Sybase, Inc.
One Sybase Drive
Dublin, CA 94568
www.sybase.com

Data Integration with Sybase Avaki Studio

Release 7.0 • August 24, 2006

DOCUMENT ID: DC00574-01-0700-01

LAST REVISED: August 2006

Set in Arial, `Courier New`, and Times New Roman. Stanley Morison, the creator of Times New Roman, said of it: “By the vice of Mammon and the misery of the machine, it is bigoted and narrow, mean and puritan.”

Credits
This product includes software developed by the Apache Software Foundation (http://www. apache.org). This product includes Hypersonic SQL and ANTLR. This product includes code licenses from RSA Security, Inc. Some portions licensed from IBM are available at http://oss.software.ibm.com/icu4j/. Contains IBM® 64-bit Runtime Environment for AIX™, Java™ 2 Technology Edition Version 1.4 Modules © Copyright IBM Corporation 1999, 2000 All Rights Reserved. Contains the SAXON XSLT Processor from Michael Kay, which is available at http://saxon.sourceforge.net. This product includes software developed by the Proxool Project (http://proxool.sourceforge.net).

Data Integration with Sybase Avaki Studio
Written by Beth Thoenen, Luis Valdez, Stephanos Bacon, and Kelly Parr Soli

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

Table of contents

Preface

This guide, *Data Integration with Sybase Avaki Studio*, explains how to use Avaki Studio software. With Studio, you can provision data sources into the Avaki data catalog and create data services that combine and transform your data. You can also manage your data catalog.

Note This book and the product's user interfaces refer to Sybase Avaki EII software as *Avaki* or *Avaki Data Grid*.

Audience

This book is written for readers with an understanding of Avaki provisioning and integration tools, including database connectors, database operations, and data services. These tools are described in the *Sybase Avaki EII Overture*, which you should read before using Avaki Studio.

We recommend that Studio users know how to use a reporting tool such as Crystal Reports or Excel.

Knowledge of SQL is necessary if you want to create your own database operations. It is helpful but not necessary for understanding view model operators, many of which are based on SQL concepts.

Familiarity with the JavaScript language is helpful for Avaki Studio users, but not always necessary. Some view models can be completed by users with no JavaScript at all, while others require partial to full working knowledge. In Avaki Studio, you can employ JavaScript in the following circumstances:

- Several operators allow or require you to enter JavaScript control expressions. In many cases, however, you can construct the expression you need using elements provided in the menus. See Chapter 7, "Operator reference," for details on the operators that accept JavaScript expressions.
- With some working knowledge of Javascript, you can write Javascript functions in JavaScript include files and call the functions from the JavaScript expressions in your operators.
- To write a custom operator, you need full working knowledge of JavaScript.

Organization

This book is organized as follows:

Chapter 1 View models in Avaki Studio	Provides an overview of view models, which represent data flows in Avaki Studio, and their components.
Chapter 2 Getting started with Avaki Studio	Explains how to start Avaki Studio for the first time and provides a tour of the tools you can use.
Chapter 3 Creating view models	A tutorial that walks you through the process of creating a view model and deploying it as a data service.
Chapter 4 The view model editor	Shows how to use the view model editor and how to use JavaScript expressions with operators.
Chapter 5 Metadata modeling	Provides an overview of metadata modeling and describes tools for creating and using metadata models.
Chapter 6 Managing the data catalog	Describes tasks you can perform in Avaki Studio's data catalog view, including modifying access control lists, creating and modifying attributes, and creating categories and adding objects to them.
Chapter 7 Operator reference	Provides detailed information on how Avaki Studio's operators, Input Source elements, and Result elements work.
Glossary	Defines terms used in this guide

Related documentation

These manuals make up the Avaki documentation set:

- *Sybase Avaki EII Overture*
- *Sybase Avaki EII Administration Guide* (includes installation instructions)
- *Data Integration with Sybase Avaki Studio*
- *Sybase Avaki EII Provisioning and Advanced Data Integration Guide*
- *Sybase Avaki EII API Guide*
- *Sybase Avaki EII Command Reference*

The manuals are included, in PDF format, on the CD with Avaki software. They are stored in the docs subdirectory of the Avaki installation directory. *Data Integration with Sybase Avaki Studio* is also accessible in Avaki Studio.

To access the manuals via Avaki's web user interface, log in to your Avaki domain and click the **Help** link at the top right corner of any page of the web UI.

How to contact Avaki support at Sybase, Inc.

For general information about Sybase technical support, see the *Customer Service Reference Guide* at

http://www.sybase.com/support/aboutsupport/guide/csrg

Please contact us with any questions or difficulties you encounter.

By telephone

In North America, call toll free: 1-800-8SYBASE

Outside North America, follow the link below to see a list of Sybase offices and phone numbers around the world.

http://www.sybase.com/contactus/support

On the web

If you are a designated contact for a technical support plan, you can log and track cases on the web using the Case Express application. At www.sybase.com, mouse over the **Support and Services** tab and select **Case Management** from the dropdown list. Use the email address and password for your mysybase account to log in.

Chapter 1

View models in Avaki Studio

Avaki Studio is a graphical, metadata-driven modeling tool for data integration that lets you combine data from heterogeneous data sources. With Avaki Studio, you can build *view models* by dragging and dropping input sources, operators, and output targets. A view model is a sequence of operations that combine or transform data from one or more sources. This chapter introduces Avaki Studio and the tools it provides for provisioning data sources, building view models, and deploying them as Avaki *data services*.

In this chapter:

Background: Avaki concepts

Before tackling the task of creating a view model, you should be familiar with some Avaki concepts and terminology. This information can be found in the *Sybase Avaki EII Overture.* You're ready to use Avaki Studio when you know the answers to these questions:

- What is a database connector?
- What is a database operation?
- What is a data service and what is it good for?
- How do data services interact with database operations and with other data services?
- What are data service dependencies and database operation dependencies?
- What is a result set?

What is a view model?

A view model is the abstract, graphical representation of the processing performed by a particular Avaki data service. The view model represents the flow of data from one or more input sources through a number of operators to an output result. You create a view model and deploy the resulting data service(s) using Avaki Studio.

The inputs to a view model can be relational result sets produced by Avaki database operations or data services; XML results produced by data services; data catalog files in CSV or XML format; HTTP operations; and web services calls.

A view model is not required to have any inputs; it can generate its own data. Nor is it required to have any operators—though there would be no point to a view model that has neither inputs nor operators.

Outputs are also in result set format. Most operators produce a single output (the exception is the **Splitter** operator). Every view model includes a **Result** element, but a view model need not produce output. View models can produce at most one output result set.

Thus, a view model consists of

- Zero or more input sources
- Zero or more operators to manipulate the data
- Exactly one **Result** element

Avaki Studio vs. writing your own data service plug-in

Because of its graphical nature and rich user interface, Studio is the preferred way to create many data services. However, data services created in Studio are fundamentally relational in nature; while they can take in nonrelational data, the first step in doing so in Studio is to specify a transformation on that data that yields a relational result. If you want your data service to work or produce results outside the relational paradigm, you'll need either to use one of the built-in plug-ins provided by Sybase or to build a data service plug-in of your own. (You might use the built-in no-operation plug-in to provision a web service's data as XML, or the XSLT plug-in to use XSLT to process one or more XML inputs.) For another format—a data service that does image processing, perhaps—you'd write your own plug-in.

For information on Avaki's built-in data service plug-ins and on writing your own plug-ins in Java, JavaScript, or XSLT, see the *Sybase Avaki EII Provisioning and Advanced Data Integration Guide*.

Schemas

Each result set in a view model, from the inputs to the result, is defined by a schema consisting of an ordered set of typed columns. The output schema of the view model is the schema of the operator that is directly connected to the **Result** element.

Studio lets you display the schema information for any element in a view model. By manipulating the properties of some operators, you can also alter the output schema. Schemas are shown in the Table Schema tab of the bottom pane of the Avaki perspective.

Studio looks for schemas (.xsd files) for input sources in the Avaki domain's /Metadata directory. Note, however, that newly created database operations and data services have no metadata. To create metadata, you must either execute the database operation or data service, or use Studio's "generate schema" feature. For information on generating schemas, see "The data catalog view" on page 18.

Data flow

Discussions of view model operators and especially of variables rely on the notion that in a view model, data flows toward the result—and generally from left to right. We use the terms *upstream* and *downstream* to indicate direction in a view model:

- Upstream: away from the **Result** element
- Downstream: towards the **Result** element

Thus, in a view model with two **Select** operators, the downstream **Select** is the one closer to the **Result** element.

Parameters and control variables

View models can have variables, which can be used by the various operators. Variables include global parameters (that is, parameters to the entire view model/data service) and control variables introduced by some operators (such as **Iterator** and **Custom**). These variables, along with the values of columns in the inputs to an operator, can be referenced in the expressions that control the behavior of the operators. Some examples are the `where` expression in a **Select** operator and the column definitions in a **Projection** operator.

Control variables introduced by various operators are accessible only to expressions in upstream operators—that is, in operators to the left, whose output directly or indirectly flows into the operator that introduces the control variable.

View model parameters are accessible from any place in the view model where you can enter expressions. View model parameter values can be singletons or arrays. To see a list of data types you can use for parameters, click in the Type column of the parameters tab and pull down the menu.

You can view, enter, and modify view model parameters on the Parameters tab in the bottom pane of Studio's Avaki perspective. See "Using global parameters" on page 63.

Expressions in operators

Several view model operators allow or require you to enter JavaScript expressions that control the behavior of the operator. For example, to use **Projection** to combine two columns, you might write an expression that concatenates the columns, multiplies them, or filters out the larger of two values.

Everywhere you can enter JavaScript expressions, you have access to an object called `variables`, which in turn gives you access to parameters, control variables for downstream operators, and columns in the input result sets.

Studio provides contextual menus for constructing JavaScript expressions. The menus let you click to select elements like column names, operators, global parameters, and predefined functions. For many expressions, no knowledge of JavaScript—or touch typing—is required.

For more information, see "Using expressions within Avaki Studio" on page 66.

Performance

Two important points affect the performance of data services deployed from Avaki Studio:

- The operators are all implemented in Java, and all the significant data manipulation happens in Java. JavaScript is used only to tie the operations together and to provide a way to evaluate expressions (for example, the `where` clause in a Select operator).
- The runtime model is implemented in a "lazy," or "pull," fashion that defers any computation until a result is requested by the next step in the model. Even when a result is requested (with a couple of exceptions), an operator performs only as much computation as necessary to produce the next row before returning. Thus, a model consisting only of a **Select** operator and a **Projection** operator will run through the input result set exactly once and will not create any intermediate result sets that need to be copied.

  Exceptions to this rule are operators that need to examine all rows of one or more of their inputs. These include **Order By** and all the operators that depend on sorted values such as **Join**, **Group By**, **Aggregate**, and **Intersection**, as well as any operator that uses the Distinct option to eliminate duplicate rows.

Operators

Avaki Studio provides a selection of elements and operators you can use to manipulate data in a view model. The operators are described briefly here. For details on how the operators work and how to use them, see Chapter 7, “Operator reference.”

Beginning and ending data flows. Two important elements identify the beginnings and the end of the flow in a view model:

- **Input Source**
  Use **Input Source** elements to specify where the data for you view model comes from, such as existing view models (data services), database operations, or grid files. A view model can have zero or more input sources.

- **Result**
  Studio puts a **Result** element at the end of each view model’s data flow. A view model has one result.

Relational operations. These operators let you perform SQL-style operations:

- **Select**
  Given one input result set and a Boolean `where` expression, **Select** produces a new result set containing the input rows for which the `where` expression evaluates to true.

- **Projection**
  Projection lets you modify the schema—you can remove columns, add computed columns, or apply formulas to alter existing column values.

- **Order By**
  Given one input result set and your specification of which column(s) to sort on and whether to sort in ascending or descending order, **Order By** produces a new result set in which the content of the input is sorted.

- **Aggregate**
  Avaki provides a number of predefined aggregation functions (including sum, count, average, minimum, maximum). Given one input result set, **Aggregate** produces a new, single-row result set by applying aggregate functions to the input.

- **Group By**
  Group By uses the same aggregation functions used in **Aggregate**. Given one input result set, **Group By** applies aggregate functions to the input and groups the rows in the result set according to input columns you specify.

- **Join**
  Given two input result sets that share a common column, **Join** combines rows and produces a new result set. **Join** supports Inner, Outer-left, Outer-right, and Outer-full join types, as well as three join algorithms: Sort Merge Join, Hash Join, and Nested Loop Join.

- **Union**
  Given two input result sets with identical schemas, **Union** generates a single result set that includes everything in both inputs.

- **Intersection**
  Given two input result sets with identical schemas, **Intersection** generates a result set containing only rows that are present in both inputs.

Advanced operations. Additional operators let you perform more powerful manipulations:

- **Multiplexer**
  Given up to five inputs and a conditional expression for each, **Multiplexer** generates a result set in which all the input result sets whose conditional expressions evaluate to true are concatenated. You can use **Multiplexer** to select from among several choices and to concatenate the selected result sets into a single result set.

- **Iterator**
  Iterator executes its primary input repeatedly; the number of iterations is determined by an input parameter or by an optional secondary input. For example, you can perform a computation once for every date in a range, or get account numbers and balances for every social security number in a list. You can also specify a condition to terminate the iteration. Iterations may be nested.

- **Splitter**
  Given one input result set, **Splitter** routes input to multiple, identical output result sets without necessarily recomputing its input. It can also cache its output.

- **Generator**
  Generator creates a new result set based on JavaScript expressions and a schema that you supply.

- **Update**
  Using an Avaki database operation, **Update** inserts data into an external database. Each update is a transaction separate from the data service itself.

Write your own operator. A final operator lets you define your own operation:

- **Custom**
  Using JavaScript logic that you supply, **Custom** performs an operation of your design.

Chapter 2

Getting started with Avaki Studio

Read this chapter for instructions on starting Avaki Studio for the first time and for general information about the tools Studio provides.

In this chapter:

Installing Avaki Studio

Avaki Studio runs on Windows 2003 and Windows XP. You install Studio from the standard Avaki installer. When you reach the installer's component selection screen, deselect any unwanted components—but be sure to leave Avaki Studio selected.

If you need more information on running the installer, see the *Sybase Avaki EII Administration Guide*.

Starting Avaki Studio

You can launch Avaki Studio using the Avaki program group or shortcut that was created during installation. By default, the installer creates an Avaki Data Grid program group in the Programs portion of your Windows Start menu.

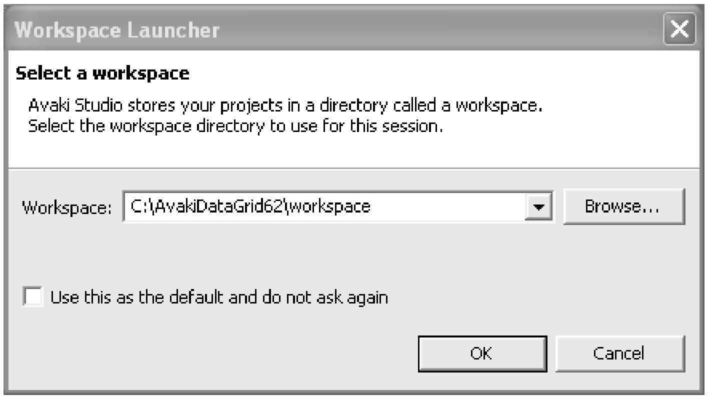
To start Studio, select Start > Programs > Avaki Data Grid 7.0 > Avaki Studio

You can also start Studio from the command line. In the Avaki installation directory, enter

```
C:\AvakiDataGrid70> avaki_studio
```

Specifying a workspace directory

When you start Avaki Studio for the first time, you must specify the directory (folder) where Studio will store your projects:



Avaki Studio stores your projects in a directory called a workspace.
Select the workspace directory to use for this session.

Workspace: C:\AvakiDataGrid62\workspace

If you don't want to use the default directory, click **Browse...** to select a different one.

Click the checkbox for "Use this as the default and do not ask again."

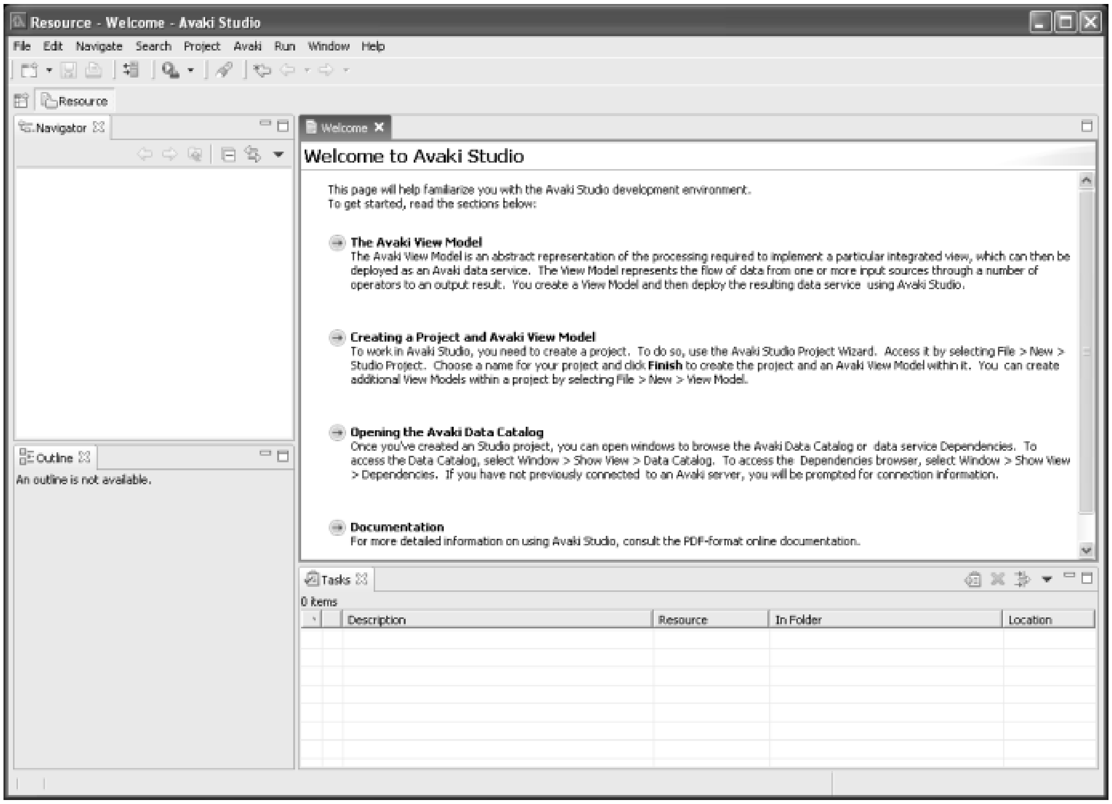
You can change the workspace directory from within Studio if you change your mind later: select **Switch Workspace** from the File menu. Note, however, that changing the workspace causes Studio to shut down and restart.

In the workspace directory, Studio creates a subdirectory for each project. Files of the following types can be found in the workspace project directories:

Filename extension	Purpose
.avm	Studio creates an Avaki view model (.avm) file for each view model.
.amm	Studio creates an Avaki metadata model (.amm) file for each metadata model.
.js	Studio creates a JavaScript (.js) file for each model; the .js file has the same base name as the model's .avm file. This is the executable form of the view model; when the view model is deployed as a data service, the JavaScript file is sometimes called the *data service plug-in*. JavaScript plug-in files are stored in the \bin subdirectory of the workspace project directory.
.jsi	You create a JavaScript include (.jsi) file if you have JavaScript functions that you want to invoke in your view model. Give the .jsi file the same base name as the model's .avm file.
.project	Eclipse Workbench creates a .project file for each project in the project directory. (Files whose names begin with a . might not be visible to you, depending on your Eclipse filter settings.)

The Resource perspective

After you specify a workspace directory, a window opens to display the Resource perspective, which is the opening screen for Eclipse Workbench.


Resource - Welcome - Avaki Studio
File Edit Navigate Search Project Avaki Run Window Help
Resource
Navigator
Welcome
Outline
An outline is not available.
Tasks
0 items
Description
Resource
In Folder
Location

Welcome to Avaki Studio

This page will help familiarize you with the Avaki Studio development environment.
To get started, read the sections below:

The Avaki View Model
The Avaki View Model is an abstract representation of the processing required to implement a particular integrated view, which can then be deployed as an Avaki data service. The View Model represents the flow of data from one or more input sources through a number of operators to an output result. You create a View Model and then deploy the resulting data service using Avaki Studio.

Creating a Project and Avaki View Model
To work in Avaki Studio, you need to create a project. To do so, use the Avaki Studio Project Wizard. Access it by selecting File > New > Studio Project. Choose a name for your project and click **Finish** to create the project and an Avaki View Model within it. You can create additional View Models within a project by selecting File > New > View Model.

Opening the Avaki Data Catalog
Once you've created an Studio project, you can open windows to browse the Avaki Data Catalog or data service Dependencies. To access the Data Catalog, select Window > Show View > Data Catalog. To access the Dependencies browser, select Window > Show View > Dependencies. If you have not previously connected to an Avaki server, you will be prompted for connection information.

Documentation
For more detailed information on using Avaki Studio, consult the PDF-format online documentation.

Background: Eclipse Workbench concepts

Avaki Studio is built on the Eclipse Workbench platform, version 3.1.2. Some of the tools and concepts you'll use to work on Avaki view models originate in Workbench:

Workspace. A container for projects. Each workspace stores files related to its projects in a workspace directory whose location you specify. The workspace directory, in turn, contains a subdirectory for each project you create.

Project. A container for view models. You can have multiple view models in one project and multiple projects in a workspace.

Perspective. A collection of Workbench panes customized to facilitate work on projects of a particular type. Studio includes a preconfigured Avaki perspective that lets you create projects and view models right away.

We strongly recommend that you look at the Eclipse documentation, which is available in Studio. Some common tasks, including importing and renaming Studio projects, are performed using Eclipse tools.

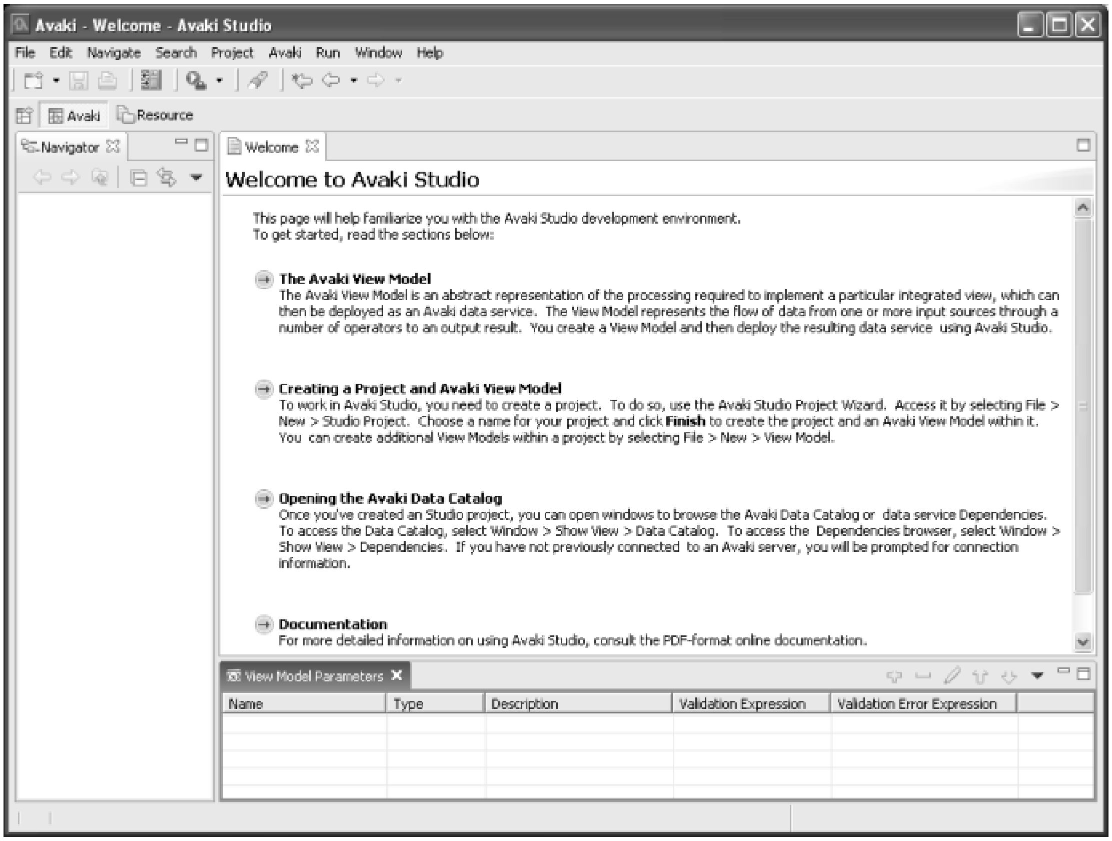
To view Eclipse's *Workbench User Guide*, select **Help Contents** from the Help menu. In the Contents pane on the left side of the resulting help window, click **Workbench User Guide**.

Creating a project

To use the tools provided by Avaki Studio, you must open the Avaki perspective. In the Avaki perspective, you can create a project and start working on your first view model.

From the Resource perspective, follow these steps to open the Avaki perspective and create a new project:

Step 1 Pull down the Window menu and select **Open Perspective > Avaki**. The Resource perspective changes to the Avaki perspective.


Avaki - Welcome - Avaki Studio
File Edit Navigate Search Project Avaki Run Window Help
Avaki
Resource
Navigator
Welcome
View Model Parameters

Welcome to Avaki Studio

This page will help familiarize you with the Avaki Studio development environment.
To get started, read the sections below:

The Avaki View Model
The Avaki View Model is an abstract representation of the processing required to implement a particular integrated view, which can then be deployed as an Avaki data service. The View Model represents the flow of data from one or more input sources through a number of operators to an output result. You create a View Model and then deploy the resulting data service using Avaki Studio.

Creating a Project and Avaki View Model
To work in Avaki Studio, you need to create a project. To do so, use the Avaki Studio Project Wizard. Access it by selecting File > New > Studio Project. Choose a name for your project and click **Finish** to create the project and an Avaki View Model within it. You can create additional View Models within a project by selecting File > New > View Model.

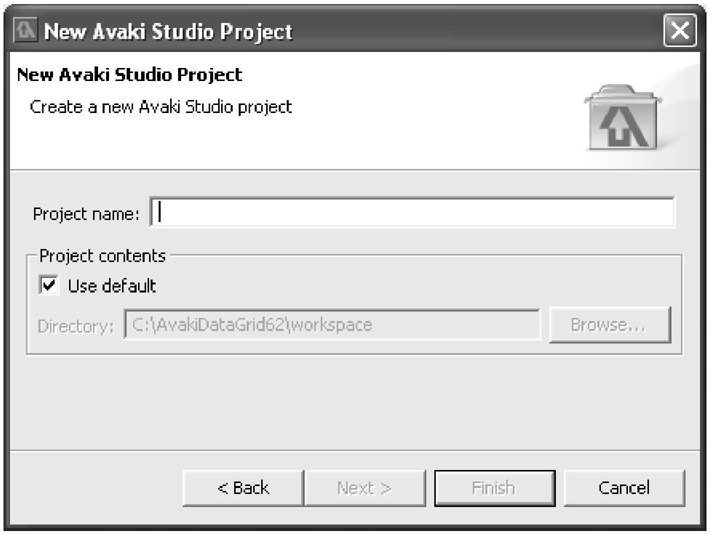
Opening the Avaki Data Catalog
Once you've created an Studio project, you can open windows to browse the Avaki Data Catalog or data service Dependencies. To access the Data Catalog, select Window > Show View > Data Catalog. To access the Dependencies browser, select Window > Show View > Dependencies. If you have not previously connected to an Avaki server, you will be prompted for connection information.

Documentation
For more detailed information on using Avaki Studio, consult the PDF-format online documentation.

Name	Type	Description	Validation Expression	Validation Error Expression

Step 2 To create a project, pull down the File menu and select **New > Studio Project...**. The

New Avaki Studio Project wizard appears.


New Avaki Studio Project

New Avaki Studio Project

Create a new Avaki Studio project

Project name:

Project contents

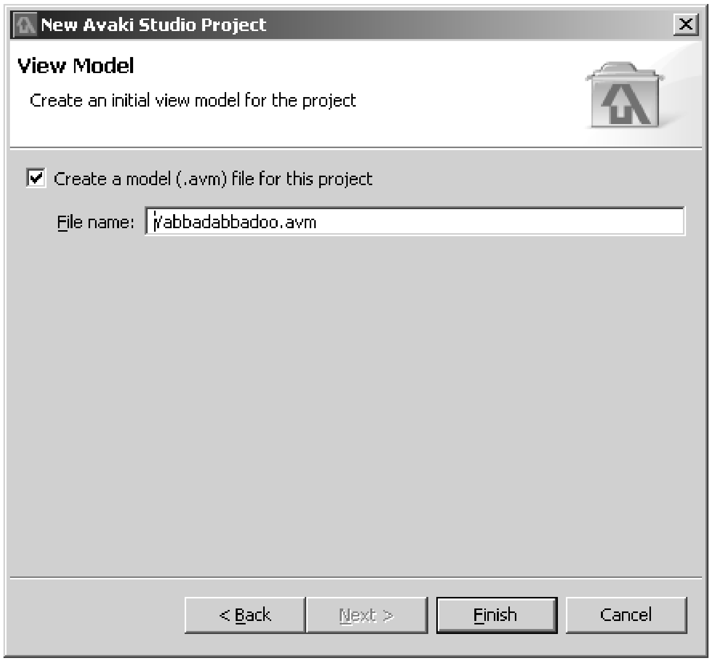
☑ Use default

Directory: C:\AvakiDataGrid62\workspace

Step 3 In the Project name field, enter a name for your project. By default, Studio creates a subdirectory with this name in your workspace directory—this is your project directory.

Step 4 (Optional) To specify an alternative location for the project directory, click to uncheck the "Use default" box, then enter or browse to the new location.

Step 5 In the File Name box, enter a name for the model (.avm) file for the project.


New Avaki Studio Project

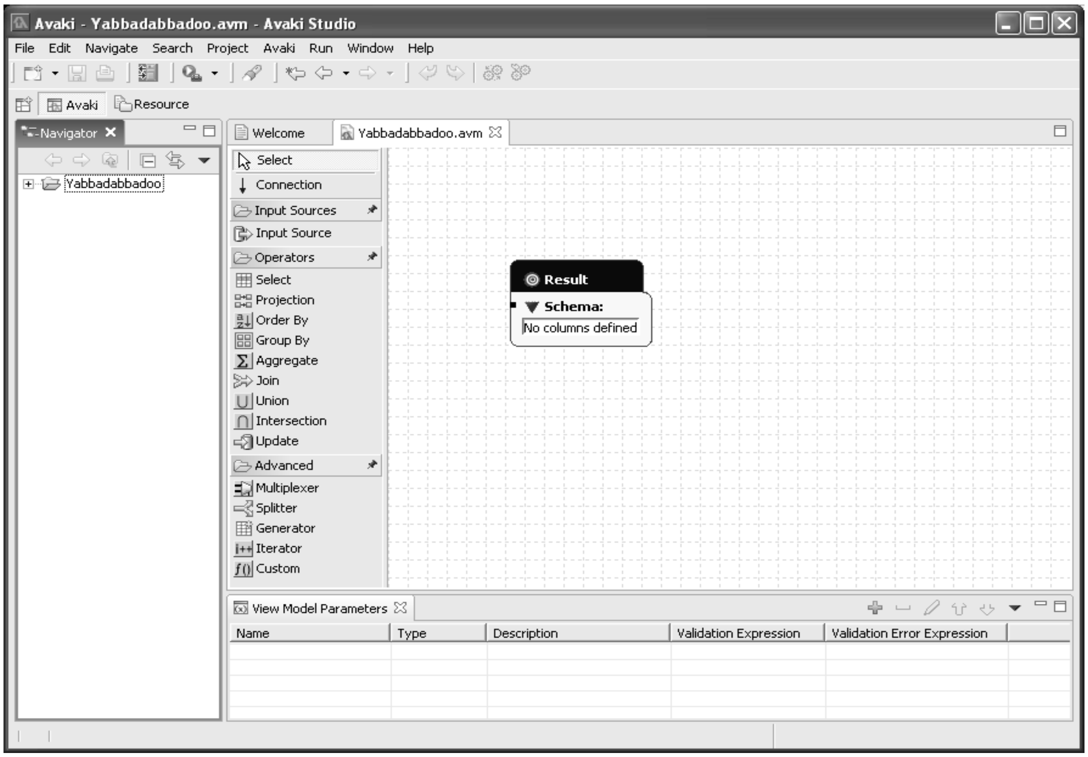
View Model

Create an initial view model for the project

☑ Create a model (.avm) file for this project

File name: yabbadabbadoo.avm

Step 6 Click **Finish** to create the project. The project includes a starter view model consisting of a result.

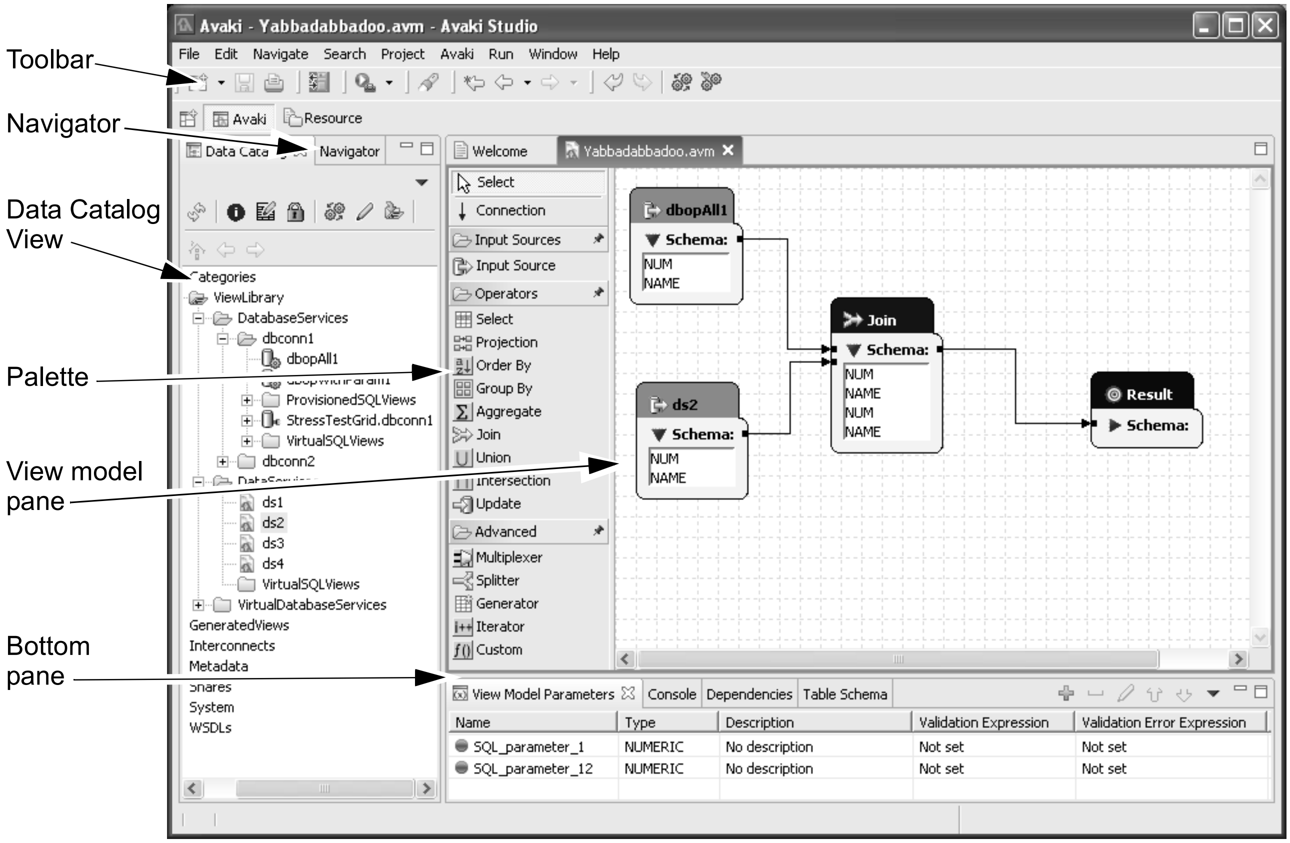


Red borders. When an element (an input source, operator, or result) in the view model is not properly set up or connected to other elements, Studio outlines the problem element in red. If you mouse over the element, Studio displays an explanation of the problem. For example, you see the following when you mouse over an unconnected **Result** element in a new view model:

```
Error: Model is not valid. Result has no inputs.
```

The Avaki perspective: A tour

This section explains the purposes of the panes and other elements in the Avaki perspective window.



Name	Type	Description	Validation Expression	Validation Error Expression
● SQL_parameter_1	NUMERIC	No description	Not set	Not set
● SQL_parameter_12	NUMERIC	No description	Not set	Not set

The toolbar

The icons in the toolbar at the top of the window let you perform tasks like creating new view models or projects, saving the current view model, and connecting to or disconnecting from the Avaki server. Mouse over an icon to see a tool tip describing what it does.

The navigator

Use the navigator, in the left-hand pane, to browse and manage the files in your project directories. Here, you can perform standard file-browser tasks like expanding and collapsing directories and opening files.

To open a saved view model, expand the view model's project folder in the navigator and double-click on the .avm file (myviewmodel.avm, for example). The view model opens in a new tab in the view model pane; to close it, click the **X** on the tab. To save the view model, select **Save** or **Save As...** from the File menu, or enter **Ctrl–s**.

The data catalog view

The data catalog view shares the left-hand pane with the Navigator. To display it, pull down the Window menu and select **Show View** > **Data Catalog**. Click **Yes** when Studio asks whether to connect to the Avaki server, and provide any requested Avaki login information.

Browsing for input sources

When Studio is connected to an Avaki domain, you can use the data catalog view to browse the domain's Avaki directories for relational and nonrelational input sources.

Relational input sources. Of interest in most domains is the Categories folder, where you can browse for relational input sources such as database operations, data services, and SQL views:

Categories/ViewLibrary/...	Contents
DatabaseServices/<dbconn>/	Database operations stored in folders named for their database connectors
DatabaseServices/<dbconn>/ProvisionedSQL-Views	SQL views provisioned from tables in external databases
DatabaseServices/<dbconn>/VirtualSQLViews	SQL views generated from database operations
DataServices	Data services
DataServices/VirtualSQLViews	SQL views generated from data services
VirtualDatabaseServices/Operations	Virtual database operations
VirtualDatabaseServices/Services	Reserved for system use
VirtualDatabaseServices/VirtualSQLViews	SQL views generated from virtual database operations

Nonrelational input sources. The location of nonrelational input sources such as XML files, CSV files, and WSDL files can vary from one Avaki domain to another;

good places to start are the /GeneratedViews, /Shares, and /WSDLs folders. (For more information on the contents of the default directories in the data catalog, see the *Avaki Overture*.)

Once you've found the input source you want, you can drag and drop it from the Data Catalog view to the view model pane. Studio automatically creates a corresponding Input Source element in the view model.

Creating and managing input sources

You can also use the Data Catalog view to create input sources (both provisioned and generated SQL views) and to manage input sources. Select an item of interest in the catalog and right-click for a context menu. You can:

- Provision a SQL view from a database table.
- View database schemas.
- Generate a SQL view from a data service, database operation, or a virtual database operation.
- Generate schema for a data service, a database operation, or a virtual database operation. Generating schema generally involves a call on the database and at least a partial execution of the data service or database operation. Studio extracts the schema from the returned metadata without iterating over the rows of the result set. The exception is updates—the schema for a database operation that includes an update operation can be generated without executing the database operation.

  Note that when you create a database operation or deploy a data service from Avaki Studio, the schema is generated at the time of creation. The generate schema utility is for database operations and data services created in the web UI or the CLI.
- Execute a data service, a database operation, or a virtual database operation.
- Add users or groups to access control lists (ACLs), remove users or groups from ACLs, and modify ACLs. For details, see "Modifying ACLs" on page 98.
- Add, remove and modify the attributes of objects in the data catalog. For details, see "Managing attributes" on page 101.
- View the dependencies for a database operation, a virtual database operation, or a data service. Dependencies are other database operations or data services that receive output from the current view model or provide input to it.

Managing the data catalog

You can use the Data Catalog view to manage the data catalog. Select an item of interest in the catalog and right-click for a context menu. You can:

- Display information about items in the catalog, including their type (such as data service or database operation), qualified name, SQL statement (for a database operation), parameters, data expiration for caching, and run-as user.
- Add and delete categories in the data catalog.
- Assign objects in the data catalog to categories.
- Refresh the contents of the data catalog to pick up changes such as new categories, data services, or database operations.

The palette

The palette, a column of buttons between the navigator and the view model pane, is a set of tools for working on view models:

Select. If the Operators are open, you'll notice that there are *two* Select buttons: one at the top, and one among the Operators. Click the top **Select** button when you want to select something in the view model.

Connection. To draw arrows that show how data will flow from one element in the view model to another, click **Connection**. Next, click on the input source or operator that data will flow *from*. Then click on the operator or result that data will flow *to*.

Input Sources. Input Sources is a collapsible list—click it to open it (which exposes the **Input Source** button) or collapse it. To create an input source, click the **Input Source** button, then click the place in the view model pane where you want the new input source to appear. (You can also create Input Source elements by dragging and dropping items that provide input data from the data catalog pane into the editor.) For details on Input Sources, see "Input Source" on page 125.

Operators and **Advanced Operators.** Like Input Sources, Operators and Advanced are collapsible lists—click one to open or collapse it. To add an operator or an advanced operator to the view model, click the operator's button, then click the point in the view model pane where you want the operator to appear. For details on all the operators, see Chapter 7, "Operator reference".

The view model editor

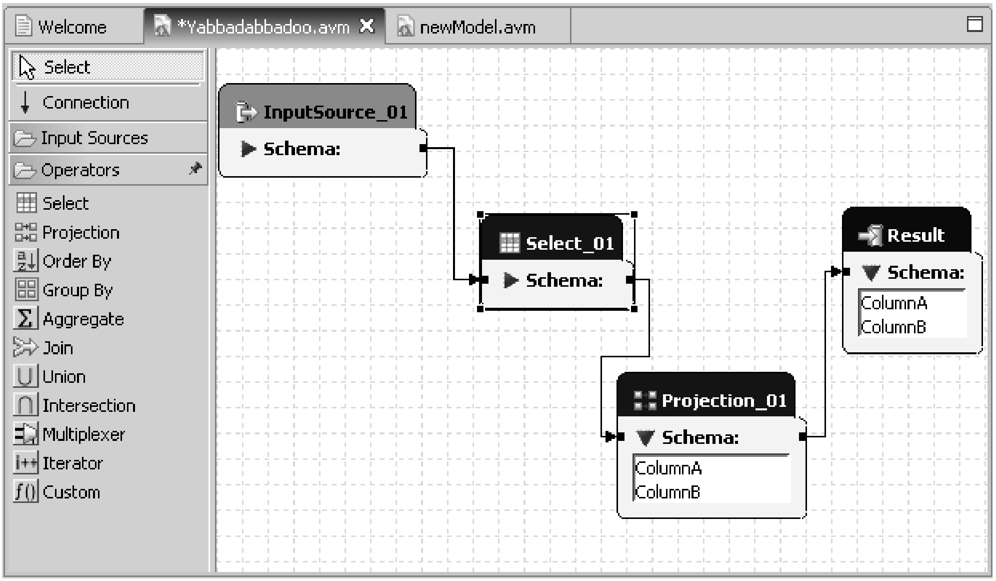
The view model editing pane lets you create, display, and manipulate the view models in the current project. For detailed information on creating and editing view models, see Chapter 4, “The view model editor”.

Adding and removing objects

Use the buttons in the palette to add objects to the view model. (See “The palette,” above.) To remove an object, select it and click right. Choose **Delete** from the menu.

View model tabs

If you create multiple view models in a project, you’ll see a tab for each view model above the view model pane, as shown here. An asterisk indicates that the view model has been modified since it was saved.

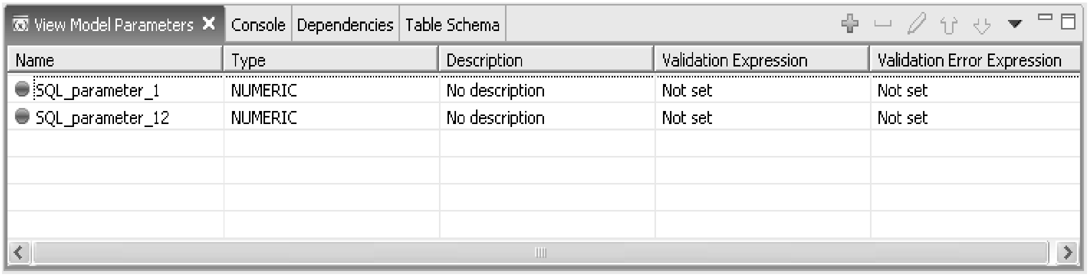


The metadata model editor

The metadata model editing pane lets you create, display, and manipulate the metadata (schema) models in the current project. For detailed information on creating and editing metadata models, see Chapter 5, “Metadata modeling”.

The bottom pane

The bottom pane of the Avaki perspective window displays view model parameters, the console, dependencies, and table schemas.



Name	Type	Description	Validation Expression	Validation Error Expression
● SQL_parameter_1	NUMERIC	No description	Not set	Not set
● SQL_parameter_12	NUMERIC	No description	Not set	Not set

Table schema view

The table schema view displays the schema—the column definitions—of a database table or an Avaki SQL view. To display a schema, select a table or SQL view in the data catalog view, click right to open the popup menu, and select **View Table Schema**.

Dependencies view

The Dependencies view displays information about data services and database operations that provide input to or receive output from another data service or database operation. To display dependencies, select a data service or database operation in the data catalog view, click right to open the popup menu, and select **View Dependencies**.

Console view

The console view displays results and execution traces when you test a view model. To display the console view, pull down the Window menu and select **Show View > Console**. To remove output from the console, click the Clear Console button .

To test a view model, pull down the Avaki menu and select **Execute Model**. For more information, see “Testing the view model” on page 50.

View Model Parameters view

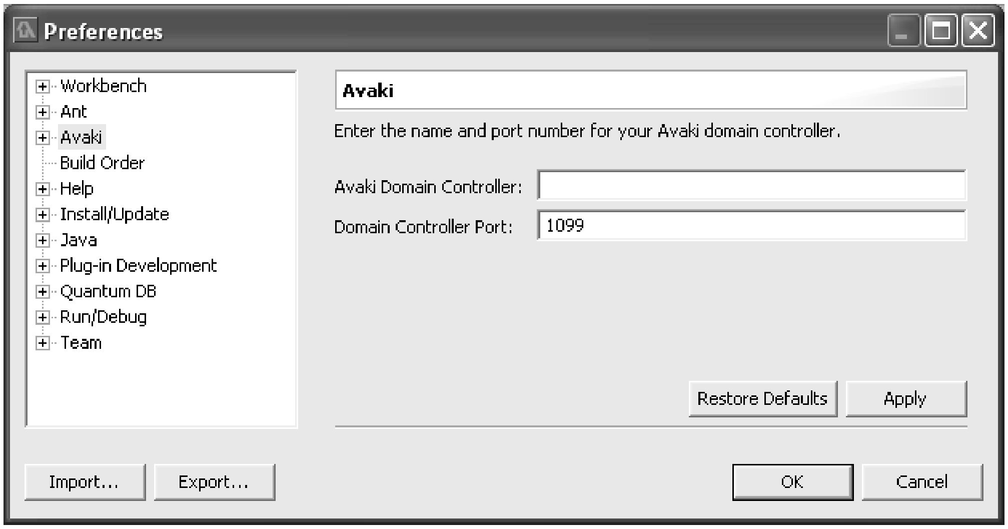
The View Model Parameters tab lets you create and manipulate global parameters for the current view model. Use the buttons (upper right) to add, remove, or modify global parameters, or to move parameters up or down in the list. For details on working with parameters, see "Using global parameters" on page 63.

Setting preferences

This section explains how to set preferences in Avaki Studio. Preferences specify which Avaki grid server Studio connects to (when you browse for an input source or when you test a view model, for example) and how Studio logs in to the Avaki domain. You can also control the color and font Studio uses to display your projects.

Step 1 In the Window menu, select **Preferences**. The Preferences dialog opens.

Step 2 In the browsing pane on the left, select **Avaki**. The server preferences dialog appears.



Avaki

Enter the name and port number for your Avaki domain controller.

Avaki Domain Controller:

Domain Controller Port: 1099

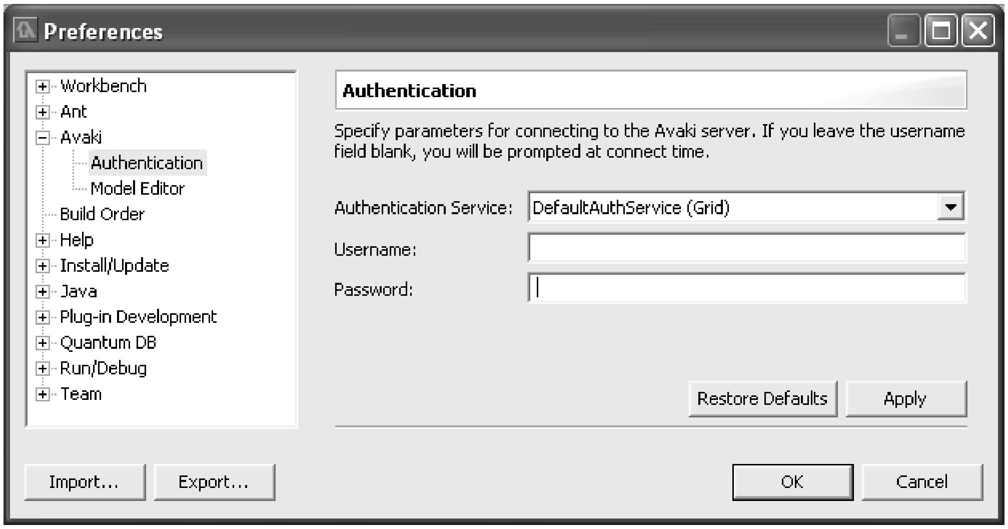
Step 3 In the Avaki Domain Controller field, enter the name of the grid domain controller (GDC) for your Avaki domain.

Step 4 If the GDC does not use the default connect port, enter the GDC's connect port number in the Domain Controller port field.

Step 5 Click **Apply** to save your changes.

Note Be sure to click **Apply** on each page of the Preferences dialog when you make changes. If you don't, you'll lose any changes to that page.

Step 6 Expand the Avaki entry in the browsing pane and select Authentication. Studio connects to your GDC (if possible) and displays the authentication preferences dialog.



Authentication

Specify parameters for connecting to the Avaki server. If you leave the username field blank, you will be prompted at connect time.

Authentication Service: DefaultAuthService (Grid)

Username:

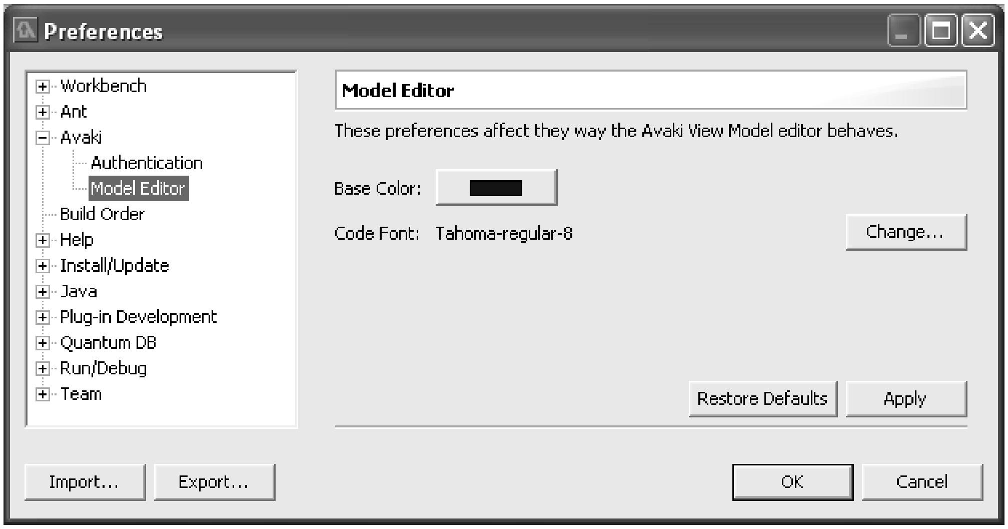
Password:

Note Fill in the Authentication Service, Username, and Password fields only if you want to use the same values every time Studio logs in to the grid. If you want to use two or more different user accounts, leave some or all of these fields empty. Studio will prompt you for any needed information when it logs in to the Avaki domain.

Step 7 Use the Authentication Service pull-down to select the authentication service for the user account you want Studio to use to log in to the data grid. (If Studio was unable to reach the GDC, the pull-down is empty.) Enter the name of the account in the Username field, and enter the account's password in the Password field.

Step 8 Click **Apply** to save your changes.

Step 9 Click **Model Editor** in the browser pane. Studio displays the Model preferences.



Model Editor

These preferences affect they way the Avaki View Model editor behaves.

Base Color:

Code Font: Tahoma-regular-8

Step 10 The Base Color field controls the color of the tabs at the top of elements in the view model. Click on the color sample to select from a palette of alternative colors.

Step 11 The Code Font field controls the font in which Studio displays JavaScript code. Click on the **Change** button to select from a list of alternative fonts.

Step 12 Click **Apply** to save your changes.

Step 13 Click **OK** to exit from the preferences dialog.

Working with Avaki Studio

This section provides an overview of one kind of workflow in Avaki Studio. The process of building and deploying a view model is as follows:

1. Create a new view model. The view model is where you specify how your data service combines and transforms data. For instructions on creating view models, see "Using the view model wizard" on page 43.
2. Create a database connector. To extract data from a relational database, your view model needs a pair of Avaki services: a database operation to select the data, and a database connector to access the database. For instructions on creating database connectors, see "Creating database connectors" on page 32.
3. Specify input sources for the view model. An Avaki data service—and the view model on which it is based—can use Avaki database operations, data services, and other objects as input sources. The process for specifying an input source is as follows:
   - Create database operations or data services. For instructions about creating database operations, see "Creating database operations" on page 38. For instructions about creating data services, see Chapter 3, "Creating view models".
   - Check dependencies. The Dependencies view displays information about data services and database operations that provide input to or receive output from another data service or database operation. For more information, see "Dependencies view" on page 22.
   - Create an input source and specify its schema. See "Configuring input sources" on page 44 and "Input Source" on page 125.
4. Specify parameters. You often will want to provide parameters to your model that affect the way it functions. Parameters to the view model are called global parameters because they are available to all of the operators and other elements within the model. For information about specifying, viewing, and editing global parameters, see "Using global parameters" on page 63.
5. If your input source has parameters, you must map them to global view model parameters. For more information, see "Mapping input parameters" on page 144.
6. Specify operators. Avaki Studio provides elements and operators you can use to manipulate data in a view model. The process for specifying an operator is as follows:
   - Create operators. A **Join** operator combines the result sets from two input sources. For more information, see "Joining result sets" on page 45. **A Projection** operator lets you modify a schema—you can remove columns, add computed columns, or apply formulas to alter existing column values. For more information, see "Using

Projection to combine columns" on page 47. For additional information about operators, see Chapter 7, "Operator reference".

7. Enter JavaScript expressions where needed, including control variables.
   - Most of the operators in Avaki Studio allow you to specify expressions that are evaluated at runtime to configure how your view model works. For more information, see "Using expressions within Avaki Studio" on page 66.
   - (Optional) Create a JavaScript include file. If you have JavaScript functions that you want to invoke in your view model, create a JavaScript include (.jsi) file. See "Using a .jsi file to enhance your model" on page 75.
   - A **Custom** operator can perform any operation that you define, using arbitrary JavaScript code that you supply. For information about writing custom operators, see "Custom" on page 111.
8. Test the view model. See "Testing the view model" on page 50.
9. Deploy the view model. See "Deploying the view model as a data service" on page 51.

Chapter 3

Creating view models

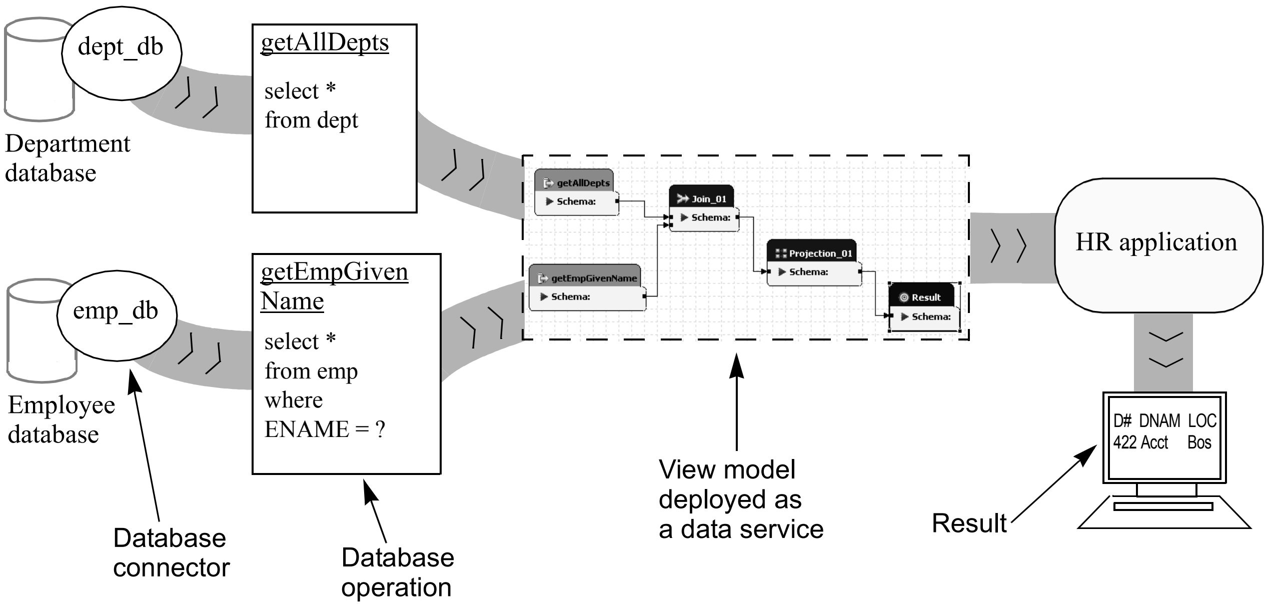
This chapter is a tutorial that walks you through the process of creating a view model in Avaki Studio and deploying your view model as a data service.

In this chapter:

The exercise

In this tutorial, we'll create a data service for the human resources department of a large organization. The HR coordinators who answer employees' questions about compensation, benefits, and transfers need quick access to information about each person they talk to, and the information is located in two different relational databases. We'll use Avaki Studio to set up the required Avaki database operations, construct and test a view model, and deploy the view model as a data service. An HR application accepts data from the data service and displays it.

This illustration shows the components involved in the workflow.



The HR data service

Given the name of an employee, the data service must return a result set with the following columns:

- DEPTNO: The employee's department number
- DNAME: The employee's department name
- LOC: The location at which the employee works
- NameNumberJob: The employee's name, employee number and job title
- MGR: The name of the employee's manager

- HIREDATE: The date the employee was hired
- SAL: The employee's salary
- COMM: The employee's commission rate

The database connectors

The required information resides in two databases. The following database connectors are needed:

- dept_db
  Connects to the department database, which has a table, EMP, with these columns: DEPTNO, DNAME, LOC
- emp_db
  Connects to the employee database, which has a table, DEPT, with these columns: EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, DEPTNO

The emp_db connector has been configured by the database administrator; we'll configure dept_db.

Tasks in this tutorial: the to-do list

To set up the HR data service, we must complete the tasks listed here. The tutorial takes about an hour to complete.

1. Configure the dept_db database connector.
2. Configure two database operations—one to extract data from each database.
3. Create a new view model.
4. Within the view model, set up two input sources—one for each of the database operations in task 2.
5. Create a view model parameter to accept the name of an employee and map the input source parameter to the new view model parameter.
6. Use a **Join** operator to combine the two input result sets.
7. Use a **Projection** operator to combine the EMPNO, ENAME, and JOB columns into a single NameNumberJob column.
8. Test the view model.
9. Deploy the view model as a data service in the Avaki data grid.

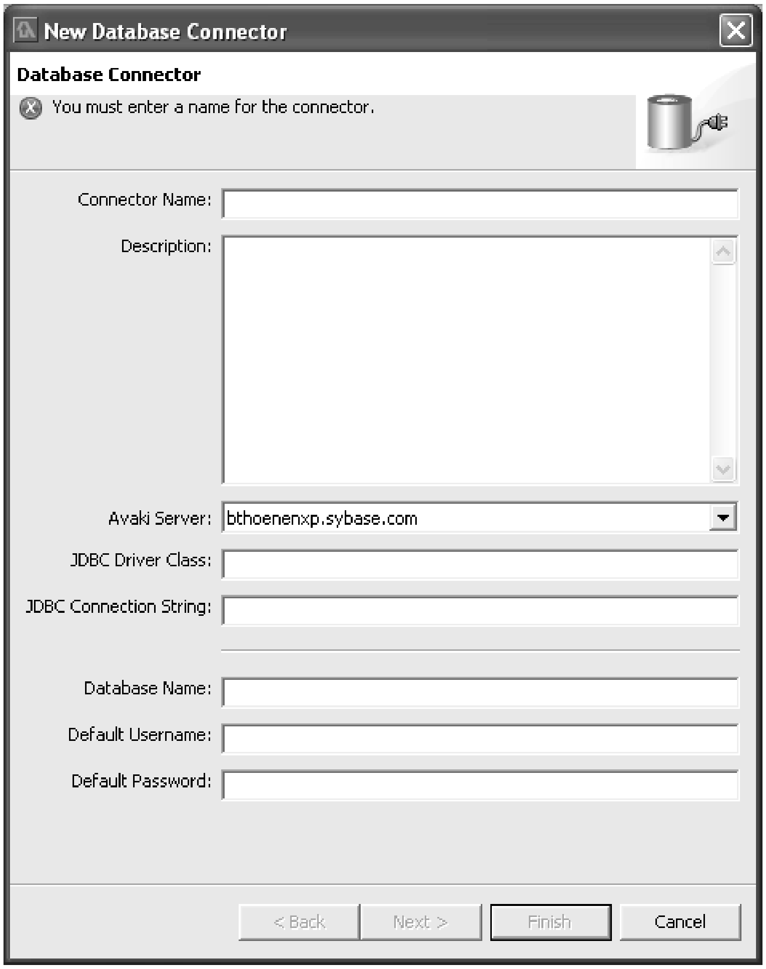
Creating database connectors

To extract data from a relational database, your view model needs a pair of Avaki services: a database operation to select the data, and a database connector to access the database. Database connectors are typically configured by database administrators.

Follow these steps to create a database connector in Avaki Studio.

Step 1 To connect Studio to a grid server, pull down the Avaki menu and select **Connect to Server**. Enter any requested information about the grid server.

Step 2 Pull down the File menu and select **New > Database Connector**. The New Database Connector window appears.


New Database Connector

Database Connector

You must enter a name for the connector.

Connector Name:	
Description:	
Avaki Server:	bthoenenxp.sybase.com
JDBC Driver Class:	
JDBC Connection String:	
Database Name:	
Default Username:	
Default Password:	

Step 3 Fill in the form:

- Connector Name: Enter a name for this database connector. For this tutorial, we'll use the name dept_db. **Note:** Do not include spaces in the name.
- Description: Optional. Enter some descriptive information about this database connector and the database it accesses.
- Avaki Server: Select the Avaki grid server on which this database connector will run.
- JDBC Driver Class: Enter the class name of your database JDBC driver. (The database connector won't work until this driver is copied into the drivers directory of the local Avaki installation directory.) For example:

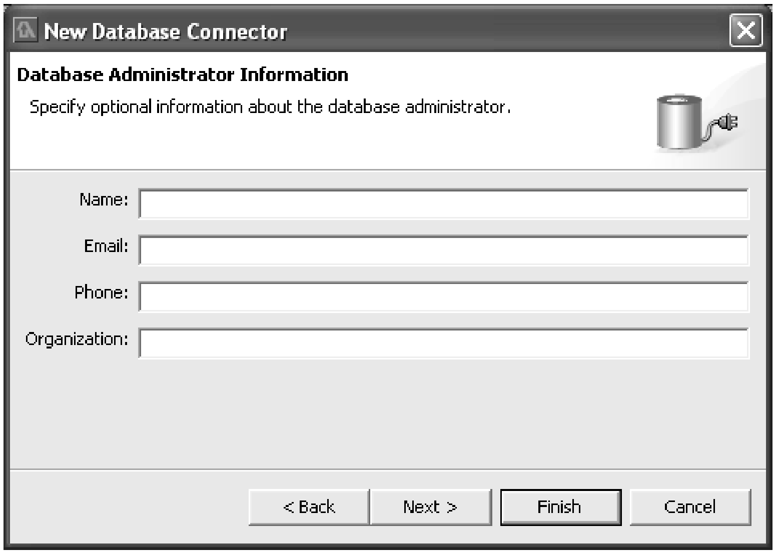
  ```
  oracle.jdbc.driver.OracleDriver
  ```

- JDBC Connection String: Enter the URL for your JDBC driver. For more information, see the documentation for your database. Here is a sample connection string:

  ```
  jdbc:oracle:thin:@gallium:1521:test1
  ```

- Database Name: Optional. Enter the name of the database.
- Default User Name: Enter the name of a database user account. The database connector will use this account to authenticate the database connection. All database operations will use this username when executed.
- Default Password: Enter the password to use to authenticate the database connection. All database operations will use this password when executed.

Step 4 Click **Next**. The second page of the form appears:



New Database Connector

Database Administrator Information

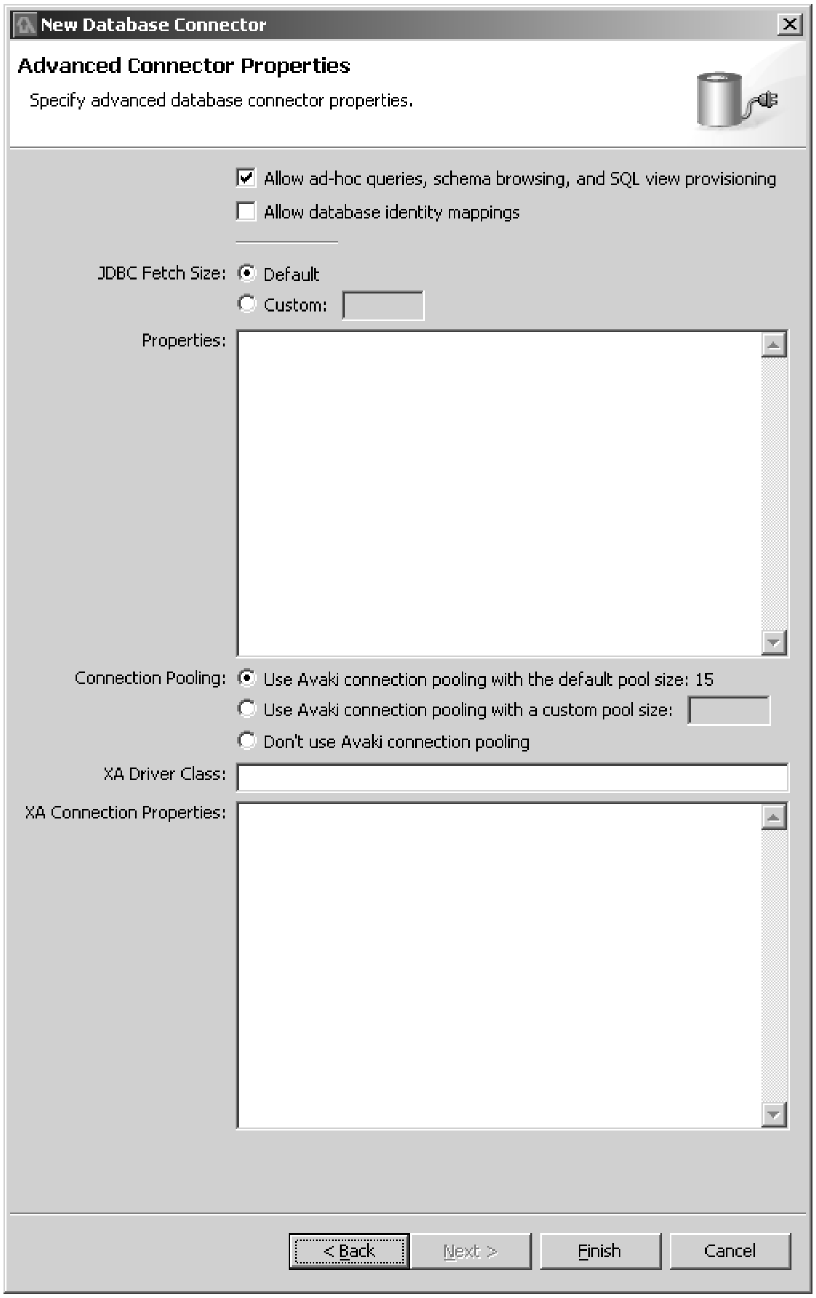
Specify optional information about the database administrator.

Name:	
Email:	
Phone:	
Organization:	

Step 5 Fill in the form:

- Name: Optional. Enter the database administrator's first and last name.
- Email: Optional. Enter the database administrator's email address.
- Phone: Optional. Enter the database administrator's telephone number.
- Organization: Optional. Enter the name of the department or group to which the database administrator belongs.

Step 6 Click **Next** to display the next page of the new database connector form.


New Database Connector

Advanced Connector Properties

Specify advanced database connector properties.

[✓] Allow ad-hoc queries, schema browsing, and SQL view provisioning

[ ] Allow database identity mappings

JDBC Fetch Size: [● Default] ○ Custom:

Properties:

Connection Pooling: [● Use Avaki connection pooling with the default pool size: 15] ○ Use Avaki connection pooling with a custom pool size: ○ Don't use Avaki connection pooling

XA Driver Class:

XA Connection Properties:

Step 7 Fill in the form:

- Allow ad-hoc queries, schema browsing, and SQL view provisioning: Select this option if you want to enable users to perform direct SQL queries against the database, browse the database's metadata, or provision SQL views. Ad-hoc queries must run through an existing Avaki database connector. Ad-hoc queries can be thought of as single-use database operations. You can run an ad-hoc query using either the CLI or the JDBC driver. For information about using the CLI to run an ad-hoc query, see the *Sybase Avaki EII Command Reference*. For information about using a JDBC driver to run an ad-hoc query, see the *Sybase Avaki EII API Guide*.

- Allow database identity mappings: Check this box if database identity mappings are allowed on this database connector. A database identity mapping is a special-purpose user alias. It allows operations performed by some Avaki user on a particular database connector to be executed in the name of an alternate user/password combination. For details about configuring database identity mappings, see the *Sybase Avaki EII Administration Guide*.
- JDBC fetch size: This parameter can be used to fine-tune performance of database operations. When an application uses the JDBC driver to execute a database operation, it typically processes the rows that are returned one after another, but the driver applies a buffering optimization by fetching rows in batches; the fetch size is the number of rows to be fetched in such a batch. In most circumstances, the driver's default fetch size will be optimal, so you'll want to keep the Default setting. But if you decide that database operations executed through this database connector should generally use a nondefault fetch size, enter the relevant number of rows in the Custom field. **Note:** you can also set the fetch size for individual database operations. See the *Sybase Avaki EII Provisioning and Advanced Data Integration Guide* for details.
- Properties: Optional. Specify any connection properties that are required for your database. For information about database-specific properties, see your database documentation.
- Connection pooling: Specify whether to enable Avaki connection pooling. When connection pooling is enabled, database connections can be reused, which typically improves performance for JDBC applications. By default, Avaki connection pooling will keep up to 15 connections open to a back-end database. If your driver has built-in connection pooling, it may not be necessary to use Avaki connection pooling. Select one of the following options:
  - Use Avaki connection pooling with the default pool size (15).
  - Use Avaki connection pooling with a custom pool size: Enable connection pooling and specify the desired connection pool size.
  - Don't use Avaki connection pooling.
- XA driver (optional): Specify an XA driver class if you plan to use this database connector to support distributed transactions. This class must reside in the JAR file

for the database vendor's JDBC driver. (For purposes of the tutorial, leave this field blank.)

These XA drivers have been tested with Avaki:

- For Sybase ASE: com.sybase.jdbc3.jdbc.SybXADataSource
  (ASE 15.0 with jConnect 6.05)
- For Oracle 10g: oracle.jdbc.xa.client.OracleXADataSource
  (Oracle 10g release 10.1.0.2.0 with JDBC driver version 10.2.0.1.0)
- For MySQL: com.mysql.jdbc.jdbc2.optional.MysqlXADataSource
  (MySQL 5.0 with MySQL Connector/J 5.0)

- XA connection properties (optional): Some XA connection properties might be required by your XA driver. (For purposes of the tutorial, leave this field blank.) Here are some that are typically specified:

  - For Sybase ASE:

    ServerName=<database-host-name>
    PortNumber=<database-port> (e.g. 5000)
    ResourceManagerType=2
    ResourceManagerName=connection
    DatabaseName=<database-name>
    NetworkProtocol=Tds
    User=<db-user-name>
    Password=<db-user's-password>

  - For Oracle 10g:

    URL=jdbc:oracle:thin:@<db-host-name>:<db-port>:<db-name>
    User=<db-user-name>
    Password=<db-user's-password>

  - For MySQL:

    URL=jdbc:mysql://<db-host-name>[:<db-port>]/<db-name>
    User=<db-user-name>
    Password=<db-user's-password>

Note For details on configuring your XA-compliant JDBC driver, including the particular XA connection properties to use, refer to the documentation for the driver.

Step 8 Click **Finish** to save the settings you've entered. Avaki Studio creates the database connector, tests it, and displays the results.

You can find the new database connector in the data catalog under Categories/ViewLibrary/DatabaseServices/<dbconn>/<domain>.<dbconn>.

Later, if you want to modify the database connector, find it in the data catalog view, right-click it, and select **Edit** from the context menu. Studio displays the database connector wizard, which lets you edit all the properties described in the steps above.

Creating database operations

An Avaki data service—and the view model on which it is based—can use Avaki database operations as input sources. Each database operation relies on an Avaki database connector to connect to a relational database, from which the database operation extracts data; the data is returned in result set format.

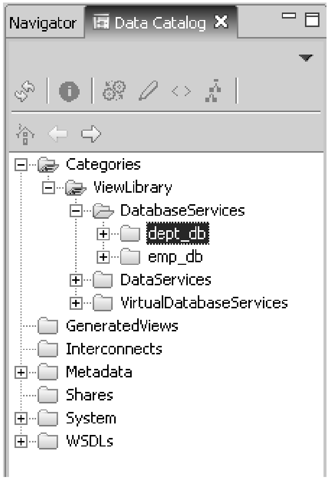
Before you try to configure a database operation, make sure that a database connector has been created for the appropriate database. Instructions for configuring a database connector can be found in the previous section, "Creating database connectors" on page 32.

Follow these steps to create a database operation in Avaki Studio.

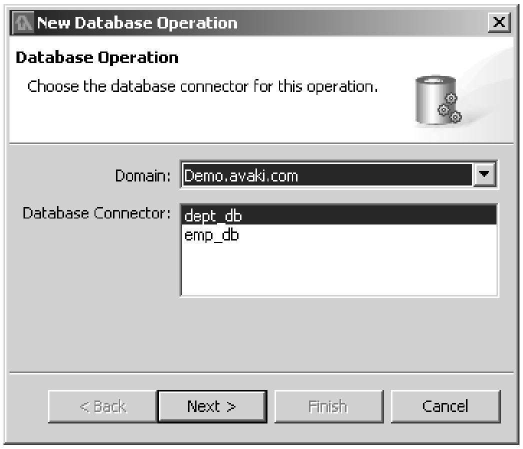
Step 1 To connect Studio to a grid server, pull down the Avaki menu and select **Connect to Server**. Enter any requested information about the grid server.

Step 2 Pull down the Window menu and select **Show View > Data Catalog**. The data catalog view appears in the left pane.

Step 3 (Optional) To locate your database connectors in the data catalog view, expand Categories > ViewLibrary > DatabaseServices. The database connectors appear as folders—here they're called dept_db and emp_db.



Step 4 To launch the Database Operation Wizard, pull down the File menu and select **New > Database Operation**.


New Database Operation

Database Operation

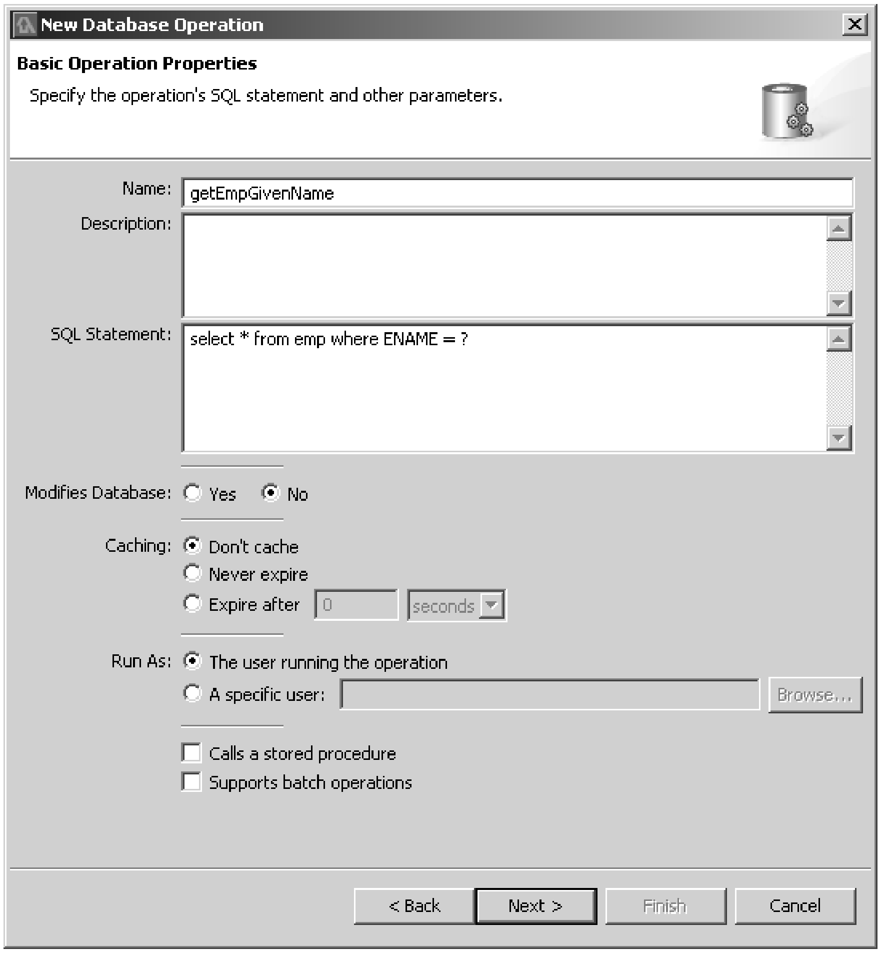
Choose the database connector for this operation.

Domain: Demo.avaki.com

Database Connector: dept_db
emp_db

Step 5 From the Domain pull-down, select the Avaki domain in which the database connector you want to use resides.

Step 6 In the Database Connector field, select a database connector, then click **Next**.


New Database Operation

Basic Operation Properties

Specify the operation's SQL statement and other parameters.

Name: getEmpGivenName

Description:

SQL Statement: select * from emp where ENAME = ?

Modifies Database: Yes [✓ No]

Caching: [✓ Don't cache] Never expire Expire after 0 seconds

Run As: [✓ The user running the operation] A specific user:

Calls a stored procedure

Supports batch operations

Step 7 In the Name field, enter a name for the database operation you're creating.

Step 8 (Optional) In the Description field, enter a description of the database operation.

Step 9 In the SQL Statement field, enter the query that this database operation will execute:

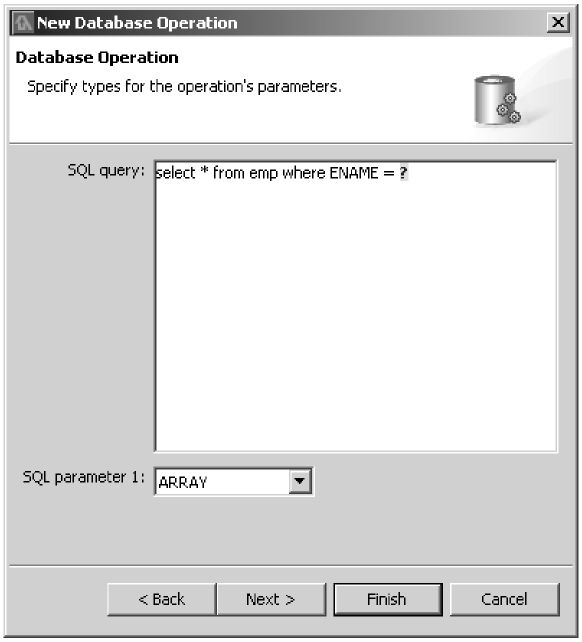
- For getAllDepts, enter `select * from dept`
- For getEmpGivenName, enter `select * from emp where ENAME = ?`

Step 10 In the Modifies Database field, leave the No button checked. (Click the Yes button when you create a database operation that performs an update.)

Step 11 In the Caching field, choose the caching behavior for the results of this database operation. (It's ok to keep the default, "Don't cache.")

Step 12 (Optional) In the Run as field, specify the user account under which this database operation will run.

Step 13 Click **Next**. (Do not check the "Calls a stored procedure" or "Supports batch operations" boxes.)



Database Operation

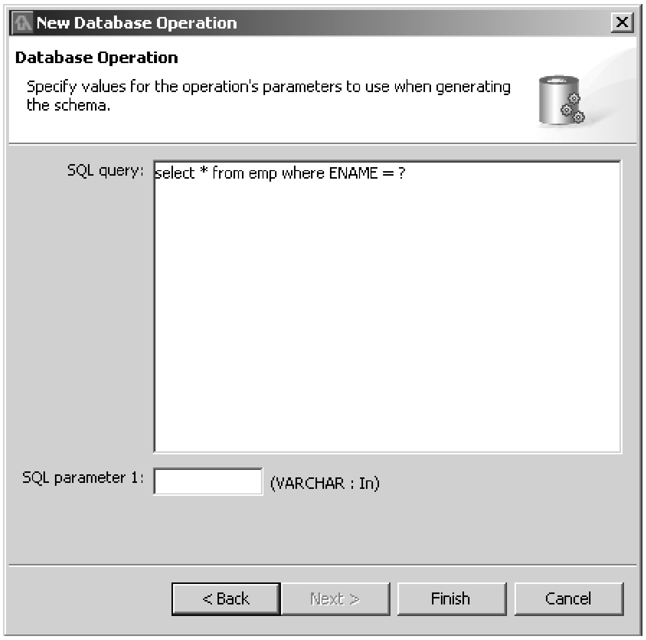
Specify types for the operation's parameters.

SQL query: select * from emp where ENAME = ?

SQL parameter 1: ARRAY

Step 14 For the EMP database operation, which has a parameter, use the pull-down menu to choose the data type `VARCHAR`.

Step 15 Click **Next**.



Database Operation

Specify values for the operation's parameters to use when generating the schema.

SQL query: select * from emp where ENAME = ?

SQL parameter 1: (VARCHAR : In)

Step 16 For the EMP database operation, which has a parameter, enter a value that Studio can use when it executes the database operation to generate the schema. (The value might be the name of an employee in the database.)

Step 17 Click **Finish**. Studio creates the new database operation. (You must refresh the data catalog view to see it.)

Repeat this procedure to create the DEPT database operation, which has no parameters.

You can also create database operations using Avaki's web UI or CLI—see the *Sybase Avaki EII Provisioning and Advanced Data Integration Guide* for instructions.

Note If your database operation includes a SQL statement that involves aggregate functions such as SUM, you must use an alias for any column names to which the function refers. The alias must follow the rules for valid JavaScript identifiers:

- Every alias must begin with a letter, an underscore (_), or a dollar sign ($).
- Every character after the first character may be a letter, number, underscore (_), or dollar sign ($).
- All characters should be valid ASCII characters.
- Do not use a JavaScript reserved word as an alias.

For example, in the following SQL statement, `dollarsum` is an alias for the sum of the column named `DOLLARS`.

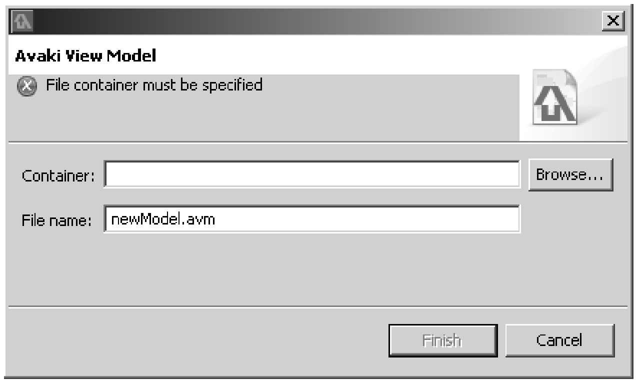
```
SELECT sum(DOLLARS) dollarsum, CUSTOMERID FROM sales GROUP
BY CUSTOMERID ORDER BY CUSTOMERID
```

Using the view model wizard

If you haven't created a Studio project yet, see "Creating a project" on page 14. To complete this procedure, you must have a project to which you can add your view model.

The view model is where you specify how your data service combines and transforms data. To set up a new view model, follow these steps.

Step 1 Pull down the File menu and select **New > View Model**. The view model wizard appears.



Avaki View Model

File container must be specified

Container:

File name: newModel.avm

Step 2 In the Container field, enter the name of an existing project directory for this view model, or use the **Browse** button to search for one.

Step 3 (Optional) Enter a new name in the File name field.

Step 4 Click **Finish**. Studio displays the new view model, which contains only a **Result** element.

Step 5 Use the mouse to drag the **Result** element to the right side of the view model pane.

Configuring input sources

In our view model, **Input Source** elements represent the database operations that provide input to the data service. Follow these steps to configure input sources.

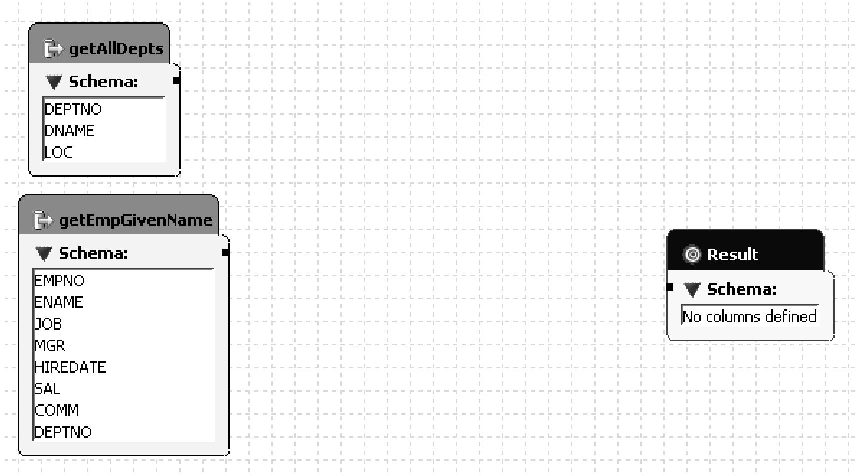
Step 1 To create an **Input Source** element, drag the database operation from the data catalog view into the view model pane. Studio creates an **Input Source** element with the same name as the database operation.

Step 2 Double-click the **Input Source** element to open the Input Source Properties dialog.

Step 3 Click in the Name field and change the name to the name of the database operation—getAllDepts or getEmpGivenName.

Step 4 If the dialog reports any unmapped parameters, click **Edit Values** to display the Parameter Properties dialog. You must map the getEmpGivenName database operation's input parameter, EmployeeName, to a new global parameter of type `VARCHAR`. For instructions on mapping parameters, see "Input Source Properties" on page 126.

Repeat this procedure to configure an **Input Source** element for another database operation. When you've configured **Input Source** elements for both the getAllDepts and GetEmpGivenName database operations, your view model looks like this.



Joining result sets

To create a **Join** operator to combine the result sets from the two input sources, follow these steps.

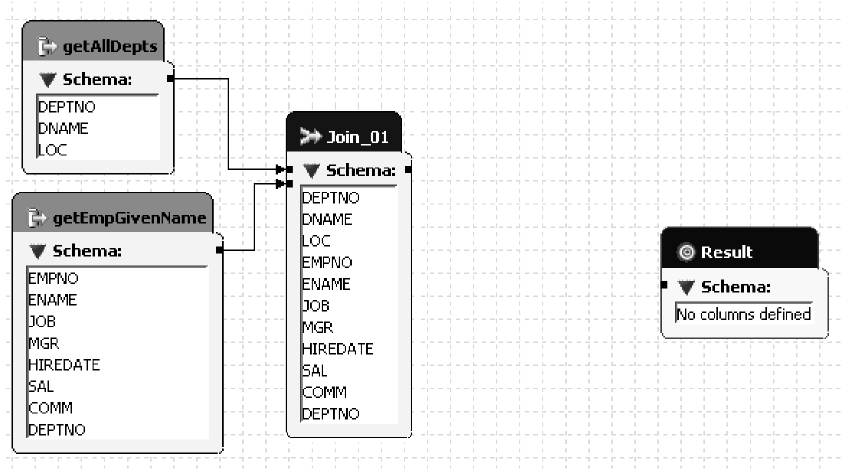
Step 1 Click **Join** in the Operators section of the palette (immediately to the left of the view model pane).

Step 2 Click in the view model pane just to the right of the two **Input Source** elements. A **Join** operator appears.

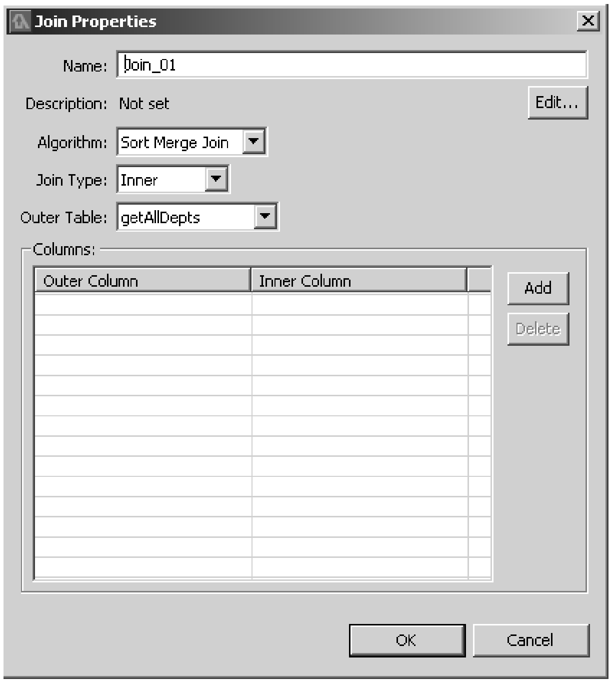
Step 3 Before you can configure the Join operation, you must connect it to the input sources. Click **Connection** in the palette.

Step 4 Click the **getAllDepts** input source, then click the new **Join** operator. A line connecting the two elements appears in the view model.

Step 5 Click **Connection**, **getEmpGivenName**, and the **Join** operator to connect the second input source. The connecting line must touch the unused connection point on the **Join** operator. Now the view model looks like this.



Step 6 Double-click the **Join** operator to open the Join Properties dialog.


Join Properties

Name: Join_01

Description: Not set

Algorithm: Sort Merge Join

Join Type: Inner

Outer Table: getAllDepts

Columns:

Outer Column	Inner Column

Step 7 Select the Sort Merge Join algorithm, the Inner join type, and set the outer table to getEmpGivenName.

Step 8 In the Columns area of the properties dialog, specify one or more pairs of columns whose values will be tested for equality. Each pair must include one column from the outer table and one from the inner table. In this case, the common column is called DEPTNO in both tables. Click **Add** to add a column. Use the pull-down menus to select DEPTNO for both the inner and outer columns

Step 9 Click **OK**.

For more information on the **Join** operator, see "Join" on page 154.

Using Projection to combine columns

To create a **Projection** operator, follow these steps.

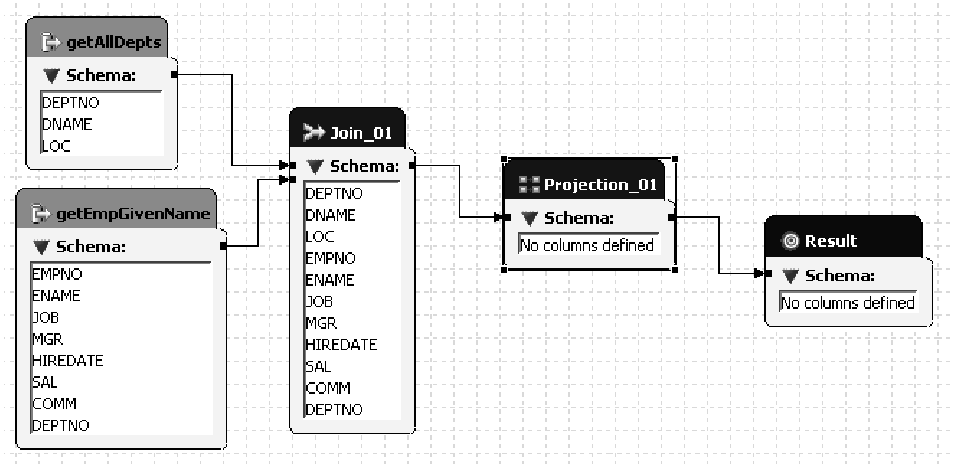
Step 1 Click **Projection** in the Operators section of the palette (immediately to the left of the view model pane).

Step 2 Click in the view model pane just to the right of the **Join** operator. A **Projection** operator appears.

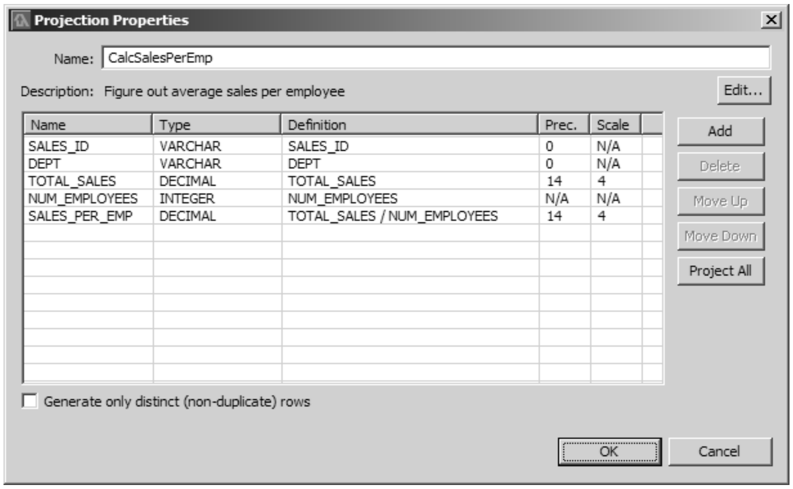
Step 3 Before you can configure the **Projection** operator, you must connect it to its input, the **Join** operator. Click **Connection** in the palette.

Step 4 Click the **Join** operator, then click the new **Projection** operator. A line connecting the two operators appears in the view model.

Step 5 To connect the **Projection** operator to the **Result** element, click **Connection** again, click the **Projection** operator, then click the **Result** element. Now the view model looks like this.



Step 6 Double-click the **Projection** operator to open the Projection Properties dialog.



Name: CalcSalesPerEmp

Description: Figure out average sales per employee

Name	Type	Definition	Prec.	Scale
SALES_ID	VARCHAR	SALES_ID	0	N/A
DEPT	VARCHAR	DEPT	0	N/A
TOTAL_SALES	DECIMAL	TOTAL_SALES	14	4
NUM_EMPLOYEES	INTEGER	NUM_EMPLOYEES	N/A	N/A
SALES_PER_EMP	DECIMAL	TOTAL_SALES / NUM_EMPLOYEES	14	4

☐ Generate only distinct (non-duplicate) rows

Step 7 Click **Project All** to carry in the columns from the input result set. Studio creates and displays an output column for each column in the input.

Step 8 Click **Add** to add a new column.

Step 9 Click in the Name field of the new column and replace the default name with **NameNumberJob**.

Step 10 In NameNumberJob's Definition field, enter the following:

```
ENAME + " (" + EMPNO + "), " + JOB
```

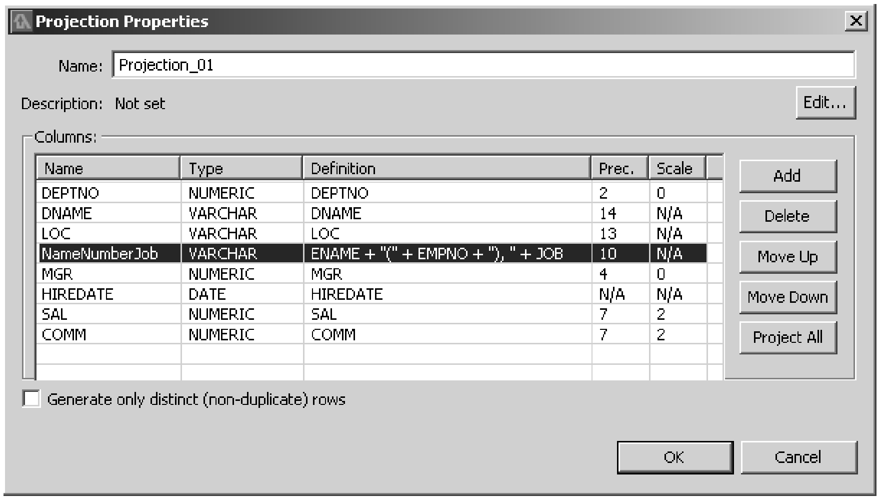
Step 11 Leave the Type field set at `VARCHAR`, the default.

Step 12 Set the precision for NameNumberJob to **50**.

Step 13 Select the following columns and use the **Delete** button to remove them:

- EMPNO
- ENAME
- JOB
- DEPTNO (remove only the bottom instance of DEPTNO)

Step 14 Use the **Move Up** button to move the NameNumberJob column the fourth position, as shown here:


Projection Properties

Name: Projection_01

Description: Not set

Columns:

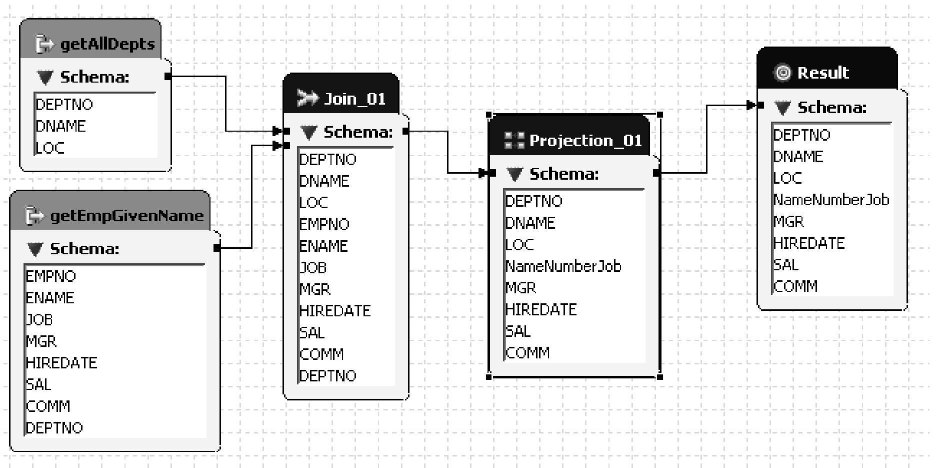
Name	Type	Definition	Prec.	Scale
DEPTNO	NUMERIC	DEPTNO	2	0
DNAME	VARCHAR	DNAME	14	N/A
LOC	VARCHAR	LOC	13	N/A
NameNumberJob	VARCHAR	ENAME + "(" + EMPNO + "), " + JOB	10	N/A
MGR	NUMERIC	MGR	4	0
HIREDATE	DATE	HIREDATE	N/A	N/A
SAL	NUMERIC	SAL	7	2
COMM	NUMERIC	COMM	7	2

Step 15 Click the box for "Generate only distinct (non-duplicate) rows".

Step 16 Click **OK** to save the Projection Properties.

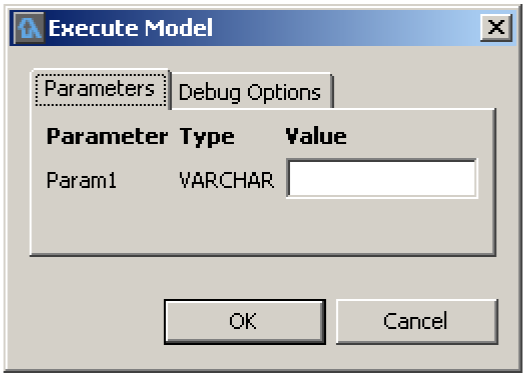
Step 17 Select **File > Save** to save your view model.

This is the completed view model:



Testing the view model

When your view model is complete, you can test it by selecting **Execute Model** from the Avaki menu. The Execute Model dialog appears.



Parameter	Type	Value
Param1	VARCHAR	

The Parameters tab provides fields where you enter parameter values required for the test execution.

Debug Options tab provides checkboxes that enable you to control which debug options are displayed.

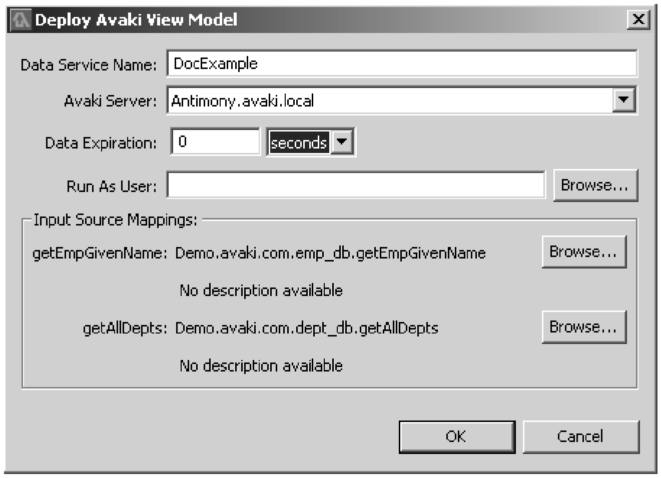
- Enable tracing of model execution: Controls whether information about the text execution will be displayed. If you check the box, the trace information is displayed in the Console view in the bottom pane.
- Include each operator's row output data: Controls whether each operator's row output data is displayed. If you check the box, the row output data is displayed.
- Display debug console view: Controls whether the debug console is displayed. If you check the box, the debug console is displayed.

Click **OK** to start the test execution.

Deploying the view model as a data service

Follow these steps to deploy your view model. After it's deployed, you can find it in the data catalog viewer under Categories > ViewLibrary > DataServices.

Step 1 Right-click in the view model pane and select **Deploy Data Service** from the popup menu. The Deploy Avaki View Model dialog appears.



Data Service Name: DocExample
Avaki Server: Antimony.avaki.local
Data Expiration: 0 seconds
Run As User:

Input Source Mappings:
getEmpGivenName: Demo.avaki.com.emp_db.getEmpGivenName
No description available
getAllDepts: Demo.avaki.com.dept_db.getAllDepts
No description available

Step 2 (Optional) Enter a name the Data Service Name field if you don't want to use the default name.

Step 3 (Optional) Select a grid server from the Avaki Server pull-down if you don't want to use the default server.

Step 4 (Optional) To control whether and for how long the results of this data service are cached, enter a value in the Data Expiration field and select a unit from the pull-down menu. There are two special values:

- 0 (the default) means don't cache the results of this data service
- –1 means cached results of this data service never expire

Step 5 In the Run As User field, enter or browse for a user account under which the data service will run.

Step 6 The Input Source Mappings area shows the database operations and data services that are configured to provide input to this data service. You can use the **Browse** button to browse and select one or more alternative input sources (one for each Input Source element in your view model).

Step 7 Click **OK** to save the data service configuration and deploy the new data service.

Thinking ahead: load sharing for data service execution

By default, a data service runs on the grid server where it was deployed. If you want data service execution to be shared among several grid servers, you or your grid administrator must configure execution service pooling. For instructions, see the *Sybase Avaki EII Administration Guide*.

Chapter 4

The view model editor

Consult this chapter for general information on using Avaki Studio’s view model editor to create view models.

In this chapter:

- “Common features of view model components” on page 54 describes features common to all of the elements—input sources, operators, and result elements.
- “Using global parameters” on page 63 describes how to add parameters to your model.
- “Using expressions within Avaki Studio” on page 66 explains how to manipulate data within your model’s operators using scripting expressions.
- “The expressions menu” on page 71 describes how Avaki Studio’s context-sensitive menu will help you easily configure your model.
- “Advanced topics” on page 75 discusses more esoteric topics such as using a .jsi file to enhance your model’s capabilities and tuning your model’s performance.

Common features of view model components

This section discusses features common to the various operators, input sources, and result elements.

Creating components

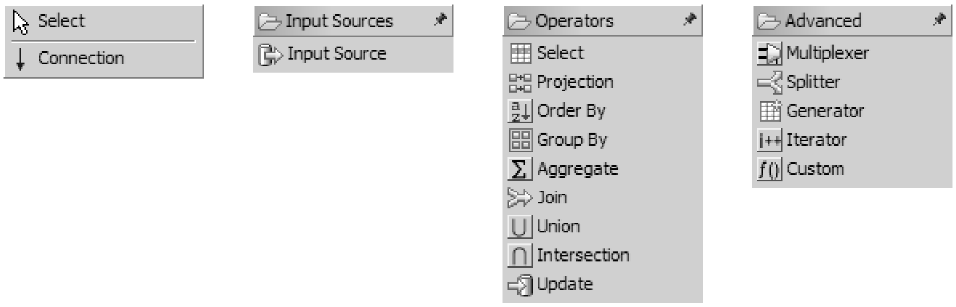
To build view models, you use the view model editor to create and manipulate objects that represent the elements in your model. These include input sources, various operators, and result elements. When you create a new view model, Studio automatically includes a **Result**. You create any other elements you require either by using the tools in the palette or by dragging and dropping from the data catalog view. Finally, you connect the elements together, using connection arrows, to define the flow of data within your model.

Creating elements using the palette

The palette contains tools for creating the elements that you can use in an Avaki view model. The palette works similarly to palettes in many graphical editor applications—you click in the palette to select the tool you wish to use and then you click in the document to place an instance of that object.

The tools in the palette are separated into four sections:

- The first group contains the **Select** tool, used to select elements in the view model in order to edit them (see "Selecting and moving elements" on page 56) and the **Connection** tool, used to connect elements together into a flow (see "Creating connections between elements" on page 57).
- The second group contains the **Input Source** tool.
- The third, the "Operators," group, contains basic relational algebra operations.
- The final group, "Advanced," contains more advanced operators.

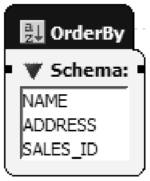


For example, to add an **Order By** operator to your model using the palette:

Step 1 Open an Avaki view model file (double-click a .avm file in the Navigator in the left pane) or create a new one (see "Creating a project" on page 14 for instructions).

Step 2 Select the **Order By** tool in the palette. Order By If the "Operators" section of the palette is collapsed, click to open it, exposing the **Order By** tool.

Step 3 Click in the view model editor pane to place an **Order By** operator.



Once the operator is in your model, you can move it around (see "Selecting and moving elements" on page 56), connect it to other elements in your model (see "Creating connections between elements" on page 57), or edit its properties (see "Properties dialogs" on page 58).

Creating Input Source elements using drag and drop

When you want to create an **Input Source** based on an item in the Avaki data catalog, you can do so by dragging and dropping an icon from the data catalog view directly into the view model editor.

Avaki Studio configures the **Input Source** element as much as possible, depending on the type of object you dropped into the view model. For example, if you drop in a database operation, Studio configures the **Input Source** element to point to that database operation, and not to have any associated transform. If, however, you drop in an XML file, Studio configures the resulting **Input Source** object to be a file type pointing to the path of your file, and having an XML-to-Result-Set transform.

You can also always construct an **Input Source** element by using the **Input Source** tool in the palette and configuring it manually. And note that you must use the Input Source tool in the palette to create input sources based on HTTP requests or web services. Refer to "Input Source" on page 125 for details on configuring input source elements.

To use drag and drop to create an Input Source element:

Step 1 Open an Avaki view model (double-click a .avm file in the Navigator in the left pane) or create a new one (see "Creating a project" on page 14 for instructions).

Step 2 Make sure you have the Avaki data catalog view open in the left pane. If it is not open, open it by choosing **Window > Show View > Data Catalog**.

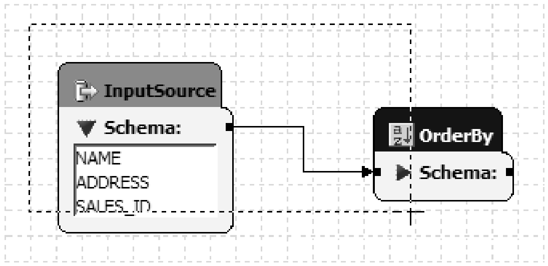
Step 3 Navigate in the data catalog view to the object you want to use as in input source. You can choose a database operation, a data service, or a file in the grid. (To create **Input Source** elements from HTTP requests or web services, use the tool in the palette, then see "Input Source" on page 125 for details on configuring the input source.)

Step 4 Drag the object from the data catalog view into the view model editor where you want the **Input Source** object to appear. Once the **Input Source** is in your model, you can move it around (see "Selecting and moving elements" on page 56), connect it to other elements in your model (see "Creating connections between elements" on page 57), or edit its properties (see "Properties dialogs" on page 58.).

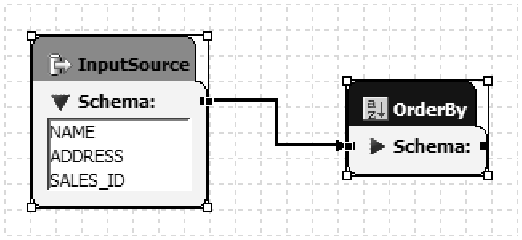
Selecting and moving elements

In order to manipulate the input sources, operators, and **Result** elements in your view model, you must first select them. To do so, you use the **Select** (arrow) tool in the palette. Select Note that this is the tool at the top of the palette, not the **Select** tool in the Operators section of the palette, which is used to filter out rows from a result set (see "Select" on page 165).

You can choose the **Select** tool by clicking it in the palette. The **Select** tool is also automatically selected after you use one of the other tools. To select an object, you simply click on it in the view model editor. To select multiple objects in your model, hold down the Shift key and click each object in turn. Alternatively, using the Select tool, click in the background of the editor pane and drag to create a marquee around the objects you want to select. When you have enclosed them all, release the mouse button and the objects will be selected.



Selected objects are indicated in the view model editor by a rectangle enclosing them as well as four selection "handles"—little boxes at the corners of the enclosing box. Selected connection arrows are thicker than unselected ones and have a selection handle at each end. You can click and drag to move selected elements around the view model. You might want to move elements around in order to make room for additional operators you want to add, for example, or to neaten up the model.



Creating connections between elements

To define the flow of data within a view model, you must connect the various elements together. The way elements are connected affects the way individual elements can be configured. For example, a **Join** operator needs to have two other elements connected to it as inputs before you can configure on which columns the join operates.

When you use the Connection tool, the order in which you select the elements you are connecting determines the direction of data flow—data will flow from the first element you select to the second.

To connect two elements in a view model:

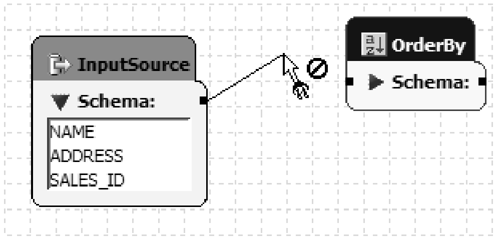
Step 1 Open an Avaki view model (double-click a .avm file in the Navigator in the left pane) or create a new one (see "Creating a project" on page 14 for instructions).

Step 2 Create at least two unconnected elements in the view model. To create an element, click on any item in the Operators section of the palette, then click in the editor window.

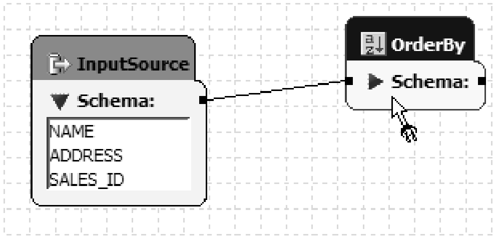
Step 3 Select the **Connection** tool ↓ Connection in the palette.

Step 4 Select the object the arrow should come *from*.

A connection line appears, attached to the output of the object you clicked and follows the cursor as you move it. The cursor has an plug-shaped icon attached to it indicating that you are making a connection. If the cursor isn't over something that can be connected, there is also a circle with a slash through it indicating this.



Step 5 Click on the object the arrow should go *to*. If this object has more than one input connection point, make sure that the arrow is snapped to the correct one before you click.



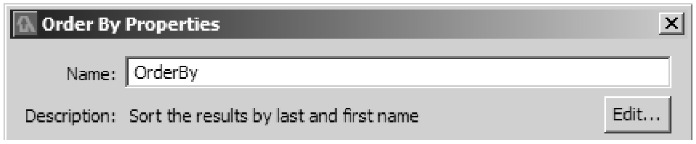
Step 6 If you decide you don't want to connect the objects after all, press **Escape** or click a different tool in the palette to abort the connection.

Properties dialogs

Each element in your view model can be configured by accessing its properties dialog. You edit the properties of an **Input Source** using the Input Source Properties dialog, those of an **Aggregate** operator using the Aggregate Properties dialog, and so on. To open these dialogs, do either of the following:

- Double-click the element in the view model editor
- Select the element, right-click to display the context menu, and choose **Edit**.

Properties dialogs all have two items in common, for setting the name and description of the element. (The only exception is the Result Properties dialog, which only has a description field since you cannot rename the **Result** element.)


Order By Properties

Name: OrderBy

Description: Sort the results by last and first name

Element names

The names of the elements in your view model are important because you refer to elements by name in the expressions you use in configuring the model. (For more information on expressions in Avaki Studio, see "Using expressions within Avaki Studio" on page 66.) Element names must be unique within each model, and must meet the same constraints as other variable names: they must start with a letter or underscore, and must contain only letters, numbers, and underscores. Other characters (including spaces, punctuation, accented characters, and so forth) are not allowed. If an element's name is invalid, or if the same name is used for more than one element, Studio displays a red border around the element in question and you will not be able to execute or deploy the model. See "Red borders: Errors in your view model" on page 60 for details.

To edit an element's name:

Step 1 Open the element's properties dialog box by double-clicking or by right-clicking and choosing the **Edit** command.

Step 2 Type a new name in the Name field.

Step 3 Click **OK** to close the properties dialog box.

Element descriptions

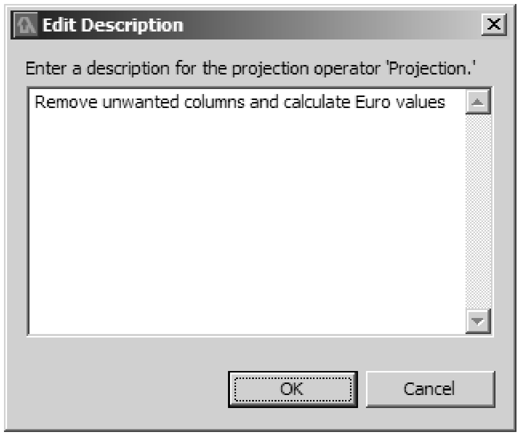
You can annotate your model by adding descriptions to the elements it contains. This can help make the model easier to understand, especially if multiple people work with it. When you move your mouse over an element in the view model editor, Studio displays the element's description in a tool tip. (If the element has an error, however, Studio displays the error in the tool tip instead of the description. See "Red borders: Errors in your view model" on page 60.)

To change an element's description:

Step 1 Open the element's properties dialog box by double-clicking or by right-clicking and choosing the **Edit** command.

Step 2 Click the **Edit…** button next to the description.

Step 3 Edit the description in the resulting Edit Description dialog.



Edit Description

Enter a description for the projection operator 'Projection.'

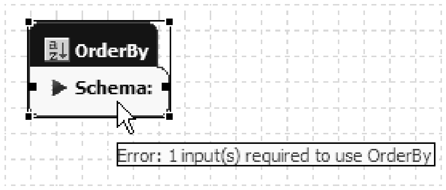
Remove unwanted columns and calculate Euro values

Step 4 Click **OK** to close the Edit Description dialog.

Step 5 Click **OK** to close the properties dialog box.

Red borders: Errors in your view model

Your view model may have errors in it. Some of these may be because certain elements are not properly configured or connected. If Studio detects an error, it notifies you by making the border around the affected elements red. You can see what the problem is by moving your cursor over the element in question—Studio displays the error message in a tool tip. You must resolve all of these errors before you can execute or deploy the view model. (Note that if an operator with an error is not actually connected to the **Result** element in any way, then you may still be able to execute and deploy the model even if it shows an error.)



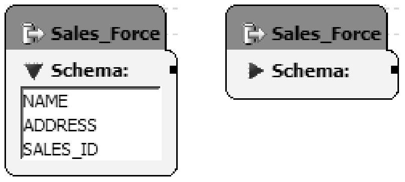
Error: 1 input(s) required to use OrderBy

Working with schemas

Many of the operators in Avaki Studio allow you to define or transform the schema of the data passing through them. The visual representation of each element in the view model can optionally display the schema of your data at that point in the flow.

Showing and hiding elements' schemas

Every element in your Avaki view model, from the input sources to the result, has a schema associated with it. Objects are created initially with the schema showing, but you may wish to hide them to make your model easier to read. You can show or hide the list of column names for any given element by clicking the triangle next to the "Schema" label.



Specifying schemas for operators

A schema is defined as an ordered list of columns where each column has the properties name, type, precision, and scale:

Name. Since they are used within expressions to access their data, names for columns must meet the same constraints as other variables: they must start with a letter or underscore, and must contain only letters, numbers, and underscore characters. Other characters (including spaces, punctuation, accented characters, and so forth) are not allowed. The name of each column in a given schema should be unique—having two columns in the same schema with the same name can cause Studio difficulty when trying to access the columns by name.

Note If your schema has column names that do not conform to these norms, you must use the methods in the ResultSet object to access them by position (using a 1-based index). We suggest setting up your database operations to alias any columns with questionable names so that they are not confusing within Studio.

Type. The data types of your columns correspond to common SQL types of the same name. The types available within Studio are ARRAY, BIGINT, BINARY, BIT, BOOLEAN, BLOB, CHAR, CLOB, DATE, DECIMAL, DISTINCT, DOUBLE, FLOAT, INTEGER, JAVA_OBJECT, LONGVARBINARY, LONGVARCHAR, NULL, NUMERIC, OTHER, REAL, REF, SMALLINT, STRUCT, TIME, TIMESTAMP, TINYINT, VARBINARY, VARCHAR, and ORACLE_CURSOR. For information about these types, consult any SQL reference.

Advanced Studio users will note that the standard JDBC type mappings are used to construct the Java representations of the column values that they can manipulate in Studio. For information about the Java types to which these map, refer to the Sun JDBC documentation:

`http://java.sun.com/j2se/1.4.2/docs/guide/jdbc/getstart/mapping.html#996857`

Precision and **scale.** The meaning of the precision and scale fields for a column depends on the column's data type. Precision and scale are not applicable to all column types. The table below gives the meaning of precision and scale for the Studio column types to which they apply.

Types	**Precision**	**Scale**
BINARY, LONGVARBINARY, VARBINARY	The size of the column data (in bytes)	*Not applicable*
CHAR, LONGVARCHAR, VARCHAR	The size of the column data (in characters)	*Not applicable*
DECIMAL, NUMERIC	The total number of digits in the column data (includes digits on both sides of the decimal point)	The number of digits to the right of the decimal point

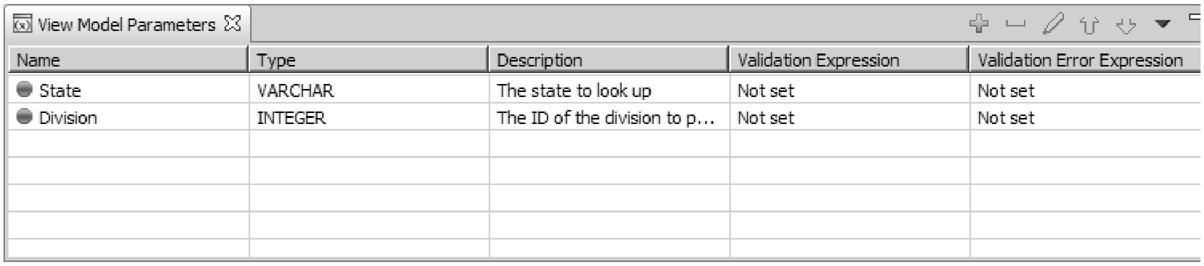
Using global parameters

Some view models obtain all necessary input from database operations, data services, and other input sources. However, you will often want to provide parameters to your model that affect the way it functions. For example, a model that processes sales figures might require a parameter specifying the sales region you're interested in.

Parameters to the view model are called global parameters because they are available to all of the operators and other elements within the model. This section shows how to display, add, modify, and delete global parameters.

Displaying global parameters

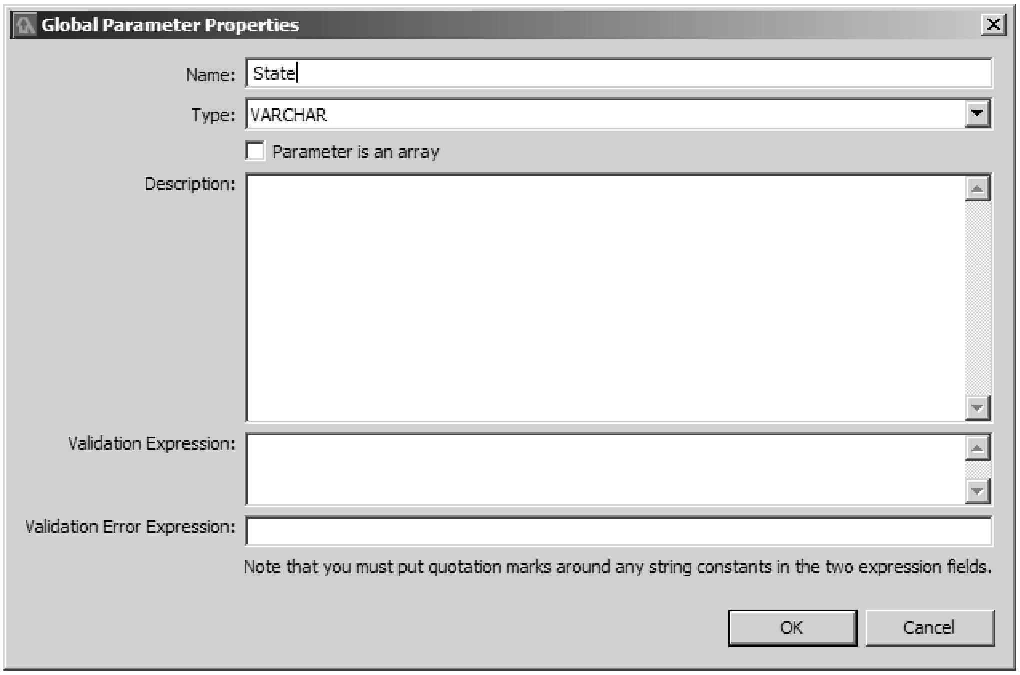
To see and manipulate your model's global parameters, use the View Model Parameters tab, which appears in the bottom pane in Studio. If it is not currently visible, you can access it using the **Window > Show View > View Model Parameters** command.



Name	Type	Description	Validation Expression	Validation Error Expression
State	VARCHAR	The state to look up	Not set	Not set
Division	INTEGER	The ID of the division to p...	Not set	Not set

Adding global parameters

To add a parameter to your model, click the **Add Parameter** button ✚ to open the Global Parameter Properties dialog.



Name: State

Type: VARCHAR

☐ Parameter is an array

Note that you must put quotation marks around any string constants in the two expression fields.

The fields are as follows:

- **Name:** A name for the parameter. Make sure it conforms to the variable naming conventions (see "Element names" on page 58).
- **Type:** Choose a type from the popup. Types are expressed in the SQL format, rather than as Java objects.
- **Parameter is an array:** Check this if you want the parameter to be an array of the selected type.
- **Description:** (Optional) Enter a description here, if you like.
- **Validation Expression:** (Optional) Enter a Boolean expression here that will be evaluated to determine the validity of the value passed for this parameter. If the expression evaluates to `false`, the data service will not execute and an error message will be returned (see the next field for details). Refer to the value of the variable being validated by its fully qualified name: `variables.global.<name>`. For example, to ensure that the length of the `State` variable is two characters, you could use this expression:

```
variables.global.State.length == 2
```

If you leave the Validation Expression field blank, Studio performs no validation for this parameter.

- **Validation Error Expression:** (Optional) enter an expression that evaluates to an error message. The error message is returned whenever an invalid parameter value causes the validation expression to evaluate to false. Enclose any strings in double quotes. Refer to the value of the variable being validated by its fully qualified name: `variables.global.<name>`. For example, for a variable called `State`, you might enter

```
"Please enter a 2-letter state code for the " +
    "variable \'State.\' You entered: " +
    variables.global.State
```

If you leave the Validation Error Expression field blank, Studio uses a generic error message.

Click **OK** to create the parameter.

Modifying global parameters

To edit global parameters, select one in the View Model Parameters tab and click the **Edit** button. You can also double-click the variable in the table. This opens the same Global Parameter Properties dialog, where you can edit the parameter's properties.

Deleting global parameters

To delete parameters, select them in the list and click the Delete Parameter button.

Reordering global parameters

You can change the order of parameters in the list. To move a parameter, select it and click the **Move Up** or **Move Down** button.

Note The order in which the parameters appear in the View Model Parameters tab is the same order in which they will appear to calling applications.

Using expressions within Avaki Studio

Most of the operators in Avaki Studio allow you to specify expressions that are evaluated at runtime in order to configure how your view model works. For the majority of cases, simple expressions that evaluate to a single value will suffice. Some of these are generic, while others must evaluate to a Boolean value. Studio's more advanced operators, however, require more complex, multi-line expressions.

Simple expressions

Some examples of expressions that evaluate to simple values are:

- `SALARY`
  "SALARY" is the name of a column in the single incoming result set.
- `Join01.LAST_NAME`
  "Join01" is one of the result sets that feed into the operator where the expression is defined, and "LAST_NAME" is one of the columns in that result set.
- `"Name: " + toUpperCase(variables.global.UserName)`
  Concatenates the string "Name: " and the value of the model parameter "UserName" after first converting the parameter to uppercase.)
- `getDoubleValue(Join01RS, 6)`
  Returns the value of the sixth column in the result set coming from the input called "Join01" as a `Double` object.

Boolean expressions

Certain expression fields within Studio require expressions that evaluate to Boolean (true or false) values. An example of this is for the "Where" parameter to the **Select** operator. Some examples of Boolean expressions are:

- `PRICE > 1000`
  "PRICE" is the name of a column in the single incoming result set. This evaluates to "true" when the value of the PRICE column is greater than 1000.
- `SalesDBOP.SALES_TERRITORY == "EAST"`
  "SalesDBOP" is the name of an input to this operator and "SALES_TERRITORY" is the name of a field in that input. Note that when testing for equality, you need to have two equals signs together. Here, the value evaluates to "true" when the value in the SALES_TERRITORY column is the character string "EAST."

- `CurrentUser.isMemberOf("myDomain", "DefaultAuthService", "Grid", "Administrators")`
  Check to see if the user executing the data service is a member of the Administrators group.

Complex expressions

Certain expression fields in Studio accept more complex expressions. These can span multiple lines and can contain multiple simpler expressions, each ending with a semicolon character (;). These are generally used in the more advanced operators, such as **Iterator**, **Generator**, and **Custom**.

Here's an example of a complex expression; it comes from the Increment field of an **Iterator** operator:

```
globals.Iterator03.counter++;
globals.Iterator03.currentZipCode =
    zipcodes[globals.Iterator03.counter];
globals.Iterator03.currentState =
    getStateForZipCode(globals.Iterator03.currentZipCode);
```

This expression consists of three separate statements that increment a variable and then use it to determine the values for two other variables. (Note that the variable "zipcodes," an array, and the function "getStateForZipCode" are not defined directly within the expression.)

Expressions and JavaScript

The expressions used within Studio are written in JavaScript. Don't worry if you don't know JavaScript. As you can see from the examples above, most of the expressions you will need to create are simple enough that knowledge of JavaScript is unnecessary. When your models begin requiring the more advanced features of Studio, however, you may need to better understand JavaScript and how it works. JavaScript is a powerful and flexible tool. The links below point to resources to help you acquaint yourself with the basics.

JavaScript resources

JavaScript tutorial:

- http://www.wdvl.com/Authoring/JavaScript/Tutorial/

JavaScript expression tester:

- http://www.arachnoid.com/javascript/interactiveJavaScript.html

JavaScript references:

- http://www.devguru.com/Technologies/ecmascript/quickref/javascript_intro.html
- http://www.topxml.com/javascript

Accessing columns from incoming result sets

When you connect one or more other elements to an operator in Studio, you can access the values in the incoming result sets' columns from expressions that you define in the operator. For example, if you have an **Input Source** feeding into a **Select** operator, in the **Select**'s "Where" field, you can refer to the **Input Source**'s columns. To do so, you simply specify the column by name. You can enter column names by hand, or you can use Studio's expressions menu to paste them in for you. For example, the expression you enter for the "Where" field might be `COLOR == "Red"` if "COLOR" is the name of a column in the **Input Source**'s schema.

If an operator has more than one input feeding into it, you must qualify the name of the column using the name of the upstream element. For example, if you have an **Iterator** operator which has inputs from "Aggregate1" and "Aggregate2," refer to the "Total" column from the first input as `Aggregate1.Total` and the "Total" column from the second input as `Aggregate2.Total`.

Advanced Users In the case of the **Custom** operator (where you cannot use the simple column name), or if, for some reason, you cannot uniquely refer to a column by its name in another type of operator (perhaps the column name is not a valid variable name or two columns have the same name), you can use the accessor methods in the `ResultSet` class to access the columns by their position. The result sets are in variables named after the incoming element's name with the suffix "RS" appended. For example, in the **Iterator** example above, there would be two result set variables accessible, named `Aggregate1RS` and `Aggregate2RS`. In addition, Avaki provides a set of utility functions you can call to ensure that `null` values are properly handled. See "Result set accessors" on page 73 for details.

Accessing global parameters

The view model parameters you've defined for your model can be referenced in the form `variables.global.<globalName>`. You can use Studio's expressions menu to insert references to global parameters into your expressions. See "The expressions menu" on page 71 for details on using the expressions menu.

Working with variables

Often you will need to create variables when constructing view models, particularly as the complexity of the model grows or as you use more advanced operators such as **Iterator**. Studio provides you with a JavaScript object named `variables`, which lets you store any variables you wish to use across multiple operators. Refer to any variables using the form `variables.<operatorInstanceName>.<variableName>` to guarantee their uniqueness across your model. The “operatorInstanceName” referred to should be the name of the operator where the variable is declared.

You can either enter the variable names directly in the expressions, or you can use Studio’s expressions menu to enter the name of any downstream variable. See “The expressions menu” on page 71 for details on using the expressions menu.

Note Variables that you declare in a given operator are only available (are only in scope) to operators that are upstream in the flow from that operator. Operators that are downstream from that operator or on a different branch of the flow cannot access your variable. If you need to use a variable across multiple branches, for example, you can declare it in your model’s .jsi file.

Updating variables—some caveats

In the expressions that you enter, it’s fine to update variables defined downstream that are not control variables. However:

- Don’t try to update variables that are defined upstream—you may see unexpected results if you change a variable outside of the operator that creates the variable. (All downstream variables appear in Studio’s expressions menu.)
- If you use a control variable—a counter introduced in and maintained by an **Iterator** operator, for example—don’t update the control variable inside any operator other than the one that introduces it.

Allowed types for variables

All values in the `variables` structure must be of one of the types listed here. Note that global parameters, while defined using SQL types, are accessed in your expressions using the corresponding Java types, according to JDBC conventions. The allowed types are:

BigDecimal	BigDecimal[]
BigInteger	BigInteger[]
Boolean	Boolean[]
Byte	Byte[]
Date	Date[]
Double	Double[]
Float	Float[]
Integer	Integer[]
Long	Long[]
Short	Short[]
String	String[]
Cloneable	ScriptableScope
Serializable	

The expressions menu

You don't need to know any JavaScript to write simple expressions in Avaki Studio. In any field where you can enter an expression, you can use Studio's expressions menu to choose the parameters, variables, logical operators, predefined functions and input column names with which you construct your expression. To display the menu, click in the expression field, then right-click.

The submenus in the expressions menu are described below.

Global Parameters

The **Global Parameters** submenu lists the global parameters defined in this view model. Select a parameter to enter it in the expression. For more on global parameters, see "Using global parameters" on page 63.

Note If there are no global parameters in your model, the **Global Parameters** submenu contains a disabled menu item: **No Global Parameters**.

Columns from the input element

There is one **Columns from** submenu for each input connected to the operator whose expression you are editing. The names of these submenus include the names of the upstream operators—for example, there might be **Columns from Select1** and **Columns from SalesFiguresDS**. Each submenu lists the columns in the upstream operator's output result set. Select a column name to enter it in the expression.

Note If no inputs are connected to the current operator, the expressions menu contains a disabled menu item: **No Inputs**.

Downstream Variables

The **Downstream Variables** submenu lists variables defined in operators downstream of (that is, closer to the **Result** than) the current operator. Select a variable to enter it in the expression.

Note If no variables are declared downstream of the current operator, the **Downstream Variables** submenu contains a disabled menu item: **No Downstream Variables**.

Operators

The **Operators** submenu lists JavaScript operators that you can use to modify your expression. If you have done any scripting or other programming, you will probably be familiar with many of these already. The most basic arithmetic operators are not included in the menu. They are addition (`+`), subtraction (`-`), multiplication (`*`), and division (`/`). To enter an operator into your expression, simply choose it from the submenu.

The table below lists the operators in the menu and describes their functionality. Note that in the descriptions, the terms "LHS" and "RHS" refer to the subexpressions on the Left Hand Side and Right Hand Side of the operator, respectively.

Menu Item	Description
Assign (`=`)	Assigns the RHS value to the LHS variable.
Equal (`==`)	Evaluates to true if the LHS and RHS are equal.
Not equal (`!=`)	Evaluates to true if the LHS and RHS are not equal.
Greater than (`>`)	Evaluates to true if the LHS is greater than the RHS.
Greater than or equal to (`>=`)	Evaluates to true if the LHS is greater than or equal to the RHS.
Less than (`<`)	Evaluates to true if the LHS is less than the RHS.
Less than or equal to (`<=`)	Evaluates to true if the LHS is less than or equal to the RHS.
Logical AND (`&&`)	Evaluates to true if the LHS and RHS are both true.
Logical OR (`||`)	Evaluates to true if either the LHS or RHS is true. (Note that if the LHS is true, the RHS isn't even evaluated.)
Logical NOT (`!`)	Evaluates to true if the RHS is false. Note that there is a unary operator—there is no LHS.
Bitwise AND (`&`)	Produces a new value whose bits are set when the corresponding bits in the LHS and RHS are both set. `0xf0a6 & 0xcd0f` evaluates to `0xc006`, for example.
Bitwise OR (`|`)	Produces a new value whose bits are set when either of the corresponding bits in the LHS and RHS is both set. `0xf0a6 | 0xcd0f` evaluates to `0xfdaf`, for example.

Menu Item	Description
Bitwise XOR (^)	Produces a new value whose bits are set when only one (but not both) of the corresponding bits in the LHS and RHS is set. `0xf0a6 ^ 0xcd0f` evaluates to `0x3da9`, for example.
Bitwise NOT (~)	Produces a new value whose bits are inverted with respect to the RHS. Note that there is a "unary" operator—there is no LHS. `~0xcd0f` evaluates to `0x32f0`, for example.
Shift left (<<)	Produces a new value by shifting the bits of the LHS *n* positions to the left, where *n* is the value of the RHS. The new rightmost bits will be 0s. `0xcd0f << 1` evaluates to `0x19a1e`, for example.
Shift right (>>)	Produces a new value by shifting the bits of the LHS *n* positions to the right, where *n* is the value of the RHS. The new leftmost bits will be 0s. `0xcd0f >> 1` evaluates to `0x6687`, for example.
Modulus (%)	Returns the remainder you would have if you had performed a division of LHS by RHS. `8 % 5` evaluates to `3`, for example (8 divided by 5 is 1 with the remainder 3).
In-line conditional (?:)	Used in the form `cond ? expr1 : expr2` where `cond` is an expression that evaluates to a Boolean value. Evaluates to `expr1` if `cond` is true, or `expr2` if `cond` is false. For example, the expression `x > 5 ? 2 : 1` evaluates to 2 when *x* is greater than 5 and evaluates to 1 when *x* is not greater than 5. This is sometimes referred to as a ternary conditional, since it has three operands.

Avaki Functions

The **Avaki Functions** submenu provides utility functions that you can use in your expressions. It contains several functions you can use to retrieve values from result sets as well as functions to add logic about who the current user is.

Result set accessors

The utility functions in the **Avaki Functions** submenu that operate on result sets automatically deal with the case where the column value is `null`. Each function takes two parameters:

- `resultSet`: the `ResultSet` object from which you want to extract the data, and

- `column`: the column whose data you want. You can specify the column either using a 1-based integer index or a string containing its name.

Note The names of the result set variables are the incoming operator's name with "RS" appended. For example, the result set object for an incoming operator named "Join1" would be named `Join1RS`.

The result set accessor functions are:

- `getSmallIntValue(resultSet, column)`
- `getIntValue(resultSet, column)`
- `getBigIntValue(resultSet, column)`
- `getRealValue(resultSet, column)`
- `getDoubleValue(resultSet, column)`
- `getBooleanValue(resultSet, column)`
- `getBigDecimalValue(resultSet, column)`

Current user methods

Studio also provides two functions that can be used to check (and therefore apply conditional logic based on) the identity of the user running the data service deployed from your model. You can either check whether the user is a particular individual or whether the user belongs to a particular group. You can use these, for example, to limit access to certain rows of data based on the user's membership in a certain group. The functions both take the Avaki domain name, authorization service name, and authorization service type where you wish to authenticate the user. The `is` function also takes the username you wish to check for, and the `isMemberOf` function takes the name of the group you wish to check against.

- `CurrentUser.is(domain, authSvc, authSvcType, username)`
- `CurrentUser.isMemberOf(domain, authSvc, authSvcType, group)`

Propagating name changes through the view model

If an operator name or a parameter name changes, Avaki Studio propagates the change to all expressions that use the changed element.

Note This propagation does not apply to the code inside **Custom** operators, nor does it apply to operator control variables. This issue will be addressed in an upcoming release.

Advanced topics

Using a .jsi file to enhance your model

Avaki Studio allows you to create a special file that is included when the view model is deployed as a data service. You can place any functions or variables you wish to access from the expressions in your model in this file, facilitating code reuse. The file is called a JavaScript Include, or .jsi, file. It has the same name as your Avaki view model (.avm) file, apart from the extension.

To create a .jsi file for your view model, right-click the background in the view model editor and choose **Open JSI File** from the contextual menu. Studio creates the file if it doesn't already exist, and opens it in a text editor. You can then edit it to include any functions, variables, or other JavaScript code you like.

For example, say your model requires converting `Date` objects into integers of the format "yyyymmdd." You could create a function as follows, and then call it from multiple places within your model (such as **Projection** column definitions):

```
// Convert a Date object to the form "YYYYMMDD" as an int
function dateToInt(dateObject) {
    return dateObject.getFullYear() * 10000 +
        (dateObject.getMonth() + 1) * 100 +
        dateObject.getDate();
}
```

Performance of sort-based operators

Several operators must sort their inputs. These include **Order By**, **Join** (only when using the Sort Merge algorithm), **Group By**, and **Intersection**, as well as any operator that uses the "Distinct" option to eliminate duplicate rows. You should be aware of two factors that can affect sort performance: sort chunk size and the location of temporary files for sorts. These are discussed below.

Note This section is for more advanced users who wish to try to increase the performance of data services deployed from view models. Many users will not need to adjust these parameters.

Sort chunk size

To enable the sort-based operators to handle arbitrarily large result sets, Avaki breaks the inputs into chunks and uses the local disk as a temporary backing store to hold intermediate results. These intermediate results are cleaned up when the operation completes.

There is a tradeoff between the amount of memory that an Avaki grid server uses for its computation and the amount of I/O that it must perform to sort or join large result sets. Avaki Studio can break the inputs into a large number of small chunks or into a smaller number of larger chunks. A large number of small chunks requires less memory at any given time, but will result in more disk I/O. A small number of large chunks will result in a larger peak memory usage, but less disk I/O overall.

By default, a view model operates with a chunk size of 10,000 rows. You can override this default by declaring the following in your model's .jsi file:

```
var overrideSortChunkSize = <new_chunk_size_value>;
```

Providing enough space for temporary sort files

When a data service sorts data, it writes temporary files to the temp directory specified by the java.io.tmpdir system property on the grid server where the data service is running. The default location is <Avaki-install-dir>/jboss/server/grid-server/tmp. (See the *Sybase Avaki EII Administration Guide* for information on setting system properties.) The temporary sort files can be quite large. If you create a data service that sorts large result sets, be sure that the temp directory has enough disk space to write large sort files. If there isn't enough disk space, the data service execution will fail because the sort operation is unable to finish.

Chapter 5

Metadata modeling

This chapter describes how to create metadata models, import them into Avaki Studio, and apply the models to Avaki objects.

In this chapter:

About metadata models

In Avaki Studio, a *metadata model* expresses a schema. Metadata models define a set of tables in which each table contains a set of named columns and each column is associated with a data type. A table in a metadata model can be mapped (linked) to an Avaki object, such as a data service or a database operation, or to a table in a database. When the metadata model is deployed, the mapping lets you call each mapped object by the name of the corresponding table in the metadata model. (Tables in the model have no relationships to one another.)

You can also derive a view model schema from a metadata model, which means ensuring that the results of any data service deployed from the view model conform to the metadata model.

There are two ways to obtain a metadata model:

- Import a file into Avaki Studio from a data modeling tool such as Sybase PowerDesigner or ERwin. For instructions, see "Importing metadata models" on page 79.
- Create a metadata model in Avaki Studio by specifying tables, columns, and data types. For instructions, see "Creating and editing metadata models" on page 85.

Metadata modeling in Avaki has two important benefits:

You have easier access to Avaki objects like data services. When you map a metadata model to an Avaki object, you can access the object using the metadata model's name, which may be the same as the name of the database table from which it was created. For example, suppose you have a Crystal Reports application that accesses a table called HumanResources.Employee in a relational database. You are building an Avaki data service, emp.DS, that selects and transforms data from the HumanResources.Employee table, and you would like to make it simple to point the CR application at either the real database or the Avaki data service for its data.

Solution: Import the schema for HumanResources.Employee into Avaki Studio as a metadata model. You now have a model called HumanResources, which contains a table called Employee. Next, map the Employee table in the HumanResources metadata model to your emp.DS data service. This mapping lets you access the emp.DS data service as HumanResources.Employee—the same name you use to access the table in the database. You can point your application at either the database or Avaki; no further changes are required.

You can ensure that Avaki data services you create conform to a particular schema. Suppose you created a schema in a modeling tool, such as Sybase PowerDesigner or ERwin, and you want to create a data service that produces results matching your schema.

Solution: Import the schema into Avaki Studio as a metadata model and derive a new view model from the metadata model. As you add operators to the view model, Studio monitors the schema and alerts you if the view model's schema diverges from that of the metadata model. Any data services you deploy from this view model will match the schema you imported from the modeling tool.

Importing metadata models

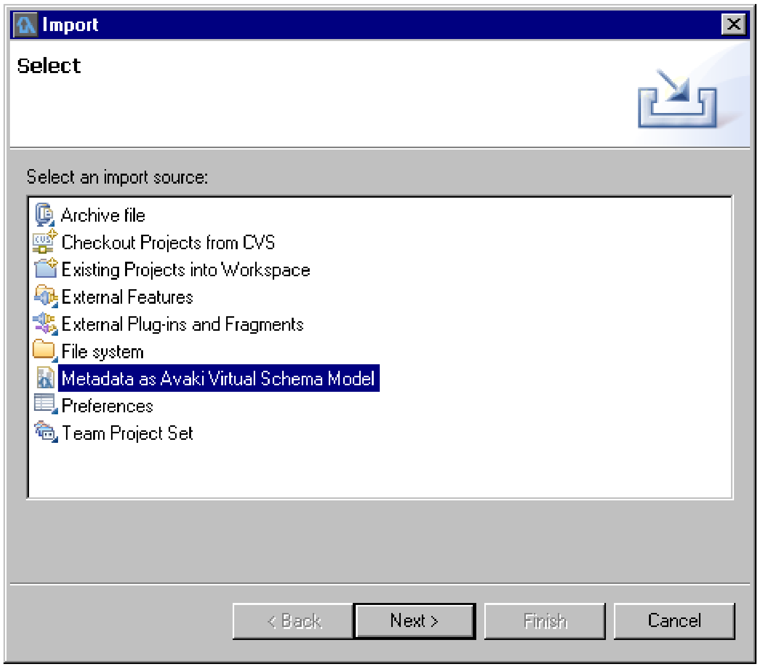
You can import several kinds of data model (schema) files into Avaki Studio. The following are the supported formats:

Format	Sources	Notes
ERwin XML	ERWin 7.0.1	
JDBC	An existing Avaki database connector	You can specify one table, multiple tables, or all tables to import into the database.
Physical data model (.pdm)	Sybase PowerDesigner 12.0	

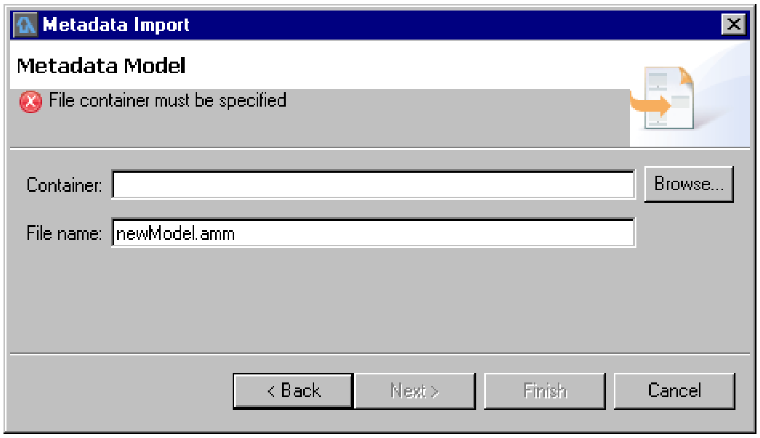
You'll import the metadata model into a project folder. If you don't want to use an existing project folder, create a new one by selecting File > New > Studio Project. For detailed instructions, see "Creating a project" on page 14.

Follow these steps to import a metadata model into Avaki Studio:

Step 1 Select File > **Import**. The Select window appears.


Import
Select
Select an import source:
Archive file
Checkout Projects from CVS
Existing Projects into Workspace
External Features
External Plug-ins and Fragments
File system
Metadata as Avaki Virtual Schema Model
Preferences
Team Project Set
< Back
Next >
Finish
Cancel

Step 2 Click **Metadata as Avaki Virtual Schema Model** and click **Next**. The first screen of the import wizard appears.


Metadata Import

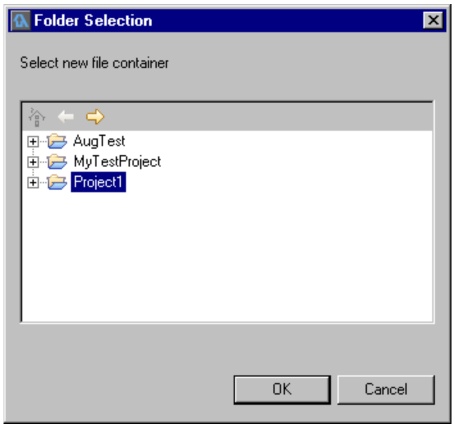
Metadata Model

File container must be specified

Container:

File name: newModel.amm

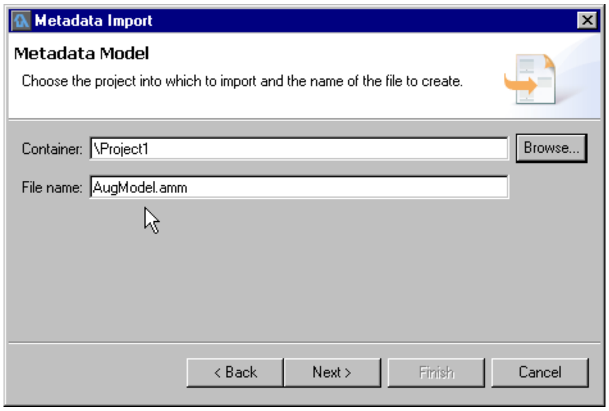
Step 3 Click **Browse** to list the project folders into which you can import the model. Select a project folder from the list and click **OK**.


Folder Selection

Select new file container

AugTest
MyTestProject
Project1

Step 4 (Optional) Change the name of the Avaki metadata model (.amm file) in the File name field.


Metadata Import

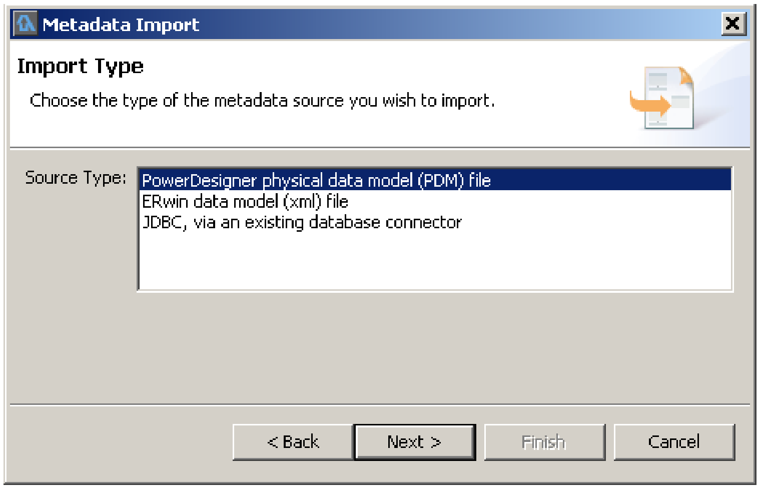
Metadata Model

Choose the project into which to import and the name of the file to create.

Container: \Project1

File name: AugModel.amm

Step 5 Click **Next**. The Import Type window appears.



Metadata Import

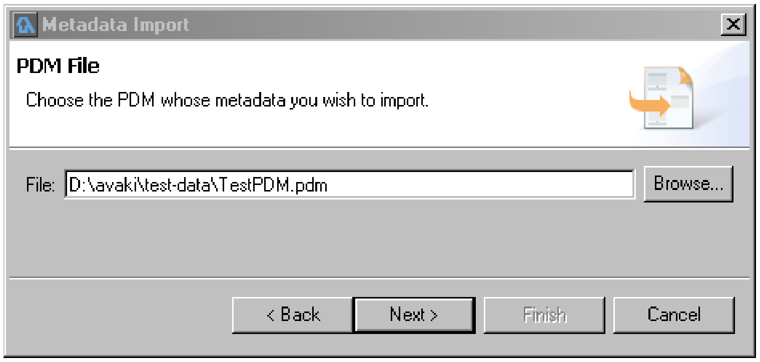
Import Type

Choose the type of the metadata source you wish to import.

Source Type:
- PowerDesigner physical data model (PDM) file
- ERwin data model (xml) file
- JDBC, via an existing database connector

Step 6 Click the data model type to import and click **Next**.

Depending on which import type you choose in Step 5, the dialogs display PDM, ERWIN, or DBconn if you chose JDBC. The example in this procedure uses Power-Designer.



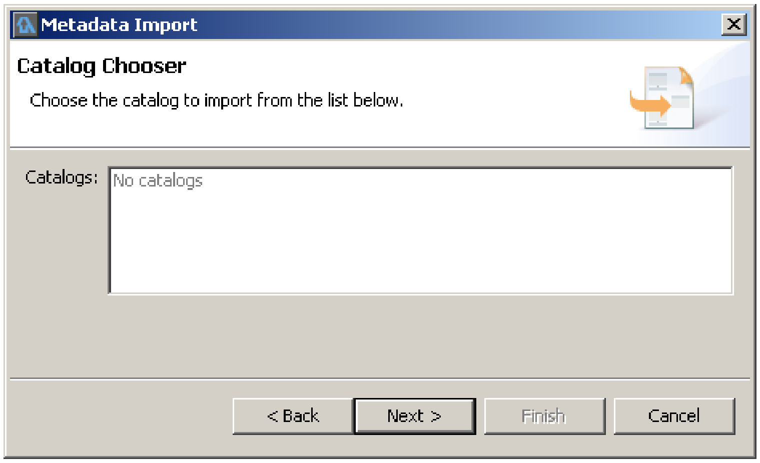
Metadata Import

PDM File

Choose the PDM whose metadata you wish to import.

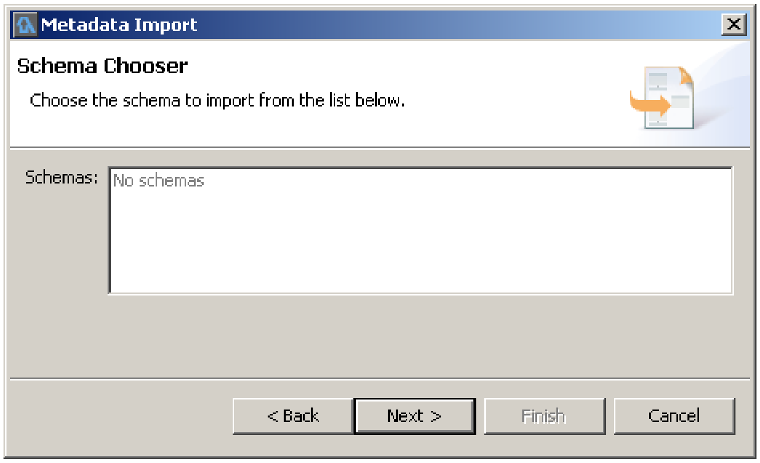
File: D:\avaki\test-data\TestPDM.pdm

Step 7 Click **Browse** to navigate to the location of your metadata model, select the file to import and click **Next**. The Catalog Chooser window appears.



Step 8 Select the catalog to import from the Catalog Chooser and click **Next**. The Schema Chooser window appears.

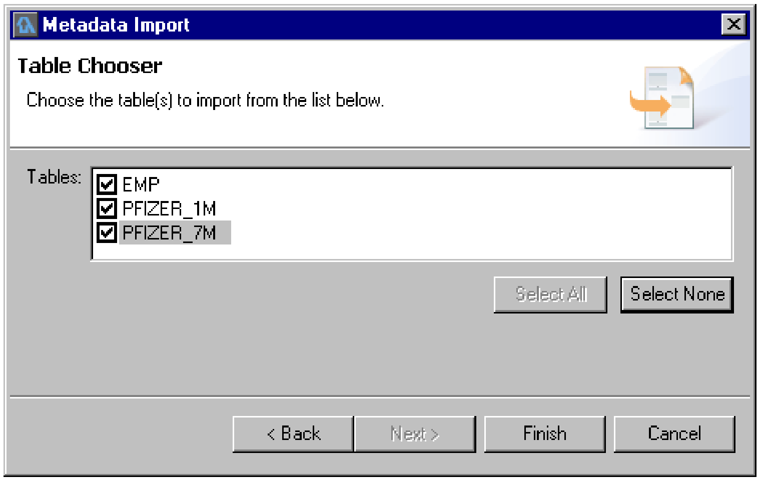
Note If your external schema contains no catalogs, you will see “No catalogs” in the Catalog Chooser window.



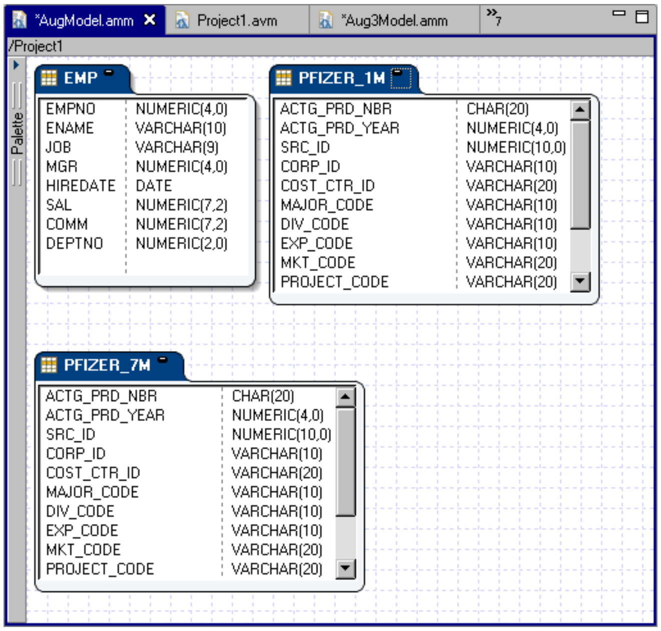
Step 9 Select the schema to import from the list and click **Next**. The Table Chooser window appears.

Note If the external schema contains no schema, you will see “No schemas” in the Schema Chooser window. In this instance, schema refers to a database user who owns a set of tables.

Step 10 Select any table or tables you want to import from the displayed list and click **Finish**.



The editor pane displays the tables from the metadata model you imported.



EMP

EMPNO	NUMERIC(4,0)
ENAME	VARCHAR(10)
JOB	VARCHAR(9)
MGR	NUMERIC(4,0)
HIREDATE	DATE
SAL	NUMERIC(7,2)
COMM	NUMERIC(7,2)
DEPTNO	NUMERIC(2,0)

PFIZER_1M

ACTG_PRD_NBR	CHAR(20)
ACTG_PRD_YEAR	NUMERIC(4,0)
SRC_ID	NUMERIC(10,0)
CORP_ID	VARCHAR(10)
COST_CTR_ID	VARCHAR(20)
MAJOR_CODE	VARCHAR(10)
DIV_CODE	VARCHAR(10)
EXP_CODE	VARCHAR(10)
MKT_CODE	VARCHAR(20)
PROJECT_CODE	VARCHAR(20)

PFIZER_7M

ACTG_PRD_NBR	CHAR(20)
ACTG_PRD_YEAR	NUMERIC(4,0)
SRC_ID	NUMERIC(10,0)
CORP_ID	VARCHAR(10)
COST_CTR_ID	VARCHAR(20)
MAJOR_CODE	VARCHAR(10)
DIV_CODE	VARCHAR(10)
EXP_CODE	VARCHAR(10)
MKT_CODE	VARCHAR(20)
PROJECT_CODE	VARCHAR(20)

If the imported tables are sitting on top of one another, you can space them out evenly by right-clicking inside the editor pane and selecting **Tidy Tables** from the submenu.

Alternatively, click the Tidy Tables icon on the menu bar .

Creating and editing metadata models

It's generally easier to import metadata models (as described in "Importing metadata models" on page 79) than to create them in Avaki Studio. If you don't have access to a suitable modeling tool, though, follow the steps given here.

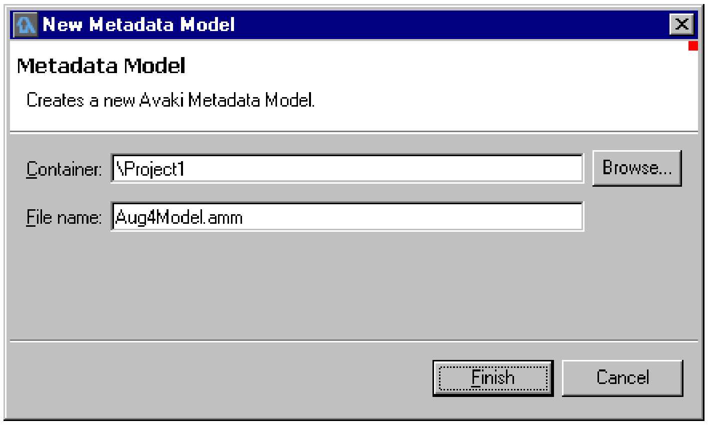
Editing metadata models is similar to using the Avaki view model (AVM) editor. It is, however, much simpler because there is only one type of element in the model—a table. There are no connections between tables. Every table has a name and zero or more column definitions.

Note To edit a metadata model after it's been deployed, you must first undeploy the model. See "Undeploying metadata models" on page 95 for instructions.

Follow these steps to create a metadata model. If you want to edit an existing model, start with Step 4.

Step 1 To create a new metadata model, pull down the File menu and select **New > Other**.

Step 2 In the Select a Wizard window, open the Avaki folder and click **Metadata Model**. The New Metadata Model wizard appears.



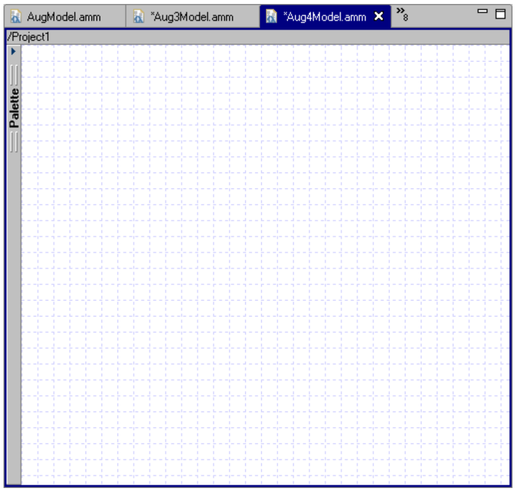
Metadata Model

Creates a new Avaki Metadata Model.

Container: \Project1

File name: Aug4Model.amm

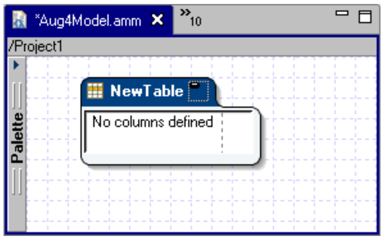
Step 3 Click **Browse** to navigate to the container where you want your new model to reside, optionally rename the file, and click **Finish.** An editor pane displays beneath the name you assigned your new metadata model.



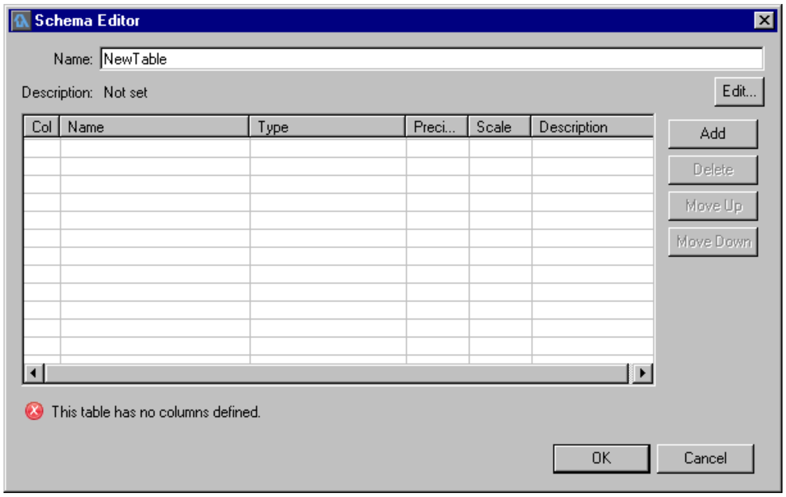
Step 4 To open an existing metadata model for editing, click the Navigator tab in the left pane. Navigate to the folder where your model is stored and double-click the .amm file for the model. The model opens in the editor pane.

Step 5 To add a new table to the metadata model, click the palette to the left of the editor pane and click **New Table**.

Step 6 Position your cursor inside the editor pane and click to insert the new table. It appears with no columns defined.



Step 7 To edit the table, double click it and the Schema Editor window appears.



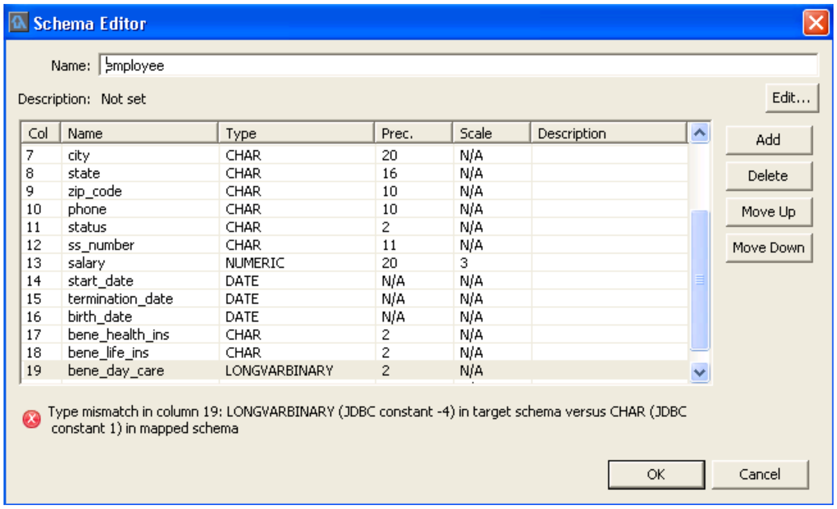
This table has no columns defined.

Step 8 Replace NewTable with a table name of your choosing by typing a name in the Name field.

Step 9 (Optional) Click **Edit** to type a description for your table.

Step 10 Click **Add** to add a new column to your table, and then position your cursor in the column fields to edit their names and values.

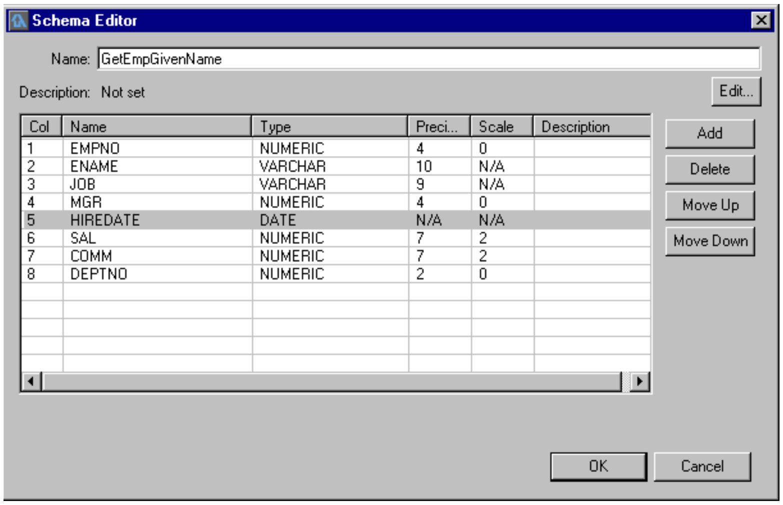
Columns are numbered to help you quickly pinpoint schema errors, which appear at the bottom of the Schema Editor window as in the following example.



Col	Name	Type	Prec.	Scale	Description
7	city	CHAR	20	N/A	
8	state	CHAR	16	N/A	
9	zip_code	CHAR	10	N/A	
10	phone	CHAR	10	N/A	
11	status	CHAR	2	N/A	
12	ss_number	CHAR	11	N/A	
13	salary	NUMERIC	20	3	
14	start_date	DATE	N/A	N/A	
15	termination_date	DATE	N/A	N/A	
16	birth_date	DATE	N/A	N/A	
17	bene_health_ins	CHAR	2	N/A	
18	bene_life_ins	CHAR	2	N/A	
19	bene_day_care	LONGVARBINARY	2	N/A	

Type mismatch in column 19: LONGVARBINARY (JDBC constant -4) in target schema versus CHAR (JDBC constant 1) in mapped schema

Step 11 Continue to click **Add** and enter data until your table contains all the columns you need.



Name: GetEmpGivenName

Description: Not set

Col	Name	Type	Preci...	Scale	Description
1	EMPNO	NUMERIC	4	0	
2	ENAME	VARCHAR	10	N/A	
3	JOB	VARCHAR	9	N/A	
4	MGR	NUMERIC	4	0	
5	HIREDATE	DATE	N/A	N/A	
6	SAL	NUMERIC	7	2	
7	COMM	NUMERIC	7	2	
8	DEPTNO	NUMERIC	2	0	

To reposition a column, select its row and click **Move Up** or **Move Down**.

Step 12 When you have finished adding and editing columns, click **OK**.

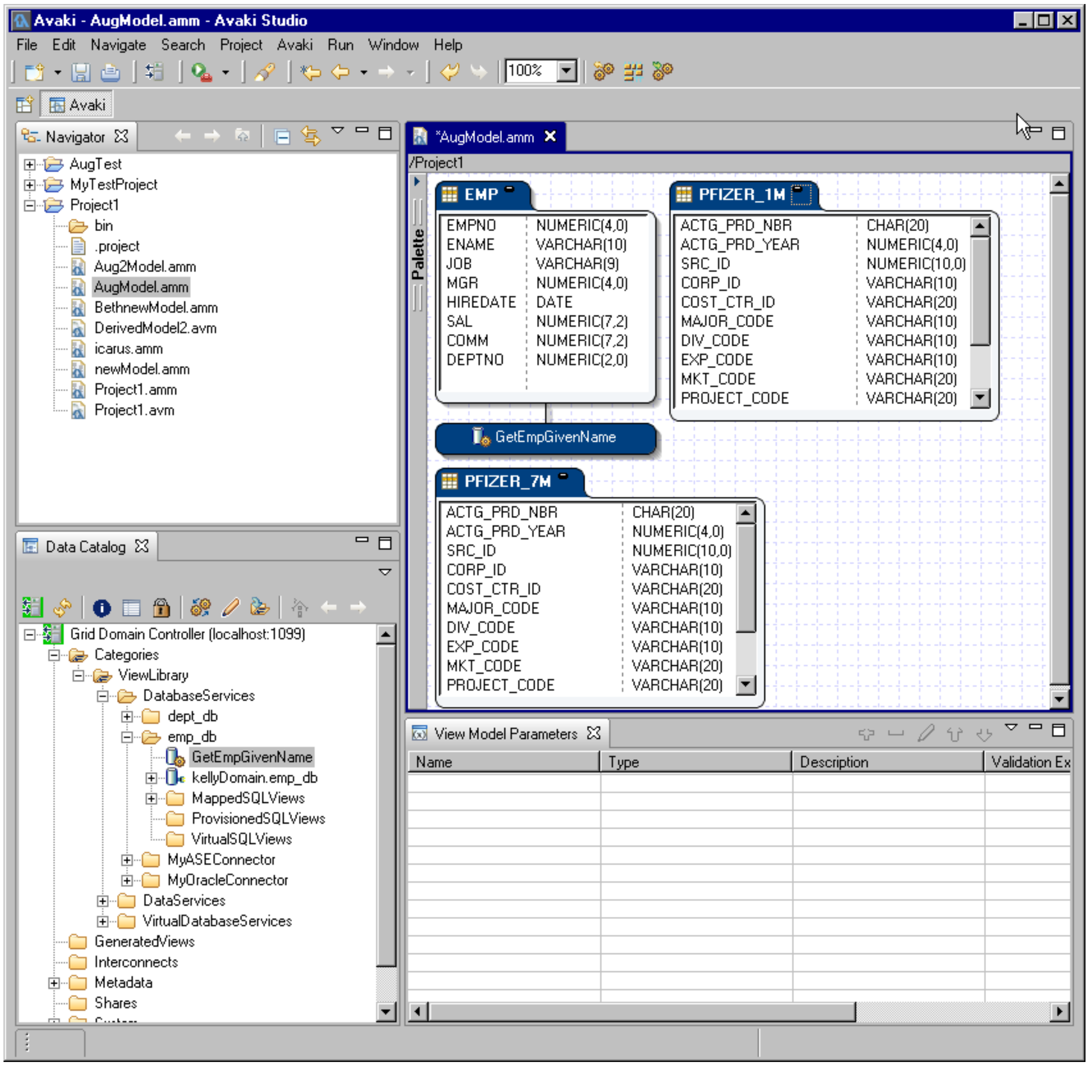
Mapping metadata models to Avaki objects

You can map (link) a table in a metadata model to an Avaki object, such as a data service or a database operation, or to a table in a database. When you deploy the metadata model, this mapping allows you to access the object using the table name in the model—probably the same name you use to access the table in the database.

This procedure maps the metadata model we created in the previous procedure, "Creating and editing metadata models" on page 85, to a database operation called GetEmpGivenName.

Step 1 Click the data catalog view in the left pane and navigate to the object you want to map. In this example, we will map the GetEmpGivenName database operation to the EMP table we imported in "Importing metadata models" on page 79.

Step 2 In the data catalog view, click GetEmpGivenName. Drag and drop it on the EMP table in the editor pane.



EMP

EMPNO	NUMERIC(4,0)
ENAME	VARCHAR(10)
JOB	VARCHAR(9)
MGR	NUMERIC(4,0)
HIREDATE	DATE
SAL	NUMERIC(7,2)
COMM	NUMERIC(7,2)
DEPTNO	NUMERIC(2,0)

PFIZER_1M

ACTG_PRD_NBR	CHAR(20)
ACTG_PRD_YEAR	NUMERIC(4,0)
SRC_ID	NUMERIC(10,0)
CORP_ID	VARCHAR(10)
COST_CTR_ID	VARCHAR(20)
MAJOR_CODE	VARCHAR(10)
DIV_CODE	VARCHAR(10)
EXP_CODE	VARCHAR(10)
MKT_CODE	VARCHAR(20)
PROJECT_CODE	VARCHAR(20)

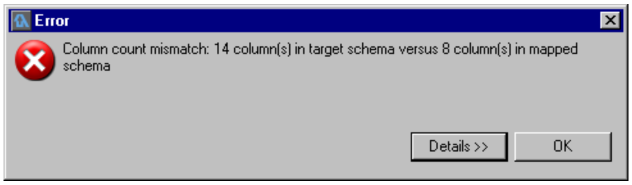
PFIZER_7M

ACTG_PRD_NBR	CHAR(20)
ACTG_PRD_YEAR	NUMERIC(4,0)
SRC_ID	NUMERIC(10,0)
CORP_ID	VARCHAR(10)
COST_CTR_ID	VARCHAR(20)
MAJOR_CODE	VARCHAR(10)
DIV_CODE	VARCHAR(10)
EXP_CODE	VARCHAR(10)
MKT_CODE	VARCHAR(20)
PROJECT_CODE	VARCHAR(20)

Name	Type	Description	Validation Ex

The metadata model editor checks to ensure that the schema of the object being dropped matches that of the table. If so, the mapping is made and the mapped object appears in the editor pane.

If the schemas do not match—for example, if you drop the GetEmpGivenName database operation onto the PFIZER_1M table, the mapping is invalid and Studio returns an error similar to the following:


Error

Column count mismatch: 14 column(s) in target schema versus 8 column(s) in mapped schema

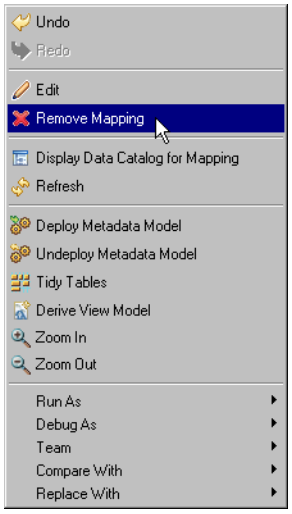
Step 3 To resolve schema errors, click **OK**, and you are left with an invalid mapped object displaying a red state. To correct the mapping:

- Edit the target schema, update the Avaki object (for example, database operation or data service), and regenerate the schema.

  Right-clicking the schema and selecting **Refresh** from the submenu revalidates the tables in the editor pane. If the connection to the grid server has been lost, Refresh will attempt to connect.

- Alternatively, drag a new Avaki object over the target schema to create a new mapping.

If you make a mapping error, right click the table and select **Remove Mapping** from the submenu.



Step 4 Now you must deploy the model before you can point your application at Avaki Studio. Simply right-click the table and select **Deploy Metadata Model** from the submenu. Alternatively, click the Deploy button on the toolbar

Step 5 Point your application at either the database or Avaki; no further changes are required.

Note Pointing the application at Avaki means you are connecting to Avaki via JDBC. See "Deploying metadata models," below, for information on naming conventions for mapped tables.

Deploying metadata models

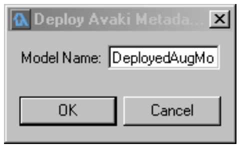
Deploying a metadata model makes the data services, database operations, or provisioned tables that are mapped to the model's tables accessible by the table names in the model through JDBC.

Note When you deploy a metadata model, the mapped tables are deployed and any unmapped tables are silently ignored.

Follow these steps to deploy your metadata model into the data catalog.

Step 1 When you are ready to deploy your model to the grid, right-click the table you just mapped and select **Deploy** from the submenu.

Step 2 Type a name in the Model Name field.

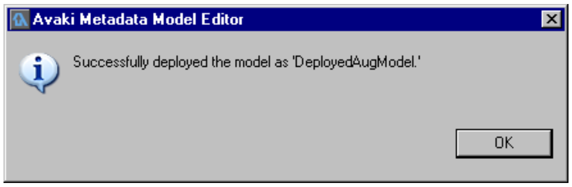

Deploy Avaki Metada...

Model Name: DeployedAugMo

Note The Model Name field accepts two-part names of the form [<catalog>.]<schema>, where the optional <catalog> part defaults to your Avaki domain name and <schema> is the arbitrary name you give this model. The string you enter must contain no spaces or punctuation other than - (hyphen), _ (underscore), and at most one . (period). The names of mapped tables in the model, then, have three parts: [<catalog>.]<schema>.<table>. If you choose to enter the database name as the model name (<schema>), you can refer to objects mapped to tables in this model as [<catalog>.]<database>.<table>.

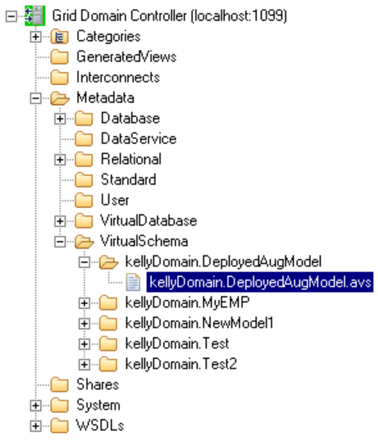
Step 3 Click **OK**. If Avaki Studio encounters errors, a message appears. Otherwise the deployment begins.

The metadata model is successfully deployed and added to the data catalog's Metadata directory when you see the following message:



Successfully deployed the model as 'DeployedAugModel.'

Step 4 You can check your newly deployed object in the data catalog:



```
Grid Domain Controller (localhost:1099)
  Categories
  GeneratedViews
  Interconnects
  Metadata
    Database
    DataService
    Relational
    Standard
    User
    VirtualDatabase
    VirtualSchema
      kellyDomain.DeployedAugModel
        kellyDomain.DeployedAugModel.avs
      kellyDomain.MyEMP
      kellyDomain.NewModel1
      kellyDomain.Test
      kellyDomain.Test2
  Shares
  System
  WSDLs
```

If the deployment fails, the operation stops, newly provisioned objects are removed from the grid, and Avaki Studio displays an error.

To undeploy a metadata model, see "Undeploying metadata models" on page 95.

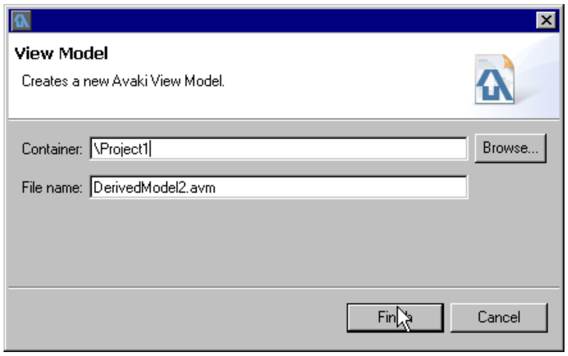
Deriving metadata models

Deriving a view model from a metadata model is a way to create a new data service with a designated output schema. When you perform this operation, Avaki Studio checks that the view model you built generates the output schema you specified. In other words, the results of a data service deployed from the view model will conform to the metadata model.

Step 1 Open a project in the Navigator pane and double click a metadata model to bring it into the editor pane.

Step 2 Right-click the table you want to derive and select **Derive View Model** from the sub-menu. The View Model window displays.

Step 3 Specify the container and file name and click **Finish**.

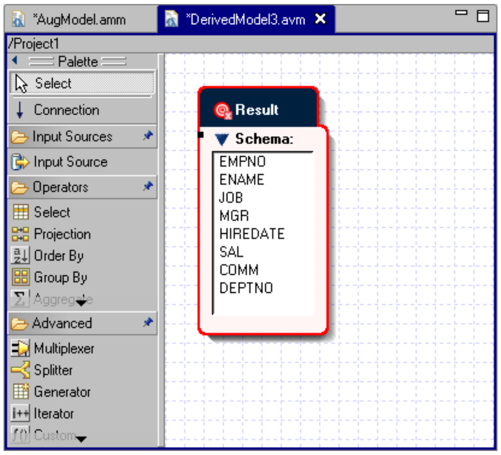

View Model

Creates a new Avaki View Model.

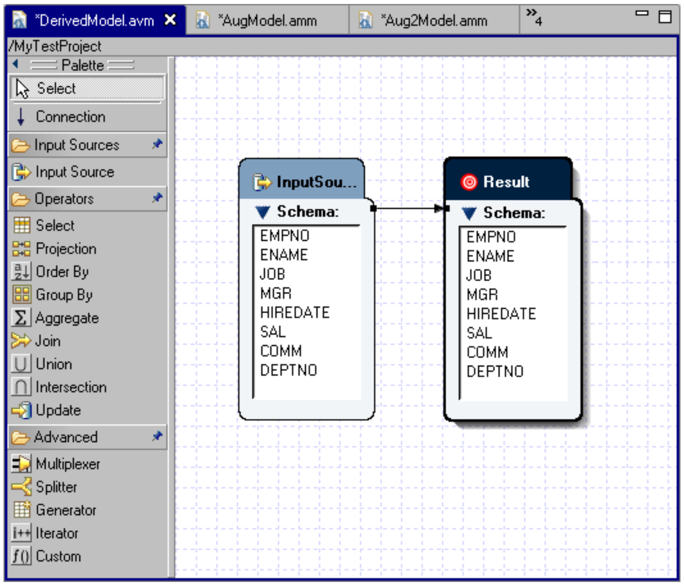
Container: \Project1

File name: DerivedModel2.avm

The result icon shows the target schema that you selected. As you design the data service, Avaki Studio checks that the output matches the target schema.



Step 4 Create a data service where you connect input to output.



Step 5 In the editor pane, click **Connection**, then click the Input Source table and then click the Result table.

As you add other operators to the view model, Avaki Studio monitors the schema and alerts you if the view model's schema diverges from that of the metadata model.

Deleting metadata models

Do not manually delete individual deployed mappings (components of a deployed metadata model) by browsing the data catalog and deleting the corresponding files and directory. Delete mappings using the **Undeploy Metadata Model** option in Avaki Studio. See "Undeploying metadata models" on page 95.

Undeploying metadata models

Undeploying a metadata model reverses the results of deploying it, which means that you can no longer access the tables in the model through JDBC using the specified names.

To edit a deployed model, you must first undeploy it.

Step 1 Right-click inside the editor pane and choose **Undeploy Metadata Model** from the submenu.

Alternatively, click the Undeploy Metadata Model from the Avaki Server button on the toolbar, directly above the editor pane .

Chapter 6

Managing the data catalog

This chapter describes tasks you can perform in Avaki Studio's data catalog view; these include:

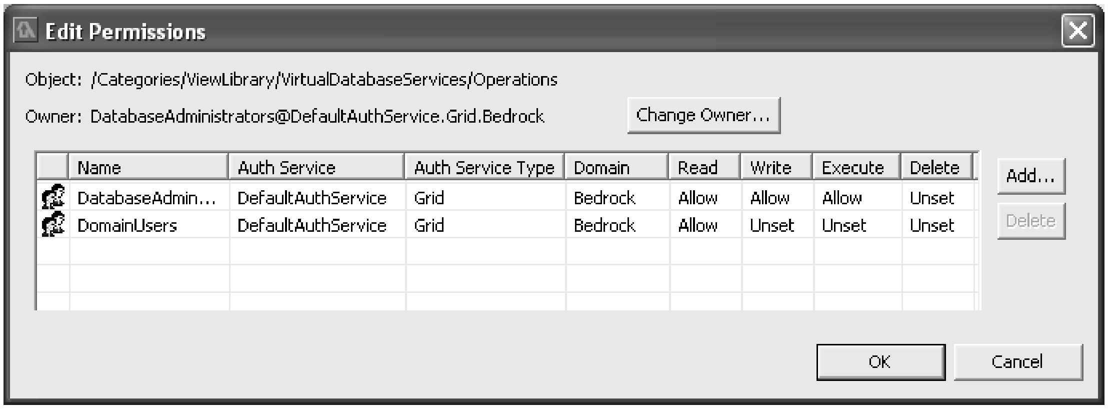
- Modifying access control lists (ACLs)—page 98
- Creating and modifying attributes—page 101
- Creating categories and adding objects to them—page 104

Avaki Studio provides a limited set of tools for managing the data catalog. The complete tool set is available in the web UI, where in addition to the tasks described here, you can move, link, and rename objects; create directories and shares; manage caching; and search the Avaki domain. For information on using the web UI, see the *Sybase Avaki EII Administration Guide*.

Modifying ACLs

This section explains how to display and modify access control lists through the data catalog view. You can set permissions, change ownership, add users or groups to an ACL, or remove users or groups.

Step 1 To display the ACL for an item in the data catalog, right-click the item in the data catalog view and select **Permissions** from the context menu. The Edit Permissions dialog appears.

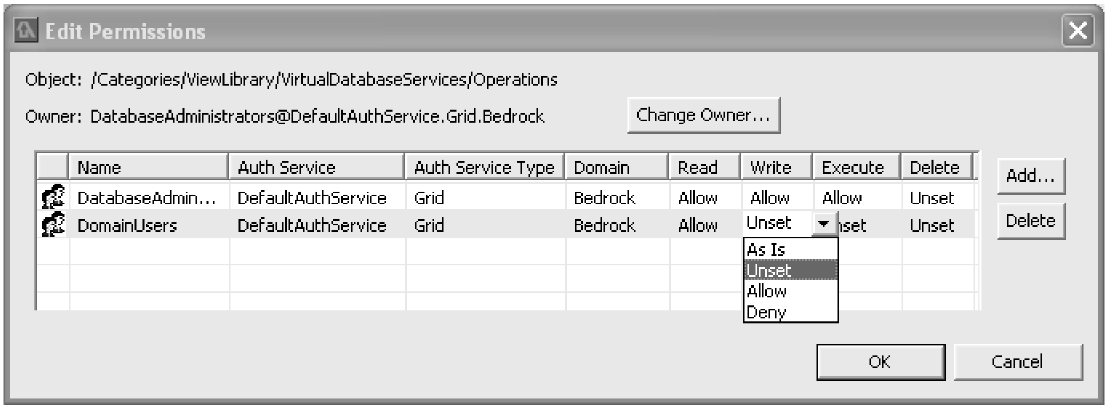


Object: /Categories/ViewLibrary/VirtualDatabaseServices/Operations

Owner: DatabaseAdministrators@DefaultAuthService.Grid.Bedrock

Name	Auth Service	Auth Service Type	Domain	Read	Write	Execute	Delete
DatabaseAdmin...	DefaultAuthService	Grid	Bedrock	Allow	Allow	Allow	Unset
DomainUsers	DefaultAuthService	Grid	Bedrock	Allow	Unset	Unset	Unset

Step 2 To change a permission for a user or group on the ACL, click on the permission in that user or group's row of the table. For example, to change the Write permission for members of the DomainUsers group, click on the cell that says "Unset" in the second row of the Write column, then pull down the menu.



Object: /Categories/ViewLibrary/VirtualDatabaseServices/Operations

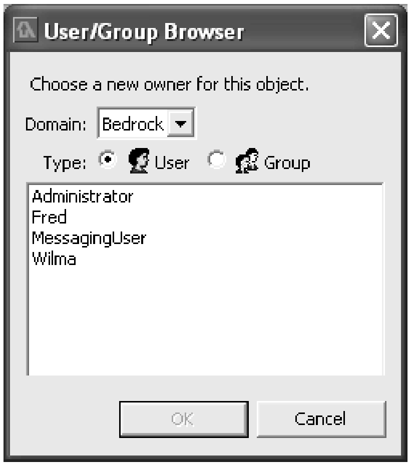
Owner: DatabaseAdministrators@DefaultAuthService.Grid.Bedrock

Name	Auth Service	Auth Service Type	Domain	Read	Write	Execute	Delete
DatabaseAdmin...	DefaultAuthService	Grid	Bedrock	Allow	Allow	Allow	Unset
DomainUsers	DefaultAuthService	Grid	Bedrock	Allow	Unset	nset	Unset

Step 3 Select a new value—**Allow**, for example—from the pull-down menu.

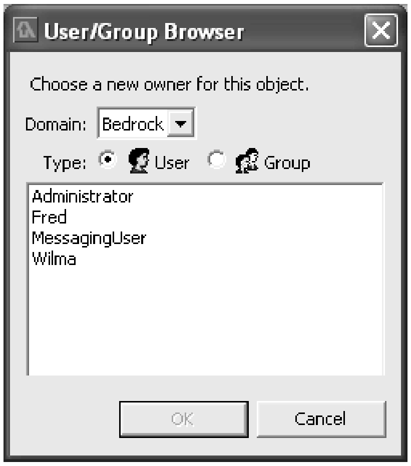
Note At any point, you can click **Cancel** to dismiss the Edit Permissions dialog without saving your changes, or click **OK** to dismiss the Edit Permissions dialog and save your changes.

Step 4 To change the object's owner, click the **Change Owner...** button. The User/Group Browser appears.



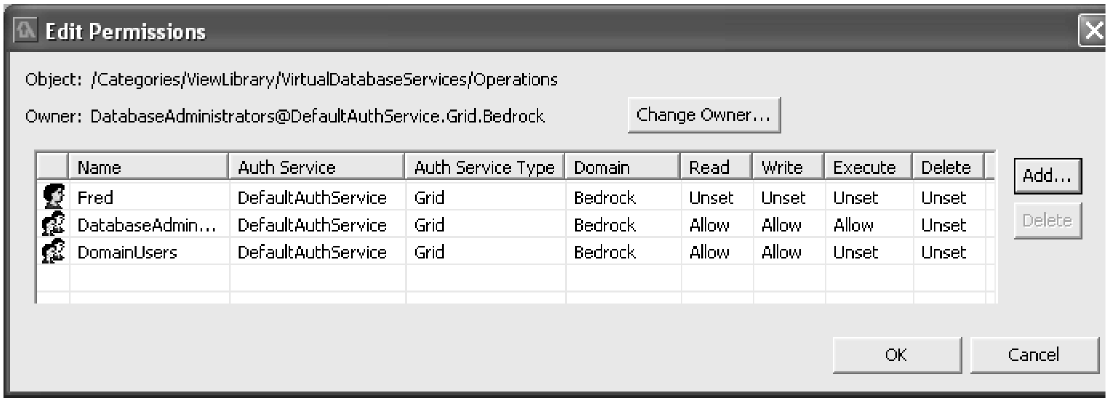
Step 5 Use the Domain pull-down if you want to select a user or group from another Avaki domain. Click the **Group** button if you want to select a group. Choose a user or a group from the list and click **OK** to dismiss the browser. Studio updates the Edit Permissions dialog to show the object's new owner.

Step 6 To add a user or group to the ACL, click the **Add** button. The User/Group Browser appears.



Step 7 Use the Domain pull-down if you want to select a user or group from another Avaki domain. Click the **Group** button if you want to select a group. Choose a user or a group from the list and click **OK** to dismiss the browser. Studio updates the Edit Per-

missions dialog to show the new line in the ACL. In this example, we've added a user called Fred.



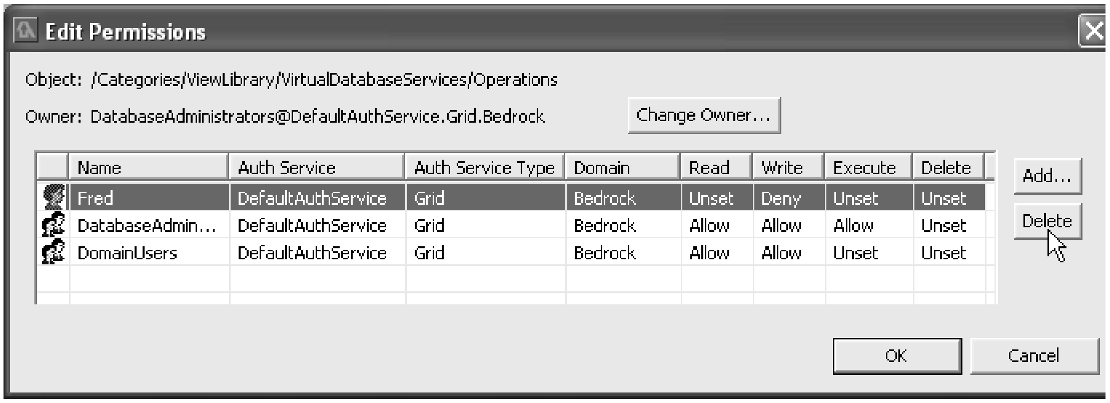
Object: /Categories/ViewLibrary/VirtualDatabaseServices/Operations

Owner: DatabaseAdministrators@DefaultAuthService.Grid.Bedrock

Name	Auth Service	Auth Service Type	Domain	Read	Write	Execute	Delete
Fred	DefaultAuthService	Grid	Bedrock	Unset	Unset	Unset	Unset
DatabaseAdmin...	DefaultAuthService	Grid	Bedrock	Allow	Allow	Allow	Unset
DomainUsers	DefaultAuthService	Grid	Bedrock	Allow	Allow	Unset	Unset

Notice that Fred's permissions are all Unset. If you leave them in that state, Studio will drop Fred from the ACL. You can use the Read, Write, Execute, and Delete pull-down menus to set permissions.

Step 8 To remove a user or group from the ACL, select it and click the **Delete** button.



Object: /Categories/ViewLibrary/VirtualDatabaseServices/Operations

Owner: DatabaseAdministrators@DefaultAuthService.Grid.Bedrock

Name	Auth Service	Auth Service Type	Domain	Read	Write	Execute	Delete
Fred	DefaultAuthService	Grid	Bedrock	Unset	Deny	Unset	Unset
DatabaseAdmin...	DefaultAuthService	Grid	Bedrock	Allow	Allow	Allow	Unset
DomainUsers	DefaultAuthService	Grid	Bedrock	Allow	Allow	Unset	Unset

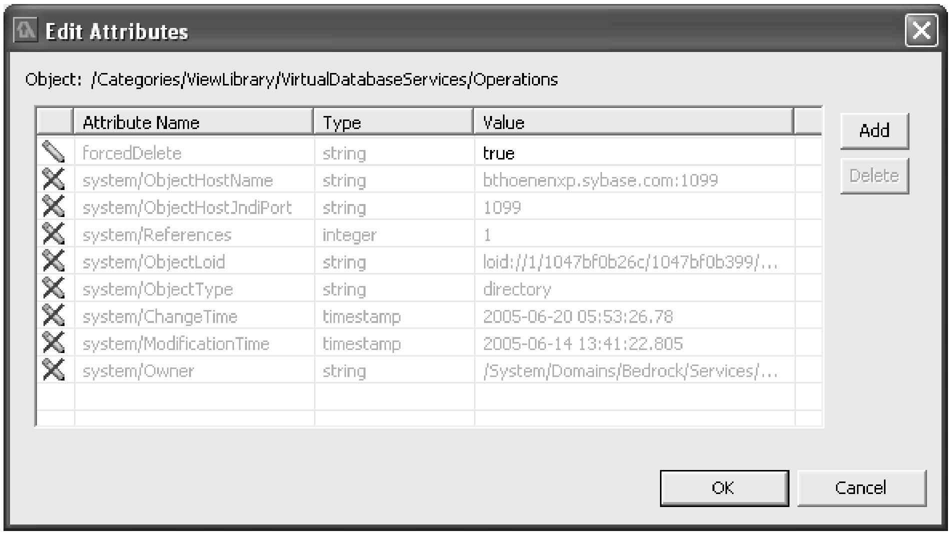
Studio removes the selected user or group from the ACL.

Step 9 Click **OK** to save your changes and dismiss the Edit Permissions dialog.

Managing attributes

Read this section for instructions on displaying, creating, modifying, and deleting the attributes associated with objects in the data catalog.

Step 1 To display the attributes for an item in the data catalog view, right-click the item and select **Attributes** from the context menu. The Edit Attributes dialog appears.



Object: /Categories/ViewLibrary/VirtualDatabaseServices/Operations

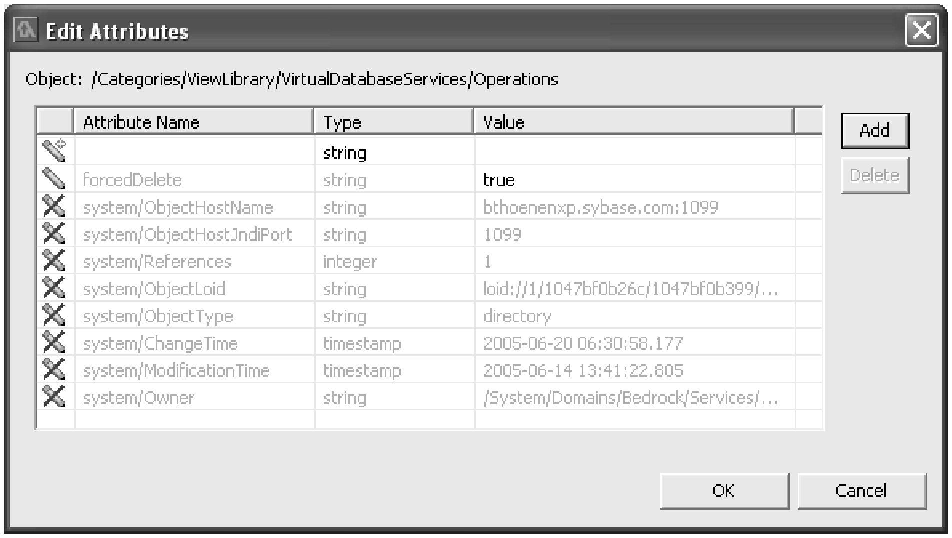
Attribute Name	Type	Value
forcedDelete	string	true
system/ObjectHostName	string	bthoenenxp.sybase.com:1099
system/ObjectHostJndiPort	string	1099
system/References	integer	1
system/ObjectLoid	string	loid://1/1047bf0b26c/1047bf0b399/...
system/ObjectType	string	directory
system/ChangeTime	timestamp	2005-06-20 05:53:26.78
system/ModificationTime	timestamp	2005-06-14 13:41:22.805
system/Owner	string	/System/Domains/Bedrock/Services/...

Notice that the Attribute Name and Type fields and many of the Value fields are grayed out and uneditable (above). You cannot change the name or data type of any attribute. Nor can you edit the values of system attributes; system attributes are displayed only for information. What can you edit? Only the values that are not grayed out—the values of user attributes. And only an administrator or the owner of an object can edit the values of its user attributes.

Step 2 To edit an attribute value, click it and type a new value. Be sure to respect the type. If you enter a word for an attribute that calls for an integer value (for example), Studio will suffer existential angst and display an error.

Note At any point, you can click **Cancel** to dismiss the Edit Attributes dialog without saving your changes, or click **OK** to dismiss the Edit Attributes dialog and save your changes.

Step 3 To add a new attribute to the object, click the **Add** button. Studio adds a row to the table.

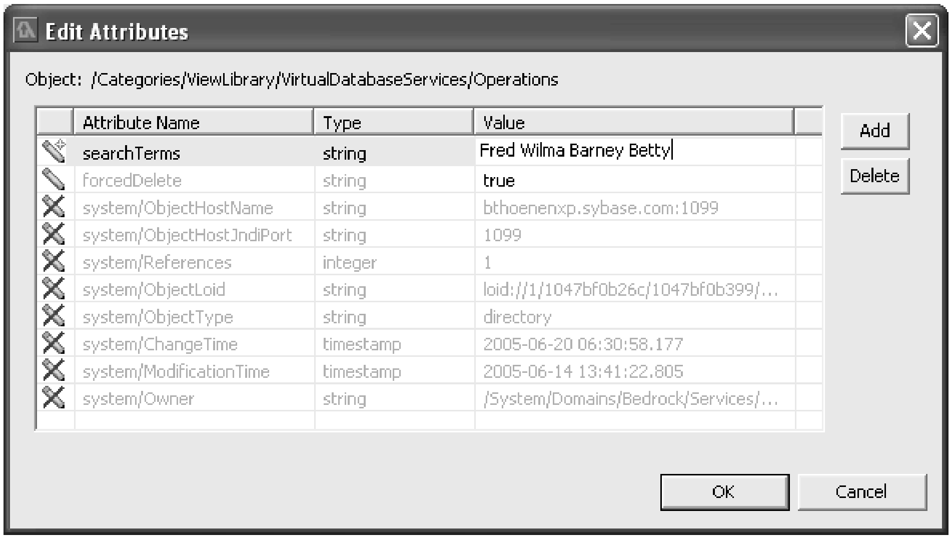


Edit Attributes

Object: /Categories/ViewLibrary/VirtualDatabaseServices/Operations

Attribute Name	Type	Value
	string	
forcedDelete	string	true
system/ObjectHostName	string	bthoenenxp.sybase.com:1099
system/ObjectHostJndiPort	string	1099
system/References	integer	1
system/ObjectLoid	string	loid://1/1047bf0b26c/1047bf0b399/...
system/ObjectType	string	directory
system/ChangeTime	timestamp	2005-06-20 06:30:58.177
system/ModificationTime	timestamp	2005-06-14 13:41:22.805
system/Owner	string	/System/Domains/Bedrock/Services/...

Step 4 Enter a name for your attribute in the Attribute Name field. In the Type field, select a data type from the pull-down menu. Enter a value in the Value field. In this example, we add an attributed called searchTerms whose type is string. Other users will be able to find this object by searching on any of the words in the value.

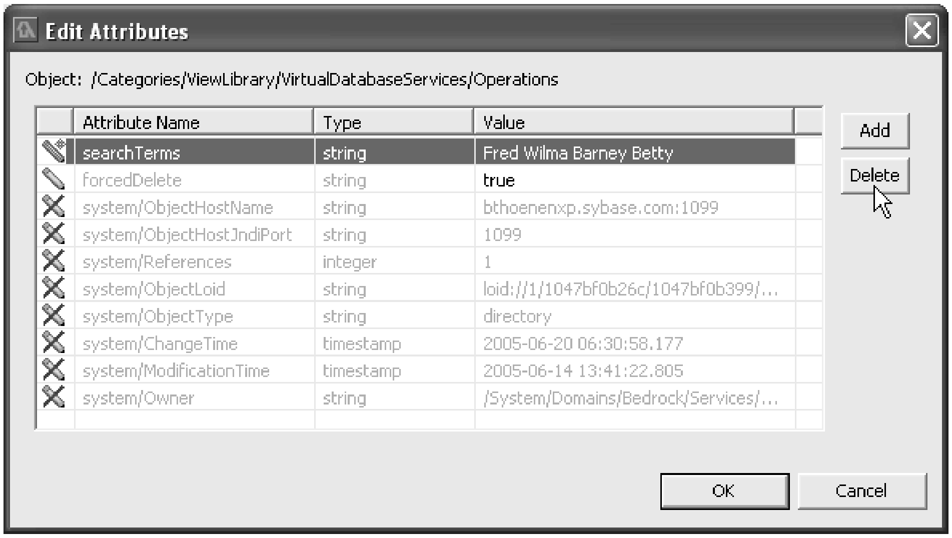


Edit Attributes

Object: /Categories/ViewLibrary/VirtualDatabaseServices/Operations

Attribute Name	Type	Value
searchTerms	string	Fred Wilma Barney Betty
forcedDelete	string	true
system/ObjectHostName	string	bthoenenxp.sybase.com:1099
system/ObjectHostJndiPort	string	1099
system/References	integer	1
system/ObjectLoid	string	loid://1/1047bf0b26c/1047bf0b399/...
system/ObjectType	string	directory
system/ChangeTime	timestamp	2005-06-20 06:30:58.177
system/ModificationTime	timestamp	2005-06-14 13:41:22.805
system/Owner	string	/System/Domains/Bedrock/Services/...

Step 5 To remove an attribute, click its pencil icon to select it and click the **Delete** button.



Object: /Categories/ViewLibrary/VirtualDatabaseServices/Operations

Attribute Name	Type	Value
searchTerms	string	Fred Wilma Barney Betty
forcedDelete	string	true
system/ObjectHostName	string	bthoenenxp.sybase.com:1099
system/ObjectHostJndiPort	string	1099
system/References	integer	1
system/ObjectLoid	string	loid://1/1047bf0b26c/1047bf0b399/...
system/ObjectType	string	directory
system/ChangeTime	timestamp	2005-06-20 06:30:58.177
system/ModificationTime	timestamp	2005-06-14 13:41:22.805
system/Owner	string	/System/Domains/Bedrock/Services/...

Studio removes the attributed from the table and redisplays the dialog.

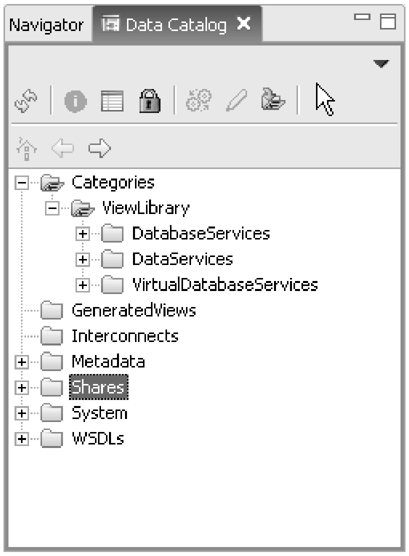
Step 6 Click **OK** to save your changes and dismiss the Edit Attributes dialog.

Managing categories

This section provides instructions on displaying, creating, and deleting categories in the data catalog view, and on adding data catalog objects to and removing them from categories.

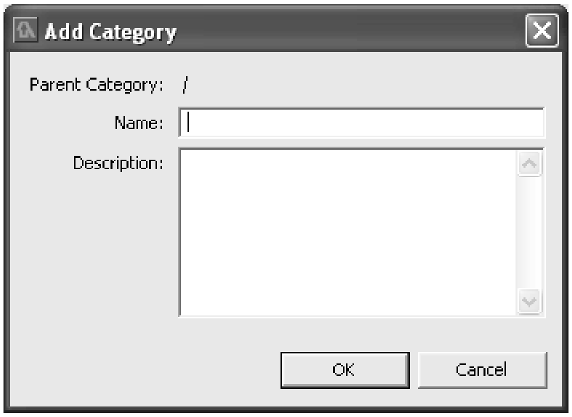
Creating and displaying categories

Step 1 To display categories in the data catalog view, expand the Categories folder. Within Categories is ViewLibrary, and within ViewLibrary are subcategories where Avaki places database connectors, database operations, virtual database operations, SQL views, and data services. (For details, see "Browsing for input sources" on page 18.)



Step 2 To add a category, select the parent category. In a data catalog where no categories have been added yet, there are only two possibilities: Categories and ViewLibrary.

Step 3 Right-click on the parent category and select Add Category from the menu. The Add Category dialog appears.



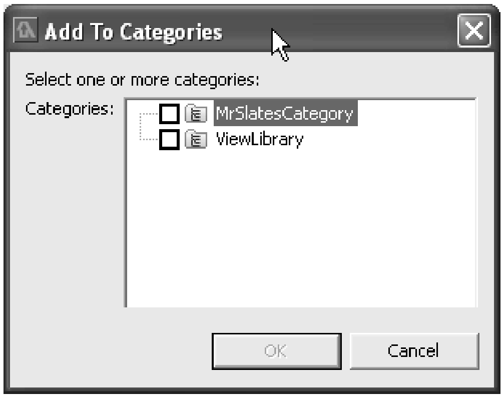
Step 4 In the Name field, enter a name for the new category. Enter a description of the category in the Description field (optional). Studio will display the description as a tool tip whenever a user mouses over your category's name in the data catalog view.

Step 5 Click **OK** to create the category. Studio reports the success or failure of the category creation; click **OK** again to dismiss the notification. The new category appears in the data catalog view.

Adding objects to categories

Step 1 To add a data catalog object (a data service, a folder, or a file, for example) to a category, browse to the object in the data catalog view.

Step 2 Right-click the object and select **Add to Category** from the menu. The Add to Categories dialog appears.



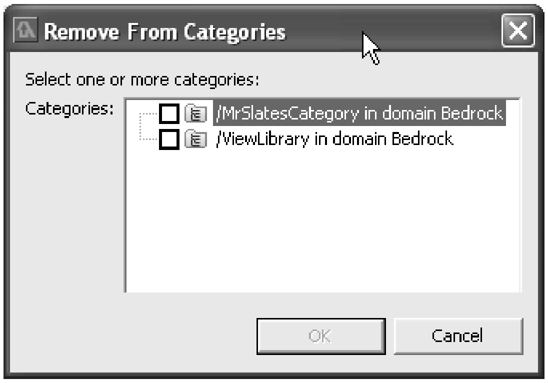
Notice that the root category, Categories, is not listed—you cannot add objects to it (though you can add categories).

Step 3 Select one or more categories.

Step 4 Click **OK** to add the object. Studio reports the success or failure of the category creation; click **OK** again to dismiss the notification. If any of the containing categories are open, the object appears in its new categories.

Removing objects from categories

Step 1 In the data catalog view, right-click the object to be removed and select **Remove from Categories** from the context menu. The Remove From Categories dialog appears.


Remove From Categories

Select one or more categories:

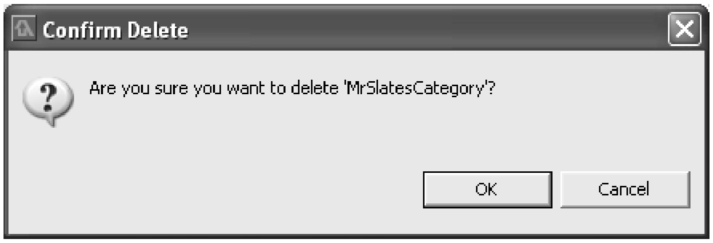
Categories:
- /MrSlatesCategory in domain Bedrock
- /ViewLibrary in domain Bedrock

Step 2 Select the categories from which to remove the object.

Step 3 Click **OK** to remove the object. Studio reports the success or failure of the removal; click **OK** again to dismiss the notification. If any of the containing categories are open, the object disappears from them.

Deleting categories

Step 1 To delete a category, right-click it and select **Delete** from the menu. Studio displays a confirmation dialog.


Confirm Delete

Are you sure you want to delete 'MrSlatesCategory'?

Click **OK** to delete the category. If the containing category is open, the object disappears.

Chapter 7

Operator reference

Consult this chapter for detailed information on the workings of the elements that make up a view model: operators, Input Source elements and Result elements. You should already be familiar with the material in Chapter 4, "The view model editor."

Each element is covered in its own section, which includes a description of the element, information about how it can be connected, details on configuring it, and in some cases, performance notes and usage examples. They are listed alphabetically:

- "Select" on page 165
- "Splitter" on page 166
- "Union" on page 168
- "Update" on page 169

Aggregate

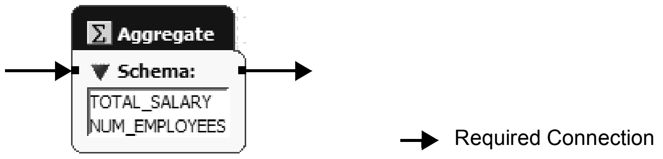
The **Aggregate** operator condenses a multi-row result set into one with a single row by applying any of a number of functions to the data. These functions perform calculations such as totalling or counting the values for each row. The output row has one column containing the result of each function defined. The functions are commonly used to operate on the contents of a single column from the input result set, but the parameter passed to them can actually be the result of an arbitrary JavaScript expression, allowing more complex calculations. The functions which Studio supports for aggregation are Sum, Average, Maximum, Minimum, Count, First, Last, Population Variance, Sample Variance, Population Standard Deviation, and Sample Standard Deviation.

For example, if you have a result set with a column called SALARY containing each employee's salary, you can define a new column using an **Aggregate** operator which calculates the total salary for all rows in the result set using the Sum function and passing the value `SALARY` as the corresponding expression.

Note that the original columns from the input result set are not carried through to the output of the **Aggregate** operator, only the newly computed ones.

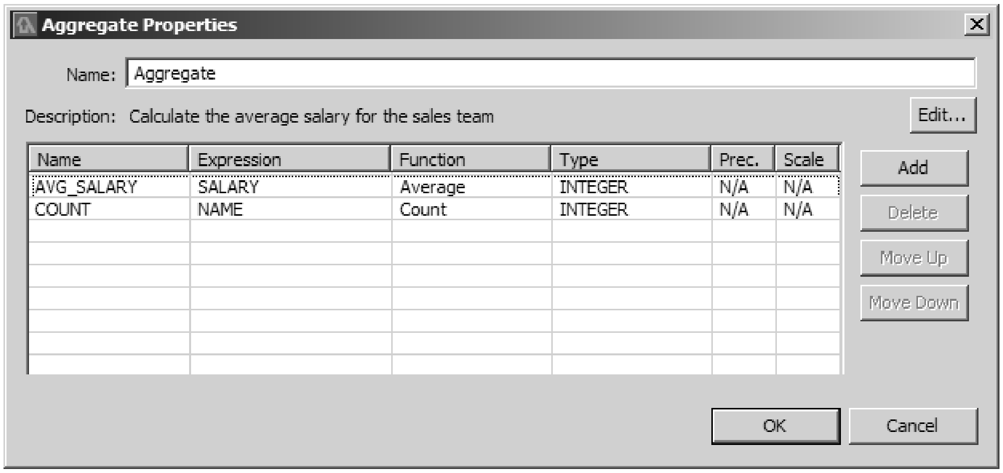
Connections

The **Aggregate** operator takes exactly one input result set, which can be the output of an **Input Source** or of any other operator. It produces exactly one output result set whose schema is defined by the operator.



Aggregate properties

Double-click an **Aggregate** operator to display the Aggregate Properties dialog. You can edit the name and description as described in "Properties dialogs" on page 58.



Name: Aggregate

Description: Calculate the average salary for the sales team

Name	Expression	Function	Type	Prec.	Scale
AVG_SALARY	SALARY	Average	INTEGER	N/A	N/A
COUNT	NAME	Count	INTEGER	N/A	N/A

Defining aggregate columns

Do the following to edit the column definitions of your aggregation.

- Click the **Add** button to add a column definition to your aggregation.
- To delete one or more columns, select them and click **Delete**.
- To reorder the columns, select one or more of them and click the **Move Up** or **Move Down** buttons. (If you want to move multiple columns, the selection must be contiguous.)
- You can edit the properties of each column. (See "Working with schemas" on page 60 for information about the setting the name, type, precision, and scale of columns.)
  - **Name:** Click in a column's Name field to rename it.
  - **Expression:** Click in the column's Expression field to edit it. This may be any JavaScript expression that you want to pass to the aggregate function, including simple column names from the incoming result set. Click the "**...**" button to open the Edit Expression dialog if you need more room than the table cell provides. You can also right click (either in the table cell directly, or in the Edit Expression dialog) to show a context-sensitive menu of column names, variables, and functions that you can paste into the Expression field. For details on using JavaScript expressions, see "Using expressions within Avaki Studio" on page 66. For details on the context menu, see "The expressions menu" on page 71.

- **Function:** Click the Function field and use the pull down list to choose the function you want to use for this column. The value of the Expression field will be passed to this function for each row in the incoming result set. The functions are described under "Aggregate functions" below.
- **Type:** Specify the type of the column. Be sure to specify a numeric type (INTEGER, for example) for aggregate columns that use the Count function.
- **Precision:** Specify the precision for the column, if applicable for the column's type.
- **Scale:** Specify the scale for the column, if applicable for the column's type.

Aggregate functions

The table below lists the available aggregation functions and describes what they do.

Function	**Description**
Sum	Adds the value of `expression` for each row to obtain a total.
Average	Averages the values of `expression` for each row—equivalent to using Sum and dividing by Count.
Maximum	Returns the highest value of `expression` from the input result set.
Minimum	Returns the lowest value of `expression` from the input result set.
Count	Returns the number of rows in the input result set. The value of `expression` is ignored. Note that this will return an integer regardless of the type of the expression.
First	Returns the value of `expression` for the first row in the input result set.
Last	Returns the value of `expression` for the last row in the input result set.
Population variance	Returns the population variance for the values of `expression` in the input result set.
Sample variance	Returns the sample variance for the values of `expression` in the input result set.

Function	Description
Population standard deviation	Returns the population standard deviation for the values of `expression` in the input result set.
Sample standard deviation	Returns the sample standard deviation for the values of `expression` in the input result set.

Custom

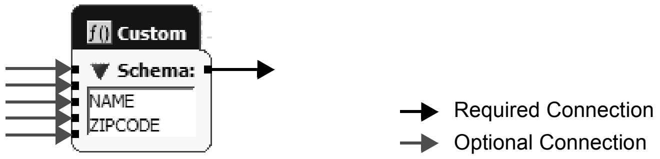
A **Custom** operator can perform any operation that you define, using arbitrary JavaScript code that you supply.

To implement a **Custom** operator, you first define the output schema you will generate. Then you write JavaScript code for the body of a function that will be called once for each output row it is expected to produce. Your code can iterate over the operator's input result sets (there may be zero or more of these) and insert values for each column in the output row according to the defined schema. Your code must return a Boolean value. A return value of true indicates that it has produced a row. It will then be invoked again to produce the next row when the runtime needs it. If your code returns false, it indicates that it has no more rows to produce.

Beyond that, you're free to set up your code as you want—you can precompute the output when the function is called the first time and then feed out the results one by one, simply act as a filter on the input, or do whatever you decide will work best for the functionality you need.

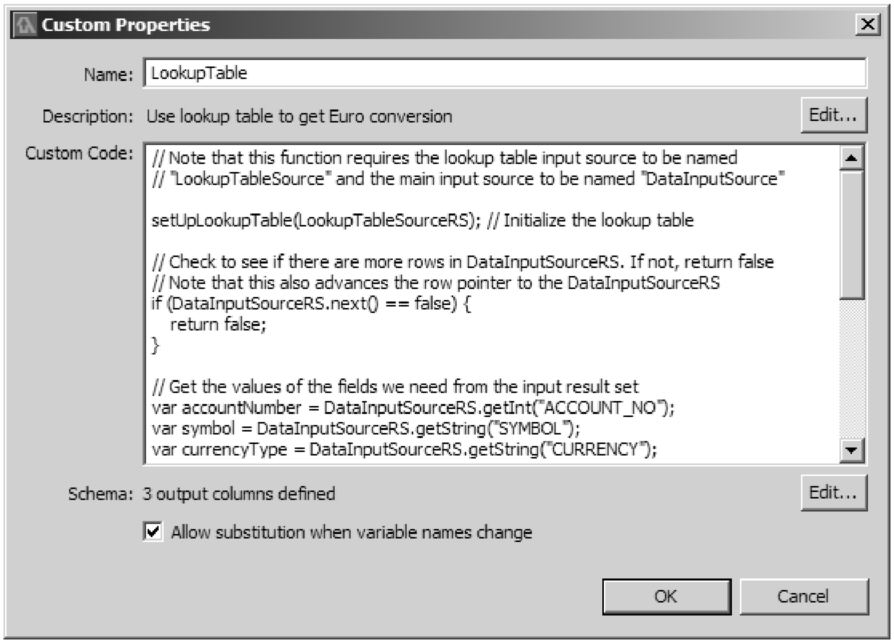
Connections

The **Custom** operator takes anywhere from zero to five input result sets, each of which can be the output of an **Input Source** or of any other operator. It produces exactly one output result set whose schema is defined by the operator.



Custom properties

Double-click a **Custom** operator to display the Custom Properties dialog. You can edit the name and description as described in "Properties dialogs" on page 58.



Name: LookupTable

Description: Use lookup table to get Euro conversion

Custom Code:

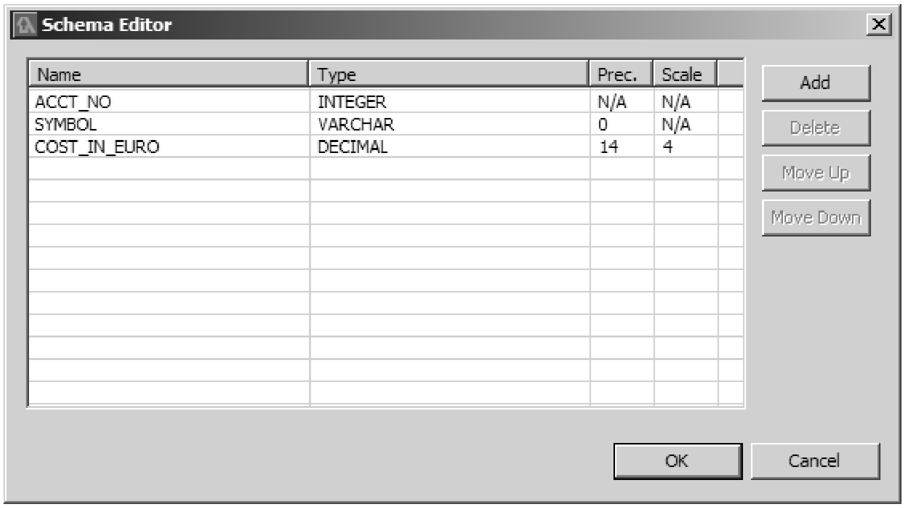
```
// Note that this function requires the lookup table input source to be named
// "LookupTableSource" and the main input source to be named "DataInputSource"

setUpLookupTable(LookupTableSourceRS); // Initialize the lookup table

// Check to see if there are more rows in DataInputSourceRS. If not, return false
// Note that this also advances the row pointer to the DataInputSourceRS
if (DataInputSourceRS.next() == false) {
   return false;
}

// Get the values of the fields we need from the input result set
var accountNumber = DataInputSourceRS.getInt("ACCOUNT_NO");
var symbol = DataInputSourceRS.getString("SYMBOL");
var currencyType = DataInputSourceRS.getString("CURRENCY");
```

Schema: 3 output columns defined

[✓] Allow substitution when variable names change

Specifying the output schema

When you define a **Custom** operator, you need to specify the schema that the output result set will have. You do this using the Schema Editor dialog, which is accessed by clicking the **Edit…** button near the bottom of the Custom Properties dialog.



Name	Type	Prec.	Scale
ACCT_NO	INTEGER	N/A	N/A
SYMBOL	VARCHAR	0	N/A
COST_IN_EURO	DECIMAL	14	4

- Click the **Add** button to add a column definition to your aggregation.

- To delete one or more columns, select them and click **Delete**.
- To reorder the columns, select one or more of them and click the **Move Up** or **Move Down** buttons. (If you want to move multiple columns, the selection must be contiguous.)
- You can edit the properties of each column. (See "Working with schemas" on page 60 for information about the setting the name, type, precision, and scale of columns.)
  - **Name:** Click in the a column's name field to rename it.
  - **Type:** Specify the type of the column.
  - **Precision:** Specify the precision for the column, if applicable for the column's type.
  - **Scale:** Specify the scale for the column, if applicable for the column's type.

Entering custom code

Enter and edit your custom code directly in the Custom Properties dialog. The code you enter forms the body of the function that is called when your **Custom** operator is asked for its next row of data. You can enter any JavaScript code here that would be valid inside a function body. See "Using expressions within Avaki Studio" on page 66 for details on using JavaScript code within Studio.

Your code has access to the `variables` structure, including an object called `variables.<`*`yourCustomOperatorName`*`>`, into which you can insert state variables, using expressions such as:

```
variables.myOperator.foo = 5;
```

In addition, you have access to all the incoming result sets from objects connected to your operator. They are named `<`*`precedingOperatorName`*`>RS` (for example, if an input source is named "CustomerInfo" there will be a variable named `CustomerInfoRS` available to access it). Finally, you have access to the output `ResultSet` object for your operator, which is named `outputRS`. This is actually a subclass of `java.sql.ResultSet` on which you can only *set* values for the columns in the current row. The runtime architecture takes care of incrementing the row each time your operator is called.

If your code needs to call external functions or if you need to import other Java classes in order to do your custom processing, you can add these to the .jsi file for your model. For details, see "Using a .jsi file to enhance your model" on page 75.

Note The **Custom** operator is an exception to the guidelines regarding JavaScript expressions in that you *must* always access the columns of both incoming and output result sets using either the accessor methods on the `ResultSet` object itself or the Avaki utility functions (as listed under "Avaki Functions" on page 73).

Example: Making a lookup table using a Custom operator and a .jsi file

The problem

Suppose you have an input source—a database operation, for example—that returns a result set of transaction data. Unfortunately, the prices in this data are in various currencies, and you need to express the prices in Euros. You have another input result set that has information about current Euro conversion rates.

The columns in the first input result set are:

- ACCT_NO - the account number for the transaction
- SYMBOL - the symbol of the stock purchased
- NO_OF_SHARES - the number of shares of stock purchased
- COST - the cost per share
- CURRENCY - the currency in which COST is expressed

The columns in the second input result set are:

- CURRENCY - the currency for which the conversion factor applies
- MULTIPLIER - the factor by which the original value is multiplied to get the Euro value

Your goal is to produce a result set consisting of:

- ACCT_NO - the account number for the transaction
- SYMBOL - the symbol of the stock purchased
- COST_IN_EURO - the cost in Euros of the transaction

One possible solution

One way to produce the result that you need (and probably the most straightforward way to use Studio) would be to use a **Join** operator, employing the "Inner Loop Join" algorithm with the second input result set as the inner table. This would produce a result set consisting of:

- ACCT_NO - the account number for the transaction
- SYMBOL - the symbol of the stock purchased
- NO_OF_SHARES - the number of shares of stock purchased
- COST - the cost per share
- CURRENCY - the currency in which COST is expressed (from the first input)
- CURRENCY - the currency for which the conversion factor applies (from the second input)
- MULTIPLIER - the factor by which the original value is multiplied to get the Euro value

You could then use a **Projection** operator to remove the unneeded columns and to compute the COST_IN_EURO column (the formula being `NO_OF_SHARES * COST * MULTIPLIER`).

Solving the problem with a Custom operator

An alternative implementation for this same functionality is to use a **Custom** operator to build a lookup table that maps currency to multiplier and then performs the computation as you read in the account result set.

Note This technique is presented here to illustrate, using a relatively simple problem, how to use **Custom** operators—not necessarily as a recommendation for best practices.

This example uses a .jsi file as well as a Custom operator. The .jsi file contains the following JavaScript:

```
var lookupTable = {}; // This will contain the mapping of currency type to
                         conversion factor.
function setUpLookupTable(rs) {
    if (lookupTable.initialized == true) {
        return; // We only need to read the result set into the map once
    }
    while (rs.next()) { // Iterate over the rows in the result set
        var key = rs.getString(rs, "CURRENCY"); // Get the currency type
        var value = rs.getRealValue("MULTIPLIER"); // Get the conversion factor
        lookupTable[key] = value; // Save the key/value pair
    }
    lookupTable.initialized = true; // So we don't try and do this again
}
```

This code reads the “currency to exchange rate” result set into a hash table. The Custom operator has two inputs—one is called “LookupTableSource,” and the other is called “DataInputSource.”

The logic in the **Custom** operator itself is as follows (remember that this is a function with a Boolean return value—Studio supplies the outer braces, you only type the function body):

```
// Note that this function requires the lookup table input source to be named
// "LookupTableSource" and the main input source to be named "DataInputSource"

setUpLookupTable(LookupTableSourceRS); // Initialize the lookup table

// Check to see if there are more rows in DataInputSourceRS. If not, return false
// Note that this also advances the row pointer to the DataInputSourceRS
if (DataInputSourceRS.next() == false) {
    return false;
}

// Get the values of the fields we need from the input result set
var accountNumber = DataInputSourceRS.getInt("ACCOUNT_NO");
var symbol = DataInputSourceRS.getString("SYMBOL");
var currencyType = DataInputSourceRS.getString("CURRENCY");
var numShares = DataInputSourceRS.getInt( "NO_OF_SHARES");

var costPerShare = DataInputSourceRS.getFloat("COST");

// Look up the Euro conversion factor for the original currency
var conversionFactor = lookupTable[currencyType];

// Compute the value of the transaction, in Euros
var totalValue = numShares * costPerShare * conversionFactor;

// Store the values in the output result set
outputRS.updateInt("ACCOUNT_NO", accountNumber);
outputRS.updateString("SYMBOL", symbol);
outputRS.updateString("COST_IN_EURO", totalValue);

return true; // Let the runtime know we provided another row
```

Generator

A **Generator** operator produces a result set without input. Like an **Iterator**, a **Generator** lets you specify JavaScript expressions for initializing variables, updating variables, and testing for termination. Unlike an **Iterator**, a **Generator** does not accept inputs. Like a **Projection** operator, a **Generator** allows you to specify the names and types of the output columns, as well as JavaScript expressions for computing the values in each row.

When a **Generator** is first called, it executes the Initialize expression you supply to initialize variables. On each call, **Generator**

1. Tests the termination condition
2. If the termination condition has not been met, it generates a row with the designated schema by executing all of the user-specified JavaScript expressions
3. Executes the Increment expression you supply to update variables.

The output of the **Generator** is the sequence of rows created in these iterations.

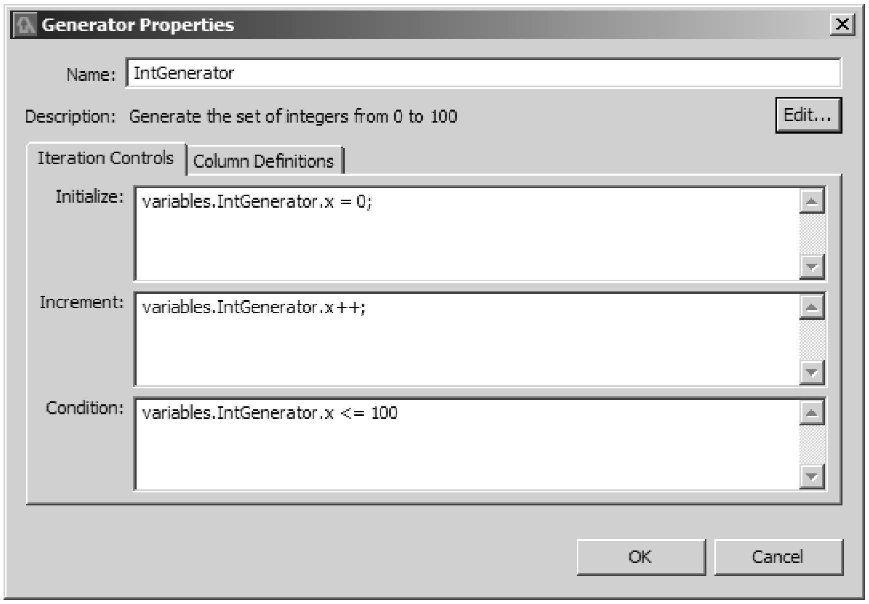
Connections

The **Generator** operator takes zero input result sets. It produces exactly one output result set whose schema is defined by the operator.

Generator
Schema:
CUSTOMER_ID
TRANSACTION_ID
Required Connection

Generator properties

Double-click a **Generator** operator to display the Generator Properties dialog. You can edit the name and description as described in "Properties dialogs" on page 58.



Name: IntGenerator

Description: Generate the set of integers from 0 to 100

Initialize: variables.IntGenerator.x = 0;

Increment: variables.IntGenerator.x++;

Condition: variables.IntGenerator.x <= 100

Your code has access to the `variables` structure, including an object called `variables.<`*`yourGeneratorOperatorName`*`>`, into which you can insert state variables, using expressions such as:

```
    variables.myOperator.foo = 5;
```

If your code needs to call external functions or if you need to import other Java classes in order to do your custom processing, you can add these to the .jsi file for your model. For details, see "Using a .jsi file to enhance your model" on page 75.

As an example, let's assume you wish to create a **Generator** that will output data with a single integer column whose values range from 0 to 100.

Entering iteration control code

You control the starting, intermediate, and ending values of your generated column values using code you enter into the Iteration Controls tab in the Generator Properties dialog. There are three fields, which you define as follows:

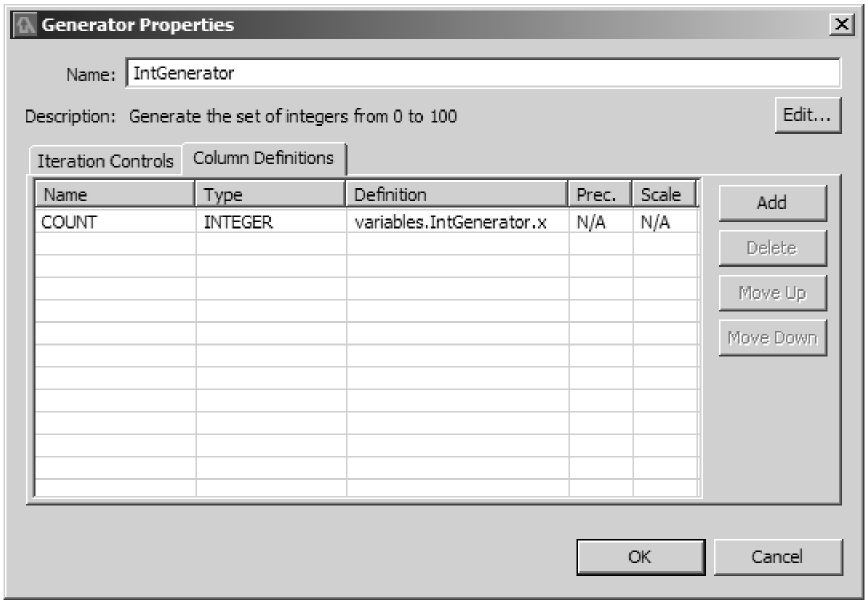
Initialize: This field contains expressions which set up any variables you wish to use in order to calculate the values of each row you are generating. In our simple example, you would initialize a counter variable to zero. You can enter any JavaScript code here that would be valid inside a function body. See "Using expressions within Avaki Studio" on page 66 for details on using JavaScript code within Studio.

Increment: This field contains expressions which increment the variables you initialized in the previous field. In our example, we would simply increment our counter variable. Here, too, you can enter any JavaScript code here that would be valid inside a function body. See "Using expressions within Avaki Studio" on page 66 for details on using JavaScript code within Studio.

Condition: This field contains a Boolean expression. While it evaluates to true, the operator will continue to be called to provide additional rows of data. When it evaluates to false, the iteration will cease. In our example, we would want to check for our counter being less than or equal to the value 100. See "Boolean expressions" on page 66 for details on constructing Boolean expressions.

Defining the output columns

When you define a **Generator** operator, you need to specify the schema that the output result set will have, as well as the values the columns will have for each iteration through the loop. You do this using the Column Definitions tab in the Generator Properties dialog.



Name: IntGenerator

Description: Generate the set of integers from 0 to 100

Name	Type	Definition	Prec.	Scale
COUNT	INTEGER	variables.IntGenerator.x	N/A	N/A

- Click the **Add** button to add a column definition to your schema.
- To delete one or more columns, select them and click **Delete**.

- To reorder the columns, select one or more of them and click the **Move Up** or **Move Down** buttons. (If you want to move multiple columns, the selection must be contiguous.)
- You can edit the properties of each column. (See "Working with schemas" on page 60 for information about the setting the name, type, precision, and scale of columns.)
  - **Name:** Click in the a column's name field to rename it.
  - **Definition:** Click in a column's definition field to enter an expression that will be evaluated each time through the loop to set the value of the column. The expression should evaluate to a value that's appropriate for the column type you choose. For our example, we would simply put the variable (fully qualified) in the Definition field. When run, this column would have the values 0, 1, 2, 3, and so on. For details on writing expressions, see "Using expressions within Avaki Studio" on page 66.
  - **Type:** Specify the type of the column. For our example, we would choose a numeric type such as INTEGER.
  - **Precision:** Specify the precision for the column, if applicable for the column's type.
  - **Scale:** Specify the scale for the column, if applicable for the column's type.

Group By

The **Group By** operator is very similar to the **Aggregate** operator. Its output includes columns generated with the same aggregation functions. The difference is that whereas the **Aggregate** operator applies these functions to the entire set of input rows, the **Group By** divides the input rows up into groups based on criteria you specify and then applies the aggregation functions to the subset of rows in each group. This creates one row for each group rather than the **Aggregate** operator's single row of output for the entire result set.

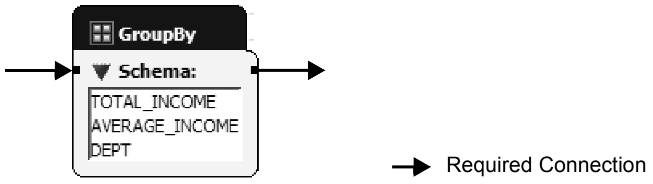
The aggregate columns in the **Group By** operator are defined identically to those in the **Aggregate** operator. See "Aggregate" on page 108 for details. To define the column or columns that will determine the grouping, you simply pick one or more of them from the input source. A group consists of all rows of the input where the values for these columns are identical. For example, if your input, which describes employee information, contains a column named DEPT_NO which is that employee's department number, you can group by that column and then obtain aggregate information for the employees in each department. If you specify more than one group-by column, the values for those columns must be the same throughout all rows of the group.

You also have the option when defining the group-by rows of including that column in the output result set. In the example above, therefore, we could choose to include the DEPT_NO column in our result set. Otherwise, only the columns calculated based on the aggregate functions are included.

The aggregate functions available for use are the same as in the Aggregate operator: Sum, Average, Maximum, Minimum, Count, First, Last, Population Variance, Sample Variance, Population Standard Deviation, and Sample Standard Deviation. They are described in detail under "Aggregate functions" on page 110.

Connections

The **Group By** operator takes exactly one input result set, which can be the output of an **Input Source** or of any other operator. It produces exactly one output result set whose schema is defined by the operator.

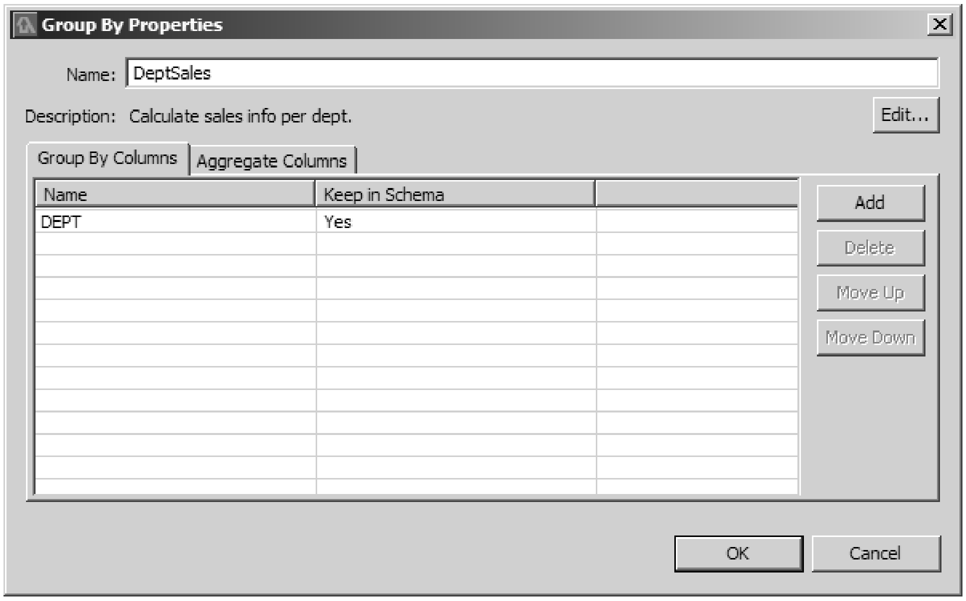


Performance notes

Because Studio needs to sort the result set in order to perform a grouping, the same performance issues described for the **Order By** operator apply to the **Group By** operator. See "Performance notes" on page 159 for details.

Group By properties

Double-click a **Group By** operator to display the Group By Properties dialog. You can edit the name and description as described in "Properties dialogs" on page 58.



Group By Properties

Name: DeptSales

Description: Calculate sales info per dept.

Name	Keep in Schema	
DEPT	Yes	

Specifying groups

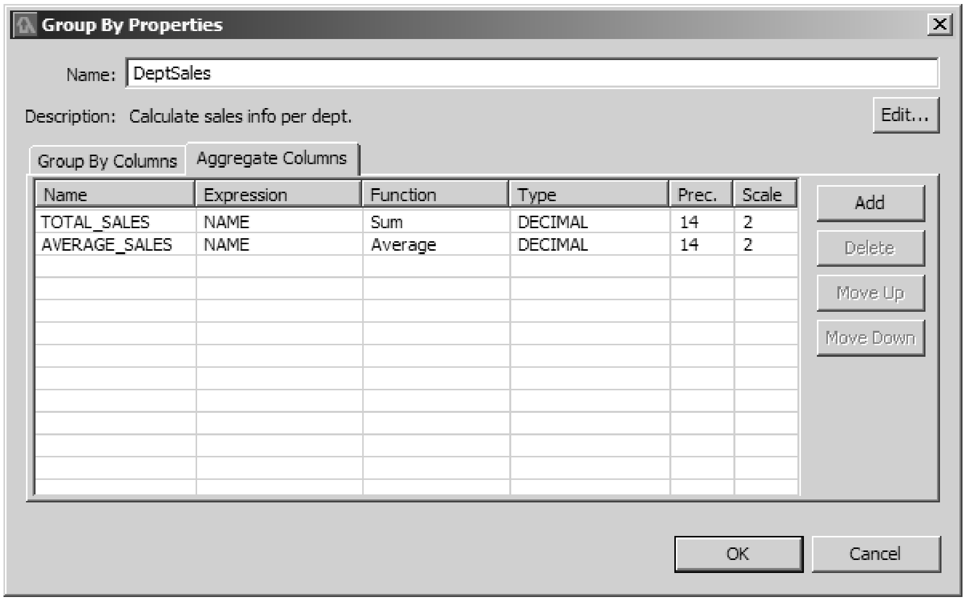
On the Group By Columns tab of the properties dialog, you list the columns from the input that the operator will use to group the incoming rows. You may note the similarity to the way the **Order By** operator specifies the columns used for sorting (see "Order By" on page 159 for more information).

- Click the **Add** button to add a column criterion to your groupings.
- To delete one or more columns, select them and click **Delete**.
- To rearrange the columns, select one or more of them and click the **Move Up** or **Move Down** buttons. (Note that if you want to move multiple columns, the selection must be contiguous.)

- You can edit the properties of each column:
  - **Name:** Click in the Name field and choose a column name from the pull down list to change it. (When you add a new column, Studio chooses the next available column from the input, which may not necessarily be the column you want to group by).
  - **Keep in Schema:** This determines whether the column in question will appear in the output schema of the **Group By** operator. To change its value, click in the field and choose either "Yes" or "No" from the pull down.

Specifying aggregate columns

Specify aggregate columns to calculate using the Aggregate Columns tab in the Group By Properties dialog. This tab functions exactly the same as the column definition section in the Aggregate Properties dialog (see "Aggregate" on page 108 for details).



Name: DeptSales

Description: Calculate sales info per dept.

Name	Expression	Function	Type	Prec.	Scale
TOTAL_SALES	NAME	Sum	DECIMAL	14	2
AVERAGE_SALES	NAME	Average	DECIMAL	14	2

To edit the column definitions of your aggregation, do the following:

- Click the **Add** button to add a column definition to your aggregation.
- To delete one or more columns, select them and click **Delete**.
- To reorder the columns, select one or more of them and click the **Move Up** or **Move Down** buttons. (If you want to move multiple columns, the selection must be contiguous.)

- You can edit the properties of each column. (See "Working with schemas" on page 60 for information about setting the name, type, precision, and scale of columns.)
  - **Name:** Click in the a column's name field to rename it.
  - **Expression:** Click in the column's Expression field to edit it. This may be any JavaScript expression that you want to pass to the aggregate function, including simple column names from the incoming result set. Click the "**…**" button to open the Edit Expression dialog if you need more room than the table cell provides. You can also right click (either in the table cell directly, or in the Edit Expression dialog) to show a context-sensitive menu of column names, variables, and functions that you can paste into the Expression field. For details on using JavaScript expressions, see "Using expressions within Avaki Studio" on page 66. For details on the context menu, see "The expressions menu" on page 71.
  - **Function:** Click the Function field and use the pull down list to choose the function you want to use for this column. The value of the Expression field will be passed to this function for each row in the incoming result set. The functions are described under "Aggregate functions" on page 110.
  - **Type:** Specify the type of the column. Be sure to specify a numeric type (e.g. INTEGER) for aggregate columns which use the Count function.
  - **Precision:** Specify the precision for the column, if applicable for the column's type.
  - **Scale:** Specify the scale for the column, if applicable for the column's type.

Input Source

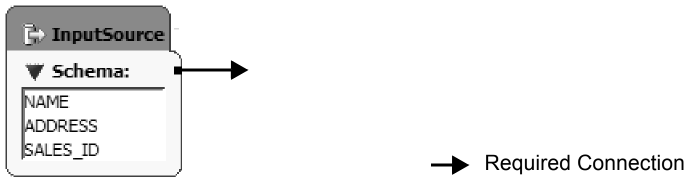
An **Input Source** element represents input from a source external to your model. While some input sources take their input from nonrelational data, all of them have a single output in which their data is formatted as a result set, as if it had come from a relational database query. To configure an **Input Source** element, you must specify the original source of the data, set up a transformation if the data isn't already in result set format, and specify values for any parameters the input requires.

To create input sources that are based on elements in your Avaki domain, you can drag and drop items directly from the Data Catalog View in Studio into your view model. These include database operations, virtual database operations, other data services, and files. Studio attempts to do as much of the configuration as possible. (For example, if you drag a data service that takes no parameters into your model, the resulting **Input Source** element will be fully configured. If you drag an XML file into your view model, you must still configure the transformation to construct a result set from the XML data.)

Note If you need to create input rows that don't come from an external input source, use the **Generator** operator—see "Generator" on page 117.

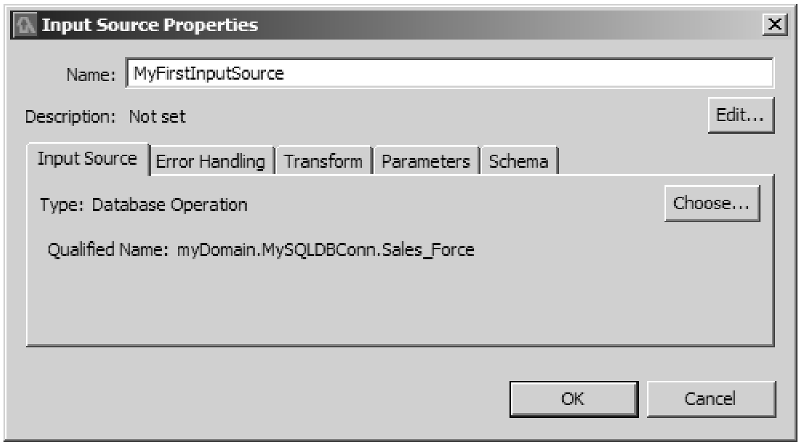
Connections

The **Input Source** element takes zero input result sets. It produces exactly one output result set whose schema is defined either by the source itself (for relational inputs) or by the transform you define (for non-relational inputs).



Input Source Properties

Double-click an **Input Source** element to display the Input Source Properties dialog. You can edit the name and description as described in "Properties dialogs" on page 58.



Input Source Properties

Name: MyFirstInputSource

Description: Not set

Type: Database Operation

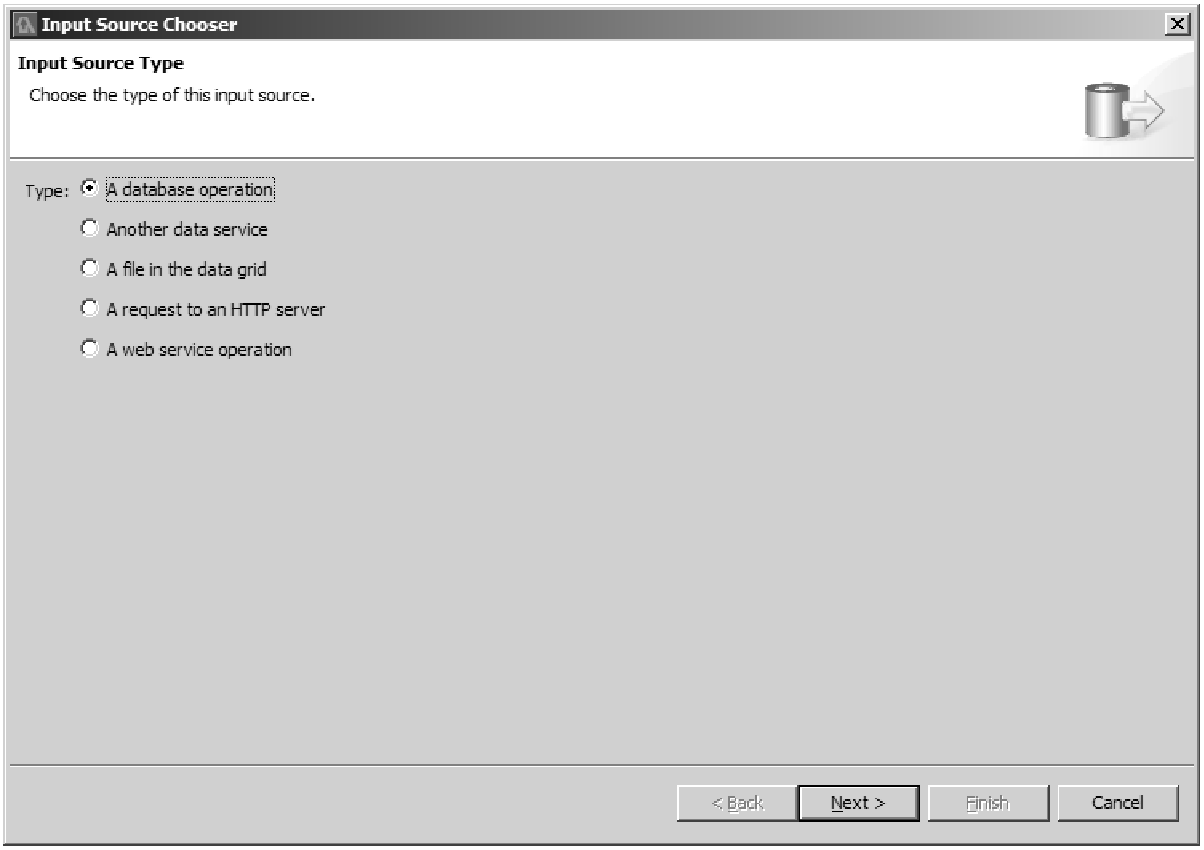
Qualified Name: myDomain.MySQLDBConn.Sales_Force

Configuring the source

The source for your **Input Source** element can be any one of the following:

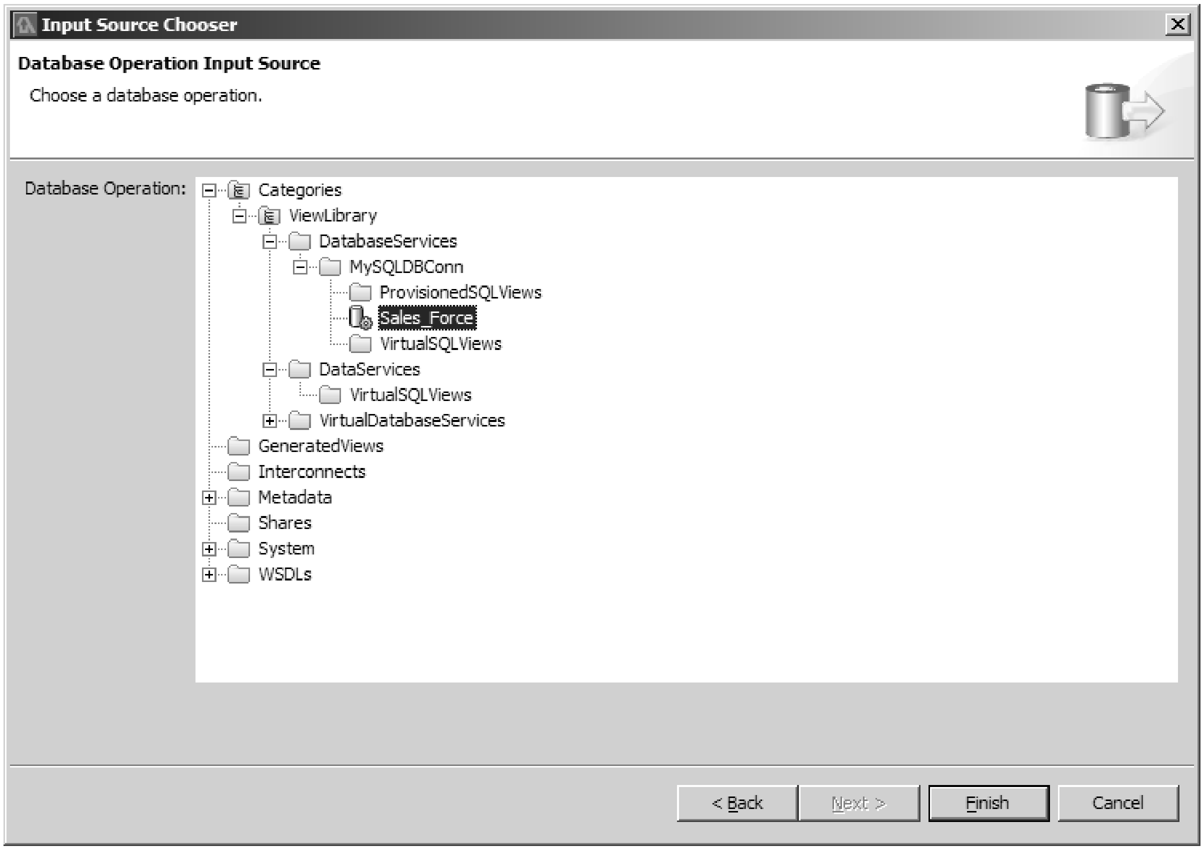
- An Avaki database operation or virtual database operation
- An Avaki data service that produces result set or XML output
- A data catalog file in XML or CSV format
- A web services call

You specify the source for your **Input Source** element using the **Input Source** tab. The tab provides a summary of the current settings. To change the values, click the **Choose…** button to launch the Input Source Chooser wizard.



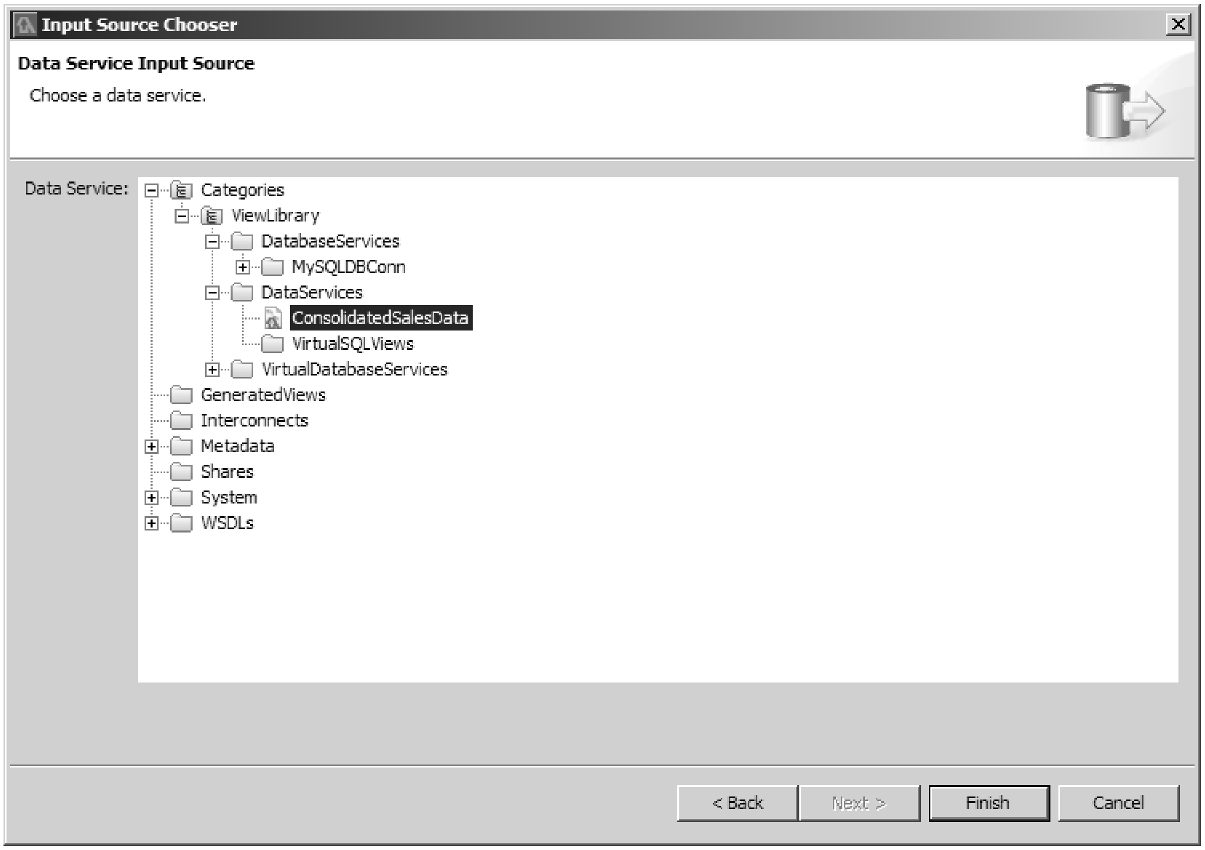
Choose the type of input you want to use and click **Next**. Depending on the type you choose, you will see a different series of pages in the wizard.

Configuring database operation input sources. On the next page ("Database Operation Input Source"), you choose a database operation (or virtual database operation) from the browser.



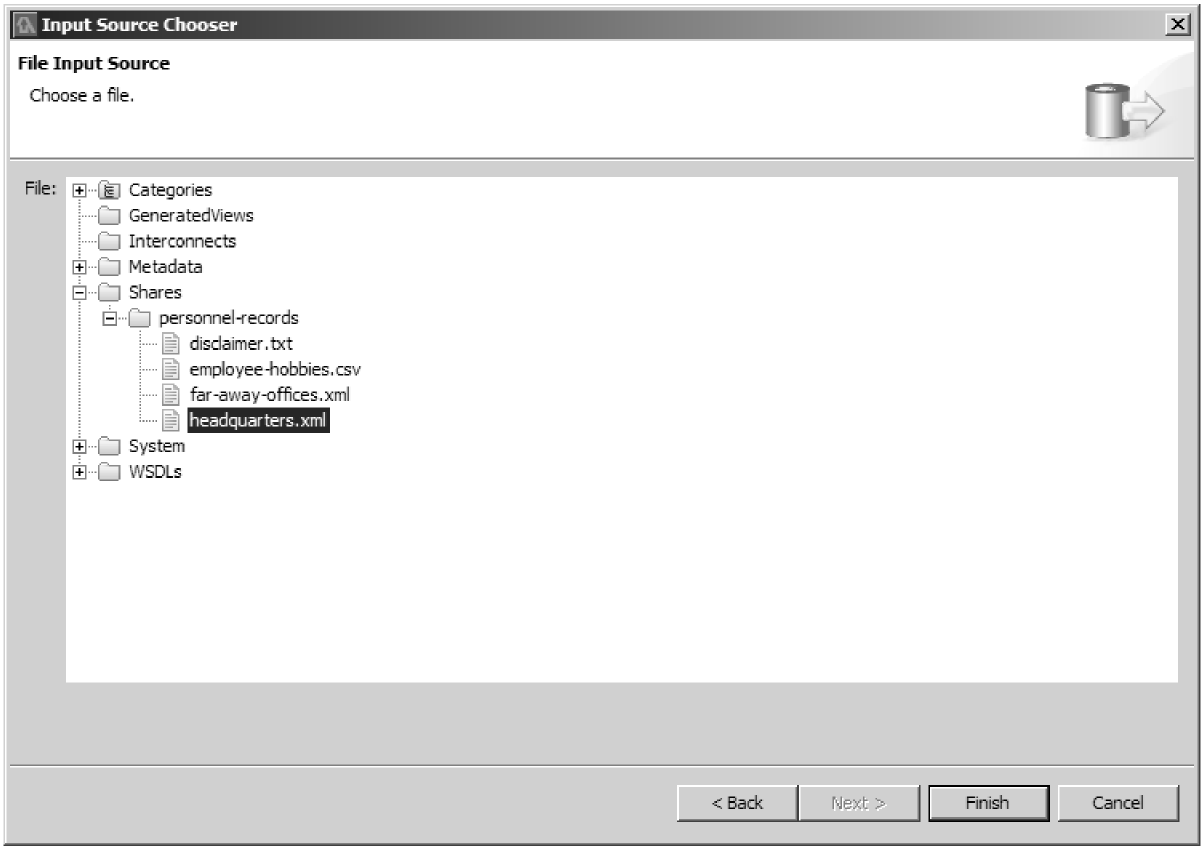
Browse to select the operation you want and click the **Finish** button to close the wizard.

Configuring data service input sources. On the next page (“Data Service Input Source”), you choose a data service from the browser.



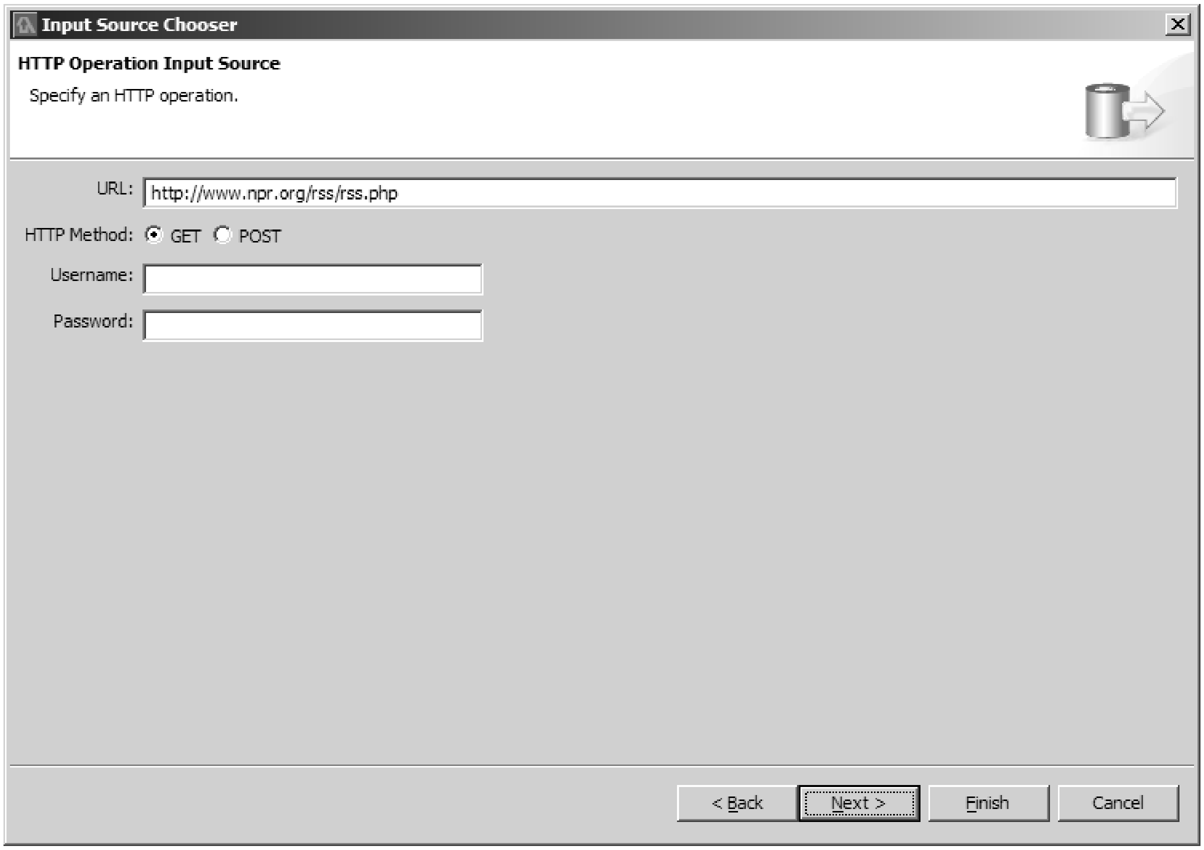
Browse to select the service you want and click the **Finish** button to close the wizard.

Configuring file input sources. On the next page ("File Input Source"), you choose a grid file from the browser.



Browse to select the file you want and click the **Finish** button to close the wizard.

Configuring HTTP input sources. On the next page ("HTTP Operation Input Source"), you specify information about the HTTP server you wish to contact.



URL: http://www.npr.org/rss/rss.php

HTTP Method: [✓ GET] POST

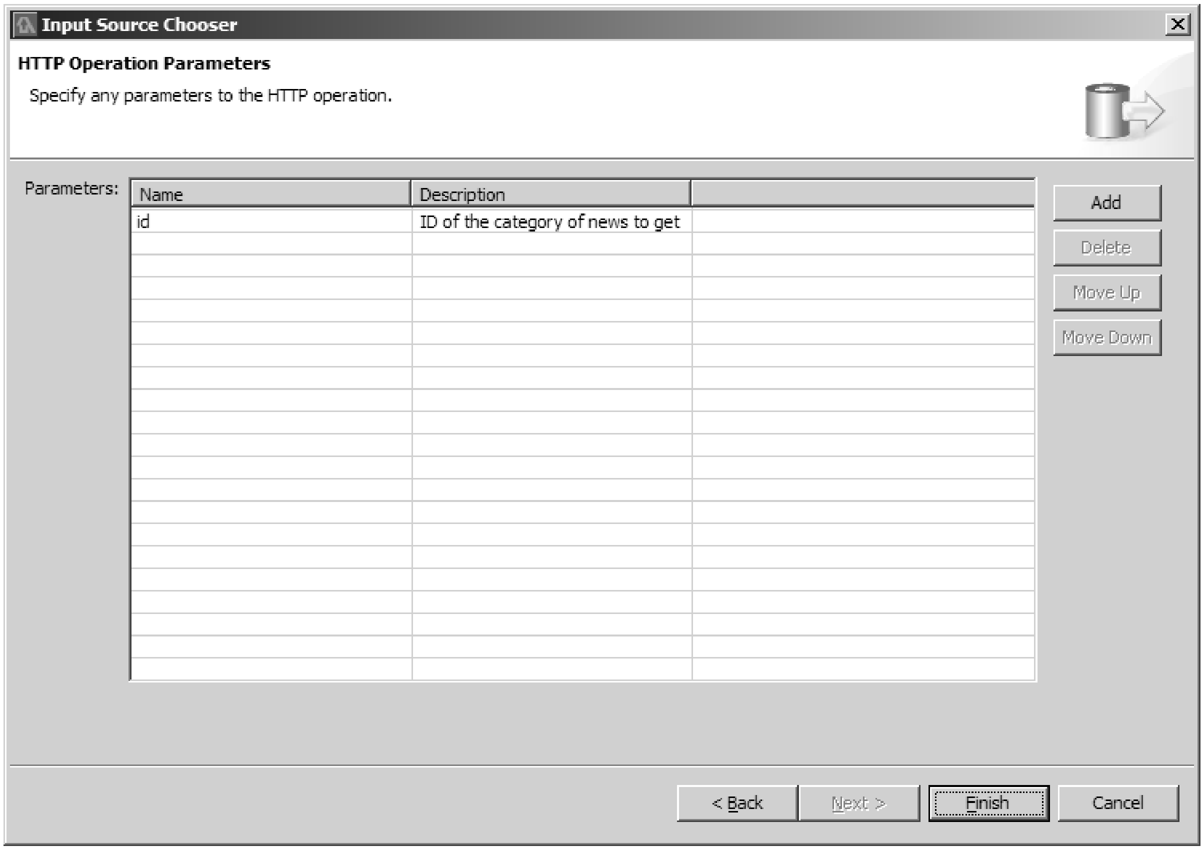
Username:

Password:

Fill in the fields as follows:

- **URL:** Enter the URL for to the HTTP source.
- **HTTP Method:** The method determines how parameters to the HTTP request get sent. Choose **GET** to encode the parameters as part of the URL or choose **POST** to include the parameters in the HTTP headers. For most HTTP requests, you will want to choose the default GET method.
- **Username:** (Optional) Enter the name to use when logging in to the HTTP server if the server requires it. If the server requires no authentication, leave this blank.
- **Password:** (Optional) Enter the password for the user account specified in the Username field. If the username is blank, this field is ignored.

When you are done with this page, click **Next** to proceed to the “HTTP Operation Parameters” page. (If your HTTP operation doesn’t require any parameters, you can click **Finish** now to close the wizard.)



Specify any parameters to the HTTP operation.

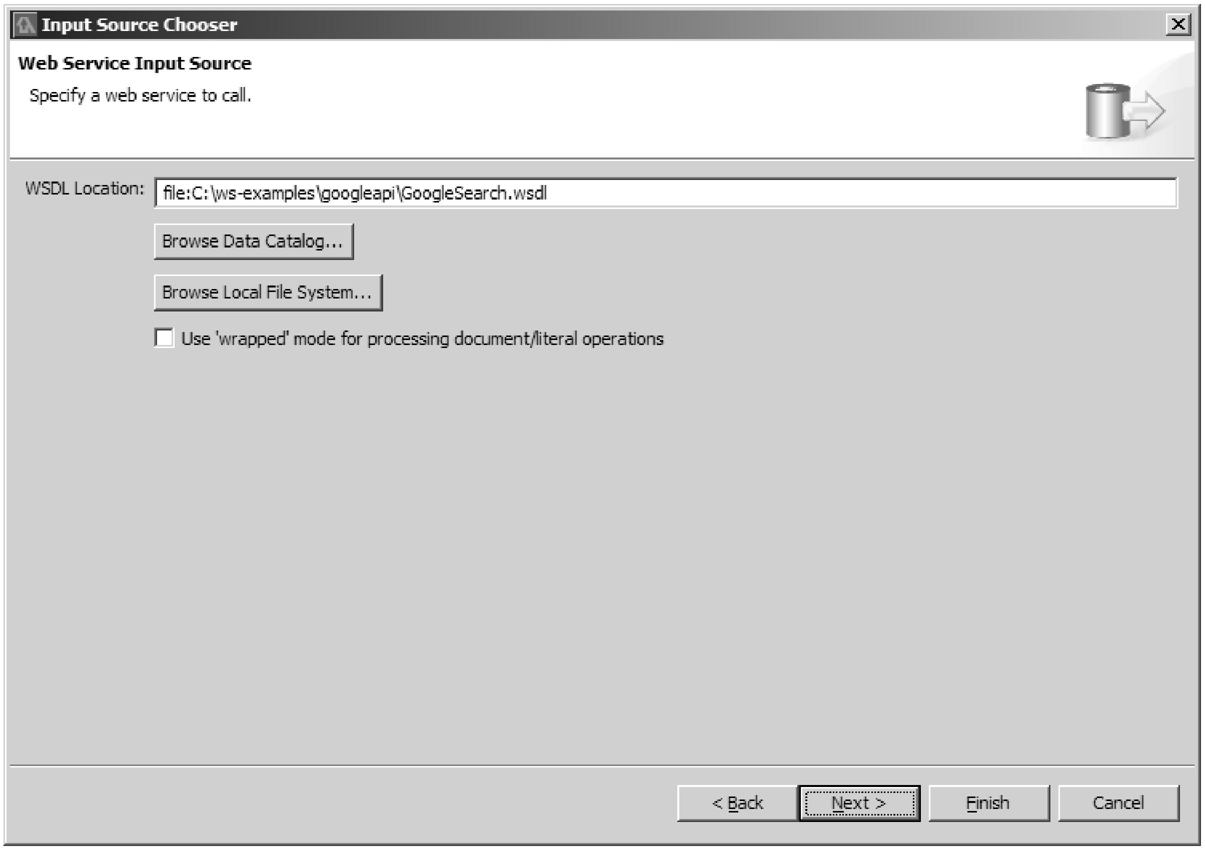
Parameters:

Name	Description	
id	ID of the category of news to get	

- Click the **Add** button to add a parameter.
- To delete one or more parameters, select them and click **Delete**.
- To rearrange the parameters, select one or more of them and click the **Move Up** or **Move Down** buttons. (Note that if you wish to move multiple parameters, the selection must be contiguous.)
- You can edit the properties of each parameter:
  - **Name:** Click in the Name field and type a name for the parameter.
  - **Description:** (Optional) To add a description for the parameter, click in the description column and type one.

When you are done with this page, click **Finish** to close the wizard.

Configuring web service input sources. On the next page ("Web Service Input Source"), you specify information about the web service you want to call.



Input Source Chooser

Web Service Input Source

Specify a web service to call.

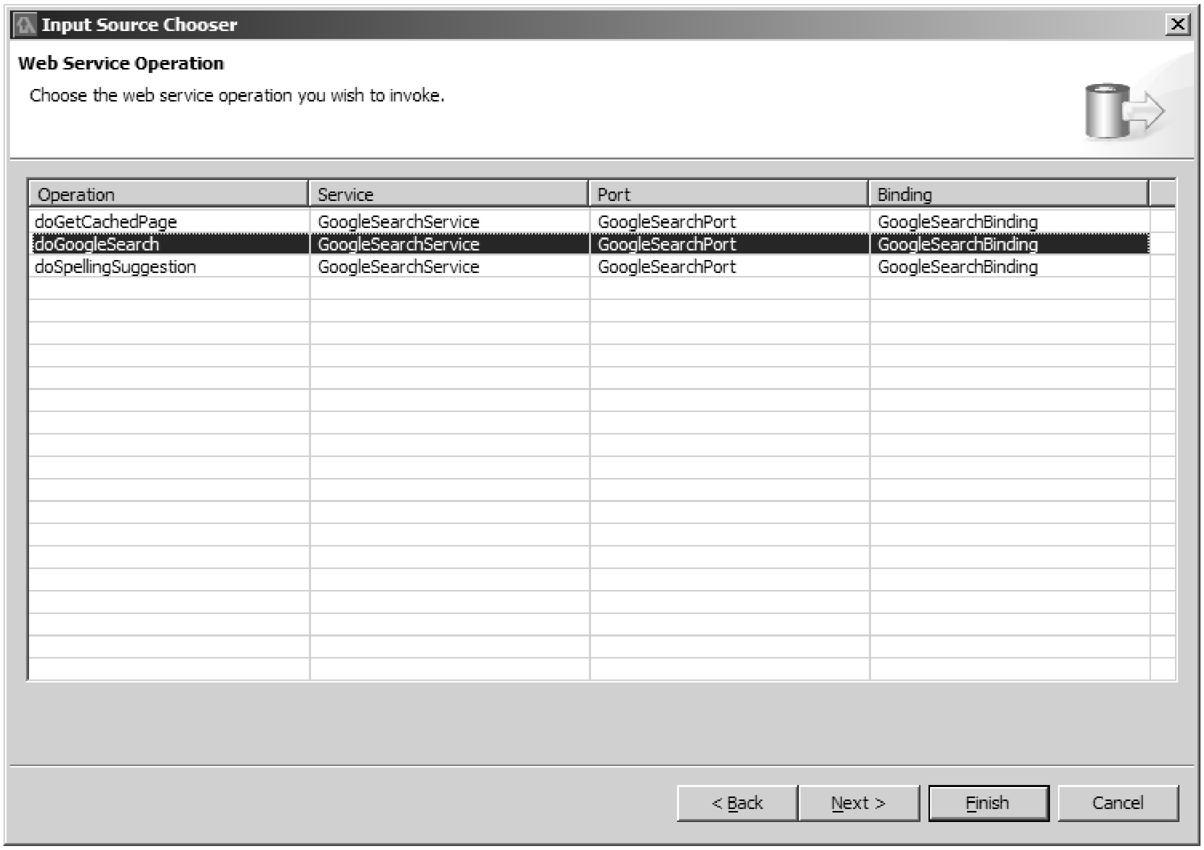
WSDL Location: file:C:\ws-examples\googleapi\GoogleSearch.wsdl

☐ Use 'wrapped' mode for processing document/literal operations

Fill in the fields as follows:

- **WSDL Location:** This is the URL to the WSDL (Web Service Description Language) file for the web service you want to call. The file may be on an HTTP server (you would specify it by typing an "http://" URL), in the Avaki domain (you would browse for it using the **Browse Data Catalog…** button or type an "avaki://" URL for it), or on your local file system (you would browse for it using the **Browse Local File System…** or type a "file://" URL for it).
- **Use 'wrapped' mode…:** If your web service uses the document/literal (as opposed to rpc/encoded) format, you would check this box to use the 'wrapped' mode for passing the parameters. Most document/literal web services will expect this—if you are experiencing trouble calling your document/literal web service, try changing the value of this checkbox. If the web service uses the rpc/encoded format, this checkbox has no effect.

Click the **Next** button to continue to the “Web Service Operation” page where you will choose the operation you want.

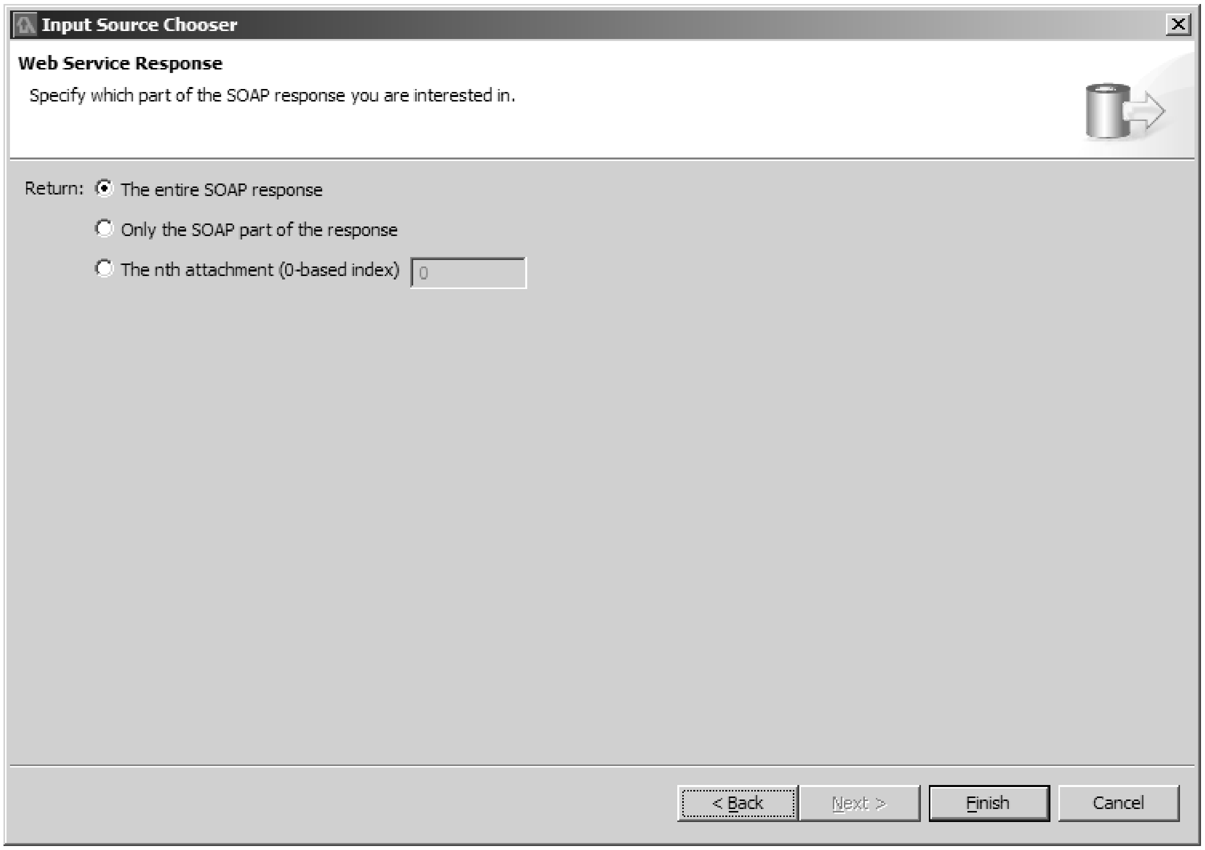


Web Service Operation

Choose the web service operation you wish to invoke.

Operation	Service	Port	Binding
doGetCachedPage	GoogleSearchService	GoogleSearchPort	GoogleSearchBinding
doGoogleSearch	GoogleSearchService	GoogleSearchPort	GoogleSearchBinding
doSpellingSuggestion	GoogleSearchService	GoogleSearchPort	GoogleSearchBinding

Click **Next** to continue to the “Web Service Response” page, where the wizard asks you to indicate which part of the SOAP (Simple Object Access Protocol) response you want this input source to return.



Specify which part of the SOAP response you are interested in.

Return: [✓ The entire SOAP response] Only the SOAP part of the response The nth attachment (0-based index) 0

Select one:

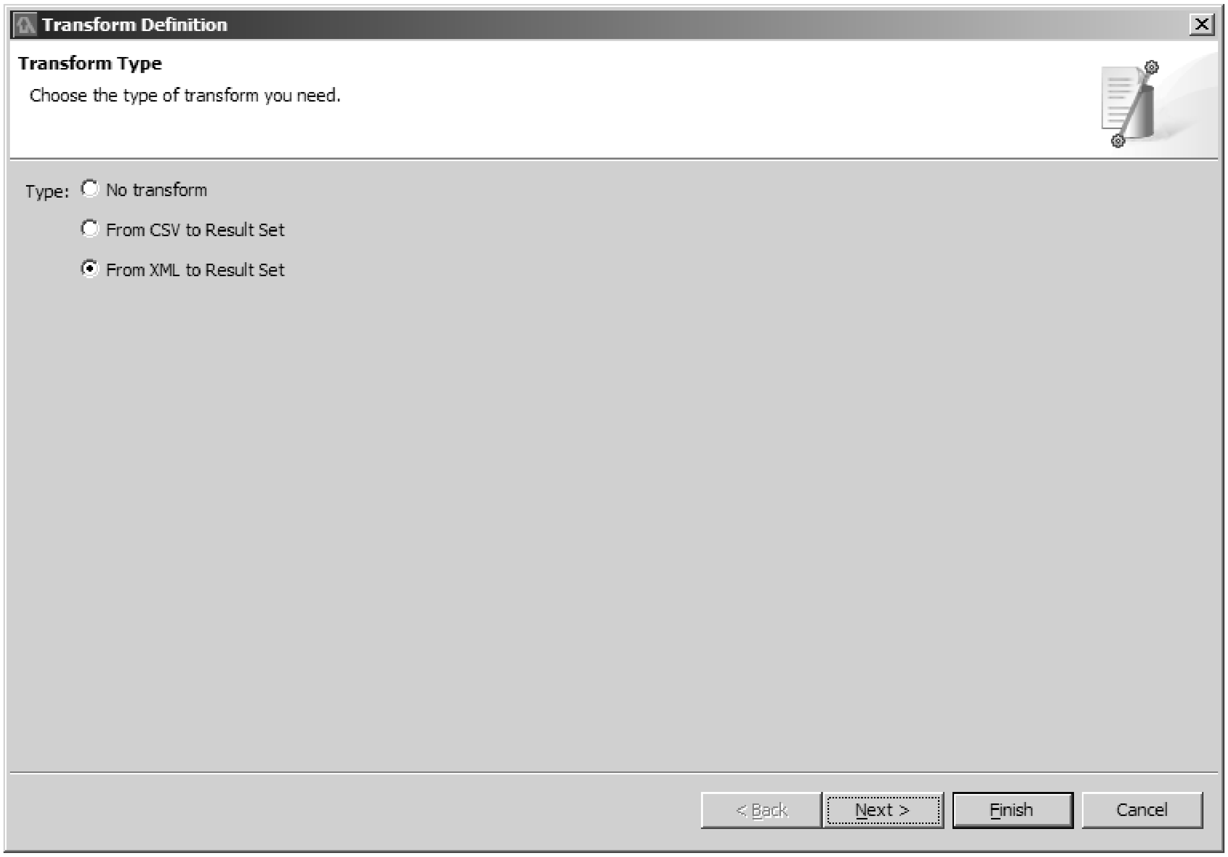
- **The entire SOAP response**
  Return the SOAP message and any attachments.
- **Only the SOAP part of the response**
  Return the SOAP message with no attachments.
- **The nth attachment (0-based index)**
  Return the specified attachment but not the SOAP message. If the web service returns multiple attachments, you can enter a number in the text box to specify the zero-based index number of the attachment to return. For example, if the web service returns four attachments and you only want the fourth, enter 3 in the text box.

Click **Finish** to close the wizard.

Configuring the transform

If the source you configured on the **Input Source** tab doesn't already contain data in the relational database result set format, you will need to configure Studio to transform the data. Studio is able to convert data from CSV (Comma-Separated Values) format (actually, comma-, pipe- ('|'), or tab-separated files) and XML (Extensible Markup Language) into result set format. Data coming from database operations, virtual database operations, and data services which are already in result set format do not need to be transformed.

To set up the transform, click the **Transform** tab in the Input Source Properties dialog. The tab displays information about the current transform setting. To change the setting, click the **Define…** button to open the Transform Chooser wizard.


Transform Definition

Transform Type

Choose the type of transform you need.

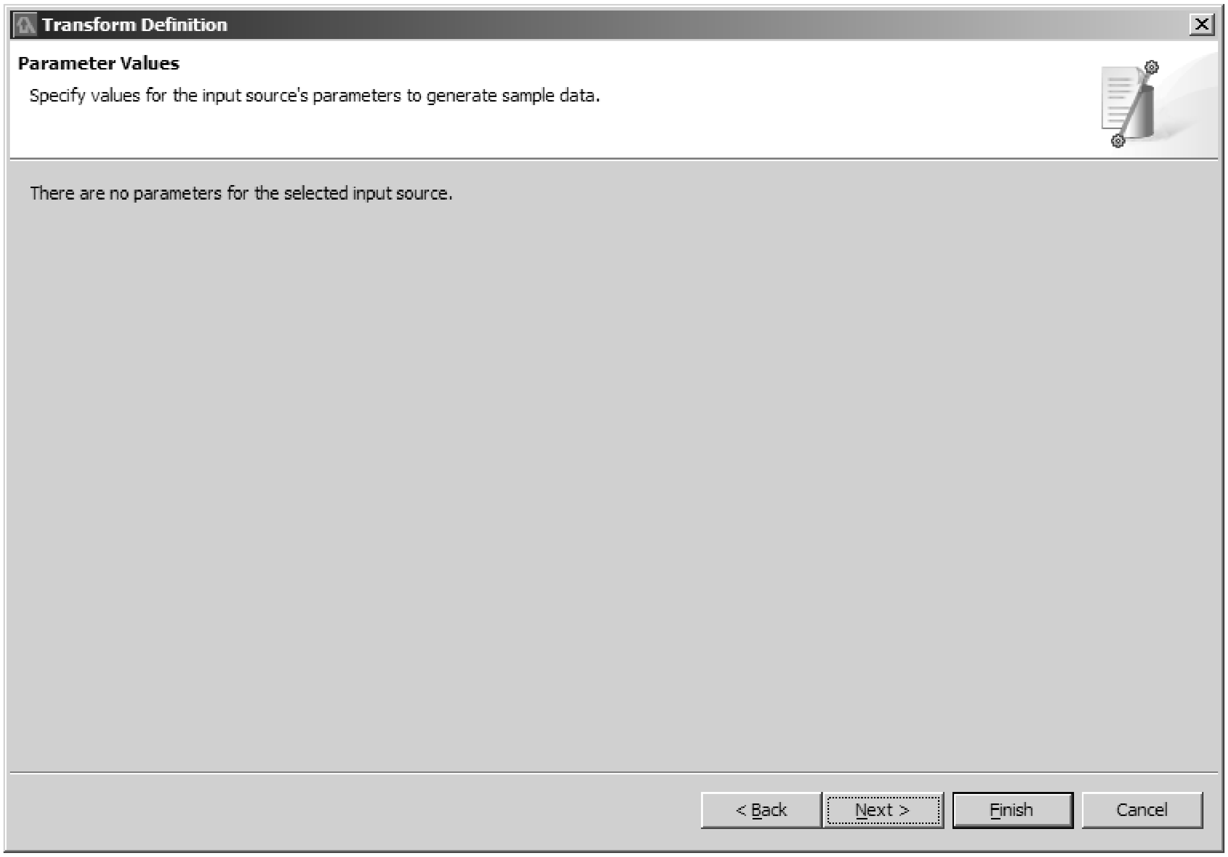
Type: ○ No transform
○ From CSV to Result Set
[✓ From XML to Result Set]

Select one:

- **No transform**
  When the input source is already in result set format and needs no transforming.

- **CSV to Result Set**
  When your data is in CSV or a similar format.
- **XML to Result Set**
  If your data is in XML format.

If you chose “No transform,” click **Finish** to close the wizard. Otherwise, click **Next** to continue. You will see the “Parameter Values” page.



Parameter Values

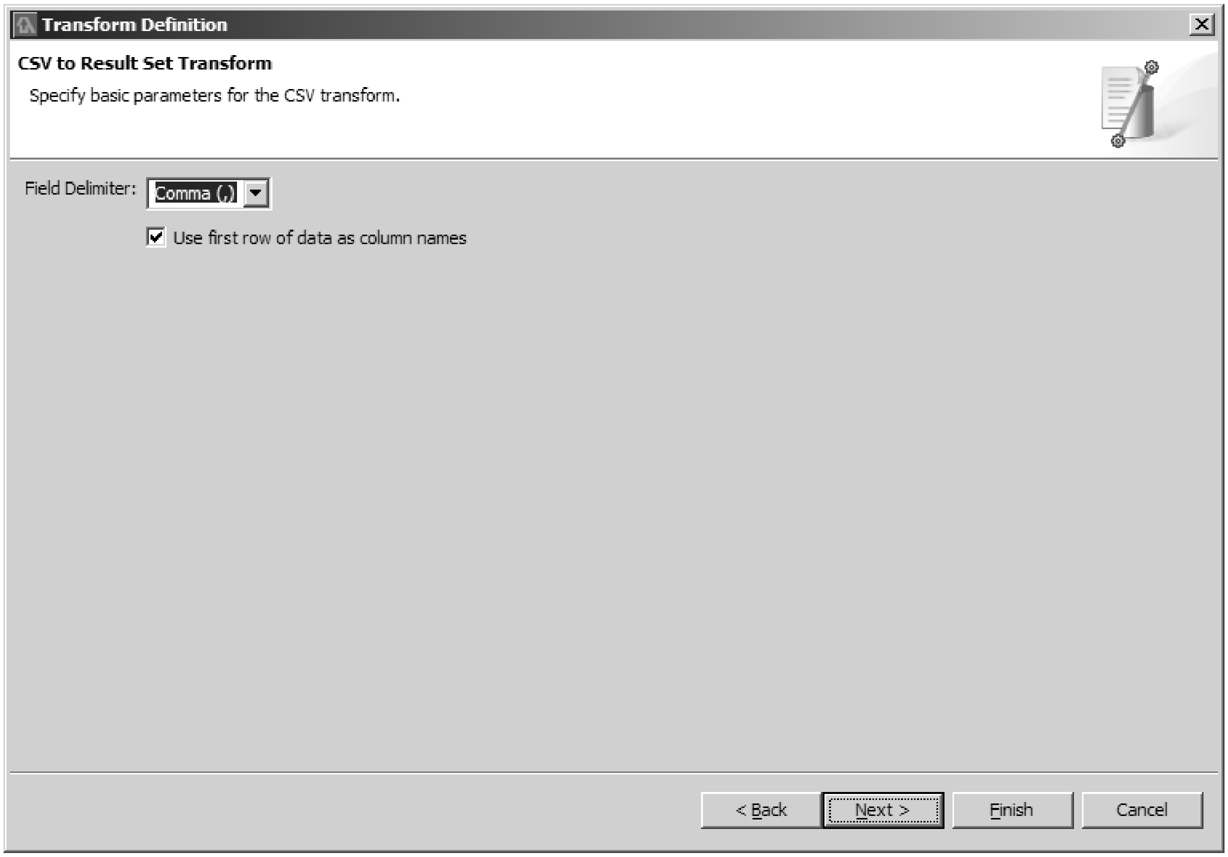
Specify values for the input source's parameters to generate sample data.

There are no parameters for the selected input source.

If your input source takes parameters, you need to enter values for them here. These are only used in the course of defining the transformation, in order to get sample data from your input source to use in the remainder of the wizard. The values you enter here are literal values, not expressions. There is no need to place quotation marks around string values, for example.

When you are done entering parameter values, or if your input source takes no parameters, click **Next**. Based on the type of transform you chose, you will see a different series of pages.

Configuring CSV transforms. If you choose to define a CSV transform, the next page will be the "CSV to Result Set Transform" page.



Transform Definition

CSV to Result Set Transform

Specify basic parameters for the CSV transform.

Field Delimiter: Comma (,)

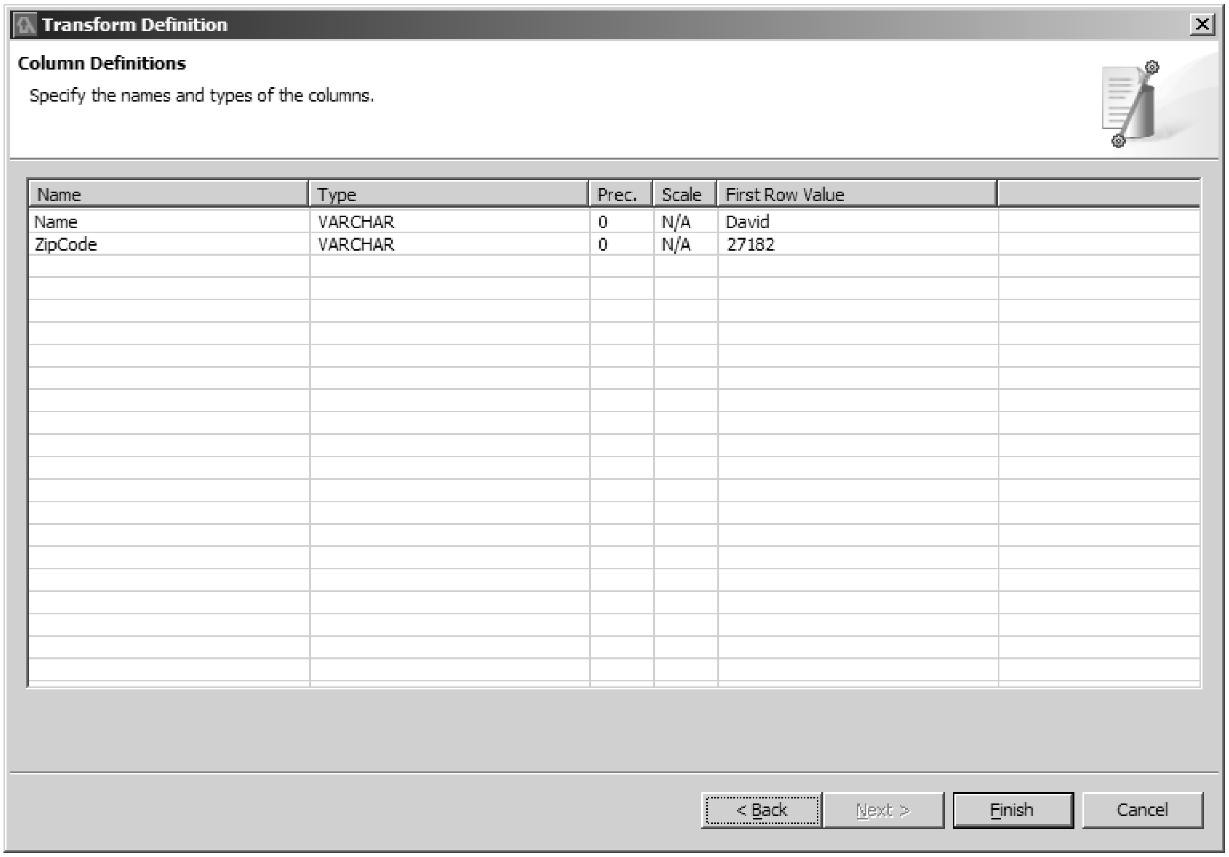
☑ Use first row of data as column names

Fill in the fields as follows:

Field Delimiter: Choose the delimiter that separates fields in the file. Studio can process files separated not only by commas (','), but also by "pipes" ('|') and tabs.

Use first row of data as column names: Check this box to treat the first row of data in the file as the names of the columns. If this checkbox is selected, then Studio will skip the first row when converting the file to a result set. In addition, on the next page in the wizard, the column names will already be populated based on the file data.

Click **Next** to display the “Column Definitions” page.



Specify the names and types of the columns.

Name	Type	Prec.	Scale	First Row Value
Name	VARCHAR	0	N/A	David
ZipCode	VARCHAR	0	N/A	27182

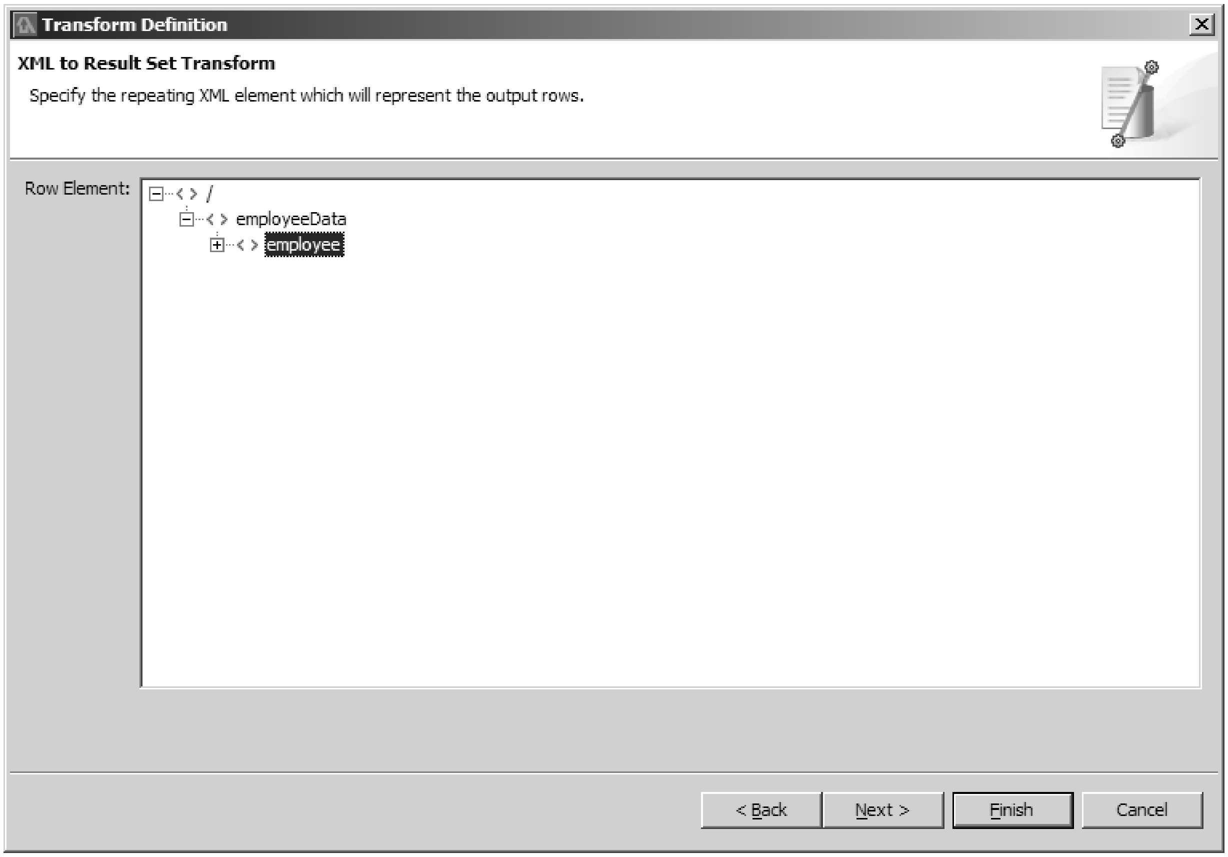
If you selected “Use first row of data as column names” on the previous page, the names of the columns will already be filled in for you. You can edit the properties of each column. (See “Working with schemas” on page 60 for information about the setting the name, type, precision, and scale of columns.)

- **Name:** Click in the Name field to edit the column’s name. Even if Studio populated the names from the first row of the file’s data, you can still edit them here.
- **Type:** Specify the type of the column’s data. When you create a new column, Studio defaults its type to VARCHAR since CSV data is by nature text-based. If you change this, make sure that the data in the document can be parsed as the type you choose (for example, if the document has the value “foo” and you choose INTEGER for the type, you will get an error when you execute your data service).
- **Precision:** Specify the precision for the column, if applicable for the column’s type.

- **Scale:** Specify the scale for the column, if applicable for the column's type.
- **First Row Value:** This column is not editable. Studio uses it to show you the values in the first row of your file, to help you decide when naming the columns or specifying their types.

When you're done, click **Finish** to close the wizard.

Configuring XML transforms. If you choose to define an XML transform, the "XML to Result Set Transform" page appears.



Transform Definition

XML to Result Set Transform

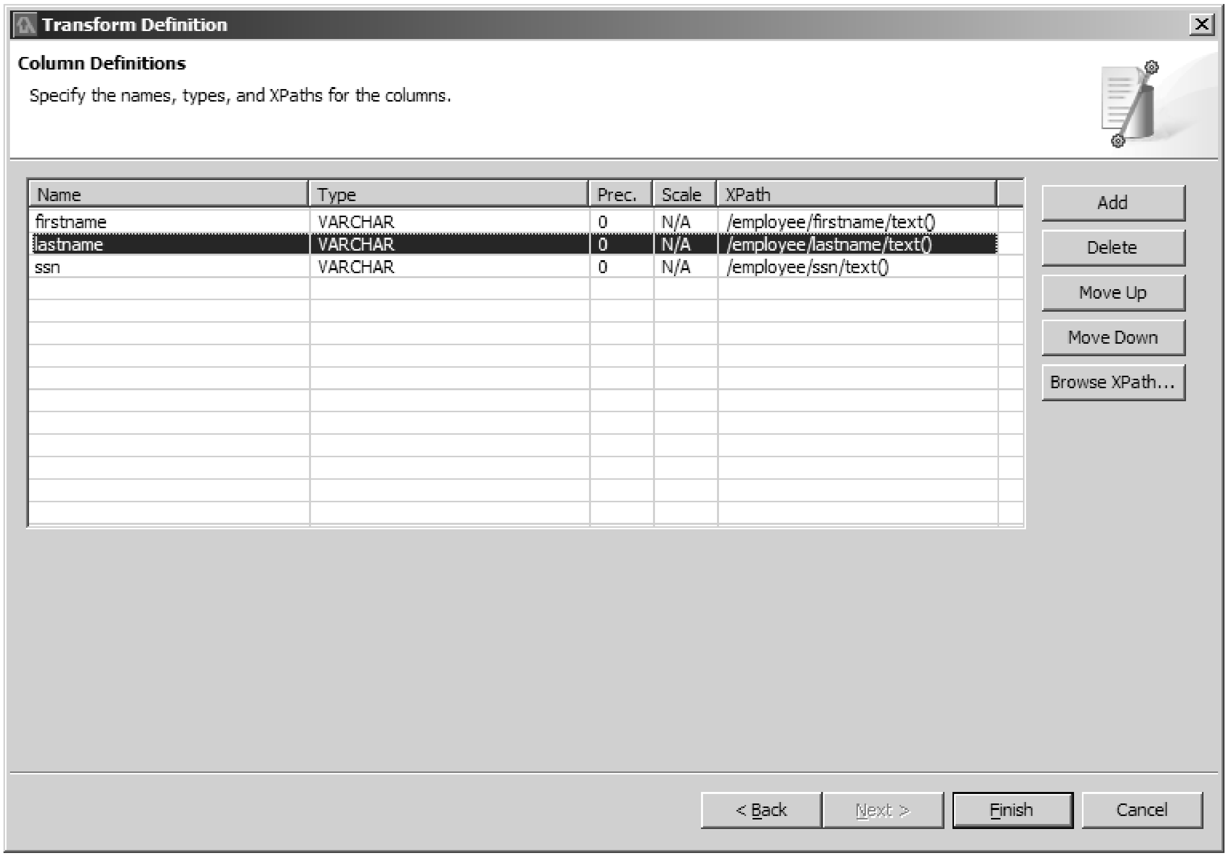
Specify the repeating XML element which will represent the output rows.

Row Element:

```
⊟··< > /
   ⊟··< > employeeData
      ⊞··< > employee
```

Row Element: Use the browser to determine which element in your XML document will be the repeating element that delimits the "rows" in your data.

When you're done, click **Next** to continue to the "Column Definitions" page.



Column Definitions

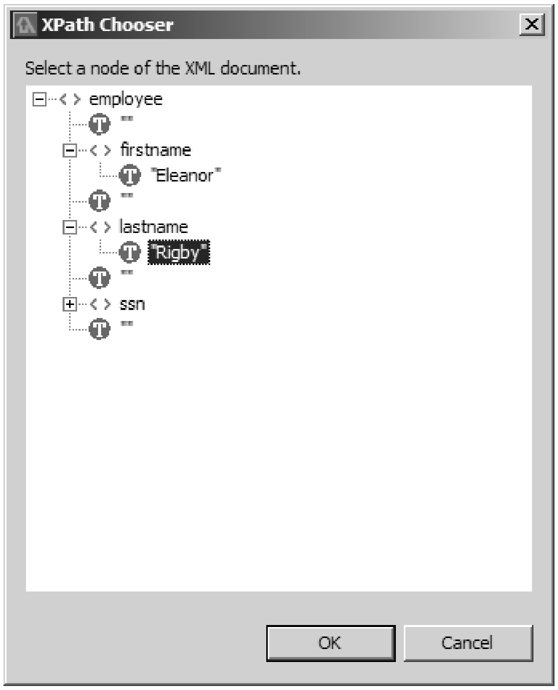
Specify the names, types, and XPaths for the columns.

Name	Type	Prec.	Scale	XPath
firstname	VARCHAR	0	N/A	/employee/firstname/text()
lastname	VARCHAR	0	N/A	/employee/lastname/text()
ssn	VARCHAR	0	N/A	/employee/ssn/text()

- Click the **Add** button to add a column.
- To delete one or more columns, select them and click **Delete**.
- To reorder the columns, select one or more of them and click the **Move Up** or **Move Down** buttons. (If you want to move multiple columns, the selection must be contiguous.)
- You can edit the properties of each column:
  - **Name:** Click in the Name field and edit the column's name. If you use the **Browse XPath…** button to choose the XML node for a given column, Studio will automatically set the column's name to the name of that node (unless you've already edited the name).
  - **Type:** Specify the type of the column's data. When you create a new column, Studio defaults its type to VARCHAR since XML data is by nature text-based. If you change this, make sure that the data in the document can be parsed as the type

you choose (for example, if the document has the value “foo” and you choose INTEGER for the type, you will get an error when you execute your data service).

- **Precision:** Specify the precision for the column, if applicable for the column’s type.
- **Scale:** Specify the scale for the column, if applicable for the column’s type.
- **XPath:** Specify an XPath expression that defines the node of the XML document whose value you want to use for this column. The XPath is relative to the repeating element you chose on the previous page of the wizard. Studio simplifies the task of choosing the XPath by providing a browser you can use to choose a node. This should be sufficient in all but the most complex cases, only requiring a knowledge of XPath syntax for highly specialized applications. To access the browser, select the column you’re defining and click **Browse XPath…** to choose the node. Make sure you choose the text node (with the Ⓣ icon) if that’s what you want. In the XPath Browser dialog, Studio shows the values of the first row’s data next to each node to help you decide what to pick.

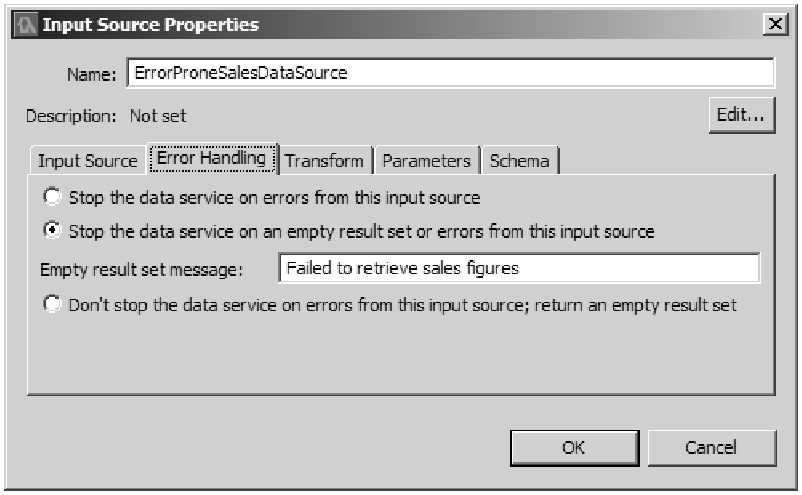


When you’re done with this page, click **Finish** to close the wizard.

Configuring error handling

You can choose how data services deployed from this view model will behave if this input source is unavailable or unresponsive. By default, a data source fails and returns an error if any input source fails to respond. If you want data services deployed from this view model to run even when this input source is unavailable, or to return an error upon receiving an empty result set, follow these steps.

Step 1 Click the **Error Handling** tab in the Input Source Properties dialog box. The dialog box displays the error handling options.



Input Source Properties

Name: ErrorProneSalesDataSource

Description: Not set

Stop the data service on errors from this input source [✓ Stop the data service on an empty result set or errors from this input source] Don't stop the data service on errors from this input source; return an empty result set

Empty result set message: Failed to retrieve sales figures

Step 2 Click one of the radio buttons:

- **Stop the data service on errors from this input source.** This is the default behavior. Choose this option if you don't want the data service to run when this input source is unavailable.
- **Stop the data service on an empty result set or errors from this input source.** Choose this option if you don't want the data service to run when it receives an empty result set from this input source, or when this input source is unavailable.
- **Don't stop the data service on errors from this input source; return an empty result set.** Choose this option if you want the data service to run with an empty result set for this input source when this input source is unavailable.

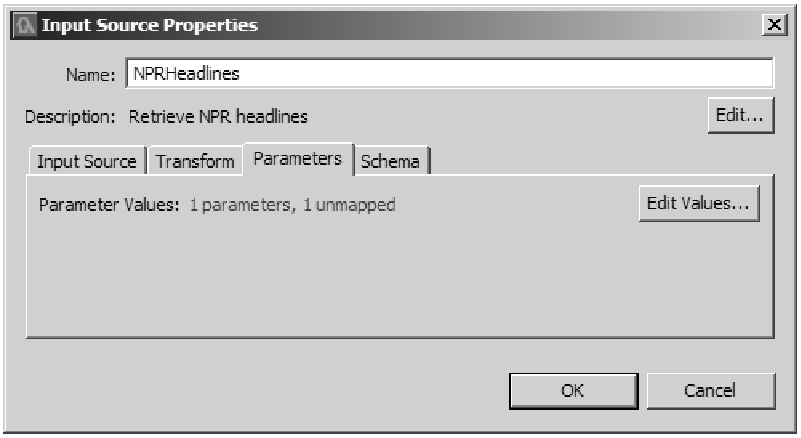
Step 3 (Optional) If you chose **Stop the data service on an empty result set...**, you can enter your own error message in the Empty result set message field. If you chose **Stop the data service on an empty result set...** and you don't enter an error message here, Studio uses a default error message that says "<data-source> returned no rows."

Step 4 Click **OK** to save your changes.

Mapping input parameters

If the input source has parameters, you must map them to global view model parameters. Follow these steps to map parameters.

Step 1 Click the **Parameters** tab to show information about the parameters the **Input Source** requires, if any. The **Parameters** tab shows you a summary, including a warning if any parameters haven't yet been mapped.

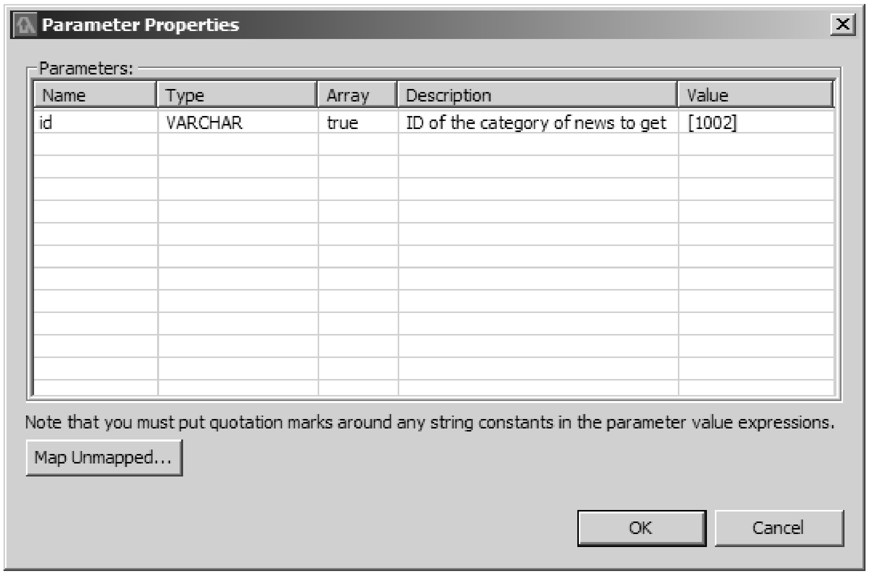


Name: NPRHeadlines

Description: Retrieve NPR headlines

Parameter Values: 1 parameters, 1 unmapped

Step 2 To edit the mappings of the element's parameters, click the **Edit Values…** button to open the Parameter Properties dialog.



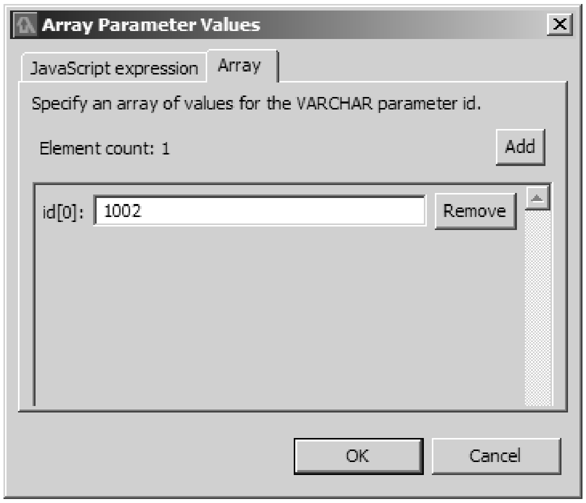
Name	Type	Array	Description	Value
id	VARCHAR	true	ID of the category of news to get	[1002]

Note that you must put quotation marks around any string constants in the parameter value expressions.

Step 3 Use an arbitrary expression to map each parameter required by an input source. This may involve global parameters to the model, variables defined downstream, such as in **Iterator** operators, constants (make sure string constants are enclosed in quotation marks), or a combination of these.

To edit the expressions, click in the Value column to make it editable. Type your mapping expression directly in the table cell, or click the "**...**" button to open the Edit Expression dialog if you need more room than the table cell provides. Right click (either in the table cell directly, or in the Edit Expression dialog) to show a context-sensitive menu of column names, variables, and functions that you can paste into the Expression field. For details on using JavaScript expressions, see "Using expressions within Avaki Studio" on page 66. For details on the context menu, see "The expressions menu" on page 71.

Arrays. If a given parameter is an array, when you click in the Value column, the Array Parameter Values dialog appears.



Specify an array of values for the VARCHAR parameter id.

Element count: 1

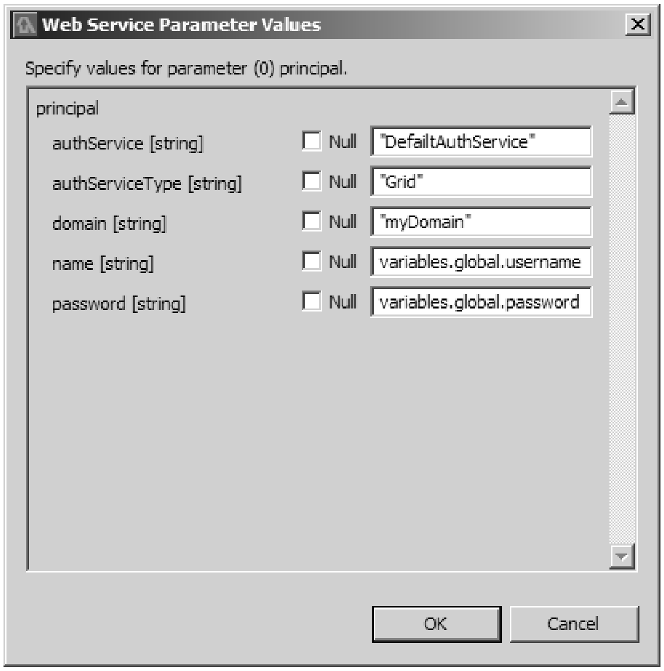
id[0]: 1002

You can enter an expression that evaluates to an array of the same type using the **JavaScript Expression** tab. For example, your model might already have a parameter that's an array of the right type, which you want to map directly to the input source parameter.

Alternatively, you can enter an array of individual expressions using the **Array** tab.

- Click the **Add** button to add a new element to the array.
- Click the **Remove** button next to any element to remove it.
- Edit the expression next to any element in the array. Remember that string constants must be surrounded by quotation marks.

Web service parameters. Parameters defined by web services are arbitrarily complex structures. When you click in the Value column for a parameter of this type (the type appears as “AROM Value”), the Web Service Parameter Values dialog appears.



Specify values for parameter (0) principal.

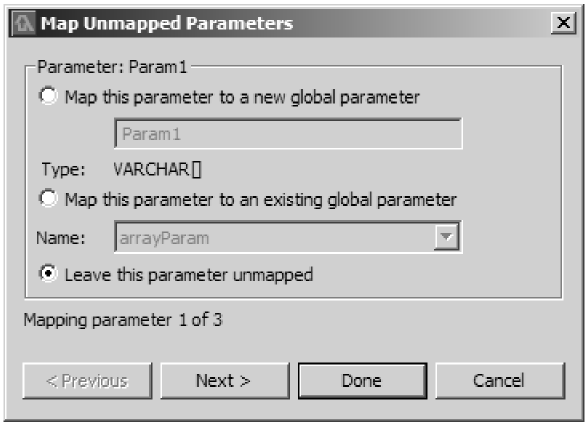
principal

authService [string]	☐ Null	"DefailtAuthService"
authServiceType [string]	☐ Null	"Grid"
domain [string]	☐ Null	"myDomain"
name [string]	☐ Null	variables.global.username
password [string]	☐ Null	variables.global.password

For many of these parameters, you simply enter expressions in the individual fields. Some more complicated structures have additional controls:

- Click the **Add** button to add a new element to an array within the complex parameter.
- Click the **Remove** button next to an array element to remove it from the array.
- If the parameter is defined so that `null` is a valid value for one of its fields, there is a Null checkbox next to it. Select this checkbox to pass the value `null` instead of typing an expression.

The Map Unmapped shortcut. Studio provides a shortcut for mapping parameters to the global parameters of your model. To access it, click the **Map Unmapped…** button in the Parameter Properties dialog. Studio displays the Map Unmapped Parameters dialog.



Map this parameter to a new global parameter

Param1

Type: VARCHAR[]

Map this parameter to an existing global parameter

Name: arrayParam

[✓ Leave this parameter unmapped]

Mapping parameter 1 of 3

This dialog walks you through the parameters in your **Input Source** that haven't yet been mapped. For each one you can:

- **Map this parameter to a new global parameter:** Choose this option to create a new global parameter to your view model and assign this **Input Source** parameter to it, all in one step. Just type the new global parameter's name in the text field provided.

- **Map this parameter to an existing global parameter:** Choose this option to choose an existing global parameter for your mapping. The pull-down list shows the global parameters that have the same type as the parameter you're mapping. Choose one from the list. This option is not available if no existing global parameters match the type of the parameter you're mapping.

- **Leave this parameter unmapped:** Do nothing with this parameter for now. Note that you will need to map it at some point, before you can deploy your model as a data service.

You can move forward and backward in the list of unmapped parameters by clicking the **Previous** and **Next** buttons. Click the **Done** button to close the dialog.

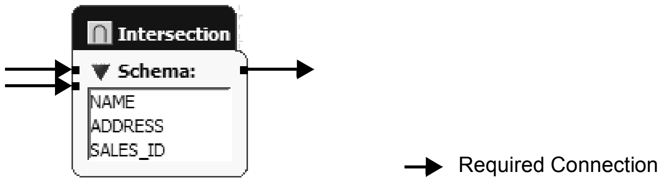
Note You can't use the Map Unmapped Parameters dialog to map parameters whose type is AROM Value. You must map these explicitly, as described in "Web service parameters" on page 146.

Intersection

The **Intersection** operation generates a result set consisting of only those rows that are common to both of its inputs. It compares all columns in each result set in computing the intersection and requires that the two input result sets have identical schemas.

Connections

The **Intersection** operator takes exactly two input result sets, which can be the output of **Input Source** elements or of any other operators. The schema of the two input result sets must be the same. It produces exactly one output result set whose schema is the same as that of the two inputs.

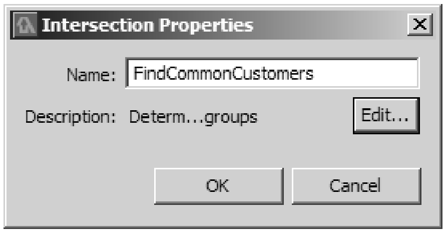


Performance notes

Computation of an intersection involves sorting the two input sources. See "Performance notes" on page 159 for details of sorting-related performance issues.

Intersection properties

Double-click an **Intersection** operator to display the Intersection Properties dialog. You can edit the name and description as described in "Properties dialogs" on page 58.


Intersection Properties

Name: FindCommonCustomers

Description: Determ...groups

There are no other properties to configure.

Iterator

The **Iterator** operator provides a looping construct for view models. Based on criteria you specify, an **Iterator** repeatedly executes the portion of the flow upstream from its position and accumulates the results. You might use an **Iterator** if, for example, you need to execute a parameterized input source once for each of a series of dates, sales regions, or customer IDs. You can use the **Iterator** operator in two ways:

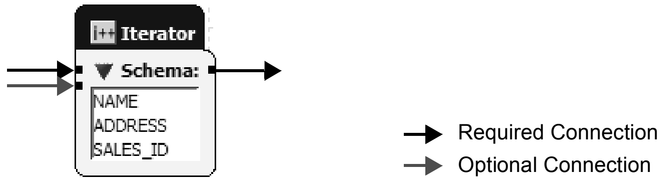
Like a `for` loop. In this approach, which is similar to `for` loops in most programming languages, the iteration is controlled by a set of expressions specifying the starting and ending conditions, as well as how to increment the loop variables. When you configure the upstream operators (those farther from the **Result**), you can use variables defined in any **Iterator** operator downstream from them. In our example, if you have an **Input Source** that takes a date parameter, you could map the parameter to a `Date` variable that the **Iterator** operator increments.

Like a `for each` loop. Studio's **Iterator** operator also gives you the option of controlling the iteration with the results of another branch of the flow, referred to as the range input. This approach is similar to the `for each` constructs in many programming languages; it allows you to execute the main branch once for each row in the range result set, which resembles a lookup table. (For example, a database operation might return a list of employee IDs, allowing you to execute the main branch of the flow once per employee.) You use expressions to define variables based on the values of each row in the range input source, and you can also use an expression to trigger the premature termination of the loop.

Note You can create nested iterations (for example, a SELECT within a SELECT) by incorporating multiple **Iterator** operators into your flow. In a pair of **Iterator** operators, the downstream one forms the outer loop and the upstream one forms the inner loop.

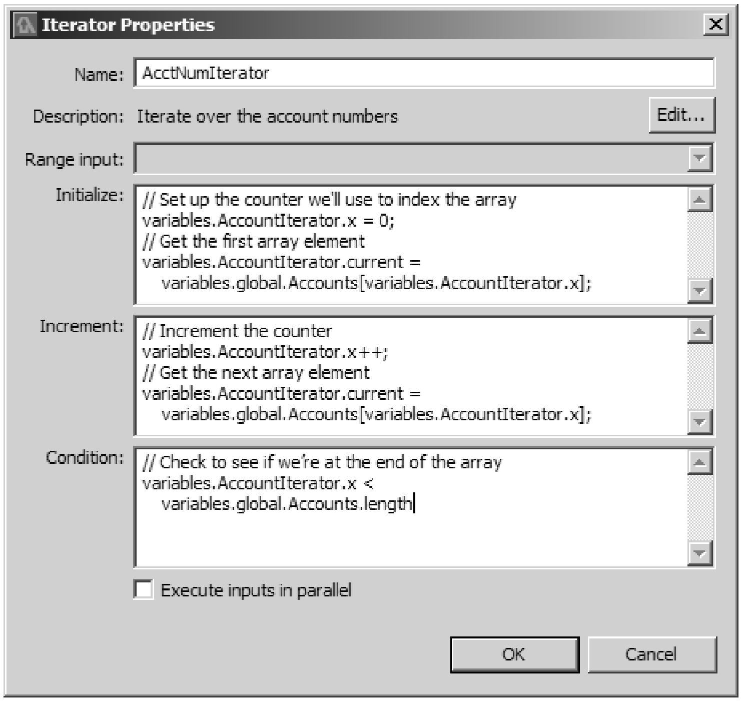
Connections

The **Iterator** operator takes either one or two input result sets. If there is a second input, it is designated the *range input* and its rows are used to perform the iteration. **Iterator** produces exactly one output result set and its schema is the same as that of the primary input.



Iterator properties

Double-click an **Iterator** operator to display the Iterator Properties dialog. You can edit the name and description as described in "Properties dialogs" on page 58.



Name: AcctNumIterator

Description: Iterate over the account numbers

Range input:

Initialize:

```
// Set up the counter we'll use to index the array
variables.AccountIterator.x = 0;
// Get the first array element
variables.AccountIterator.current =
   variables.global.Accounts[variables.AccountIterator.x];
```

Increment:

```
// Increment the counter
variables.AccountIterator.x++;
// Get the next array element
variables.AccountIterator.current =
   variables.global.Accounts[variables.AccountIterator.x];
```

Condition:

```
// Check to see if we're at the end of the array
variables.AccountIterator.x <
   variables.global.Accounts.length
```

☐ Execute inputs in parallel

Use the remaining fields to define the iteration as follows:

- **Range Input:** If you have connected two inputs to your **Iterator**, designate one of them as the range input using this pull-down list. If you're using a range input, the operator automatically loops through each row in the range result set, stopping when there are no more rows. You must set up other expressions to set the value of any variables based on the range result set's columns. However, you need not maintain your own iteration control variables (such as a counter that you increment). When you configure a range input, Studio automatically checks for the end of the result set and terminates the loop under that condition. You can add an expression in the Condition field (see below) to cause the looping to terminate while there are still rows left in the range input, but you cannot cause the iteration to continue after all rows in the range result set have been used.

- **Initialize:** Enter expressions that set up any variables you need, either to control the iteration or to be made available to upstream operators. You can enter any JavaScript code here that would be valid inside a function body. See "Using expressions within Avaki Studio" on page 66 for details on using JavaScript code within Studio.
- **Increment:** Enter expressions that update the variables you initialized in the previous field. If you're using a variable to control the iteration, increment its value here. If you're using any other variables that upstream operators need to access, update their values here as well. If you're using a range input, you can update the variables based on the values of the columns in the current row of the range input. Here, too, you can enter any JavaScript code that would be valid inside a function body. See "Using expressions within Avaki Studio" on page 66 for details on using JavaScript code within Studio.
- **Condition:** Enter a Boolean expression. (Optional.) While the Boolean expression evaluates to true, the operator continues to iterate, re-executing its primary input. When the Boolean expression evaluates to false, the iteration ceases. If you're using a range input, Studio first checks for the end of the range result set. If its rows aren't yet exhausted, Studio evaluates your condition as well, and if your expression evaluates to `false`, iteration stops. See "Boolean expressions" on page 66 for details on constructing Boolean expressions.
- **Execute inputs in parallel:** Normally, the **Iterator** operator waits to advance to the next iteration until the result set from the previous iteration has no more rows of data left. If you select this checkbox, the **Iterator** will iterate until the condition expression evaluates to `false`, queuing up the result sets and then passing them downstream as they are requested. If the elements upstream of the **Iterator** involve operations external to the current data service (such as **Input Source** elements), they will essentially all be executed simultaneously, generally resulting in improved performance.

Example: Iterating over the elements of an array parameter

The problem

Suppose you want to create a data service that summarizes account information for a list of customers. The data service must take a parameter containing the array of account numbers and must produce a result set with a row for each account that includes:

- ACCOUNT_NUMBER
- LAST_TRANSACTION_DATE
- BALANCE
- NOTES

For simplicity, assume that there is already a database operation named "GetAccountInfo" that returns the LAST_TRANSACTION_DATE, BALANCE, and NOTES values for a single account, given the account's number.

The solution

Using Avaki Studio, you create a model with a parameter named "Accounts" of type `INTEGER[]` (array of integers). Then, you use the GetAccountInfo database operation to create an input source (see "Input Source" on page 125), connecting its output to a new **Projection**, the output of which goes into an new **Iterator** and then, finally, to the **Result** element.

Configure the **Iterator** as follows:

- **Name:** Enter "AccountIterator" for the operator's name. The name must match the name used in specifying the variables in the three fields below.
- **Initialize:** Enter the following code:

```
// Set up the counter we'll use to index the array
variables.AccountIterator.x = 0;
// Get the first array element
variables.AccountIterator.current =
    variables.global.Accounts[variables.AccountIterator.x];
```

- **Increment:** Enter the following code:

```
// Increment the counter
variables.AccountIterator.x++;
// Get the next array element
variables.AccountIterator.current =
    variables.global.Accounts[variables.AccountIterator.x];
```

- **Condition:** Enter the following code:

  ```
  // Check to see if we're at the end of the array
  variables.AccountIterator.x <
      variables.global.Accounts.length
  ```

Configure the **Projection** operator next (see "Projection" on page 161 for details). Use the **Project All** button to pass the results of the input source straight through. Then add a new column, named ACCOUNT_NUMBER that references the `current` variable in the **Iterator** (you can use the expressions menu to enter it if you like). The expression should be:

```
variables.AccountIterator.current
```

Finally, configure the **Input Source** parameter mapping (see "Mapping input parameters" on page 144 for details). For the "AccountNumber" parameter, use the expression:

```
variables.AccountIterator.current
```

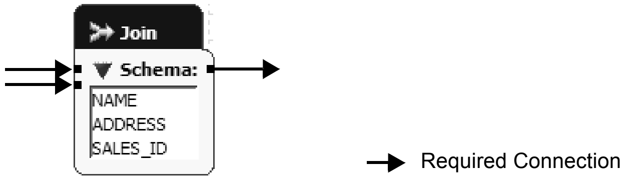
When you execute this model, the **Iterator** operator will iterate over the values in the Accounts array and execute the GetAccountInfo database operation once for each value, passing the next account number as a parameter. The **Projection** operator will then add the ACCOUNT_NUMBER column to the schema. The results of these executions will be concatenated together and returned as the overall result of the model.

Join

The **Join** operator performs a relational join operation on its two input result sets, matching up corresponding rows from the inputs. It does this matching by checking for equality of one or more pairs of columns (where column "A" of the first result set matches column "B" of the second, for example). In addition to specifying the columns to match, you specify the type of join (inner, outer left, outer right, or outer full) you want Studio to perform. You can also specify the algorithm Studio uses, or let it choose the most efficient one for you.

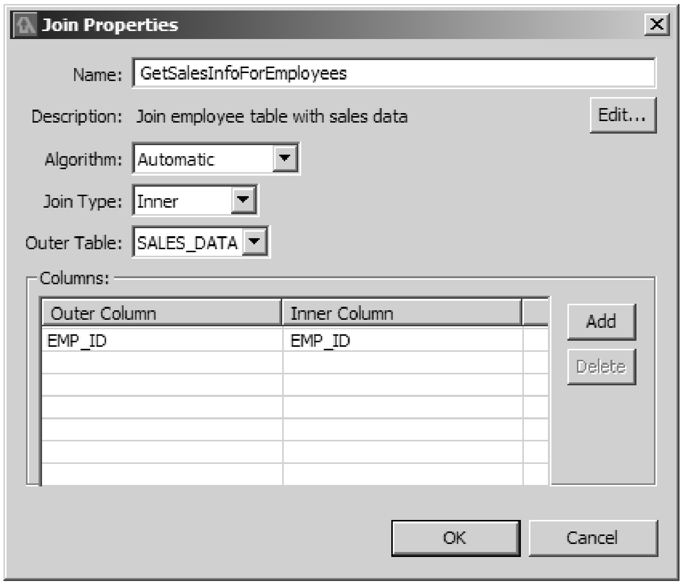
Connections

The **Join** operator takes exactly two input result sets, each of which can be the output of an **Input Source** or of any other operator. It produces exactly one output result set whose schema consists of the all of the columns from both input schemas.



Join properties

Double-click a **Join** operator to display the Join Properties dialog. You can edit the name and description as described in "Properties dialogs" on page 58.


Join Properties

Name: GetSalesInfoForEmployees

Description: Join employee table with sales data

Algorithm: Automatic

Join Type: Inner

Outer Table: SALES_DATA

Columns:

Outer Column	Inner Column
EMP_ID	EMP_ID

Inner and outer tables

The algorithm and join type choices refer to inner and outer tables. Studio doesn't require you to specify which is which when you connect your inputs to the **Join** operator. Instead, you can choose which of the two connected inputs will be the inner table by using the **Inner Table** popup.

Note The terms "inner table," "inner result set," and "inner input" are used interchangeably in the discussion of join operations. Similarly, "outer table," "outer result set," and "outer input" are equivalent.

Join algorithms

You can choose the algorithm that Studio uses to process the join operation. Generally, we recommend the "Automatic" option, which causes Studio to choose the algorithm based on the actual data with which it is presented at runtime. However, if you know specifics about the size and shape of the two inputs, you can decide to choose an algorithm yourself. The algorithms are as follows:

Algorithm	Description
Automatic	Studio automatically determines the most appropriate algorithm based on the size of the incoming result sets. It starts by trying to perform a hash join. If the data in the inner table is too large to fit in memory, Studio tries to read the outer table into memory. If the outer table fits, Studio performs a hash join with the tables reversed. If neither table fits, Studio switches to the sort-merge algorithm.
Sort Merge	This is a scalable algorithm that works well with data sets of any size. Studio breaks the data into chunks in order to sort it before performing the join.
Nested Loop	The nested loop algorithm processes each row in the inner table once for each row in the outer table. It is scalable, and uses less disk space than the sort-merge algorithm. If one of the tables being joined fits in memory, however, the hash algorithm will generally be more efficient.
Hash	Studio reads the inner table into memory and hashes it, allowing very quick lookup as it reads each row of the outer table. This is very efficient if the inner table is small enough to easily fit into memory.

Join types

Avaki Studio supports inner, outer-left, outer-right, and outer-full join types. These join types work in the same way as their SQL counterparts. They're summarized here:

Join Type	Description
Inner	Inner joins find the intersection between the two result sets. If no matches appear in the outer input for a given row in the inner input, no row is output. If a given inner input matches multiple outer input rows, then the data from the inner input is repeated for each mach, combined with each of the outer input's rows.
Outer Left	In the case of the outer left join, if a row in the inner input doesn't match anything in the outer input, a row will still be output, but with the value `NULL` for each of the columns which comes from the outer input.
Outer Right	As you might expect, this is the opposite of the outer left join. In the case of the outer right join, if a row in the outer input doesn't match anything in the inner input, a row will still be output, but with the value `NULL` for each of the columns which comes from the inner input.
Outer Full	This is a combination of the outer left join and the outer right join types. If a row from either input result set doesn't have a matching row in the other result set, it is still included, but with `NULL` values for the missing columns.

Defining join columns

Studio needs to know which columns to match in order to join your input result sets. You tell it using the table at the bottom of the Join Properties dialog.

- Click the **Add** button to add a column pair to your join.
- To delete one or more column pairs, select them and click **Delete**.
- You can edit the specific columns in each pair (When you add a new column pair, Studio chooses the next available column from the two inputs, which may not necessarily be the columns you want to join on). Note that if you have already defined columns and you change which input result set is the inner table, Studio swaps your column choices to match.
  - **Inner Column:** Click in field and use the pull-down list to choose a column from the inner table.

- **Outer Column:** Click in field and use the pull-down list to choose the corresponding column from the outer table.

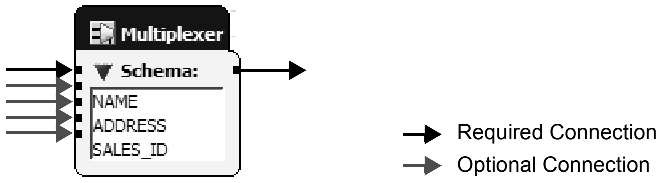
Multiplexer

Given up to five input result sets and a conditional expression for each, the **Multiplexer** operator generates a result set that concatenates all the inputs whose conditional expressions evaluate to `true`. You can use **Multiplexer** to select one or more inputs from among several choices. Think of it as a graphical means of implementing programming functionality afforded by constructs such as `if/then/else` statements and `case` statements.

Use a **Multiplexer** if, for example, you want to treat recent and older data differently—suppose you want to perform one type of processing on data that's less than one month old, and a second type on any data that's older than that. Within the two conditional expressions, you would check the value of the model's `startingDate` parameter relative to today—if it is older than a month, one input path would be executed, if not, the other would.

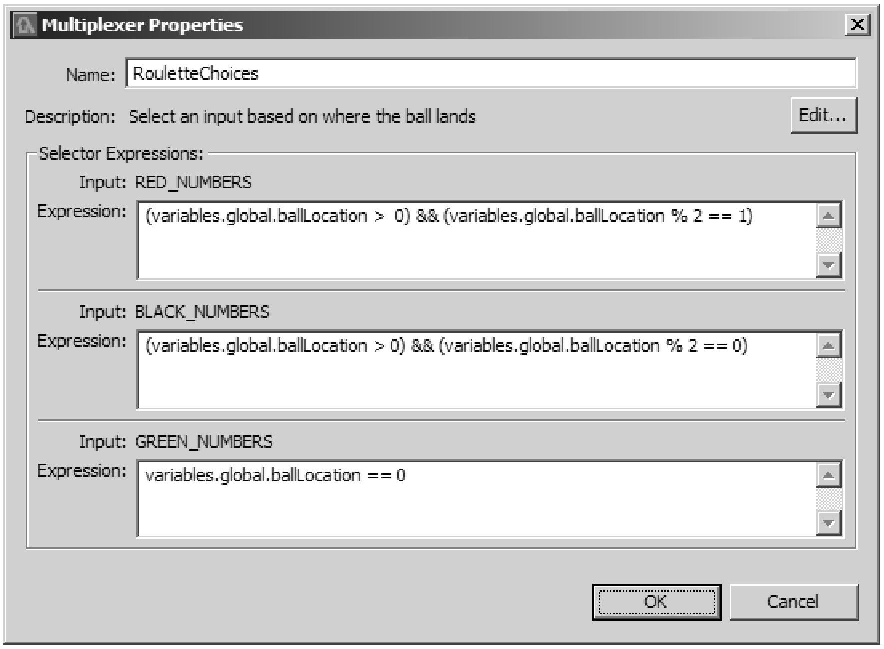
Connections

The **Multiplexer** operator takes between one and five input result sets, each of which can be the output of an **Input Source** or of any other operator. The schemas of the input result sets must be the same. **Multiplexer** produces exactly one output result set whose schema is the same as that of the inputs.



Multiplexer properties

Double-click a **Multiplexer** operator to display the Multiplexer Properties dialog. You can edit the name and description as described in "Properties dialogs" on page 58.



Name: RouletteChoices

Description: Select an input based on where the ball lands

Selector Expressions:

Input: RED_NUMBERS
Expression: (variables.global.ballLocation > 0) && (variables.global.ballLocation % 2 == 1)

Input: BLACK_NUMBERS
Expression: (variables.global.ballLocation > 0) && (variables.global.ballLocation % 2 == 0)

Input: GREEN_NUMBERS
Expression: variables.global.ballLocation == 0

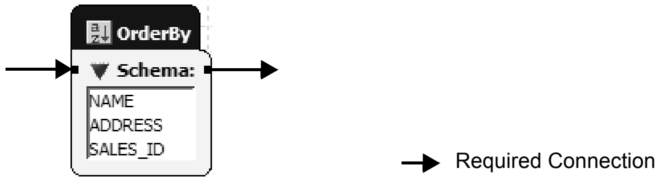
There will be one Expression field for each of the inputs you have connected to the **Multiplexer** operator, each labeled with the name of the corresponding input. In each Expression field, enter a Boolean expression which will determine whether the corresponding input will be included in the operator's output. For information on using Boolean expressions within Studio, refer to "Boolean expressions" on page 66.

Order By

The **Order By** operator sorts its input based on the criteria you specify. **Order By** does not affect the result set's schema or the number of rows it contains.

Connections

The **Order By** operator takes exactly one input result set, which can be the output of an **Input Source** or of any other operator. It produces exactly one output result set whose schema is the same as the input.



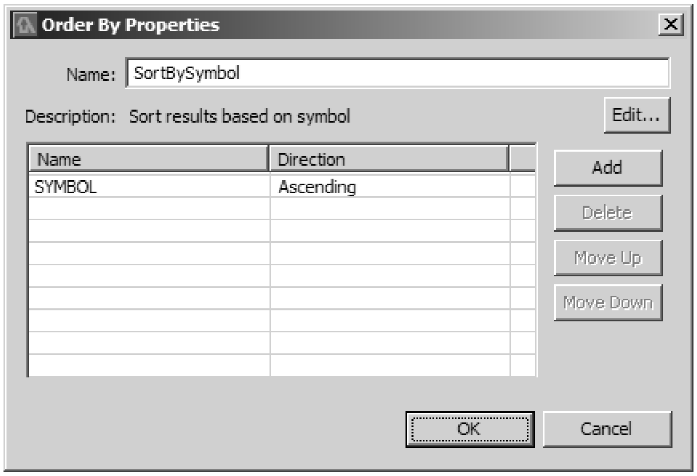
Performance notes

Avaki Studio uses an $O(n \log n)$ algorithm for performing all its sorting operations. This algorithm writes temporary data to disk if the result set to be sorted is too large to fit into memory; thus it scales to accommodate arbitrarily large result sets.

When operators in your model sort their data, Studio remembers which columns the data was sorted by and passes that information along. This way, operators that need sorted data don't have to re-sort data unnecessarily. This can dramatically increase the performance of Studio-generated data services.

Order By properties

Double-click an **Order By** operator to display the Order By Properties dialog. You can edit the name and description as described in "Properties dialogs" on page 58.



Name: SortBySymbol

Description: Sort results based on symbol

Name	Direction
SYMBOL	Ascending

Specifying the sort criteria

In the Order By Properties dialog, you list the columns from the input that the operator will use to sort the incoming rows. If you choose to sort by multiple columns, Studio sorts by the first column first; then, within the rows where that column's value is the same, it sorts by the second column, and so on. Each column can be sorted in either ascending or descending order.

- Click the **Add** button to add a column criterion to your sorting.
- To delete one or more criteria, select them and click **Delete**.
- To reorder the criteria, select one or more of them and click the **Move Up** or **Move Down** buttons. (If you want to move multiple sort criteria, the selection must be contiguous.)
- You can edit the properties of each criterion.
  - **Name:** Click in the Name field and choose a column name from the pull down list to change it. (When you add a new column, Studio chooses the next available column from the input, which may not necessarily be the column you want to sort on).
  - **Direction:** Choose the direction in which this column should be sorted—either "Ascending" or "Descending."

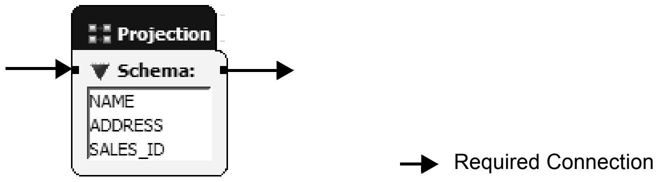
Projection

The **Projection** operator lets you modify a schema. You can pass certain columns through, unaltered, you can remove columns completely, and you can create new columns using arbitrary JavaScript expressions.

By default, **Projection** produces one row of output for every row of input. However, you have the option of omitting duplicate rows from the output result set.

Connections

The **Projection** operator takes exactly one input result set, which can be the output of an **Input Source** or of any other operator. It produces exactly one output result set whose schema is defined by the operator.

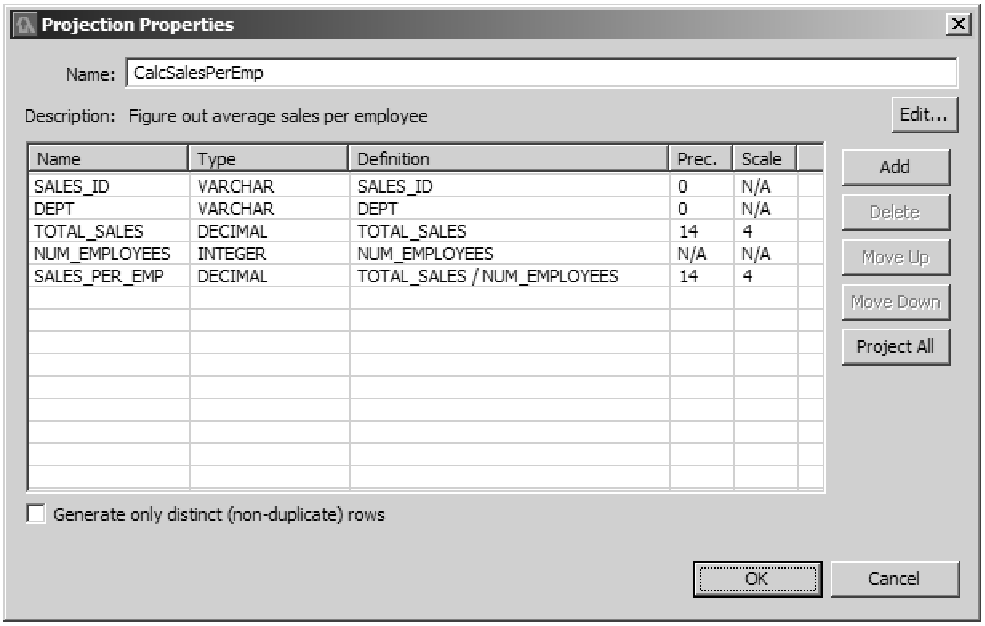


Performance notes

In order to perform the "distinct" operation, Studio must sort the results of your projection, therefore similar performance issues apply as for the Order By operator. See "Performance notes" on page 159 for details.

Projection properties

Double-click a **Projection** operator to display the Projection Properties dialog. You can edit the name and description as described in "Properties dialogs" on page 58.



Name: CalcSalesPerEmp

Description: Figure out average sales per employee

Name	Type	Definition	Prec.	Scale
SALES_ID	VARCHAR	SALES_ID	0	N/A
DEPT	VARCHAR	DEPT	0	N/A
TOTAL_SALES	DECIMAL	TOTAL_SALES	14	4
NUM_EMPLOYEES	INTEGER	NUM_EMPLOYEES	N/A	N/A
SALES_PER_EMP	DECIMAL	TOTAL_SALES / NUM_EMPLOYEES	14	4

☐ Generate only distinct (non-duplicate) rows

Defining projection columns

You can do the following to edit the column definitions of your projection.

- Click the **Project All** button to add column definitions for all of the columns in the input result set. This makes it easy to create your new set of columns if you only want to tweak the existing schema. If you want to project all columns, do so before defining any other columns, since doing so will replace all of your existing definitions.
- Click the **Add** button to add a column definition to your projection.
- To delete one or more columns, select them and click **Delete**.
- To reorder the columns, select one or more of them and click the **Move Up** or **Move Down** buttons. (If you want to move multiple columns, the selection must be contiguous.)
- You can edit the properties of each column (see "Working with schemas" on page 60 for information about setting the name, type, precision, and scale of columns):
  - **Name:** Click in a column's Name field to rename it.

- **Expression:** Click in the column's Expression field to edit it. This can be any JavaScript expression, including simple column names from the incoming result set. Click the "**…**" button to open the Edit Expression dialog if you need more room than the table cell provides. You can also right click (either in the table cell directly, or in the Edit Expression dialog) to show a context-sensitive menu of column names, variables, and functions that you can paste into the Expression field. For details on using JavaScript expressions, see "Using expressions within Avaki Studio" on page 66. For details on the context menu, see "The expressions menu" on page 71.
- **Type:** Specify the type of the column. Be sure to specify a type that will match the result of evaluating the column's expression.
- **Precision:** Specify the precision for the column, if applicable for the column's type.
- **Scale:** Specify the scale for the column, if applicable for the column's type.

Removing duplicates

Select the **Generate only distinct (non-duplicate) rows** checkbox to eliminate from the output result set any rows whose data duplicates that of a row already in the output.

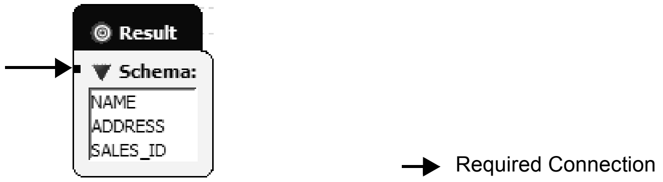
Result

A view model's **Result** element is created automatically when you create the view model. Because every view model has one and only one result, **Result** elements cannot be deleted or created separately from the view model itself.

A **Result** element takes its schema from the element connected to it. The schema of the **Result** element is also the schema of the model as a whole, and therefore, of any data services deployed from the model.

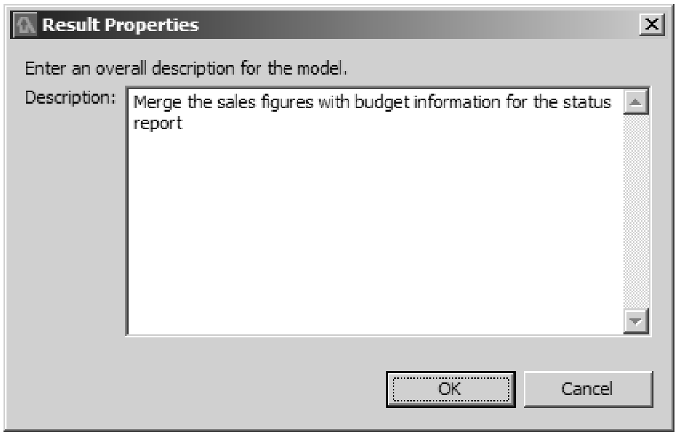
Connections

The **Result** element takes exactly one input result set, which can be the output of an **Input Source** or of any other operator. It has no outputs.



Result properties

Double-click a **Result** element to display the Result Properties dialog. Since you cannot rename the result, or affect its schema, you can only specify a description. This description is the one given to new data services that are deployed from the model.



Enter an overall description for the model.

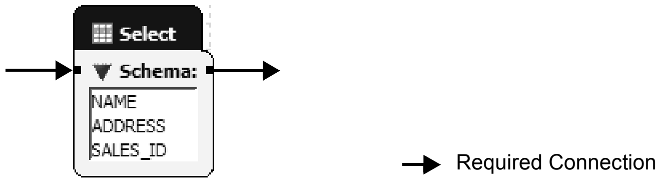
Description: Merge the sales figures with budget information for the status report

Select

Use a **Select** operator to filter rows from an input result set. You provide a Boolean expression that is tested for each row. If the expression evaluates to true for a given row, the row is passed to the output. Otherwise it is thrown away. **Select** does not change the schema of its input result set.

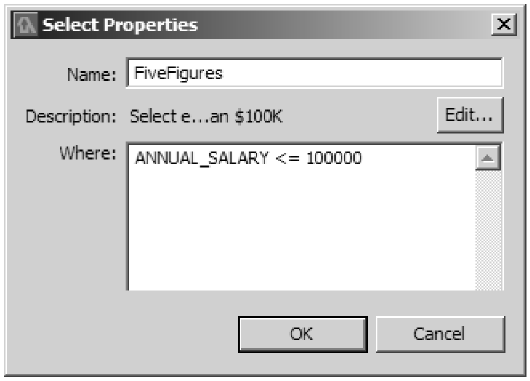
Connections

The **Select** operator takes exactly one input result set, which can be the output of an **Input Source** or of any other operator. It produces exactly one output result set whose schema is the same as that of the input.



Select properties

Double-click a **Select** operator to display the Select Properties dialog. You can edit the name and description as described in "Properties dialogs" on page 58.



Name: FiveFigures
Description: Select e...an $100K
Where: ANNUAL_SALARY <= 100000

Where: Define your select by supplying a Boolean expression for this field. When this expression evaluates to `true` for a given row, that row will be included in the output; rows for which this expression evaluates to `false` will not be included. See "Boolean expressions" on page 66 for information on using Boolean expressions in Studio. For example, if you want to list only those employees in your data whose salary (the

ANNUAL_SALARY column in the input result set) is $100,000 or less, you could use the expression below:

```
ANNUAL_SALARY <= 100000
```

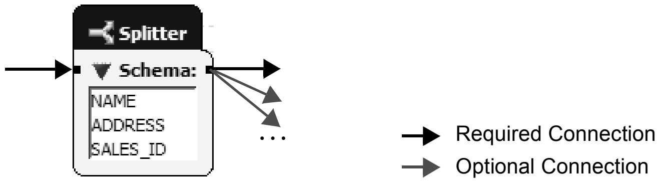
Splitter

Given one input result set, the **Splitter** operator produces one or more identical output result sets. Use **Splitter** if you need to operate on the same data in multiple execution paths within your view model.

You can configure the **Splitter** to cache the data. If you enable caching, the operator caches its input the first time you call it. On subsequent calls, the **Splitter** returns the cached data instead of re-executing upstream operators. If you configure an **Iterator** operator downstream of the **Splitter**, the scope might change across successive calls, and it might be necessary for the **Splitter** to re-execute its input in order to provide valid results. In such cases you can specify a list of variables for the **Splitter** to track—**Splitter** returns cached results as long as these variables remain constant; when any of the tracked variables change, the **Splitter** flushes its cache and re-executes its input.

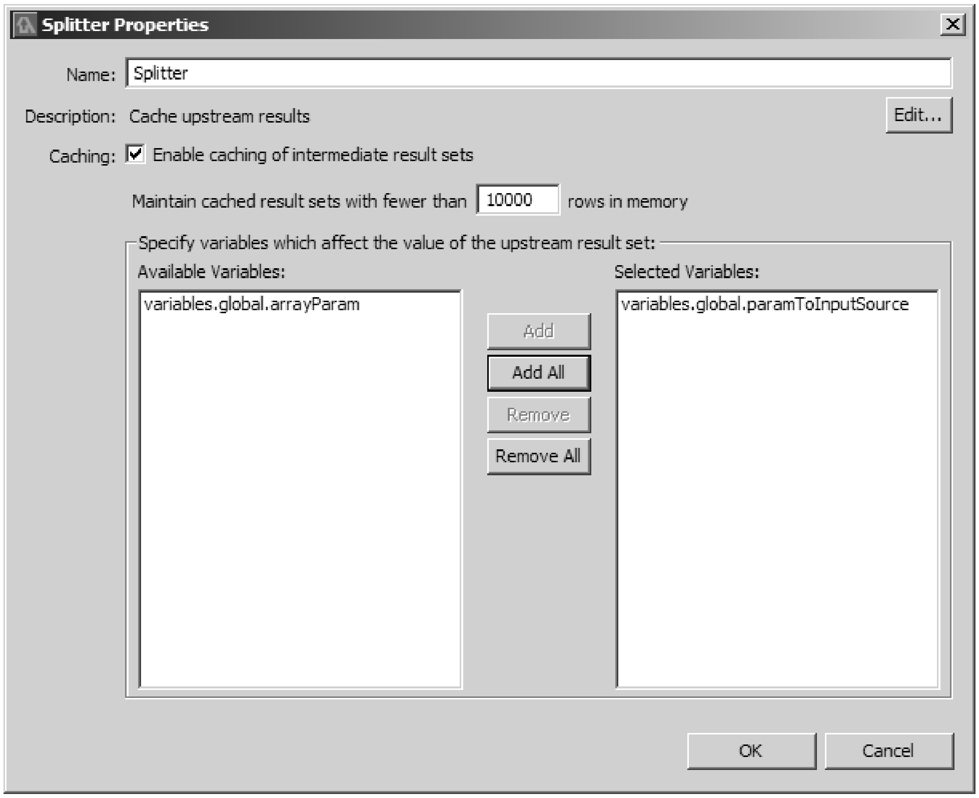
Connections

The **Splitter** operator takes exactly one input result set, which can be the output of an **Input Source** or of any other operator. It produces one or more output result sets whose schema is the same as the input.



Splitter properties

Double-click a **Splitter** operator to display the Splitter Properties dialog. You can edit the name and description as described in "Properties dialogs" on page 58.



Splitter Properties

Name: Splitter

Description: Cache upstream results

Caching: [✓] Enable caching of intermediate result sets

Maintain cached result sets with fewer than 10000 rows in memory

Specify variables which affect the value of the upstream result set:

Available Variables: variables.global.arrayParam

Selected Variables: variables.global.paramToInputSource

Caching result sets

If you want your **Splitter** operator to cache its result set, check the **Enable caching of intermediate result sets** checkbox. If you check this, you can configure the other fields in the dialog.

Maintain cached result sets…: In this field, you can specify the threshold below which Studio will maintain the results in memory rather than writing them to disk. Note that this is the number of rows of data, rather than the size of the data in bytes. If you set this value too high, the **Splitter** uses too much memory. If you set this value too low, the **Splitter** writes to disk (and thus runs more slowly) when it could be avoided.

Specify variables which affect…: In this section, you can designate any of the variables defined in your model by operators downstream of the **Splitter** as affecting the value of the cached results. For example, if you have an **Iterator** downstream that defines a variable that's mapped to a parameter in an **Input Source** element upstream of the **Splitter**, make sure that it's selected here, to ensure the validity of the cached data. The list on the left contains the downstream variables that don't affect the caching; the list on the right contains those that do. To manipulate the lists:

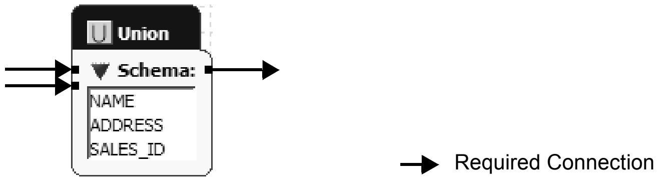
- To move one or more selected variables to the right-hand list, click **Add**.
- To move all of the variables to the right-hand list, click **Add All**.
- To move one or more selected variables back to the left-hand list, click **Remove**.
- To move all of the variables back to the left-hand list, click **Remove All**.

Union

The **Union** operator produces a single result set that is the union of its two input result sets—that is, the result is a combination of everything in both inputs. The two input result sets must have the same schema. **Union** also provides the option of ensuring that the result set contains no duplicate rows.

Connections

The **Union** operator takes exactly two input result sets, which can be the output of an **Input Source** or of any other operator. The schema of the input result sets must be the same. **Union** produces exactly one output result set whose schema is the same as that of the inputs.

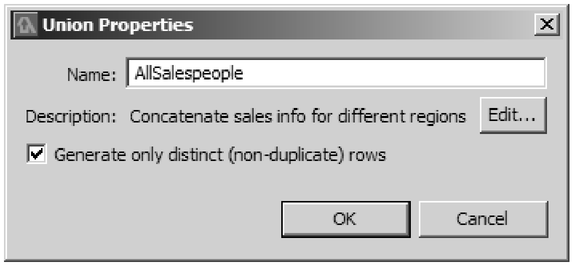


Performance notes

In order to perform the "distinct" operation, Studio must sort the results of your union; therefore similar performance issues apply as for the Order By operator. See "Performance notes" on page 159 for details.

Union properties

Double-click a **Union** operator to display the Union Properties dialog. You can edit the name and description as described in "Properties dialogs" on page 58.



Name: AllSalespeople

Description: Concatenate sales info for different regions

☑ Generate only distinct (non-duplicate) rows

Removing duplicates

Select the **Generate only distinct (non-duplicate) rows** checkbox to eliminate from the output result set any rows whose data duplicates that of a row already in the output.

Update

Use the **Update** operator to invoke an Avaki database operation that performs SQL insert or update operations against a connected database. The database operation must have the "Modifies Database" option set in order for it to work with an **Update** operation.

There are two ways to use an **Update** operator:

Mapped Parameter Mode. You can use an Update operator to perform single-row updates based on parameter values. In this mode, it behaves analogously to an **Input Source**, in that it has no input result set—it just takes the parameter values/expressions that you specify and passes them on to the database operation, returning a result set consisting of one row and one column (UPDATE_COUNT).

Batch Mode. Alternatively, you can route any flow whose result set matches the parameters of the database operation as an input to the **Update** operator. In order to use this mode,

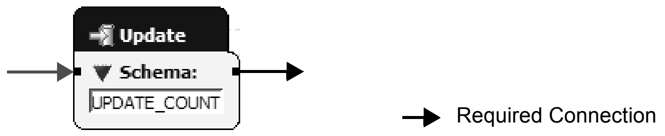
- You must have connected an input to the **Update** operator
- The input to the **Update** operator must have the same number of columns as the database operation has parameters

- The data types of the result set columns (by ordinal position) and the parameters of the database operation (by ordinal position) must match.
- The database operation must be defined with the "Supports batch operations" option enabled.

In batch mode, the output of the **Update** operator can be either the input result set (passed through with no changes) or, like in mapped parameter mode, a single-row result set consisting of the UPDATE_COUNT column.

Connections

The **Update** operator takes zero or one input result sets, which can be the output of an **Input Source** or of any other operator. It produces exactly one output result set. The schema of the output result set is either the same as the input or a single column named UPDATE_COUNT, of type INTEGER. See "Specifying the output" on page 173 for details.

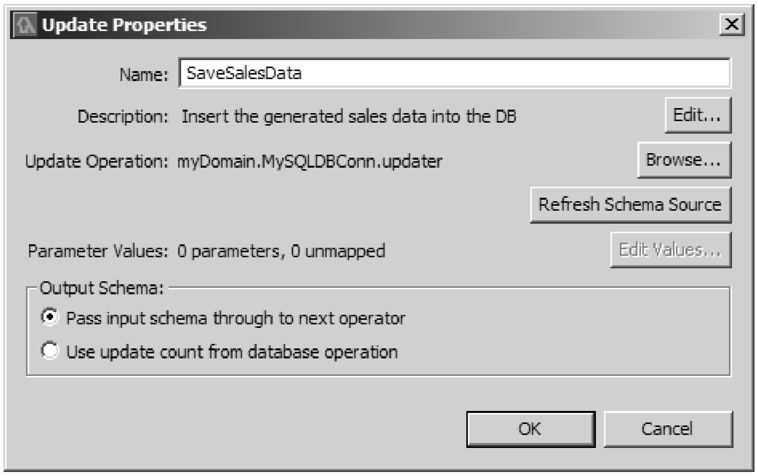


Performance notes

An **Update** operator produces no output until the update operation is done, regardless of the mode.

Update properties

Double-click an **Update** operator to display the Update Properties dialog. You can edit the name and description as described in "Properties dialogs" on page 58.


Update Properties

Name: SaveSalesData

Description: Insert the generated sales data into the DB

Update Operation: myDomain.MySQLDBConn.updater

Parameter Values: 0 parameters, 0 unmapped

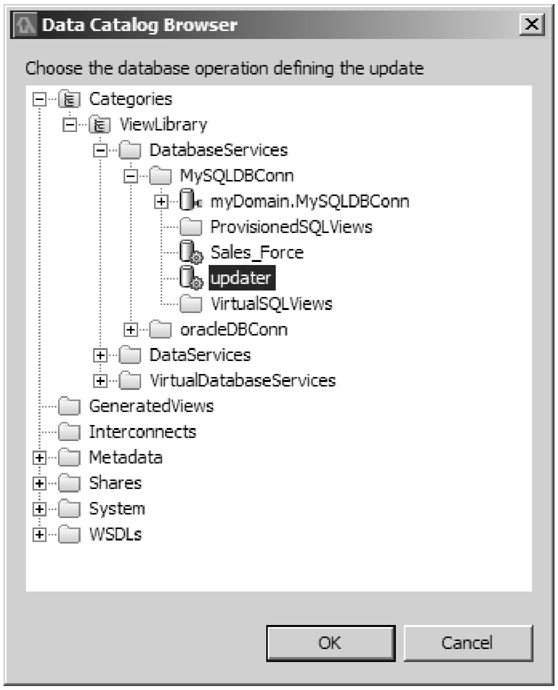
Output Schema:

(●) Pass input schema through to next operator

( ) Use update count from database operation

Choosing a database operation

The **Update** operator executes a database operation defined in your Avaki domain. Use the **Browse…** button to choose the database operation you want to execute. The operation you choose must have been defined with the "Modifies Database" option set to "Yes" in order for it to work in an **Update** operator. (If you used Studio to create the database operation, the option is a checkbox labelled "Operation modifies the database".)

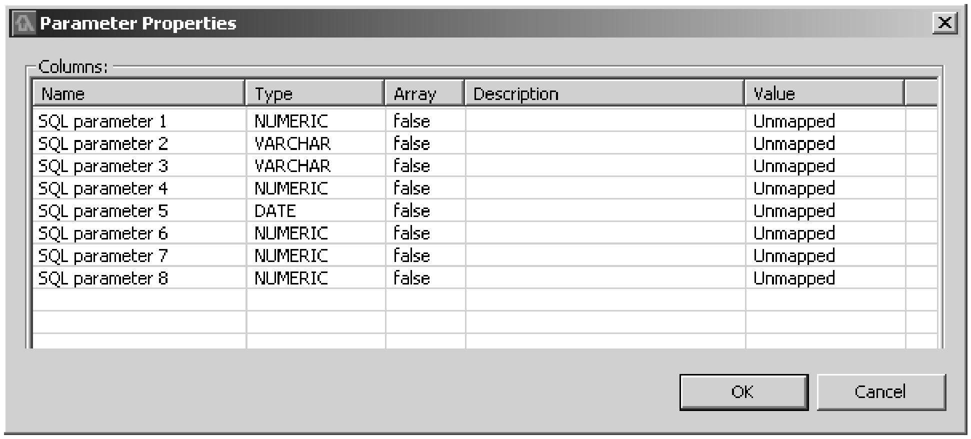


If you have already chosen a database operation, but you have changed its schema or parameters, you can click the **Refresh Schema Source** button to refresh your **Update** operator.

Mapping update parameters

If you're using your **Update** operator in mapped parameter mode, you must map its parameters. (If you are using batch mode, you don't need to explicitly map parameters, but the input schema must match the parameter types as described in "Batch Mode" on page 169.)You configure parameter mappings for an **Update** operator similarly to the way you do for an **Input Source** element. See "Mapping input parameters" on page 144 for details on mapping **Input Source** parameters.

To bring up the Parameter Properties dialog, click the **Edit Values…** button.



Name	Type	Array	Description	Value
SQL parameter 1	NUMERIC	false		Unmapped
SQL parameter 2	VARCHAR	false		Unmapped
SQL parameter 3	VARCHAR	false		Unmapped
SQL parameter 4	NUMERIC	false		Unmapped
SQL parameter 5	DATE	false		Unmapped
SQL parameter 6	NUMERIC	false		Unmapped
SQL parameter 7	NUMERIC	false		Unmapped
SQL parameter 8	NUMERIC	false		Unmapped

To map a parameter, click in the Value column to make it editable. Type your mapping expression directly in the table cell. You can also click the “**…**” button to open the Edit Expression dialog if you need more room than the table cell provides. Right click (either in the table cell directly, or in the Edit Expression dialog) to show a context-sensitive menu of column names, variables, and functions that you can paste into the Expression field. For details on using JavaScript expressions, see “Using expressions within Avaki Studio” on page 66. For details on the context menu, see “The expressions menu” on page 71.

Specifying the output

The output of the **Update** operator can be either of the following:

- **Pass input schema through to next operator:** The input result set is passed directly on to the downstream operator. This option is only applicable if an input source is connected to the **Update** operator (that is, in batch mode).
- **Use update count from database operation:** The output result set will consist of a single row containing a single INTEGER column named UPDATE_COUNT.

Glossary

Terms printed in *italics* are defined in the glossary.

access control list
(ACL) A list, for a given file, directory, or other Avaki object, of permissions—read, write, execute, delete, and owner—that control which users and groups can view, modify, invoke, and remove the object, and edit the object's ACL.

ACL
See *access control list*.

ad-hoc query
A mechanism that lets you directly query a database in SQL. The query must run through an existing Avaki *database connector*. You can run an ad-hoc query using either the CLI or a *JDBC driver*. Ad-hoc queries can be thought of as single-use *database operations*.

attribute
A property of an *Avaki directory*, file, *service*, or other object. Each attribute has a name, a type (string, integer, float, date, time, or timestamp) and a value. System attributes are read-only; you can change the values of other attributes. You can also create new attributes and add them to objects as needed.

authentication service
A *service* associated with an *Avaki domain* that authenticates an Avaki user's identity and provides security credentials each time the user logs in. Avaki can be configured to use third-party directory services as authentication services for login; for user accounts created directly in the Avaki domain, Avaki uses its own default authentication service.

Avaki directory

Avaki software creates a single, unified namespace that is accessible (subject to Avaki *access control lists*) to all users in the *Avaki domain*. The namespace, called the *data catalog*, is arranged as a hierarchy of Avaki directories (folders). The catalog directory structure is stored by the domain's grid servers and its GDC, while the physical files remain in their original locations in your local file systems. When you work with directories, it's important to distinguish between Avaki directories, which are part of the data catalog, and local directories, which reside in your local file system.

Avaki domain

The basic administrative unit of the Avaki EII system. An Avaki domain consists, at a minimum, of one *grid domain controller* and may also include one or more *grid servers*, *share servers*, *proxy servers*, *data grid access servers*, and *command clients*. See also *domain name*.

Avaki group

A set of users who have the same permissions on one or more Avaki objects. You can use the group name in place of a user name when you set permissions or create *access control lists*.

Avaki installation directory

The directory in your local file system where Avaki software is installed. This is not a *data catalog* directory.

Avaki share

(Also shared directory.) A pointer in the Avaki *data catalog* to a directory or file in the underlying local file system. When you browse the data catalog, Avaki shares look like—and can be accessed like—other Avaki directories. Contrast with *CIFS share.*

Avaki server

A *service* that starts, stops, and monitors other Avaki services on a particular computer. Every server is part of an *Avaki domain*. A server is permanently attached to the computer where it is started. There are several types of server: *data grid access servers*, *grid domain controllers*, *grid servers*, *share servers*, and *proxy servers*.

Avaki Studio

A graphical, metadata-based data integration tool that lets you

- Build data flows by dragging and dropping input sources, operators, and output targets. You can deploy your data flows as Avaki *data services*.
- Import or create *metadata models* and apply them to Avaki objects or use them to build new data services.

In addition, you can use Studio to perform provisioning tasks (creating *database connectors*, *database operations*, *virtual database operations*, and *SQL views*), manipulate *categories*, and edit *ad-hoc queries* and *attributes*.

cache service

(Formerly proxy cache service.) A staging service that stores copies of files, *database operation* results, and *data service* results. Caching improves retrieval performance. To ensure that an object is stored in the cache, you can *pin* a file or directory in the data catalog, or schedule a database operation or data service. A cache service can provide remote caching, local caching, or both. The freshness of cached data is controlled by a data expiration interval that determines how long cached data is considered valid and by a cache coherence window that tells the cache service how often to check whether cached data is still valid. If cached data is too old to satisfy a new request (or is not stored in this cache), the cache service does one of the following:

- If the database operation or data service that produced the data is local to this cache service, the cache service triggers execution of the database operation or data service.
- If the database operation or data service that produced the data is remote from this cache service, this cache service requests the data from the data source's local cache service.

A cache service can be associated with a *data grid access server*, a *grid server*, or a local user in a CLI session. See also *local cache*, *remote cache*, *on-demand caching*, and *scheduled caching*.

category

A mechanism for classifying and organizing the contents of the *data catalog*. Like *Avaki directories*, categories serve as containers for objects in the data catalog. Anything in the data catalog—views, data services, shared files, even Avaki directories themselves—can be assigned to a category. Categories are hierarchical, they have attributes, and Avaki *access control lists* regulate access to them.

CIFS client

A machine that mounts files or directories from the Avaki *data catalog* by connecting to a *CIFS share* through an Avaki *data grid access server.* A CIFS client need not have Avaki software installed. (CIFS—Common Internet File System—is a file-sharing protocol based on the file system implemented by Windows.)

CIFS share

A directory or file that has been exported (shared) from the Avaki *data catalog*. A CIFS share can be mapped into a Windows file system like a network drive. When you browse the Windows file system, CIFS shares look like—and can be accessed like—other files and directories. CIFS shares are created through a *data grid access server*. Contrast with *Avaki share*.

client

Avaki supports several types of client: *Avaki Studio*, *CIFS clients*, *command clients*, JDBC/ODBC clients, *NFS clients*, *web clients, and WS clients*.

command client

A machine that can issue Avaki commands but does not contribute resources to the *Avaki domain*.

connect port

The connect port on a *grid domain controller*, *grid server*, *data grid access server*, *proxy server*, or *share server* accesses the JNDI naming service or RMI registry for the underlying application server. The connect port is one of many ports that a GDC or server uses to communicate with other Avaki objects. You must supply the connect port number of a target grid server or GDC whenever you connect a new object (another server, a copy of Avaki Studio, or a *command client*, for example) to an *Avaki domain*. When you *interconnect* two Avaki domains, you must supply each domain's connect port number to the other one.

data catalog

A hierarchical structure similar to a file system that encompasses all objects in an *Avaki domain*. The data catalog contains *Avaki directories* and files, *Avaki shares*, *Avaki servers*, *SQL views*, *database operations* and *data services*, and other objects.

data grid access server

(DGAS) An *Avaki server* that makes *Avaki directories* and their contents available to *CIFS clients* and *NFS clients*.

data service

An operation that transforms data obtained from sources in the *data catalog*. Input data can come from any number of sources, including:

- other data services
- data catalog files (which can be *generated views*)
- Avaki *database operations* (which in turn extract the data from relational databases)
- HTTP requests
- Web service invocations

You can generate the code that manipulates the data by creating a *view model* in *Avaki Studio*, or by writing a custom *data service plug-in* using Java, JavaScript, or XSLT. Data service output can be in rowset or XML format. Data services are run by the *execution services* on *grid servers*, they can be scheduled, and their results can be cached.

data service plug-in

The logic for a *data service*, written in Java, JavaScript, or XSLT. Data service plug-ins are modular—you can use the same plug-in for multiple data services. *Avaki Studio* creates data services and plug-ins simultaneously, so if you use Avaki Studio to create data services, you don't have to worry about plug-ins. You can also use the Avaki Plug-in Wizard to create data service plug-ins.

database connector

A mechanism that enables one or more *database operations, SQL views*, or *ad-hoc queries* to connect to a relational database.

database operation

(DBOP) A mechanism that can

- extract data from a relational database and deliver it on demand to a *view generator* or a *data service*, or
- modify data in a relational database.

A database operation can be a SQL statement or a stored procedure call.

dependency

A relationship in which an Avaki object requires input from other Avaki objects. A *data service* might require input from one or more *database operations* or from other data services. A *view generator* might depend on a database operation for input. A database operation can serve as an input source for one or more data services or view generators. Generated *SQL views* depend on database operations, virtual database operations, or data services. You can use *Avaki Studio*, the web UI, or the CLI to list input and output dependencies for any data service, database operation, or view.

DGAS

See *data grid access server*.

distributed transaction

A set of related operations (typically SQL operations such as SELECT, INSERT, UPDATE, DELETE, and CALL) that

- involve one or more databases, and
- might lead to unwanted results (such as leaving participating databases in an inconsistent state or producing inconsistent reads) if some of the operations complete and others do not, and therefore
- must all be executed at once, as a single transaction.

The individual operations that make up a distributed transaction are performed by *database operations* that use *database connectors* configured with XA-capable *JDBC drivers*; all the database operations are executed, using the two-phase commit protocol, by a specially configured *data service*. The two-phase commit protocol is designed to ensure that the participating databases will be left in a consistent state—that is, that all the operations in the distributed transaction will be completed, or none of them will.

domain name

A unique alphanumeric identifier for an *Avaki domain*. The domain name is assigned by the Avaki administrator when the Avaki domain is initialized. The domain name has a maximum length of 30 characters.

enterprise information integration

(EII) A software system that

- enables applications and users to access, without replication, both raw and integrated data from multiple heterogeneous distributed data sources while hiding the complexity of the data sources, and
- provides tools enabling users and data owners to further integrate and transform data.

exclusion

See *schedule exclusion*.

execution service

Execution services execute *data services*. There is an execution service on every *grid server*, and you can configure a pool of execution services for load-sharing. When a pool is in place, a data service can be run by any execution service in its grid server's pool.

failover
The transition of control from a failing or unreachable primary *grid domain controller* to a secondary grid domain controller.

federated data access
A scheme that allows independently controlled elements to be shared into a single namespace. Files, user accounts, and other objects maintain their separate identities and remain under the control of their owners, but—subject to access controls—the objects can be accessed, managed, and viewed as if they were part of a single system.

GDC
See *grid domain controller*.

generated view
A file created by a *view generator;* it may contain data obtained from a *database operation*, a *data service*, a file, or an HTTP source. Like other files, generated views exist in a local file system and are shared into the *data catalog*.

grid
A heterogeneous group of networked resources that appears and functions as one operating environment. A data grid like the Avaki Enterprise Information Integration (EII) system provides secure, shared access to data.

grid directory
See *Avaki directory*.

grid domain
See *Avaki domain*.

grid domain controller
(GDC) The first server in an *Avaki domain* is the grid domain controller. The GDC maintains a portion of the Avaki domain's namespace and provides authentication services. It can also run Avaki commands, share data, and monitor other servers. (That is, the GDC functions as a *grid server.*) If the domain is configured for *failover*, it has both a primary GDC and a secondary GDC; the secondary is updated at regular intervals and takes over management of the domain if the primary fails. Any Avaki shares managed by the primary are read-only on the secondary.

grid server
An *Avaki server* that maintains a portion of the *Avaki domain*'s namespace, runs Avaki services such as shares, execution services, caches, and searches, and allows you to run Avaki's web UI and execute Avaki commands.

group
See *Avaki group*.

hard link
Provides an alternate name for an item in the *data catalog*. Changes to the object's other names have no effect on the hard link: you can move or change a file's original name and the hard link will still know where to find the file. To delete a hard-linked object, you must remove its original name. Contrast with *soft link*.

interconnect
To create a unidirectional link from one *Avaki domain* to another. Interconnecting lets an Avaki domain make its *data catalog* visible to users in another domain (subject to Avaki access controls).

JDBC driver
JDBC (Java Database Connectivity) drivers allows application programmers to access database data shared in the *data catalog*. When a JDBC driver accesses data, it returns a JDBC result set that's immediately available to your program. JDBC drivers can:

- Call any *data service* in the data catalog
- Call any *database operation* in the data catalog
- Perform SQL `select` operations against *SQL views* in the data catalog

Sybase offers three JDBC drivers for use with Avaki EII software:

- The included Avaki JDBC driver
- jConnect, Sybase's standard JDBC driver
- An XA-capable driver for use with *database connectors* that support *distributed transactions*

link
See *hard link* and *soft link*.

local cache
A *cache service* that runs on the same *grid server* as a *database operation* or a *data service* that generates cachable data. The local cache stores results produced by local database operations and data services so they don't have to execute for every new request. See also *remote cache*.

metadata model
A construct in *Avaki Studio* that expresses a schema by defining a set of tables. A table in a metadata model can be mapped (linked) to an Avaki object such as a *data service* or a *database operation*, or to a table in a relational database. The mapping lets you address each mapped object by the name of the corresponding table in the metadata model. You can also derive a *view model* schema from a metadata model. When you do this, you ensure that the results of any data service deployed from the view model will conform to the metadata model's schema.

NFS client
A machine that mounts the Avaki *data catalog* (or a portion of it) as a directory by connecting to an Avaki *data grid access server*. An NFS client need not have Avaki software installed. (NFS—Network File System—lets you add file systems located on a remote computer to the directory structure on your own computer.)

ODBC
ODBC (Open DataBase Connectivity) is an API for databases on Windows. An ODBC driver (such as the the Sybase Organic ODBC driver included with Sybase ASE) allows Avaki to communicate with Windows database applications.

on-demand caching
A scheme by which an object is cached only if it's used—for example, results are cached when a *database operation* or a *data service* is executed, or a file is cached when a user or application reads it. On-demand caching uses a fixed expiration interval to determine data freshness. On-demand caching is suitable for objects that are rarely accessed or that change at irregular intervals. Contrast with *scheduled caching*.

pin
To mark an *Avaki directory* or file for *scheduled caching*. See also *cache service*.

plug-in
See *data service plug-in*.

primary GDC
See *grid domain controller*.

proxy server
An *Avaki server* that allows *Avaki domains* on opposite sides of a firewall or a Network Address Translator (NAT) to communicate with one another.

queries
See *ad-hoc query*.

query engine
An Avaki *service* that executes SQL queries against the *SQL views* (tables) that make up the Avaki *virtual database*. A query engine analyzes queries, pushes as much of the work as possible down to the underlying relational database (if there is one), and performs the remaining operations (such as joins across tables from different databases) itself. There is a query engine on each *grid server.*

remote cache
A *cache service* that runs on a grid server that is remote from an Avaki service (a *database operation* or a *data service*) that generates cachable data. The remote cache stores results produced by distant services so the results don't have to be fetched over the network to satisfy every new request. Users and applications that access remote data through the cache may have access to cached copies even when the remote data source is unavailable. See also *local cache*.

secondary GDC
See *grid domain controller*.

scheduled caching
A scheme by which an object is cached according to a schedule that you create. The schedule specifies when the object is first cached and how often (or following what trigger event, such as a change to a file) the cache is refreshed. If the object is a *data service* or a *database operation*, the schedule runs it to put fresh results in the cache. Scheduled caching, which overrides other types of caching, is suitable for objects that are updated frequently or on a regular basis. Contrast with *on-demand caching*.

schedule exclusion
A named period of time during which scheduled activities can be prevented from running. You can apply an exclusion to as many schedules as you want. Scheduled activities include refreshing *Avaki shares* and imported user accounts, and caching files, directories, and the results of *database operations*, *data services*, and *generated views*.

service
An Avaki object that performs a function in the domain (stores data or authenticates users, for example). Services provided in Avaki software include *Avaki directories*, *Avaki shares*, *Avaki servers*, *authentication services*, *execution services*, and user accounts.

share
A point of connection between the Avaki *data catalog* and a native file system or file system tool. Avaki supports two kinds of shares: *Avaki shares* and *CIFS shares*.

share server
An *Avaki server* whose only task is to manage *Avaki shares*—local directories that are exported (shared) into the *data catalog*. (Grid servers can also manage shares.)

shared directory
See *Avaki share*.

soft link
A pointer to a particular location (name) in the Avaki *data catalog*. If the object at that location is moved, deleted, or renamed, the soft link leads nowhere. Soft links can be created only in the CLI. Contrast with *hard link*.

SQL view
A virtual table—a *data catalog* entry that represents a table in a relational database, a *database operation*, or a *data service*. SQL views can be created in three ways:

- Provisioned directly from a table in an underlying database
- Generated from a database operation or data service
- Mapped from a database table, a database operation, or a data service, using the *Avaki Studio* metadata model editor

Every SQL view is part of the Avaki *virtual database*. SQL views are treated as relational tables by the Avaki *query engine*. SQL view data can be accessed using standard SQL statements by connecting to Avaki with ODBC or JDBC, or via an Avaki *virtual database operation*.

update notification
A message issued when a *generated view* is updated. A view that receives data from another view can be configured to regenerate itself (using the new data) upon receipt of an update notification.

view generator
A mechanism that does one of the following: extracts data from a file or an HTTP source, obtains data from an Avaki *data service*, or uses an Avaki *database operation* to extract data from a relational database. The view generator can display the data, perform an XSLT transform, save the data as a *generated view* file, and/or update a database. Contrast with *data service*.

view model
The graphical representation of a data flow that you can build in *Avaki Studio*. A view model typically includes one or more input sources (such as *database operations* or *data services*), one or more operations to combine or transform the data, and an output target. When you deploy a view model, it becomes an Avaki data service.

virtual database
The set of all *SQL views* in an *Avaki domain*, including those provisioned from external databases and those generated from *data services* and *database operations*. You can execute SQL queries on the SQL views in the virtual database as if they were tables in a single database.

virtual database operation
A *database operation* whose source database is the Avaki *virtual database* itself. Use virtual database operations if you want to encapsulate and reuse SQL SELECT queries against *SQL views* (provisioned or generated).

web services client
See *WS client*.

WS client
(Also web services client.) A tool or a piece of code that is part of a customer application and that makes SOAP calls to web services on an Avaki grid server. The SOAP calls can request data from the Avaki *data catalog*, from a *database operation*, or from a *data service*.

Master Index

In electronic copies of this book, the index links to other books in the documentation set work only as long as the PDF files are stored in the same directory.

Key
AD: *Administration Guide*
API: *API Guide*
C: *Command Reference*
O: *Overture*
P: *Provisioning & Advanced Data Integration Guide*
S: *Data Integration with Avaki Studio*