

ActuateOne™

One Design
One Server
One User Experience

Using Information Object Query Builder

Information in this document is subject to change without notice. Examples provided are fictitious. No part of this document may be reproduced or transmitted in any form, or by any means, electronic or mechanical, for any purpose, in whole or in part, without the express written permission of Actuate Corporation.

© 1995 - 2011 by Actuate Corporation. All rights reserved. Printed in the United States of America.

Contains information proprietary to:

Actuate Corporation, 2207 Bridgepointe Parkway, San Mateo, CA 94404

www.actuate.com

www.birt-exchange.com

The software described in this manual is provided by Actuate Corporation under an Actuate License agreement. The software may be used only in accordance with the terms of the agreement. Actuate software products are protected by U.S. and International patents and patents pending. For a current list of patents, please see <http://www.actuate.com/patents>.

Actuate Corporation trademarks and registered trademarks include:

Actuate, ActuateOne, the Actuate logo, Archived Data Analytics, BIRT, Collaborative Reporting Architecture, e.Analysis, e.Report, e.Reporting, e.Spreadsheet, Encyclopedia, Interactive Viewing, OnPerformance, Performancesoft, Performancesoft Track, Performancesoft Views, Report Encyclopedia, Reportlet, The people behind BIRT, X2BIRT, and XML reports.

Actuate products may contain third-party products or technologies. Third-party trademarks or registered trademarks of their respective owners, companies, or organizations include:

Adobe Systems Incorporated: Flash Player. Apache Software Foundation (www.apache.org): Axis, Axis2, Batik, Batik SVG library, Commons Command Line Interface (CLI), Commons Codec, Derby, Shindig, Struts, Tomcat, Xerces, Xerces2 Java Parser, and Xerces-C++ XML Parser. Bits Per Second, Ltd. and Graphics Server Technologies, L.P.: Graphics Server. Bruno Lowagie and Paulo Soares: iText, licensed under the Mozilla Public License (MPL). Castor (www.castor.org), ExoLab Project (www.exolab.org), and Intalio, Inc. (www.intalio.org): Castor. Codejock Software: Xtreme Toolkit Pro. DataDirect Technologies Corporation: DataDirect JDBC, DataDirect ODBC. Eclipse Foundation, Inc. (www.eclipse.org): Babel, Data Tools Platform (DTP) ODA, Eclipse SDK, Graphics Editor Framework (GEF), Eclipse Modeling Framework (EMF), and Eclipse Web Tools Platform (WTP), licensed under the Eclipse Public License (EPL). Jason Hsueh and Kenton Varda (code.google.com): Protocole Buffer. ImageMagick Studio LLC.: ImageMagick. InfoSoft Global (P) Ltd.: FusionCharts, FusionMaps, FusionWidgets, PowerCharts. Mark Adler and Jean-loup Gailly (www.zlib.net): zLib. Matt Ingenthron, Eric D. Lambert, and Dustin Sallings (code.google.com): Spymemcached, licensed under the MIT OSI License. International Components for Unicode (ICU): ICU library. KL Group, Inc.: XRT Graph, licensed under XRT for Motif Binary License Agreement. LEAD Technologies, Inc.: LEADTOOLS. Microsoft Corporation (Microsoft Developer Network): CompoundDocument Library. Mozilla: Mozilla XML Parser, licensed under the Mozilla Public License (MPL). MySQL Americas, Inc.: MySQL Connector. Netscape Communications Corporation, Inc.: Rhino, licensed under the Netscape Public License (NPL). Oracle Corporation: Berkeley DB. PostgreSQL Global Development Group: pgAdmin, PostgreSQL, PostgreSQL JDBC driver. Rogue Wave Software, Inc.: Rogue Wave Library SourcePro Core, tools.h++. Sam Stephenson (prototype.conio.net): prototype.js, licensed under the MIT license. Sencha Inc.: Ext JS. Sun Microsystems, Inc.: JAXB, JDK, Jstl. ThimbleWare, Inc.: JMemcached, licensed under the Apache Public License (APL). World Wide Web Consortium (W3C)(MIT, ERCIM, Keio): Flute, JTIty, Simple API for CSS. XFree86 Project, Inc.: (www.xfree86.org): xvfb. Yuri Kanivets (code.google.com): Android Wheel gadget, licensed under the Apache Public License (APL). ZXing authors (code.google.com): ZXing, licensed under the Apache Public License (APL).

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

Document No. 110812-2-731302 July 28, 2011

Contents

| | |
|---|----------|
| About Using Information Object Query Builder | v |
|---|----------|

Chapter 1

| | |
|---|----------|
| Using Information Object Query Builder | 1 |
|---|----------|

| | |
|--|---|
| Examining the Information Object Query Builder | 2 |
|--|---|

| | |
|--|---|
| Opening Information Object Query Builder | 2 |
|--|---|

| | |
|---|---|
| Choosing an information object query editor | 4 |
|---|---|

| | |
|------------------------------------|---|
| Using the expression builder | 4 |
|------------------------------------|---|

| | |
|--|---|
| Creating an information object query in the Basic Design interface | 5 |
|--|---|

| | |
|--|---|
| Creating a customized graphical information object query | 9 |
|--|---|

| | |
|---|----|
| Selecting one or more information objects | 11 |
|---|----|

| | |
|--------------------------------|----|
| Hiding column categories | 12 |
|--------------------------------|----|

| | |
|-------------------------------|----|
| Defining output columns | 13 |
|-------------------------------|----|

| | |
|---------------------------------|----|
| Setting column properties | 14 |
|---------------------------------|----|

| | |
|-------------------------|----|
| Specifying a join | 15 |
|-------------------------|----|

| | |
|-------------------|----|
| About joins | 15 |
|-------------------|----|

| | |
|------------------------|----|
| Optimizing joins | 18 |
|------------------------|----|

| | |
|----------------------|----|
| Filtering data | 20 |
|----------------------|----|

| | |
|-----------------------------------|----|
| Creating a filter condition | 20 |
|-----------------------------------|----|

| | |
|---|----|
| Creating multiple filter conditions | 28 |
|---|----|

| | |
|-----------------------------------|----|
| Prompting for filter values | 31 |
|-----------------------------------|----|

| | |
|--|----|
| Setting dynamic filter prompt properties | 33 |
|--|----|

| | |
|---------------------|----|
| Grouping data | 36 |
|---------------------|----|

| | |
|----------------------------------|----|
| Creating a GROUP BY clause | 37 |
|----------------------------------|----|

| | |
|--|----|
| Removing a column from the GROUP BY clause | 39 |
|--|----|

| | |
|--|----|
| Filtering on an aggregate column | 41 |
|--|----|

| | |
|---------------------------|----|
| Defining parameters | 42 |
|---------------------------|----|

| | |
|--|----|
| Specifying a parameter's prompt properties | 44 |
|--|----|

| | |
|------------------------------------|----|
| Setting parameter properties | 46 |
|------------------------------------|----|

| | |
|---------------------------------|----|
| Setting source parameters | 48 |
|---------------------------------|----|

| | |
|---------------------------------------|----|
| Synchronizing source parameters | 49 |
|---------------------------------------|----|

| | |
|---|----|
| Creating a textual information object query | 50 |
|---|----|

| | |
|---------------------------------|----|
| Displaying output columns | 52 |
|---------------------------------|----|

| | |
|-----------------------------|----|
| Displaying parameters | 52 |
|-----------------------------|----|

| | |
|--|----|
| Displaying information object query output | 53 |
|--|----|

Chapter 2

| | |
|------------------------------------|-----------|
| Actuate SQL reference | 55 |
|------------------------------------|-----------|

| | |
|-------------------------|----|
| About Actuate SQL | 56 |
|-------------------------|----|

| | |
|--|----|
| Differences between Actuate SQL and ANSI SQL-92 | 56 |
| Limitations compared to ANSI SQL-92 | 56 |
| Extensions to ANSI SQL-92 | 57 |
| Database limitations | 60 |
| FILTERS statement in report designers | 60 |
| Actuate SQL syntax | 61 |
| Actuate SQL grammar | 62 |
| Using white space characters | 66 |
| Using keywords | 66 |
| Using comments | 67 |
| Specifying maps and information objects in Actuate SQL queries | 68 |
| Using identifiers in Actuate SQL | 68 |
| Using column aliases in Actuate SQL | 68 |
| Specifying parameter values | 68 |
| Using subqueries in Actuate SQL | 70 |
| Using derived tables in Actuate SQL | 71 |
| Data types and data type casting | 71 |
| Facets | 71 |
| Casting rules | 72 |
| String comparison and ordering | 73 |
| Functions and operators | 74 |
| Comparison operators: =, <>, >=, >, <=, < | 74 |
| Range test operator: BETWEEN | 74 |
| Comparison operator: IN | 75 |
| Arithmetic operators: +, -, *, / | 75 |
| Numeric functions | 76 |
| FLOOR, CEILING, MOD | 76 |
| ROUND | 77 |
| POWER | 77 |
| Null test operators: is [not] null | 78 |
| Logical operators: and, or, not | 78 |
| String functions and operators | 78 |
| Case conversion functions: UPPER, LOWER | 78 |
| Concatenation operator: | 79 |
| Length function: CHAR_LENGTH | 79 |
| LIKE operator | 79 |
| Substring functions: LEFT, RIGHT, SUBSTRING | 80 |
| Trimming functions: LTRIM, RTRIM, TRIM | 81 |
| Search function: POSITION | 82 |
| Timestamp functions | 82 |
| CURRENT_TIMESTAMP | 83 |
| CURRENT_DATE | 83 |
| DATEADD | 84 |

| | |
|---|-----------|
| DATEDIFF | 84 |
| DATEPART | 85 |
| DATESERIAL | 85 |
| Aggregate functions: COUNT, MIN, MAX, SUM, AVG | 86 |
| System function: CURRENT_USER | 87 |
| Providing query optimization hints | 87 |
| Indicating that a table in a join is optional | 88 |
| Using the OPTIONAL keyword with a computed field | 89 |
| Using the OPTIONAL keyword with parentheses () | 90 |
| Using the OPTIONAL keyword with aggregate functions | 91 |
| Specifying the cardinality of a join | 92 |
| Using pragmas to tune a query | 93 |
| Disabling cost-based optimization | 93 |
| Disabling indexing | 95 |
| Specifying a threshold value for indexing | 96 |
| Index | 97 |

About Using Information Object Query Builder

Using Information Object Query Builder provides information about using an information object as a data source in a BIRT spreadsheet report or e.report design.

Using Information Object Query Builder includes the following chapters:

- *About Using Information Object Query Builder.* This chapter provides an overview of this guide.
- *Chapter 1. Using Information Object Query Builder.* This chapter describes procedures for accessing data from information objects created in Actuate Information Object Designer.
- *Chapter 2. Actuate SQL reference.* This chapter describes the differences between Actuate SQL and ANSI SQL-92.

1

Using Information Object Query Builder

This chapter contains the following topics:

- Examining the Information Object Query Builder
- Creating an information object query in the Basic Design interface
- Creating a customized graphical information object query
- Creating a textual information object query
- Displaying information object query output

Examining the Information Object Query Builder

Information Object Query Builder supports defining a query on information objects created using Actuate Information Object Designer. Use the query builder to specify the data to include, the sort order, and parameters for filtering the data. Information Object Query Builder produces a query that obtains data from the information object data source. A report developer uses this query builder to create an information object query or change an existing information object query.

BIRT Spreadsheet Designer and e.Report Designer Professional use Information Object Query Builder. Information Object Query Builder runs as an application separate from the designer applications. You must exit Information Object Query Builder before returning to work in the designer application. When you close Information Object Query Builder, Windows sometimes does not display the designer application as the top window. In that event, select the designer application from the Windows task bar.

Opening Information Object Query Builder

After creating a connection to an information object data source, the next step is to open Information Object Query Builder to create a query. You can use this step to reopen an existing information object query.

How to open Information Object Query Builder using BIRT Spreadsheet Designer

- 1 In BIRT Spreadsheet Designer, choose Report➤Create Data Set.
- 2 On New Data Set, in Data Set Name, type a name for your data set.
- 3 In Data Source, select an Actuate Data Integration Service Connection data source. The Data Set Type is already set to Actuate Data Integration Service Connection Data Source, as shown in Figure 1-1.



Figure 1-1 Creating an Actuate Data Integration Service Connection data set
Choose Finish.

- 4 In Actuate Data Integration Service Data Source Properties, type the name of the data set and press Edit. Connection Properties appears.

- 5 Type the appropriate values for the fields that are described in Table 1-1. Then, choose OK.

| Table 1-1 Property values for accessing Information Object Query Builder | |
|---|--|
| Field | Description |
| BIRT iServer | The URL to the iServer that manages the Encyclopedia volume with the information objects |
| Port number | The port number to access the iServer |
| Volume | The Encyclopedia volume containing the information objects |
| User name | The name of the Encyclopedia volume account with access to the information objects |
| Password | The password for the Encyclopedia volume account with access to the information objects |

Information Object Query Builder Basic Design appears. You can then choose how you want to develop your information object query.

How to open Information Object Query Builder using e.Report Designer Professional

- 1 Select an information object data source component and choose Data. If you use a report wizard to create a report with an information object data source, that wizard starts Information Object Query Builder.
- 2 On Connection Properties, type in the appropriate values for the fields that are described in Table 1-2, then choose OK.

| Table 1-2 Property values for accessing Information Object Query Builder | |
|---|--|
| Field | Description |
| Password | The password for the Encyclopedia volume account with access to the information objects |
| ServerUri | The URI to the iServer, including the server name and port number, that manages the Encyclopedia volume with the information objects, for example http://MyServer:8000 |
| User name | The name of the Encyclopedia volume account with access to the information objects |
| Volume | The Encyclopedia volume containing the information objects |

- 3 Choose Finish. Information Object Query Builder Basic Design appears. You can then choose how you want to develop your information object query.

Choosing an information object query editor

You can create an information object query in three ways, depending on the number of information objects you access in your query and your familiarity with SQL concepts and the SQL language. Table 1-3 provides a brief description of the differences between the interfaces. When you open Information Object Query Builder, you view the Basic Design interface by default.

Table 1-3 Differences between the three Information Object Query Builder interfaces

| Interface capability | Basic Design (graphical) | Advanced Design (graphical) | Advanced Design (textual) |
|---|--------------------------|-----------------------------|---------------------------|
| Can access more than one information object in the query | No | Yes | Yes |
| Provides the ability to specify sorting options, filtering, and parameters | Yes | Yes | Yes |
| Provides the ability to create a complex query | No | Yes | Yes |
| Provides a graphical interface | Yes | Yes | No |
| Can access the expression builder | Yes | Yes | No |
| Requires understanding of SQL concepts such as joins, grouping, distinct rows, and filtering on an aggregate column | No | Yes | Yes |
| Requires the ability to write Actuate SQL code | No | No | Yes |
| Can open an existing query created or modified in the Basic Design interface | Yes | Yes | Yes |
| Can open an existing query created or modified in the Advanced Design interface | No | Yes | Yes |
| Can open an existing query created or modified in the textual editor | No | No | Yes |



To access the Advanced Design graphical interface, open Information Object Query Builder, and choose Advanced Perspective.



To access the textual editor, open Information Object Query Builder, and choose Advanced Perspective. On Query Design, choose SQL Editor.

Using the expression builder

When designing a query, Actuate SQL expressions support specifying filters or joins, creating aggregate data, and so on. For example, the expression, `officeID = 101`, specifies that data returned by the query must have 101 in the `officeID` column. You can type these expressions in any of the three editors. In the

textual editor, the expressions are part of the SQL SELECT statement. In the Basic Design and Advanced Design graphical query editors, you can type the expressions or use the expression builder to develop expressions. Expression Builder helps you create expressions by providing a graphical interface for selecting column names, constants, functions, operators, and so on from lists.

Use the expression builder to create Actuate SQL expressions on the filter page of Information Object Query Builder Basic Design and many pages in Information Object Query Builder Advanced Design.

Figure 1-2 shows expression builder. Drag items from the left pane to the right pane or insert items by choosing the appropriate icon. If you select a function in the left pane, the function signature appears in the bottom pane.

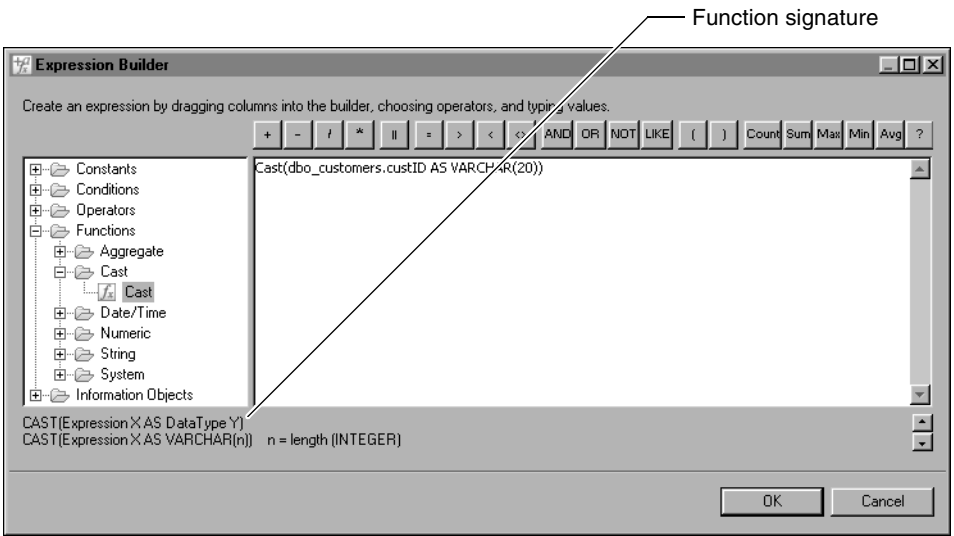


Figure 1-2 Using Expression Builder to create expressions

Creating an information object query in the Basic Design interface

You can create a query on a single information object using Information Object Query Builder Basic Design. This query editor supports specifying sorting options, filtering, and parameters. If you need to work with more than one information object or require more customization than this query editor supports, use Information Object Query Builder Advanced Design.

You can browse an iServer System Encyclopedia volume in Information Object Query Builder to find and select the information object for the query. If you have already chosen your information object and are re-entering Information Object

Query Builder, the editor displays the previously selected information object. If an information object that is used by an existing query no longer exists, you must specify a new information object and its columns, parameters, and filters.

An expanded folder does not reflect changes to the volume. If you or someone else loads a new information object to the volume after you expand a folder, you must collapse and expand the folder to see the changes in the information object.

To specify a basic information object query, perform these tasks:

- Start Information Object Query Builder.
- Select an information object for this query.
- Specify the columns that you want to include in this query.
- Specify any additional information that you want in the query:
 - Specify how you want to sort the data.
 - Specify whether users can provide parameter values when they run the report.
 - Specify the default values of information object parameters.
 - Specify any filtering that you want to restrict the data returned by the information object query.

How to create a basic information object query

- 1 Start Information Object Query Builder. Information Object Query Builder displays the Basic Design interface by default.
- 2 Select an information object and columns by following these steps:
 - 1 In iServer Explorer, expand the Servers node and the Encyclopedia volume node, as shown in Figure 1-3. Expand folders as necessary to view the information objects.

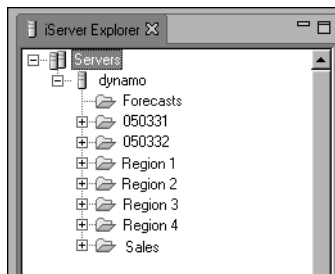


Figure 1-3 iServer Explorer

- 2 Expand the information object to see the columns and parameters, as shown in Figure 1-4.

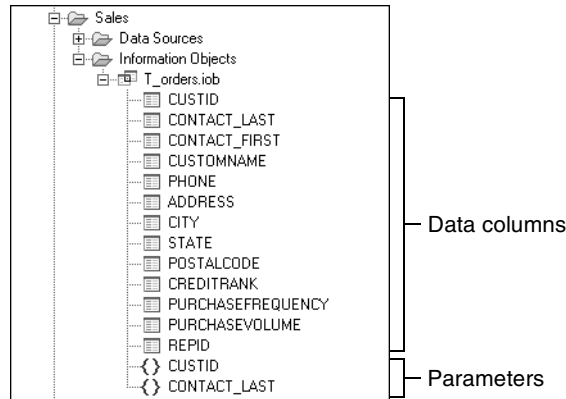


Figure 1-4 Expanding an information object

- 3 In iServer Explorer, double-click the information object to use in the query. All columns in the information object appear in Output Columns. Beside each column, a check mark indicates that the query uses the column, as shown in Figure 1-5.

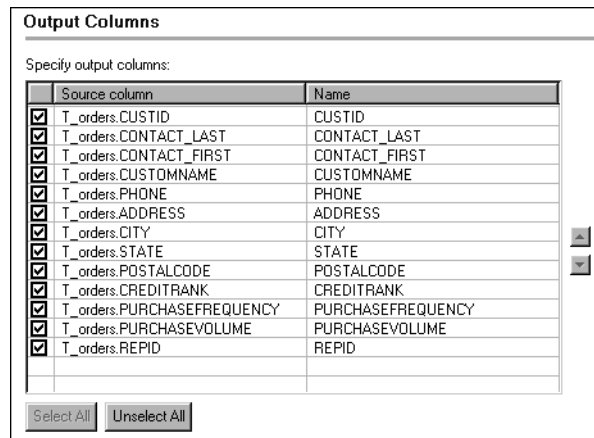


Figure 1-5 Specifying columns to use in the query

- 4 Deselect any columns that you do not want to include in the output.
- 5 To move a column in the list, select the column, and choose the up or down arrow until the column is in the right position.
- 3 Specify the order in which to sort the data:
 - 1 On Query Design, choose Select Sort Order.
 - 2 On Sort, in Available, expand the information object to see the columns.
 - 3 Double-click a column on which to sort. The column appears in Selected.

- 4 Under Order, as shown in Figure 1-6, click the field beside the first column on which you want to sort, and use the drop-down list to specify the sort direction for the column:
 - ❑ For ascending order, select Asc.
 - ❑ For descending order, select Desc.

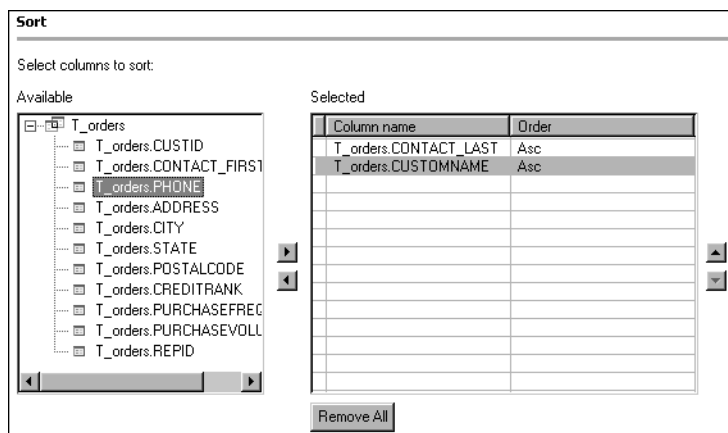


Figure 1-6 Specifying data sort order

- 5 Repeat substeps 3 and 4 for any additional columns on which to sort.
 - 6 To change the position of a column in the sorting hierarchy, select the column and choose the up or down arrow key. The data returned by the query is sorted first by the column at the top of the list, then by each subsequent column in the list.
- 4 For each available parameter, specify any default parameter values and whether users can specify the values of those parameters.
- 1 On Query Design, choose Define Parameters. Parameters appears, as shown in Figure 1-7.

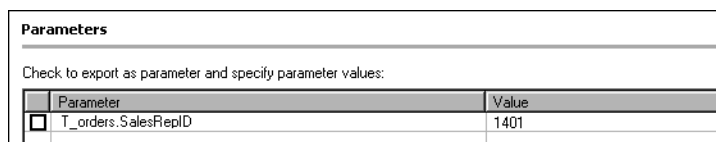


Figure 1-7 Specifying parameters to export

- 2 To enable report users to specify single parameter values, export the parameters. In the left column, select each available parameter that you want to export. Parameters appear on the Requester page.
- 3 Under Value, specify a for each available parameter. If you export the parameter, specifying a default value is optional.

- 5 Specify filters to limit the data returned from the query by following these steps:
 - 1 On Query Design, choose Define Filters.
 - 2 Define the filters. The procedures for adding, editing, and removing filter conditions in Information Object Query Builder Basic Design are the same as the procedures for adding, editing, and removing filter conditions in Information Object Query Builder Advanced Design.
- Choose OK to save the query and return to the report design.

Creating a customized graphical information object query

To specify a graphical information object query using Information Object Query Builder Advanced Design, perform the following tasks:

- Start Information Object Query Builder and enter the Advanced Design perspective.
- Define the query:
 - Select one or more information objects for this query.
 - Define the columns that you want to include in this query.
 - If you have more than one information object, specify the joins to use in this query.
 - Specify any filtering that you want on individual columns.
 - Specify the sort order for the data.
 - Select the columns that you want to group in the query.
 - Specify any aggregate columns that you want to filter in the query.
 - Specify parameters that report users can provide values for when they run the report.
 - Choose OK to save the query and return to the report design.

While developing your query, you can use the following tools:

- Problems pane



As you design your queries using Information Object Query Builder Advanced Design, error messages appear in Problems, as shown in Figure 1-8. If an error description is truncated, select the error message. Information Object Query Builder Advanced Design displays an ellipsis button at the end of the description column for that error message. To view the complete

description for that error message, choose ellipsis. Line numbers refer to the standard Actuate SQL query, not the extended Actuate SQL query.



To filter the error messages, choose Filters. For more information about using Problems or Filters, see the *Workbench User Guide* in the Information Object Query Builder Advanced Design online help.

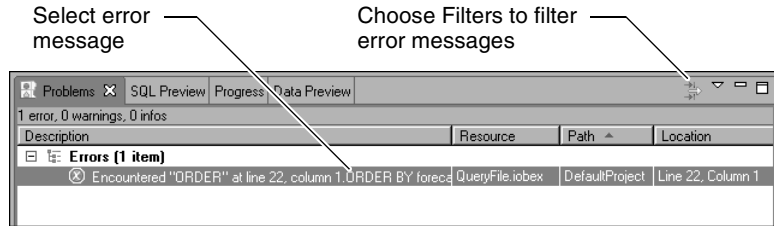


Figure 1-8 Locating errors and filtering error messages

■ SQL Preview pane

While using Information Object Query Builder Advanced Design to create a query, you can view the resulting Actuate SQL query statement. To display the query, choose SQL Preview, as shown in Figure 1-9. If you modify the query, choose Refresh to update the display.

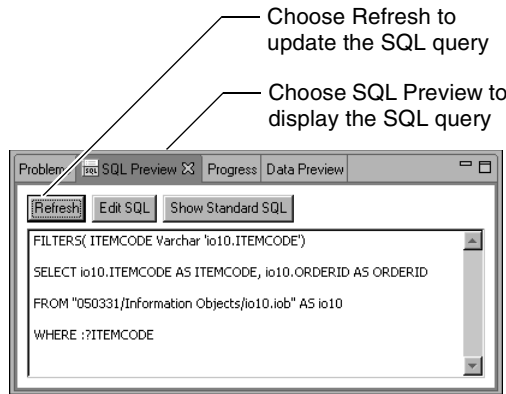


Figure 1-9 Displaying the SQL for an information object query

If you use a dynamic filter in your query, the Actuate SQL query includes a FILTERS clause and :? syntax. The FILTERS clause and :? syntax are part of extended Actuate SQL. The corresponding standard Actuate SQL statements substitute dynamic filters with WHERE clause conditions of the correct data type. To see the standard Actuate SQL query, choose Show Standard SQL in the SQL Preview pane.

Information Object Query Builder uses the standard Actuate SQL statement to validate the syntax of the query. If Information Object Query Builder reports a syntax error, the line number in the error message refers to the syntax that appears in standard SQL. If the query contains syntax errors, use Show

Standard SQL to locate and identify the syntax errors. You can then return to viewing the extended Actuate SQL syntax by choosing Show Extended SQL.

- **Progress pane**
Choose Progress to view the progress of long-running tasks. Typically, tasks do not run long enough for Progress to be used. For more information about using Progress, see the *Workbench User Guide* in the Information Object Query Builder Advanced Design online help.
- **Data Preview pane**
Choose Data Preview to view the data rows returned by the query.

Selecting one or more information objects

You can browse an iServer Encyclopedia volume using Information Object Query Builder to find and select the information objects for the query.

If you have already chosen your information objects and are re-entering Information Object Query Builder, the editor displays the previously selected information objects. If an information object used by an existing query no longer exists, you must specify a new information object and specify its columns, parameters, and filters.

An expanded folder does not reflect changes to the Encyclopedia volume. If you or someone else loads a new information object in the volume after you expand a folder, you must collapse and expand the folder to see the information object.

How to select an information object

- 1 In iServer Explorer, expand the Servers node and the Encyclopedia volume node, as shown in Figure 1-10.

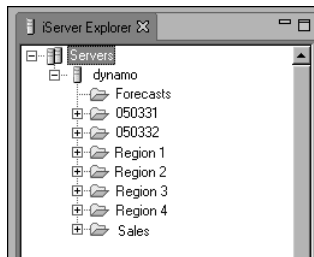


Figure 1-10 Viewing information objects in iServer Explorer

- 2 To view your information objects, expand the appropriate folders.
- 3 Drag the appropriate information objects from iServer Explorer to the upper pane of Query Design. The items appear in the upper pane of Query Design, as shown in Figure 1-11. By default, Information Object Query Builder selects all columns in each information object.

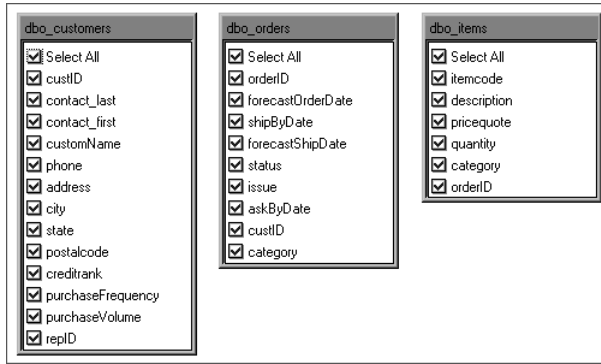


Figure 1-11 Selected information objects as they appear in Query Design

Hiding column categories

To help you locate information object columns, the data modeler can organize them into categories. For example, for an information object that returns customer data, the data modeler can create a Customer address category that contains the columns StreetAddress, City, State, and PostalCode.



Column categories appear in iServer Explorer, the upper pane of Query Design, and expression builder. To hide column categories in Query Design, select Toggle categories view in the upper right corner of the information object, as shown in Figure 1-12. The information object on the left displays the Customer address category. The information object on the right does not display column categories.

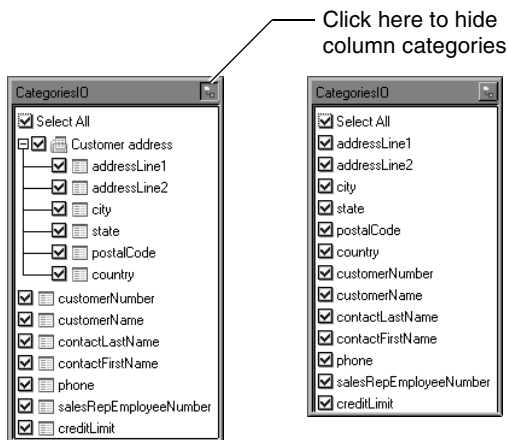


Figure 1-12 Information object with and without categories displayed

To hide column categories by default, choose Window→Preferences and deselect Show categories in graphical editor by default in Preferences—Information Objects, as shown in Figure 1-13.

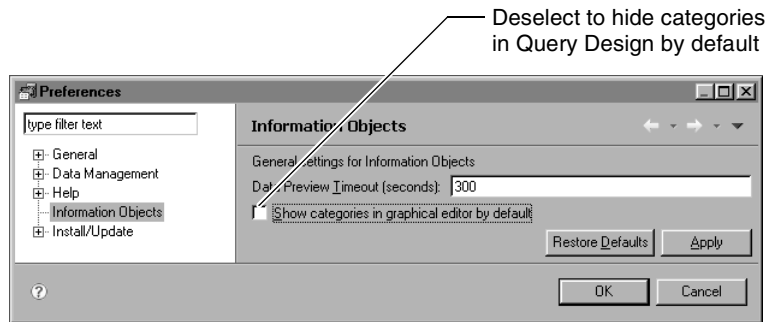


Figure 1-13 Preferences—Information Objects

Defining output columns

To define the output columns for an information object query, use Query Design—Columns. For example, you can create the following SQL fragment:

```
SELECT ename AS employee, (salary * 12) AS annual_comp
FROM Employees
```

How to define output columns

- 1 In Query Design, choose Columns.
- 2 In the upper pane of Query Design, select the columns to include and deselect the columns to exclude from the query. To select all columns, select Select All at the top of the listing for that information object. By default, the query includes all columns in an information object. Selected columns appear in Columns.
- 3 In Query Design—Columns:
 - To return only distinct rows, select Distinct values only. Some queries return duplicate rows. In a group of duplicate rows, each selected column contains the same value for all the rows in the group. To return only one row for each group of duplicate rows, select Distinct values only. This setting affects only rows in which all column values match. The query returns rows in which only some of the column values match.
 - To change a column alias, type the new alias in Name. If a column alias contains a special character, such as a period (.) or a space, enclose the alias in quotation marks ("). Do not use column aliases that are identical except for case. For example, do not use both status and STATUS as column aliases.
 - To enter an expression, select the source column, and type the expression or choose ellipsis, as shown in Figure 1-14. Choosing ellipsis opens expression builder.





- To change the order of the columns, use the up and down arrows.

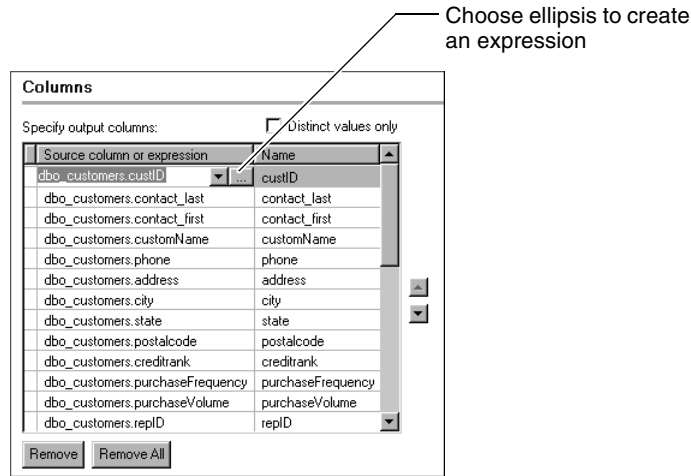


Figure 1-14 Defining output columns

- 4 To define column properties, such as the display name, select the column in Columns, and define the properties in Properties.

How to delete output columns

To delete an output column, select the column in Query Design—Columns, and choose Remove. To delete all output columns, choose Remove All.

Setting column properties

Table 1-4 lists column properties and a description of each property.

Table 1-4 Column properties

| Column property | Can set? | Description |
|-----------------|----------|---|
| Aggregate Type | Not used | Not used. |
| Analysis Type | Yes | Analysis type in e.Analysis. If the column contains numeric values or the data type is unknown, the default is Measure. If the column contains data of type TIMESTAMP , the default is Dimension. If the column contains data of type VARCHAR or BOOLEAN , the default is Attribute. If the column is a primary key, a foreign key, or an indexed column in the database, the default is Dimension even if the column contains numeric values. |
| Category Path | No | Path for column category and subcategories. |

Table 1-4 Column properties

| Column property | Can set? | Description |
|----------------------|-------------------------|--|
| Data Type | No | Actuate SQL data type. If the data type is unknown, choose the Synchronize button. |
| Description | Not used | Not used. |
| Display Format | Not used | Not used. |
| Display Length | Yes | Number of characters to allow for display of column values in report output. |
| Display Name | Not used | Not used. |
| Expression | Yes, on the Columns tab | Expression for a computed field. |
| Has Null | Yes | If column contains NULLs, set to True. Otherwise, set to False. |
| Heading | Not used | Not used. |
| Help Text | Not used | Not used. |
| Horizontal Alignment | Not used | Not used. |
| Indexed | No | Indicates whether the column is indexed in the data source. True indicates that the column is indexed. False indicates that it is not indexed. |
| Name | Yes, on the Columns tab | The alias for the column in the information object query. |
| Text Format | Not used | Not used. |
| Word Wrap | Not used | Not used. |

Specifying a join

To define the joins for an information object query, use Query Design—Joins. For example, the following SQL fragment specifies that the value of the custID column in the Customers table must match the value of the custID column in the Orders table. The query returns no rows for Customer records having no matching Order records.

```
FROM Customers  
INNER JOIN Orders ON (Customers.custID = Orders.custID)
```

About joins

A join specifies how to combine data from two information objects. The information objects do not have to be based on the same data source. A join

consists of one or more conditions that must all be true. In the resulting SQL SELECT statement, join conditions are linked with AND.

A join can consist of multiple conditions in the following form:

```
columnA = columnB
```

A join can have only one condition that uses an operator other than equality (=) or an expression, for example:

```
columnA < columnB
```

Information Object Query Builder Advanced Design does not support right outer joins or full outer joins.

How to define a join condition

- 1 In Query Design, choose Joins.
- 2 In the upper pane of Query Design, drag the join column from the first information object, and drop it on the join column in the second information object.

The upper pane shows the join condition, like the one in Figure 1-15, and the join columns and operator are listed in Query Design—Joins.

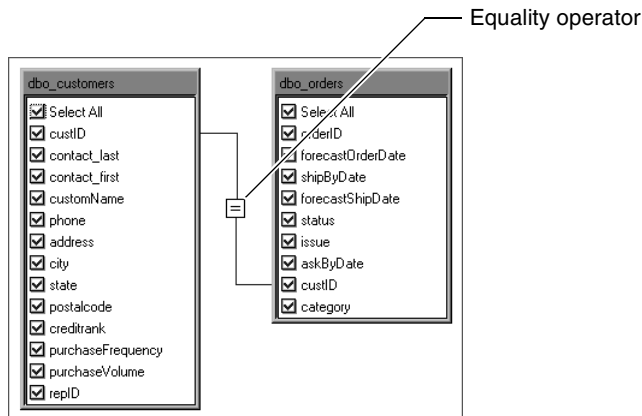



Figure 1-15 Joined columns from two information objects

- 3 In Query Design—Joins, select the row that describes the new join condition.
- 4 If necessary, select a different join condition operator from the drop-down list. By default, Information Object Query Builder Advanced Design uses the equality operator (=) to relate two columns.
- 5  To change a column name to an expression, select the column name, and type the expression, or choose Ellipsis to display the expression builder, as shown in Figure 1-16.

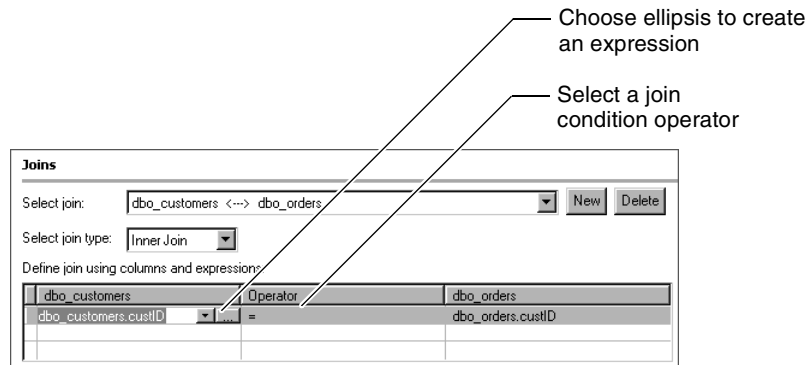


Figure 1-16 Defining a join condition

If the join has a condition that uses an operator other than equality (=) or an expression, the symbol shown in Figure 1-17 appears in the upper pane of Query Design.

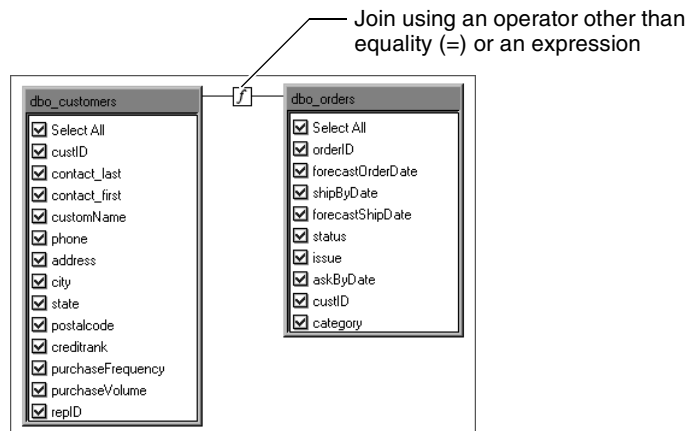


Figure 1-17 A join that uses an expression or an operator other than equality

- 6 If the join consists of more than one condition, repeat this procedure for the other conditions.
- 7 Choose one of the following join types:
 - Inner join
 - Left outer join
- 8 Optimize the join.

How to delete a join condition

To delete a join condition, select the join condition in the upper pane of Query Design, and press Delete.

Optimizing joins

You can improve a query's performance by optimizing the joins. To optimize a join, you can specify the cardinality of the join. Specifying the cardinality of the join adds the **CARDINALITY** keyword to the Actuate SQL query. If your query is based on two or more information objects that are based on different data sources, you can also optimize the joins by specifying join algorithms.

Figure 1-18 shows how to specify the cardinality of a join and how to specify a join algorithm in Query Design—Joins.

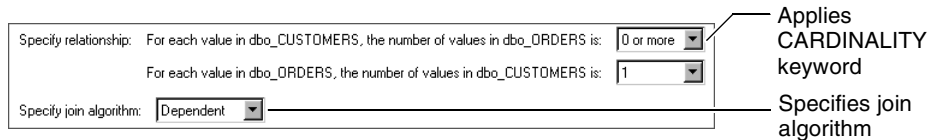


Figure 1-18 Optimizing a join

When joining information objects built from different data sources, the Actuate SQL compiler chooses a join algorithm. If you have a good understanding of the size and distribution of the data, however, you can specify the join algorithm. Choosing the correct join algorithm can significantly reduce information object query execution time. Actuate SQL supports three join algorithms:

- Dependent
- Merge
- Nested Loop

When you join information objects that are built from the same data source, specifying a join algorithm has no effect. The join is processed by the data source.

About dependent joins

A dependent join is processed in the following way:

- The left side of the join statement is executed, retrieving all the results. The results are then processed one at a time (pipelined).
- For each left side result, the right side of the join is executed, parameterized by the values provided by the current left side row.

A dependent join is advantageous when the cardinality of the left side is small, and the selectivity of the join criteria is both high and can be delegated to the data source. When the cardinality of the left side is high, a dependent join is relatively slow because it repeatedly executes the right side of the join.

Dependent joins can be used for any join criteria, but only join expressions that can be delegated to the right side's data source result in improved selectivity performance.

About merge joins

A merge join is processed in the following way:

- The left side of the join statement executes, retrieving all the results sorted by the left side data source. The results are then processed one at a time (pipelined).
- The right side of the join statement executes, retrieving all the results sorted by the right side data source. The results are then processed one at a time (pipelined).

A merge join supports only an equijoin. A merge join has much lower memory requirements than a nested loop join and can be much faster. A merge join is especially efficient if the data sources sort the rows.

About nested loop joins

A nested loop join is processed in the following way:

- The left side of the join statement is executed, retrieving all the results. The results are then processed one at a time (pipelined).
- The right side of the join statement is executed. The results are materialized in memory. For each row on the left side, the materialized results are scanned to find matches for the join criteria.

A nested loop join is advantageous when the cardinality of the right side is small. A nested loop join performs well when the join expression cannot be delegated to the data source. A nested loop join supports any join criteria, not just an equijoin.

A nested loop join is a poor choice when the cardinality of the right side is large or unknown, because it may encounter memory limitations. Increasing the memory available to the Integration service removes this limitation. The Integration service parameter Max memory per query specifies the maximum amount of memory to use for an Integration service query. For more information about this parameter, see *Configuring BIRT iServer*.

How to specify a join algorithm

In Query Design—Joins, select the appropriate join and choose one of the following from the Specify join algorithm drop-down list shown in Figure 1-19:

- Dependent
- Merge
- Nested loop

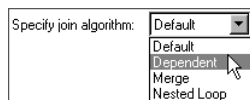


Figure 1-19 Specifying the join algorithm

Filtering data

If an information object query returns more data rows than you need, restrict the number of data rows by using a filter. For example, rather than list all customer sales, create a filter to select only the sales data for a particular week or only the sales data for a particular region.

Filtering data helps you work effectively with large amounts of data. It enables you to find the necessary pieces of information to answer specific business questions, such as which sales representatives generated the top ten sales accounts, which products generated the highest profit in the last quarter, which customers have not made a purchase in the past 90 days, and so on.

Filtering data can also have a positive effect on processing speed. Limiting the number of data rows can reduce the load on the databases because the information object query does not need to return all the rows every time it is run.

Creating a filter condition

When you create a filter, you define a condition that specifies which data rows to return. A filter condition is an If expression that must evaluate to true in order for a data row to be returned. For example:

```
If the order total is greater than 10000  
If the sales office is San Francisco  
If the order date is between 4/1/2008 and 6/30/2008
```

Filter conditions are appended to the information object query's WHERE clause, for example:

```
WHERE OrderTotal > 10000 AND SalesOffice LIKE 'San Francisco%' AND  
      OrderDate BETWEEN TIMESTAMP '2008-04-01 00:00:00' AND TIMESTAMP  
      '2008-06-30 00:00:00'
```

Figure 1-20 shows an example of a condition defined in Filter Conditions. Filter Conditions helps you define the condition by breaking it down into the following parts:

- The column to evaluate, such as credit limit.
- The comparison operator that specifies the type of comparison test, such as > (greater than).
- The value to which all values in the column are compared, such as 10000.

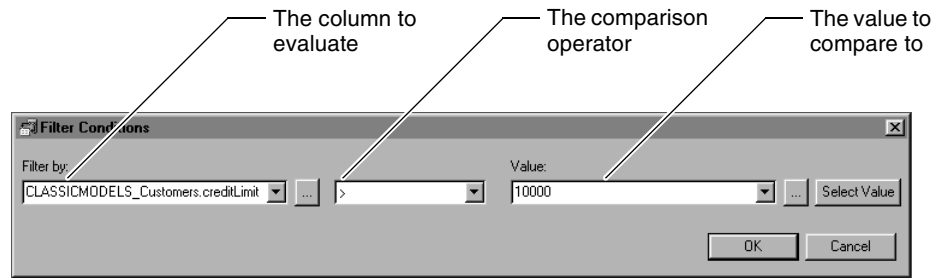


Figure 1-20 Filter Conditions displaying a filter condition

Table 1-5 lists the operators you can use when you create expressions for filter conditions.

Table 1-5 Operators in filter condition expressions

| Operator | Use to test whether | Example |
|-------------------------------|---|--|
| BETWEEN | A column value is between two specified values. | Profit BETWEEN 1000 AND 2000 |
| = (Equal to) | A column value is equal to a specified value. | CreditLimit = 100000 |
| > (Greater than) | A column value is greater than a specified value. | Total > 5000 |
| >= (Greater than or equal to) | A column value is greater than or equal to a specified value. | Total >= 5000 |
| IN | A column value is in the specified set of values. | Country IN ('USA', 'Canada', 'Mexico') |
| IS NOT NULL | A column value is not a null value. A null value means that no value is supplied. | CreditLimit IS NOT NULL |
| IS NULL | A column value is a null value. | CreditLimit IS NULL |
| < (Less than) | A column value is less than a specified value. | Total < 5000 |
| <= (Less than or equal to) | A column value is less than or equal to a specified value. | Total <= 5000 |
| LIKE | A column value matches a string pattern. | ProductName LIKE 'Ford%' |
| NOT BETWEEN | A column value is not between two specified values. | Profit NOT BETWEEN 1000 AND 2000 |
| <> (Not equal to) | A column value is not equal to a specified value. | CreditLimit <> 100000 |

(continues)

Table 1-5 Operators in filter condition expressions (continued)

| Operator | Use to test whether | Example |
|----------|---|--|
| NOT IN | A column value is not in the specified set of values. | Country NOT IN ('USA', 'Canada', 'Mexico') |
| NOT LIKE | A column value does not match a string pattern. | ProductName NOT LIKE 'Ford%' |

How to create a filter condition

- 1 In Query Design, choose Filters.
- 2 On Query Design—Filters, choose New.
- 3 In Filter Conditions, in Filter by, do one of the following:
 - Select a column from the drop-down list. The drop-down list contains the non-aggregate columns that you defined on Query Design—Columns. To create a filter for an aggregate column, use Query Design—Having.
 - Type an expression.
 - Choose Ellipsis to create an expression.
- 4 Select the comparison test, or operator, to apply to the selected column or expression. Depending on the operator you select, Filter Conditions displays one or two additional fields, or a completed filter condition.
- 5 If you selected an operator that requires a comparison value, specify the value in one of the following ways:
 - Type the value or expression.
 - If you selected a column in Filter by, choose Select Value to select from a list of values. Figure 1-21 shows the selection of Boston from a list of possible sales office values.

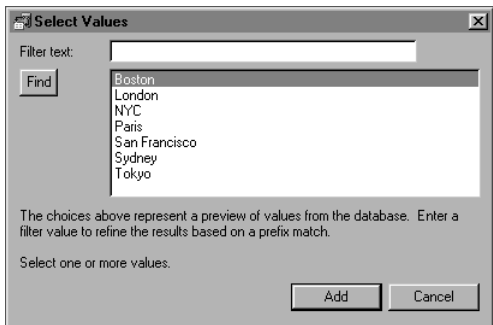


Figure 1-21 Select Values showing the values in the selected column

- Select a parameter or column from the drop-down list. You create parameters on Query Design—Parameters.



- Choose Ellipsis to create an expression.
Figure 1-22 shows the completed filter condition.

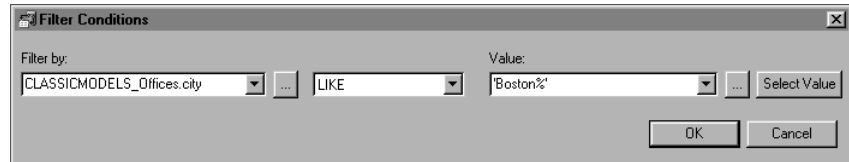


Figure 1-22 Filter Conditions displaying a completed filter condition

Choose OK. The filter condition appears on Query Design—Filters as shown in Figure 1-23.

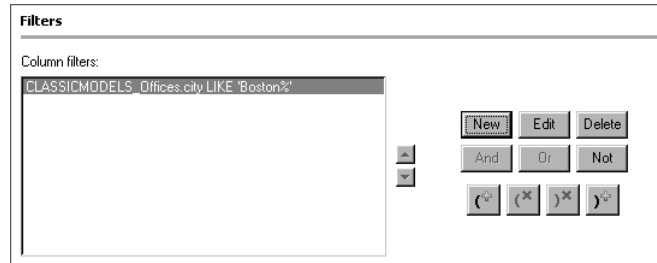


Figure 1-23 Query Design—Filters displaying a filter condition

- 6 Display the Actuate SQL query. Verify that the filter condition is appended to the WHERE clause and that the syntax is correct, for example:

```
WHERE SalesOffice LIKE 'Boston%'
```

How to create a filter condition using Actuate SQL

- 1 In Query Design, choose Filters.
- 2 On Query Design—Filters, complete the following tasks:
 - Click in the text box.
 - Type the filter condition using Actuate SQL, as shown in Figure 1-24. If a table or column identifier contains a special character, such as a space, enclose the identifier in double quotation marks (").

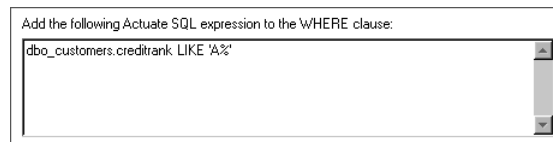


Figure 1-24 Using Actuate SQL to create a filter condition

Selecting multiple values for a filter condition

So far, the filter examples specify one comparison value. Sometimes you need to view more data, for example, sales details for several sales offices, not for only one office. To select more than one comparison value, select the IN operator, choose Select Values, then select the values. To select multiple values, press Ctrl as you select each value. To select contiguous values, select the first value, press Shift, and select the last value. This action selects the first and last values and all the values in between.

Figure 1-25 shows the selection of London and Paris from a list of sales office values.

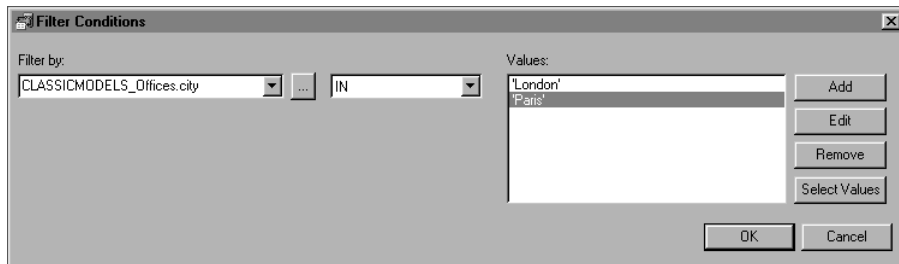


Figure 1-25 Filter Conditions showing the selection of multiple comparison values

Excluding data

You use comparison operators, such as = (equal to), > (greater than), or < (less than), to evaluate the filter condition to determine which data to include. Sometimes it is more efficient to specify a condition that excludes a small set of data. For example, you need sales data for all countries except USA. Instead of selecting all the available countries and listing them in the filter condition, simply use the NOT LIKE operator. Similarly, use NOT BETWEEN to exclude data in a specific range, and <> (not equal to) to exclude data that equals a particular value.

For example, the following filter condition excludes orders with amounts between 1000 and 5000:

```
OrderAmount NOT BETWEEN 1000 AND 5000
```

The filter condition in the next example excludes products with codes that start with MS:

```
ProductCode NOT LIKE 'MS%'
```

Filtering empty or blank values

Sometimes, rows display nothing for a particular column. For example, suppose a customer database table contains an e-mail field. Some customers, however, do not supply an e-mail address. In this case, the e-mail field might contain an empty value or a blank value. An empty value, also called a null value, means no value is supplied. A blank value is entered as " (two single quotes without spaces) in the

database table field. Blank values apply to string fields only. Null values apply to all data types.

You can create a filter to exclude data rows where a particular column has null or blank values. You use different operators to filter null and blank values.

When filtering to exclude null values, use the IS NOT NULL operator. If you want to view only rows that have null values in a particular column, use IS NULL. For example, the following filter condition excludes customer data where the e-mail column contains null values:

```
email IS NOT NULL
```

The following filter condition displays only rows where the e-mail column contains null values:

```
email IS NULL
```

When filtering blank values, use the NOT LIKE operator with " (two single quotes without spaces) as the operand. For example, to exclude rows with blank values in an e-mail column, specify the following filter condition:

```
email NOT LIKE ''
```

Conversely, to display only rows where the e-mail column contains blank values, create the following condition:

```
email LIKE ''
```

In a report, you cannot distinguish between an empty value and a blank value in a string column. Both appear as missing values. If you want to filter all missing values whether they are null or blank, specify both filter conditions as follows:

```
email IS NOT NULL AND email NOT LIKE ''
```

Specifying a date as a comparison value

When you create a filter condition that compares the date-and-time values in a column to a specific date, the date value you supply must be in the following format regardless of your locale:

```
TIMESTAMP '2008-04-01 12:34:56'
```

Do not use locale-dependent formats such as 4/1/2008.

Specifying a number as a comparison value

When you create a filter condition that compares the numeric values in a column to a specific number, use a period (.) as the decimal separator regardless of your locale, for example:

```
123456.78
```

Do not use a comma (,).

Comparing to a string pattern

For a column that contains string data, you can create a filter condition that compares each value to a string pattern instead of to a specific value. For example, to display only customers whose names start with M, use the LIKE operator and specify the string pattern, M%, as shown in the following filter condition:

```
Customer LIKE 'M%'
```

You can also use the % character to ensure that the string pattern in the filter condition matches the string in the column even if the string in the column has trailing spaces. For example, use the filter condition:

```
Country LIKE 'USA%'
```

instead of the filter condition:

```
Country = 'USA'
```

The filter condition Country LIKE 'USA%' matches the following values:

```
'USA'
'USA  '
'USA    '
```

The filter condition Country = 'USA' matches only one value:

```
'USA'
```

You can use the following special characters in a string pattern:

- % matches zero or more characters. For example, %ace% matches any value that contains the string ace, such as Ace Corporation, Facebook, Kennedy Space Center, and MySpace.
- _ matches exactly one character. For example, t_n matches tan, ten, tin, and ton. It does not match teen or tn.

To match the percent sign (%) or the underscore character (_) in a string, precede those characters with a backslash character (\). For example, to match S_10, use the following string pattern:

```
S\_10
```

To match 50%, use the following string pattern:

```
50\%
```

Comparing to a value in another column

Use a filter condition to compare the values in one column with the values in another column. For example, in a report that displays products, sale prices, and MSRP (Manufacturer Suggested Retail Price), you can create a filter condition to compare the sale price and MSRP of each product, and display only rows where the sale price is greater than the MSRP.

How to compare to a value in another column

- 1 In Query Design, choose Filters.
- 2 On Query Design—Filters, choose New.
- 3 In Filter Conditions, in Filter by, select a column from the drop-down list.
- 4 Select the comparison test, or operator, to apply to the selected column.
- 5 In Value, select a column from the drop-down list. Figure 1-26 shows an example of a filter condition that compares the values in the priceEach column with the values in the MSRP column.

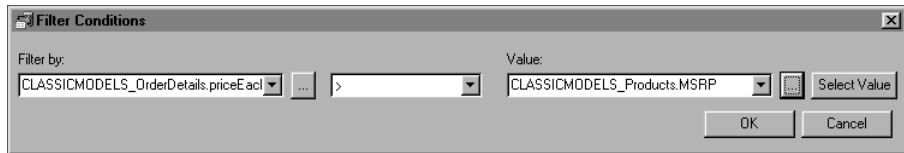


Figure 1-26 Comparing the values in priceEach with the values in MSRP
Choose OK.

Using an expression in a filter condition



An expression is any combination of Actuate SQL constants, operators, functions, and information object columns. When you create a filter condition, you can use an expression in Filter by, Value, or both. You create an expression with the expression builder.

For example, in an information object query that returns customer and order data, you want to see which orders shipped less than three days before the customer required them. You can use the DATEDIFF function to calculate the difference between the ship date and the required date:

```
DATEDIFF('d', shippedDate, requiredDate) < 3
```

Figure 1-27 shows this condition in Filter Conditions.

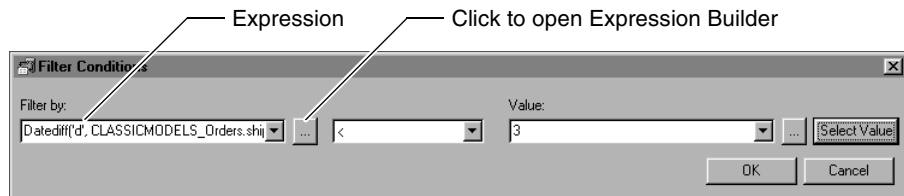


Figure 1-27 Filter Conditions with expression in Filter by

In an information object query that returns order data, you want to see which orders were placed today. You can use the CURRENT_DATE function to return today's date:

```
orderDate = CURRENT_DATE( )
```

Figure 1-28 shows this condition in Filter Conditions.

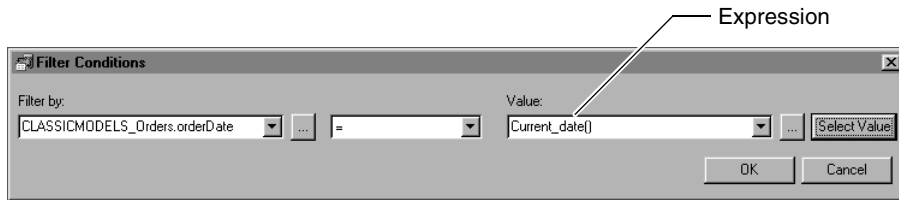


Figure 1-28 Filter Conditions with expression in Value

In an information object query that returns employee data, you want the information object query to return only data for the user who is currently logged in to the Encyclopedia volume. Use the LEFT function and the concatenation operator (||) to construct the employee's user name, and the CURRENT_USER function to return the name of the user who is currently logged in:

```
LEFT(firstName, 1) || lastName = CURRENT_USER( )
```

Figure 1-29 shows this condition in Filter Conditions.

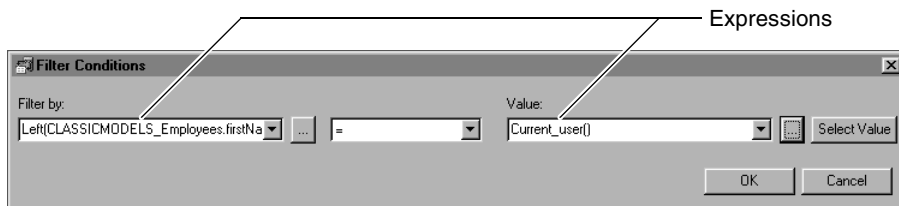


Figure 1-29 Filter Conditions with expressions in Filter by and Value

Creating multiple filter conditions

When you create a filter, you can define one or more filter conditions. Each condition you add narrows the scope of data further. For example, you can create a filter that returns rows where the customer's credit rank is either A or B and whose open orders total between \$250,000 and \$500,000. Each condition adds complexity to the filter. Design and test filters with multiple conditions carefully. If you create too many filter conditions, the information object query returns no data.

Adding a condition

You use Query Design—Filters, shown in Figure 1-23, to create one or more filter conditions. To create a filter condition, you choose New and complete the Filter Conditions dialog, shown in Figure 1-22. When you create multiple filter conditions, Information Object Query Builder precedes the second and subsequent conditions with the logical operator AND, for example:

```
SalesOffice LIKE 'San Francisco%' AND  
ProductLine LIKE 'Vintage Cars%'
```

This filter returns only data rows that meet both conditions. Sometimes, you want to create a filter to return data rows when either condition is true, or you want to create a more complex filter. To accomplish either task, use the buttons on the right side of Query Design—Filters, shown in Figure 1-30.

If you create more than two filter conditions and use different logical operators, use the parentheses buttons to group conditions to determine the order in which they are evaluated. Display the query output to verify the results.

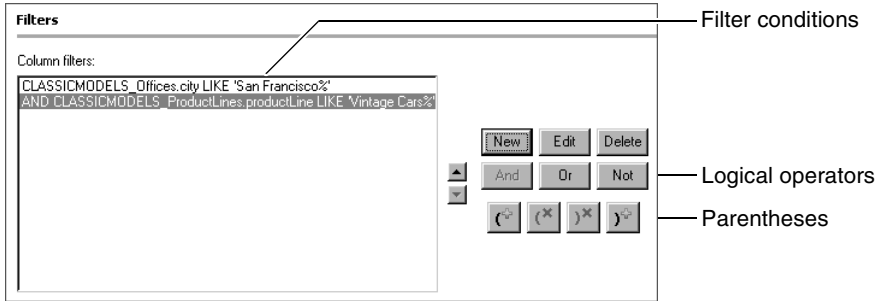


Figure 1-30 Query Design—Filters displaying two conditions

Selecting a logical operator

As you add each filter condition, the logical operator AND is inserted between each filter condition. You can change the operator to OR. The AND operator means both filter conditions must be true for a data row to be included in the report. The OR operator means only one condition has to be true for a data row to be included. You can also add the NOT operator to either the AND or OR operators to exclude a small set of data. For example, the following filter conditions return only sales data for classic car items sold by the Boston office:

```
SalesOffice LIKE 'Boston%' AND ProductLine LIKE 'Classic Cars%'
```

The following filter conditions return all sales data for the San Francisco and Boston offices:

```
SalesOffice LIKE 'San Francisco%' OR SalesOffice LIKE 'Boston%'
```

The following filter conditions return sales data for all product lines, except classic cars, sold by the San Francisco office:

```
SalesOffice LIKE 'San Francisco%' AND  
NOT (Product Line LIKE 'Classic Cars%')
```

Specifying the evaluation order



Information Object Query Builder evaluates filter conditions in the order in which they appear. You can change the order by selecting a filter condition in Query Design—Filters, shown in Figure 1-23, and moving it up or down using the arrow buttons. Filter conditions that you type in the Actuate SQL text box, shown in Figure 1-24, are preceded by AND and are evaluated last.

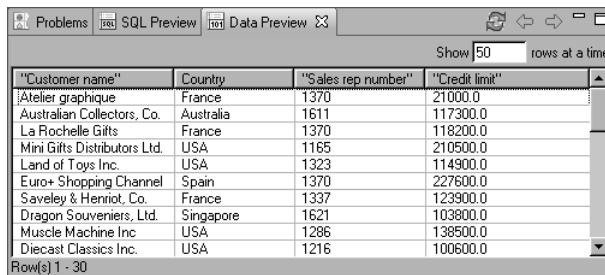
If you define more than two conditions, you can use parentheses to group conditions. For example, A AND B OR C is evaluated in that order, so A and B must be true or C must be true for a data row to be included. In A AND (B OR C), B OR C is evaluated first, so A must be true and B or C must be true for a data row to be included.

The following examples illustrate the difference a pair of parentheses makes.

The following filter contains three ungrouped conditions:

```
Country IN ('Australia', 'France', 'USA') AND
SalesRepNumber = 1370 OR CreditLimit >= 100000
```

Figure 1-31 shows the first 10 data rows returned by the query. Although the filter specifies the countries Australia, France, and USA and sales rep 1370, the data rows display data for other countries and sales reps. Without any grouped conditions, the filter includes rows that meet either conditions 1 and 2 or just condition 3.



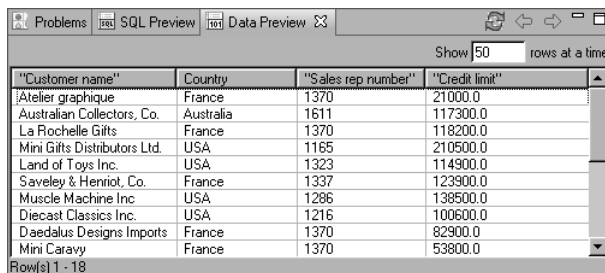
| "Customer name" | Country | "Sales rep number" | "Credit limit" |
|------------------------------|-----------|--------------------|----------------|
| Atelier graphique | France | 1370 | 21000.0 |
| Australian Collectors, Co. | Australia | 1611 | 117300.0 |
| La Rochelle Gifts | France | 1370 | 118200.0 |
| Mini Gifts Distributors Ltd. | USA | 1165 | 210500.0 |
| Land of Toys Inc. | USA | 1323 | 114900.0 |
| Euro+ Shopping Channel | Spain | 1370 | 227600.0 |
| Saveley & Henriot, Co. | France | 1337 | 123900.0 |
| Dragon Souvenirs, Ltd. | Singapore | 1621 | 103800.0 |
| Muscle Machine Inc. | USA | 1286 | 138500.0 |
| Diecast Classics Inc. | USA | 1216 | 100600.0 |

Figure 1-31 Results of a complex filter without parentheses grouping

The following filter contains the same three conditions, but this time the second and third conditions are grouped:

```
Country IN ('Australia', 'France', 'USA') AND
(SalesRepNumber = 1370 OR CreditLimit >= 100000)
```

Figure 1-32 shows the first 10 data rows returned by the query. The Country IN ('Australia', 'France', 'USA') condition must be true, then either the SalesRepNumber = 1370 condition or the CreditLimit >= 100000 condition is true.



| "Customer name" | Country | "Sales rep number" | "Credit limit" |
|------------------------------|-----------|--------------------|----------------|
| Atelier graphique | France | 1370 | 21000.0 |
| Australian Collectors, Co. | Australia | 1611 | 117300.0 |
| La Rochelle Gifts | France | 1370 | 118200.0 |
| Mini Gifts Distributors Ltd. | USA | 1165 | 210500.0 |
| Land of Toys Inc. | USA | 1323 | 114900.0 |
| Saveley & Henriot, Co. | France | 1337 | 123900.0 |
| Muscle Machine Inc. | USA | 1286 | 138500.0 |
| Diecast Classics Inc. | USA | 1216 | 100600.0 |
| Daedalus Designs Imports | France | 1370 | 82900.0 |
| Mini Caravy | France | 1370 | 53800.0 |

Figure 1-32 Results of a complex filter with parentheses grouping

Changing a condition

You can change any of the conditions in Query Design—Filters.

How to change a filter condition

- 1 In Query Design—Filters, shown in Figure 1-23, select the filter condition. Choose Edit.
- 2 In Filter Conditions, shown in Figure 1-22, modify the condition by changing the values in Filter by, Operator, or Value. Choose OK.

Deleting a condition

To delete a filter condition, in Query Design—Filters, select the condition. Then, choose Delete. Verify that the remaining filter conditions still make sense.

Prompting for filter values

You can prompt the report user for a single filter value or for multiple filter values. To prompt for a single value, create an Actuate SQL parameter. To prompt for multiple values, create a dynamic filter.

Prompting for a single value

Use an Actuate SQL parameter to prompt the report user for a single filter value. An Actuate SQL parameter enables the report user to restrict the data rows returned by the information object query without having to modify the WHERE clause. For example, for an information object query that returns sales data by sales office, instead of creating a filter that returns data for a specific office, you can create an Actuate SQL parameter called param_SalesOffice to prompt the report user to select an office. The WHERE clause is modified as follows:

```
WHERE SalesOffice LIKE :param_SalesOffice
```

You create Actuate SQL parameters and define their prompt properties on Query Design—Parameters. Prompt properties include the parameter's default value, a list of values for the user to choose from, and whether the parameter is required or optional. Parameters appear in the Value drop-down list in Filter Conditions with a : (colon) preceding the parameter name, as shown in Figure 1-33.

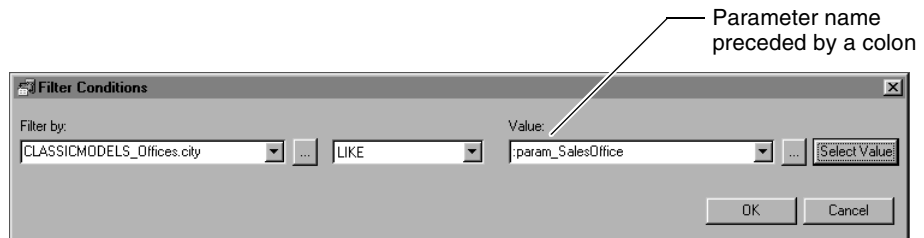


Figure 1-33 Filter Conditions with a parameter in the Value field

Do not use an Actuate SQL parameter in a filter condition with the IN operator, for example:

```
Country IN :param_Country
```

Actuate SQL parameters accept only a single value, but the IN operator takes multiple values. Instead, create a dynamic filter.

Prompting for multiple values

Use a dynamic filter to prompt the report user for multiple values. A dynamic filter can accept a single value, a list of values, or a range of values. For example, you want to prompt the report user for the location of one or more sales offices. You create a dynamic filter for the SalesOffice column. When the report user runs the report, they type the expression Boston | NYC to display data for the Boston and New York sales offices.

Data modelers can create one or more predefined filters when they design an information object. If an information object used in a query has a predefined filter, the predefined filter appears in the Information Object Query Builder as a dynamic filter.

Dynamic filters are also called ad hoc parameters. The syntax that the report user employs to provide a list of values or a range of values is called QBE syntax. For more information about ad hoc parameters and QBE syntax, see *Using Information Console*.

How to create a dynamic filter

- 1 In Query Design, choose Filters.
- 2 On Query Design—Filters, scroll down to see Select dynamic filters, as shown in Figure 1-34.

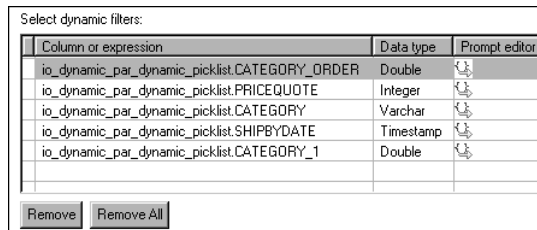


Figure 1-34 Selecting a dynamic filter



- 3 To create a new dynamic filter, click in the first blank row in the Column or expression column. Use the drop-down menu to add a column or choose Ellipsis to create an expression.
- 4 For each dynamic filter, you can perform the following steps:
 - 1 To set or change the data type, click the filter's row under Data type, and select a data type from the drop-down list.



- 2 To specify how the user is prompted when running the report, click the filter's row under Prompt Editor. This action displays Prompt editor, where you can change the prompt properties for the filter.

How to remove a dynamic filter

- 1 In Query Design, choose Filters.
- 2 On Query Design—Filters, complete the appropriate task:
 - To delete a single dynamic filter, select the filter in Select dynamic filters, and choose Remove.
 - To delete all dynamic filters, choose Remove All.

Setting dynamic filter prompt properties

When you create a dynamic filter, use the Prompt editor to specify the filter's display control type, list of values, and default value. You create a list of values by specifying the values or by typing an Actuate SQL query that retrieves the values. Information Object Query Builder does not validate the query. You can specify the filter values as well as the values displayed to the report user. If you type a query, the query must meet the following requirements:

- The query must retrieve one or two columns from an information object or map, for example:

```
SELECT DISTINCT CAST(custID AS VARCHAR(50)), customName
FROM "MyInformationObject.iob"
ORDER BY 2
```

The first column contains the filter values and must be of string data type. The second column contains the values displayed to the report user. The information object or map must reside in the same volume as the report executable. You must use an absolute path to reference the information object or map. If the information object or map defines a parameter, you must provide a value for the parameter, for example:

```
SELECT DISTINCT CAST(custID AS VARCHAR(50)), customName
FROM "MyInformationObject.iob" ['CA']
ORDER BY 2
```

- The query must not contain a WITH clause.

The filter values are interpreted as QBE expressions. Certain characters, for example, the comma (,) and the pipe sign (|), are interpreted as operators in a QBE expression. For example, the QBE expression:

16M x 1 Dynamic Ram, 3.3 volts

is interpreted as:

```
WHERE description LIKE '16M x 1 Dynamic Ram%'
OR description LIKE '3.3 volts%'
```

If you want these characters to be interpreted literally, enclose the strings in four single quotation marks (''') as shown in the following Actuate SQL queries:

- To match a string exactly:

```
SELECT ''' || description || '''
FROM "MyInformationObject.iob"
```

- To match a string using the LIKE operator:

```
SELECT ''' || description || '%'
FROM "MyInformationObject.iob"
```

|| is the concatenation operator.

The values returned by the query appear when a report user specifies a value for the dynamic filter when running the report in the Encyclopedia volume. The values do not appear when running the report in the report designer. In this environment, the prompt is a text box.

How to set the prompt properties of a dynamic filter

Setting the prompt properties of a dynamic filter affects how the report user sees the filter when running the report.

- 1 In Query Design, choose Filters.
- 2 In Query Design—Filters, scroll down to Select Dynamic Filters.
- 3 In the row for the filter, choose Prompt editor.
- 4 On Prompt editor, complete the following tasks, as shown in Figure 1-35:



Specify the prompt properties for this parameter. Prompt properties specify the behavior and appearance of filters that appear on requester pages for reports using this information object. Users can specify values for these filters to limit the data in a report.

Show as:

- ☐ Text box
- ☒ Drop-down list (read only)
- ☐ Combo box (editable)
- ☐ Radio buttons
- ☐ Dynamic list of values
- ☐ Auto suggest

Start Auto suggest after character(s)

Values:

Default value:

Select Values

| Value | Display name |
|-------|---------------|
| CA | California |
| MA | Massachusetts |
| NY | New York |
| PA | Pennsylvania |

Remove Remove All Sort Alphabetically

Reset OK Cancel

Select from a list of database values

Type values and display names

Figure 1-35 Typing values and display names for a filter

- In Show as, select the display control type. The choices available and appearance of the page depend on the display control type you select.
- If you choose a display control type other than Text box, you can specify a list of values for the user to choose from by typing the values and, optionally, the display names. Alternatively, you can select from a list of database values by choosing Select Values.
- In Default value, specify the default value. The default value can be a QBE expression.

To create an Actuate SQL query that retrieves the values, select Dynamic list of values, as shown in Figure 1-36, and type the query.

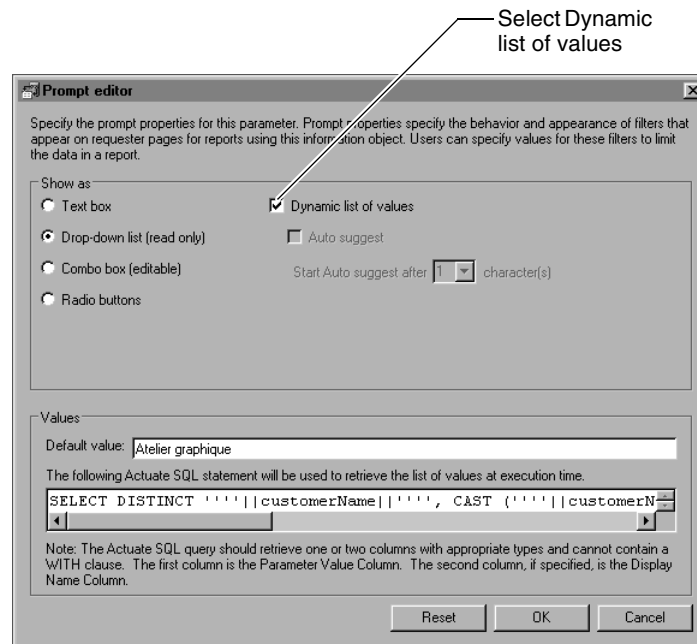


Figure 1-36 Creating an Actuate SQL query to generate values for a filter

If you select Combo box (editable), Dynamic list of values, and Auto suggest, a list appears after the report user types the number of characters specified in Start Auto suggest after N character(s). The list contains values that begin with the characters the user typed. For example, if the user typed 'Atel' and N=4, the list contains the value 'Atelier graphique'. In this case, the query that retrieves the values must select two columns, a value column and a display name column.

Choose OK.

Grouping data

A GROUP BY clause groups data by column value. For example, consider the following query:

```
SELECT orderNumber
FROM OrderDetails
```

The first 10 data rows returned by this query are as follows:

```
orderNumber
10100
10100
10100
10100
10101
10101
10101
10101
10102
10102
```

Each order number appears more than once. For example, order number 10100 appears four times. If you add a GROUP BY clause to the query, you can group the data by order number so that each order number appears only once:

```
SELECT orderNumber
FROM OrderDetails
GROUP BY orderNumber
```

The first 10 data rows returned by this query are as follows:

```
orderNumber
10100
10101
10102
10103
10104
10105
10106
10107
10108
10109
```

Typically, you use a GROUP BY clause to perform an aggregation. For example, the following query returns order numbers and order totals. The Total column is an aggregate column. An aggregate column is a computed column that uses an aggregate function such as AVG, COUNT, MAX, MIN, or SUM.

```
SELECT orderNumber, (SUM(quantityOrdered*priceEach)) AS Total
FROM OrderDetails
GROUP BY orderNumber
```

Figure 1-40 shows the first 10 data rows returned by the information object query. The data is grouped by order number and the total for each order appears.

Creating a GROUP BY clause

By default, Information Object Query Builder creates a GROUP BY clause automatically. If you prefer, you can create a GROUP BY clause manually.

Creating a GROUP BY clause automatically

When an information object query's SELECT clause includes an aggregate column and one or more non-aggregate columns, the non-aggregate columns must appear in the GROUP BY clause. If the non-aggregate columns do not appear in the GROUP BY clause, Information Object Query Builder displays an error message. For example, consider the following information object query:

```
SELECT orderNumber, (SUM(quantityOrdered * priceEach)) AS Total
FROM OrderDetails
```

When you attempt to compile the information object query, the error message shown in Figure 1-37 appears in the Problems view.

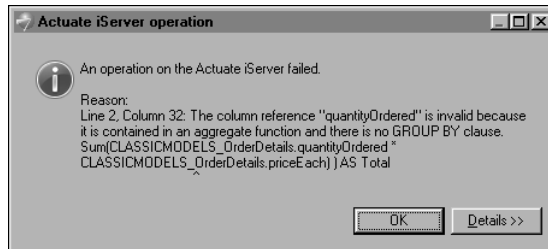


Figure 1-37 Information object query requires a GROUP BY clause

To avoid this problem, Information Object Query Builder automatically creates a GROUP BY clause:

```
SELECT orderNumber, (SUM(quantityOrdered * priceEach)) AS Total
FROM OrderDetails
GROUP BY orderNumber
```

If more than one column appears in the GROUP BY clause, you can change the order of the columns using the up and down arrows in Group By, as shown in Figure 1-38.

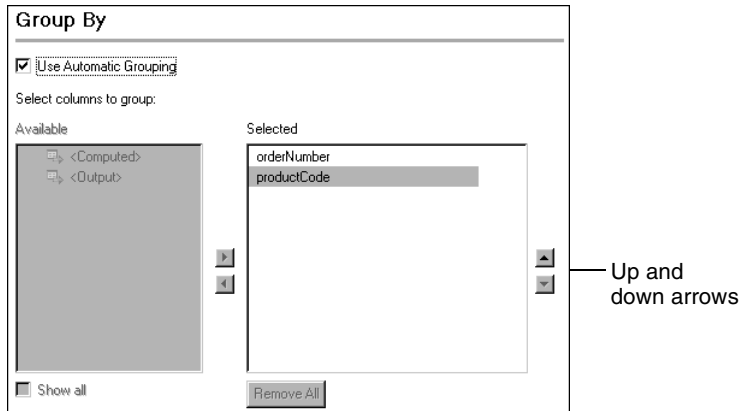


Figure 1-38 Changing the order of GROUP BY columns

Creating a GROUP BY clause manually

If automatic grouping does not generate the desired SQL query, create the GROUP BY clause manually. Create the GROUP BY clause manually if you want to group on a column that does not appear in the SELECT clause, for example:

```
SELECT (SUM(quantityOrdered * priceEach)) AS Total
FROM OrderDetails
GROUP BY orderNumber
```

How to create a GROUP BY clause manually

- 1 In Query Design, choose Group By.
- 2 In Query Design—Group By, deselect Use Automatic Grouping.
- 3 In Available, expand the Computed and Output nodes to view the available columns.

By default, Information Object Query Builder displays only output columns and non-aggregate computed fields. To group on a column that is not an output column, choose Show all.



- 4 In Available, select the appropriate column, and choose Select. This action moves the column name to Selected, as shown in Figure 1-39.



- 5 Repeat the previous step for each GROUP BY column.
- 6 To change the order of the GROUP BY columns, select a column in Selected, and use the up or down arrow.

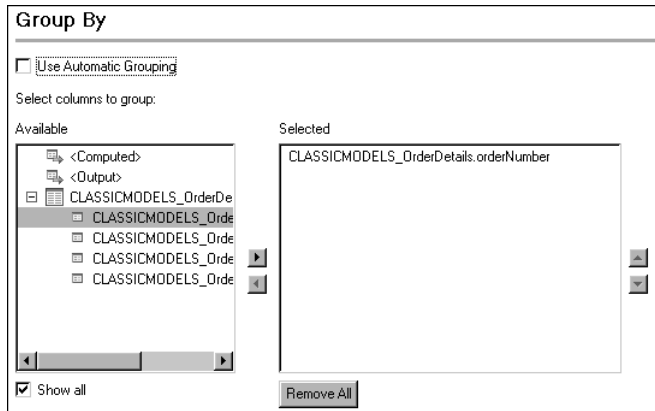


Figure 1-39 Selecting a GROUP BY column

Removing a column from the GROUP BY clause

By default, Information Object Query Builder removes GROUP BY columns automatically. If you disable automatic grouping, you must remove GROUP BY columns manually.

Removing a GROUP BY column automatically

Information Object Query Builder automatically removes a column from the GROUP BY clause when:

- You remove the column from the SELECT clause. For example, consider the following information object query:

```
SELECT orderNumber, productCode, (SUM(quantityOrdered *
    priceEach)) AS Total
FROM OrderDetails
GROUP BY orderNumber, productCode
```

You remove the productCode column from the SELECT clause. Information Object Query Builder automatically removes productCode from the GROUP BY clause:

```
SELECT orderNumber, (SUM(quantityOrdered * priceEach)) AS Total
FROM OrderDetails
GROUP BY orderNumber
```

- You manually add a column to the GROUP BY clause that does not appear in the SELECT clause and then enable automatic grouping. For example, consider the following information object query:

```
SELECT orderNumber, (SUM(quantityOrdered * priceEach)) AS Total
FROM OrderDetails
GROUP BY orderNumber, productCode
```

The `productCode` column appears in the `GROUP BY` clause but not in the `SELECT` clause. You enable automatic grouping. Information Object Query Builder automatically removes `productCode` from the `GROUP BY` clause:

```
SELECT orderNumber, (SUM(quantityOrdered * priceEach)) AS Total
FROM OrderDetails
GROUP BY orderNumber
```

Information Object Query Builder automatically removes the `GROUP BY` clause when:

- You remove all aggregate columns from the `SELECT` clause. For example, consider the following information object query:

```
SELECT orderNumber, (SUM(quantityOrdered * priceEach)) AS Total
FROM OrderDetails
GROUP BY orderNumber
```

You remove the aggregate column `SUM(quantityOrdered * priceEach)` from the `SELECT` clause. Information Object Query Builder automatically removes the `GROUP BY` clause:

```
SELECT orderNumber
FROM OrderDetails
```

- You remove all non-aggregate columns from the `SELECT` clause. For example, consider the following information object query:

```
SELECT orderNumber, (SUM(quantityOrdered * priceEach)) AS Total
FROM OrderDetails
GROUP BY orderNumber
```

You remove the `orderNumber` column from the `SELECT` clause. Information Object Query Builder automatically removes the `GROUP BY` clause:

```
SELECT (SUM(quantityOrdered * priceEach)) AS Total
FROM OrderDetails
```

Removing a `GROUP BY` column manually

If you disable automatic grouping, you must remove `GROUP BY` columns manually.

How to remove a `GROUP BY` column manually

- 1 In Query Design, choose Group By.
- 2 In Query Design—Group By, complete one of the following tasks:



- Select the column in Selected, and choose Deselect.
- To remove all Group By columns, choose Remove All.

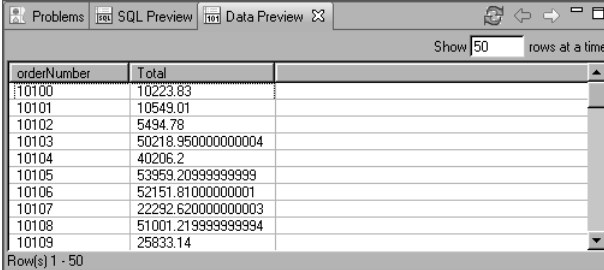
Filtering on an aggregate column

If an information object query includes a GROUP BY clause, you can restrict the data rows the query returns by adding a HAVING clause. The HAVING clause places a filter condition on one or more aggregate columns. An aggregate column is a computed column that uses an aggregate function such as AVG, COUNT, MAX, MIN, or SUM, for example SUM(quantityOrdered * priceEach).

For example, the following query returns order numbers and order totals. The Total column is an aggregate column. The data is grouped by order number and no filter condition is placed on the Total column.

```
SELECT orderNumber, (SUM(quantityOrdered * priceEach)) AS Total
FROM OrderDetails
GROUP BY orderNumber
```

Figure 1-40 shows the first 10 data rows returned by this information object query.



| orderNumber | Total |
|-------------|--------------------|
| 10100 | 10223.83 |
| 10101 | 10549.01 |
| 10102 | 5494.78 |
| 10103 | 50218.950000000004 |
| 10104 | 40206.2 |
| 10105 | 53959.209999999999 |
| 10106 | 52151.810000000001 |
| 10107 | 22232.620000000003 |
| 10108 | 51001.219999999994 |
| 10109 | 25833.14 |

Figure 1-40 Data rows returned by query with GROUP BY clause

You can add a HAVING clause to this information object query to place a filter condition on the Total column. The following information object query returns only rows for which the order total is greater than or equal to 50000:

```
SELECT orderNumber, (SUM(quantityOrdered * priceEach)) AS Total
FROM OrderDetails
GROUP BY orderNumber
HAVING SUM(quantityOrdered * priceEach) >= 50000
```

Figure 1-41 shows the first 10 data rows returned by this information object query.

The procedures for creating filter conditions for aggregate columns are identical to the procedures for creating filter conditions for other columns, except that you use Query Design—Having instead of Query Design—Filters. Filter conditions that you create using Query Design—Filters are evaluated before filter conditions that you create using Query Design—Having. In other words, filter conditions in the WHERE clause are applied before filter conditions in the HAVING clause.

| orderNumber | Total |
|-------------|--------------------|
| 10103 | 50218.950000000004 |
| 10105 | 53959.209999999999 |
| 10106 | 52151.810000000001 |
| 10108 | 51001.219999999994 |
| 10122 | 50824.659999999996 |
| 10126 | 57131.92 |
| 10127 | 58841.35 |
| 10135 | 55601.840000000004 |
| 10142 | 56052.560000000001 |
| 10145 | 50342.74 |

Figure 1-41 Data rows returned by query with GROUP BY and HAVING clauses

Defining parameters

An Actuate SQL parameter is a variable that is used in an information object query. When a report developer runs the report on the desktop, they provide a value for this variable. When a user runs the report in an Encyclopedia volume, the user provides a value for this variable on the Requester page in Information Console.

For example, the following Actuate SQL query uses the parameters lastname and firstname in the WHERE clause:

```
WITH ( lastname VARCHAR, firstname VARCHAR )
SELECT lname, fname, address, city, state, zip
FROM customerstable
WHERE (lname = :lastname) AND (fname = :firstname)
```

If an Actuate SQL query defines a parameter in a WITH clause but does not use the parameter, the query does not return any rows if no value is provided for the parameter when the report runs. For example, the following query does not return any rows if no values are provided for the lastname and firstname parameters when the report runs:

```
WITH ( lastname VARCHAR, firstname VARCHAR )
SELECT lname, fname, address, city, state, zip
FROM customerstable
```

How to define a parameter

- 1 In Query Design, choose Parameters.
- 2 In Query Design—Parameters, click the top empty line, and complete the following tasks:
 - In Parameter, type the name of the parameter. If a parameter name contains a special character, such as a period (.) or a space, enclose the name in double quotation marks (").
 - In Data type, select a data type from the drop-down list.
 - In Default value, type the default value:

- 

If Default value is a timestamp, it must be of the following form:

If Default value is a number, use a period (.) as the decimal separator, as shown in the following example:

NULL is not a valid parameter value. You cannot use a QBE expression.

- Parameters**
- Create a parameter by specifying a name, data type, and default value:
- | Parameter | Data type | Default value | Prompt editor |
|------------|-----------|---------------|---------------|
| paramState | Varchar | 'CA' | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
-
- Remove Remove All**

Figure 1-42 Choosing Prompt editor to specify a parameter's prompt properties

- ## How to delete a parameter

- Chapter 1, Using Information Object Query Builder 43

Specifying a parameter's prompt properties

Use the Prompt editor to specify a parameter prompt's properties, including display control type, list of values, and default value. You can specify the parameter values and, if desired, a corresponding set of display values that report users choose from. You create a list of values by typing the values or by typing an Actuate SQL query that retrieves the values.

The query must meet the following requirements:

- The query must retrieve one or two columns from an information object or map, as shown in the following example:

```
SELECT DISTINCT custID, customName
FROM "MyInformationObject.iob"
ORDER BY 2
```

The first column contains the parameter values. The second column contains the values that are displayed to the report user. The information object or map must reside in the same volume as the report executable. You must use an absolute path to reference the information object or map. If the information object or map defines a parameter, you must provide a value for the parameter, as shown in the following example:

```
SELECT DISTINCT custID, customName
FROM "MyInformationObject.iob" ['CA']
ORDER BY 2
```

- The first column's data type must match the parameter's data type.
- The query must not contain a WITH clause.

The query editor does not validate the query. The values returned by the query appear when a user specifies a value for the parameter. The values do not appear when a report developer specifies a value for the parameter on the desktop.

How to specify a parameter's prompt properties



- 1 Locate the appropriate parameter in Query Design—Parameters, and choose Prompt editor.
- 2 On the Prompt editor, in Show as, select the method of prompting the user, as shown in Figure 1-43. If you use a type of display other than text box, you can specify a list of values for the user to choose from.

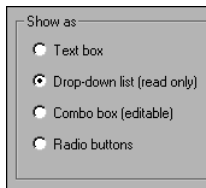


Figure 1-43 Selecting the method of prompting the user

You can create a list of values by typing the parameter values and, optionally, the display names, as shown in Figure 1-44. If you do not provide display names, the parameter values are displayed to the user.

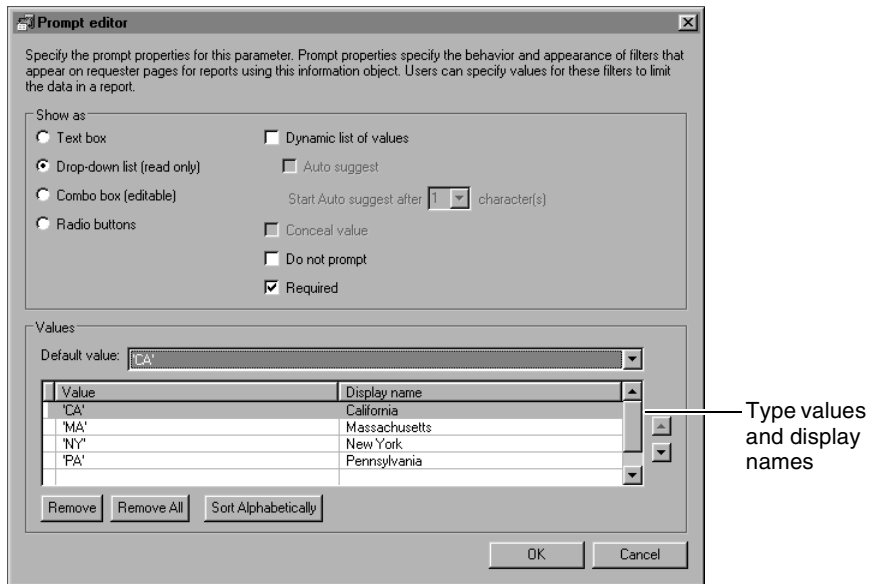


Figure 1-44 Typing a list of values and display names

You can create an Actuate SQL query that retrieves the parameter values or both the values and the corresponding display names. If the query has two columns, the values in the second column are used as the display names. To use a query to create the list of values, select Dynamic list of values, as shown in Figure 1-45, and type the query.

If you select Combo box (editable), Dynamic list of values, and Auto suggest, a drop-down list appears after the report user types the number of characters specified in Start Auto suggest after N character(s). The list contains values that begin with the characters the user typed. For example, if the user typed 'Atel and N=4, the list contains the value 'Atelier graphique'. In this case, the query that retrieves the values must select two columns, a value column and a display name column.

- 3 In Default value, specify the default value.
- 4 You also can specify values for the following additional properties:
 - Conceal value
 - Do not prompt
 - Required

When you finish specifying the property values for the prompt, choose OK.

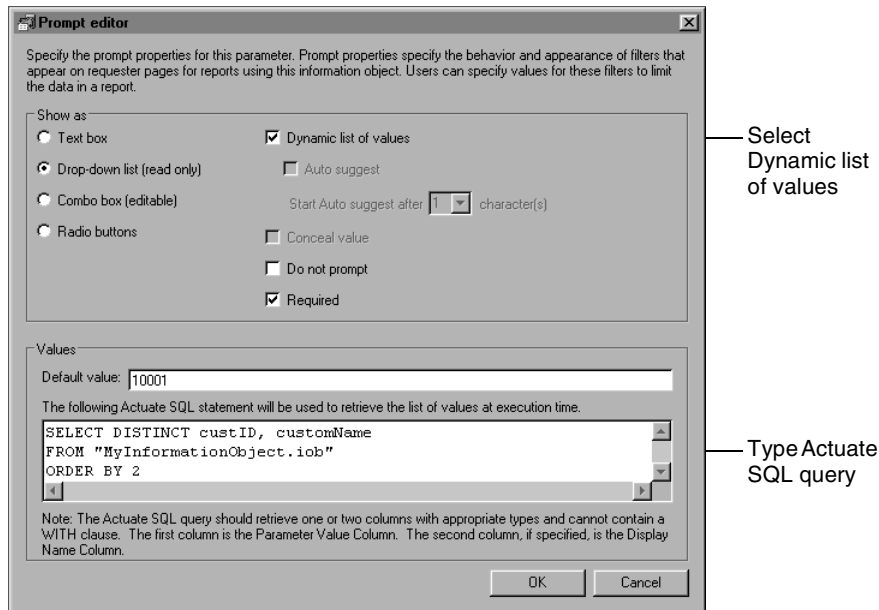


Figure 1-45 Specifying an Actuate SQL query to provide a dynamic list of values

Setting parameter properties

Table 1-6 lists parameter properties and provides a description of each property.

Table 1-6 Parameter properties

| Parameter property | Can set? | Description |
|--------------------|----------------------------|--|
| Conceal Value | Yes, in the Prompt editor | Visibility of the value that the user provides for this parameter. To conceal the value, set to True. To display the value, set to False. This parameter property applies only to parameters with the varchar data type and the text box display type. |
| Data Type | Yes, on the Parameters tab | Parameter's data type. |
| Default Value | Yes, in the Prompt editor | Parameter's default value. If a parameter does not have a default value, and the Required property is set to False, the parameter takes one of the following values if the user does not provide a value: |

Table 1-6 Parameter properties

| Parameter property | Can set? | Description |
|------------------------------|--|---|
| Default Value (continued) | Yes, in the Prompt editor (continued) | <ul style="list-style-type: none"> ■ 0 if the parameter is of type decimal, double, or integer. ■ Empty string if the parameter is of the varchar data type. ■ Current date and time if the parameter is of the timestamp data type. |
| Description | Not used | Not used. |
| Display Control Type | Yes, in the Prompt editor | Control type for this parameter. The options are text box, read-only drop-down list, editable drop-down list, or radio buttons. |
| Display Format | Not used | Not used. |
| Display Length | Not used | Not used. |
| Display Name | Not used | Not used. |
| Do Not Prompt | Yes, in the Prompt editor | Visibility of the parameter to the user. To hide this parameter, set to True. To display the parameter, set to False. |
| Heading | Not used | Not used. |
| Help Text | Not used | Not used. |
| Horizontal Alignment | Not used | Not used. |
| Name | Yes, on the Parameters tab | Parameter name. |
| Parameter Mode | Yes | Setting for parameters that are in stored procedures and ODA data source queries to specify the input or output type of the parameter. The options are Input, Output, InputAndOutput, or ReturnValue. ReturnValue is used only for stored procedures and is equivalent to Output. |
| Required | Yes, in the Prompt editor | Indicator of whether the parameter is required. To require a value for this parameter, set to True. Otherwise, set to False. |
| Size | Yes | The size of the parameter if the parameter data type is varchar. Otherwise, not used. Must be set if size is greater than 1300. |

Setting source parameters

A source parameter is a parameter that is defined in an information object from which you are building an information object query. If a query contains an information object with a parameter, Query Design—Parameters has a Source parameter field.

You can set a source parameter to one of the following types of values:

- A single scalar value
- A local parameter in the information object query that you are creating

You cannot set a source parameter to a column reference, such as `ORDERS.ORDERID`, or an Actuate SQL expression.

When you set a source parameter to a local parameter, you can indicate that the local parameter inherits the values of its prompt properties from the source parameter. The available prompt properties are Conceal Value, Default Value, Display Control Type, Do Not Prompt, and Required. If you specify that the local parameter inherits its prompt property values from the source parameter, and prompt property values for the source parameter change, the changes are propagated to the local parameter. For example, if the display control type for the source parameter changes from text box to read-only drop-down list, the display control type for the local parameter also changes from text box to read-only drop-down list.

If you change a prompt property value for a local parameter, its prompt property values are no longer inherited from the source parameter. For example, if you change the display control type for the local parameter to editable drop-down list, and the display control type for the source parameter later changes to text box, the change is not propagated to the local parameter. To reinstate inheritance, choose Reset in the Prompt editor. Choosing Reset returns all property values in the local parameter to inherited values, and the local parameter inherits any future changes to property values in the source parameter.

To set source parameters, use Query Design—Parameters. To define a local parameter and set a source parameter to the local parameter in one step, drag the source parameter from Source parameter, and drop it in Parameter, as shown in Figure 1-46.

How to set a source parameter

- 1 In Query Design, choose Parameters.
- 2 On Parameters, complete the following tasks:
 - In Source parameter, select the appropriate parameter.
 - In Value, complete one of the following tasks:

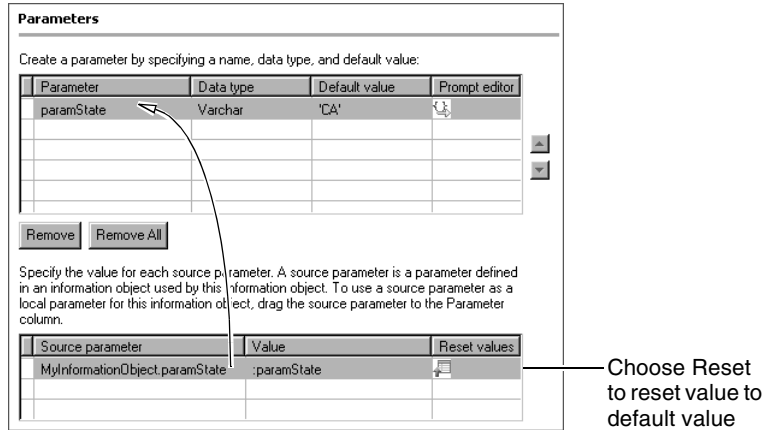


Figure 1-46 Setting a source parameter to a local parameter

- ❑ Choose a parameter from the drop-down list. The drop-down list contains the local parameters for the information object query you are building.
- ❑ Type a value, as shown in Figure 1-47:
 - ❑ If Value is a string, enclose the string in single quotation marks, as shown in the following example:
 'New York City'
 - ❑ If Value is a timestamp, it must be in the following form:
 TIMESTAMP '2001-02-03 12:11:10'
 - ❑ If Value is a number, use a period (.) as the decimal separator, as shown in the following example:
 123456.78
 - ❑ If Value is a parameter, precede the parameter name with a colon (:).

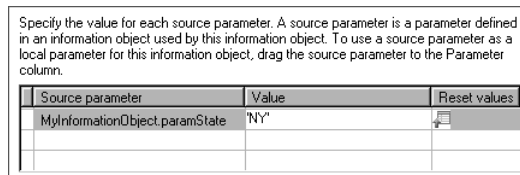


Figure 1-47 Providing a value for a source parameter

Synchronizing source parameters



You must synchronize source parameters when parameters in a source information object are added, removed, or reordered, or their data types or other

properties change. To synchronize source parameters, choose Synchronize in the upper pane of Query Design, as shown in Figure 1-48. Synchronizing source parameters refreshes the list of source parameters on Query Design—Parameters.

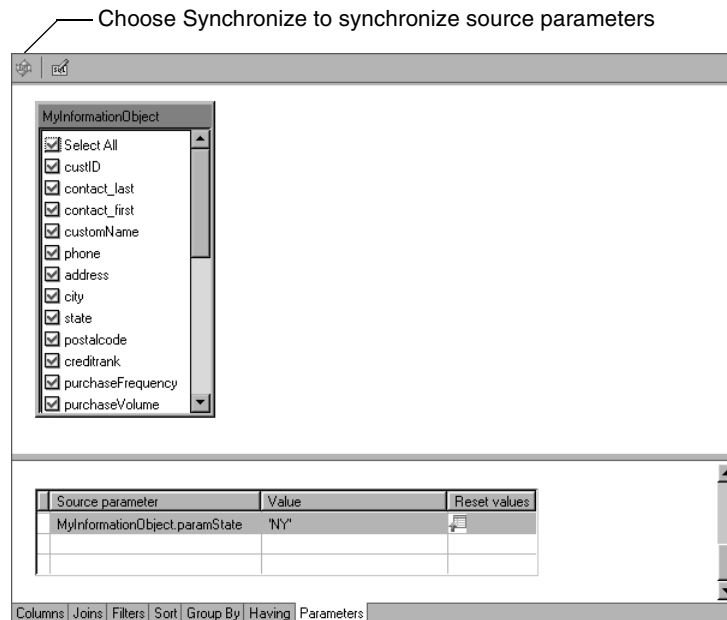


Figure 1-48 Synchronizing source parameters

Creating a textual information object query

Use the Actuate SQL text editor if either of the following conditions is true:

- Information Object Query Builder Advanced Design does not generate the desired Actuate SQL query, so you must edit the query. For example, if the query includes OR or UNION, you must use the Actuate SQL text editor to edit the query.
- You want to type or paste an Actuate SQL query instead of creating it graphically.

If you save a query in the Actuate SQL text editor, you cannot modify the query in the graphical editor.

To display the Actuate SQL text editor, complete one of the following tasks:



- In Query Design, choose SQL Editor, as shown in Figure 1-49.

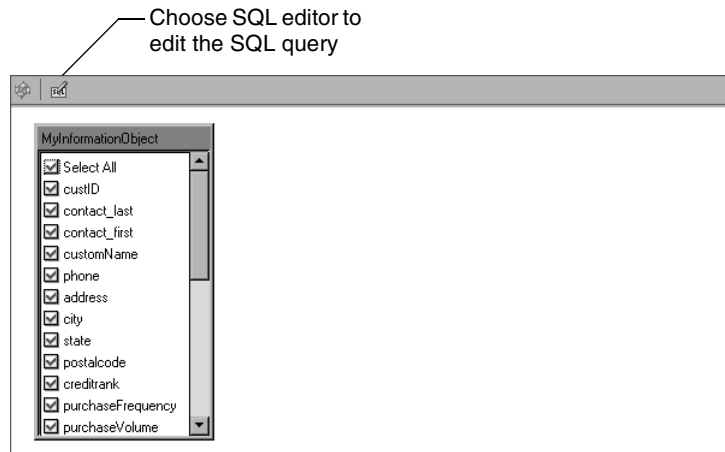


Figure 1-49 Choosing SQL editor to edit an Actuate SQL query

- On SQL Preview, choose Edit SQL.

Figure 1-50 shows the Actuate SQL text editor.

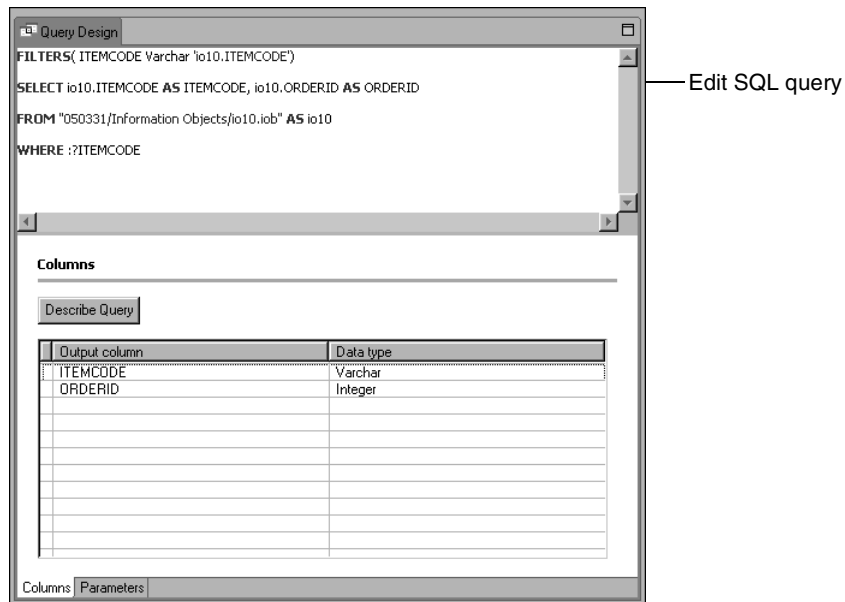


Figure 1-50 Editing the Actuate SQL query in the text editor

You edit the query in the upper pane of the Actuate SQL text editor. The lower pane displays output columns or parameters.

When you edit a query in the SQL text editor, do not use table and column aliases that are identical except for case. For example, do not use both status and STATUS as column aliases.

Use absolute paths in the Information Object Query Builder because the report executable and information object may not be in the same Encyclopedia volume.

Displaying output columns

In SQL Text Editor—Columns, to display the query's output columns and the data type for each column, choose Describe Query, as shown in Figure 1-51.

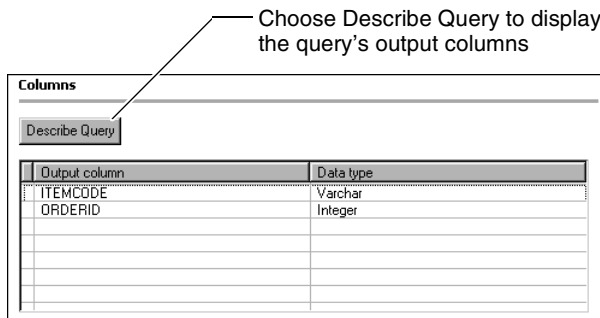


Figure 1-51 Using Describe Query to display the query's output columns

To specify column property values, select the column and specify the property values in Properties.

Displaying parameters

On SQL Text Editor—Parameters, choose Describe Query to display the query's parameters and the data type for each parameter. You can type a default value for a parameter in Default value, as shown in Figure 1-52.

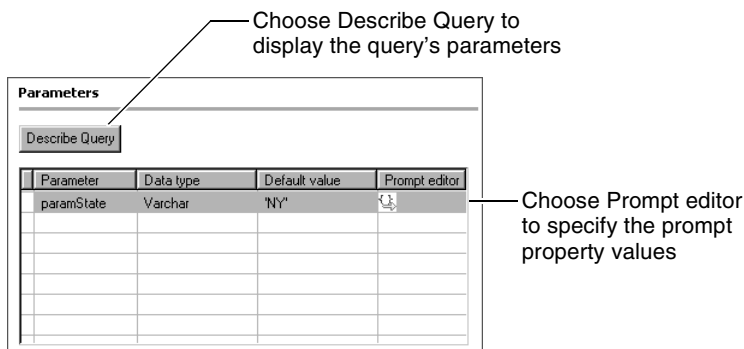


Figure 1-52 Using Describe Query to display the query's parameters

You can choose Prompt Editor to set the prompt property values.

Displaying information object query output

You can display information object query output in Data Preview.

How to display information object query output

- 1 Choose Data Preview.
- 2 In Data Preview, choose Refresh. Parameter Values, shown in Figure 1-53, appears if the information object query defines parameters.

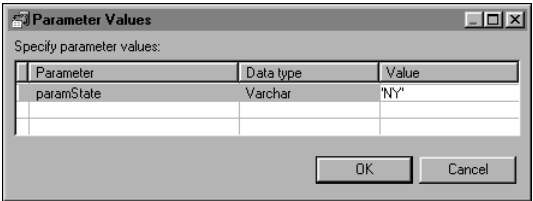


Figure 1-53 Specifying parameter values

- 3 On Parameter Values, type the parameter values. A parameter value must be a single value, not a list of values. Choose OK. Information object query output appears, as shown in Figure 1-54.

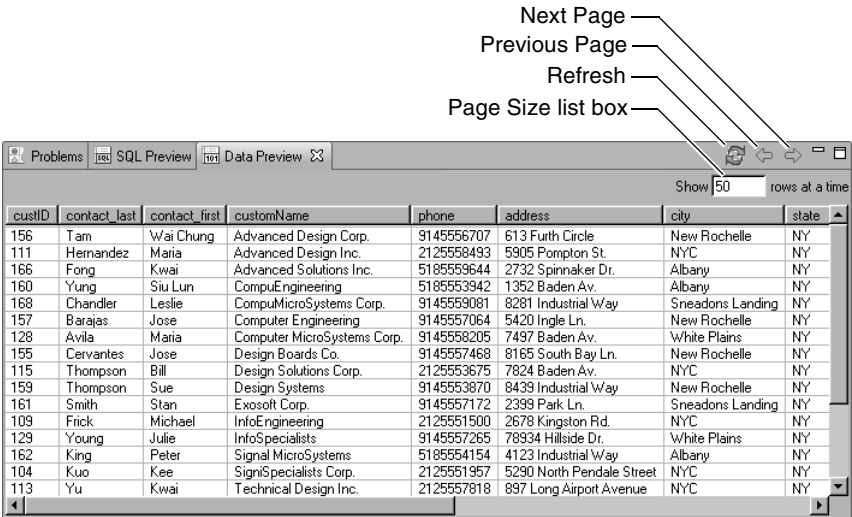


Figure 1-54 Viewing the information object query's output

- 4 Use the scroll bars to view all columns and displayed rows. Use the Page Size list box to change the number of rows displayed on each page. Use the Next Page and Previous Page icons to navigate through the data preview one page at a time.

Actuate SQL reference

This chapter contains the following topics:

- About Actuate SQL
- Differences between Actuate SQL and ANSI SQL-92
- Actuate SQL syntax
- Data types and data type casting
- Functions and operators
- Providing query optimization hints
- Using pragmas to tune a query

About Actuate SQL

An information object encapsulates an Actuate SQL query. You can create the Actuate SQL query that defines an information object in Information Object Designer by typing the Actuate SQL query in the textual query editor or by specifying the desired query characteristics in the graphical query editor. If you use the graphical query editor, you can view the resulting Actuate SQL query in SQL Preview.

If you already have one or more existing information objects, you can access the information object data by specifying a query on the information object using a report designer's Information Object Query Builder. You can create the query on the information object by typing a Actuate SQL query in the textual query editor or by specifying the desired query characteristics in a graphical query editor. If you use the graphical query editor, you can view the resulting Actuate SQL query in SQL Preview.

A query that defines an information object and a query on an information object both use Actuate SQL. Actuate SQL is based on the ANSI SQL-92 standard. This chapter describes the differences between Actuate SQL and ANSI SQL-92. This chapter also describes the FILTERS statement that you can use when creating a query from Information Object Query Builder in a report designer.

Differences between Actuate SQL and ANSI SQL-92

Actuate SQL is based on ANSI SQL-92. This section provides an overview of the differences between Actuate SQL and ANSI SQL-92. This section also provides an overview of the FILTERS statement that is available from report designers. Report designers support using the FILTERS statement with Actuate SQL to dynamically filter SELECT statements.

Limitations compared to ANSI SQL-92

Actuate SQL has the following limitations compared to ANSI SQL-92:

- Only the SELECT statement is supported.
- Data types are limited to integers, strings, timestamps, floating point numbers, and decimals.
- Intersection and set difference operations are not available.
UNION and UNION DISTINCT are not supported. UNION ALL is supported. To obtain the same results as a UNION DISTINCT operation, perform a UNION ALL operation followed by a SELECT DISTINCT operation. For example, IO3 performs a UNION ALL operation on IO1 and IO2:


```

SELECT empNo, eName
FROM "IO1.iob" AS IO1
UNION ALL
SELECT empNo, eName
FROM "IO2.iob" AS IO2

```

To obtain distinct results from IO3, create IO4, which performs a SELECT DISTINCT operation on IO3:

```

SELECT DISTINCT empNo, eName
FROM "IO3.iob" AS IO3

```

- LIKE operator patterns and escape characters must be literal strings, parameters, or expressions. The LIKE operator does not support column references, subqueries, or aggregate functions, for example, MAX and AVG.
- Unnamed parameters are not supported.
- Some subqueries are not supported.
- Not all ANSI SQL-92 functions and operators are available.

Extensions to ANSI SQL-92

Actuate SQL has the following extensions to ANSI SQL-92:

- Parameterized queries with named parameters
A parameterized query starts with a WITH clause that specifies the names and types of parameters that the query uses. The following example shows using parameters to specify returning rows where salesTotal is within a range specified by two parameters:

```

WITH ( minTotal DECIMAL, maxTotal DECIMAL )
SELECT o.id, o.date
FROM "/sales/orders.sma" o
WHERE o.salesTotal BETWEEN :minTotal AND :maxTotal

```

A query with a parameterized SELECT statement is typed and is subject to the same casting rules as a function call, except that parameter declarations specify the maximum scale, precision, and length of parameter values. All parameter values are required. A parameter value must be a literal, for example 'abc', NULL, a parameter reference, or an Actuate SQL expression. A parameter value cannot be a column reference, for example ORDERS.ORDERID.

- Parameterized table, view, and query references
A parameterized table or view reference in a query enables specification of the query without knowing the table or view until run time. At run time, the values of the parameters specify the table. In the following example, the table is specified by the IOB name and the team and position parameters:

```

WITH( team VARCHAR, position VARCHAR, minGamesPlayed
      INTEGER )
SELECT playername
FROM "/sports/baseball/japan/players.iob" [:team,:position]
WHERE GamesPlayed > :minGamesPlayed

```

Parameter passing is typed and is subject to the same casting rules as a function call.

- **Scalar subqueries**

A scalar subquery is a query that returns a scalar value that is used in a second query. For example, the following query returns a scalar value:

```

SELECT SUM(B.Quantity * B.UnitPrice)
FROM "Order Detail.sma" AS B

```

This second query uses the previous query as a scalar subquery, evaluating the result of the scalar subquery and checking if the result is greater than 1000:

```

SELECT CustomerID
FROM "Customers.sma" C
LEFT OUTER JOIN
"Orders.sma" O
ON ( C.CustomerID=O.CustomerID )
WHERE
( SELECT SUM(B.Quantity * B.UnitPrice)
FROM "Order Detail.sma" AS B
) > 1000

```

- **Join control syntax specifying the join algorithm**

In Actuate SQL, you can specify the algorithm to use for joins. There are three join algorithms in Actuate SQL:

- **Dependent join**

A dependent join specifies obtaining all the results for the left side of the join and then using each resulting row to process the right side of the join. This algorithm is especially efficient when the left side of the join does not return many rows and the data source of the right side can handle evaluating the join criteria.

- **Nested loop join**

A nested loop join specifies obtaining and storing in memory all the results for the right side of the join. Then, for each row resulting from the left side, a nested loop join evaluates the right side results for matches to the join criteria. This algorithm is especially efficient when the right side of the join does not return many rows and the join expression cannot be delegated to the data source of the right side.

- Merge join

A merge join specifies obtaining the results for the right and left sides of the join and comparing these results row by row. Merge joins are applicable only for joins where the value on the left must be equal to the value on the right. This algorithm uses less memory than a nested loop join. This algorithm is especially efficient if the data sources sort the rows but presorting is not required.

The following example shows a merge join in a simple SELECT statement:

```
SELECT customers.custid, customers.customname,  
       customers.city, salesreps.lastname, salesreps.email  
FROM customers MERGE JOIN salesreps  
ON customers.repid = salesreps.repid
```

The following example shows a dependent join in a parameterized SELECT statement:

```
WITH ( minRating INTEGER )  
SELECT c.name, o.date, o.shippingStatus  
FROM  
    "/uk/customers.sma" c  
DEPENDENT JOIN  
    "/sales/orders.sma" o  
ON c.id = o.custId  
WHERE c.rating >= :minRating
```

You can also specify whether the join is an inner join or left outer join. The following example shows a SELECT statement with a left outer join:

```
SELECT customers.custid, customers.contact_last,  
       customers.contact_first, salesreps.lastname,  
       salesreps.firstname  
FROM salesreps LEFT OUTER JOIN customers  
ON salesreps.firstname = customers.contact_first
```

For information about inner and outer joins, see the SQL reference guide for your database.

- Pragmas to alter query semantics
- Additional functions
- The ability to have ORDER BY items other than SELECT items or aliases, for example:

```
SELECT customers.contact_first || ' ' ||  
       customers.contact_last  
       "MOST_VALUED_CUSTOMERS"  
FROM "/customers.sma" customers  
WHERE customers.purchasevolume > 3  
ORDER BY customers.purchasevolume DESC
```

If an ORDER BY item is not a SELECT item or an alias, it must be a grouping column if a GROUP BY clause is present. ORDER BY items must be SELECT items if SELECT DISTINCT is specified.

Use ORDER BY only when creating a query in a report designer. Do not use ORDER BY when you create an information object in Information Object Designer.

- The ability to have GROUP BY items that are expressions, for example:

```
SELECT DATEPART('yyyy', orders.shipbydate) "YEAR",
       DATEPART('m', orders.shipbydate) "MONTH",
       COUNT(*) "NUM_ORDERS"
FROM "/orders.sma" orders
GROUP BY DATEPART('yyyy', orders.shipbydate),
         DATEPART('m', orders.shipbydate)
```

To use an expression as a GROUP BY item, the expression must appear as a SELECT item. Aggregate functions are not allowed in GROUP BY expressions unless they are outer references from a subquery and the subquery is contained in the HAVING clause of the parent query. Complex GROUP BY expressions cannot be used in the HAVING clause of the query.

- The ability to use references to aliases

Database limitations

Because the Integration service delegates many of its operations to the databases, these operations are affected by database limitations, such as the maximum precision of decimal types or the treatment of zero-length strings.

FILTERS statement in report designers

In addition to Actuate SQL's extensions to ANSI SQL-92, report designers support using a FILTERS statement with Actuate SQL to dynamically filter SELECT statements. A dynamically filtered SELECT statement enables the user to specify additional filters in the WHERE clause or HAVING clause when running a SELECT statement or a parameterized SELECT statement. The FILTERS statement specifies one or more dynamic filters, their data types, and the beginning of each filter. The user completes conditions using operators, constants, and column names:

```
FILTERS ( filter1 Integer 'o.salesRepID' , filter2 Varchar
         'o.territory = ')
WITH ( minTotal Decimal, maxTotal Decimal )
SELECT o.id, o.date
FROM "/sales/orders.sma" o
WHERE o.salesTotal BETWEEN :minTotal AND :maxTotal AND :?filter1
      AND :?filter2
```

Information Object Designer does not support use of the FILTERS statement.

Actuate SQL syntax

Actuate SQL syntax is similar to SQL-92 syntax. Actuate SQL has additional syntax for naming tables and columns. Table 2-1 provides a description of the typographical conventions used in describing Actuate SQL grammar.

Table 2-1 Typographical conventions used in describing Actuate SQL grammar

| Convention | Used for... |
|---------------------------------------|---|
| NORMAL UPPERCASE | Actuate SQL keywords. |
| <i>ITALICIZED</i> <i>UPPERCASE</i> | Tokens. |
| (vertical bar) | Separating syntax items. You choose one of the items. |
| [] (brackets) | Optional syntax items. Do not type the brackets. |
| { } (braces) | Required syntax items. Do not type the braces. |
| [...n] | Indicating that the preceding item can be repeated n number of times. The item occurrences are separated by commas. |
| [...n] | Indicating that the preceding item can be repeated n number of times. The item occurrences are separated by blanks. |
| <label> | The name for a block of syntax. This convention is used to label syntax that can appear in more than one place within a statement. Each location in which the block of syntax can appear is shown with the label enclosed in chevrons, for example <label>. |

Table 2-2 lists the tokens used in the Actuate SQL grammar.

Table 2-2 Tokens used in describing the Actuate SQL grammar

| Token | Definition |
|------------|--|
| IDENTIFIER | A sequence of Unicode letters, digits, dollar signs, and underscores combining characters and extenders. The first character must be a letter. |

(continues)

Table 2-2 Tokens used in describing the Actuate SQL grammar (continued)

| Token | Definition |
|-------------------------------|---|
| IDENTIFIER <i>(continued)</i> | Use double quotes to quote identifiers. To represent a double quote within a quoted identifier, use two double quotes. Quoted identifiers can include any characters except carriage return or new line. |
| CHAR_LITERAL | Any Unicode text between single quotes other than carriage return or new line. To represent a single quote, use two single quotes. Multiple consecutive character literals are concatenated. |
| DECIMAL_LITERAL | An integer literal followed by a decimal point and an optional integer representing the fractional part. Syntax: (INTEGER_LITERAL .) (. INTEGER_LITERAL) (INTEGER_LITERAL. [INTEGER_LITERAL]) |
| DOUBLE_LITERAL | A number of the form 1.2E+3. If the sign is omitted, the default is positive. Syntax: ((. INTEGER_LITERAL) (INTEGER_LITERAL. [INTEGER_LITERAL])) [(e E) [- +] INTEGER_LITERAL] |
| INTEGER_LITERAL | One or more consecutive digits. |
| TIMESTAMP_STRING | A literal string that is interpreted as a timestamp value, such as '2002-03-31 13:56:02.7'. Years are 4 digits. Seconds are 2 digits with an optional fraction up to 3 digits. All other fields are 2 digits. The space between the date and time sections is required. Format: 'yyyy-mm-dd hh:mm:ss.msec' |

Actuate SQL grammar

The Actuate SQL grammar contains one statement. The syntax of this statement is:

```
[<Pragma>] [...n] [<QueryParameterDeclaration>] <SelectStatement>
```

Report designers also use a FILTERS statement that incorporates Actuate SQL. Information Object Designer does not support use of the FILTERS statement. The syntax for the FILTERS statement is:

```
<FilterClause> <QueryParameterDeclaration> <SelectStatement>
```

Table 2-3 provides the syntax for the grammar parts used in these statements.

Table 2-3 Syntax for the Actuate SQL grammar parts

| Grammar part | Syntax |
|-------------------------|--|
| AdditiveExpression | <MultiplicativeExpression> {(+ -) <MultiplicativeExpression>} [...n] |
| AdHocParameter | :? <i>IDENTIFIER</i> Use AdHocParameter only in a FILTERS statement, which is available only from a report designer. AdHocParameter cannot be used in a WITH clause. |
| AggrExpression | COUNT (([ALL DISTINCT] <ValueExpression> *)) (AVG MAX MIN SUM) ([ALL DISTINCT] <ValueExpression>) |
| AndExpression | {<UnaryLogicalExpression>} [AND...n] |
| CardinalityType | 1 ? * + |
| CaseExpression | CASE [<ValueExpression>] {<WhenClause>} [...n] [ELSE <ValueExpression>] END |
| CastExpression | CAST((<ValueExpression> NULL) AS <ScalarDataType>) |
| ColumnAlias | IDENTIFIER |
| CondExpr | {<AndExpression>} [OR...n] |
| ConditionalPrimary | (<CondExpr>) <SimpleCondition> <AdHocParameter> Use AdHocParameter only in a FILTERS statement. |
| DataType | <ScalarDataType> |
| ExplicitInnerOuterType | LEFT [OUTER] INNER |
| ExplicitJoinType | MERGE NL DEPENDENT |
| ExpressionList | {<ValueExpression>} [...n] |
| FilterClause | FILTERS(<i>IDENTIFIER</i> DataType 'ValueExpression' [...n]) Use FILTERS only from a report designer. |
| FromClause | {FROM <FromTableReference>} [...n] |
| FromTableName | IDENTIFIER [((<TableParameters>)] [[AS] IDENTIFIER] If the identifier is not enclosed in quotes, it is interpreted as a table. If the identifier is enclosed in quotes, it is interpreted as an absolute or relative path in the Encyclopedia volume. |
| FromTableReference | <JoinExpression> (<JoinExpression>) <FromTableName> |
| FunctionCallOrColumnRef | IDENTIFIER (([<ExpressionList>]) [. IDENTIFIER]) |

(continues)

Table 2-3 Syntax for the Actuate SQL grammar parts (continued)

| Grammar part | Syntax |
|---------------------------|--|
| GroupByClause | GROUP BY {<ValueExpression>} [...n] ValueExpression can be an expression as long as the expression also appears as a SELECT item. |
| HavingClause | HAVING <CondExpr> |
| JoinCondition | ON <CondExpr> [{CARDINALITY('<CardinalityType> - <CardinalityType>')}] |
| JoinElement | (<JoinExpression>) <FromTableName> |
| JoinExpression | <JoinElement> {<JoinSpec><JoinElement> [<JoinCondition>]} [...n] |
| JoinSpec | [[[LEFT RIGHT] OPTIONAL] <ExplicitInnerOuterType>] [<ExplicitJoinType>] JOIN |
| Length | INTEGER_LITERAL |
| MultiplicativeExpression | <UnaryExpression> {(* /) UnaryExpression} [...n] |
| NamedParameter | <i>IDENTIFIER</i> |
| OrderByClause | ORDER BY {<ValueExpression> (ASC DESC)? } [...n] ValueExpression is not limited to SELECT items or aliases. If ValueExpression is not a SELECT item or an alias, it must be a grouping column if a GroupByClause is present. Use ORDER BY only when creating a query in a report designer. Do not use ORDER BY when you create an information object in Information Object Designer. |
| ParameterDeclaration | <i>IDENTIFIER</i> [<AS>] <Data Type> |
| ParamPlaceholder | <NamedParameter> |
| Pragma | PRAGMA <i>IDENTIFIER</i> := <i>CHAR_LITERAL</i> |
| Precision | INTEGER_LITERAL |
| PrimaryExpression | <FunctionCallOrColumnRef> <ParamPlaceholder> <UnsignedLiteral> <AggrExpression> (<ValueExpression>) <CastExpression> |
| QueryParameterDeclaration | WITH ({<ParameterDeclaration>} [...n]) All parameters are required. |
| RelationalOperator | = <> <=> >=> |

Table 2-3 Syntax for the Actuate SQL grammar parts (continued)

| Grammar part | Syntax |
|--------------------|---|
| ScalarDataType | VARCHAR [(<i><Length></i>)] DECIMAL [(<i><Precision></i> , <i><Scale></i>)] INTEGER DOUBLE [<i><Precision></i>] TIMESTAMP |
| Scale | INTEGER_LITERAL |
| SelectItem | <i><ValueExpression></i> [[AS] <i><ColumnAlias></i>] |
| SelectList | { <i><SelectItem></i> } [,...n] |
| SelectStatement | (<i><SelectWithoutOrder></i> [<i><OrderByClause></i>]) <i><SelectWithoutFrom></i> |
| SelectWithoutFrom | SELECT <i><ValueSelectList></i> |
| SelectWithoutOrder | ((SELECT [ALL DISTINCT] <i><SelectList></i> <i><FromClause></i> [<i><WhereClause></i>] [<i><GroupByClause></i>] [<i><HavingClause></i>] [<i><SetClause></i>]) (<i><SelectWithoutOrder></i>)) [<i><SetClause></i>] |
| SetClause | UNION ALL (<i><SelectWithoutOrder></i> <i><SelectWithoutFrom></i>) |
| SignedLiteral | CHAR_LITERAL [+ -]INTEGER_LITERAL [+ -]DOUBLE_LITERAL [+ -]DECIMAL_LITERAL TIMESTAMP TIMESTAMP_STRING |
| SimpleCondition | EXISTS <i><SubQuery></i> <i><SubQuery></i> <i><RelationalOperator></i> <i><ValueExpression></i> <i><ValueExpression></i> (<i><RelationalOperator></i> ([ANY ALL] <i><SubQuery></i>) <i><ValueExpression></i>) |

(continues)

Table 2-3 Syntax for the Actuate SQL grammar parts (continued)

| Grammar part | Syntax |
|--------------------------------|---|
| SimpleCondition (continued) | IS [NOT] NULL [NOT] (BETWEEN <ValueExpression> AND <ValueExpression> LIKE <ValueExpression> [ESCAPE <ValueExpression>] IN <SubQuery> IN (ExpressionList))) |
| SubQuery | (<SelectWithoutOrder> [OPTION (SINGLE EXEC)]) |
| TableParameter | (<SignedLiteral> NULL <ParameterReference> <ValueExpression>) |
| TableParameters | <TableParameter> [,...n] |
| UnaryExpression | [+ -] <PrimaryExpression> |
| UnaryLogicalExpression | [NOT] <ConditionalPrimary> |
| UnsignedLiteral | CHAR_LITERAL INTEGER_LITERAL IDOUBLE_LITERAL IDECIMAL_LITERAL ITIMESTAMP TIMESTAMP_STRING |
| ValueExpression | <AdditiveExpression> <CaseExpression> |
| ValueSelectItem | <ValueExpression> [[AS] <ColumnAlias>] |
| ValueSelectList | {<ValueSelectItem>} [,...n] |
| WhenClause | WHEN <ValueExpression> THEN <ValueExpression> |
| WhereClause | WHERE <CondExpr> |

Using white space characters

White space characters include the space, tab, and new line characters. Multiple white space characters are not significant outside of literal strings and quoted identifiers.

Using keywords

The Actuate SQL keywords are shown in the following list:

- | | | |
|---------------|-----------|-------------|
| ■ ALL | ■ EXEC | ■ ON |
| ■ AND | ■ EXISTS | ■ OPTION |
| ■ ANY | ■ FALSE | ■ OPTIONAL |
| ■ AS | ■ FILTERS | ■ OR |
| ■ ASC | ■ FROM | ■ ORDER |
| ■ AVG | ■ GROUP | ■ OUTER |
| ■ BETWEEN | ■ HAVING | ■ PRAGMA |
| ■ BY | ■ IN | ■ PRECISION |
| ■ CARDINALITY | ■ INNER | ■ RIGHT |
| ■ CASE | ■ INT | ■ SELECT |
| ■ CAST | ■ INTEGER | ■ SINGLE |
| ■ COUNT | ■ IS | ■ SUM |
| ■ DEC | ■ JOIN | ■ THEN |
| ■ DECIMAL | ■ LEFT | ■ TIMESTAMP |
| ■ DEPENDENT | ■ LIKE | ■ TRUE |
| ■ DESC | ■ MAX | ■ UNION |
| ■ DISTINCT | ■ MERGE | ■ VARCHAR |
| ■ DOUBLE | ■ MIN | ■ WHEN |
| ■ ELSE | ■ NL | ■ WHERE |
| ■ END | ■ NOT | ■ WITH |
| ■ ESCAPE | ■ NULL | |

Actuate SQL keywords are not case-sensitive. To prevent incompatibility with other versions of SQL, do not use SQL-92 keywords. If you use an identifier that is also a keyword, place double quotes around the identifier.

Using comments

Precede a single-line comment with two hyphens. Enclose a multiline comment with `/*` and `*/`.

Specifying maps and information objects in Actuate SQL queries

In Information Object Designer, a map or information object name should be qualified by its relative path in the Encyclopedia volume. The path is relative to the IOB file. Use slashes to separate components of the path, for example:

```
../Data Sources/MyDatabase/dbo.customers.sma
```

In a query from a report designer, a map or information object name should be qualified by its absolute path in the Encyclopedia volume. Use slashes to separate components of the path, for example:

```
/MyProject/Data Sources/MyDatabase/dbo.customers.sma
```

Using identifiers in Actuate SQL

Identifiers include table and column names. Actuate SQL identifiers have the same limitations as standard SQL identifiers. For example, you must enclose an identifier in double quotes if it contains an illegal character such as a space or if it is identical to a SQL-92 keyword. Unlike the SQL-92 standard, however, there is no length limitation on Actuate SQL identifiers. Identifiers can contain Unicode characters.

Using column aliases in Actuate SQL

When you use column aliases, the following rules apply:

- The column and alias names of the items in the first SELECT statement of a UNION of statements are definitive.
- Within the items in a SELECT statement, you can use previously defined aliases to create expressions, for example:

```
SELECT col1 AS a, col2 AS b, a+b
```
- Only SELECT and ORDER BY can use aliases.
- You cannot use an alias in an aggregate expression, for example, MAX(a).
- You can use aliases defined in an outer SELECT statement in a nested SELECT statement.
- You can use aliases from the items in the first SELECT statement in a set of UNION statements in the ORDER BY clause of the query.

Specifying parameter values

A parameter value must be one of the following:

- A literal value, for example 'abc' or 123

- The NULL literal value
- A parameter reference
- An expression consisting of literal values, parameter references, and Actuate SQL functions and operators

A parameter value cannot include column references, subqueries, or aggregate functions.

Examples MyInformationObject uses the parameters p1, p2, and p3. The following query passes the parameter values :p1, -100, and 'abc' to MyInformationObject. :p1 represents the value of parameter p1 provided by the user. -100 and 'abc' are literal values:

```
WITH ( p1 INTEGER, p2 INTEGER, p3 VARCHAR )
SELECT ...
FROM "MyInformationObject.iob" [:p1, -100, 'abc']
```

MyInformationObject uses the parameter p1. The following query passes the NULL literal value to MyInformationObject:

```
WITH ( p1 INTEGER )
SELECT ...
FROM "MyInformationObject.iob" [NULL]
```

MyInformationObject uses the parameter p1. The following query passes the NULL literal value, cast as integer data type, to MyInformationObject:

```
WITH ( p1 INTEGER )
SELECT ...
FROM "MyInformationObject.iob" [CAST(NULL AS INTEGER)]
```

MyInformationObject uses the parameter p1. The following query passes the expression :p1 + 10 to MyInformationObject:

```
WITH ( p1 INTEGER )
SELECT ...
FROM "MyInformationObject.iob" [:p1 + 10]
```

MyInformationObject uses the parameters p1 and p2. The following query passes the parameter reference :p1 and the expression :p1 || :p2 to MyInformationObject:

```
WITH ( p1 VARCHAR, p2 VARCHAR )
SELECT ...
FROM "MyInformationObject.iob" [:p1, :p1 || :p2]
```

MyInformationObject uses the parameters p1 and p2. The following query passes two expressions to MyInformationObject:

```
WITH ( p1 INTEGER, p2 INTEGER )
```

(continues)

```
SELECT ...
FROM "MyInformationObject.iob" [:p1 + 10, CASE WHEN :p2 > 100 THEN
    100 ELSE 0 END]
```

Using subqueries in Actuate SQL

Subqueries have the following limitations:

- Subqueries are supported in every clause except the FROM clause. Specifically:
 - Subqueries cannot be used in Actuate SQL parameters or JOIN conditions.
 - Subqueries cannot constitute derived tables.

Derived tables are tables in a FROM clause that are the result of running a subquery.
- Subqueries must be operands to the operators IN or EXISTS, or operands to a comparison operator such as =, >, or >=ALL. Only one operand of the comparison operator can be a subquery, not both.
- Only single-column subqueries are supported. In other words, each subquery must have only one SELECT item.
- Subqueries cannot have more than one SELECT statement. In other words, set operators such as UNION ALL are not allowed in subqueries.

Subqueries can use OPTION (SINGLE EXEC). The SINGLE EXEC option improves the performance of a query when the query cannot be pushed to the database. When the SINGLE EXEC option is specified, the non-correlated portion of the subquery is executed once against the target database, while the correlated portion is executed within the Integration service.

By default, a subquery from a different database is implemented using a dependent join. Using the SINGLE EXEC option, a subquery can be executed using a single dependent query instead of executing one dependent query for each row of the outer query, for example:

```
SELECT DISTINCT CUSTOMERS.CUSTID AS "CUSTID",
    ORDERS.ORDERID AS "ORDERID"
FROM "../Data Sources/MyDatabase/CUSTOMERS.SMA" CUSTOMERS
INNER JOIN "../Data Sources/MyDatabase/ORDERS.SMA" ORDERS
ON CUSTOMERS.CUSTID = ORDERS.CUSTID
WHERE (SELECT count (ITEMS.PRICEQUOTE)
FROM "../Data Sources/YourDatabase/ITEMS.SMA" ITEMS
WHERE ORDERS.ORDERID = ITEMS.ORDERID
OPTION (SINGLE EXEC) ) < 100
ORDER BY CUSTOMERS.CUSTID, ORDERS.ORDERID
```

Using derived tables in Actuate SQL

A derived table is a virtual table that is calculated on the fly from a SELECT statement. A derived table can be used in a FROM clause, WHERE clause, HAVING clause, or subquery, for example:

```
SELECT Column01
FROM (Derived_table)
```

A derived table can have parameters.

Data types and data type casting

Table 2-4 lists Actuate SQL data types and a description of each data type.

Table 2-4 Actuate SQL data types

| Data type | Description |
|----------------------------------|--|
| String ("VARCHAR") | A sequence of Unicode characters. You can specify a maximum character length for the string. For example, VARCHAR (30) represents strings with a maximum length of 30 Unicode characters. |
| Integer number ("INTEGER") | 32-bit two's-complement arithmetic numbers. |
| Decimal number ("DECIMAL") | Fixed point numbers consisting of up to 100 digits. You can specify a maximum scale and a maximum precision using the syntax (precision, scale). For example, DECIMAL (15, 4) represents decimals that can have up to 15 digits in all and up to 4 digits after the decimal point. |
| Floating point number ("DOUBLE") | 64-bit IEEE double precision floating point numbers. |
| Timestamp ("TIMESTAMP") | A combined date and time (hour/minute/second). |

Facets

The precision, scale, and length associated with a database data type are called facets. Facets are supported for the corresponding Actuate SQL data type.

An Actuate SQL expression that evaluates to a scalar value has facets. These facets are determined by the Actuate SQL functions used in the expression and

the facets on the columns in the expression. You can specify facets for an Actuate SQL expression by using a cast, for example:

```
CAST (EXPR AS DECIMAL (38, 8))
```

```
CAST (EXPR AS VARCHAR(25))
```

Parameters in Actuate SQL queries have facets. These facets determine the maximum precision, scale, and length of parameter values. When no facets are specified for a parameter or a cast expression, the defaults are used. The default precision and scale for the Actuate SQL DECIMAL data type are (20, 8). The default length for the Actuate SQL VARCHAR data type is 50.

If the decimal value passed into a parameter or cast expression is too large for the precision and scale, an error results. Actuate SQL truncates digits after the decimal point to force the decimal value to fit within the precision and scale.

If the string or timestamp value passed into a string parameter, or into a cast expression to VARCHAR, is too large for the string length specified, the string or timestamp is truncated. If the string value passed into a cast expression to DECIMAL is too large for the precision and scale specified, an error results.

By default, Actuate SQL has a decimal precision of 38. The decimal precision can be set to a smaller or larger value up to 100. Results of calculations that exceed this limit may have their precision and scale truncated. Calculations may also be limited by the database. The same applies to operations on strings in the database.

Casting rules

The following casting rules apply:

- Integers can be implicitly cast to decimals and doubles. For implicit casts to decimals, the resulting decimals have a precision of 10 and a scale of 0. Integers can be explicitly cast to these types, as well as to strings.
- Decimals can be implicitly cast to doubles. Decimals can be explicitly cast to doubles, as well as to integers and strings. Conversion to integer type may result in rounding or truncation of data.
- Doubles can be explicitly cast to strings, as well as to integers and decimals. Conversion to decimal and integer types may result in rounding or truncation of data.
- Timestamps can be explicitly cast to strings. Casting to other types is not permitted.
- Strings can be implicitly or explicitly cast to timestamps. For explicit casting, the strings must be of the form:

```
yyyy-MM-dd hh:mm:ss.fff
```

Strings can be explicitly cast to integers, decimals, and doubles.

- Because databases vary in their implementation, casts to strings do not have a defined format. For example, the same value can be represented as 6E5, 60000, or 60000.00.
- All types can be implicitly cast to the same type.

Table 2-5 summarizes the casting rules for Actuate SQL data types.

Table 2-5 Casting rules for Actuate SQL types

| | To INTEGER | To DECIMAL | To DOUBLE | To VARCHAR | To TIMESTAMP |
|-----------------------|---------------------------|---------------------------|---------------------------|---------------------------|-------------------------|
| From INTEGER | Implicit casting occurs | Implicit casting occurs | Implicit casting occurs | Explicit casting required | Casting not permitted |
| From DECIMAL | Explicit casting required | Implicit casting occurs | Implicit casting occurs | Explicit casting required | Casting not permitted |
| From DOUBLE | Explicit casting required | Explicit casting required | Implicit casting occurs | Explicit casting required | Casting not permitted |
| From VARCHAR | Explicit casting required | Explicit casting required | Explicit casting required | Implicit casting occurs | Implicit casting occurs |
| From TIMESTAMP | Casting not permitted | Casting not permitted | Casting not permitted | Explicit casting required | Implicit casting occurs |

String comparison and ordering

The BIRT iServer Integration service compares and orders strings according to the Unicode code point value of each character. For example, Bright-Abbott is sorted before Brightman because the hyphen (-) has a Unicode value of 45, while lowercase m has a Unicode value of 109. The expression:

```
'Kirsten' LIKE 'ki%'
```

evaluates to False because uppercase K is different from lowercase k.

Although string comparison is case-sensitive by default, you can configure the Integration service to do case-insensitive comparison and ordering.

Functions and operators

Actuate SQL supports several built-in operators and named functions. Functions and operators are described in the following topics, grouped by related functionality.

Comparison operators: =, <>, >=, >, <=, <

Comparison operators are used to compare the value of two expressions, returning True if the comparison succeeds, and False if it does not. The following rules apply to the use of comparison operators handled by the Integration service:

- For numeric data types, the usual rules of arithmetic comparisons apply.
- For string comparisons, the shorter of the two strings is padded with space characters to equal the length of the longer string before the comparison is performed, as in SQL-92.
- Timestamps are compared using chronological order.
- An equality comparison between two floating point numbers does not return an error.

For information about the Integration service, see *Configuring BIRT iServer*. Comparison operations delegated to a remote data source may vary from the rules for comparison operators handled by the Integration service.

Range test operator: BETWEEN

The BETWEEN operator tests a value to see if it occurs in a given range including the endpoints. For example, the expression:

```
col BETWEEN 10 AND 20
```

evaluates to True if and only if the value of col is at least 10 but no more than 20. Table 2-6 shows the result type for using BETWEEN for each operand data type.

Table 2-6 Result data types for using BETWEEN with various operand types

| First operand type | Second operand type | Third operand type | Result type |
|--------------------|---------------------|--------------------|-------------|
| Boolean | Boolean | Boolean | Boolean |
| Integer | Integer | Integer | Boolean |
| Decimal | Decimal | Decimal | Boolean |
| Double | Double | Double | Boolean |
| Varchar | Varchar | Varchar | Boolean |
| Timestamp | Timestamp | Timestamp | Boolean |

The BETWEEN operator follows the same rules as the comparison operators.

Comparison operator: IN

The IN operator tests a row or scalar value to see if it occurs in a set of values. For example, the expression:

```
column IN (1,3,5,7,9)
```

evaluates to True if and only if the value of column is 1, 3, 5, 7, or 9.

Arithmetic operators: +, -, *, /

These operators implement addition, subtraction, multiplication, and division on the supported numeric data types. For decimal data types, the result's precision and scale are shown in Table 2-7. d1 represents an operand expression with precision p1 and scale s1, and d2 represents an operand expression with precision p2 and scale s2. The result's precision and scale may be truncated due to database limitations.

Table 2-7 Precision and scale of arithmetic operation results

| Operation | Result's precision | Result's scale |
|-----------|---|------------------------|
| d1 + d2 | $\max(s1, s2) + \max(p1-s1, p2-s2) + 1$ | $\max(s1, s2)$ |
| d1 - d2 | $\max(s1, s2) + \max(p1-s1, p2-s2) + 1$ | $\max(s1, s2)$ |
| d1 * d2 | $p1 + p2 + 1$ | $s1 + s2$ |
| d1 / d2 | $p1 - s1 + s2 + \max(6, s1 + p2 + 1)$ | $\max(6, s1 + p2 + 1)$ |

Integer arithmetic operations are performed using 32-bit two's-complement semantics. Floating point operations are performed according to the IEEE double precision standard.

These general rules apply to operations handled by the Integration service. Operations delegated to remote data sources may vary in their semantics. For information about the Integration service, see *Configuring BIRT iServer*.

Table 2-8 shows the result type of using arithmetic operators with each operand type.

Table 2-8 Result data types for using arithmetic operators with various operand types

| Left operand type | Right operand type | Result type |
|-------------------|--------------------|-------------|
| Integer | Integer | Integer |
| Decimal | Decimal | Decimal |
| Double | Double | Double |

Numeric functions

Actuate SQL supports the following numeric functions:

- FLOOR, CEILING, MOD
- ROUND
- POWER

FLOOR, CEILING, MOD

FLOOR returns the largest integer not greater than the argument's value. The result is cast to the specified type:

```
Decimal FLOOR( value Decimal )  
Double FLOOR( value Double )
```

Example The following code:

```
SELECT FLOOR(123.45), FLOOR(-123.45), FLOOR(0.0)
```

returns

```
123, -124, 0
```

CEILING returns the smallest integer not less than the argument's value. The result is cast to the specified type:

```
Decimal CEILING( value Decimal )  
Double CEILING( value Double )
```

Example The following code:

```
SELECT CEILING(123.45), CEILING(-123.45), CEILING(0.0)
```

returns

```
124, -123, 0
```

MOD returns the remainder after division of two integers:

```
Integer MOD( v1 Integer, v2 Integer )
```

Example The following code:

```
SELECT CUSTOMERS.CUSTID, CUSTOMERS.CUSTOMNAME  
FROM "../Data Sources/MyDatabase/CUSTOMERS.SMA" CUSTOMERS  
WHERE MOD(CUSTOMERS.CUSTID, 2) = 1
```

returns

```
101,Signal Engineering  
109,InfoEngineering  
111,Advanced Design Inc.  
...
```

For decimal data types, the result's precision and scale for the FLOOR and CEILING functions are $(p + 1, s)$, where (p, s) are the precision and scale of the operand.

ROUND

ROUND returns the number closest in value to the first argument, rounding away from zero. The second argument specifies the precision, with positive values indicating a position to the right of the decimal point, and negative values indicating a position to the left of the decimal point. All positions to the right of the specified position are zero in the result:

```
Integer ROUND( value Integer, precision Integer )  
Decimal ROUND( value Decimal, precision Integer )  
Double ROUND( value Double, precision Integer )
```

Example The following code:

```
SELECT ROUND(123.4567, 2), ROUND(123.4567, -1)
```

returns

```
123.46, 120
```

For decimal data types, the result's precision and scale are $(p + 1, s)$, where (p, s) are the precision and scale of the operand.

POWER

POWER raises the left argument (base) to the power of the right argument (exponent):

```
Integer POWER( base Integer, exponent Integer )  
Decimal POWER( base Decimal, exponent Integer )  
Double POWER( base Double, exponent Integer )
```

Example The following code:

```
SELECT CUSTOMERS.CUSTID, POWER(CUSTOMERS.CUSTID, 2)  
FROM "../Data Sources/MyDatabase/CUSTOMERS.SMA" CUSTOMERS
```

returns

```
101,10201
```

```
102,10404
```

```
104,10816
```

```
...
```

For decimal data types, the result's precision and scale are (P, s) , where P is the maximum precision in the database or the Integration service, and s is the scale of the operand.

Null test operators: IS [NOT] NULL

These operators allow expressions to be tested for NULL values. For example, the expression:

```
column IS NULL
```

evaluates to True if and only if column has the value NULL.

Logical operators: AND, OR, NOT

These operators implement Boolean conjunction, disjunction, and negation, respectively. AND and OR take two Boolean operands each, while NOT takes a single operand. All return Boolean values.

For AND and OR, both operands may be evaluated even if one operand is undefined, particularly in queries against multiple databases. For example, the clause:

```
WHERE QUANTITY <> 0 AND TOTALCOST / QUANTITY > 50
```

may result in an error for rows where QUANTITY = 0.

String functions and operators

Actuate SQL supports the following string functions and operators:

- Case conversion functions: UPPER, LOWER
- Concatenation operator: ||
- Length function: CHAR_LENGTH
- LIKE operator
- Substring functions: LEFT, RIGHT, SUBSTRING
- Trimming functions: LTRIM, RTRIM, TRIM
- Search function: POSITION

Case conversion functions: UPPER, LOWER

These functions return a string formed by converting the characters in the argument to uppercase or lowercase respectively, provided the character is alphabetic:

```
Varchar UPPER( value Varchar )  
Varchar LOWER( value Varchar )
```

Examples The following code:

```
SELECT CUSTOMERS.CUSTID, CUSTOMERS.CUSTOMNAME,  
       UPPER (CUSTOMERS.CUSTOMNAME)
```

```
FROM "../Data Sources/MyDatabase/CUSTOMERS.SMA" CUSTOMERS
```

returns

```
101,Signal Engineering,SIGNAL ENGINEERING
109,InfoEngineering,INFOENGINEERING
111,Advanced Design Inc.,ADVANCED DESIGN INC.
...
```

The following code:

```
SELECT CUSTOMERS.CUSTID, CUSTOMERS.CUSTOMNAME,
       LOWER(CUSTOMERS.CUSTOMNAME)
FROM "../Data Sources/MyDatabase/CUSTOMERS.SMA" CUSTOMERS
```

returns

```
101,Signal Engineering,signal engineering
109,InfoEngineering,infoengineering
111,Advanced Design Inc.,advanced design inc.
...
```

Concatenation operator: ||

This operator concatenates two string values, returning a new string that contains the characters from the left operand followed by the characters from the right operand.

Length function: CHAR_LENGTH

This function computes the length of a string, returning an integer count of its characters. Trailing spaces are significant:

Integer **CHAR_LENGTH**(value Varchar)

Example The following code:

```
SELECT CUSTOMERS.CUSTID, CUSTOMERS.CONTACT_FIRST
FROM "../Data Sources/MyDatabase/CUSTOMERS.SMA" CUSTOMERS
WHERE CHAR_LENGTH(CUSTOMERS.CONTACT_FIRST) > 5
```

returns

```
102,Leslie
109,Michael
116,William
...
```

LIKE operator

The LIKE operator is used in an expression such as:

```
column LIKE 'Mar%'
```

In this example, values of column, such as Mary or Martin, satisfy the test because both start with Mar.

A LIKE operator pattern must be a literal string, for example, 'abc%', a parameter, or an expression. The LIKE operator does not support column references, subqueries, or aggregate expressions. Other examples include:

```
column LIKE :paramState  
column LIKE CURRENT_USER( )
```

The following rules apply:

- Literal pattern characters must match exactly. LIKE is case-sensitive.
- An underscore character (_) matches any single character.
- A percent character (%) matches zero or more characters.

Escape a literal underscore, percent, or backslash character with a backslash character (\). Alternatively, use the following syntax:

```
test_string LIKE pattern_string ESCAPE escape_character
```

The escape character must obey the same rules as the LIKE operator pattern.

Substring functions: LEFT, RIGHT, SUBSTRING

These functions transform a string by retrieving a subset of its characters.

LEFT and RIGHT return the leftmost or rightmost n characters, respectively. Each takes the string as the first argument and the number of characters to retrieve as the second argument:

```
Varchar LEFT( value Varchar, offset Integer )  
Varchar RIGHT( value Varchar, offset Integer )
```

Specifying an offset that is less than zero results in an error. If the offset is greater than the length of the string, these functions return the entire string.

SUBSTRING takes three arguments: the input string, the start position (one-based offset from the left side), and the number of characters to retrieve. It returns the substring located at this position:

```
Varchar SUBSTRING( input Varchar, start Integer, length Integer )
```

The following actions result in an error:

- Specifying a start position that is less than or equal to zero.
- Specifying a length that is less than zero.

Examples The following code:

```
SELECT CUSTOMERS.CUSTID, CUSTOMERS.CUSTOMNAME  
FROM "../Data Sources/MyDatabase/CUSTOMERS.SMA" CUSTOMERS  
WHERE LEFT(CUSTOMERS.CUSTOMNAME, 4) = 'Info'
```


returns

```
109,InfoEngineering
117,InfoDesign
129,InfoSpecialists
...
```

The following code:

```
SELECT CUSTOMERS.CUSTID, CUSTOMERS.CUSTOMNAME
FROM "../Data Sources/MyDatabase/CUSTOMERS.SMA" CUSTOMERS
WHERE RIGHT(CUSTOMERS.CUSTOMNAME, 5) = 'Corp.'
```

returns

```
104,SigniSpecialists Corp.
115,Design Solutions Corp.
118,Computer Systems Corp.
...
```

The following code:

```
SELECT CUSTOMERS.CUSTID, CUSTOMERS.CUSTOMNAME,
       SUBSTRING(CUSTOMERS.CUSTOMNAME, 2, 5)
FROM "../Data Sources/MyDatabase/CUSTOMERS.SMA" CUSTOMERS
```

returns

```
101,Signal Engineering,ignal
102,Technical Specialists Co.,echni
104,SigniSpecialists Corp.,igniS
...
```

Trimming functions: LTRIM, RTRIM, TRIM

These functions strip space characters from a string. LTRIM strips only from the left side, RTRIM only from the right side, and TRIM from both sides. In all cases the result value is a string identical to the argument except for the possible removal of space characters from either side. Other white space characters, including tabs and newlines, are not removed by these functions:

```
Varchar LTRIM( value Varchar )
Varchar RTRIM( value Varchar )
Varchar TRIM( value Varchar )
```

Examples The following code:

```
SELECT LTRIM(' Title '), 'Author'
```

returns

```
Title ,Author
```

The following code:

```
SELECT RTRIM('    Title    '), 'Author'
returns
```

```
    Title, Author
```

The following code:

```
SELECT TRIM('    Title    '), 'Author'
returns
```

```
Title, Author
```

Search function: POSITION

The POSITION function takes two arguments: a substring and a search string. The POSITION function returns the position of the substring in the search string as an integer or as 0 if the substring is not found. If the substring is the empty string, the POSITION function returns 1. The POSITION function is case-sensitive:

```
Integer POSITION( substring Varchar, searchstring Varchar )
```

Example The following code:

```
SELECT CUSTOMERS.CUSTID, CUSTOMERS.CUSTOMNAME
FROM "../Data Sources/MyDatabase/CUSTOMERS.SMA" CUSTOMERS
WHERE POSITION('Inc.', CUSTOMERS.CUSTOMNAME) > 0
```

returns

```
106, Technical MicroSystems Inc.
111, Advanced Design Inc.
113, Technical Design Inc.
...
```

Timestamp functions

These functions perform operations on timestamp values:

- CURRENT_TIMESTAMP
- CURRENT_DATE
- DATEADD
- DATEDIFF
- DATEPART
- DATESERIAL

When using these functions, use the control strings listed in Table 2-9 to represent units of time. The control string used in a function must be a literal string, not an expression or a parameter.

Table 2-9 Control strings for various units of time

| Unit of time | Control string |
|--------------|----------------|
| year | yyyy |
| quarter | q |
| month | m |
| day | d |
| day of year | y |
| day of week | w |
| hour | h |
| minute | n |
| second | s |

CURRENT_TIMESTAMP

CURRENT_TIMESTAMP returns a timestamp value for the current date and time:

Timestamp **CURRENT_TIMESTAMP()**

Example The following code:

```
SELECT CURRENT_TIMESTAMP()
```

returns

```
2004-10-27 14:49:23.0
```

CURRENT_DATE

CURRENT_DATE returns a timestamp value for the current date with the time set to 00:00:00.0:

Timestamp **CURRENT_DATE()**

Example The following code:

```
SELECT CURRENT_DATE()
```

returns

```
2004-10-27 00:00:00.0
```

DATEADD

DATEADD takes three arguments: a control string, an integer delta value, and a timestamp value. It returns a timestamp that applies the delta value to the specified part of the original timestamp. The operation carries if the sum of the original field value and the delta is illegal:

```
Timestamp DATEADD( control Varchar, delta Integer,  
                    value Timestamp )
```

Example The following code:

```
SELECT ORDERS.ORDERID, ORDERS.SHIPBYDATE,  
       DATEADD('d', 14, ORDERS.SHIPBYDATE) AS ExpectedDelivery  
FROM "../Data Sources/MyDatabase/ORDERS.SMA" ORDERS
```

returns

```
1645,1995-05-22 00:00:00.0,1995-06-05 00:00:00.0  
1340,1995-06-03 00:00:00.0,1995-06-17 00:00:00.0  
1810,1995-04-12 00:00:00.0,1995-04-26 00:00:00.0  
...
```

DATEDIFF

DATEDIFF takes three arguments: a control string, a start timestamp, and an end timestamp. It returns the integer delta between the part of the two timestamps specified by the control string. Components smaller than the control string are ignored. Components larger than the control string contribute to the result:

```
Integer DATEDIFF( control Varchar, start Timestamp,  
                  end Timestamp )
```

Examples The following code:

```
SELECT ORDERS.ORDERID, ORDERS.SHIPBYDATE, ORDERS.FORECASTSHIPDATE,  
       DATEDIFF('d', ORDERS.SHIPBYDATE, ORDERS.FORECASTSHIPDATE) AS  
       ShipDateDifference  
FROM "../Data Sources/MyDatabase/ORDERS.SMA" ORDERS
```

returns

```
1645,1995-05-22 00:00:00.0,1995-06-02 00:00:00.0,11  
1340,1995-06-03 00:00:00.0,1995-06-10 00:00:00.0,7  
1810,1995-04-12 00:00:00.0,1995-04-27 00:00:00.0,15  
...
```

The following expression:

```
DATEDIFF('d', CAST('2005-12-31 23:59:59.0' AS TIMESTAMP),  
         CAST('2006-01-01 00:00:00.0' AS TIMESTAMP))
```

returns 1. The control string d indicates that the difference is in days. The difference between December 31, 2005 and January 1, 2006 is one day. The hours, minutes, and seconds components are ignored.

The following expression:

```
DATEDIFF('m', CAST('2005-12-31 23:59:59.0' AS TIMESTAMP),  
         CAST('2006-01-01 00:00:00.0' AS TIMESTAMP))
```

returns 1. The control string m indicates that the difference is in months. The difference between December 31, 2005 and January 1, 2006 is one month. The day, hours, minutes, and seconds components are ignored.

DATEPART

DATEPART takes two arguments: a control string and a timestamp. It returns the part of the timestamp specified by the control string:

```
Integer DATEPART( control Varchar, value Timestamp )
```

Example The following code:

```
SELECT ORDERS.ORDERID, ORDERS.SHIPBYDATE  
FROM "../Data Sources/MyDatabase/ORDERS.SMA" ORDERS  
WHERE DATEPART('m', ORDERS.SHIPBYDATE) = 5
```

returns

```
1645,1995-05-22 00:00:00.0  
1725,1995-05-10 00:00:00.0  
1125,1995-05-03 00:00:00.0  
...
```

DATESERIAL

DATESERIAL has two forms. The first form takes three arguments: a year value, a month value, and a day value. It returns a timestamp for the date corresponding to the specified year, month, and day with the time set to 00:00:00.0:

```
Timestamp DATESERIAL( year Integer, month Integer, day Integer )
```

The second form of DATESERIAL takes six arguments: values for the year, month, day, hour, minute, and second. It returns the timestamp for the specified values:

```
Timestamp DATESERIAL( year Integer, month Integer, day Integer,  
                      hour Integer, minute Integer, second Integer )
```

Example The following code:

```
SELECT ORDERS.ORDERID, ORDERS.ASKBYDATE  
FROM "../Data Sources/MyDatabase/ORDERS.SMA" ORDERS  
WHERE ORDERS.ASKBYDATE >= DATESERIAL(1995, 6, 15, 12, 59, 59)
```

returns

```
1555,1995-06-28 00:00:00.0
```

(continues)

```
1725,1995-06-23 00:00:00.0
1720,1995-06-17 00:00:00.0
...
```

Aggregate functions: COUNT, MIN, MAX, SUM, AVG

These functions aggregate an entire column of values into a single scalar result. For decimal data types:

- For the MIN, MAX, and AVG functions, the result's precision and scale are the same as the precision and scale of the operand.
- For the SUM function, the result's precision and scale are (P, s), where P is the maximum precision in the database or the Integration service, and s is the scale of the operand.

The COUNT function reduces any argument type to a single integer representing the number of non-NULL items. As in SQL-92, COUNT(*) counts the number of rows in a table:

```
Integer COUNT( column )
```

Example The following code:

```
SELECT COUNT(ORDERS.ORDERID) AS NumberOfOrders
FROM "../Data Sources/MyDatabase/ORDERS.SMA" ORDERS
```

returns

```
111
```

MIN and MAX accept any type and return the minimum or maximum value, using the same rules that apply to comparison of individual items:

```
ColumnType MIN( column )
```

```
ColumnType MAX( column )
```

Examples The following code:

```
SELECT MIN(ITEMS.QUANTITY)
FROM "../Data Sources/MyDatabase/ITEMS.SMA" ITEMS
```

returns

```
2
```

The following code:

```
SELECT MAX(ITEMS.QUANTITY)
FROM "../Data Sources/MyDatabase/ITEMS.SMA" ITEMS
```

returns

```
6203
```

SUM and AVG can be applied to any of the three numeric types and produce the sum or average of all the numbers:

ColumnType **SUM**(*column*)

ColumnType **AVG**(*column*)

Examples The following code:

```
SELECT SUM(ITEMS.QUANTITY)
FROM "../Data Sources/MyDatabase/ITEMS.SMA" ITEMS
returns
606177
```

The following code:

```
SELECT AVG(ITEMS.QUANTITY)
FROM "../Data Sources/MyDatabase/ITEMS.SMA" ITEMS
returns
319
```

System function: CURRENT_USER

CURRENT_USER returns a string containing the user name for the current user:

Varchar **CURRENT_USER**()

Example The following code:

```
SELECT CURRENT_USER()
returns
user1
```

Providing query optimization hints

A report developer or business user uses an information object to create an Actuate SQL query. When you create the information object, you can provide hints that help to optimize the query. Specifically, you can:

- Indicate that a table in a join is optional.
- Specify the cardinality of a join.

For query optimization hints to take effect, you must create join conditions with the ON clause, not the WHERE clause.

Indicating that a table in a join is optional

When you create an information object, you indicate that a table in a join is optional using the **OPTIONAL** keyword. If you indicate that a table is optional and none of its columns appear in the query created by a report developer or business user (except in a join condition), the table is dropped from the optimized query.

The **OPTIONAL** keyword has no effect in queries created in the Information Object Query Builder.

For example, consider the following information object CustomersOrders:

```
SELECT Customers.custid, Customers.customname,  
       Customers.contact_last, Orders.orderid, Orders.custid,  
       Orders.amount, Orders.shipbydate  
FROM Customers.sma LEFT OPTIONAL INNER JOIN Orders.sma  
ON (Customers.custid = Orders.custid)
```

Now consider the following Actuate SQL query created by a report developer or business user using CustomersOrders:

```
SELECT Orders.custid, Orders.orderid, Orders.amount  
FROM CustomersOrders.iob  
WHERE Orders.amount BETWEEN 10000 and 20000
```

Because no column from the Customers table appears in the query, and because the join in CustomersOrders includes the **LEFT OPTIONAL** keywords, the Customers table is dropped from the optimized query:

```
SELECT Orders.custid, Orders.orderid, Orders.amount  
FROM Orders.sma  
WHERE Orders.amount BETWEEN 10000 and 20000
```

Now consider another Actuate SQL query created by a report developer or business user using CustomersOrders:

```
SELECT Customers.custid, Customers.contact_last  
FROM CustomersOrders.iob  
WHERE Customers.city = 'NYC'
```

No column from the Orders table appears in the query. But because the Orders table is not optional, it is not dropped from the query:

```
SELECT Customers.custid, Customers.contact_last  
FROM Customers.sma INNER JOIN Orders.sma  
ON (Customers.custid = Orders.custid)  
WHERE Customers.city = 'NYC'
```

If you use the **OPTIONAL** keyword without the **LEFT** or **RIGHT** qualifier, it applies to both tables in the join.

The OPTIONAL keyword is ignored when it applies to:

- A table whose columns appear in the query created by a report developer or business user, for example in the SELECT list or in the ORDER BY, GROUP BY, HAVING, or WHERE clauses.
- The middle table in an information object, for example:

```
SELECT Customers.custid, Items.orderid, Items.itemcode,  
       Items.description  
FROM Customers RIGHT OPTIONAL INNER JOIN Orders  
ON (Customers.custid = Orders.custid)  
LEFT OPTIONAL INNER JOIN Items ON (Orders.orderid =  
       Items.orderid)
```

In this information object, Orders is the middle table.

An information object that uses the OPTIONAL keyword cannot be joined to another information object. Therefore, an Actuate SQL query created by a report developer or business user cannot include more than one information object if that information object uses the OPTIONAL keyword.

Using the OPTIONAL keyword with a computed field

Do not define a computed field in an information object that contains the OPTIONAL keyword. Instead, define the computed field in a lower level information object.

For example, consider the information object MyInformationObject:

```
SELECT dbo_CUSTOMERS.CUSTID AS CUSTID, dbo_CUSTOMERS.CONTACT_FIRST  
       AS CONTACT_FIRST, dbo_CUSTOMERS.CONTACT_LAST AS CONTACT_LAST,  
       dbo_CUSTOMERS.CITY AS CITY, dbo_ORDERS.SHIPBYDATE AS  
       SHIPBYDATE, dbo_ORDERS.FORECASTSHIPDATE AS FORECASTSHIPDATE,  
       dbo_CUSTOMERS.ADDRESS AS ADDRESS,  
       ( dbo_ITEMS.PRICEQUOTE * dbo_ITEMS.QUANTITY ) AS Total  
FROM "dbo.CUSTOMERS.sma" AS dbo_CUSTOMERS  
OPTIONAL INNER JOIN "dbo.ORDERS.sma" AS dbo_ORDERS  
ON ( dbo_CUSTOMERS.CUSTID=dbo_ORDERS.CUSTID )  
OPTIONAL INNER JOIN "dbo.ITEMS.sma" AS dbo_ITEMS  
ON ( dbo_ORDERS.ORDERID=dbo_ITEMS.ORDERID )
```

MyInformationObject defines the computed field Total and also contains the OPTIONAL keyword.

Now consider the following Actuate SQL query created by a report developer or business user using MyInformationObject:

```
SELECT MyInformationObject.CUSTID AS CUSTID,  
       MyInformationObject.CONTACT_FIRST AS CONTACT_FIRST,  
       MyInformationObject.CITY AS CITY,  
       MyInformationObject.CONTACT_LAST AS CONTACT_LAST  
FROM "MyInformationObject.iob" AS MyInformationObject
```

The ORDERS and ITEMS tables are not dropped from the query even though the OPTIONAL keyword is applied to both tables in MyInformationObject and the SELECT clause does not contain columns from either table. The tables are not dropped because in MyInformationObject the columns ITEMS.PRICEQUOTE and ITEMS.QUANTITY are used in a computation outside the join condition.

To avoid this situation, define the computed field in a lower level information object such as ITEMS.iob. MyInformationObject then contains the following query:

```
SELECT dbo_CUSTOMERS.CUSTID AS CUSTID, dbo_CUSTOMERS.CONTACT_FIRST
AS CONTACT_FIRST, dbo_CUSTOMERS.CONTACT_LAST AS CONTACT_LAST,
dbo_CUSTOMERS.CITY AS CITY, dbo_ORDERS.SHIPBYDATE AS
SHIPBYDATE, dbo_ORDERS.FORECASTSHIPDATE AS FORECASTSHIPDATE,
dbo_CUSTOMERS.ADDRESS AS ADDRESS, ITEMS.Total AS Total
FROM "dbo.CUSTOMERS.sma" AS dbo_CUSTOMERS
OPTIONAL INNER JOIN "dbo.ORDERS.sma" AS dbo_ORDERS
ON ( dbo_CUSTOMERS.CUSTID=dbo_ORDERS.CUSTID )
OPTIONAL INNER JOIN "ITEMS.iob" AS ITEMS
ON ( dbo_ORDERS.ORDERID=ITEMS.ORDERID )
```

Using the OPTIONAL keyword with parentheses ()

You can control the processing of the OPTIONAL keyword with parentheses. For example, in the following query the tables CUSTOMERS and ORDERS can be dropped:

```
SELECT ITEMS.ORDERID, ITEMS.PRICEQUOTE, ITEMS.QUANTITY
FROM "CUSTOMERS.sma" AS CUSTOMERS INNER JOIN "ORDERS.sma" AS
ORDERS ON (CUSTOMERS.CUSTID = ORDERS.CUSTID) LEFT OPTIONAL
INNER JOIN "ITEMS.sma" AS ITEMS ON
(ORDERS.ORDERID = ITEMS.ORDERID)
```

In the following query, however, only the ORDERS table can be dropped because the join that includes the LEFT OPTIONAL keywords is enclosed in parentheses:

```
SELECT ITEMS.ORDERID, ITEMS.PRICEQUOTE, ITEMS.QUANTITY
FROM "CUSTOMERS.sma" AS CUSTOMERS INNER JOIN ("ORDERS.sma" AS
ORDERS LEFT OPTIONAL INNER JOIN "ITEMS.sma" AS ITEMS ON
(ORDERS.ORDERID = ITEMS.ORDERID) ) ON
(CUSTOMERS.CUSTID = ORDERS.CUSTID)
```

In the following examples, A, B, C, and D are tables.

Consider the following query that includes the RIGHT OPTIONAL keywords:

```
A RIGHT OPTIONAL JOIN B RIGHT OPTIONAL JOIN C RIGHT OPTIONAL
JOIN D
```

The Actuate SQL compiler interprets this query as:

```
((A RIGHT OPTIONAL JOIN B) RIGHT OPTIONAL JOIN C)
RIGHT OPTIONAL JOIN D
```

Tables B, C, and D can be dropped from the query.

Consider the following query that includes the LEFT OPTIONAL keywords without parentheses:

```
A LEFT OPTIONAL JOIN B LEFT OPTIONAL JOIN C LEFT OPTIONAL  
JOIN D
```

The Actuate SQL compiler interprets this query as:

```
((A LEFT OPTIONAL JOIN B) LEFT OPTIONAL JOIN C) LEFT OPTIONAL  
JOIN D
```

Tables A, B, and C can be dropped from the query. It is not possible, however, to drop table C without dropping tables A and B, or to drop table B without dropping table A, without using parentheses.

Consider the following query that includes the LEFT OPTIONAL keywords with parentheses:

```
A LEFT OPTIONAL JOIN (B LEFT OPTIONAL JOIN (C LEFT OPTIONAL  
JOIN D))
```

Table C can be dropped from the query without dropping tables A and B. Table B can be dropped from the query without dropping table A.

Consider the following query that includes the OPTIONAL keyword without the LEFT or RIGHT modifier:

```
A OPTIONAL JOIN B OPTIONAL JOIN C OPTIONAL JOIN D
```

The Actuate SQL compiler interprets this query as:

```
((A OPTIONAL JOIN B) OPTIONAL JOIN C) OPTIONAL JOIN D
```

Any table or set of tables can be dropped from the query.

Using the OPTIONAL keyword with aggregate functions

If a query created by a report developer or business user contains the function COUNT(*), the OPTIONAL keyword, if it appears in the information object, is ignored. If a query contains another aggregate function, for example SUM or COUNT(column), the value returned by the aggregate function depends on whether the information object includes the OPTIONAL keyword. For example, consider the following Actuate SQL query created by a report developer or business user using the CustomersOrders information object:

```
SELECT COUNT(Customers.custid) AS CustomerCount  
FROM CustomersOrders.iob
```

In the first case, consider the following information object CustomersOrders, which applies the OPTIONAL keyword to the Orders table:

```

SELECT Customers.custid, Customers.customname,
       Customers.contact_last, Orders.orderid, Orders.custid,
       Orders.amount, Orders.shipbydate
FROM Customers.sma RIGHT OPTIONAL INNER JOIN Orders.sma
ON (Customers.custid = Orders.custid)

```

Because no column from the Orders table appears in the query and because the join in CustomersOrders includes the RIGHT OPTIONAL keywords, the Orders table is dropped from the optimized query:

```

SELECT COUNT(Customers.custid) AS CustomerCount
FROM Customers.sma

```

In the second case, consider the following information object CustomersOrders, which does not apply the OPTIONAL keyword to the Orders table:

```

SELECT Customers.custid, Customers.customname,
       Customers.contact_last, Orders.orderid, Orders.custid,
       Orders.amount, Orders.shipbydate
FROM Customers.sma INNER JOIN Orders.sma
ON (Customers.custid = Orders.custid)

```

In this case, the Orders table is not dropped from the query:

```

SELECT COUNT(Customers.custid) AS CustomerCount
FROM Customers.sma INNER JOIN Orders.sma
ON (Customers.custid = Orders.custid)

```

The value of CustomerCount depends on whether the OPTIONAL keyword is applied to the Orders table in the CustomersOrders information object.

Specifying the cardinality of a join

You can specify the right-to-left and left-to-right cardinality of a join. Table 2-10 lists the cardinality types and a description of each type.

Table 2-10 Cardinality types

| Cardinality type | Description |
|------------------|---|
| 1 | One record in the first table matches one record in the second table. |
| ? | One record in the first table matches zero or one record in the second table. |
| * | One record in the first table matches zero or more records in the second table. |
| + | One record in the first table matches one or more records in the second table. |

The right-to-left cardinality type is followed by a hyphen (-), and then by the left-to-right cardinality type. The cardinality type depends on the join column.

For example:

```
Customers JOIN Orders ON (Customers.custid = Orders.custid)
    {CARDINALITY ('1-+') }
```

indicates that:

- One record in Orders matches one record in Customers.
- One record in Customers matches one or more records in Orders:

```
Customers JOIN Orders ON (Customers.custid = Orders.custid)
    {CARDINALITY ('1-*') }
```

indicates that:

- One record in Orders matches one record in Customers.
- One record in Customers matches zero or more records in Orders:

```
Customers JOIN Orders ON (Customers.custid = Orders.custid)
    {CARDINALITY ('*-?') }
```

indicates that:

- One record in Orders matches zero or more records in Customers.
- One record in Customers matches zero or one record in Orders.

Using pragmas to tune a query

If an information object query joins maps or information objects that are based on different data sources, you may be able to tune the query using the following pragmas:

- EnableCBO
- applyIndexing
- MinRowsForIndexing

These pragmas are described in the following topics.

Disabling cost-based optimization

If you provide values for the map and join column properties, the Actuate SQL compiler uses these values to do cost-based query optimization. You can disable cost-based optimization using the pragma EnableCBO.

For example, consider the following query based on SQL Server and Oracle database tables:

```

SELECT
    NATION.N_NAME,
    SUM(LINEITEM.L_EXTENDEDPRICE * (1 - LINEITEM.L_DISCOUNT))
    AS Revenue
FROM
    "/SQL_Server/CUSTOMER.SMA" CUSTOMER,
    "/Oracle/ORDERS.SMA" ORDERS,
    "/SQL_Server/LINEITEM.SMA" LINEITEM,
    "/SQL_Server/SUPPLY.SMA" SUPPLY,
    "/Oracle/NATION.SMA" NATION,
    "/Oracle/REGION.SMA" REGION
WHERE
    CUSTOMER.C_CUSTKEY = ORDERS.O_CUSTKEY
    AND LINEITEM.L_ORDERKEY = ORDERS.O_ORDERKEY
    AND LINEITEM.L_SUPPKEY = SUPPLY.S_SUPPKEY
    AND CUSTOMER.C_NATIONKEY = SUPPLY.S_NATIONKEY
    AND SUPPLY.S_NATIONKEY = NATION.N_NATIONKEY
    AND NATION.N_REGIONKEY = REGION.R_REGIONKEY
    AND REGION.R_NAME = 'ASIA'
    AND ORDERS.O_ORDERDATE >= TIMESTAMP '1993-01-01 00:00:00'
    AND ORDERS.O_ORDERDATE < TIMESTAMP '1994-01-01 00:00:00'
GROUP BY NATION.N_NAME

```

If you provide values for the map and join column properties, part of the query plan looks similar to Figure 2-1.

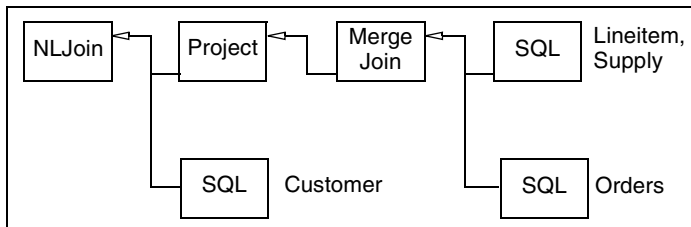


Figure 2-1 Example of part of the query plan for which values for the map and join column properties have been provided

To disable cost-based optimization for the query, set the pragma EnableCBO to False:

```

PRAGMA "EnableCBO" := 'false'
SELECT
    NATION.N_NAME,
    SUM(LINEITEM.L_EXTENDEDPRICE * (1 - LINEITEM.L_DISCOUNT))
    AS Revenue
FROM
    "/SQL_Server/CUSTOMER.SMA" CUSTOMER,
    "/Oracle/ORDERS.SMA" ORDERS,

```

```

"/SQL_Server/LINEITEM.SMA" LINEITEM,
"/SQL_Server/SUPPLY.SMA" SUPPLY,
"/Oracle/NATION.SMA" NATION,
"/Oracle/REGION.SMA" REGION
WHERE ...

```

Now this part of the query plan looks similar to Figure 2-2.

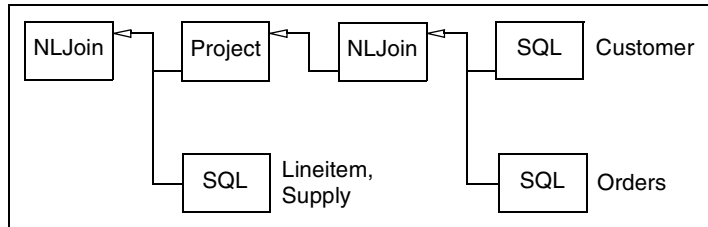


Figure 2-2 Example of part of a query plan with cost-based optimization disabled

Disabling cost-based optimization changes the join sequence and the join algorithm. The Customer and LineItem, Supply database subqueries switch positions, and the merge join is replaced with a nested loop join.

If you create a query using an information object for which cost-based optimization is disabled, cost-based optimization is disabled for the query as well.

You can disable cost-based optimization for all information object queries by setting the BIRT iServer configuration variable Enable cost based optimization to False. For more information about BIRT iServer configuration variables, see *Configuring BIRT iServer*.

Disabling indexing

By default, the Actuate SQL compiler creates indexes for rows that are materialized in memory during query execution, for example the rows returned when the right side of a nested loop join is executed. You can disable indexing using the pragma `applyIndexing`.

For example, to disable indexing for a query, set the pragma `applyIndexing` to False:

```

PRAGMA "applyIndexing" := 'false'
SELECT
    NATION.N_NAME,
    SUM(LINEITEM.L_EXTENDEDPRICE * (1 - LINEITEM.L_DISCOUNT))
    AS Revenue

```

(continues)

```

FROM
  "/SQL_Server/CUSTOMER.SMA" CUSTOMER,
  "/Oracle/ORDERS.SMA" ORDERS,
  "/SQL_Server/LINEITEM.SMA" LINEITEM,
  "/SQL_Server/SUPPLY.SMA" SUPPLY,
  "/Oracle/NATION.SMA" NATION,
  "/Oracle/REGION.SMA" REGION
WHERE ...

```

If you create a query using an information object for which indexing is disabled, indexing is disabled for the query as well.

Specifying a threshold value for indexing

If cost-based optimization is enabled and you provide values for the map and join column properties, the Actuate SQL compiler creates an index when 100 rows are materialized in memory during query execution. You can change the number of materialized rows that triggers indexing using the pragma `MinRowsForIndexing`.

If cost-based optimization is disabled, or you do not provide values for the map and join column properties, an index is created for materialized rows if a suitable column is available.

For example, to change the number of materialized rows that triggers indexing to 1000, set the pragma `MinRowsForIndexing` to 1000:

```

PRAGMA "MinRowsForIndexing" := '1000'
SELECT
  NATION.N_NAME,
  SUM(LINEITEM.L_EXTENDEDPRICE * (1 - LINEITEM.L_DISCOUNT))
  AS Revenue
FROM
  "/SQL_Server/CUSTOMER.SMA" CUSTOMER,
  "/Oracle/ORDERS.SMA" ORDERS,
  "/SQL_Server/LINEITEM.SMA" LINEITEM,
  "/SQL_Server/SUPPLY.SMA" SUPPLY,
  "/Oracle/NATION.SMA" NATION,
  "/Oracle/REGION.SMA" REGION
WHERE ...

```

Specifying the number of materialized rows that triggers indexing for an information object has no effect on queries that use the information object.

You can specify the number of materialized rows that triggers indexing for all information object queries by setting the BIRT iServer configuration variable `Minimum rows to trigger creation of an index during materialize operation`. For more information about BIRT iServer configuration variables, see *Configuring BIRT iServer*.

Index

Symbols

:? operator 10
* operator 75
/ operator 75
+ operator 75
< operator 21
<> operator 21
<= operator 21
= operator 16, 21
> operator 21
>= operator 21
|| operator 28
|| operator 79
- operator 75

A

accessing
 data 2
 Encyclopedia volumes 3
 expression builder 5, 13
 information objects 56
 iServer 3
 Prompt editor 33, 44
 query editors 4, 50
accounts 3
Actuate SQL 56, 57
Actuate SQL conventions 61, 62
Actuate SQL data types 15, 71, 72
Actuate SQL expressions 4, 48, 71
Actuate SQL identifiers 67, 68
Actuate SQL join algorithms 18
Actuate SQL keywords 66, 67
Actuate SQL operators 74
Actuate SQL parameters 42
Actuate SQL queries 50, 56
 See also queries; textual queries
Actuate SQL tokens 61
ad hoc parameters 63
 See also dynamic data filters
adding
 data sets 2
 dynamic filters 32, 34

 expressions to queries 4
 filter conditions 20–30
 functions to expressions 5
 join algorithms 18, 19
 join conditions 15, 16, 17, 87
 parameters to queries 8, 42–43, 48, 57
addition operator 75
AdditiveExpression declaration 63
AdHocParameter declaration 63
Advanced Design perspective 4, 5, 9
aggregate columns 36, 37, 40, 41
aggregate expressions 63
aggregate functions 36, 60, 86, 91
Aggregate Type property 14
AggrExpression declaration 63
aliases
 column names 13, 15, 52, 68
 referencing 60
 sort operations and 59
 table names 52
AND operator 16, 28, 29, 78
AndExpression declaration 63
ANSI SQL conventions 56, 57
applyIndexing pragma 95
arithmetic operators 75
Auto suggest option 35
averages 87
AVG function 86, 87

B

Basic Design perspective 3, 4, 5
BETWEEN operator 21, 74
BIRT iServer property 3
BIRT Spreadsheet Designer 2
blank values 24, 25
Boolean values 78

C

calculations 72
cardinality (joins) 92
CARDINALITY keyword 18
CardinalityType declaration 63

- case conversion functions 78
- CASE statements 63
- CaseExpression declaration 63
- case-insensitive comparisons 73
- CAST function 72
- CAST statements 63
- CastExpression declaration 63
- casting rules 72, 73
- Category Path property 14
- CEILING function 76
- changing
 - column aliases 13
 - filter conditions 31
 - prompt properties 48
 - queries 2, 50, 51
 - source parameters 49
- CHAR_LENGTH function 79
- CHAR_LITERAL token 62
- characters
 - as decimal separators 25
 - Auto suggest option for 35
 - column aliases and 13
 - converting case 78
 - filter conditions and 23
 - handling white space 66
 - identifiers and 68
 - literal text and 62
 - parameter names and 42
 - pattern matching and 26
 - QBE expressions and 33
 - returning leftmost or rightmost 80
 - SQL identifiers and 61
 - string comparisons and 73
 - timestamps and 62
 - trimming white space 81
- closing Information Object Query Builder 2
- column aliases 13, 15, 52, 63, 68
- column categories 12, 14
- column names 16, 61, 68
 - See also* column aliases
- column subcategories 14
- ColumnAlias declaration 63
- columns
 - categorizing 12
 - comparing values across multiple 26, 27
 - defining computed. *See* computed fields
 - defining output. *See* output columns
 - displaying 6
 - grouping data and 36, 37, 38
 - removing from SELECT clause 39, 40
 - renaming 13
 - selecting 7, 11, 13
- Columns page (SQL editor) 52
- combo boxes 35
- comments 67
- comparison operators 74, 75
- comparisons
 - date-and-time values 25
 - filter conditions and 22, 24, 26
 - numeric values 25, 74
 - string values 26, 57, 73, 74, 79
- computed fields 15, 89
- concatenation 62, 79
- concatenation operator 28, 79
- Conceal Value property 46
- CondExpr declaration 63
- conditional expressions 63, 65
- ConditionalPrimary declaration 63
- connections 2, 3
- control types 35, 47
- conversions 72
- cost-based optimization (joins) 93–95, 96
- COUNT function 86, 91
- creating
 - data sets 2
 - dynamic data filters 32, 34
 - filter conditions 20–30
 - join algorithms 18, 19
 - joins 15–19, 58
 - list of values 33, 35, 45
 - queries 2, 4, 5, 6, 50
 - SQL expressions 4
 - subqueries 57, 58, 60, 70
- CURRENT_DATE function 27, 83
- CURRENT_TIMESTAMP function 83
- CURRENT_USER function 28, 87
- customizing queries 9

D

- data
 - See also* values
 - accessing 2
 - aggregating 36

- displaying 11, 53
- filtering 9, 20–35, 41, 60
- grouping 36–41
- sorting 7, 60
- data filters
 - comparing values with 22, 24, 25, 26
 - defining dynamic 32, 34
 - deleting 33
 - prompting for values and 32, 33, 34
 - setting at run time 60
 - setting conditions for. *See* filter conditions
 - setting with graphical editors 9, 22, 23, 60
 - testing 28
- Data Preview pane 11, 53
- data rows. *See* rows
- data sets 2
- data sources
 - accessing multiple information objects and 15, 18, 93
 - connecting to 2, 3
 - filtering data in 20
 - querying remote 74, 75
 - retrieving distinct values from 13, 57
- data type declarations 63, 65
- Data Type property 15, 46
- data types
 - assigning to parameters 42, 46
 - casting 72, 73
 - creating dynamic filters and 32
 - creating queries and 56, 71
 - displaying output column 52
- database views 57
- databases 20, 60, 70
- DataType declaration 63
- date stamps 83, 84, 85
- date values 25, 27
- DATEADD function 84
- DATEDIFF function 84
- DATEPART function 85
- DATESERIAL function 85
- DECIMAL data type 71, 72
- decimal precision 64, 71, 72
- decimal separators 25
- decimal values
 - aggregating data and 86
 - arithmetic operations and 75
 - converting 72
 - creating facets and 72
 - defining 71
 - rounding 77
- DECIMAL_LITERAL token 62
- default values 8, 35, 42
- deleting
 - dynamic data filters 33
 - filter conditions 31
 - information objects 11
 - join conditions 17
 - output columns 14
 - parameters 43
- dependent joins 18, 58
- derived tables 71
- Describe Query button 52
- Description property 15, 47
- designer applications 2, 56
- directory paths 33, 52, 68
- Display Control Type property 47
- Display Format property 15, 47
- Display Length property 15, 47
- Display Name property 15, 47
- display names 45
- displaying
 - column categories 12
 - columns 6
 - data 11, 53
 - error messages 9
 - information objects 6, 11
 - join conditions 16
 - parameters 6, 51
 - query output 51, 52, 53
 - report parameters 52
 - SQL statements 10
- DISTINCT keyword 60
- Distinct values only setting 13
- division 75, 76
- Do Not Prompt property 47
- documentation v
- DOUBLE data type 71, 72
- double values 71, 72, 74, 75
- DOUBLE_LITERAL token 62
- duplicate names 13
- duplicate rows 13
- dynamic data filters 32, 33, 34, 60

E

- e.Report Designer Professional 2, 3
- empty values 24
- EnableCBO pragma 93, 94
- Encyclopedia volumes 3, 6, 11, 52
- Equal to operator 21
- equality comparisons 74
- equality operator 16
- equijoins 19
- error messages 9
- errors 10
- escape characters 66, 80
- ESCAPE keyword 80
- exiting Information Object Query Builder 2
- expanded folders 11
- experts. *See* wizards
- ExplicitInnerOuterType declaration 63
- ExplicitJoinType declaration 63
- exponentiation 77
- exporting parameters 8
- expression builder 5, 12, 13
- Expression property 15
- ExpressionList declaration 63
- expressions. *See* SQL expressions

F

- facets (defined) 71
- file paths 33, 52, 68
- filter conditions
 - aggregating data and 41
 - changing 31
 - creating 20–30
 - defining multiple 28, 28–30
 - deleting 31
 - entering at runtime 60
 - excluding sets of values with 24, 25, 29
 - grouping 29
 - selecting multiple values for 24
- Filter Conditions dialog 20, 22
- filter expressions 20, 22, 27
- filter operators 21, 22, 25
- FilterClause declaration 63
- filtering
 - data 9, 20–35, 41, 60
 - error messages 10
 - string values 24

filters

- comparing values with 22, 24, 25, 26
- defining dynamic 32, 34
- deleting 33
- prompting for values and 32, 33, 34
- setting at run time 60
- setting conditions for. *See* filter conditions
- setting with graphical editors 9, 22, 23, 60
- testing 28
- FILTERS clause 10
- Filters page 22, 23, 29, 32
- FILTERS statements 56, 60, 63
- fixed point numbers 71
- floating point numbers 71, 74, 75
- FLOOR function 76
- folders 6, 11
- formula bar. *See* function signatures
- FROM clause 63
- FromClause declaration 63
- FromTableName declaration 63
- FromTableReference declaration 63
- full outer joins 16
- function signatures 5
- FunctionCallOrColumnRef declaration 63
- functions

- See also* SQL functions
- adding to expressions 5
- aggregating data and 36, 86, 91
- ANSI SQL queries and 57, 59
- subqueries and 60

G

- graphical query editors 4, 5, 50, 56
- Greater Than operator 21
- Greater Than or Equal to operator 21
- GROUP BY clause
 - creating 36, 37–38
 - entering expressions and 60
 - removing columns from 39–40
 - restricting output for 41
 - sorting data and 60
- GroupByClause declaration 64
- grouping
 - data 36–41
 - filter conditions 29

H

- Has Null property 15
- HAVING clause 41, 60, 64
- HavingClause declaration 64
- Heading property 15, 47
- Help Text property 15, 47
- hidden parameters 47
- hiding column categories 12
- hints 87
- Horizontal Alignment property 15, 47

I

- IDENTIFIER token 61
- identifiers 61, 67, 68
- illegal characters 68
- IN operator 21, 24, 75
- Indexed property 15
- indexes (SQL queries) 95, 96
- information object data source components 3
- information object data sources 15
 - See also* information objects
- Information Object Designer 60, 61
- information object names 68
- Information Object Query Builder
 - accessing expression builder in 5
 - creating joins and 16
 - creating queries and 5, 9, 10, 50
 - defining multiple conditions and 28, 29
 - defining optional tables and 88
 - exiting 2
 - filtering data and 41, 56
 - grouping data and 37, 38, 39
 - hiding column categories in 12
 - overview 2
 - prompting for values and 33
 - selecting information objects and 11
 - starting 2, 3
- information objects
 - accessing 56
 - building data sets for 2
 - building queries for. *See* queries
 - categorizing columns in 12
 - defining computed fields and 89
 - defining joins for 15–19, 93
 - defining parameters in. *See* source parameters

- deleting 11
- disabling indexing for 95
- displaying objects in 6
- displaying output for 53
- displaying parameters for 52
- filtering data in 9, 20–35, 41, 60
- optimizing for queries 87, 93
- retrieving data in 2
- selecting 6, 11
- sorting data in 7, 60
- synchronizing parameters in 49–50
- viewing 6, 11
- inherited properties 48
- inner joins 17, 59, 63
- input 32, 33, 34, 44, 48
- INTEGER data type 71, 72
- INTEGER_LITERAL token 62
- integers 62, 71, 72, 75
- Integration service 19, 60, 73, 74, 75
- intersection operations 56
- IS NOT NULL operator 21, 25, 78
- IS NULL operator 21, 25, 78
- iServer 3
- iServer Explorer 6, 12

J

- join algorithms 18, 19, 58, 95
- join conditions 15, 16, 17, 87
- join operators 16
- join types 17, 58, 63
- JoinCondition declaration 64
- JoinElement declaration 64
- JoinExpression declaration 64
- joins
 - accessing multiple data sources and 93
 - creating 15–19, 58
 - defining subqueries and 70
 - disabling cost-based optimization for 93, 94
 - optimizing 18, 19
 - setting conditions for. *See* join conditions
 - specifying cardinality of 18, 19, 92
 - specifying optional tables for 88
- Joins page 15, 16
- JoinSpec declaration 64

K

keywords (Actuate SQL) 66, 67

L

LEFT function 28, 80

LEFT OPTIONAL keywords 91

left outer joins 17, 59, 63

Length declaration 64

Less Than operator 21

Less Than or Equal to operator 21

LIKE operator 21, 26, 57, 79

line numbers 10

literal characters 26, 34, 62, 65, 66

literal numbers 62

literal strings 62, 80

local parameters 48

See also source parameters

logical operators 29, 78

LOWER function 78

LTRIM function 81

M

maps (information objects) 33, 68, 93

matching character patterns 26, 57, 79

MAX function 86

memory 19, 95, 96

merge joins 19, 59

MIN function 86

MinRowsForIndexing pragma 96

missing values 25

MOD function 76

multiline comments 67

multiplication operator 75

MultiplicativeExpression declaration 64

N

Name property 15, 47

NamedParameter declaration 64

naming

data sets 2

output columns 13

parameters 42, 47

nested loop joins 19, 58

New Data Set dialog 2

NOT BETWEEN operator 21, 24

Not Equal to operator 21

NOT IN operator 22

NOT LIKE operator 22, 24, 25

NOT operator 29, 78

null values 15, 25, 43, 78

numbers

arithmetic operations and 75

as literal characters 62

assigning to parameters 49

averaging 87

comparing 25, 74

rounding 77

setting default values and 43

numeric data types 71, 72, 74, 75

numeric functions 76

numeric tokens 62

O

ODA data sources 47

online documentation v

opening

expression builder 5, 13

Information Object Query Builder 2, 3

Prompt editor 33

operators. *See* SQL operators

optimizing

joins 18, 19

queries 87, 93

subqueries 70

OPTION clause 70

OPTIONAL keyword 88, 89, 90, 91

OR keyword 50

OR operator 29, 78

ORDER BY clause 59, 60, 64

OrderByClause declaration 64

outer joins 16, 17, 59, 63

output 53

See also result sets

output columns

defining 13–14

deleting 14

displaying 51, 52, 53

naming 13

setting character lengths for 15

setting order of 14

setting properties for 14, 52

P

- Parameter Mode property 47
- parameter passing conventions 58
- Parameter Values dialog 53
- ParameterDeclaration declaration 64
- parameterized queries 57, 60
- parameterized tables 57
- parameters
 - adding to queries 8, 42–43, 48, 57
 - assigning data types to 42, 46
 - assigning null values to 43
 - assigning to parameters 49
 - changing properties for 48
 - defining in information objects. *See* source parameters
 - deleting 43
 - displaying query output and 53
 - exporting 8
 - filtering data and. *See* dynamic data filters
 - hiding 47
 - naming 42, 47
 - prompting for values and 33, 44, 48
 - setting facets for 72
 - setting properties for 44, 46
 - setting values for 8, 42, 48, 68, 69
 - specifying required 47
 - viewing 6, 51, 52
- Parameters page (Query Design) 43, 44, 50
- Parameters page (SQL editor) 52
- ParamPlaceholder declaration 64
- Password property 3
- paths 33, 52, 68
- pattern matching 26, 57, 79
- performance 18, 19, 20
- Port number property 3
- POSITION function 82
- POWER function 77
- Pragma declaration 62, 64
- pragmas 59, 93
- precision 71, 72
- Precision declaration 64
- predefined data filters 32
- PrimaryExpression declaration 64
- Problems pane 9
- Progress pane 11
- Prompt editor 33, 34, 44

- prompting for values 32, 33, 34, 44, 48
- properties

- cost-based optimization and 96
 - data source connections 2, 3
 - dynamic data filters 34
 - inheriting 48
 - output columns 14, 52
 - parameters 44, 46

Q

- QBE expressions 33, 35
- QBE syntax 32
- queries
 - See also* SQL statements; textual queries
 - accessing multiple information objects and 15, 18, 93
 - accessing remote data sources and 74, 75
 - adding parameters to 8, 42–43, 48, 57
 - building data sets for 2
 - changing 2, 50, 51
 - converting column names for 16
 - copying 50
 - creating 2, 4, 5, 6, 50
 - customizing 9
 - defining derived tables and 71
 - defining optional tables for 88
 - defining output columns for 13–14
 - disabling cost-based optimization for 93, 94
 - optimizing 18, 19, 87, 93
 - prompting for values and 33, 34, 35, 44, 45
 - referencing aliases in 60
 - referencing information objects in 68
 - referencing tables or views in 57
 - removing parameters from 43
 - restricting number of rows in 20, 41
 - returning duplicate rows and 13
 - running Integration service and 60
 - saving 9, 50
 - setting dynamic filters and 10
 - unknown data types in 15
 - validating 10, 29, 33
 - viewing columns selected for 7
 - viewing errors with 9, 10
 - viewing output from 51, 52, 53

Query Builder. *See* Information Object Query Builder
Query Design 4
query editors 4, 5
query operators. *See* SQL operators
QueryParameterDeclaration declaration 64

R

range of values 74
range test operator 74
RelationalOperator declaration 64
remainders 76
remote data sources 74, 75
report designer applications 2, 56
report parameters. *See* parameters
report wizard 3
Required property 47
reserved words (Actuate SQL) 66, 67
result sets
 See also queries
 changing column order in 14
 defining multiple conditions for 28, 28–30
 defining output columns for 13–14
 excluding duplicate rows from 13
 generating computed fields for 15
 handling null values in 15
 missing values in 42
 previewing data in 11, 53
 removing output columns from 14
 restricting number of rows in 20, 41
 returning distinct values for 13, 57
 returning scalar values in 58
 viewing output columns in 51, 52, 53
RIGHT function 80
RIGHT OPTIONAL keywords 90, 92
right outer joins 16
ROUND function 77
rows
 defining multiple conditions for 29
 disabling indexing for 95
 excluding duplicate 13
 previewing 11, 53
 restricting number returned 20, 41
 specifying threshold values for 96
RTRIM function 81

S

saving queries 9, 50
scalar subqueries 58
scalar values 58, 71, 75
ScalarDataType declaration 65
Scale declaration 65
search function 82
SELECT clause 65
SELECT statements
 See also SQL statements
 adding expressions to 5
 adding parameters to 57
 adding subqueries to 58, 70
 defining derived tables and 71
 defining dynamic filters for 60
 defining joins and 58, 59
 disabling automatic grouping and 40
 dynamically filtering 56
 grouping data and 37, 38, 40, 41, 60
 removing columns from 39, 40
 sorting data and 60
SelectItem declaration 65
SelectList declaration 65
SelectStatement declaration 65
SelectWithoutFrom declaration 65
SelectWithoutOrder declaration 65
serial values 85
ServerUri property 3
set difference operations 56
SetClause declaration 65
SignedLiteral declaration 65
SimpleCondition declaration 65
SINGLE EXEC keywords 70
Size property 47
sort order 8
sorting data 7, 60
source parameters 48–50
space characters 62, 66, 68, 81
special characters. *See* characters
spreadsheet designer 2
SQL conventions 56
 See also Actuate SQL
SQL editor. *See* textual query editor
SQL Editor button 4, 50
SQL expressions
 See also QBE expressions

- adding 4
- comparing values and 73, 74, 75
- converting column names to 16
- counting non-null values and 91
- creating joins and 17
- defining facets for 72
- defining GROUP BY items and 60
- defining output columns and 13
- entering characters in 23
- entering functions in. *See* functions
- entering source parameters in 48
- filtering data and 20, 22, 27, 32
- generating computed fields and 15
- SQL functions
 - aggregation. *See* aggregate functions
 - numeric values and 76
 - string values and 78
 - substrings and 80, 82
 - system information and 87
 - timestamp values and 82
- SQL operator reference 74
- SQL operators
 - ANSI SQL conventions and 57
 - filter conditions 21, 22, 25
 - joins 16
 - subqueries and 70
- SQL parameters 42
- SQL Preview pane 10, 51
- SQL statements
 - See also* queries
 - adding expressions to. *See* SQL expressions
 - adding subqueries to. *See* subqueries
 - defining joins with 15–19, 58
 - defining multiple conditions in 28, 28–30
 - displaying 10
 - entering manually. *See* textual queries
 - filtering data with 9, 20–35, 41, 56, 60
 - grouping data with 36–41, 60
 - returning distinct values for 13, 57
 - sorting data with 7, 59
- SQL text editor. *See* textual query editor
- SQL-92 keywords 67
- starting Query Builder 2, 3
- stored procedures 47
- string data types 71, 72
- string functions 78, 80, 81, 82
- string operators 78
- string token 62
- strings
 - assigning to parameters 49
 - casting rules for 72
 - comparing 73, 74
 - comparing patterns in 26, 57, 79
 - concatenating 28, 62, 79
 - converting case 78
 - creating QBE expressions and 34
 - defining facets and 72
 - getting length of 79
 - returning substrings in 80, 82
 - setting default values and 43
 - setting maximum length for 71
 - testing for blank values in 25
 - trimming white space in 81
- subqueries 57, 58, 60, 70
- SubQuery declaration 66
- SUBSTRING function 80
- substring functions 80
- substrings 80, 82
- subtraction operator 75
- SUM function 86, 87, 91
- synchronizing source parameters 49–50
- syntax errors 10

T

- table names 52, 61, 63, 68
- TableParameter declaration 66
- TableParameters declaration 66
- tables 57, 71, 88
- text 62
- text boxes 34
- Text Format property 15
- textual queries
 - creating 50–52
 - displaying output columns for 52
 - displaying parameters for 52
 - filtering data with 10, 23, 29
 - prompting for values and 33, 35
 - saving 50
- textual query editor 4, 5, 50, 51, 56
- time 25
- TIMESTAMP data type 71, 72
- timestamp functions 82
- TIMESTAMP_STRING token 62

- timestamps
 - casting rules for 72
 - comparing 74
 - defining 49, 62
 - defining facets and 72
 - returning current 83
 - setting default values and 43
- tokens 61
- trailing spaces 26
- TRIM function 81
- truncated messages 9
- truncated numeric values 72
- type casting 72, 73
- types. *See* data types

U

- UnaryExpression declaration 66
- UnaryLogicalExpression declaration 66
- UNION keyword 50
- UNION statements 56, 65
- unknown data types 15
- unnamed parameters 57
- UnsignedLiteral declaration 66
- updating Encyclopedia volumes 11
- UPPER function 78
- URIs 3
- URLs 3
- user accounts 3
- User name property 3
- user names 3, 28, 87

V

- value expressions 63, 66
- ValueExpression declaration 66
- values
 - See also* data
 - assigning to parameters 42, 43, 48, 68, 69
 - averaging 87
 - comparing. *See* comparisons
 - counting non-null 86, 91
 - creating list of 33, 35, 45

- filtering empty or blank 24, 25
- filtering on multiple 24, 26, 27, 28
- hiding 46
- prompting for 32, 33, 34, 44, 48
- returning distinct 13, 57
- returning largest 76
- returning smallest 76
- rounding 77
- selecting 35
- setting control types for 35
- setting default 8, 35, 42
- testing for null 15, 25, 78
- testing range of 74
- testing sets of 75
- ValueSelectItem declaration 66
- ValueSelectList declaration 66
- VARCHAR data type 71, 72
- variant data 47, 72
- viewing
 - column categories 12
 - columns 6
 - data 11, 53
 - error messages 9
 - information objects 6, 11
 - join conditions 16
 - parameters 6, 51
 - query output 51, 52, 53
 - report parameters 52
 - SQL statements 10
- views 57
- Volume property 3

W

- WHEN clause 66
- WhenClause declaration 66
- WHERE clause 10, 20, 60, 66
- WhereClause declaration 66
- white space characters. *See* space characters
- WITH clause 42, 57, 64
- wizards 3
- Word Wrap property 15