The UNIX command, executed by the command operator, receives and processes data normally from either standard input or a file.

The following procedure summarizes the action of the UNIX command operator:

1. To convert input records to a format compatible with the UNIX command, Orchestrate exports the data set records.

You control how the output of the `export` operator is delivered to the input of the UNIX command: either through the command's standard input or through a file name. If you use a file name, rather than writing the data to disk Orchestrate creates a UNIX pipe that the command reads in the same way that it reads a file.

2. The UNIX command processes its input and writes its output to standard output or to a file.
3. The result of the UNIX command is sent to the `import` operator, which converts it to an Orchestrate data set.

You control how the output of the UNIX command is delivered to the input of the `import` operator: either through the command's standard output or through a file name. If you use a file name, rather than writing the data to disk Orchestrate creates a UNIX pipe that import reads as it reads a file.

For UNIX command operators with multiple inputs, Orchestrate separately exports data for each input. For each output of the operator, Orchestrate separate imports data for each output.

## Default Properties for Operator Export and Import

By default, a UNIX command operator uses the following export properties:

- A new-line character delimits the end of each record.
- An ASCII space (0x20) separates all fields.
- The record uses a text representation for all data including numeric data (an unpacked ASCII representation with one byte per digit).

This export format corresponds to the record schema:

```
record()
```

With this default export format, the embedded UNIX command receives its input as text strings terminated by new-line characters.

By default, a UNIX command operator imports data by creating one output record from each new-line delimited text string that is output by the embedded UNIX shell command. Each output record contains a single, variable-length, string field named *rec*. This default import format corresponds to the record schema:

```
record(rec:string;)
```

Many UNIX command operators require a non-default schema for import or export. For information on specifying the import and export schemas, see the section “Handling Operator Inputs and Outputs” on page 13-7 for more information.

Orchestrate can perform several kinds of performance optimization on your UNIX command operators, based on their import and export record schemas. See the section “How Orchestrate Optimizes Command Operators” on page 13-36 for more information.

## Handling Other Input/Output Types

The example UNIX command operators shown above have taken data sets as inputs and generated data sets as outputs. However, many UNIX commands operators handle other types of inputs and outputs, including;

- Input parameter files used to configure the command
- Environment variables specifying configuration information
- Output message and error logs containing information generated by the command during execution
- Output files containing results or result summaries

Inputs and outputs of these types do not have to be represented by Orchestrate data sets. Instead, the UNIX command can read directly from these inputs or write directly to these outputs.

For example, when a step passes the name of a parameter file as an argument to a UNIX command operator, the UNIX command directly reads the parameter file for its configuration information. See the section “Handling Configuration and Parameter Input Files” on page 13-25 for more information.

Likewise, an embedded UNIX command can write directly to message and error logs. However, if you are running the UNIX command operator in parallel, you must make sure that each instance of the operator writes to a separate file, as defined by a file name or a path name. See the section “Handling Command Exit Codes” on page 13-35.

If an embedded UNIX command uses environment variables, the command must be able to access the environment variable to obtain runtime configuration information. If the UNIX command operator runs in parallel, you must make sure that the environment variable is set correctly on all processing nodes executing the operator. See the section “Using Environment Variables to Configure UNIX Commands” on page 13-26 for more information.

## Handling Operator Inputs and Outputs

One of the most important aspects of creating UNIX command operators is handling the various kinds of inputs and outputs that UNIX commands can require. UNIX commands can take the following kinds of inputs:

- Data (from a file, or standard input)
- Command-line arguments
- Parameter files

UNIX commands can write the following kinds of output:

- Data (to a file, standard output, or standard error)
- Messages (to a file, standard output, or standard error)
- Results (to a file or standard output)

Orchestrate uses a data-flow programming environment, in which you define applications by connecting operators using data sets. In this way, data flows from the beginning of a step to the end, as each operator processes a stream of input records to create an output record stream.

You use Orchestrate data sets to input data to an operator and to hold the output data written by an operator. By using data sets, you let Orchestrate handle the partitioning of your data to distribute it to the processing nodes executing a parallel operator. In addition, data sets let you store and retrieve your data in parallel.

For command operators that invoke UNIX commands that require other types of inputs and outputs, you must decide how your operator will handle those inputs and outputs. This section

describes how to handle different inputs to and outputs from an UNIX command operator, through the following topics:

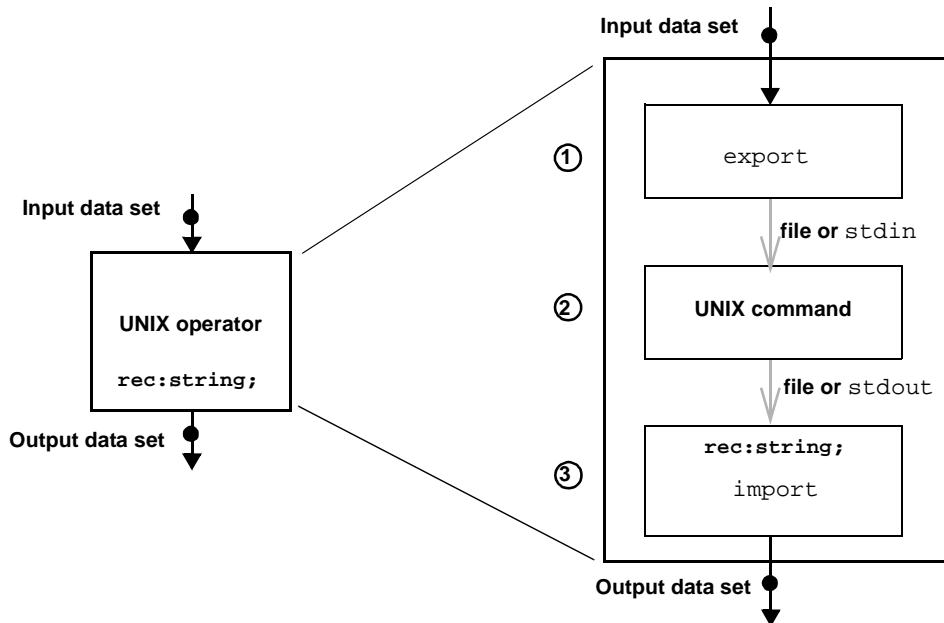
- “Using Data Sets for Inputs and Outputs” on page 13-8
- “Handling Command Exit Codes” on page 13-35
- “Handling Configuration and Parameter Input Files” on page 13-25
- “Using Environment Variables to Configure UNIX Commands” on page 13-26

## Using Data Sets for Inputs and Outputs

The application data processed by a UNIX command operator is usually represented by an Orchestrate data set. Orchestrate data sets have a parallel representation that lets you partition your data to distribute it to the multiple processing nodes in a parallel system.

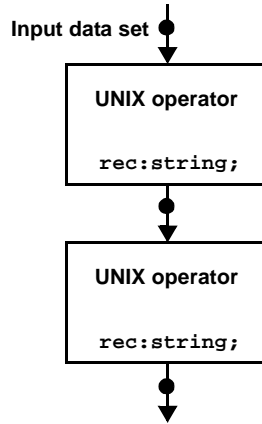
When you create a UNIX command operator, you specify how input and output data sets are connected to the UNIX command. For example, UNIX commands usually read input data from standard input (`stdin`) or from a file. For output, commands usually write output data to standard output (`stdout`) or to a file.

For example, the following figure shows a UNIX command operator taking a single input data set and creating a single output data set:



In this example, the UNIX command operator performs an export of the data set to the UNIX command, runs the UNIX command, and then imports the data to the output data set.

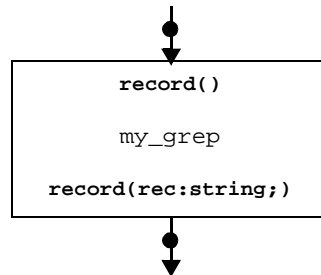
UNIX command operators use a default record schema for the input and output data sets, as described in the section “Default Properties for Operator Export and Import” on page 13-6 for a description of these defaults. The following figure shows two UNIX command operators, each using the default input and output record schemas. The two command operators are connected by a data set:



In this case, the output data set of the first UNIX command operator contains a single string field. By default, the second operator converts that string to a new-line delimited string and then inputs it to its UNIX command.

## Example: Operator Using Standard Input and Output

This section describes how to create a parallel UNIX command operator that runs the UNIX command `grep`. The following figure shows this sample operator, `my_grep`:



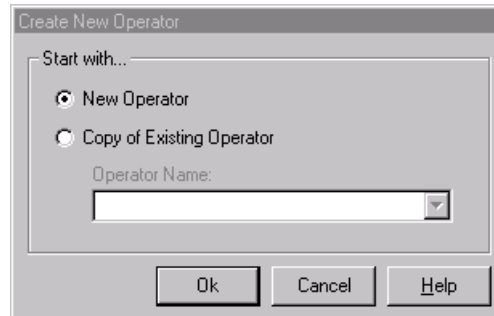
In this example, the `grep` command:

- Takes its input from standard input.
- Writes its output to standard output.
- Uses the default export schema on its input data set and the default import schema on its output data set.
- Takes no user-specified arguments.
- Is invoked with the following command line: `grep MA`.

This command line defines a simple filter operator that outputs only the records containing the string MA.

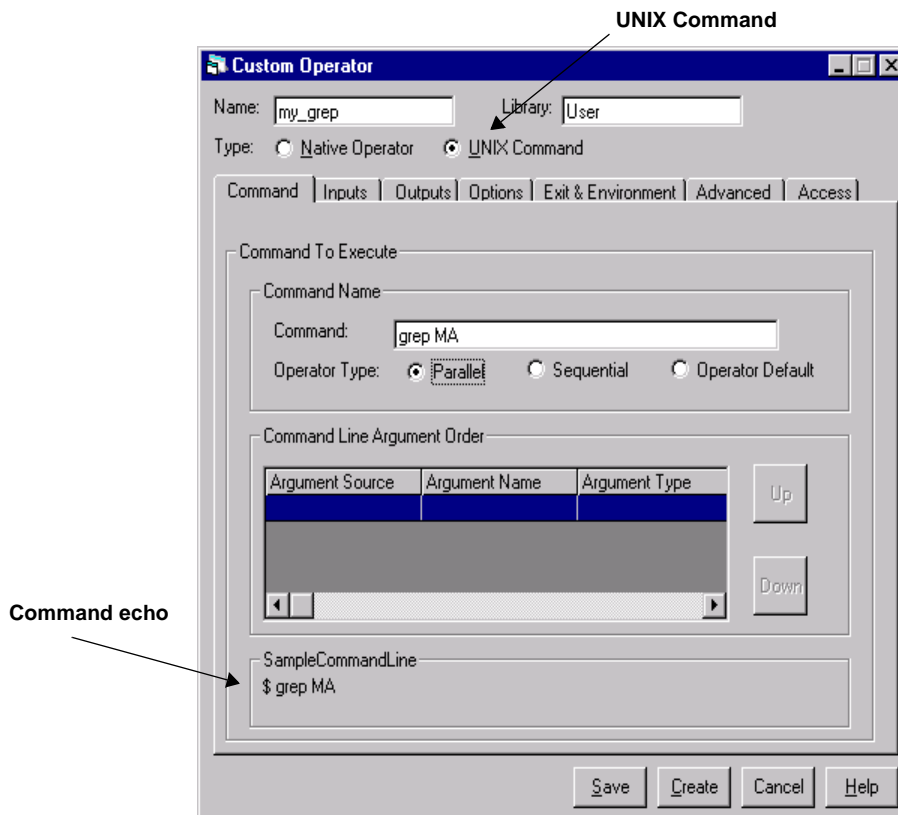
To create a UNIX command operator, perform the following steps:

1. Choose **Custom -> Define Custom Operator** from the Orchestrate menu. This command opens the following dialog box:



This dialog box allows you to create a new operator (default), or to copy the definition of an existing operator and edit that definition to create a new operator. You create a **New Operator** in this example.

2. Click **OK** to open the **Custom Operator** dialog box.
3. Click the **UNIX Command** button, as shown in the figure below:



4. Specify the **Name** of the operator: `my_grep`.
5. Specify the **Library** name for the operator: `User`.
6. Specify the **Command**: `grep MA`.

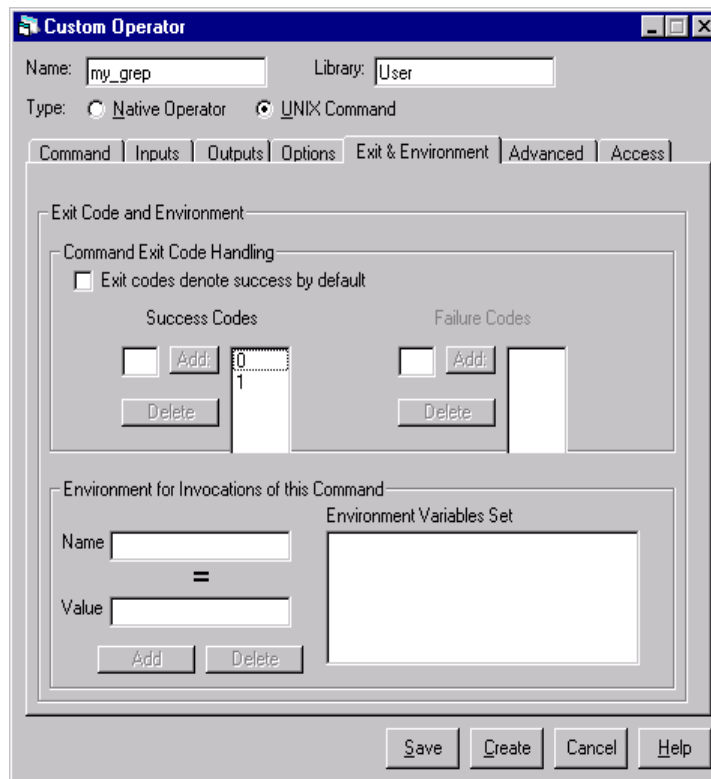
As you enter the command, you see it echoed in the **Sample Command Line** area of the dialog box. This area shows how the UNIX application will be invoked by the operator.

In the **Command Line Argument Order** area, the list of operator arguments is empty. For a description of adding and using command operator arguments, see the section “Handling Operator Inputs and Outputs” on page 13-7.

7. Specify the **Operator Type** as **Parallel**.

You can specify **Parallel**, **Sequential**, or **Operator Default** (default). **Operator Default** allows the operator user to set the execution mode when inserting the operator into a step.

8. Choose the **Exit & Environment** tab:



This dialog box lets you define the environment variables used by the UNIX command and control how Orchestrate handles exit codes returned by the command.

In the **Command Exit Code Handling** area, specify how Orchestrate interprets the exit codes returned from the UNIX command. By default, Orchestrate treats an exit code of 0 as successful and all others as errors.

For this operator, specify:

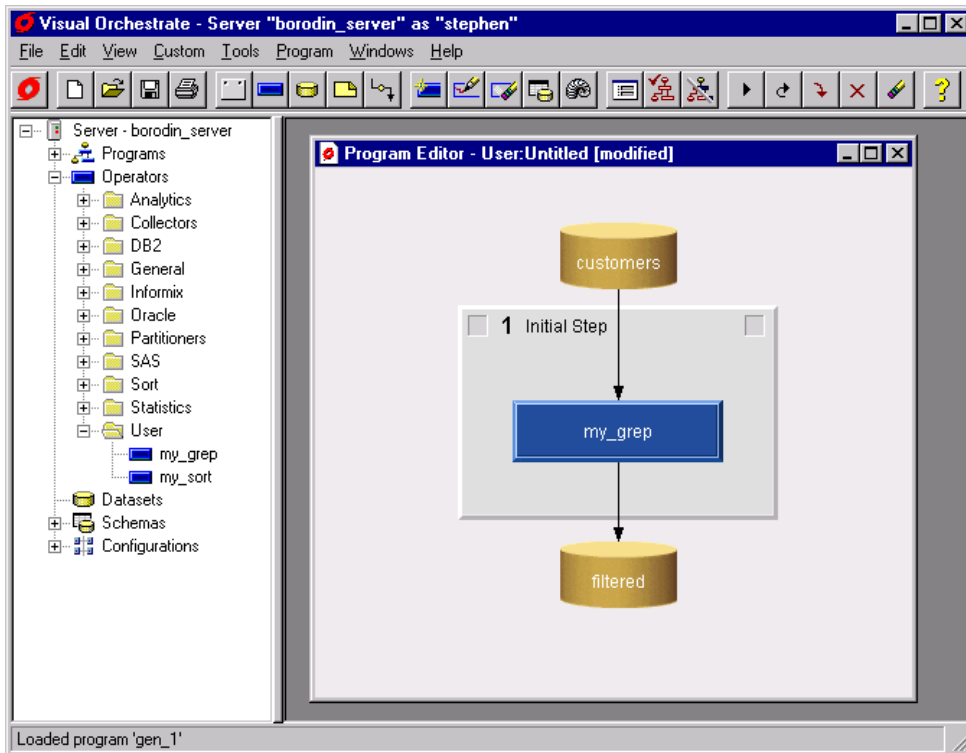
**Success Codes** = 0 and 1: Exit codes 0 and 1 indicate success; all others are errors.

You must set 1 as a valid exit code because `grep` returns both 0 and 1 to indicate success. An exit code of 0 means `grep` found the search string; an exit code of 1 means it did not.

See the section “Handling Command Exit Codes” on page 13-35 for more information.

9. Enter 1 in the **Success Code** field.
10. Press the **Add** button to add 1 as a success code.
11. Use the **New Operator** dialog box buttons to do one of the following:
  - **Save** the information about the operator, but do not create it. A saved operator appears in the Server View area in parentheses, to indicate that it has not yet been created.
  - **Create** the operator so that you can use it in your application. You must create the operator before an operator user can use it.
  - **Cancel** operator creation.
  - Open on-line **Help**.

If you chose **Create** above, you can now use the operator in an Orchestrate step, as you use a predefined Orchestrate operator. The following figure shows a step using this operator:



### Additional Tabs in the Custom Operator Dialog Box

For this example, you can leave the settings for the three tabs **Inputs**, **Outputs**, and **Options** in their default states. The operator in this example takes as its input, data received over the default standard input and uses the default export schema. It writes its output to the default standard output, and it uses the default import schema. The operator defines no user-settable options (for a descrip-



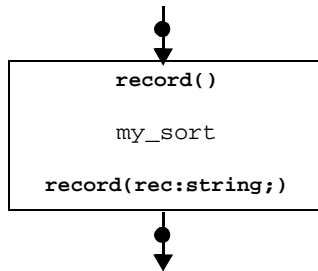
tion of setting options, see the section “Handling Operator Inputs and Outputs” on page 13-7).

You can use the **Advanced** tab to set the **Constraints** option for the operator. Constraints let you specify node pool or resource pool constraints on the operator. See the chapter “Constraints” for more information on using constraints.

To change the owner or access rights for the operator, use the **Access** tab.

## Example: Operator Using Files for Input and Output

This section shows how to create a UNIX command operator that runs the UNIX SyncSort utility in parallel. The following figure shows this sample operator, `my_sort`:



In this example, the SyncSort command is invoked with the following command line:

```

syncsort /fields age 1 integer 1 /fields state 12 char 2
  /keys age
  /statistics
  /infile source /outfile dest
  
```

This command invokes the SyncSort utility to sort records, using a 1-byte integer field `age`. The `/fields` argument specifies the name, position, data type, and length of a field in the command's input. The `/statistics` option configures SyncSort to output statistical information about the sort operation.

In the `my_grep` example (in the section “Example: Operator Using Standard Input and Output” on page 13-9), the `grep` command takes its input from standard input and writes its output to standard output. In this example, SyncSort accesses its input and output via file names specified by command-line arguments. SyncSort takes its input from the file specified by the `/infile` argument and writes its output to a file specified by the `/outfile` argument.

Therefore, the `my_sort` UNIX command operator delivers input records to SyncSort using a file name. Orchestrate does not write the data to a disk file, but instead creates a UNIX named pipe that the command reads in the same way that it reads a file. The `my_sort` operator similarly takes the SyncSort output from a named pipe.

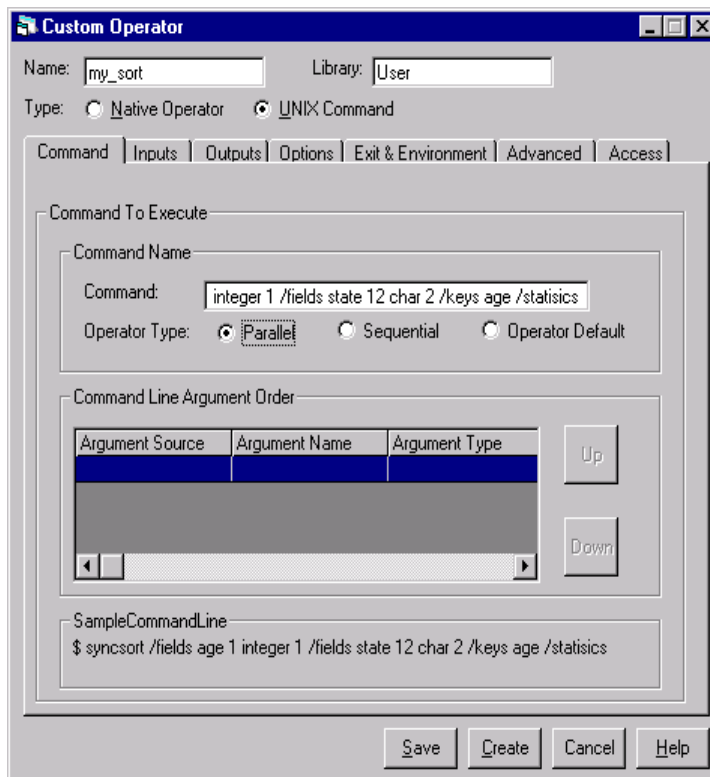
The name of the pipe is defined at run time. When you define this operator, Orchestrate creates a temporary variable for the pipe name, which it replaces at run time.

To create the UNIX command operator `my_sort`, perform the following steps:

1. Choose **Custom -> Define Custom Operator** from the Orchestrate menu.
2. Click the **UNIX Command** button to create a UNIX command operator.
3. Specify the **Name** of the operator: `my_sort`
4. Specify the **Library** name for the operator: `User`.
5. Specify the **Command**:

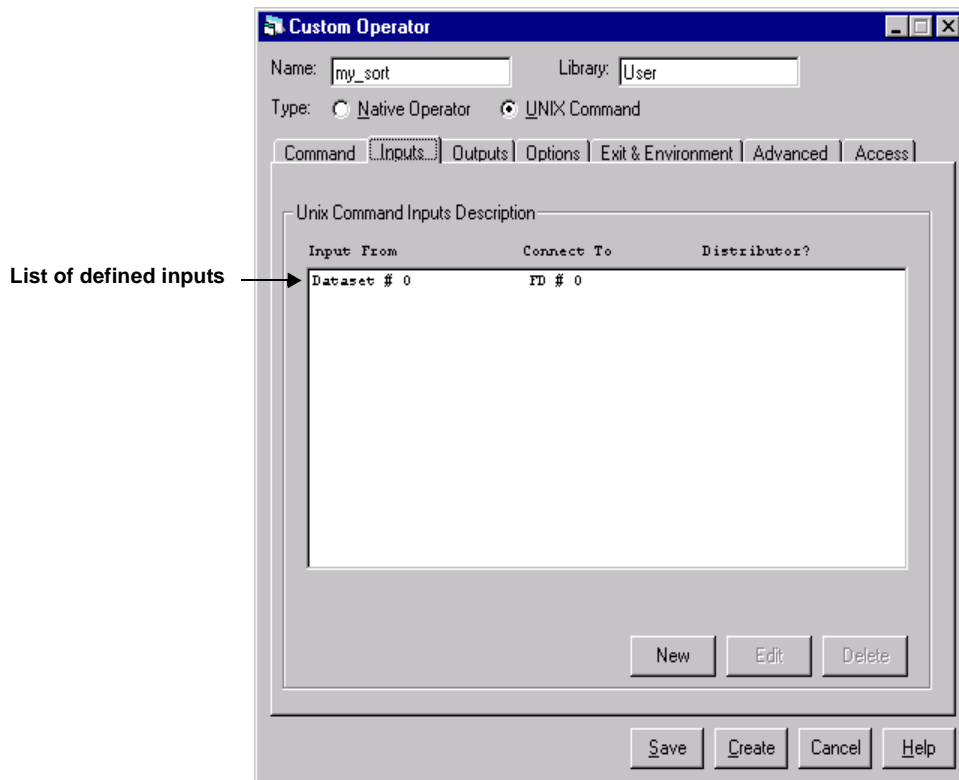
```
syncsort /fields age 1 integer 1 /fields state 12 char 2
/keys age /statistics
```

Note that you omit any reference to the input or output files in the command string.



6. Specify the **Operator Type** as **Parallel**.

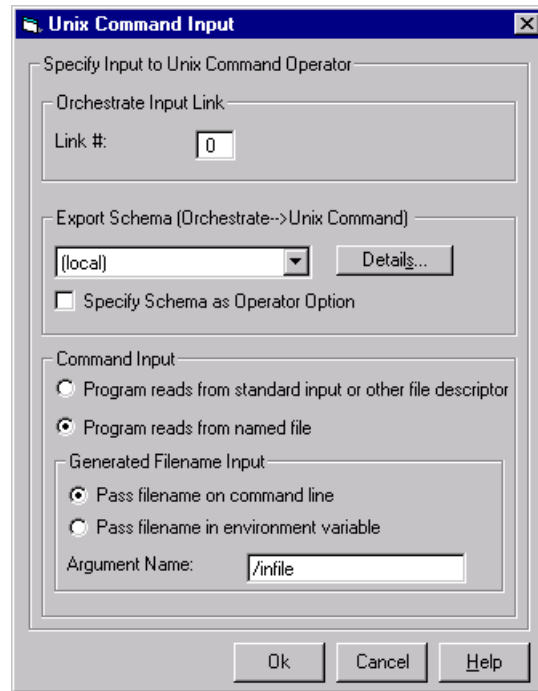
7. Choose the **Input** tab to configure the input data set. Shown below is the **Input** dialog box:



By default, the first input data set is defined to pass the input stream to the command through standard input. You must modify this input data set to use a file name, as follows.

8. Select **Data set # 0**.

- Press the **Edit** button to open the **UNIX Command Input** dialog box, shown below, to configure the input:



- Specify the **Orchestrate Input Link**: 0

The first input data set is link 0, the second is link 1, and so forth.

- Do not modify the **Export Schema**, as this example uses the default export schema.
- Under **Command Input**, choose how the source data is delivered to the UNIX command. This setting defines how the data from the input data set is delivered to the UNIX command.

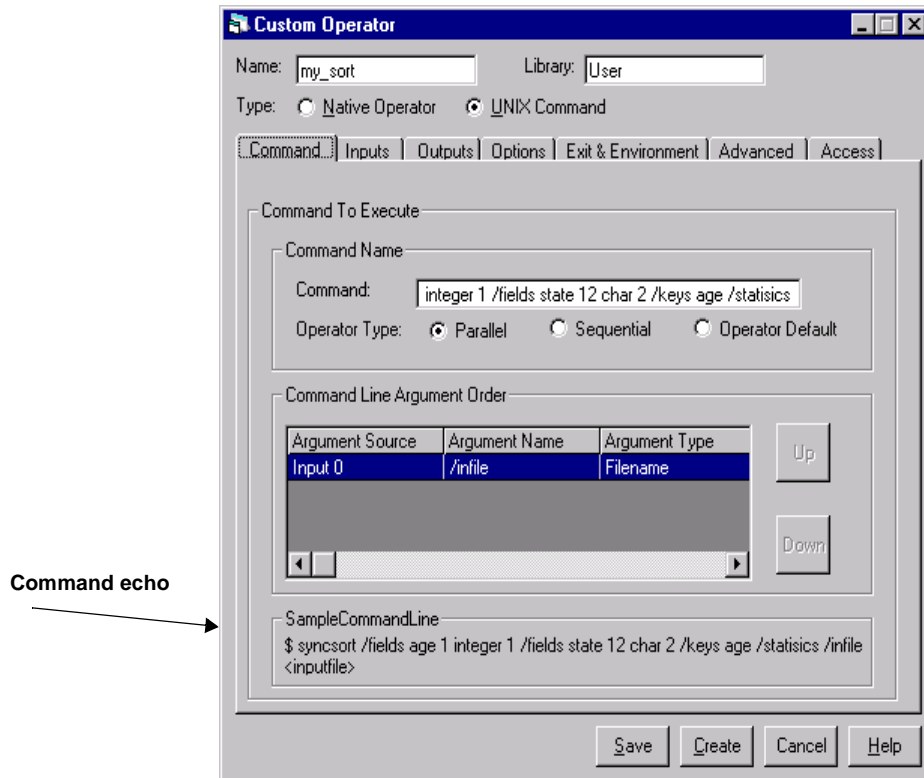
Click **Program reads from a named file** to indicate that the input data set is connected to the command using a file name.

- Under **Generated Filename Input** select **Pass filename on command line**.

- For **Argument Name**, specify `/infile`, the name of the SyncSort command-line argument used to specify the input file.

- Click **OK** to save the settings.

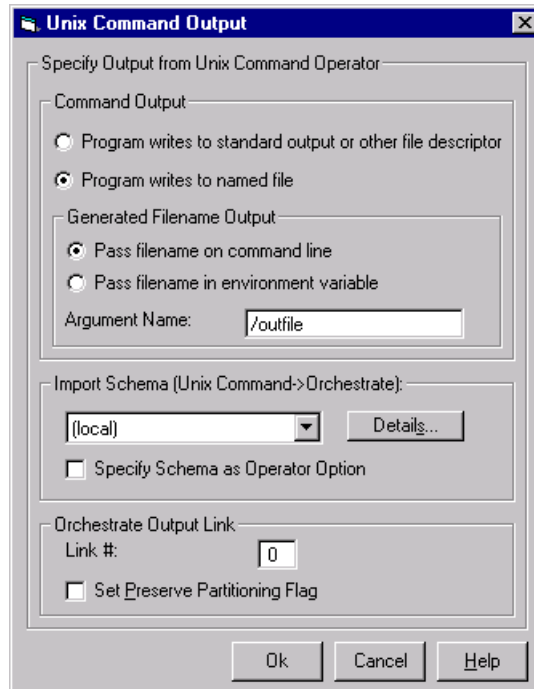
16. Click the **Command** tab. The **Sample Command Line** area of the dialog box displays the command and the new arguments for the input data set: `/infile <inputfile>`.



`<inputfile>` is a placeholder for the name of a UNIX pipe that connects the input data set to the UNIX command. At run time, Orchestrate replaces `<inputfile>` with the actual pipe name. UNIX commands, such as SyncSort, do not differentiate between reading data from a named pipe from reading data from an actual file.

17. Choose the **Outputs** tab to configure the operator's output.

18. From the **Output to** area of the **Outputs** tab, select the default output, output 0, and then press **Edit**. This opens the following dialog box:



By default, the first output data set is defined to read the output stream of the command over standard output. You must modify this output data set to use a file name.

19. Click **Program writes to named file**.
20. For **Argument Name**, enter `/outfile`.
21. Click **Set Preserve Partitioning Flag**.

When set, the preserve-partitioning flag indicates that a data set must not be repartitioned. By default, the flag is clear, to indicate that repartitioning is permissible. For details on the preserve-partitioning flag, see the section “The Preserve-Partitioning Flag” on page 8-11.

22. Click **OK**.
23. Click the **Command** tab to view the **Sample Command Line**.

The command `/outfile <outputfile>` appears on the sample UNIX command line.

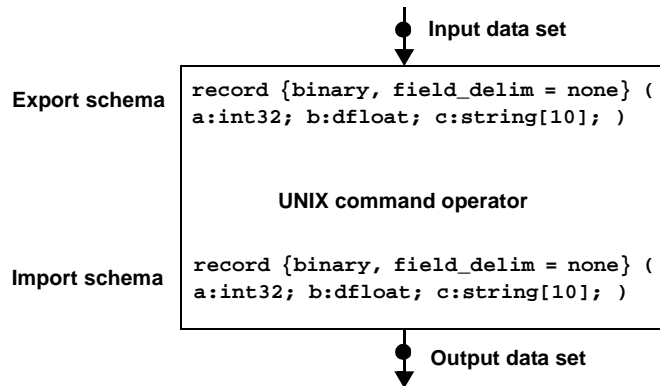
`<outputfile>` is a placeholder for the name of a UNIX pipe that connects the output of the UNIX command to the output data set. At run time, Orchestrate replaces `<outputfile>` with the actual pipe name. UNIX commands, such as SyncSort, do not differentiate between writing data to a named pipe and writing data to an actual file.

24. Click **Create** to create the operator.

## Example: Specifying Input and Output Record Schemas

You can as define a UNIX command operator to handle input and output data of types other than ASCII strings delimited by spaces. For example, you may want to create an operator that runs a UNIX command that processes binary data. To configure your command operator to handle non-default data types and formats, you define schema for the import and/or output performed by the command operator.

For example, the following figure shows a UNIX command operator with a defined export and import schema:

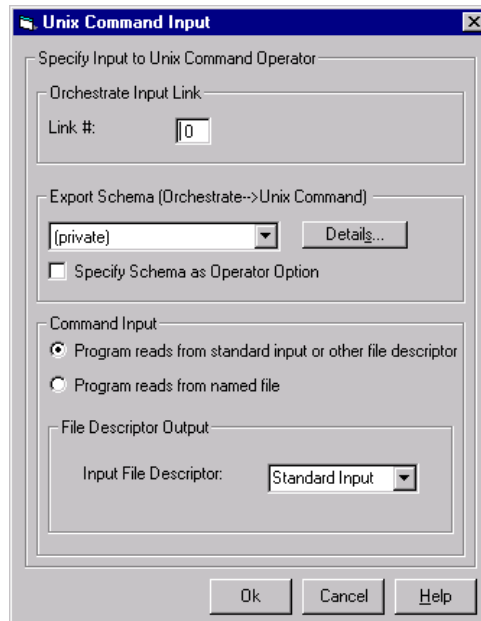


The UNIX command in this example requires input data in binary format with no field delimiters, as defined in the export schema. In addition, the output data set requires a non-default data format, as defined in the import schema. The following steps describe how to use Visual Orchestrate to define the input and output schemas for this sample UNIX command operator.

To set the record schemas of the input and output, do the following:

1. Choose the **Input** tab of the **Custom Operator** dialog box to configure the operator input.  
The first time you select the **Input** tab, you see input **Data set # 0**, the default input data set, already defined.
2. Select **Data set # 0**.

3. Press the **Edit** button to edit **Data set # 0**. This opens the **UNIX Command Input** dialog box, shown below:

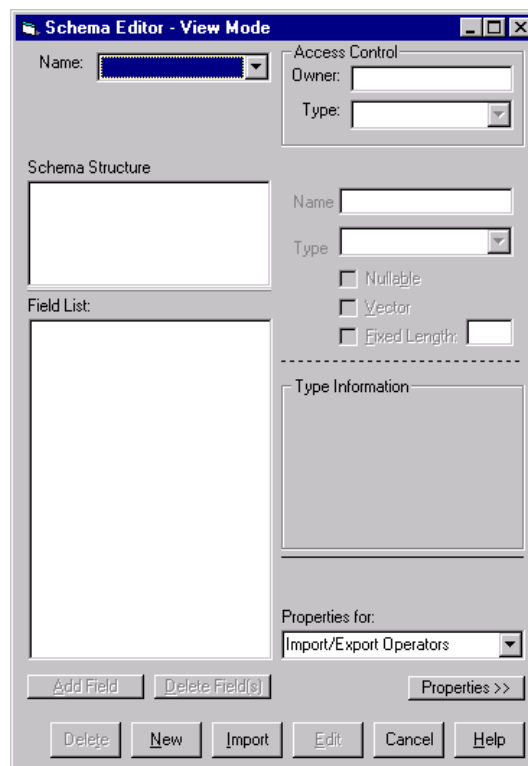


The **UNIX Command Input** dialog box is used to configure input for a Unix command operator. It contains the following sections:

- Specify Input to Unix Command Operator**
  - Orchestrate Input Link**: A text box labeled "Link #:" with the value "0".
- Export Schema (Orchestrate->Unix Command)**
  - A dropdown menu showing "(private)".
  - A "Details..." button.
  - An unchecked checkbox labeled "Specify Schema as Operator Option".
- Command Input**
  - Two radio buttons: "Program reads from standard input or other file descriptor" (selected) and "Program reads from named file".
- File Descriptor Output**
  - An "Input File Descriptor:" label and a dropdown menu showing "Standard Input".

Buttons at the bottom: Ok, Cancel, Help.

4. Press the **Details** button to define the record schema for input data set 0. The **Schema Editor** window opens, as shown below:



The **Schema Editor - View Mode** window is used to define the record schema for input data set 0. It contains the following sections:

- Name:** A dropdown menu.
- Access Control**
  - Owner:** A text box.
  - Type:** A dropdown menu.
- Schema Structure**: A large empty text area.
- Field List**: A large empty text area.
- Name**: A text box.
- Type**: A dropdown menu.
- Nullable**: An unchecked checkbox.
- Vector**: An unchecked checkbox.
- Fixed Length:** A text box.
- Type Information**: A large empty text area.
- Properties for:** A dropdown menu showing "Import/Export Operators".

Buttons at the bottom: Add Field, Delete Field(s), Properties >>, Delete, New, Import, Edit, Cancel, Help.



5. Press **New** to define a new record schema.
6. Use the **Schema Editor** to define the input record schema shown in the figure above.
7. Press **Save** to save the record schema.
8. In the **UNIX Command Input** dialog box, choose how the records of the input data set are delivered to the UNIX command. Choose either:

**Program reads from standard input or other file descriptor**

The UNIX command reads its input from standard input, or from a file descriptor that you specify.

**Program reads from named file**

In this case, you have two options:

- **Pass filename on command line**

Specify the **Argument Name**. Orchestrate adds the argument name and a file name to the UNIX command in the form:

```
/arg_name <inputfile>
```

where *arg\_name* is the value specified for **Argument Name** and *<inputfile>* is a temporary variable. At run time, Orchestrate replaces *<inputfile>* with a UNIX pipe name. UNIX commands do not differentiate between reading data over a named pipe and reading data from an actual file.

- **Pass filename in environment variable**

Specify the **Variable Name**. Orchestrate creates the environment variable at run time and initializes it to a UNIX pipe name. UNIX commands do not differentiate between reading data over a named pipe and reading data from an actual file.

See the section “Example: Passing File Names Using Environment Variables” on page 13-26 for an example using environment variables.

9. Click the **Output** tab to define the export schema of the output data set.  
The first time you select the **Output** tab, you see input **Data set # 0** already defined, as the default output data set.
10. Select **Data set # 0**.
11. Click the **Edit** button to edit **Data set # 0**.
12. Click the **Details** button to define the record schema. The **Schema Editor** window opens.
13. Use the **Schema Editor** to define the output record schema shown above.
14. Specify how the records of the input data set are delivered to the UNIX command. Choose one of these two:

**Program writes to standard output or other file descriptor**

The UNIX command writes its output to standard output, or to a file descriptor that you specify.

**Program writes to named files**

In this case, you have two options:

- **Pass filename on command line**

Specify the **Argument Name**. Orchestrate adds the argument name and a file name to the UNIX command in the form:

```
/arg_name <outputfile>
```

where *arg\_name* is the value specified for **Argument Name** and *<outputfile>* is a temporary variable. At run time, Orchestrate replaces *<outputfile>* with a UNIX pipe name. UNIX commands do not differentiate between writing data over a named pipe and writing data to an actual file.

- **Pass filename in environment variable**

Specify the **Variable Name**. Orchestrate creates the environment variable at run time and initializes it to a UNIX pipe name. UNIX commands do not differentiate between writing data over a named pipe and writing data to an actual file.

For a description of passing environment variables to a UNIX command, see the section “Example: Passing File Names Using Environment Variables” on page 13-26.

## Passing Arguments to and Configuring UNIX Commands

This section describes how to define your UNIX command operator to call the UNIX command with a script for more flexibility in passing arguments, to define user-settable options to pass to the UNIX command, to use environment variables to configure a UNIX command, and to handle command input and output files with configuration and other information. This section covers the following topics:

- “Using a Shell Script to Call the UNIX Command” on page 13-22
- “Handling Message and Information Output Files” on page 13-24
- “Handling Configuration and Parameter Input Files” on page 13-25
- “Using Environment Variables to Configure UNIX Commands” on page 13-26
- “Example: Passing File Names Using Environment Variables” on page 13-26
- “Example: Defining User-Settable Options for a UNIX Command” on page 13-28

### Using a Shell Script to Call the UNIX Command

You can define the operator so that it calls a UNIX shell script instead of directly calling the UNIX command or application. Using a script this way is useful when your operator must perform processing on options passed to the operator, before it can pass those options as arguments to the UNIX command.

Orchestrate calls the shell script, passing it the dynamic options set by the operator user. The shell script parses the options and configures the UNIX command accordingly. While not mandatory, using a shell script is a convenient way to handle parameter parsing in Orchestrate.

You can also use a shell script to pass configuration information to a particular instance of an operator run in parallel. For information, see the sections “Handling Message and Information Output Files” on page 13-24.

For example, suppose you define a UNIX command operator and specify the following command under the **Command** tab:

```
my_shell_script -a -c -e /data/my_data_file
```

In your shell script, you can reference the arguments with shell variables of the form  $\$1 \dots \$n$ . The table below shows the values of the shell variables for the example above, as well as other shell variables you can use in your script.

Shell Variable	Argument
$\$0$	Wrapper file name Example: <code>my_shell_script</code>
$\$1$	First argument Example: <code>-a</code>
$\$2$	Second argument Example: <code>-c</code>
$\$3$	Third argument Example: <code>-e</code>
$\#\#$	Number of arguments passed to the script Example: <code>4</code>
$\$*$	All arguments as a single string

For example, you could code your shell script so that when a user specifies `-e` on an SMP, the script appends `APT_PARTITION_NUMBER` to the file name `/data/my_dataFile`, thus generating a unique file name from each processing node.

## Using the Here-Doc Idiom in Shell Scripts

UNIX command operators that take user-specified options often call a shell script to process those options before invoking the UNIX command. For commands that read configuration information from a parameter file, the shell script can write any parameter values calculated by the shell script to a file, before calling the UNIX command.

One technique that you can use in a shell script to write information to a file is the *here-document* idiom. Shown below is an excerpt from a shell script using this idiom:

```
cat > file_name <<EOF
.
.
.
EOF
```

In this example, `cat` writes everything after `<<EOF` to the file named `file_name`, until it encounters an `EOF` on a line by itself. This procedure lets you calculate values from user-supplied options in the shell script and write those values to the parameter file.

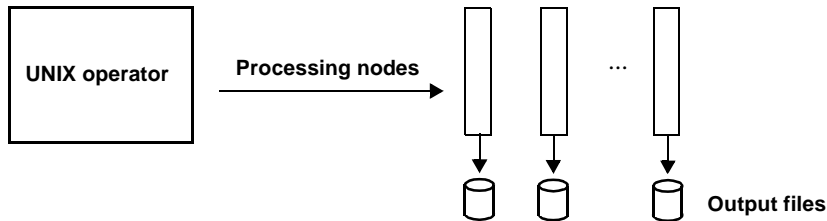
## Handling Message and Information Output Files

UNIX commands can create output files containing messages (information and error log files) or output summaries. For operator outputs that do not contain data, the command can write directly to the files without using an Orchestrate data set. Suppose, for example, that you want to run a UNIX command with the form:

```
unix_command options -e error_file
```

This command creates a file containing any error messages that the command generates.

When executing this command in parallel with a UNIX command operator, you can let the command generate a file on each processing node, as shown in the following figure:



**On an MPP**, each processing node has its own disk storage. The UNIX command writes its output files to the local disk storage on the processing node. Upon completion of the Orchestrate application, you can examine the output files.

**On an SMP**, all processing nodes frequently share disk drives. You must assign unique names to the output files, so that a file generated by one processing node does not overwrite a file generated by another processing node.

You can define the UNIX operator to call a shell script to execute the UNIX command. You code the shell script to create a unique file name for each error file. For example, the following shell script executes `unix_command` on an SMP:

```
#!/bin/ksh
my_command options -e error_file.$APT_PARTITION_NUMBER
```

`APT_PARTITION_NUMBER` is an environment variable, set by Orchestrate, that contains the partition number of the operator on each processing node. If an operator has three partitions (runs on three processing nodes), Orchestrate sets `APT_PARTITION_NUMBER` to 0 on the first processing node, 1 on the second, and 2 on the third processing node. Therefore, this shell script would create three output files: `error_file.0`, `error_file.1`, and `error_file.2`.

By using `APT_PARTITION_NUMBER`, you guarantee that each instance of the UNIX command operator on an SMP creates an error file with a unique name. When the operator runs again, it will overwrite all the error files.

Note that you must create a shell script to run the command shown above. You cannot use the **Command** tab in the **Custom Operator** dialog box to enter and run the command, because Orchestrate evaluates the command in the **Command** tab at the time you invoke your application. At invocation time, `APT_PARTITION_NUMBER` evaluates to the number of the processing node that invokes the application; therefore, each processing node executing the command would use the same value for `APT_PARTITION_NUMBER`.

A shell script, by contrast, is executed only when the operator runs on each processing node, so that `APT_PARTITION_NUMBER` evaluates to the number of the processing node executing the shell script.

See the section “Using a Shell Script to Call the UNIX Command” on page 13-22 for more information.

## Handling Configuration and Parameter Input Files

Many UNIX commands and applications take input files that contain configuration or parameter information. You then modify the input file instead of editing the operator to change the operator's action. This section describes how to use files as input to the UNIX command called by your operator.

For example, the following command line, specified in the **Command** tab of the **Custom Operator** dialog box, uses a parameter file:

```
unix_command options -p p_file
```

In this example, the command reads the parameter file `p_file` to determine the command's action.

**On an MPP**, each instance of a parallel operator executes on a different processing node, with its own disk storage. You must make sure that there is a copy of the parameter file in the same directory on each processing node executing the operator.

**On an SMP**, all processing nodes, corresponding to the CPUs in the system, frequently share disk storage. You need only one copy of the file, as all instances of the operator on the SMP will access the same file.

You can also use the environment variable `APT_PARTITION_NUMBER` and a shell script to execute the UNIX command, as described in the section “Handling Command Exit Codes” on page 13-35. Using a shell script enables you create a different configuration file for each processing node. For example, you could invoke the command in the following shell script:

```
#!/bin/ksh
unix_command options -p p_file.$APT_PARTITION_NUMBER
```

This shell script causes the command to look for a parameter file named `p_file.0` on the first processing node, `p_file.1` on the second processing node, and so forth.

## Using Environment Variables to Configure UNIX Commands

One common way to configure a UNIX command is to use environment variables. Environment variables can define the location of system resources or hold configuration options for a command. For a UNIX command to access an environment variable, the variable must be set in the UNIX shell that invokes the command.

To run a command from a UNIX command operator, Orchestrate first creates a UNIX shell using the login name of the user invoking the application. The shell contains environment variables defined by the user's execution environment.

In addition, in defining your UNIX command operator, you can create environment variables that hold the following:

- File names used as inputs to or outputs from a command.  
You create these environment variables when you configure the input and output data set connections used by the operator. The example below describes how to pass file names using environment variables.
- Configuration settings for the command.

The **Exit & Environment** tab area of the **Custom Operator** dialog box lets you define environment variables that Orchestrate will set before it invokes your UNIX command. In this way, you can pass information to the command.

## Example: Passing File Names Using Environment Variables

Many UNIX commands access an environment variables to determine an input or output file name. In addition to letting you handle command-line arguments to a UNIX command in your operator, Orchestrate lets you define your UNIX command operator to handle environment variables.

Suppose, for example, that you want to run the following command using a UNIX command operator:

```
filter args
```

This command filters its input based on the arguments passed to it. The command also obtains the name of an input file by the value of the environment variable `FILTER_INPUT`, and the name of an output file from the environment variable `FILTER_OUTPUT`.

Orchestrate creates the environment variable at run time and initializes it to a UNIX pipe name. UNIX commands do not differentiate between reading data from a named pipe and reading data from an actual file.

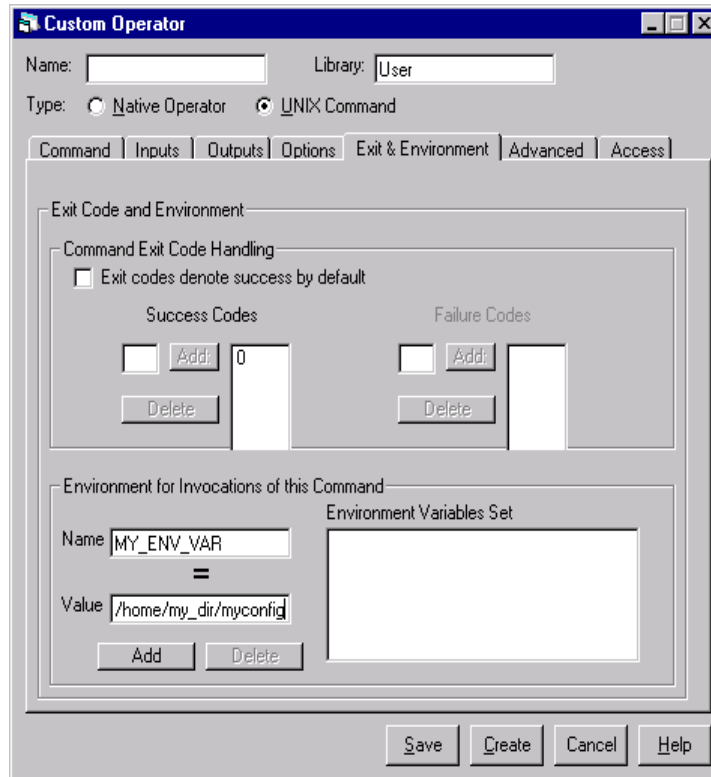
To define the environment variables for this example, do the following:

1. Choose the **Input** tab of the **Custom Operator** dialog box to configure the operator input.
2. Select **Data set # 0**.
3. Press the **Edit** button to edit **Data set # 0**. This opens the **UNIX Command Input** dialog box.
4. In the **UNIX Command Input** dialog box, choose how the records of the input data set are delivered to the UNIX command. Select the following:  
**Program reads from named file**
5. Choose how the file name is passed to the UNIX command. For this example, select the following:  
**Pass filename in environment variable**
6. Specify the **Variable Name**: `FILTER_INPUT`.  
Orchestrate creates the environment variable at run time and initializes it to a UNIX pipe name.
7. Click the **Output** tab to define the export schema of the output data set.
8. Select **Data set # 0**.
9. Click the **Edit** button to edit **Data set # 0**.
10. Choose how the record of the input data set are delivered to the UNIX command. Choose:  
**Program writes to named files**
11. Specify how the file name is passed to the UNIX command. For this example, select the following:  
**Pass filename in environment variable**
12. Specify the **Variable Name**: `FILTER_OUTPUT`.  
Orchestrate creates the environment variable at run time and initializes it to a UNIX pipe name.

## Example: Defining an Environment Variable for a UNIX Command

In this example, a UNIX operator executes a UNIX command that uses the environment variable `MY_ENV_VAR` to determine the location of its parameter file. To configure the UNIX command operator to set that environment variable, perform the following steps:

1. In the **Custom Operator** dialog box, select the **Exit & Environment** tab, shown below:



2. Under **Environment for Invocations of this Command**, enter the following information:  
**Name:** MY\_ENV\_VAR  
**Value:** /home/my\_dir/myconfig
3. Click the **Add** button.

This environment variable will be set in the UNIX shell when Orchestrate executes the operator.

## Example: Defining User-Settable Options for a UNIX Command

The `my_sort` example (in the section “Example: Operator Using Standard Input and Output” on page 13-9) describes a UNIX command operator that always runs the `SyncSort` command with a predefined command line. The user interface to this command operator is *static*, always sorting the input file using the same sorting key and always generating statistics about the sort.

Orchestrate also lets you define UNIX command operators with a *dynamic* interface, which allows the operator user to specify options when using the operator in a step. Suppose, that the operator user needs to perform multiple sorting operations, in which the sorts differ by the key fields used to perform the sort. Rather than creating a different, static UNIX command operator for each type of



sort operation, you can create a single, dynamic sort operator that allows the user to pass configuration options to the operator.

To create a dynamic command operator, you define at least part of the UNIX command as dynamic. For example, in the **Command** tab you could specify the following as the static portion of the sample UNIX command above:

```
syncsort /fields age 1 integer 1 /fields state 12 char 2
```

The static portion of this command defines the data format of the sorted data, but does not specify the sorting keys or statistics option. The dynamic portion of the command lets the user optionally specify the `/key` and `/statistics` options. The operator user can specify one, both, or neither option to the operator each time the operator is invoked in a step.

This example describes how to create a command operator that lets the operator user set two options. It also describes how the operator user sets the options, and the results of setting the options.

## Adding Options to an Operator

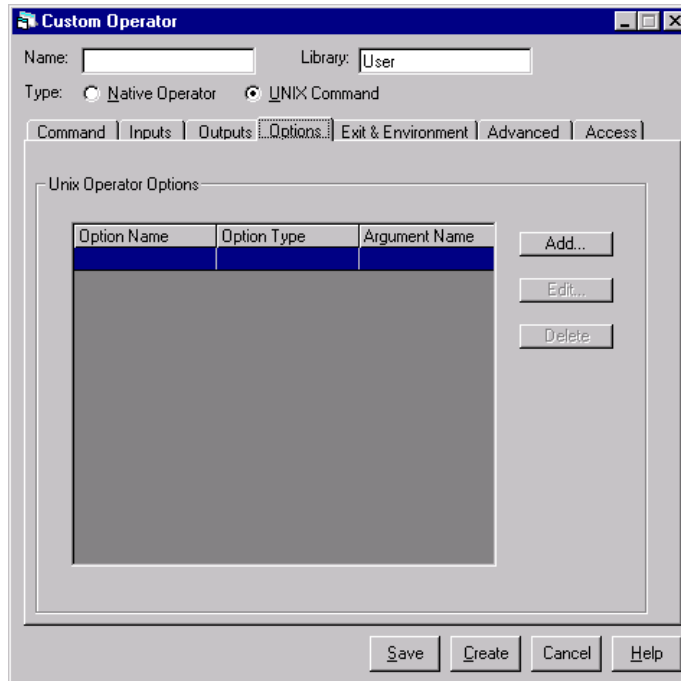
To create a sort operator that takes user-settable options, perform the following steps:

1. Create the `my_sort` operator as described in the section “Example: Operator Using Files for Input and Output” on page 13-13. However, instead of the command line used in that operator definition, enter the following command line in the **Command** tab:

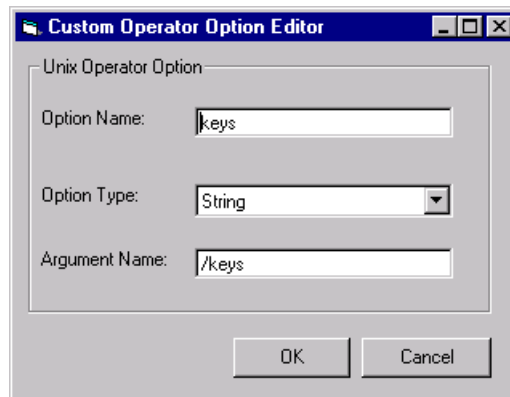
```
syncsort /fields age 1 integer 1 /fields state 12 char 2
```

Note that this command line contains no specification for `/keys` or `/statistics`.

- To create user-settable options for the operator, first choose the **Options** tab of the **Custom Operator** dialog box:



- Press the **Add** button to open the **Custom Operator Option Editor** dialog box:



- Specify the **Option Name**.  
This is the name of the option that the operator user selects in the **Option Editor** dialog box.
- Select the **Option Type**.  
This is the data type of any value specified to the option. Available data types are **Boolean**, **Integer**, **Float**, **String**, and **Pathname**. For details on the operator user interface to options of each type, see the section “How the Operator User Sets Options” on page 13-31.
- Set the **Option Name** to `keys`.

Note that Orchestrate sets the default **Argument Name** to `-keys`.

7. Set the **Option Type** to `String`.

You let the user specify a string as the option's value, because the `/keys` argument takes a string specifying one or more sorting keys, separated by commas.

8. Set the **Argument Name** to `/keys`.
9. Click **OK** to close the **Custom Operator Option Editor**.
10. Click **Add** to add the `stats` option.
11. Set the **Option Name** to `stats`.
12. Set the **Option Type** to `Boolean`.

A flag argument means that the option takes no additional value from the user.

13. Set the **Argument Name** to `/statistics`.
14. Click **OK**.
15. Click the **Command** tab to view the command. Note that the **Sample Command Line** shows both arguments. These arguments will be included in the command only if the user specifies them when using the operator in a step.

Also note that the options appear in the **Command Line Argument Order** list. You can use the **Up** and **Down** buttons to change the order of the arguments to the UNIX command, as reflected in the Sample Command Line area.

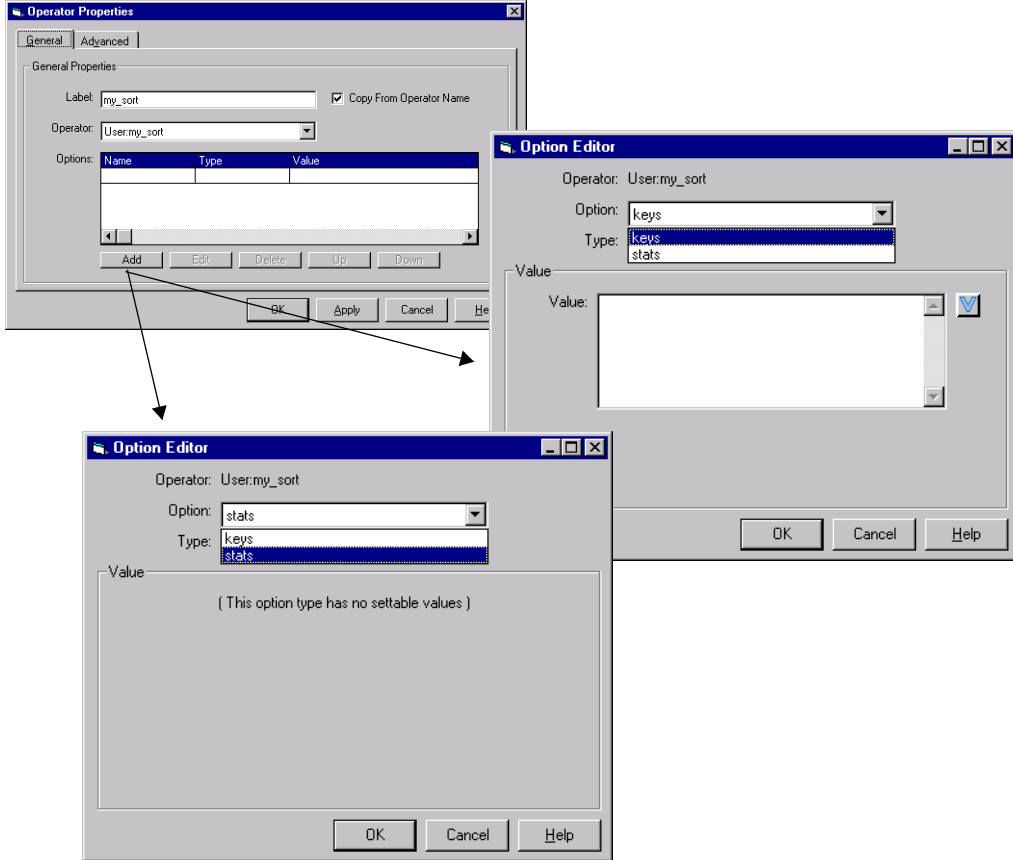
16. Press **Create** to create the operator.

An operator user inserting this operator into a step can now specify the `key` option to set the sorting keys and the `stats` option to enable statistic generation.

## How the Operator User Sets Options

To access the user-settable options on this sample command operator, the operator user inserts the operator into the application. Then, the operator user double clicks the operator to open the **Operator Properties** dialog box. The user presses the **Add** button to open the **Option Editor** dialog box. From the list in that dialog box, the user selects and sets one or more operator options.

Shown below are the **Operator Properties** dialog box, and the **Option Editor** dialog box after each option (`keys` and `stats`) has been selected:



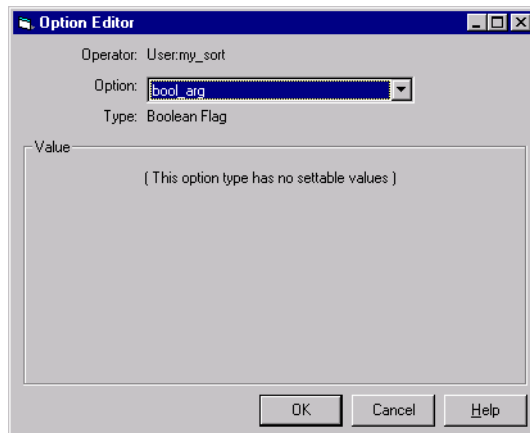
For the `key` option in this example, the operator can specify one or more sorting keys, separated by commas. For the `stats` option, simply specifying `stats` enables the option.

Below are detailed descriptions of the operator user interface for options of each supported type.

### Setting Boolean Options

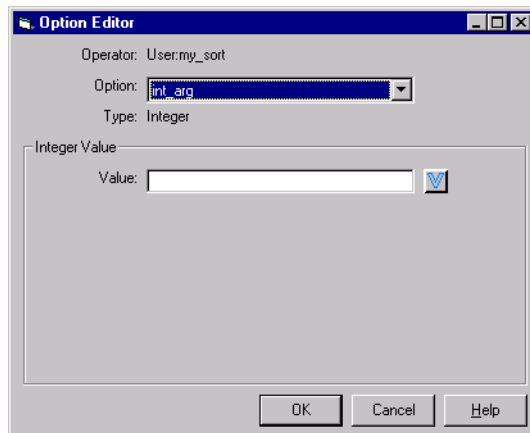
Options with the type **Boolean** do not take a value. Specifying the option name is all that is required. For example, the SyncSort `/statistics` option does not require a value; simply specifying the option enables the generation of sorting statistics.

An option using **Boolean** has the following dialog box in the **Option Editor**:



### Setting Integer Options

The option takes a single integer value. **Integer** options have the following dialog box in the **Option Editor**:

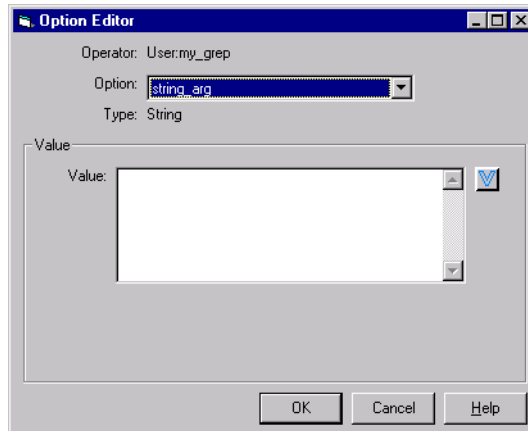


### Setting Float Options

**Float** options have the same dialog box as **Integer**, described above.

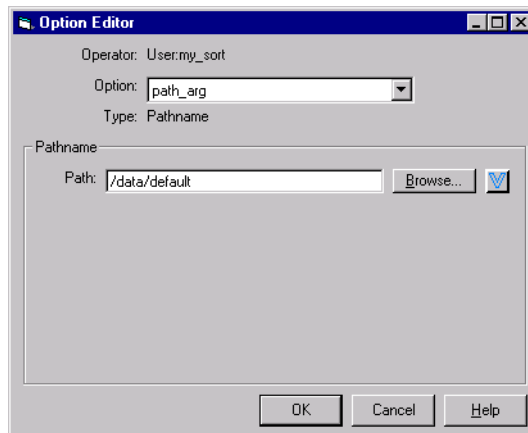
### Setting String Options

An option using **String** has the following dialog box in the **Option Editor**:



### Setting Pathname Options

**Pathname** arguments have the following dialog box in the **Option Editor**:



The default value of **Path** is the current working directory (set in the **Paths** tab of the **Program Properties** dialog box, as described in the section “Setting Program Directory Paths” on page 2-13). Pressing **Browse** opens the Visual Orchestrate file browser to allow the operator user to set the **Paths** value.

### Results of Setting Options

If the operator user sets no option, the UNIX command operator executes only the static portion of the command line:

```
syncsort /fields age 1 integer 1 /fields state 12 char 2
/infile source /outfile dest
```

However, the operator user can also set the **keys** option to the value **age**. In this case, the operator executes the command line:

```
syncsort /fields age 1 integer 1 /fields state 12 char 2
/keys age
/infile source /outfile dest
```

A user specifying the `stats` option and the `key` option with a value of `age` would invoke the command line:

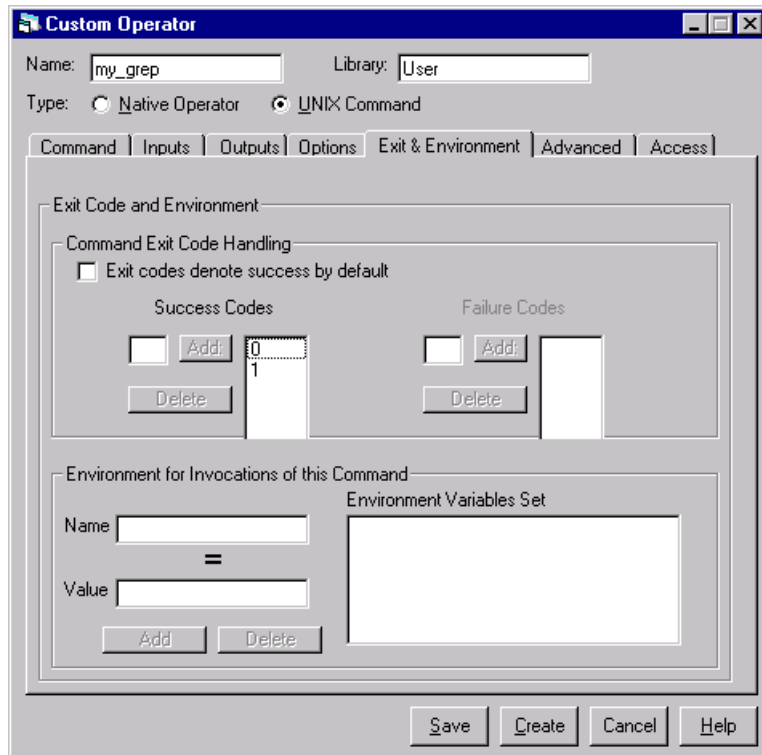
```
syncsort /fields age 1 integer 1 /fields state 12 char 2
/keys age /statistics
/infile source /outfile dest
```

## Handling Command Exit Codes

All UNIX commands return an exit code indicating the success, failure, or other result of the command. Orchestrate interprets the exit code when it executes a UNIX command operator, to determine whether the command has executed successfully. You can control how Orchestrate interprets exit codes.

**Note:** If, for any reason, the UNIX command returns an exit code that Orchestrate interprets as a failure, the operator fails. When any operator in a step fails, Orchestrate terminates the entire step.

When you create a UNIX command operator, you use the **Exit & Environment** tab in the **Custom Operator** dialog box to define how Orchestrate interprets exit codes. This tab is shown below:



By default, Orchestrate interprets an exit code of 0 as a success and all other exit codes as a failure.

In the **Command Exit Code Handling** area you can set the following options:

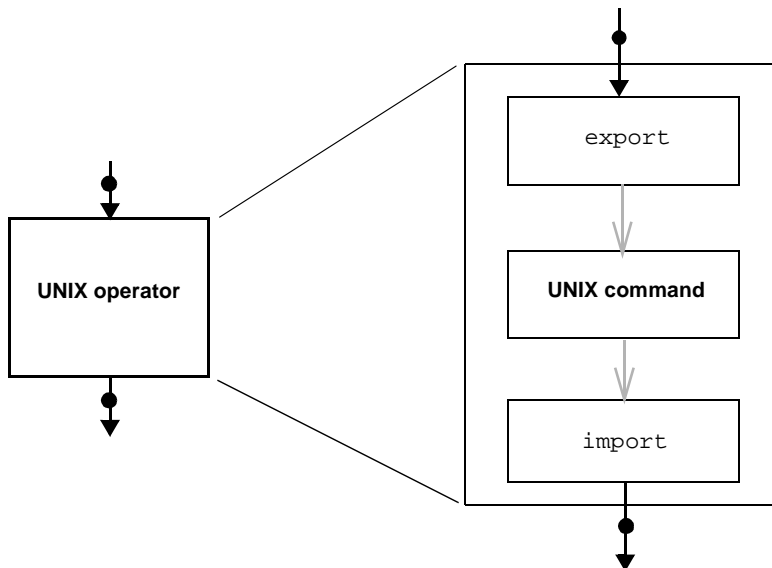
- **Exit codes default successful:** Causes all exit codes to be treated as success, except codes explicitly designated as errors using **Failure Codes**. Ignores any settings defined by **Success Codes**.
- **Success Codes:** Specifies exit codes defining successful execution of the command; all codes that you have not specified as **Success Codes** are considered errors. Configures Orchestrate to ignore any settings defined by **Failure Codes**.

You can specify *either* **Success Codes** or **Exit codes default successful**, but not both. Some common specifications are:

- **Success Codes = 0 and 1:** Exit codes 0 and 1 indicate success; all others are errors.
- **Exit codes default successful** selected: All exit codes indicate success.
- **Exit codes default successful** selected and **Failure Codes = 1:** All exit codes other than 1 indicate success; exit code 1 indicates an error.

## How Orchestrate Optimizes Command Operators

As described in the section “Execution of a UNIX Command Operator” on page 13-5, by default a UNIX command operator consists of the three parts shown on the right-hand side of the following figure:



When executing the operator, Orchestrate creates one UNIX process each for the export, command, and import parts of the operator on each processing node executing the operator. For example, on a four node system, Orchestrate creates 12 UNIX processes for the four instances of the operator.



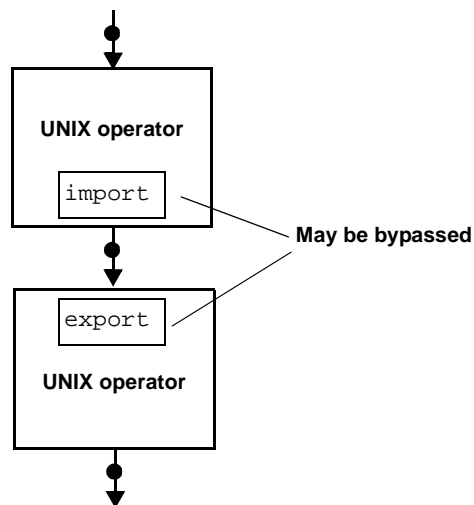
This section describes how Orchestrate can optimize your UNIX command operator. In many of these optimizations, Orchestrate skips either the export or the import process.

This section covers the following topics:

- “Cascading UNIX Command Operators” on page 13-37
- “Using Files as Inputs to UNIX Command Operators” on page 13-38
- “Using FileSets as Command Operator Inputs and Outputs” on page 13-39
- “Using Partial Record Schemas” on page 13-39

## Cascading UNIX Command Operators

In creating an Orchestrate step, you can cascade (connect in succession) UNIX command operators, as shown below:



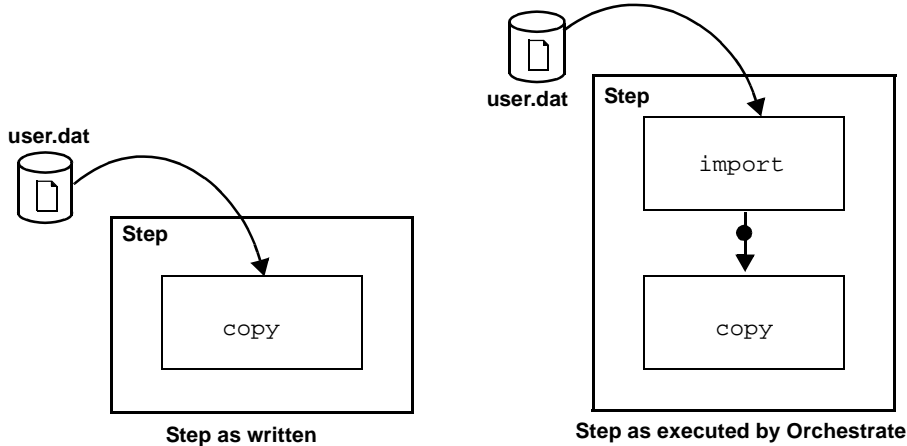
By default, Orchestrate performs an import on the output of the first operator and an export on the input of the second. However, Orchestrate eliminates both the import and the export, if all three of the following conditions are met:

- Both operators have the same execution mode (parallel or sequential).
- Both operators have the same constraints.
- The output port of the first operator and the input port of the second operator use the same record schema.

The last condition is the most important. If the two ports do not have the same record schema, Orchestrate must perform the export and the import operations in order to convert the output of the first operator to the record format required by the UNIX command of the second. For example, if the first operator specifies an ASCII text mode representation of the imported data, but the export of the second operator specifies a binary representation, Orchestrate must perform both the import and the export.

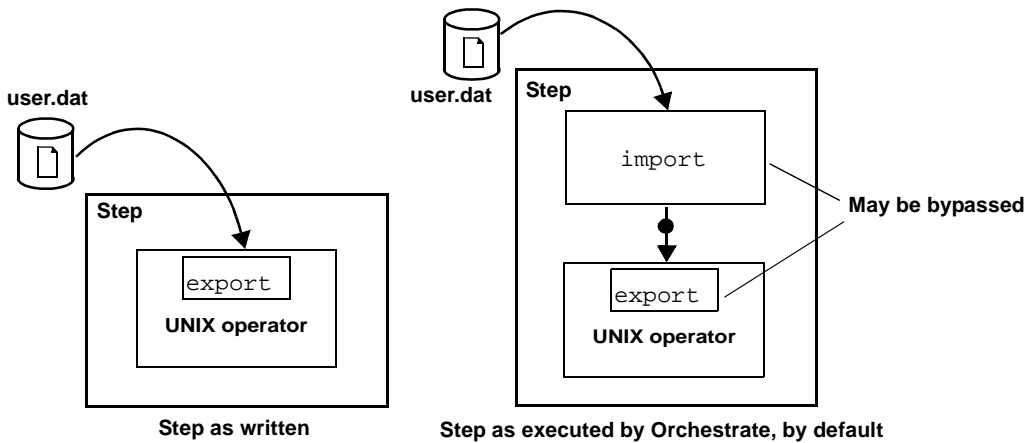
## Using Files as Inputs to UNIX Command Operators

Orchestrate lets you specify a UNIX file instead of a data set, as an input to any operator. Orchestrate automatically performs an import to convert the input file to an Orchestrate data set. Shown below is an example of this import operation:



The left side of this figure shows the step as defined by the user. The right side shows the step as executed by Orchestrate, as it automatically performs the import required to read the UNIX data files.

However, when the operator is a UNIX command operator, the first action of the operator on input is to export the input data set to convert it to the file format required by the UNIX command, as shown below:



In the case shown above, Orchestrate can bypass the import and export, to connect the input file directly to the UNIX command.

If an input file has the same record schema as the operator input interface, then the UNIX command operator directly reads the file. If the operator executes in parallel on an SMP, each instance of the operator directly reads the file, in parallel.

## Using FileSets as Command Operator Inputs and Outputs

The previous section described how Orchestrate optimizes a UNIX command operator when you specify a file as the operator's input or output. You can also specify a *fileset* as an operator's input or output. A fileset is an ASCII text file that contains either a list of UNIX source files for input, or a list of destination files for output. The files referenced by a fileset are UNIX data files, not Orchestrate data sets. The fileset must contain one file name per line and can optionally contain a record schema defining the layout of the data in the data files. All data files referenced by the fileset must have the same layout.

You can use the Orchestrate operator `export` to create a fileset, as described in the chapter on the `export` operator in the *Orchestrate User's Guide: Operators*. Or if you wish, you can create a fileset using a text editor outside the Orchestrate environment.

Orchestrate optimizes the data access of filesets by parallel UNIX command operators in much the same way that it optimizes for individual files. If, on input, the fileset has the same record schema as the input port of the operator, Orchestrate bypasses the `export` operation and connects the fileset directly to the UNIX command. If, on output, the fileset has the same record schema as defined for the output port of the operator, Orchestrate bypasses the `import` and writes the data directly to the fileset.

## Using Partial Record Schemas

When importing data using an Orchestrate partial record schema, you define only the fields of interest in your data. Your records may contain many individual fields — perhaps even hundreds. However, to process the records, your application may access only a few fields to use as sorting keys, partitioning keys, or input fields to Orchestrate operators. You can simplify the import/export procedure by providing only enough schema information to identify the fields required by your application. See the chapters on the `import` and `export` operators in the *Orchestrate User's Guide: Operators* for a complete description of partial record schemas.

Shown below is a sample record schema with field information for only two fields:

**Record after import, as stored in a data set**

	<code>name:string[20]</code>		<code>income:dfloat</code>	
--	------------------------------	--	----------------------------	--

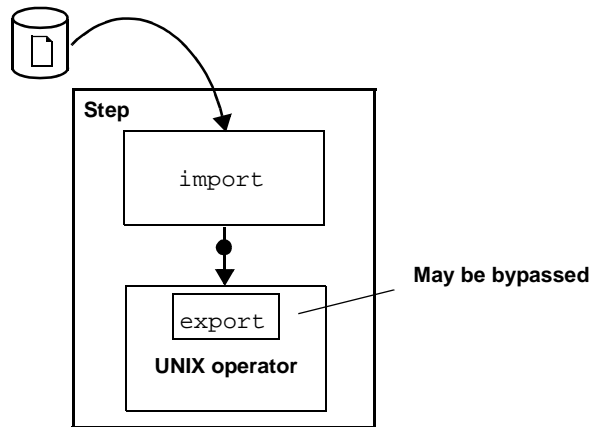
The following record schema defines the two fields of this record:

```
record { intact=rName, record_length=82, record_delim_string='\r\n' }
  ( name: string[20] { position=12, delim=none };
```

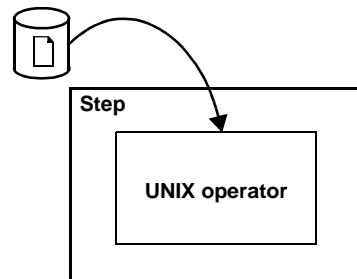
```
income:dfloat { position=40, delim=',', text };
)
```

One advantage of using a partial record schema is that the `import` operator does not perform a conversion on the data. Therefore, the intact portion of the record contains data in its original format.

Shown below is an example application containing an `import` operator followed by a UNIX command operator:



If the data set created by the `import` operator contains a partial record schema, Orchestrate can omit the `export` of the UNIX command operator, because the input data set contains data in the original format. The following figure shows the application, in this case:



# Index

## A

- accessors 4-17
- aggregate data types
  - subrecord 3-6
  - tagged 3-6
- aggregate fields
  - in input data sets 5-7
- applications
  - components of 1-4
  - creating 2-1
  - deploying 2-2, 2-6
  - display-intensive 6-8
  - multiple-step 6-4
  - performance
    - effect of output display 6-8
  - running 2-7
  - single-step 6-3
  - steps in 6-4
  - validating 2-6
- APT\_Collector class 9-3
- APT\_CONFIG\_FILE 1-14
- APT\_Date class 12-21
- APT\_Decimal class 12-23
- APT\_ERROR\_CONFIGURATION 11-4
- APT\_Partitioner class 8-4
- APT\_RawField class 12-25
- APT\_StringField class 12-24
- APT\_Time class 12-21
- APT\_TimeStamp class 12-21
- APT\_WRITE\_DS\_VERSION 4-33
- arcs
  - performance monitor display of 7-4

## B

- branching
  - in steps 6-3

## C

- cluster/MPP systems
  - See* MPP systems

- collecting 9-1
  - defined 9-2
  - method 9-1
    - any 9-3
    - ordered 9-3
    - other 9-3
    - round robin 9-3
    - sorted merge 9-3
  - operators and 9-2
  - preserve-partitioning flag and 9-2
- collection method 9-1
  - any 9-3
    - example of use 9-5
  - ordered 9-3
  - other 9-3
  - round robin 9-3
  - sorted merge 9-3
    - example of use 9-6
- collectors
  - defined 9-1
  - See also* collecting
- components of Orchestrate application 1-4
- configuration file
  - disk pools and 10-4
  - node allocation and 1-3, 10-2
  - node definitions and 10-5
  - node pools and 10-2
- Constraint Editor dialog box 10-6, 10-8
- constraints
  - applying 1-10, 10-1
  - combining node and resource 10-8
  - data sets and 10-9
  - logical nodes and 10-2, 10-4, 10-5
  - MPP systems 10-2
  - node definitions and 10-5
  - node maps and 10-1
  - node pools and 10-1
  - operators and 10-1
  - Orchestrate configuration file and 10-5
  - resource pools and 10-1

- SMP systems 10-4
- creating operators 12-1
- Custom Operator dialog box 12-5, 13-10
- custom operators
  - action of 12-3
  - arguments to 12-2
  - C++ compiler for 12-4
  - characteristics 12-2
  - coding
    - data types and 12-20
    - examples 12-32, 12-34
    - for multiple inputs 12-32
  - creating 12-7
  - date fields and 12-21
  - decimal fields and
    - example 12-23
  - defined 12-1
  - example 12-15, 12-16, 12-17, 12-21
  - execution mode 12-2
  - execution of 12-8, 12-31
  - flow-control macros 12-27
  - functions
    - [index] 12-26
    - \_clearnull() 12-26
    - \_null() 12-26
    - \_setnull() 12-26
    - fieldname\_clearnull() 12-26
    - fieldname\_null() 12-26
    - fieldname\_setnull() 12-26
    - setVectorLength() 12-26
    - vectorLength() 12-26
- I/O macros 12-28
  - examples 12-28
- included header files 12-4
- informational macros 12-27
- input and output interfaces
  - defining 12-9
- input data sets and 12-4
- input interface 12-6
- input interface schema 12-8
- input ports
  - indexing 12-9
  - naming 12-9
- inputs
  - auto read 12-9, 12-32, 12-34
  - noauto read 12-34
- interface schema and 12-10, 12-13
  - input 12-10, 12-13
  - nullable fields 12-25
  - output 12-10, 12-13
  - vector fields 12-26
- interface schemas
  - null fields in 12-25
- introduction to 1-10
- macros 12-27
  - discardRecord() 12-28
  - discardTransfer() 12-28
  - doTransfer() 12-29
  - doTransfersFrom() 12-29
  - doTransfersTo() 12-29
  - endLoop() 12-27
  - failStep() 12-27
  - holdRecord() 12-28
  - inputDone() 12-28
  - inputs() 12-27
  - nextLoop() 12-27
  - outputs() 12-27
  - readRecord() 12-28
  - transferAndWriteRecord() 12-29
  - transfers() 12-27
  - writeRecord() 12-28
- nullable fields and 12-25
- numeric fields and 12-21
- option definition 12-7
- options for 12-17
  - data types of 12-19
  - defining 12-20
- Orchestra server administrator and 12-4
- output data sets and 12-4
- output interface 12-6
- output interface schema 12-8
- output ports
  - indexing 12-9
  - naming 12-9
- outputs
  - auto write 12-9
- partitioning method 12-2
- Per-record code 12-6
- Post-loop code 12-7
- Pre-loop code 12-7
- processing loop 12-3
- raw fields and
  - example 12-25
- reject output
  - example 12-30
- saving 12-7

- See also* native operators
- string fields and
  - example 12-24
- time fields and 12-21
- timestamp fields and 12-21
- transfer macros 12-29
  - examples 12-30
- transfers and 12-2, 12-16
- user options for
  - data types of 12-19
- user-settable options for 12-18
- using 12-1
- vector fields and 12-26

## D

data flow 1-5

- See* data-flow models

data flows

- performance monitor display of 7-4

Data Set Properties dialog box 4-8

Data Set Viewer 4-14

- data sets and 4-14

- using 4-14

data sets

- as input to operators 5-10

- as output from operators 5-10

- configuring 4-8

- constraints and 10-9

- syntax 10-9

- using 10-9

- creating

- orchadmin and 4-35

- Data Set Properties dialog box 4-8

- Data Set Viewer and 4-14

- dialog box 4-8

- disk pools and 10-9

- export and 4-7

- file naming 4-37

- flat files 1-8

- import and 4-7

- introduction to 1-5

- Link Properties dialog box and 4-11

- multiple inputs and 4-4

- multiple outputs and 4-4

- operators and 4-3

- data type compatibility 3-8, 5-17

- Orchestra version and 4-33

- output

- record schema 5-8

- parallel representation 4-32

- data files 4-32

- descriptor file 4-32

- segments 4-32

- partitioning 1-2, 1-9, 4-7

- disk pools and 10-10

- performance monitor and 7-4

- persistent

- examples 4-6

- record count 4-16

- viewing 4-14

- record fields

- default value 4-27

- record schema 1-6

- representation of 4-32

- segments and 4-32

- storage format 4-33

- structure of 4-1, 4-32

- viewing 4-14

- virtual

- examples 4-5

data type conversions

- data set fields and 3-8, 5-17

- default 3-9

- examples 3-10

- modify and 3-9

- examples 3-10

- operators and 3-8, 5-17

- record schemas and 3-8

data types

- compatibility 5-17

- conversion

- errors 5-17

- string and decimal fields 5-20

- warnings 5-17

- conversions 5-17

- date 3-2

- decimal 3-4

- floating-point 3-5

- integer 3-6

- introduction to 3-1

- nulls 3-2

- raw 3-6

- string 3-6

- subrecord 3-6

- tagged 3-6

- time 3-6

- timestamp 3-7
  - data-flow diagrams
    - See* data-flow models
  - data-flow models
    - and steps 6-2
    - definition of 1-5
    - directed acyclic graph 6-2
    - partitioning and 8-5
    - UNIX command operators and 13-7
  - date 3-2
    - range 3-2
  - date data type
    - See* data types
      - date
  - date fields 4-20
    - data type 4-20
    - schema definition of 4-20
  - DB2
    - DB2 partitioner 8-4
    - partitioning and 8-4
  - decimal 3-4
    - assignment to
      - strings and 3-5
    - precision
      - range 3-4
    - range 3-5
    - representation size 3-4
    - scale
      - range 3-4
    - sign nibble 3-4
      - values 3-4
    - strings and 3-5
    - valid representation 3-4
    - zero representation 3-4
  - decimal data type
    - See* data types
      - decimal
  - decimal fields 4-21
    - data type 4-21
    - precision limit 4-21
    - range limit 4-21
    - schema definition of 4-21
  - default value
    - record fields 4-27
  - development environment 2-1
  - dialog box
    - Constraint Editor 10-6, 10-8
    - Custom Operator 12-5
    - UNIX Command 13-10
    - Data Set Properties 4-8
    - Data Set Viewer 4-14
    - Input Interface Editor 12-9
    - Link Properties 4-11, 8-14
    - Operator Properties 5-4, 6-14, 10-6
    - Option Editor 5-5
    - Output Interface Editor 12-9
    - Program Editor 10-6
    - Program Properties 2-4
    - Step Properties 6-6
  - directed acyclic graph
    - defined 6-2
  - disk pools 10-1, 10-4
    - assigning data sets to 10-9
    - constraining operators to 10-10
    - data set partitions and 10-10
    - data sets and 10-4
  - disks
    - allocating 10-5
    - constraints and 10-5
    - node definitions and 10-5
- ## E
- enumerated fields 5-13
  - environment variables
    - APT\_CONFIG\_FILE 1-14
    - APT\_PARTITION\_NUMBER 13-24
    - APT\_WRITE\_DS\_VERSION 4-33
    - setting 6-10
  - error handling 5-18
  - error log 11-1
  - error messages
    - default components 11-4
    - display control
      - APT\_ERROR\_CONFIGURATION
        - and 11-4
      - default display 11-4
      - keywords 11-4
    - example 11-4
    - keywords
      - errorIndex 11-2
      - ipaddr 11-3
      - jobid 11-2
      - lengthprefix 11-3
      - message 11-3
      - moduleId 11-2
      - nodename 11-3



- nodeplayer 11-3
- opid 11-3
- severity 11-2
- timestamp 11-2
- vseverity 11-2
- subprocesses and
  - default display 11-3
- errors
  - error log 11-1
  - reporting 11-1
- Excel
  - performance monitor spreadsheet and 7-8
- execution mode
  - See* operators
    - execution mode
- Execution Window
  - message headers and 2-10
  - text wrap and 2-10
- export 4-7

**F**

- field accessors 4-17
- field adapters
  - schema variables and 5-13
- fields
  - and schema variables 5-13
  - enumerated 5-13
  - fixed-length 4-1
  - See* record fields
  - variable-length 4-1
- fixed-length field 4-1
- fixed-length records 4-1
- flat files 1-8
- floating-point data type
  - See* data types
    - floating-point
- floating-point fields
  - schema definition 4-21

**G**

- get\_orchserver\_variable 6-13
- grids
  - performance monitor display 7-7
- group operator
  - and partitioning 8-7

**H**

- hash partitioner

- characteristics 8-9
- key field distribution 8-10
- using 8-9
- hash partitioning 8-9
- header files
  - for custom operators 12-4
- here-document idiom
  - UNIX command operators and 13-23

**I**

- import 4-7
- import/export utility 4-7
- Input Interface Editor dialog box 12-9
- integer data type
  - See* data types
    - integer
- integer fields
  - record schema 4-21

**J**

- job-manager 2-7
  - examples 2-7, 2-8
  - syntax 2-7
    - abandon 2-8
    - errors 2-9
    - kill 2-8, 2-9
    - run 2-8

**L**

- link numbers
  - enabling 2-9
- Link Properties
  - Adapters tab 4-10
  - Advanced tab 4-10
  - dialog box 4-10
  - Schema tab 4-10
- Link Properties dialog box 4-11, 8-14
- links 4-10
  - configuring 4-10
  - persistent data sets and 4-10
- Lock Manager 2-15
- locking objects 2-15
- locks
  - clearing 2-15
  - in Visual Orchestrate 2-15
- logical nodes 10-2, 10-4
  - configuring 10-5

**M**

- massively parallel processing 1-3
- memory
  - setting in Visual Orchestrate 6-9
- messages
  - error 11-1
  - error log 11-1
  - Orchestrate format 11-2
  - warning 5-18, 11-1
- modify operator
  - conversions with 5-18
  - data type conversions and 3-10
  - preventing errors with 5-18
  - suppressing warning messages with 5-18
- movie files
  - performance monitor and 7-10
- MPP systems
  - constraints and 10-2
  - performance and 1-3
  - UNIX command operators and 13-24

**N**

- naming
  - fields 4-19
- native operators
  - defined 12-1
  - See also* custom operators
- node definitions 10-5
- node maps 10-1, 10-8
  - constraining operators to 10-8
  - default 10-8
  - example 10-8
  - operators and 10-1
  - using 10-8
- node names 10-5
- node pool constraints 10-6
  - data sets and 10-6
  - operators and 10-6
  - syntax 10-6
  - using 10-6
- node pools 10-1, 10-2, 10-5
  - default operator usage 10-1
- normalized table
  - data sets and 4-2
- nullability 4-3
  - introduction to 3-2
  - See* nulls
  - Vectors and 4-24

## nullable fields

- introduction to 3-2

## nulls 3-2, 4-3

- data types and 3-2
- default value of 4-27
- defined 4-19
- handling 4-19
- representation of 4-19
- vectors and 4-19

**O**

- operator interface schema 5-6
  - data set compatibility 3-8, 5-7, 5-17
    - aggregate fields 5-21
    - date fields 5-20
    - decimal fields 5-19
    - nulls and 5-21
    - string and numeric fields 5-18
    - time fields 5-20
    - timestamp fields 5-20
    - vector fields 5-21
  - data type conversion 3-8
  - matching to input data sets 3-8, 5-17
  - output data sets and 5-8
  - schema variables 5-11
- Operator Properties dialog box 6-14, 10-6
- operators
  - and output data sets 5-8
  - collecting and 9-1
  - configuring 5-3
    - Operator Properties dialog box and 5-4
    - Option Editor 5-5
  - constraining 10-1
  - constraints
    - node maps and 10-8
    - node pools and 10-6
  - creating 13-1
    - custom operators and 12-1
    - See* custom operators
  - creating steps from 6-2
  - data sets with 4-3, 5-2
  - data type conversions 3-8
  - disk pools
    - constraining to 4-36, 10-6, 10-7
  - dynamic interface schema
    - example 5-15
    - input schema 5-15
    - output schema 5-15

- setting interface schema 5-15
    - using 5-15
  - errors and 11-1
  - execution mode 1-10
    - controlling 5-2
    - parallel 5-2
    - sequential 5-2
  - input data sets with 5-7
  - inputs 5-1
  - interface schema 5-6
    - defined 5-6
  - introduction to 1-10
  - multiple inputs and outputs 5-1
  - node maps 10-1
    - constraining to 10-8
  - node pools 10-1
    - constraining to 4-36, 10-6
    - default usage of 10-1
  - nulls fields and 5-21
  - Operator Properties dialog box 5-4
  - Option Editor 5-5
  - output data sets and 5-8
  - outputs 5-1
  - parallel execution of 1-9
  - partitioning
    - preserve-partitioning flag and 8-11
  - partitioning and 8-1
  - performance monitor and 1-13
  - prebuilt 1-10
  - preserve-partitioning flag and 8-11
  - resource pools
    - constraining to 4-36, 10-6, 10-7
  - resources
    - constraining to 4-36, 10-6, 10-7
  - schema variables and 5-11
  - UNIX command operators 13-1
    - creating 13-1
  - vector fields and 5-21
- Option Editor 5-5
- Orchestra
- configuration file
    - default 2-12
    - validating 2-14
  - data types 3-1
    - date 3-2
    - decimal 3-4
    - floating-point 3-5
    - integer 3-6
    - raw 3-6
    - string 3-6
    - subrecord 3-6
    - tagged 3-6
    - time 3-6
    - timestamp 3-7
  - development environment 2-1
  - disk pools and 10-4
  - Orchestra Analytics Library (optional)
    - and partitioning 8-8
  - Orchestra application development
    - parallel execution mode 1-14
    - sequential execution mode 1-14
  - Orchestra applications
    - creating 2-1
    - See also* applications
  - Orchestra classes
    - APT\_Collector 9-3
    - APT\_Date 12-21
    - APT\_Decimal 12-23
    - APT\_Partitioner 8-4
    - APT\_RawField 12-25
    - APT\_StringField 12-24
    - APT\_Time 12-21
    - APT\_TimeStamp 12-21
  - Orchestra Installation and Administration Manual* 1-14
  - Orchestra server administrator 2-2
  - orchview
    - invoking 7-4
    - See also* performance monitor
  - osh
    - constraints and
      - combined node and resource 10-8
      - Visual Orchestra and 2-15
  - Output Interface Editor dialog box 12-9
- P**
- parallelism
    - partition 1-2
    - pipeline 1-2
  - partial record schema 4-18
  - partial record schemas
    - UNIX command operators and 13-39
  - partial schema definition 4-18
  - partition parallelism 1-2
  - partitioners
    - See* partitioning

- partitioning
  - data sets 1-2, 1-9, 4-7
  - fan-in 8-6
  - fan-out 8-6
  - method 8-1
    - DB2 8-4
    - entire 8-4, 8-5
    - examples 8-5
    - hash 8-4
    - modulus 8-4
    - other 8-4
    - random 8-4, 8-8
    - range 8-4
    - round robin 8-4, 8-7
    - same 8-4, 8-5
    - selecting 8-8
  - operators
    - keyed 8-9
    - keyless 8-9
    - selecting 8-8
  - operators and 8-1
  - parallel operators and 8-6
  - preserve-partitioning flag and 8-11
  - sequential operators and 8-6
- partitioning method 8-1, 8-3
  - any 8-4
  - DB2 8-4
  - default 8-4
  - entire 8-4, 8-5
  - hash 8-4
  - modulus 8-4
  - other 8-4
  - random 8-4, 8-8
  - range 8-4
  - round robin 8-4
  - same 8-4, 8-5
- partitioning operators
  - See* partitioning
- partitions
  - number of 8-8
  - See also* partitioning
  - similar-size 8-3
  - size of 8-3, 8-8
- performance
  - parallelization and 1-3
- performance monitor 7-1
  - controls
    - display 7-7
  - data sets 7-4
    - display 7-4
    - grid size 7-4
    - persistent 7-4
    - virtual 7-4
  - display 7-7
    - controls 7-7
    - data sets in 7-7
    - grids in 7-7
    - operators in 7-6
  - display window 7-1
  - Edit menu 7-8
  - invoking 7-4
  - movie files and 7-10
  - operators
    - display 1-13
  - Options dialog box 7-5, 7-7
  - records
    - rate of transfer 7-8
    - volume of transfer 7-8
  - rotate 7-1, 7-5
  - sampling interval 7-4
  - spreadsheets and 7-8
  - statistics
    - data sets 7-7
    - operators 7-6
    - spreadsheet 7-8
  - using 7-1
  - zoom 7-1, 7-5
- persistent data sets
  - collecting and 9-5
  - configuration file and 4-34
  - creating
    - orchadmin and 4-35
  - data files 4-34
  - descriptor file 4-34
  - dialog box 4-8
  - examples 4-6
  - file naming 4-37
  - introduction to 1-7
  - Link Properties dialog box and 4-11
  - operators and 5-2
  - representation of 4-32
- pipe safety 13-4
- pipeline parallelism 1-2
- post scripts 6-12
- pre scripts 6-12
- prebuilt operators 1-10

- preferences
  - setting 2-9
- preserve-partitioning flag 8-11
  - collecting and 9-2
  - examples of use 8-11, 8-14
  - sequential operators and 8-13
  - setting and clearing 8-14
  - usage rules 8-13
- Program Editor dialog box 10-6
- Program Editor window 6-5
- Program Properties
  - dialog box 2-4
  - Parameters tab 2-15
  - Server tab 2-5
- psort operator
  - and partitioning 8-7
- R**
- range partitioner 8-10
  - key field distribution 8-10
  - using 8-10
- raw data type
  - See* data types
  - raw
- raw fields
  - aligned 4-21
  - schema definition 4-21
- record count 4-16
- record fields 1-6
  - date fields
    - schema definition 4-20
  - decimal fields
    - schema definition 4-21
  - default values 4-27
  - floating-point fields
    - schema definition 4-21
  - integer fields
    - schema definition 4-21
  - naming 4-19
  - raw fields
    - schema definition 4-21
  - string fields
    - schema definition 4-22
  - subrecord fields
    - schema definition 4-25
  - tagged aggregate fields
    - schema definition 4-26
  - time fields
    - schema definition 4-23
    - timestamp fields
      - schema definition 4-23
- record schema 1-6
  - automatic creation of 4-17
  - date fields 4-20
  - decimal fields 4-21
  - defined 4-2
  - example 4-16
  - field identifier 4-3
  - floating-point fields 4-21
  - import of 4-17
  - importing 4-32
  - inheritance of 4-17
  - integer fields 4-21
  - matching with operators 3-8, 5-17
  - nullability 4-3
  - partial 4-18
  - raw fields 4-21
  - Schema Editor and 4-27
    - aggregate fields 4-30
    - creating 4-29
    - importing 4-32
  - schema variables and 5-11
  - string fields 4-3, 4-22
    - fixed length 4-22
    - length 4-3
    - variable length 4-22
  - subrecord fields 4-25
  - syntax 4-16
  - tagged aggregate fields 4-26
  - time fields 4-23
  - timestamp fields 4-23
- record schema editing
  - Schema Editor and 4-30
- record transfers 5-14
  - and schema variables 5-11
- records 1-6
  - fixed-length 4-1
  - variable-length 4-1
- resource pools 10-1, 10-5
- resources 10-1
- roundrobin operator 8-7
- running an application 2-6
- run-time errors 11-1
  - See also* error messages
  - See also* warning messages

**S**

- sampling interval
  - performance monitor 7-4
- scalability 1-3
- schema definition
  - complete 4-18
  - partial
    - when to use 4-18
- schema definition files 4-17
- Schema Editor 4-27
  - aggregate fields and 4-30
  - creating schema with 4-29
  - dialog box 4-28
  - editing schema with 4-30
  - importing schema 4-32
  - new schema from existing schema 4-30
  - using 4-27
- schema variables 5-11
  - field adapters and 5-13
  - output
    - record schema of 5-13
    - record schema of 5-12
    - transfer mechanism and 5-14
- scratch disks 10-5
- script generation 2-15
- scripts
  - pre and post 6-12
- sequential execution mode 1-14
  - virtual data sets and 6-11
- server
  - automatic connection 2-12
  - connecting to 2-3
  - default 2-11
  - introduction to 2-2
  - variables 6-12
- server variables 6-12
- set\_orchserver\_variable 6-12
- shared-nothing systems 1-3
- shell scripts
  - steps and 6-12
- SMP systems
  - constraints and 10-4
  - disk I/O and 1-3
  - performance and 1-3
  - scaling and 1-3
  - UNIX command operators and 13-24
- sortmerge operator
  - example of use 9-6
- spreadsheets
  - performance monitor 7-8
- Step Properties dialog box 6-6
  - env properties 6-10
  - execution mode properties 6-10
  - paths properties
    - compiler path 2-13
    - conductor host 2-13
    - sort dir 2-13
  - post properties 6-12
  - pre properties 6-12
  - server properties 6-8
    - config 6-8
    - database 6-8
- steps
  - branching in 6-3
  - check flag and 6-7
  - checking 6-7
  - configuring 6-6
    - Step Properties dialog box 6-6
  - creating 6-5
  - data-flow models of 6-2
  - defined 5-1
  - designing 6-1
  - errors and 11-1
  - executing 6-7
  - get\_orchserver\_variable and 6-13
  - introduction to 1-11
  - multiple-step applications 6-4
  - performance monitor and 7-1
  - running 6-7
  - set\_orchserver\_variable and 6-12
  - shell scripts and 6-12
  - single-step applications 6-3
  - Step Properties dialog box and 6-6
    - server properties 6-8
  - using operators to create 6-2
  - virtual data sets in 6-3
- string data type
  - See* data types
  - string
- string fields
  - schema definition 4-22
- subrecord data type
  - See* data types
  - subrecord
- subrecord fields
  - nested 4-25

- referencing 4-25
- schema definition 4-25

## T

tagged aggregate fields

- nullability of 4-27
- referencing 4-26
- schema definition 4-26

tagged data type

- See* data types
- tagged

temporary file usage 2-13

time data type

- See* data types
- time

time fields 4-23

- data type 4-23
- schema definition of 4-23

timestamp data type

- See* data types
- timestamp

timestamp fields 4-23

- data type 4-23
- schema definition of 4-23

transfer mechanism 5-14

transfers 5-11

- See* record transfers

## U

UNIX command operators 13-1

- advantages of 13-1
- argument order 13-11
- characteristics of 13-2
- checkpointing and 13-13
- constraints and 13-13
- creating 13-1
  - example 13-9, 13-10, 13-13, 13-28
- data sets and
  - input 13-8
  - output 13-8
  - record schemas 13-8
- environment variables and 13-26
  - example 13-26
  - using 13-26
- example 13-9, 13-13, 13-28
- execution mode 13-2, 13-5
- execution model 13-5
- exit codes and 13-11, 13-35

- example 13-11

export and 13-5

- controlling 13-6
- default 13-6

filesets and 13-38

import and 13-5

- controlling 13-6
- default 13-6

input files and 13-13

- example 13-13
- MPP 13-25
- SMP 13-25

inputs and 13-6, 13-7

- data sets 13-8
- default record schema 13-9

intact schemas and 13-39

optimizations of 13-36

options and 13-28

- data types of 13-30
- defining 13-28

output files and 13-13

- APT\_PARTITION\_NUMBER 13-24
- example 13-13
- MPP 13-24
- SMP 13-24
- using 13-24

outputs and 13-6, 13-7

- data sets 13-8
- default record schema 13-9

parameter files and 13-6

partial record schemas and 13-39

pipe safety 13-4

record schemas and 13-6, 13-8

- default 13-8
- defining 13-8
- example 13-19

scripts in 13-22

shell scripts and 13-22, 13-24

- environment variables 13-24
- example 13-24
- here-document idiom 13-23
- using 13-22

stdin

- example 13-10
- using 13-10

stdout

- example 13-10
- using 13-10

- SyncSort
      - example 13-13
    - UNIX commands and 13-3
      - requirements for 13-4
    - UNIX shell commands
      - defined 13-3
    - user options and 13-28
      - data types of 13-30
      - defining 13-28
      - example 13-28
  - UNIX commands
    - characteristics of 13-3
    - UNIX command operators and 13-3
      - requirements for 13-4
- V**
- variable-length field 4-1
  - variable-length records 4-1
  - vector fields
    - defining 4-24
    - in input data sets 5-7
    - operators and 5-21
  - vectors
    - data types and 4-24
    - introduction to 3-2
    - nullability of elements 4-24
    - nulls and 4-19
    - numbering elements 4-24
    - of subrecords 4-25
    - referencing elements 4-24
  - vendor icons
    - enabling 2-10
  - virtual data sets
    - examples 4-5
    - introduction to 1-7
    - Link Properties dialog box and 4-11
    - operators and 5-2
    - representation of 4-32
    - sequential execution mode and 6-11
    - steps and 6-3
  - Visual Orchestrate 2-2
    - automatic connection 2-12
    - configuration validation 2-14
    - configuring 2-4
    - connecting 2-3
    - connection timeout 2-12
    - creating applications 2-2
    - Data Set Viewer 4-14
    - data sets
      - dialog box 4-8
    - default configuration 2-12
    - default directory 2-12, 2-13
    - default library 2-12
    - default server 2-11
    - deploying applications 2-2
    - development environment 2-1
    - enabling
      - link numbers 2-9
      - vendor icons 2-10
    - introduction to 2-1
    - locking objects in 2-15
    - main window 2-2
    - osh and 2-15
    - osh script generation 2-15
    - preferences 2-9
    - program parameters 2-15
    - program properties 2-4
    - RDBMS configuration and 2-4
    - running an application 2-6
    - scripts 6-12
    - steps 6-4
    - text wrap and 2-10
    - user preferences and 2-9
    - validating applications 2-6
- W**
- warning messages
    - default components 11-4
    - display control
      - APT\_ERROR\_CONFIGURATION and 11-4
      - default display 11-4
      - keywords 11-4
    - example 11-4
    - keywords
      - errorIndex 11-2
      - ipaddr 11-3
      - jobid 11-2
      - lengthprefix 11-3
      - message 11-3
      - moduleId 11-2
      - nodename 11-3
      - nodeplayer 11-3
      - opid 11-3
      - severity 11-2
      - timestamp 11-2



- vseverity 11-2
  - subprocesses and
    - default display 11-3
- warnings
  - log 5-18
  - Orchestrate handling 5-18

**X**

- X Windows
  - performance monitor and 7-1