MarkLogic Server

SQL Data Modeling Guide

MarkLogic 8
February, 2015

Last Revised: 8.0-1, February, 2015

Table of Contents

SQL Data Modeling Guide

1.0 SQL on MarkLogic Server

The views module is used to create and manage SQL schemas and views.

The main topics in this chapter are:

1.1 Terms Used in this Guide

The following are the definitions for the terms used in this guide:

- A *view* is a representation of a SQL view. A view is implemented as an XML document in the schemas database and consists of a unique id, a name (which must be unique in the context of a particular schema), a view scope, and a sequence of column specifications. A view may also include fields that allow you to do more precise queries with the MATCH operator, as demonstrated in "Using MLSQL" on page 26 and "MATCH Operator" on page 80.
- A *schema* is a representation of a SQL schema. A schema is implemented as an XML document in the schemas database and consists of a unique id, a name (which must also be unique), and a collection of views. During SQL execution, the schema provides the naming context for its views, which enables you to have multiple views of the same name in different schemas. The default schema is called "main." It is default is the sense that it is always implicitly available and first on the default schema search path for name resolution in SQL. Even though the "main" schema is a default, you must create this schema.
- A *column* in a view has a name and a range index reference that identifies a particular document element or attribute. The range index for each column must be created before creating the view.
- A *view scope* is used to constrain the subset of the database to which the view applies. A view scope can either limit rows in the view to documents with a specific element (localname + namespace) or to documents in a particular collection.

  Note: You must have the `view-admin` role to execute the functions in the View library.

1.2 How SQL is Implemented

The SQL supported by the core SQL engine is SQL92 as implemented in SQLITE with the addition of SET, SHOW, and DESCRIBE statements.

For details on SQLITE, see: http://sqlite.org/index.html.

1.3 Schemas and Views

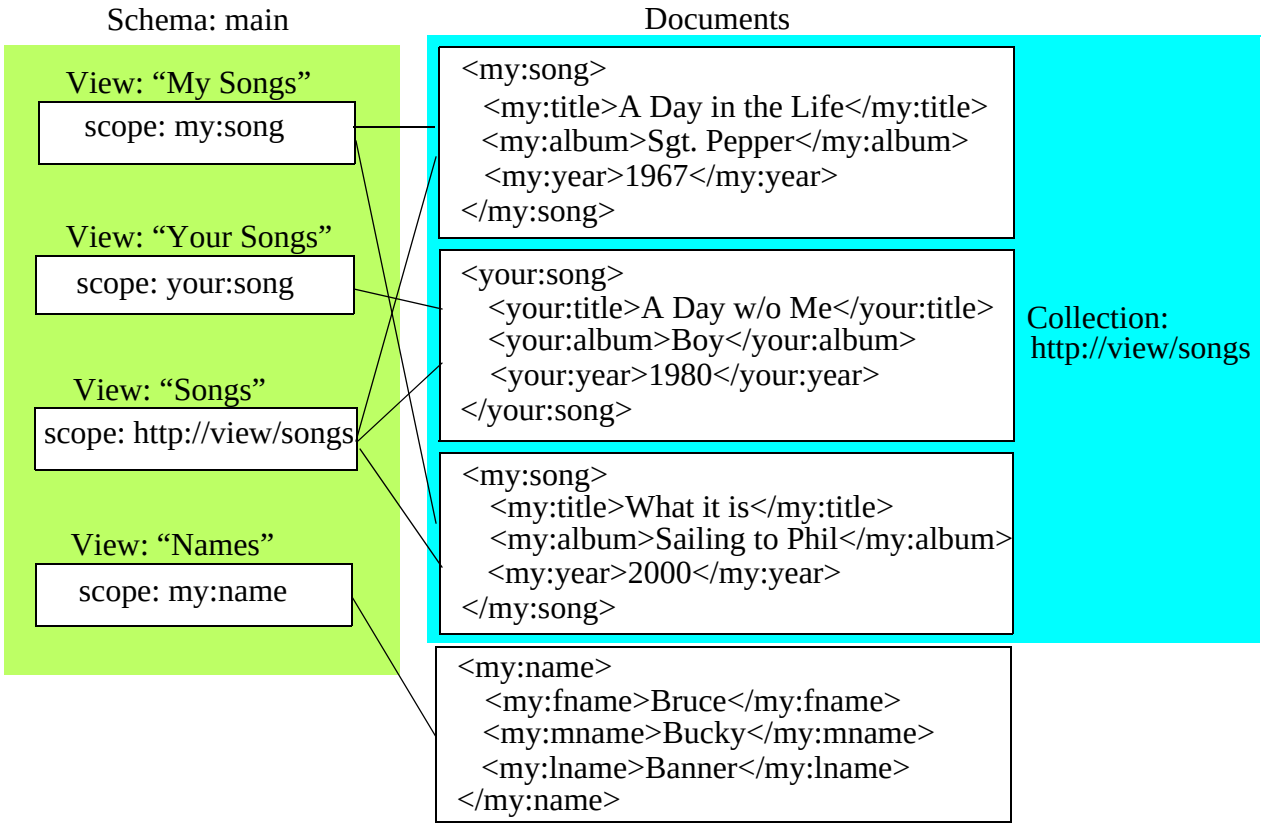
Schemas and views are the main SQL data-modeling components used to represent content stored in a MarkLogic Server database to SQL clients. Schemas and views are created in memory from schema and view *specifications*, which are XML documents stored on MarkLogic Server in the Schemas database in a protected collection.

A schema is a naming context for a set of views and user access to each schema can be controlled with a different set of permissions. Each view in a schema must have a unique name. However, you can have multiple views of the same name in different schemas. For example, you can have three views, named ‘Songs,’ each in a different schema with different protection settings.

A view is a virtual read-only table that represents data stored in a MarkLogic Server database. Each column in a view is based on a range index in the content database, as described in “Columns and Range Indexes” on page 8. User access to each view is controlled by a set of permissions.

Each view has a specific scope that defines the documents from which it reads the column data. The view scope constrains the view to a specific element in the documents (localname + namespace) or to documents in a particular collection. The figure below shows a schema called 'main' that contains four views, each with a different view scope. The view "My Songs" is constrained to documents that have a `song` element in the `my` namespace; the view "Your Songs" is constrained to documents that have a `song` element in the `your` namespace; the view "Songs" is constrained to documents that are in the `http://view/songs` collection, and the view "Names" is constrained to documents that have a `name` element in the `my` namespace.

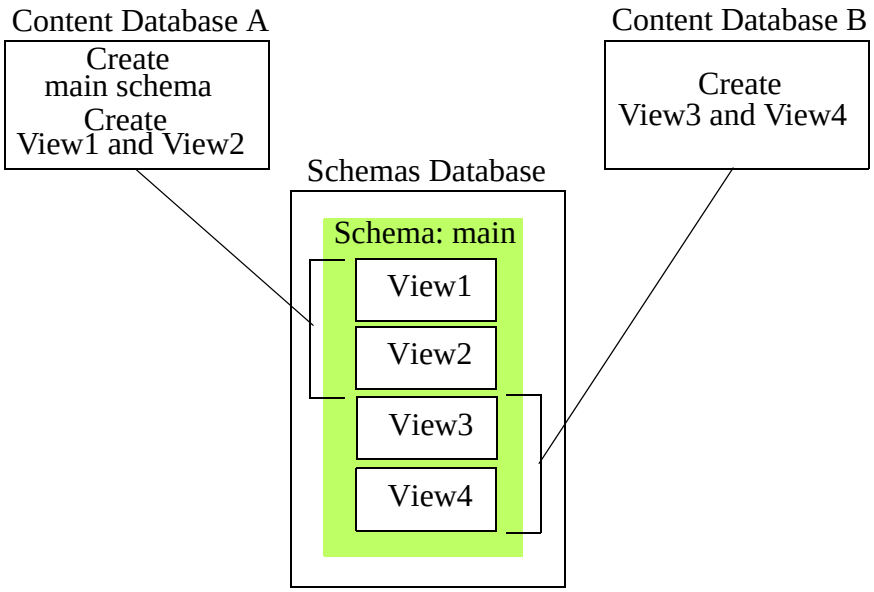
Note: You can set the scope of a view to any element in the documents, whether it is the document root element or a descendant of the root element.



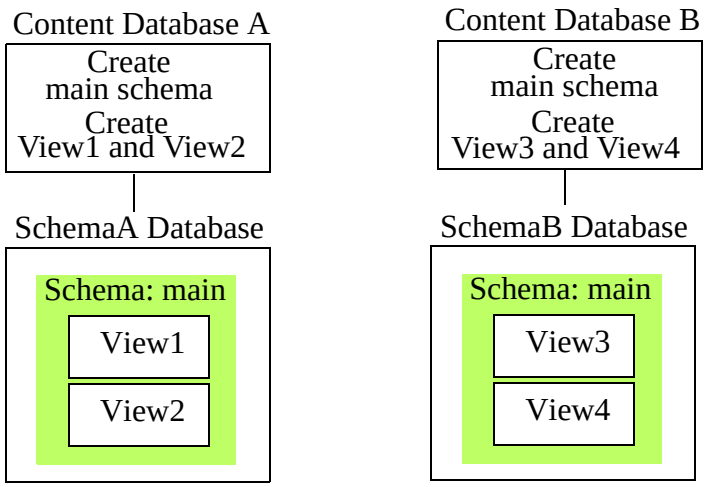
As described above, schemas and views are stored as documents in the schema database associated with the content database for which they are defined. The default schema database is named 'Schemas.' If multiple content databases share a single schema database, each content database will have access to all of the views in the schema database.

For example, in the figure below, you have two content databases, Database A and Database B, that both make use of the Schemas database. In this example, you create a single schema, named ‘main,’ that contains two views, View1 and View2, on Database A. You then create two views, View3 and View4, on Database B and place them into the ‘main’ schema. In this situation, both Database A and Database B will each have access to all four views in the ‘main’ schema.

Note: The range indexes that back the columns defined in views 1, 2, 3, and 4 have to be defined in both content databases A and B for the views to work. You will get a runtime error if you attempt to use a view that contains a column based on a non-existent range index.



A more “relational” configuration is to assign a separate schema database to each content database. In the figure below, Database A and Database B each have a separate schema database, SchemaA and SchemaB, respectively. In this example, you create a ‘main’ schema for each content database, each of which contains the views to be used for its respective content database.



1.4 Representation of SQL Components in MarkLogic Server

This section provides an overview of mapping from MarkLogic Server to SQL The table below lists SQL components and how each is represented in MarkLogic Server:

SQL	MarkLogic	Configuration
Column	A value in range index	column spec
Row	A sequence of range index values over the same document	view spec columns
Table/view	A document element or collection of documents	view spec
Database	A logical database	schema spec

There are two basic approaches for representing document data stored in MarkLogic Server:

- Configure range indexes and views so that you can execute SQL queries on unstructured document data. An example of this approach is provided in “Data Modeling Example” on page 39.
- Make your document data more structured and relational so that SQL queries behave the way they would in an relational database. An example of this approach is provided in “SQL on MarkLogic Server Quick Start” on page 9.

1.4.1 Columns and Range Indexes

Each column in a view is based on a range index in the content database. Range indexes are described in the Range Indexes and Lexicons chapter in the *Administrator's Guide*. This section provides examples of what type of range index you might use to store your column data.

Consider a document of the following form:

```
<book>
   <title subject="oceanography">Sea Creatures</title>
   <pubyear>2011</pubyear>
   <keyword>science</keyword>
   <author>
       <name>Jane Smith</name>
       <university>Wossamotta U</university>
   </author>
   <body>
       <name type="cephalopod">Squid</name>
           Fascinating squid facts...
       <name type="scombridae">Tuna</name>
           Fascinating tuna facts...
       <name type="echinoderm">Starfish</name>
           Fascinating starfish facts...
   </body>
</book>
```

You can create columns based on an element range indexes for the `title`, `pubyear`, `keyword`, `author`, and `university` elements without violating any of the "relational behavior" rules listed in "Guidelines for Relational Behavior" on page 44. Creating a column based on an element range index for the `name` element would violate the relational rules. However, you could use a path range index to create a column for the `/book/author/name` element without violating the relational rules. You might also want to create a column based on an attribute range index for the `subject` attribute in the `title` element.

You may chose to model your data so that it is not truly relational. In this case, you could create columns based on a path range index for the `book/body/name` element and `book/body/name/@type` attribute.

1.4.2 Searchable Fields

In MarkLogic Server, fields allow you to narrow searches to specific elements. Like the range indexes, fields can be bound to a view. Fields allow you to do more precise queries with the MATCH operator, as demonstrated in "SQL on MarkLogic Server Quick Start" on page 9.

2.0 SQL on MarkLogic Server Quick Start

This chapter describes how to set up your MarkLogic Server for SQL. This chapter describes how to set up a typical development environment in which the SQL client and MarkLogic Server are configured on the same machine. For a production environment, you would typically configure your SQL client and MarkLogic Server on separate machines.

> **Note:** You must have the admin role on MarkLogic Server to complete the procedures described in this chapter.

The main topics in this chapter are:

- Setup MarkLogic Server
- Create Views
- Load the Data
- Enter SQL Queries to Test
- Using MLSQL

2.1 Setup MarkLogic Server

Install MarkLogic Server on the database server, as described in the *Installation Guide*. and follow these procedures:

- Create a Schema Database and a SQL Database
- Create Range Indexes for the Database
- Create a Field for the Database
- Create an ODBC App Server

2.1.1 Create a Schema Database and a SQL Database

How to create a database is described in detail in Creating a New Database in the *Administrator's Guide*. This section provides a quick-start procedure for creating the database used in this example.

Warning Every SQL database must have its own separate schema database.

1. Open your browser and navigate to the Admin Interface:

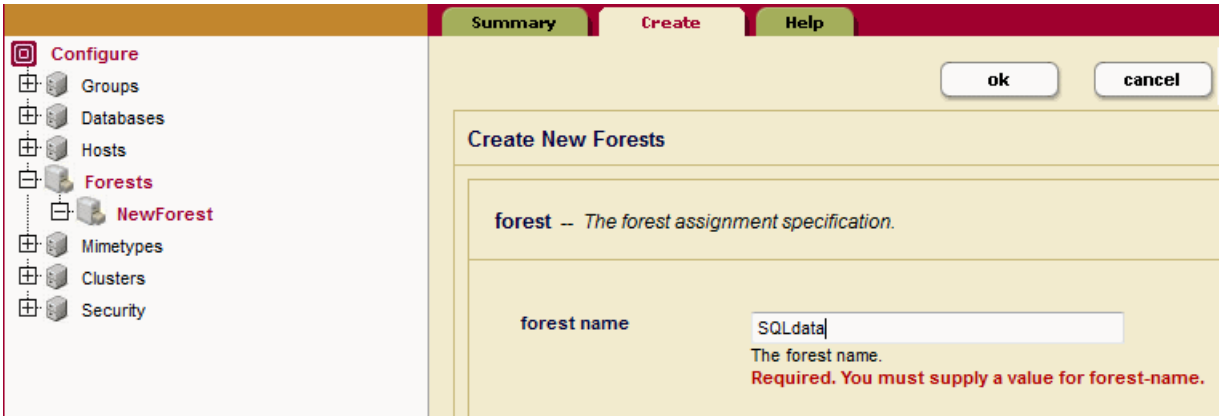
   ```
   http://hostname:8001
   ```

   Where *hostname* is the name of your MarkLogic Server host machine.

2. Click the Forests icon in the left frame.

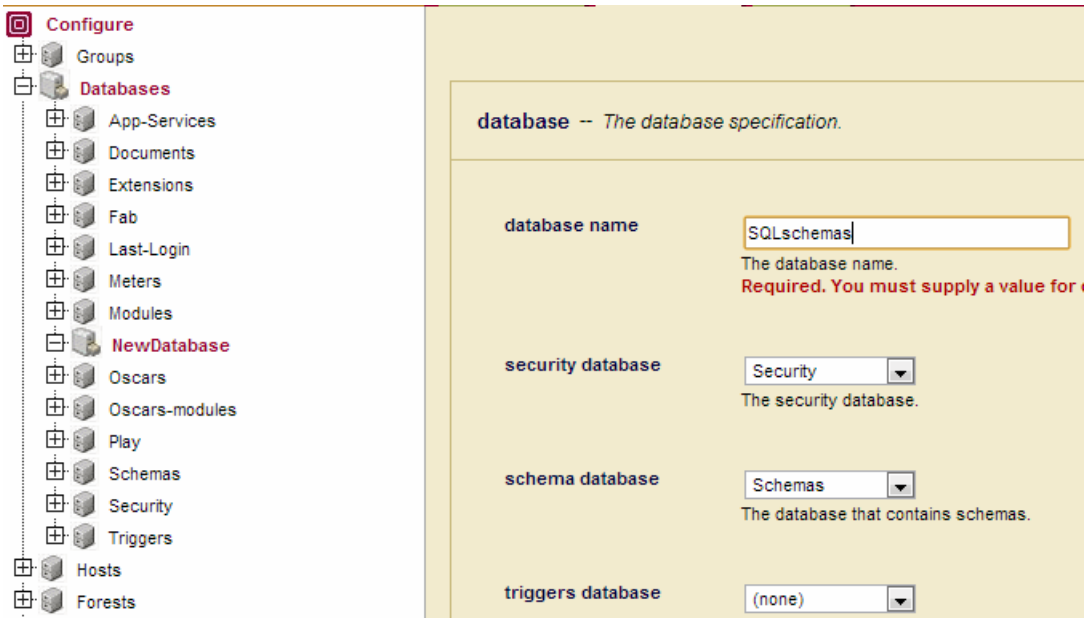
3. Click the Create tab at the top right. The Create Forest page displays. Enter ‘SQLschemas’ as the name of your forest in the Forest Name textbox. Click OK.

4. Click the Create tab at the top right. The Create Forest page displays. Enter ‘SQLdata’ as the name of your forest in the Forest Name textbox. Click OK.



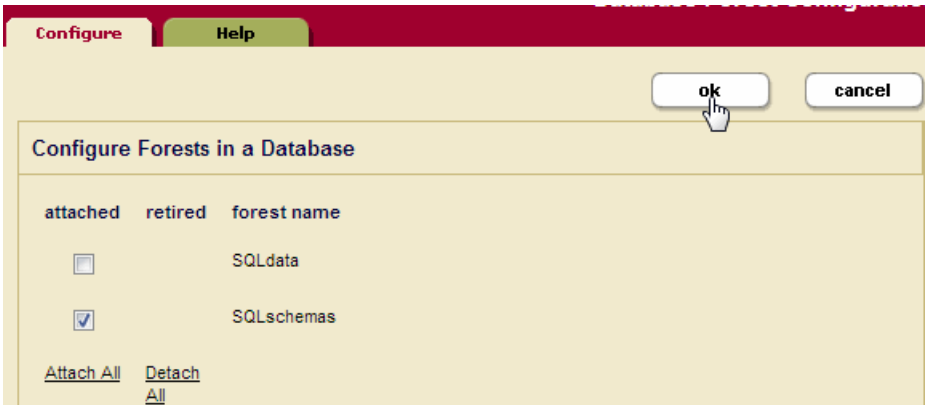
5. Click the Databases icon in the left tree menu.

6. Click the Create tab at the top right. The Create Database page displays. Enter 'SQLschemas' as the name of the new database and click Ok:

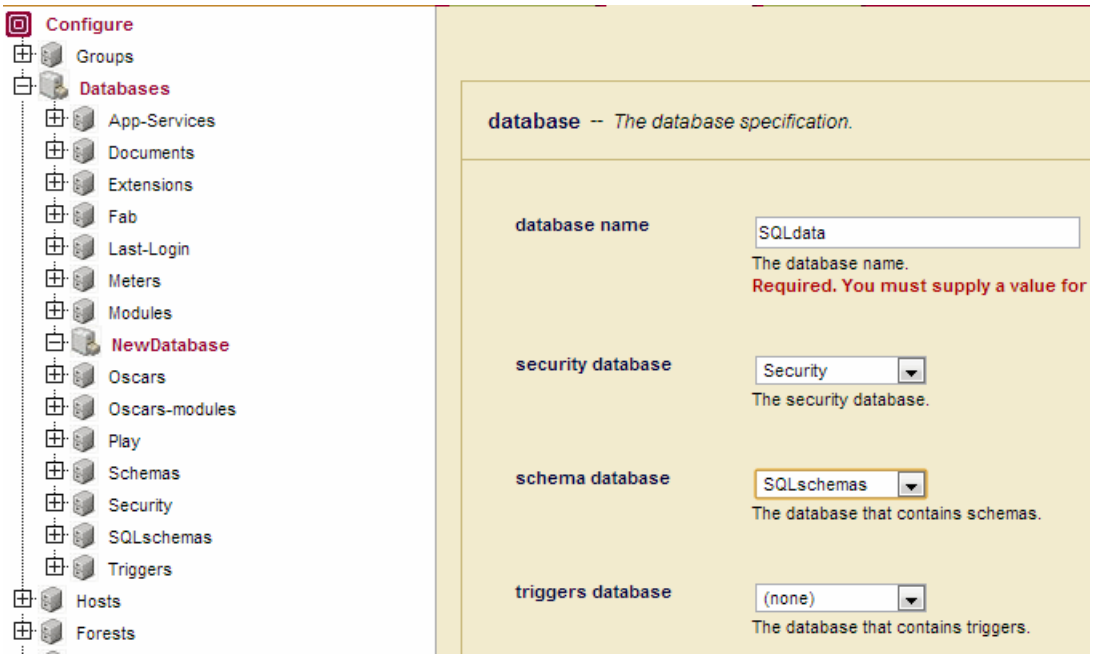


7. At the top of the page click Database->Forests

8. Check the SQLschemas box to attach the SQLschemas forest. Click Ok:



9. Click the Create tab at the top right. The Create Database page displays. Enter ‘SQLdata’ as the name of the new database and select ‘SQLschemas’ as the Schema Database. Click Ok:



10. At the top of the page click Database->Forests

11. Check the SQLdata box to attach the SQLdata forest. Click Ok:

2.1.2 Create Range Indexes for the Database

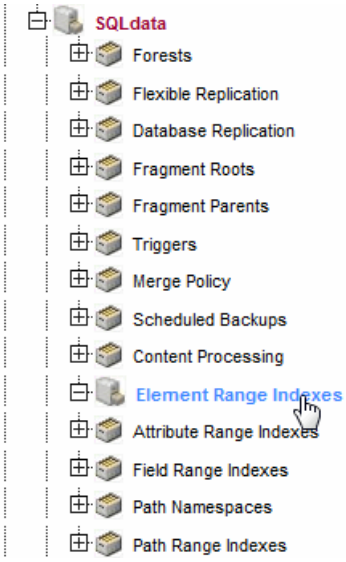
The sample JSON documents modeled in this chapter take the following form:

```
{ "Employee": {
      "EmployeeID": 1,
      "FirstName": "John",
      "LastName": "Widget",
      "Position": "Manager of Human Resources" }}

{ "Expenses": {
      "EmployeeID": 2,
      "Date": "2012-06-27",
      "Amount": 131.02}}
```

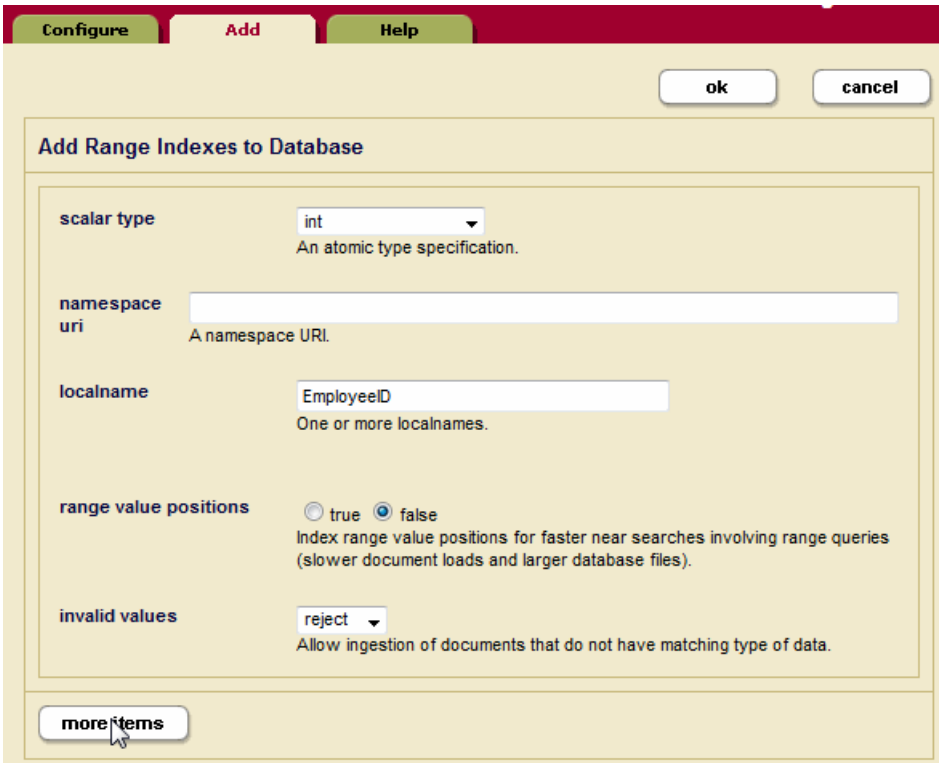
In order to model such documents as SQL tables, we need to create range indexes for each "column" (`EmployeeID`, `FirstName`, `LastName`, `Date`, and `Amount`).

> **Note:** In the following section, "Create a Field for the Database" on page 15, you will create a field for the `Position` element.

1. Click the Element Range Indexes icon in the tree menu, under the 'SQLdata' database.



2. Click the Add tab. The Element Range Index Configuration page or the Element Word Lexicon Configuration page displays. Set the Scalar Type to `int` and enter `EmployeeID` in the localname field to create a range index for the `EmployeeID` element. Click More Items.



3. Create element range indexes for the remaining elements, as shown in the following table. Leave all other range index fields empty or unchanged with their default settings.

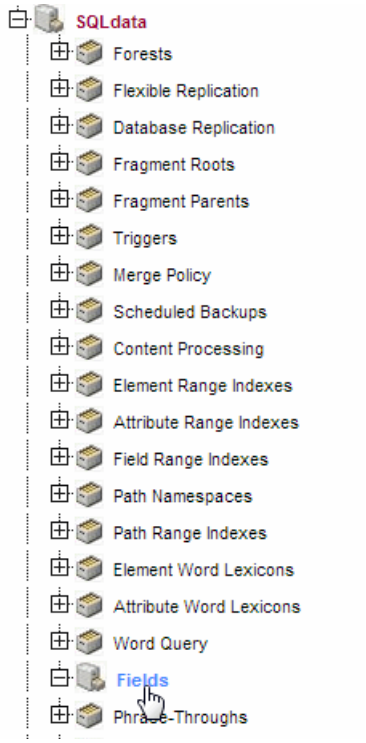
Local Name	Scalar Type
`FirstName`	`string`
`LastName`	`string`
`Date`	`date`
`Amount`	`decimal`

4. Click Ok when finished.

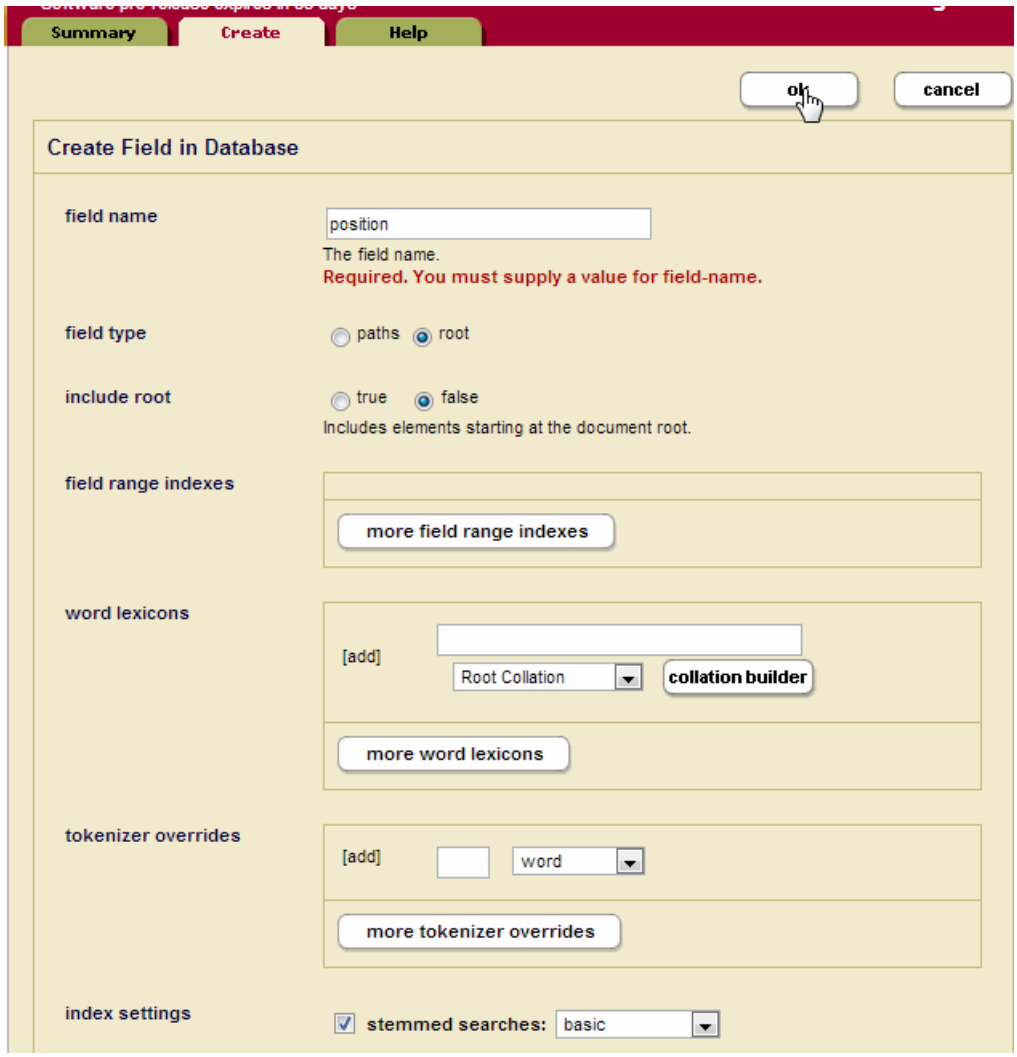
2.1.3 Create a Field for the Database

In MarkLogic Server, fields allow you to narrow searches to specific elements. Like the range indexes created in “Create Range Indexes for the Database” on page 13, the field created in this section will be bound to a view. Fields allow you to do more precise searches with the MATCH operator, as demonstrated in “Using MLSQL” on page 26 and “MATCH Operator” on page 80.

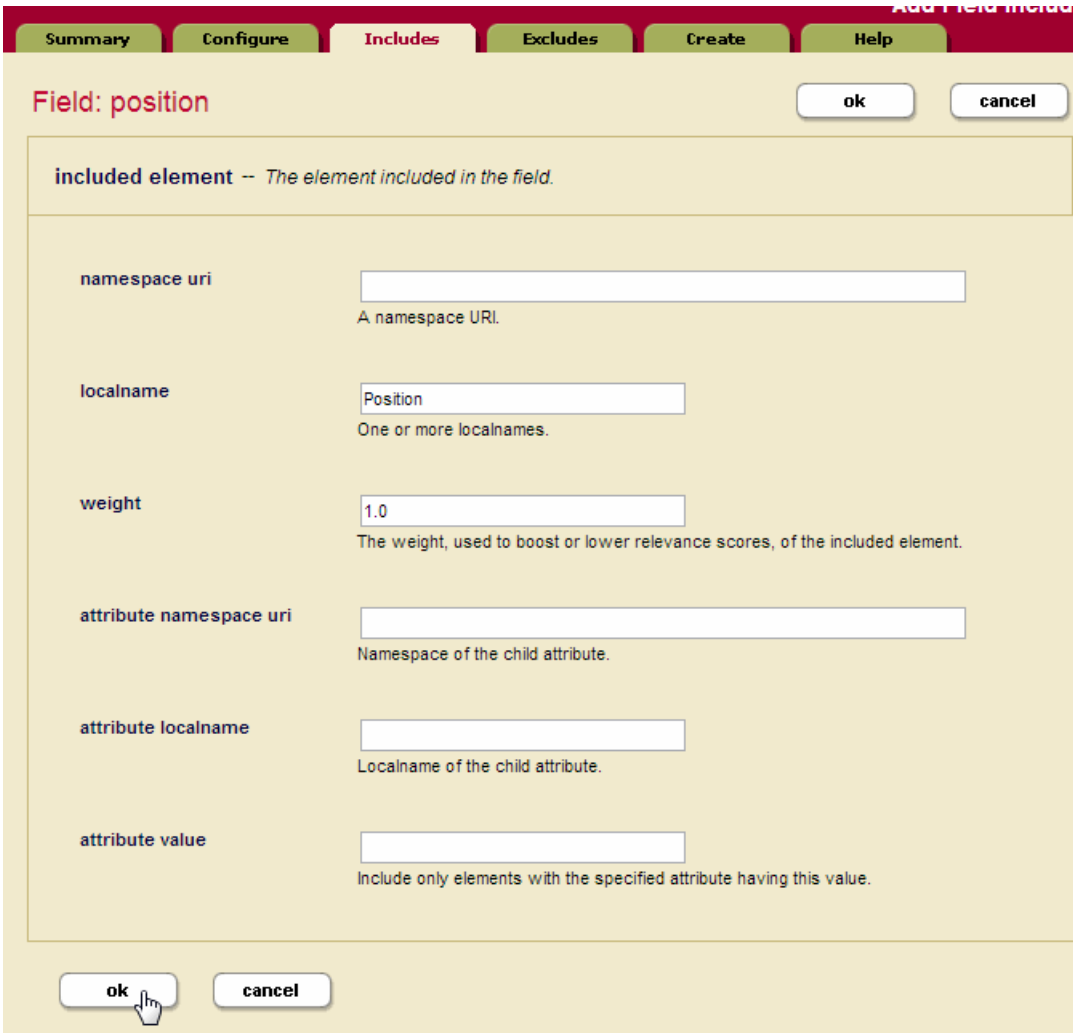
1. Click the Fields icon in the tree menu, under the ‘SQLdata’ database.



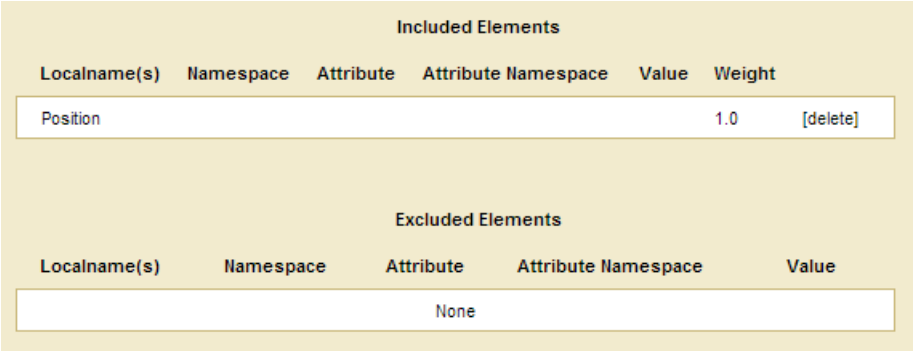
2. Click the Create tab. The Database Fields Configuration page displays. Select a field type of `root` and enter `position` for the field name. Leave all other settings as the defaults. Click Ok.



3. The tabs will expand to display additional tabs. Click on the Includes tab and enter `Position` in the localname field. Click Ok.



4. The bottom section of the Database Fields Configuration page for the `position` field will now display that it includes the `Position` field:



2.1.4 Create an ODBC App Server

Schemas and views represent content stored in a MarkLogic Server database. Each content database used by a SQL client is managed by an ODBC App Server that accepts SQL queries from the SQL client and responds by returning MarkLogic Server data in tuple form. An ODBC App Server can manage only one content database. However, a single content database can be managed by multiple ODBC App Servers.

ODBC App Servers are described in detail in the ODBC Servers chapter in the *Administrator's Guide*.

Open the Admin Interface

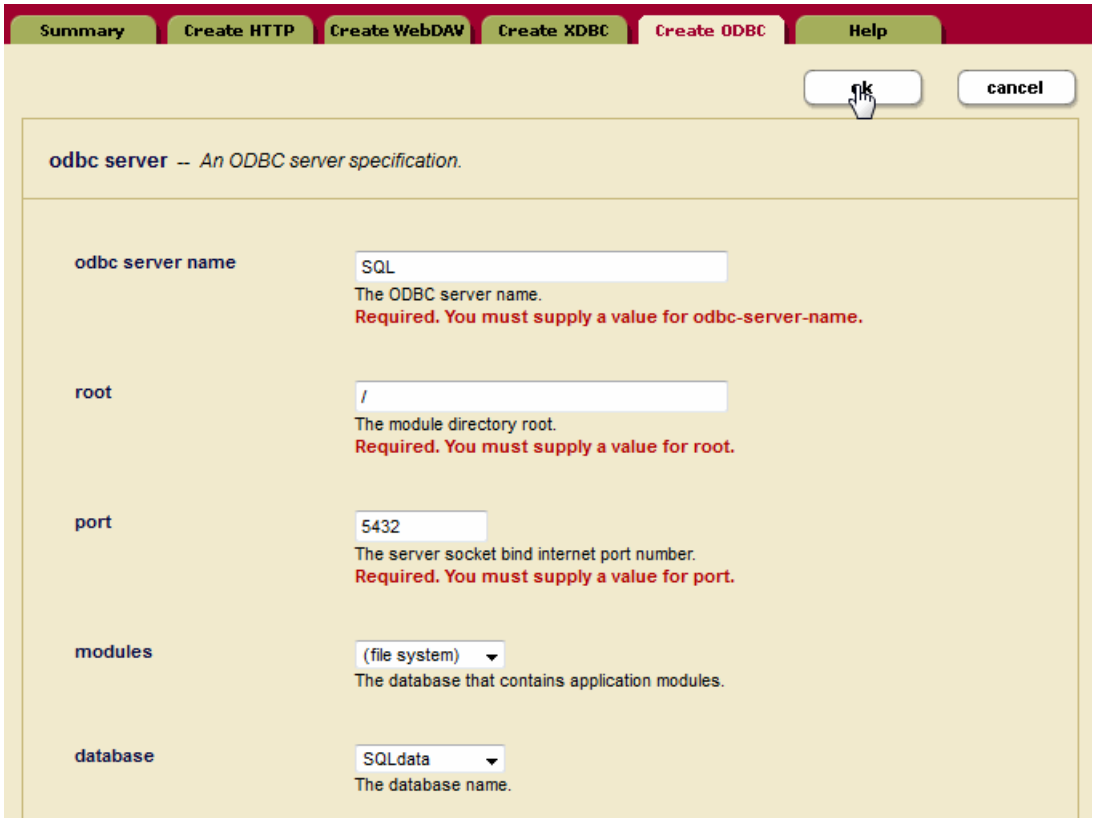
To create a new server, complete the following steps:

1. Click the Groups icon in the left frame.

2. Click the group in which you want to define the ODBC server (for example, Default).

3. Click the App Servers icon on the left tree menu.

4. Click the Create ODBC tab at the top right. The Create ODBC Server page will display:



5. In the Server Name field, enter a shorthand name for this ODBC server. In this example, the name of the App Server is ‘SQL.’

6. In the Root directory field, enter /.

7. In the Port field, enter the port number through which you want to make this ODBC server available. The default PostgreSQL listening socket port is 5432.

8. Leave the Modules field as `(file system)`.

9. In the Database field, select the ‘SQLdata’ database you created in “Create a Schema Database and a SQL Database” on page 9.

2.2 Create Views

This section describes how to use the REST management API to create the views used by SQL queries.

1. Use `POST:/manage/v2/databases/{id|name}/view-schemas` to create a schema, named ‘main’.

```
curl -X POST  --anyauth --user admin:admin \
--header "Content-Type:application/json" -d '{"view-schema-name": "main"}' \
http://gordon-2:8002/manage/v2/databases/SQLdata/view-schemas?format=json
```

2. Use `POST:/manage/v2/databases/{id|name}/view-schemas/{schema-name}/views` to create a view in the ‘main’ schema, named ‘employees’. Specify a scope on the ‘Employee’ element and columns for the ‘firstname’, ‘lastname’, and ‘employeeid’ range indexes. Specify the ‘position’ field as a searchable field.

```
curl -X POST  --anyauth --user admin:admin \
--header "Content-Type:application/json" \
-d '{
     "view-name": "employees",
     "element-scope":{"namespace-uri":"", "localname":"Employee"},
     "column": [
       {
         "column-name": "employeeid",
         "element-reference": {
            "namespace-uri": "",
            "localname": "EmployeeID",
            "scalar-type": "int"
         }
       },
       {
         "column-name": "firstname",
         "element-reference": {
            "namespace-uri": "",
            "localname": "FirstName",
            "scalar-type": "string",
            "collation": "http://marklogic.com/collation/"
         }
       },
       {
         "column-name": "lastname",
         "element-reference": {
            "namespace-uri": "",
            "localname": "LastName",
            "scalar-type": "string",
            "collation": "http://marklogic.com/collation/"
         }
       }
     ],
     "field": [
            {"field-name": "position"}
     ]
   }' \
http://gordon-2:8002/manage/v2/databases/SQLdata/view-schemas/main/views?forma
t=json
```

3. Use `POST:/manage/v2/databases/{id|name}/view-schemas/{schema-name}/views` to create a second view in the ‘main’ schema, named ‘expenses’, with a scope on the ‘Expenses’ element and columns for the ‘EmployeeID’, ‘Date’, and ‘Amount’ range indexes.

```
curl -X POST  --anyauth --user admin:admin \
--header "Content-Type:application/json" \
-d '{
     "view-name": "expenses",
     "element-scope":{"namespace-uri":"", "localname":"Expenses"},
     "column": [
       {
         "column-name": "employeeid",
         "element-reference": {
            "namespace-uri": "",
            "localname": "EmployeeID",
            "scalar-type": "int"
         }
       },
       {
         "column-name": "date",
         "element-reference": {
            "namespace-uri": "",
            "localname": "Date",
            "scalar-type": "date"
         }
       },
       {
         "column-name": "amount",
         "element-reference": {
            "namespace-uri": "",
            "localname": "Amount",
            "scalar-type": "decimal"
         }
       }
     ]
   }' \
http://gordon-2:8002/manage/v2/databases/SQLdata/view-schemas/main/views?forma
t=json
```

4. List the views in the ‘main’ schema.

```
curl -v -X GET --anyauth -u admin:admin \
--header "Content-Type:application/json" \
http://gordon-2:8002/manage/v2/databases/SQLdata/view-schemas/main/views?forma
t=json
```

Note: If you change a range index used by a column, you need to delete and recreate the view in order for the view to use the new index.

2.3 Load the Data

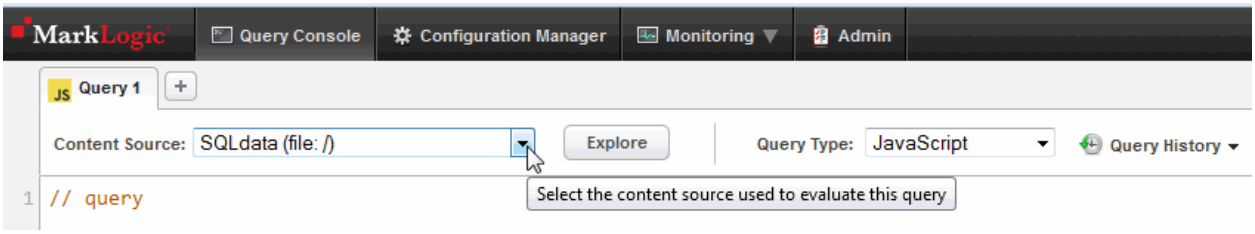
This section describes the procedure for loading the sample documents.

1. Go to the following URL to open Query Console:

   ```
   http://hostname:8000/qconsole/
   ```

   Where *hostname* is the name of your MarkLogic Server host.

2. Select the SQLdata database from the Content Source pulldown menu and JavaScript from the Query Type menu.

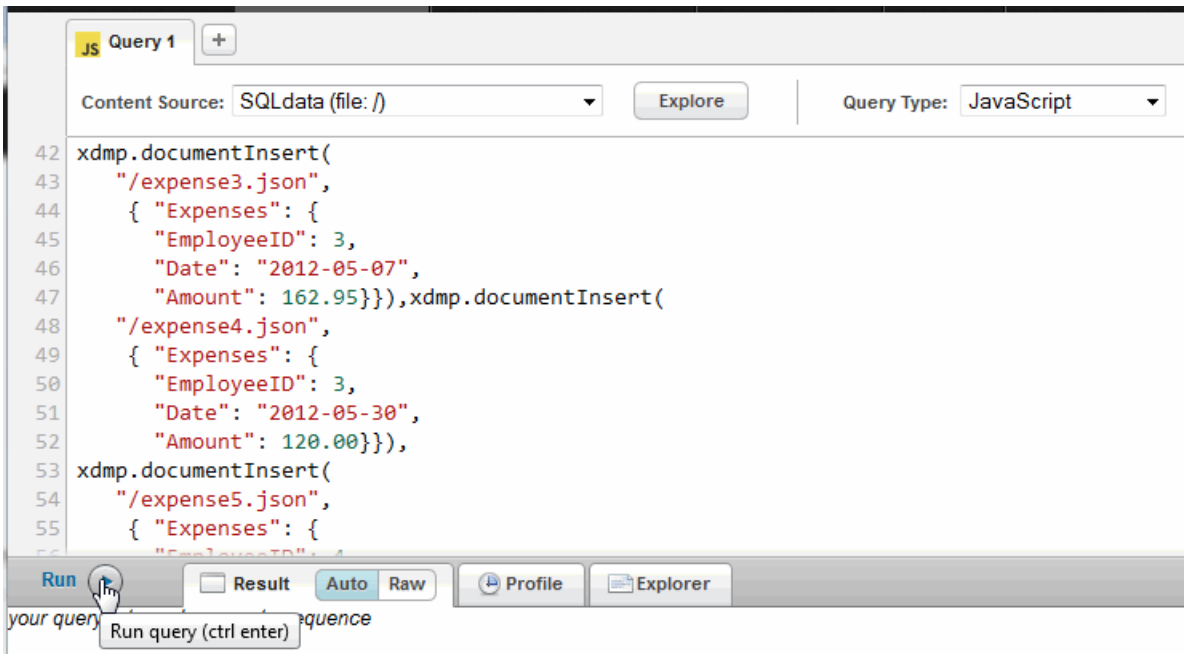


3. Cut and paste the following JavaScript into Query Console:

```
declareUpdate();
xdmp.documentInsert(
   "/employee1.json",
    { "Employee": {
      "EmployeeID": 1,
      "FirstName": "John",
      "LastName": "Widget",
      "Position": "Manager of Human Resources" }}),

xdmp.documentInsert(
   "/employee2.json",
    { "Employee": {
      "EmployeeID": 2,
      "FirstName": "Jane",
      "LastName": "Lead",
      "Position": "Manager of Widget Research" }}),

xdmp.documentInsert(
   "/employee3.json",
    { "Employee": {
      "EmployeeID": 3,
      "FirstName": "Steve",
      "LastName": "Manager",
      "Position": "Senior Technical Lead" }}),
```

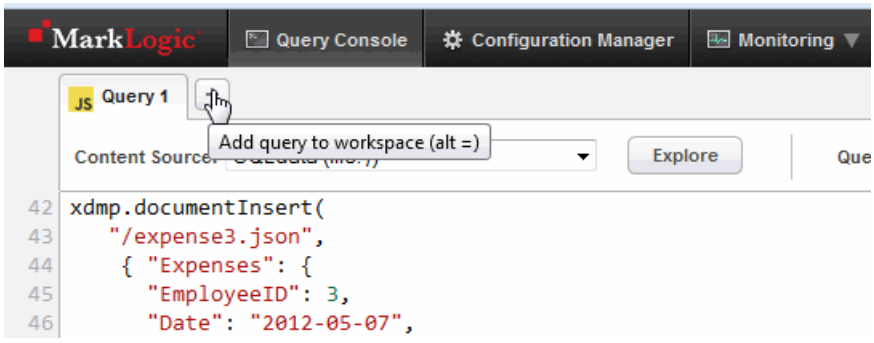
```
xdmp.documentInsert(
   "/employee4.json",
    { "Employee": {
      "EmployeeID": 4,
      "FirstName": "Debbie",
      "LastName": "Goodall",
      "Position": "Senior Widget Researcher" }}),

xdmp.documentInsert(
   "/expense1.json",
    { "Expenses": {
      "EmployeeID": 2,
      "Date": "2012-06-27",
      "Amount": 131.02}}),

xdmp.documentInsert(
   "/expense2.json",
    { "Expenses": {
      "EmployeeID": 1,
      "Date": "2012-08-03",
      "Amount": 59.95}}),

xdmp.documentInsert(
   "/expense3.json",
    { "Expenses": {
      "EmployeeID": 3,
      "Date": "2012-05-07",
      "Amount": 162.95}}),

xdmp.documentInsert(
   "/expense4.json",
    { "Expenses": {
      "EmployeeID": 3,
      "Date": "2012-05-30",
      "Amount": 120.00}}),

xdmp.documentInsert(
   "/expense5.json",
    { "Expenses": {
      "EmployeeID": 4,
      "Date": "2012-03-23",
      "Amount": 155.55}}),

xdmp.documentInsert(
   "/expense6.json",
    { "Expenses": {
      "EmployeeID": 4,
      "Date": "2012-06-05",
      "Amount": 104.29}})
```

4. In the control bar below the query window, click Run:

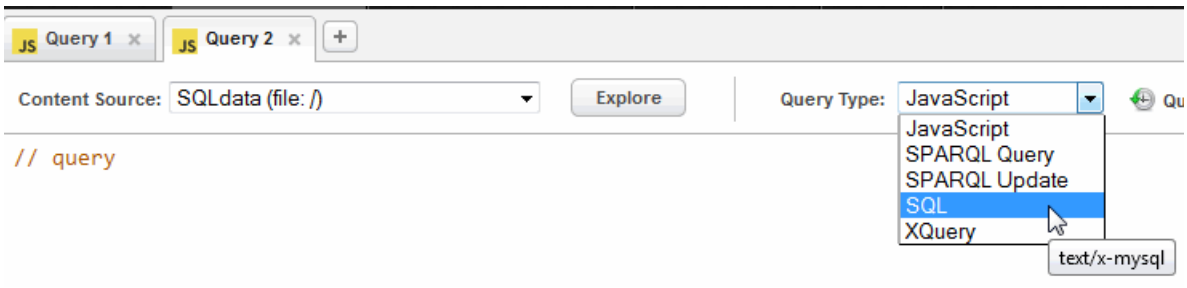


2.4 Enter SQL Queries to Test

1. To test that everything is working correctly, click + to open another query window:



2. In the new query window, make sure you have ‘SQLdata’ selected in the Content Source pull-down menu. Select a Query Type of SQL:

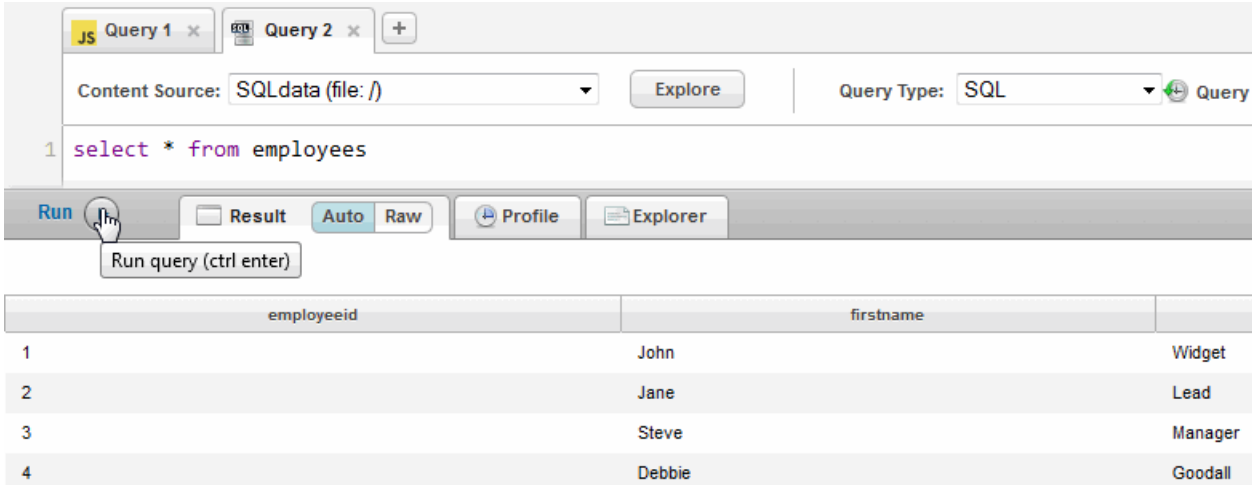


Enter the following query:

```
select * from employees
```

3. In the control bar below the query window, select Run.

4. You should see results that look like the following:



Note: MarkLogic Server treats SQL as case insensitive. Uppercase and lowercase characters are treated the same.

2.5 Using MLSQL

The MLSQL tool is a command line interface for issuing SQL statements. The executable MLSQL file is located in a MarkLogic installation at the following location:

- Windows: `c:\Program Files\MarkLogic\mlsql\mlsql.exe`
- Linux/Unix: `/opt/MarkLogic/bin/mlsql`

Note: On Linux/Unix, MLSQL is installed along with the ODBC driver, as described in "Installing the ODBC Driver on Linux" on page 54.

Note: MLSQL is not supported on Mac OS.

You must be assigned the `sql-execution` role on MarkLogic Server to use MLSQL.

To use the MLSQL tool, open a shell window and enter:

```
mlsql -h hostname -p 5432 -U username
```

Enter your password when you see a prompt like:

```
username=>
```

Enter a few SQL queries, like the following:

```
username=> SELECT * FROM employees;

username=> SELECT employees.FirstName, employees.LastName,
SUM(expenses.Amount) AS ExpensesPerEmployee
FROM employees, expenses
WHERE employees.EmployeeID = expenses.EmployeeID
GROUP BY employees.FirstName, employees.LastName;

username=> SELECT employees.FirstName, employees.LastName,
SUM(expenses.Amount) AS ExpensesPerEmployee
FROM employees JOIN expenses
ON employees.EmployeeID = expenses.EmployeeID
GROUP BY employees.FirstName, employees.LastName
ORDER BY ExpensesPerEmployee;
```

Note: A semicolon (;) is used in MLSQL to designate the end of a SQL query.

To demonstrate the purpose of the Searchable Field in the view, try the following queries:

```
username=> SELECT * from employees WHERE employees MATCH "Manager";

username=> SELECT * from employees WHERE employees
MATCH "position:Manager";
```

The first query searches for the word “Manager” in all of the document elements. The `position:Manager` specification in the second query narrows the search for “Manager” to the elements included in the `position` field, which in this case is the `Position` element.

To exit MLSQL, enter: `\q`

If you get results from the SQL queries, you can proceed to connecting your BI tool to MarkLogic Server, as described in “Connecting Cognos to MarkLogic Server” on page 66 and “Connecting Tableau to MarkLogic Server” on page 56.

Note: If you add or change the contents of a view, database, or documents, you must exit and restart MLSQL.

The MLSQL session commands are:

Command	Description
\copyright	Returns distribution terms.
\h	Returns list of available SQL commands.
\?	Returns list of available psql commands.
\g	Re-executes the last query.
\q	Exits MLSQL.

The syntax of a MLSQL command is:

```
mlsql [OPTION]... [DBNAME [USERNAME]]
```

Where:

Connection options:

Option	Description
`-h, --host=HOSTNAME`	Database server host or socket directory (default: "local socket")
`-p, --port=PORT`	ODBC server port (default: "5432")
`-U, --username=USERNAME`	Database user name (default: "username")
`-w, --no-password`	Never prompt for password
`-W, --password`	Force password prompt (should happen automatically)

General options:

Option	Description
`-c, --command=COMMAND`	Run only single command (SQL or internal) and exit
`-d, --dbname=DBNAME`	Database name to connect to (default: "gfurbush")
`-f, --file=FILENAME`	Execute commands from file, then exit
`-l, --list`	List available databases, then exit
`-v, --set=, --variable=NAME=VALUE`	Set psql variable NAME to VALUE

Option	Description
`-X, --no-psqlrc`	Do not read startup file (~/.psqlrc)
`-1 ("one"), --single-transaction`	Execute command file as a single transaction
`--help`	Show help, then exit
`--version`	Output version information, then exit

Input and output options:

Option	Description
`-a, --echo-all`	Echo all input from script
`-e, --echo-queries`	Echo commands sent to server
`-E, --echo-hidden`	Display queries that internal commands generate
`-L, --log-file=FILENAME`	Send session log to file
`-n, --no-readline`	Disable enhanced command line editing (readline)
`-o, --output=FILENAME`	Send query results to file (or |pipe)
`-q, --quiet`	Run quietly (no messages, only query output)
`-s, --single-step`	Single-step mode (confirm each query)
`-S, --single-line`	Single-line mode (end of line terminates SQL command)

Output format options:

Option	Description
`-A, --no-align`	Unaligned table output mode
`-F, --field-separator=STRING`	Set field separator (default: "|")
`-H, --html`	HTML table output mode
`-P, --pset=VAR[=ARG]`	Set printing option VAR to ARG (see \pset command)

Option	Description
`-R, --record-separator=STRING`	Set record separator (default: newline)
`-t, --tuples-only`	Print rows only
`-T, --table-attr=TEXT`	Set HTML table tag attributes (e.g., width, border)
`-x, --expanded`	Turn on expanded table output

3.0 SQL Data Modeling

This chapter describes how to configure MarkLogic Server and create views to model your MarkLogic data for access by SQL. Views can also be created using the Views API described in *MarkLogic XQuery and XSLT Function Reference*.

> **Note:** You must have the `admin` role on MarkLogic Server to complete the procedures described in this chapter.

The main topics are:

3.1 Creating Range Indexes for Column Specifications

You must create range indexes for a database before creating view columns that make use of the range indexes. In addition, range indexes are constructed during the document loading process, so they should be created before you load any XML documents into the database, otherwise the content must be either reindexed or reloaded to take advantage of the new range indexes. For details on how to create range indexes, see Range Indexes and Lexicons in the *Administrator's Guide*.

The following table lists the types range indexes that can be used for columns.

Range Index Type	Description
Path Range Index	Creates a range index on an element or attribute, as defined by an XPath expression.
Element Range Index	Creates a range index on an element.
Attribute Range Index	Creates a range index on an attribute in an element.
Field Range Index	Creates a range index based on the included and excluded elements in a field.

3.2 Creating Searchable Fields for use by Views

Fields provide a convenient mechanism for querying a portion of the database based on element QNames. A field can be defined for one or more elements, as described in Fields Database Settings in the *Administrator's Guide*. Binding a field to a view is useful when you don't want to create a column on the element, but you want the ability to query content in one or more elements simply and efficiently as a single unit. The procedure for binding a field to a view is described in "Creating View Fields" on page 37.

Field values are computed by concatenating tokens from all the "included" elements of a field. However, efficient evaluation of range queries on field values will need range indexes on these values, as described in Creating a Range Index on a Field in the *Administrator's Guide*.

> **Note:** A field cannot have the same name as a range index.

3.3 Creating a View

Each column in a view has a name and a range index reference. You can create a schemas and views using the XQuery `view` API or by means of the REST API.

This section describes how to create views using the REST API with the JSON document format. The topics are:

- Naming the View
- Creating and Setting the Schema
- Setting Schema and View Permissions
- Creating View Columns
- Creating View Columns for URI and Collection Lexicons
- Creating View Fields
- Defining View Scope

3.3.1 Naming the View

The view name must be unique in the context of the schema in which it is created. A valid view name is a single word that starts with an alpha character. The view name may contain numeric characters, but, with the exception of underscores ('_'), cannot contain punctuation or special characters.

For example, to create a view, named "`employees`":

```
"name": "employees"
```

3.3.2 Creating and Setting the Schema

As described in “Schemas and Views” on page 4, a schema is a naming context for a set of views. Each view must belong to a schema.

Every SQL deployment must include a default schema, called "main." The main schema is created automatically and is the default schema set for new views. To create a new schema, check the New Schema button and enter the name of the schema in the adjacent field.

> **Note:** The schema name must be unique. A valid schema name is a single word that starts with an alpha character. The schema name may contain numeric characters, but, with the exception of underscores (‘_’), cannot contain punctuation or special characters.

You can use `POST:/manage/v2/databases/{id|name}/view-schemas` to create a new schema. For example to create a schema, named “`mySchema`”, for the `SQLdata` database:

```
curl -X POST  --anyauth --user admin:admin \
--header "Content-Type:application/json" \
-d '{"view-schema-name": "mySchema"}' \
http://gordon-2:8002/manage/v2/databases/SQLdata/view-schemas?format=json
```

> **Note:** You can use `PUT:/manage/v2/databases/{id|name}/view-schemas/{schema-name}/views/{id|name}/properties` to set or add permissions to a schema.

3.3.3 Setting Schema and View Permissions

Permissions set on a schema and/or view determine which users have access to the schema or view. A permission consists of a role name, such as app-user, and a capability, such as read, insert, update, or execute. Users are assigned roles, as described in Role-Based Security Model in the *Understanding and Using Security Guide*. You can enable and disable views for different users by assigning permissions that correspond to a particular user’s role, along with the capabilities you want that users to possess for that view.

By default, views are assigned the following permissions:

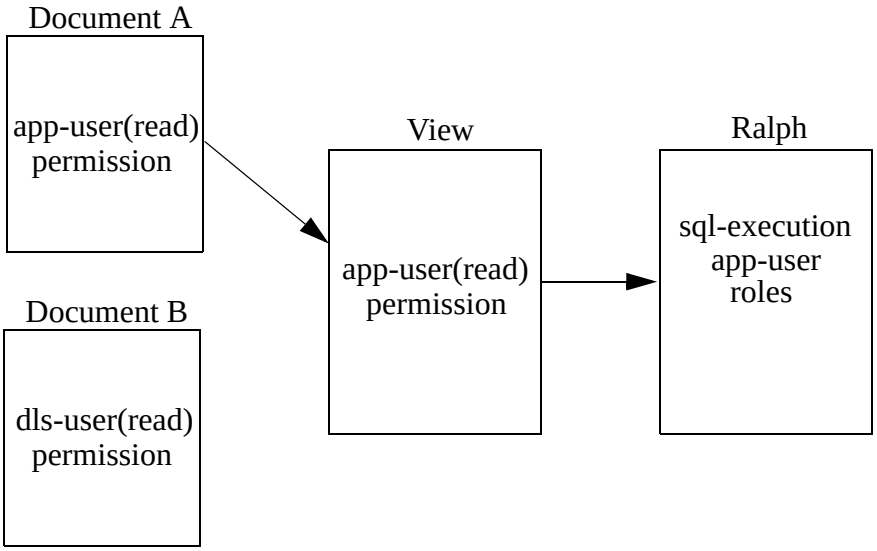
- sql-execution(read)
- view-admin(read)
- view-admin(update)

> **Note:** Unlike other documents, user default permissions are not assigned to the view or schema.

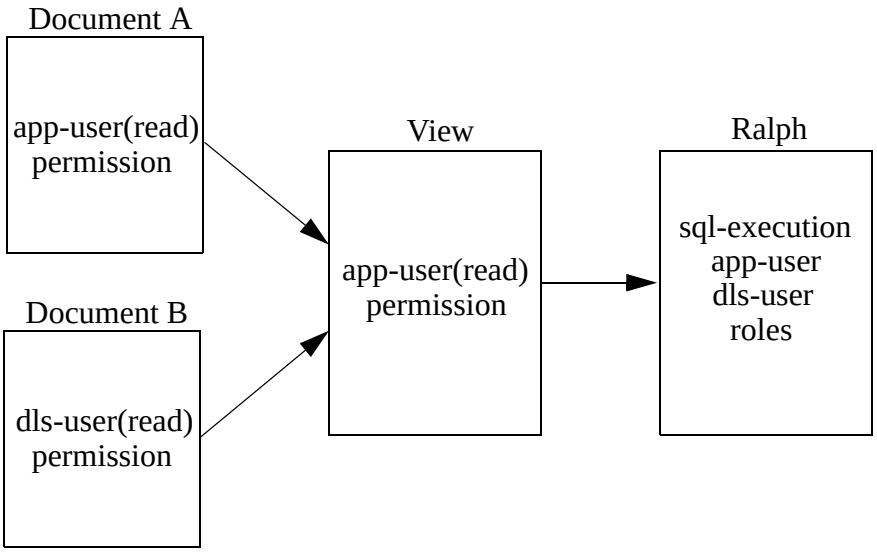
This means that users with the `sql-execution` role can execute SQL SELECT statements and get functions on the view, such as `view:get` or `view:get-column`, but cannot modify or otherwise manage the view. Only users with both the `sql-execution` and `view-admin` roles can fully access and manage views.

You can use the view API to set additional permissions on a schema or view to further restrict which users can access the schema or view. You set permissions on a schema by calling `view:schema-set-permissions` on an existing schema or by calling `view:schema-create` when creating a new schema. You set permissions on a view by calling `view:set-permissions` on an existing view or by calling `view:create` when creating a new view.

Schemas and views are simply documents stored in a schema database, so setting permissions on a schema or view has the same security implications as permissions set on any other type of document. This means a user must be assigned the correct roles to access the view. For example, Ralph has the `sql-execution` and `app-user` role. You set a view with the `app-user(read)` permission. This means that Ralph can read data from the view, but only documents that are loaded with the `app-user(read)` permission. Now, let's say we have documents that were loaded with the `dls-user(read)` permission. Ralph does not have the `dls-user` role, so he cannot read the data from these documents from this or any other view.



However, if we assign the `dls-user` role to Ralph, he can now read the documents loaded with the `dls-user(read)` permission through the view, regardless of the permissions set on the view. In this way, the permissions set on the view control only which users can access the view, rather than which documents can be seen through the view.



3.3.4 Creating View Columns

When creating columns in your view be sure that their settings map to the range index to be used for the column. The table below describes the JSON payload to create a column for each type of range index.

Range Index Type	REST Payload for View Column

Path Range Index	`{"column-name": "name",`
	`"path-reference":{`
	`"path-expression":"path",`
	`"scalar-type":"type",`
	`"collation":"http://marklogic.com/collation/codepoint"}`
	`}`

Element Range Index	`{"column-name": "name",`
	`"element-reference":{`
	`"namespace-uri":"",`
	`"localname":"name",`
	`"scalar-type":"type",`
	`"collation":"http://marklogic.com/collation/"}`
	`}`
	
	Note: The `collation` element is optional.

Range Index Type	REST Payload for View Column

Attribute Range Index	`{"column-name": "name",`
	`"element-attribute-reference":{`
	`"parent-namespace-uri":"",`
	`"parent-localname":"name",`
	`"namespace-uri":"",`
	`"localname":"name",`
	`"scalar-type":"type",`
	`"collation":"http://marklogic.com/collation/"}`
	`}`
	
	Note: The `collation` element is optional.

Field Range Index	`{"column-name": "name",`
	`"field-reference": [{`
	`"field-name": "name"`
	`}]`

Range indexes on elements or attributes of type string are associated with a collation that specify the order in which strings are sorted and how they are compared. A collation is required for columns that use path and field range indexes and are optional for columns that use element and attribute range indexes. For more details on collations, see Collations in the *Search Developer's Guide*.

To make the column nullable, specify nullable as true:

```
"nullable":true
```

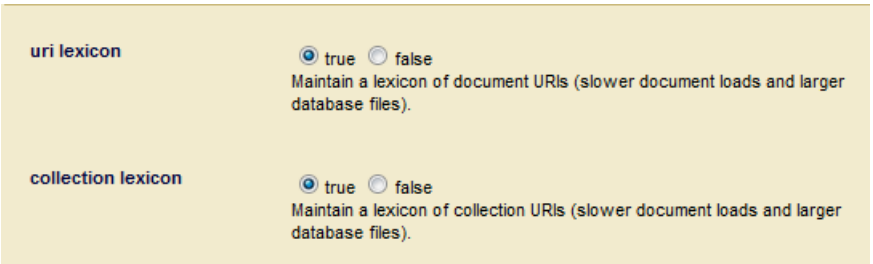
For example, to specify the "`subject`" column as nullable:

```
{ "column-name": "subject",
    "element-reference": {
         "namespace-uri": "",
         "localname": "subject",
         "scalar-type": "string",
         "nullable":true}
}
```

3.3.5 Creating View Columns for URI and Collection Lexicons

In addition to range indexes, you can also create view columns for uri and collection lexicons, as described in URI and Collection Lexicons in the *Search Developer's Guide*. To create columns on uri and/or collection lexicons, you must enable the capability for the database.

1. In the Admin Interface, open the database that contains your content and scroll down to the uri and collection lexicon fields. Click on true to enable either or both types of lexicons.



2. To create a column for the uri lexicon, use:

```
{"column-name":"uri", "uri-reference":null}
```

3. To create a column for the collection lexicon, use:

```
{"column-name":"collection", "collection-reference":null}
```

3.3.6 Creating View Fields

The following procedure describes how to bind a field to a view.

1. Create a searchable field, as described in "Creating Searchable Fields for use by Views" on page 32.

2. In the view description, enter:

```
"field-reference": [{
    "field-name": "name",
}]
```

For example, to create a view field for the `position` field:

```
"field-reference": [{
    "field-name": "position"
}]
```

Note: For information on how to do a search on a field, see "MATCH Operator" on page 80.

3.3.7 Defining View Scope

The scope of the view used to constrain the view to a subset of the documents in the database. The scope can either limit rows in the view to documents containing a specific element (localname + namespace) or to documents in a particular collection. The scope is optional, so do not specify a scope if you elect not to set the scope of the view.

For example, you want to constrain the view to read only documents that have a `messages` element in the '`mail`' namespace. To set the scope of the view to the `messages` element, use `element-scope` and enter '`messages`' as the `localname` and '`mail`' as the `namespace-uri`.

```
"element-scope": {
     "namespace-uri":"mail", "localname":"messages"}
```

Note: If you have enabled "element word positions" on the database, the view scope will be limited to descendants of the specified parent element.

To set the scope for a collection, use `collection-scope` and enter the collection uri.

```
"collection-scope":{
     "collection":"/xdmp/view/messages"}
```

3.4 Data Modeling Example

Data stored in MarkLogic Server is typically unstructured. The data modeling challenge is to determine how to identify the XML elements and attributes in the data and present them as relational. The purpose of this section is to provide an example of how unstructured data, such as emails, might be modeled for SQL access.

- The Email Data
- The Range Indexes
- The View

3.4.1 The Email Data

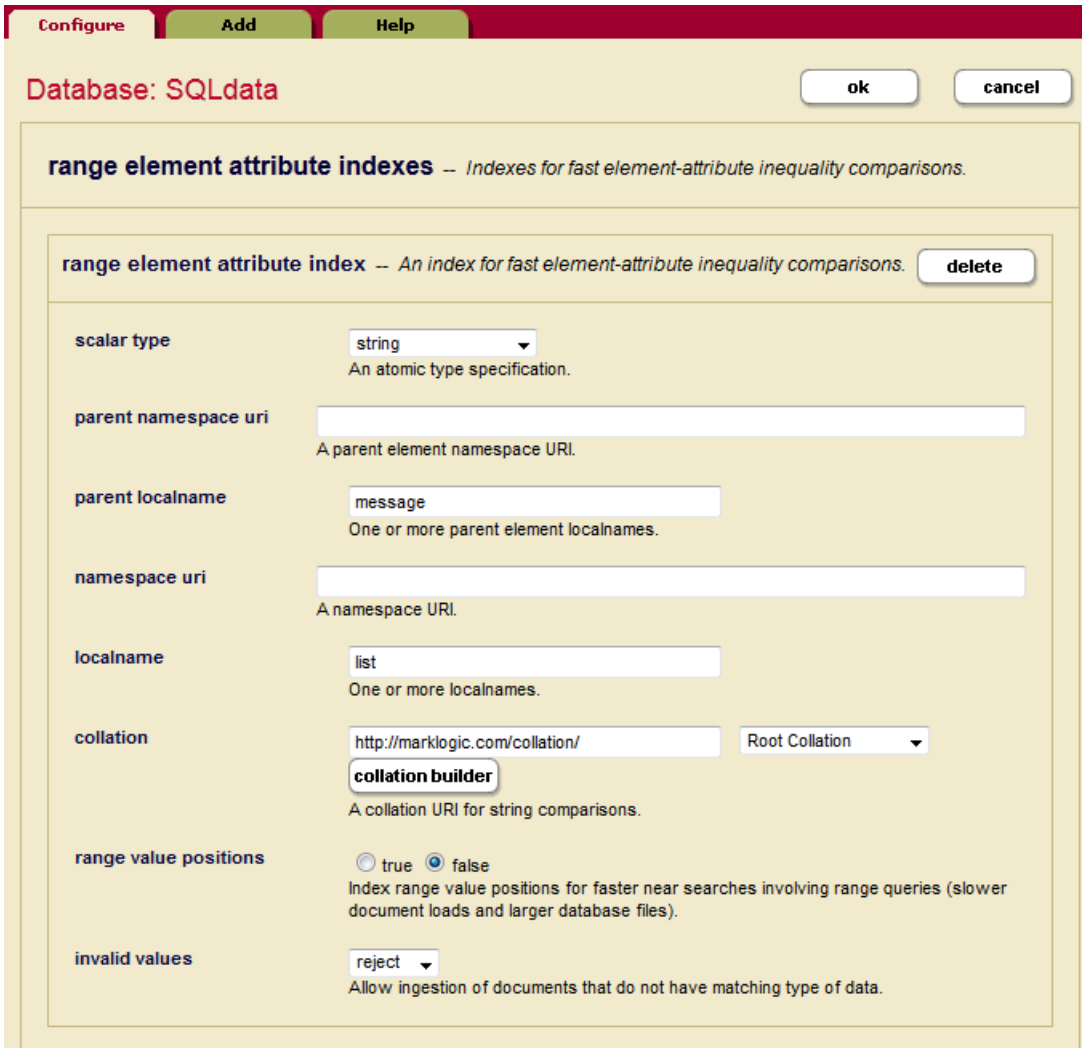
The procedures in this section assume you are loading documents like the following in your content database. The elements highlighted in yellow are those to be modeled as columns in the view.

```
<?xml version="1.0" encoding="UTF-8"?>
<message list="org.codehaus.grails.user" id="3mbfddak67aiefea"
            date="2010-05-17T01:00:21.627923-08:00">
  <headers>
     <from personal="Ian Roberts">i.roberts@dcs.shef.ac.uk</from>
     <to personal="Grails User">user@grails.codehaus.org</to>
     <subject>How to inject a session-scoped service into another service</subject>
  </headers>
  <body type="text/plain; charset=us-ascii">
    <para>
      <url>http://ldaley.com/56/scoped-services-in-grails</url> Covers the same thing, but has a little
       bit more detail WRT testing etc.
    </para>
    <para>
      Note rather than inject the application context you can also do <function>myServiceProxy</function>
      (org.springframework.aop.scope.ScopedProxyFactoryBean){ targetBeanName = 'myService'
       proxyTargetClass = true}
    </para>
    <note>
      See <url>http://ldaley.com/42/proxies-in-grails</url> for an intro to proxies.
    </note>
    <footer type="signature" depth="1" hash="1986520897999785197">--
      <name>Ian Roberts</name>  | Department of Computer Science
      <email>i.roberts@dcs.shef.ac.uk</email>
      <affiliation>University of Sheffield, UK</affiliation>
   </footer>
  </body>
</message>
```

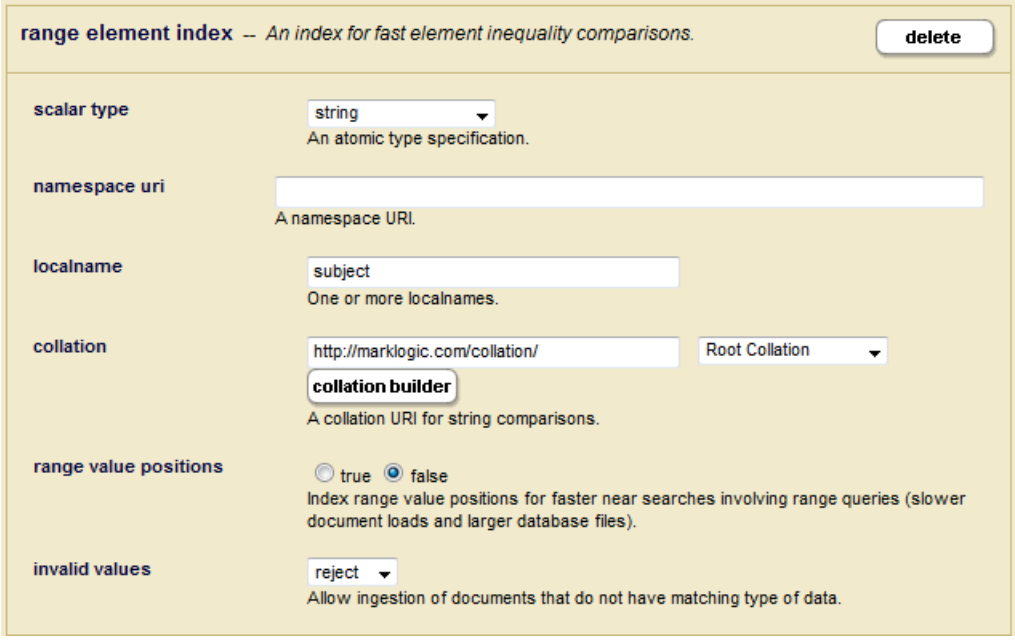
3.4.2 The Range Indexes

This section describes how to create the range indexes for the view described in “The View” on page 43.

Create an Attribute Range Index for the ‘list’ attribute in the `message` element:



Create an Element Range Index for the `subject` element:



Create additional Element Range Indexes for the following elements:.

Local Name	Scalar Type
`function`	`string`
`name`	`string`
`affiliation`	`string`

When you want to create columns for an element of the same name, but with different parent elements, you can create path range indexes for each. For example, in our message document we have `url` elements with different parents, `para` and `note`. In order to define these as separate columns

3.4.3 The View

Create a view, named ‘mail’ with a scope on the element, `message`.

> **Note:** There may be other documents in the database that contain some of the same elements as mail documents, but we can disambiguate the elements in mail documents by setting the scope of the view to the element, `message`.

The following call to `POST:/manage/v2/databases/{id|name}/view-schemas/{schema-name}/views` creates a view with columns for all of the range indexes created in “The Range Indexes” on page 40.

```
curl -X POST  --anyauth --user admin:admin \
--header "Content-Type:application/json" \
-d '{
     "view-schema-name": "mail",
     "element-scope":{"namespace-uri":"", "localname":"message"},
     "column": [
       {
         "column-name": "message_list",
         "element-attribute-reference": {
            "namespace-uri":"",
            "parent-namespace-uri" : "",
            "parent-localname": "message",
            "localname": "list",
            "scalar-type": "string"
         }
       },
       {
         "column-name": "subject",
         "element-reference": {
            "namespace-uri": "",
            "localname": "subject",
            "scalar-type": "string"
         }
       },
       {
         "column-name": "function",
         "element-reference": {
            "namespace-uri": "",
            "localname": "function",
            "scalar-type": "string"
         }
       },
       {
         "column-name": "name",
         "element-reference": {
            "namespace-uri": "",
            "localname": "name",
            "scalar-type": "string"
         }
       },
       {
```

```
            "column-name": "affiliation",
            "element-reference": {
               "namespace-uri": "",
               "localname": "affiliation",
               "scalar-type": "string"
            }
         },
         {
            "column-name": "body_url",
            "path-reference": {
               "path-expression": "/message/body/url",
               "scalar-type": "anyURI"
            }
         },
         {
            "column-name": "para_url",
            "path-reference": {
               "path-expression": "/message/body/para/url",
               "scalar-type": "anyURI"
            }
         }
      ]
   }' \
http://gordon-2:8002/manage/v2/databases/SQLdata/view-schemas/main/vie
ws?format=json
```

3.5 Guidelines for Relational Behavior

For conventional relational behavior, data should be modeled such that:

- Every document represents exactly one row.
- Every row has at least one column that is declared as non-nullable. If this is not possible, then you should enable the URI lexicon.
- Every non-nullable column is present in every document.
- Sufficient range indexes are enabled so that a query constraining the presence of a column or the table scope can be resolved from the index.
- Sufficient range indexes are enabled so that a query representing a where clause constraint can be resolved from the index. For simple relations (equals, less than, etc.), such constraints can and will be checked redundantly by the SQL VM, but full-text constraints cannot and will not. Full-text constraints on a URI or collection column will not work.

Note: Nullable columns impede performance, so you should avoid them when possible. A column that has no null values is one that may or may not be declared nullable. In other words, “nullable” is about the configuration and “has no null values” is about the data.

Consider an XML document of the following form, with element range indexes on the `title`, `pubyear`, `author`, and `keyword` elements and a view, named `books`, defined over those range indexes:

```
<book>
    <title>An Example</title>
    <pubyear>2011</pubyear>
    <author>Jane Smith</author>
    <keyword>science</keyword>
    <author>John Doe</author>
    <keyword>nature</keyword>
    <body>
       Lots of exciting full text content here...
    </body>
</book>
```

The same document can be expressed in JSON as follows:

```
{"book": {
    "title" : "An Example",
    "pubyear" : "2011",
    "author" : ["Jane Smith", "John Doe"],
    "keyword" : ["science", "nature"],
    "body": "Lots of exciting full text content here..."}
}
```

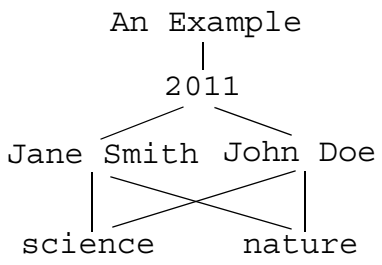
Because this document contains two `author` and `keyword` elements at the same level, it violates the first data modeling rule listed above. As a result, a `select *` on this view will produce multiple rows for the single document:

```
select * from books

=>

| title      | pubyear| author    | keyword
--------------+---------+------------+---------
| An Example | 2011   | Jane Smith| science
| An Example | 2011   | Jane Smith| nature
| An Example | 2011   | John Doe  | science
| An Example | 2011   | John Doe  | nature
```

Each time a view encounters a column element in a document, it returns the contents of its associated range index. In the above example, the contents of the range indexes associated with this document are:

- title: `An Example`
- pubyear: `2011`
- author: `Jane Smith, John Doe`
- keyword: `science, nature`

The results of the query are the cross-product of these indexes. As a result, four rows are returned:



If cross-product results are undesirable, avoid queries that return more than one range index containing multiple values for the document. For example, you could omit the `keyword` column:

```
select title, pubyear, author from books

=>

| title      | pubyear| author
--------------+---------+------------
| An Example | 2011   | Jane Smith
| An Example | 2011   | John Doe
```

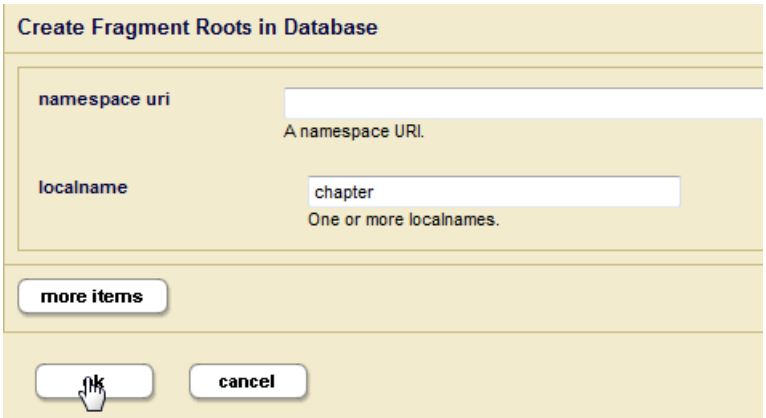
In other circumstances, you might want to set a root fragment on the database. For example, your document data is structured as follows:

```
<book>
   <chapter>
       <title>Chapter 1</title>
       <section>Section 1</section>
       <section>Section 2</section>
       <section>Section 3</section>
       <section>Section 4</section>
   </chapter>
   <chapter>
       <title>Chapter 2</title>
       <section>Section 1</section>
       <section>Section 2</section>
   </chapter>
</book>
```

You create a view, named `books`, on the `title` and `section` elements. The results of a `select *` query are:

```
select * from books

=>

title      | section
----------+-----------
Chapter 1 | Section 1
Chapter 1 | Section 1
Chapter 1 | Section 2
Chapter 1 | Section 2
Chapter 1 | Section 3
Chapter 1 | Section 4
Chapter 2 | Section 1
Chapter 2 | Section 1
Chapter 2 | Section 2
Chapter 2 | Section 2
Chapter 2 | Section 3
Chapter 2 | Section 4
(12 rows)
```

Creating a fragment root on the chapter element makes the document appear to the view as two separate documents, each with `chapter` as their root element. The details on defining fragments on a database, are described in Fragments in the *Administrator's Guide*.



Now, with a fragment root set on the `chapters` element, the results of a `select *` query are:

```
select * from books

=>

title     |  section
-----------+-----------
Chapter 1 | Section 1
Chapter 1 | Section 2
Chapter 1 | Section 3
Chapter 1 | Section 4
Chapter 2 | Section 1
Chapter 2 | Section 2
(6 rows)
```

In other situations, you might want to create more than one view for a particular document structure. For example, your document data is structured as follows:

```
<book>
   <meta>
       <title>An Example</title>
       <pubyear>2011</pubyear>
       <author>Jane Smith</author>
       <keyword>science</keyword>
   </meta>
   <chapter>
       <title>Chapter 1</title>
       <section>Section 1</section>
       <section>Section 2</section>
       <section>Section 3</section>
       <section>Section 4</section>
   </chapter>
   <chapter>
       <title>Chapter 2</title>
       <section>Section 1</section>
       <section>Section 2</section>
   </chapter>
</book>
```

The views are defined as follows:

View Name	Scope	Columns
meta	meta	title pubyear author keyword
chapter	chapter	title section

Note: If you are setting the view scope to an element that contains descendant elements of the same name but with different parent elements (as is the case with the `title` element in the above example), then you should consider creating path range indexes to distinguishes the same-name elements from one another.

3.6 Limitations to SQL Support

The SQL supported by MarkLogic Server is SQL92 as implemented in SQLITE with some additions and extensions as noted.

- Triggers, coherency constraints, keys, and foreign keys are not supported.
- MarkLogic views are read-only. You cannot update, delete, or insert data into a view. You cannot manage data stored in MarkLogic through DDL statements in SQL.
- SQL statements operate on range indexes in MarkLogic. If the information is not in a range index, it is not available via SQL. Exception: the whole document is available as a special hidden column which can be a target for a search constraint.
- Search constraints are unfiltered.
- The MATCH operator (full-text search) will not work on columns backed by the URI or collection lexicons.
- There must be exactly one row in each fragment and one fragment in each row. Failure to do so will produce anomalous results that may cause trouble for consuming applications. For example, if a fragment contains more than one row, where clause constraints on that row will only rule out fragments for which none of the rows matches the where clause constraint unless redundant checking is enabled (and even if it is for full-text constraints). If a row spans multiple fragments, it may not be selected when it should.

3.7 Errors, Exceptions, and Diagnostics

Errors will be thrown if attempts are made to use views that lack the necessary backing range indexes, or to use them in a way that those backing range indexes do not support (for example, a view cannot be ordered unless all the backing range indexes have positions). Errors will be thrown if the SQL statement is invalid, or the SQL engine encounters some kind of problem. In general, all errors encountered in processing SQL statements will be thrown as `SQL-ERROR`.

The following errors may be thrown when creating or modifying a view:

Error	Description
VIEW-NOTFOUND	Attempt to modify a non-existent view.
VIEW-FIELDNOTFOUND	Attempt to fetch a field binding that is not part of a view.
VIEW-DUPFIELD	Attempt to add a field binding whose name is the same as some other field or column and the error

Error	Description
VIEW-FIELDDUPVIEW	Attempt to add a field binding whose name is the same as the view name.

You can use trace events to write SQL operations on MarkLogic Server to the log:

Trace Event	Description
SQL Trace	Shows all the SQL being executed by the core as well as the constraining queries constructed to execute SQL.
SQL Trace Details	Equivalent to executing "pragma vbde_trace=1" which dumps a detailed execution trace of the SQLITE virtual machine to the log.
SQL Listing	Equivalent to executing "pragma vdbe_listing=1" which dumps the compiled SQLITE virtual machine program to the log.

To use the trace events, you must enable tracing (at the group level) for your configuration and set events. Perform the following to enable and set trace events:

1. Log into the Admin Interface.

2. Select Groups > *group_name* > Diagnostics.

   The Diagnostics Configuration page appears.

3. Click the `true` button for `trace events activated`.

4. Enter the trace events described in the above table you want to enable.

5. Click the OK button to activate the events.

After you configure the trace events, when any of the configured events occur, a line is added to the `ErrorLog.txt` file, indicating which document is involved the event.

> **Note:** The trace events are designed as development and debugging tools, and they might slow the overall performance of MarkLogic Server. Also, enabling many trace events will produce a large quantity of messages, especially if you are processing a high volume of documents. When you are not debugging, disable the trace event for maximum performance.

4.0 Installing and Configuring the MarkLogic Server ODBC Driver

Cognos, Tabeau, and other SQL tools require an ODBC driver on the client machine to communicate with MarkLogic Server. This chapter describes how to install and configure your MarkLogic Server ODBC driver on your client.

There is a 32-bit and 64-bit Windows MarkLogic ODBC driver and a 64-bit Linux ODBC driver, which are available from the MarkLogic Developer site:

http://developer.marklogic.com/products

Locate the ODBC driver for MarkLogic Server 7 and follow the appropriate setup procedure to install it on your client machine:

- Installing the ODBC Driver on Windows
- Installing the ODBC Driver on Linux

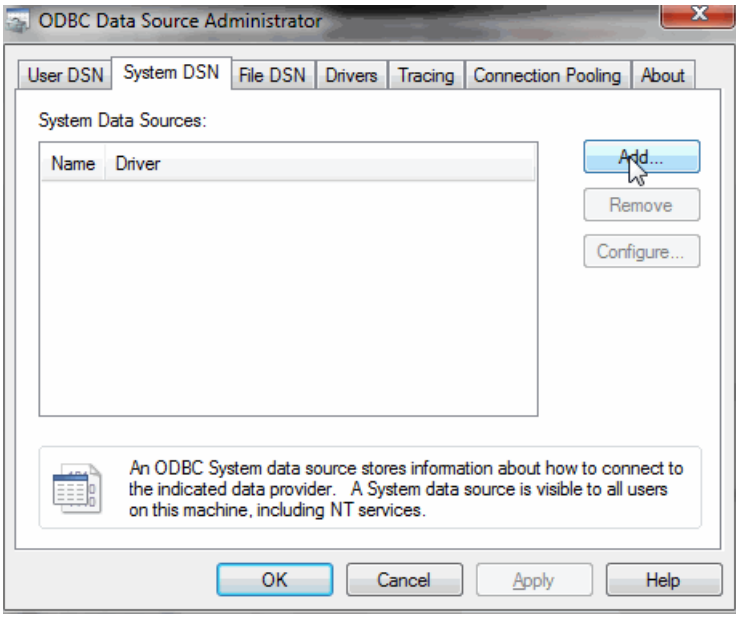
4.1 Installing the ODBC Driver on Windows

Cognos and Tableau communicate with MarkLogic Server via a 32-bit or 64-bit ODBC driver. This section describes how to configure your ODBC driver for use with MarkLogic.

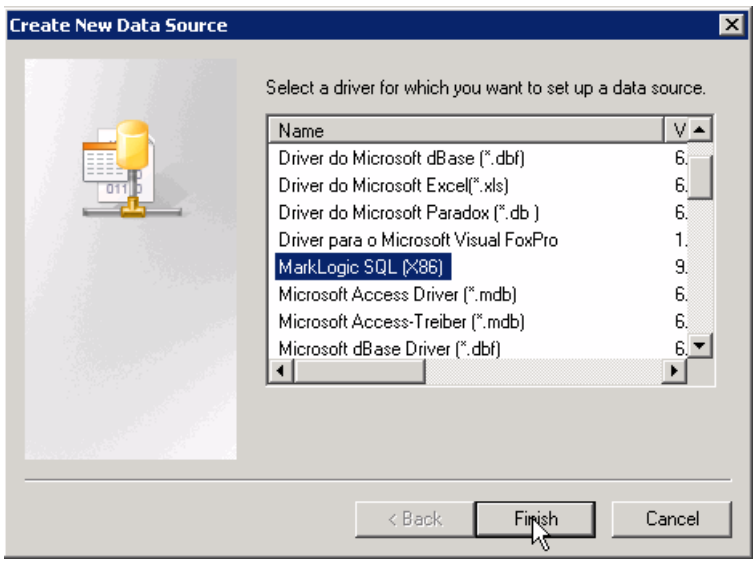
1. To launch the ODBC Data Source Administrator, open your Control Panel and navigate to:

   Administrative Tools > ODBC Data Source Administrator

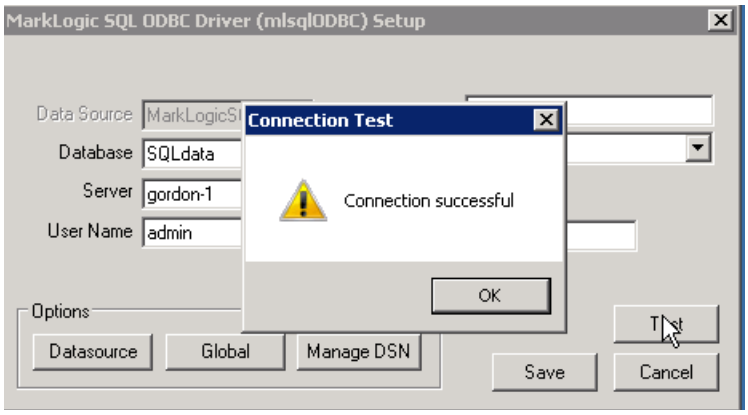
2. Click the System DSN tab and click Add:



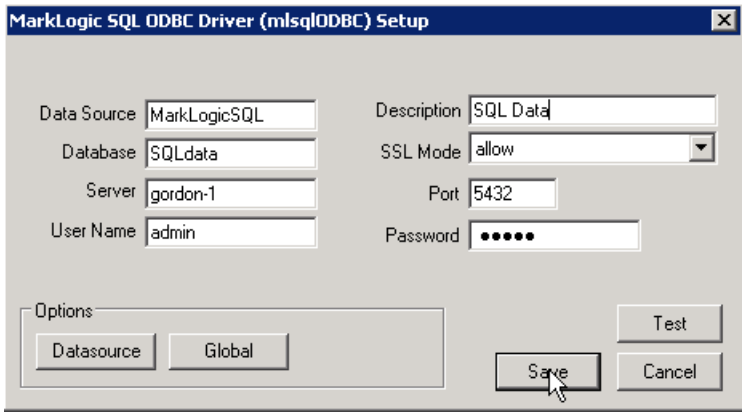
3. Select either the MarkLogicSQL (64-bit) or MarkLogicSQL (x86) (32-bit) driver and click Finish:



4. In the MarkLogic SQL ODBC Driver Setup dialog, enter a name for your data source, the database name (SQLdata), the name of the machine that hosts your MarkLogic Server, the port number of your MarkLogic ODBC App Server (5432), set SSL mode to 'allow', and your MarkLogic Server login credentials. Click Test to test the connection to MarkLogic Server.



5. If your connection test was successful, click Save. Otherwise, recheck your settings and retest.



4.2 Installing the ODBC Driver on Linux

Dependencies:openssl and unixODBC.

The following procedure describes how to install the MarkLogic ODBC driver on Linux:

1. Obtain a copy of unixODBC (2.3.2). You might be able to install it with `yum`, but if not, you can download the correct version from http://www.unixodbc.org/ to your `/tmp` directory and use the following procedure to install:

```
cd /tmp
tar -xvzf unixODBC-2.3.2.tar.gz
cd /tmp/unixODBC-2.3.2
./configure
make
sudo make install
```

2. If you want to communicate with MarkLogic over SSL, you can install the openssl libraries as follows:

```
yum install openssl-libs
```

   You can optionally install the GUI tools for unixODBC:

```
yum install unixODBC-gui-qt
```

3. Install the ODBC driver package (named `mlsqlodbc-1.1-20140528.x86_64.rpm` in this example):

```
rpm -i mlsqlodbc-1.1-20140528.x86_64.rpm
```

4. Call `odbcinst` to write the DSN to the current user's `.odbc.ini` file:

```
odbcinst -i -s -f /opt/MarkLogic/templates/mlsql.template
```

5. The name of the ODBC driver is MarkLogicSQL. Use `isql` to connect to MarkLogicSQL to confirm that the ODBC driver was correctly installed (the MarkLogic username and password in this example is admin/admin):

```
isql -v MarkLogicSQL admin admin
```

6. If you don't what to have to enter your username and password each time you run `isql`, you can edit the `~/.odbc.ini` file to add your MarkLogic username and password:

```
[MarkLogicSQL]
Description      = MarkLogicSQL
Driver           = MarkLogicSQL
Trace            = No
TraceFile        =
Database         = marklogic
Servername       = localhost
Username         = admin
Password         = admin
Port             = 5432
Protocol         = 7.4
ReadOnly         = No
SSLMode          = disable
UseServerSidePrepare  = Yes
ShowSystemTables      = No
ConnSettings          =
```

7. Test using `isql` without a username and password:

```
isql -v MarkLogicSQL
```

Note: If you encounter problems, make sure that the settings in the configuration files point to the right locations for your environment. Calling `odbcinst -j` will return the list of the configuration files for the ODBC driver.

5.0 Connecting Tableau to MarkLogic Server

This chapter describes how to set up your Tableau to communicate with MarkLogic Server. THe main topics are:

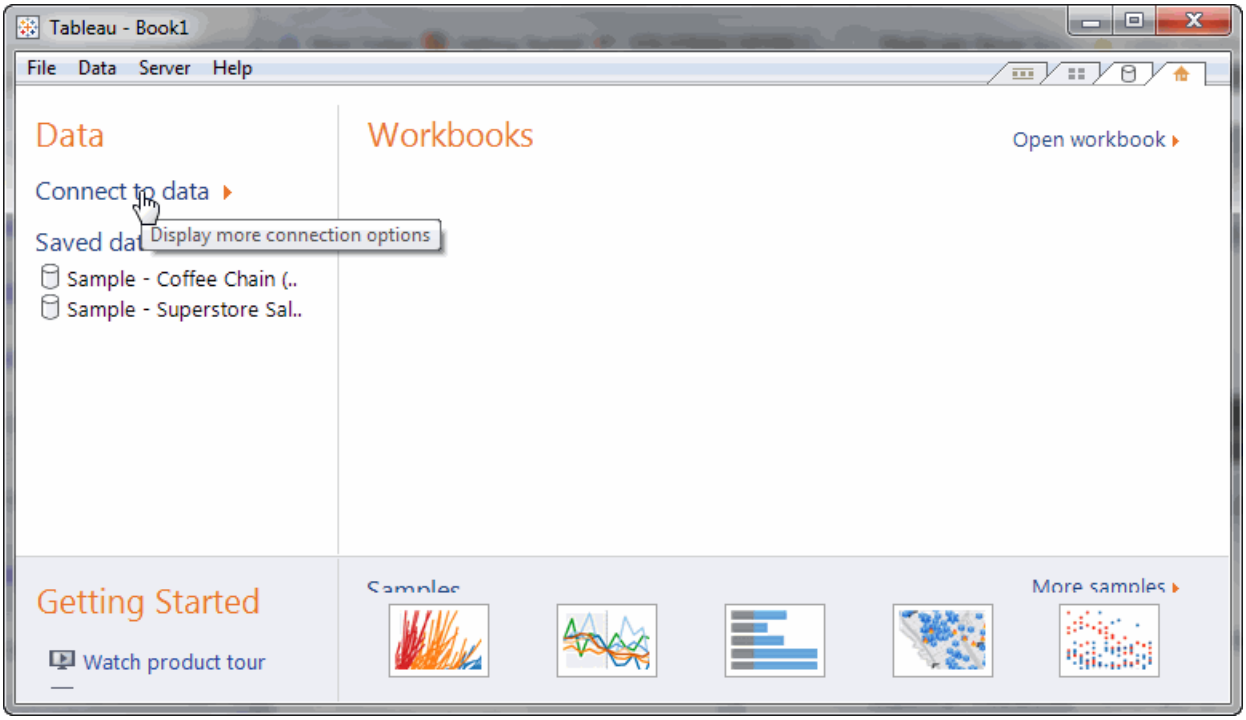
- Connect Tableau to MarkLogic Server
- Add Tables to Tableau Workbook

5.1 Connect Tableau to MarkLogic Server

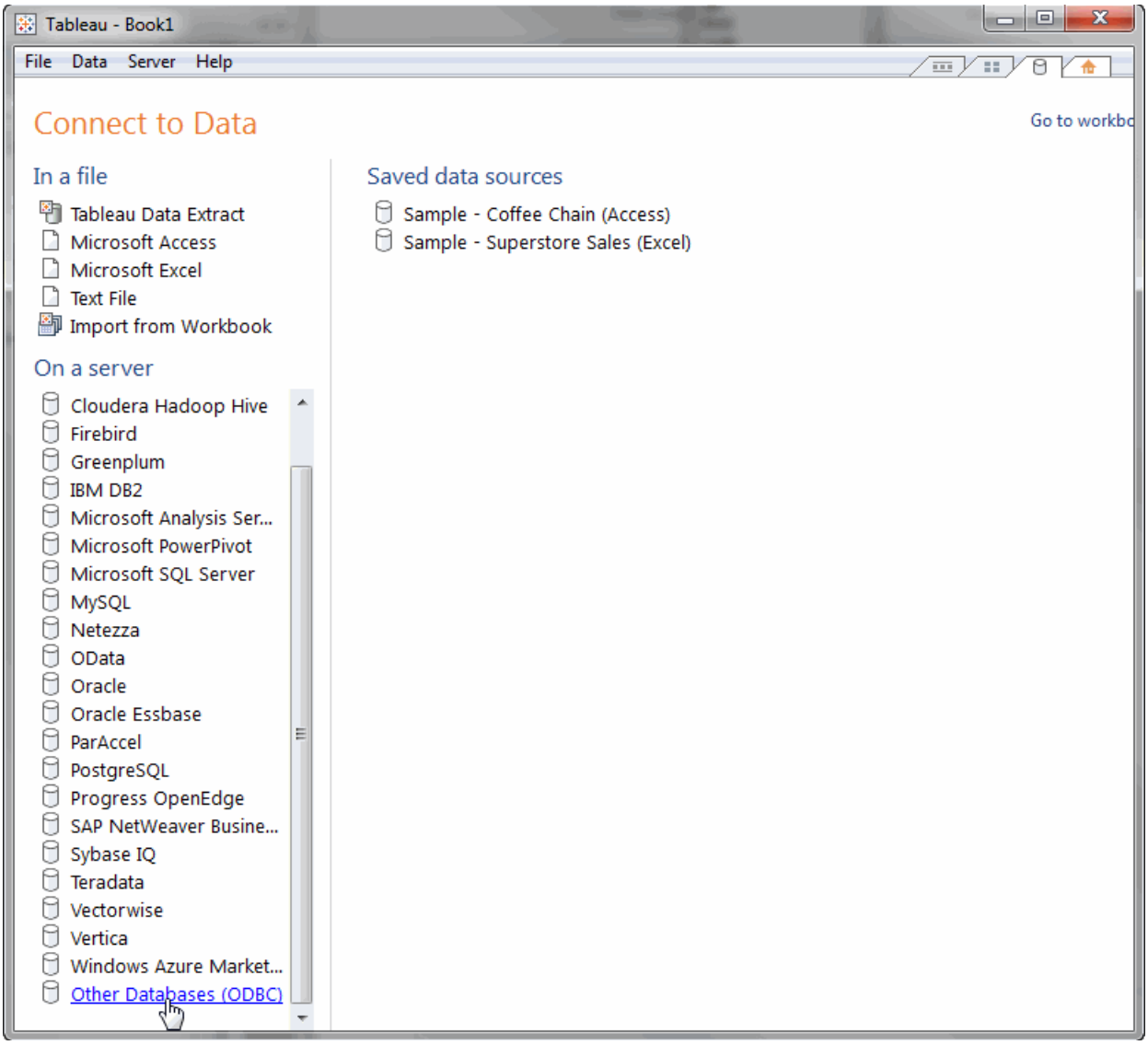
This section describes how to connect Tableau to MarkLogic Server.

The procedure described in this section assumes you have first installed the MarkLogic ODBC driver and configured it as an ODBC data source on the client server, as described in “Installing and Configuring the MarkLogic Server ODBC Driver” on page 51.

1. Open Tableau and click Connect to Data:

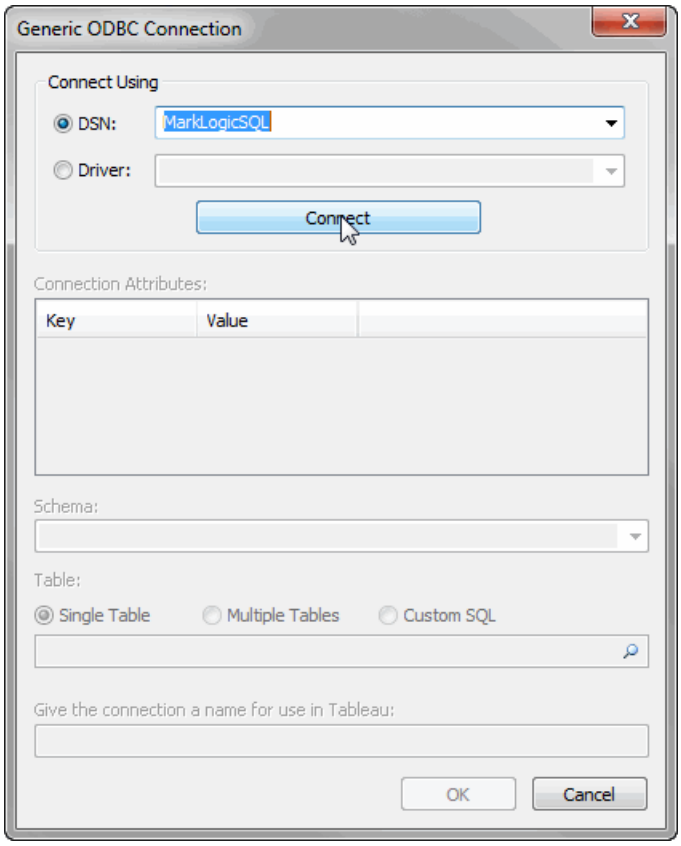


2. In the Connect to Data page, click on Other Databases (ODBC):



3. In the Generic ODBC Connection dialog, select the data source you created in “Installing and Configuring the MarkLogic Server ODBC Driver” on page 51 from the DSN pull-down menu:

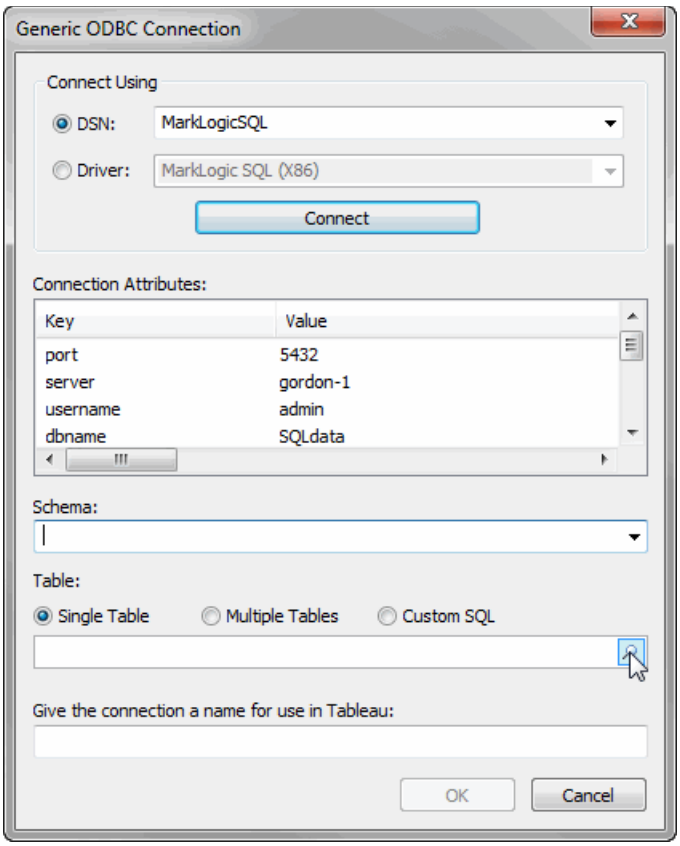
4. Click Connect:



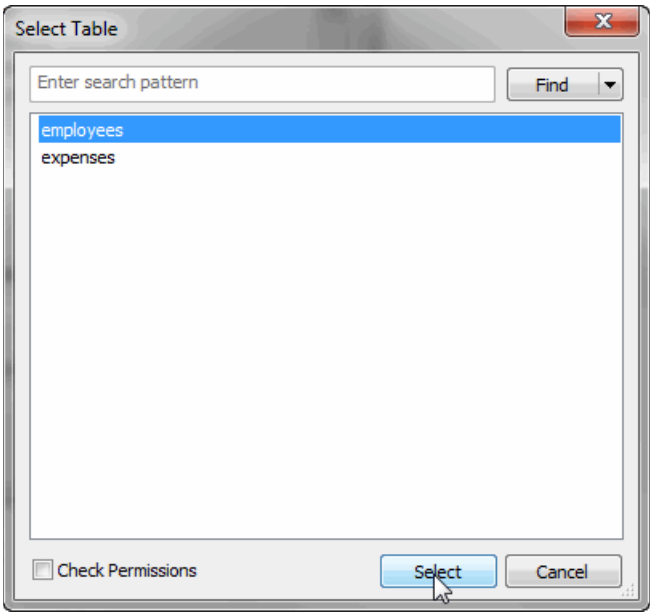
5.2 Add Tables to Tableau Workbook

After successfully connecting Tableau to MarkLogic Server, you can add the defined views, as tables, to your workbook.

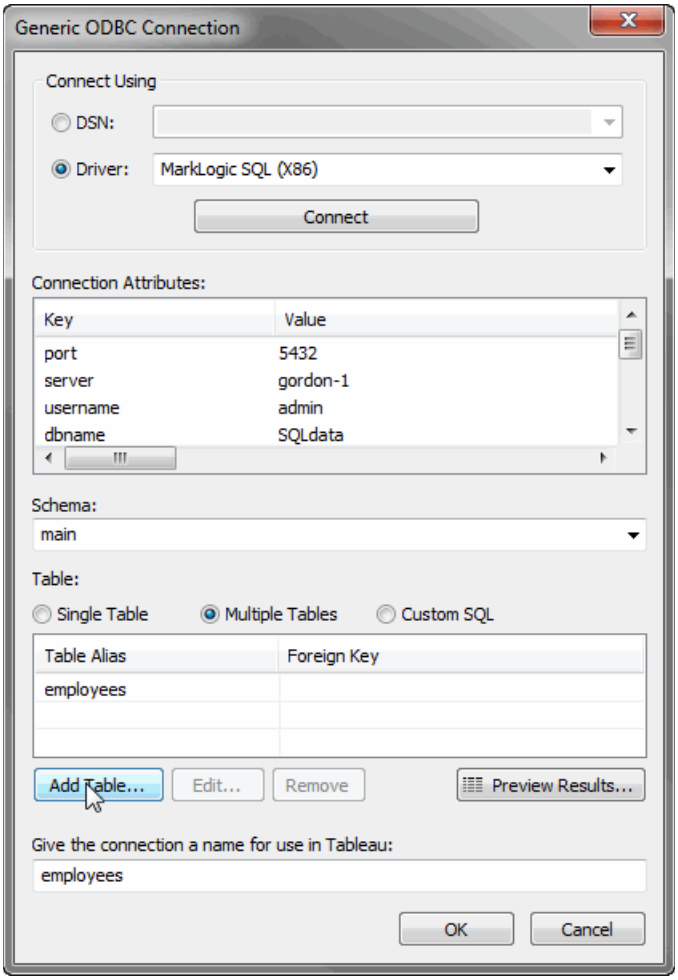
1. Select 'main' from the Schema pull-down menu, click on Single Table and click on the magnifying glass icon:



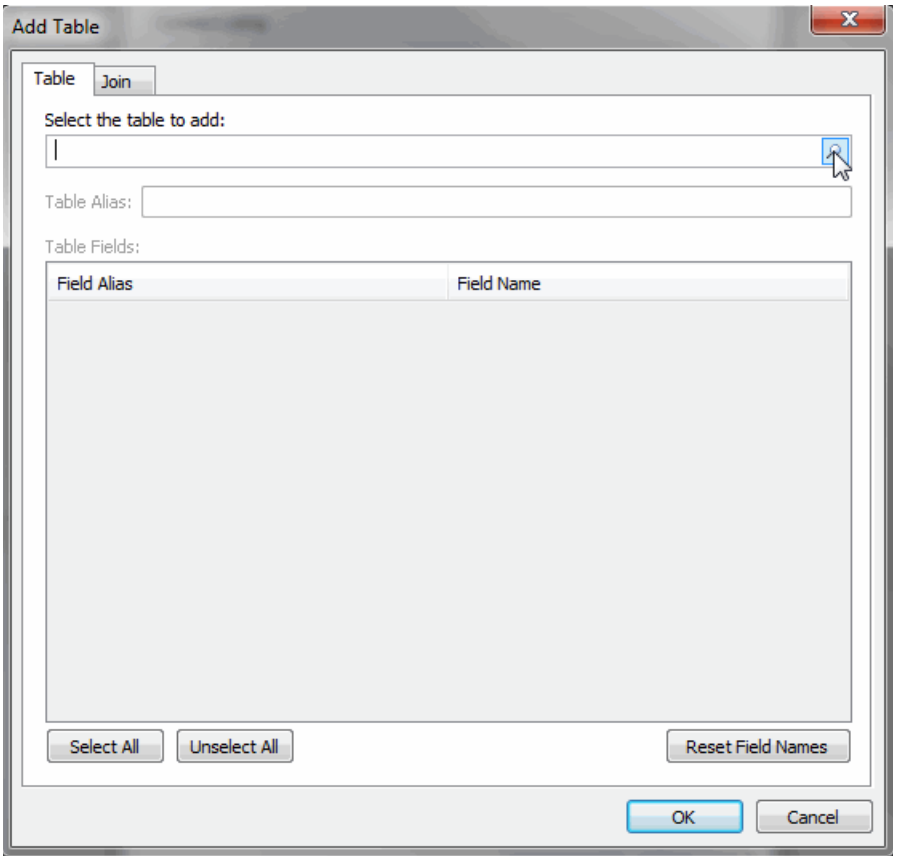
2. In the Select Table dialog, select ‘employee’ and click Select:



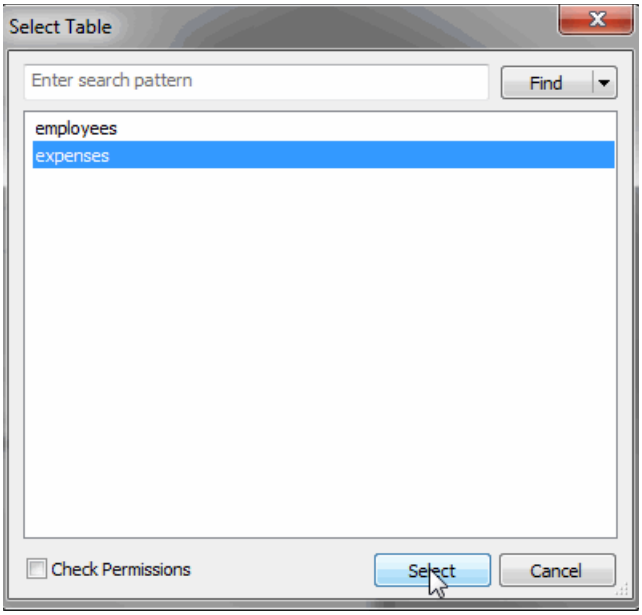
3. In the Generic ODBC Connection dialog, select Multiple Tables and click Add Table:



4. In the Add Table dialog, click on the magnifying glass icon:

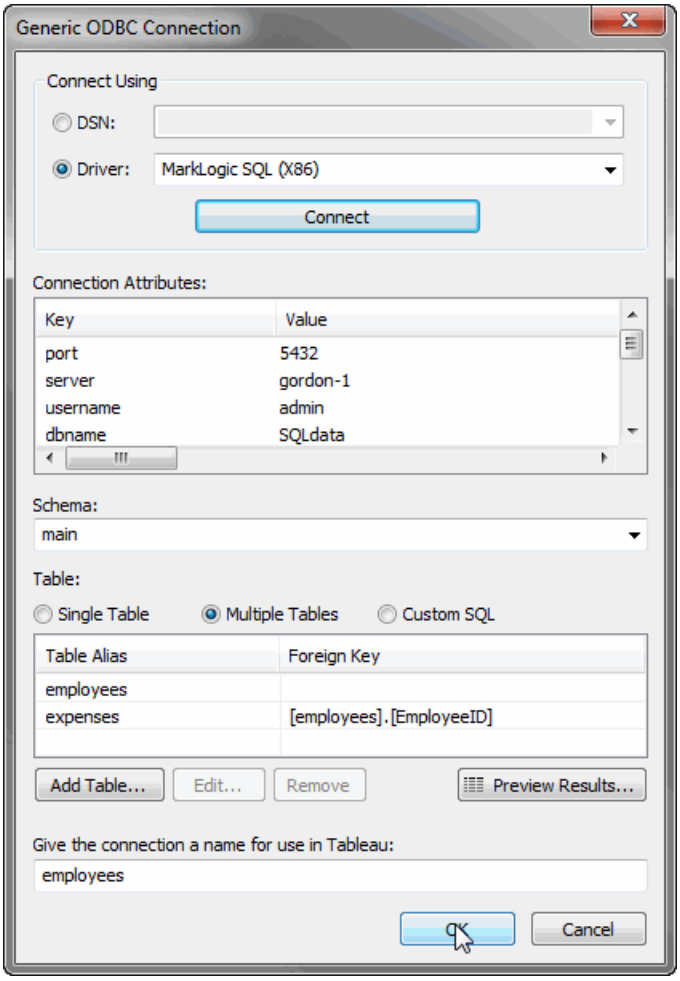


5. Select ‘expenses’ from the list of tables and click Select:

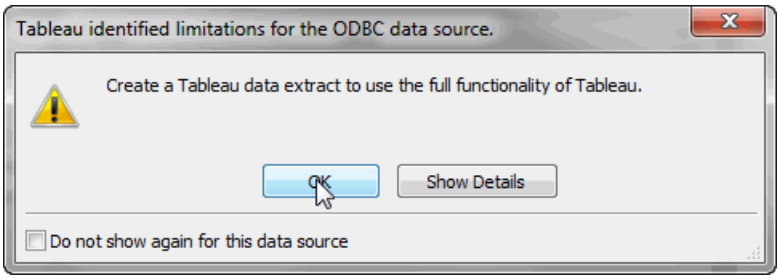


6. Click Ok to close the confirmation window.

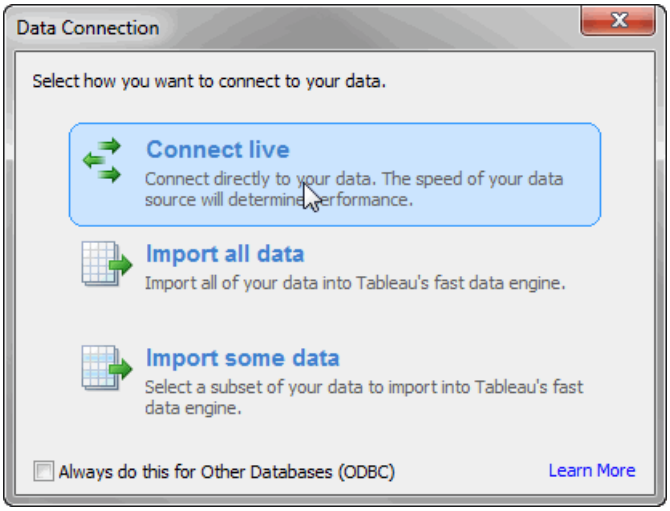
7. In the Generic ODBC Connection dialog, click OK:



8. In the "Create a Tableau data extract to use the full functionality of Tableau prompt," click Ok:

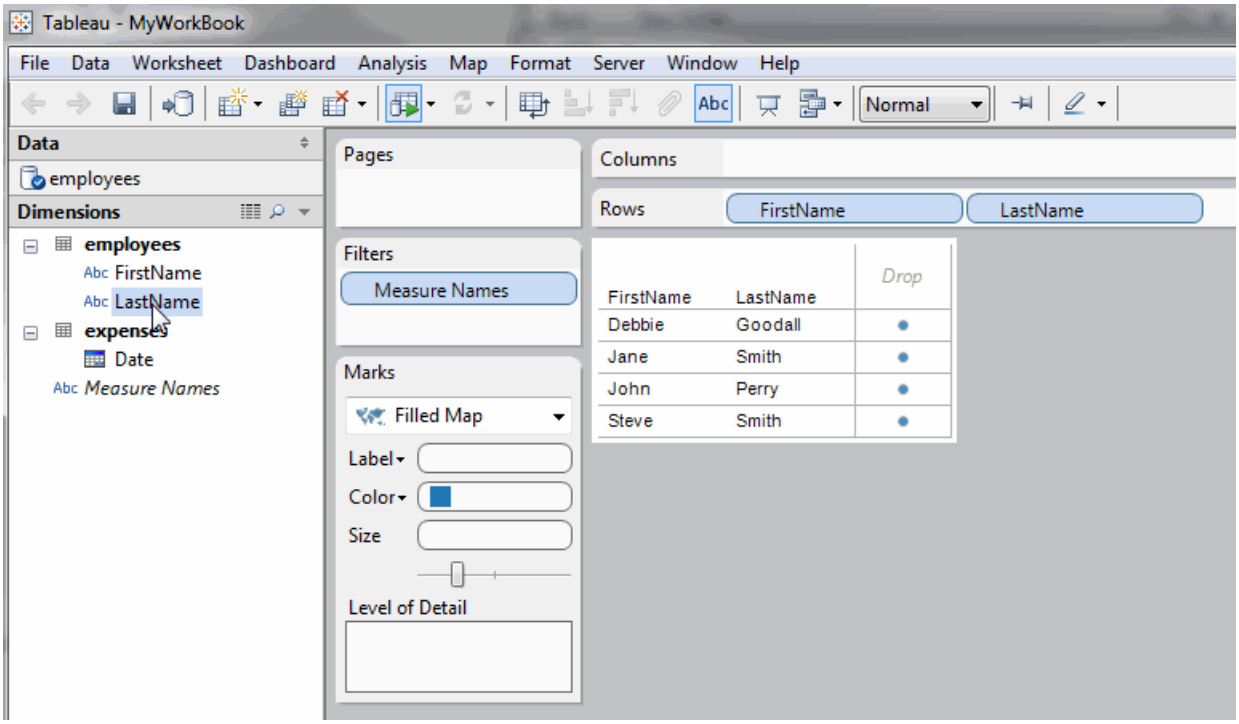


9. In the Data Connection dialog, select the desired data connection type:



Connection Type	Description
Connect live	Creates a direct connection to your data. As reports are generated, data is pulled live from the data source. The speed of your data source will determine performance.
Import all data	Imports the entire data source into Tableau's fast data engine as an extract. The extract is saved with the workbook.
Import some data	Imports a subset of the data into Tableau's fast data engine as an extract. This option requires that you to specify what data you want to extract using filters.

10. In the Data View window, click on FirstName and LastName. You should see your data stored in MarkLogic Server displayed in tables on the right.



6.0 Connecting Cognos to MarkLogic Server

This chapter describes how to set up your MarkLogic Server for SQL.

- Enable Internet Information Services (IIS)
- Connect Cognos to your ODBC Data Source
- Create a New Cognos Project that uses your ODBC Data Source

6.1 Enable Internet Information Services (IIS)

To run Cognos, you must have Internet Information Services (IIS) enabled on your machine. See your Windows documentation for directions.

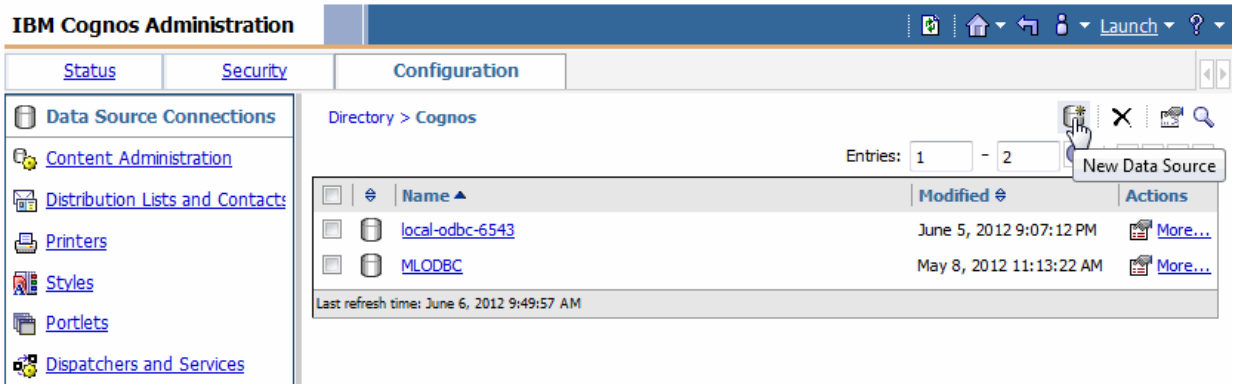
6.2 Connect Cognos to your ODBC Data Source

This section describes how to create a new data source to represent your MarkLogic Server data.

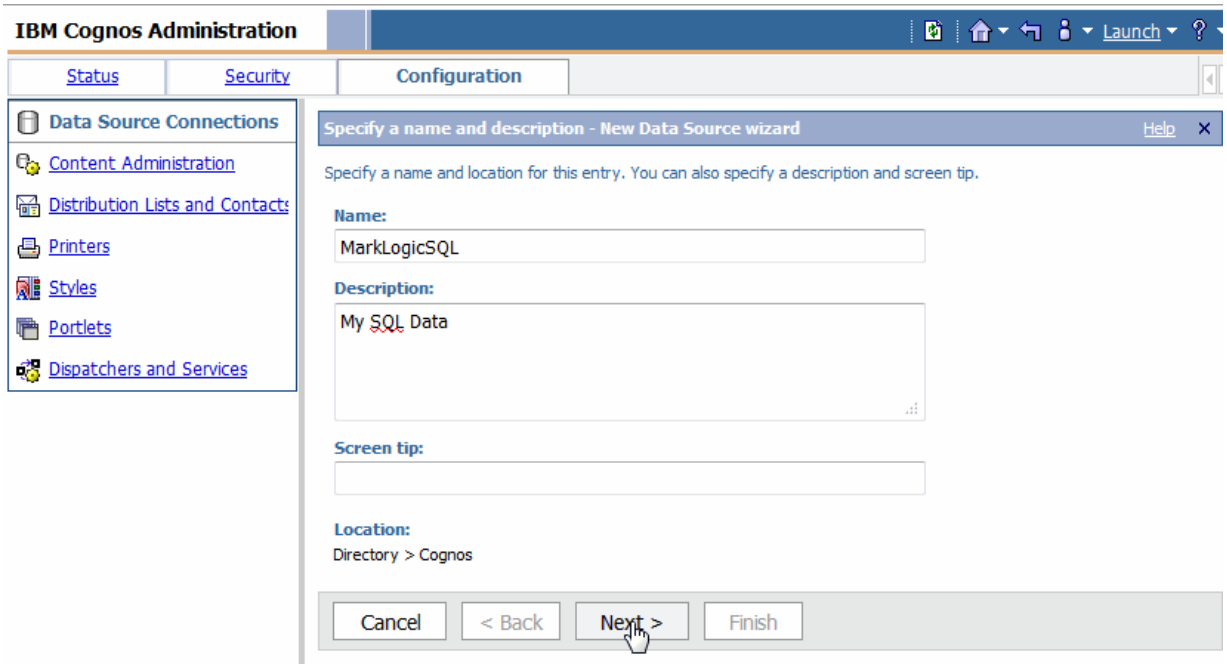
The procedure described in this section assumes you have first installed the MarkLogic ODBC driver and configured it as an ODBC data source on the client server, as described in “Installing and Configuring the MarkLogic Server ODBC Driver” on page 51.

1. Open IBM Cognos Administration

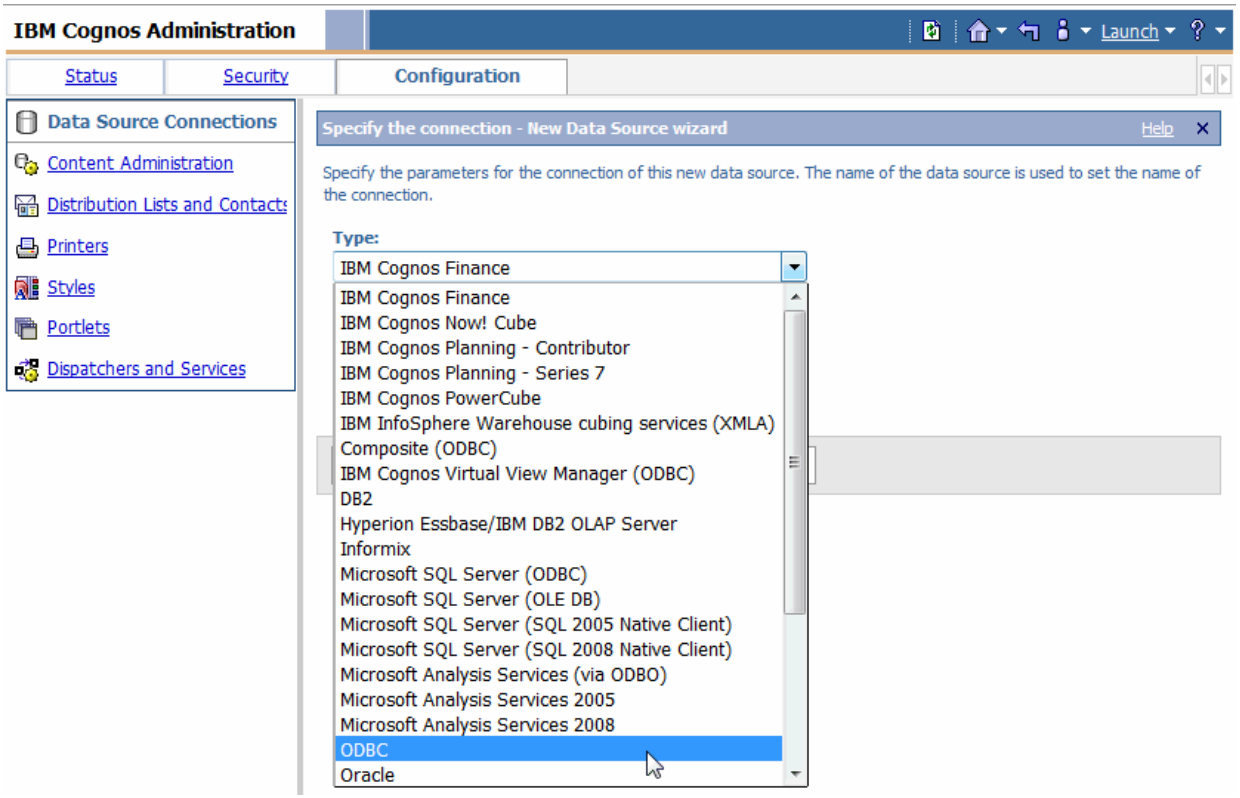
2. Click on the “disk” icon to create a new data source.



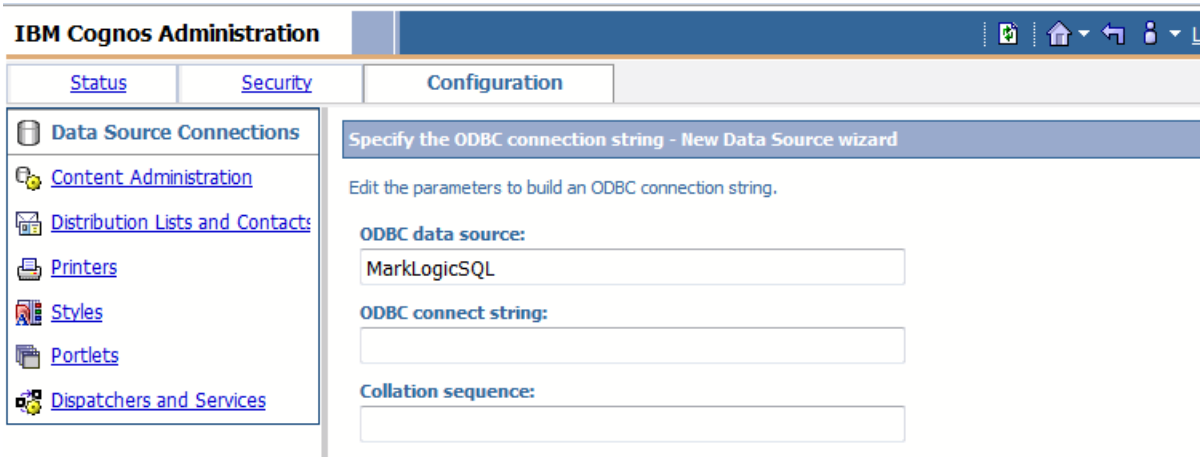
3. Provide a name for the new data source and click Next.



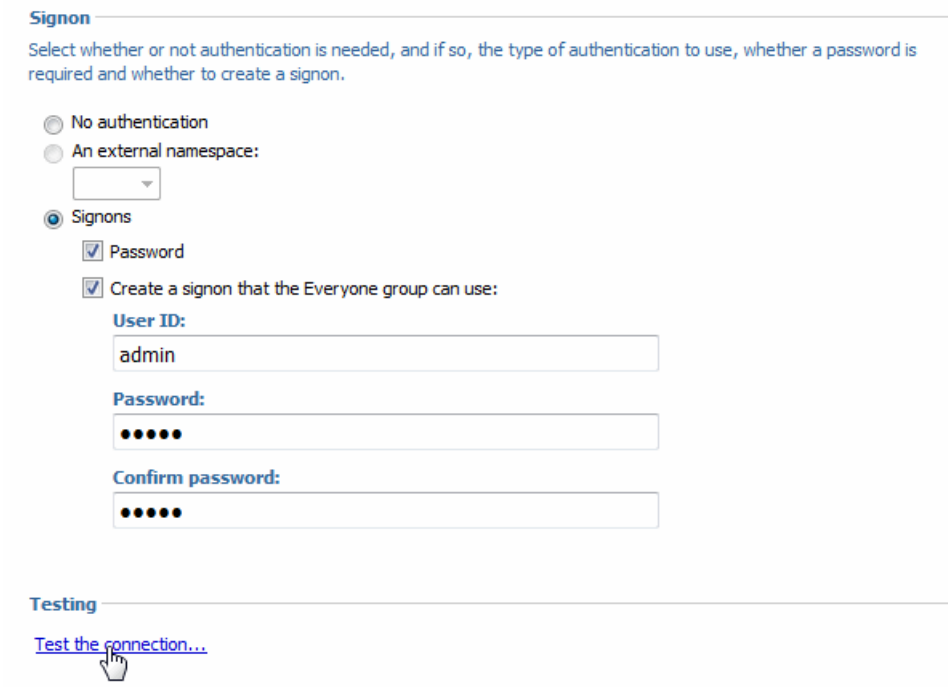
4. Select ‘ODBC’ from the list of Data Source Types.



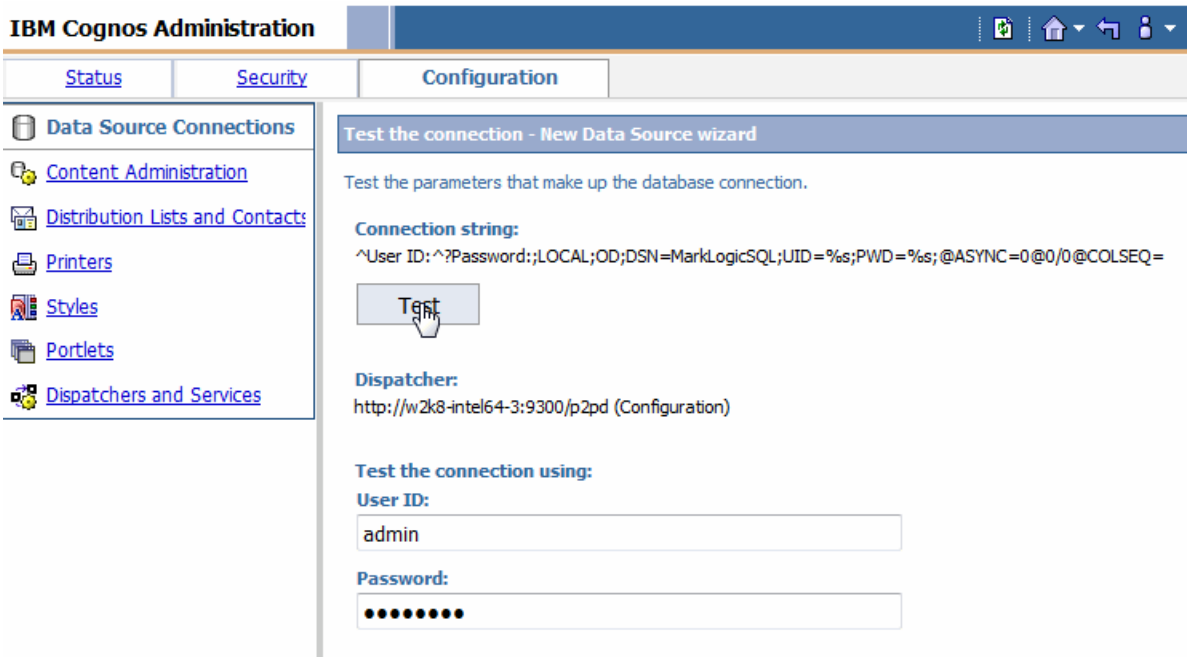
5. Enter ODBC data source name you provided in “Installing and Configuring the MarkLogic Server ODBC Driver” on page 51.



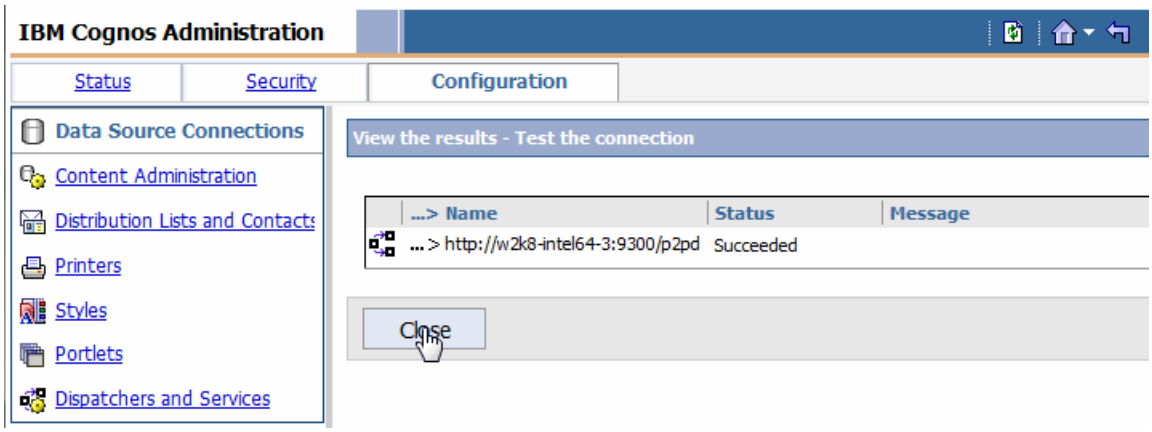
6. Check password and enter your MarkLogic Server login credentials. (You must have the view-admin role on MarkLogic Server). Click Test the connection.



7. Click Test.



8. If the resulting status of the test is ‘Succeeded’, click Close. Otherwise, recheck your settings and retest.



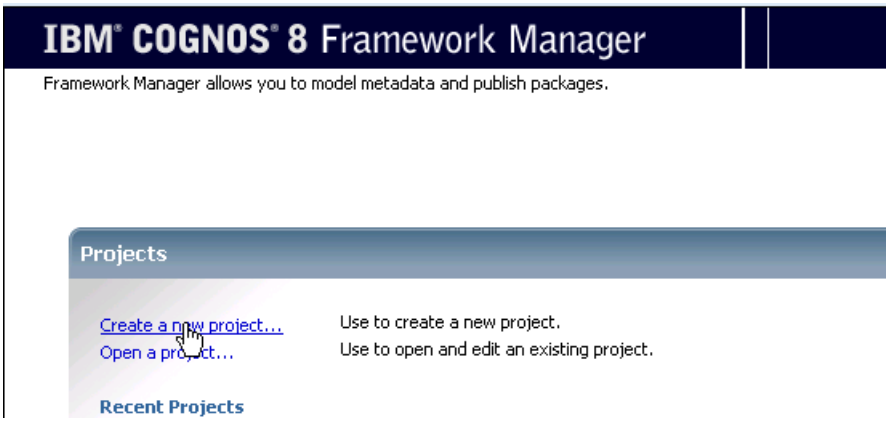
9. Close all remaining windows and click Finish. Your new data source connection will be added to the list.

6.3 Create a New Cognos Project that uses your ODBC Data Source

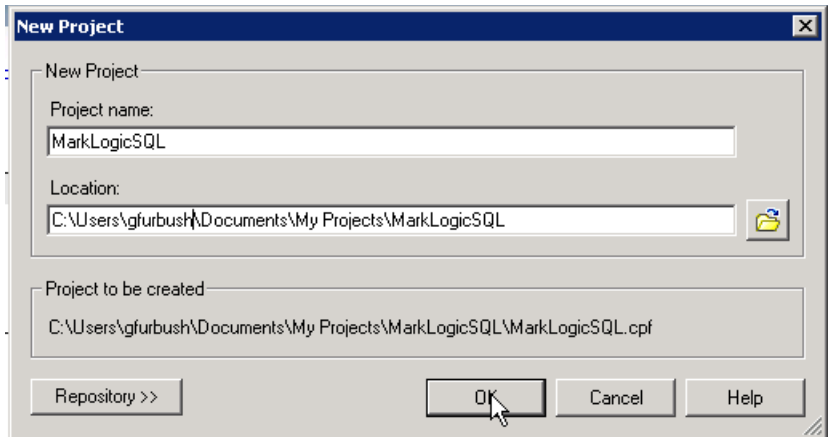
This section describes how to create a new project in Cognos that makes use of the data on MarkLogic Server.

1. Open IBM Framework Manager in your Start menu.

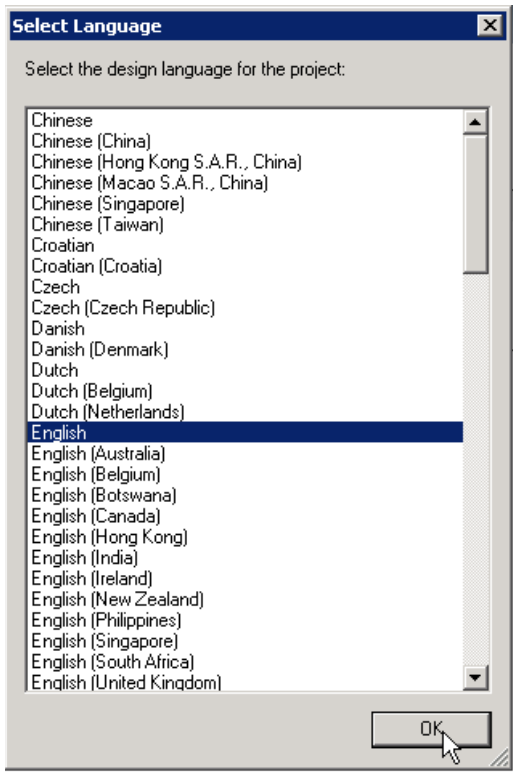
2. Click Create a new project.



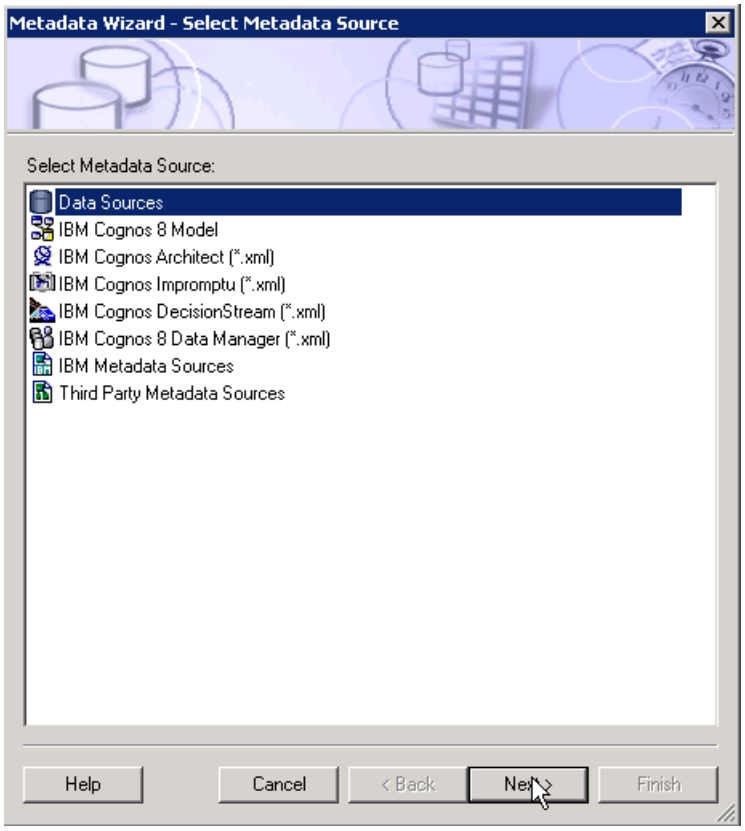
3. Provide a project name and location for the project files. If you receive a notice that 'The directory you specified does not exist. Do you want to create it?', click Ok.



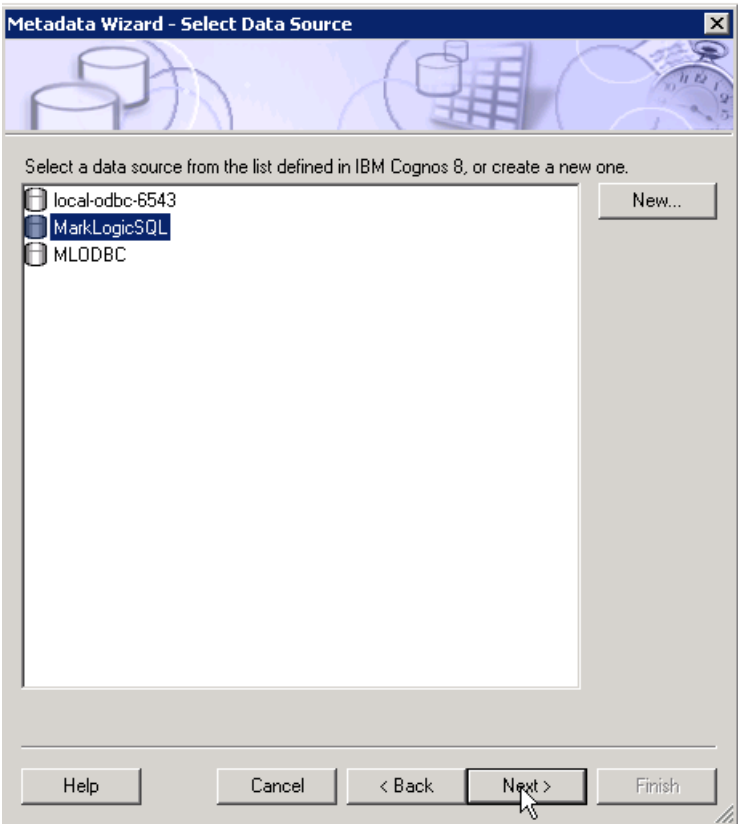
4. Select your language from the list.



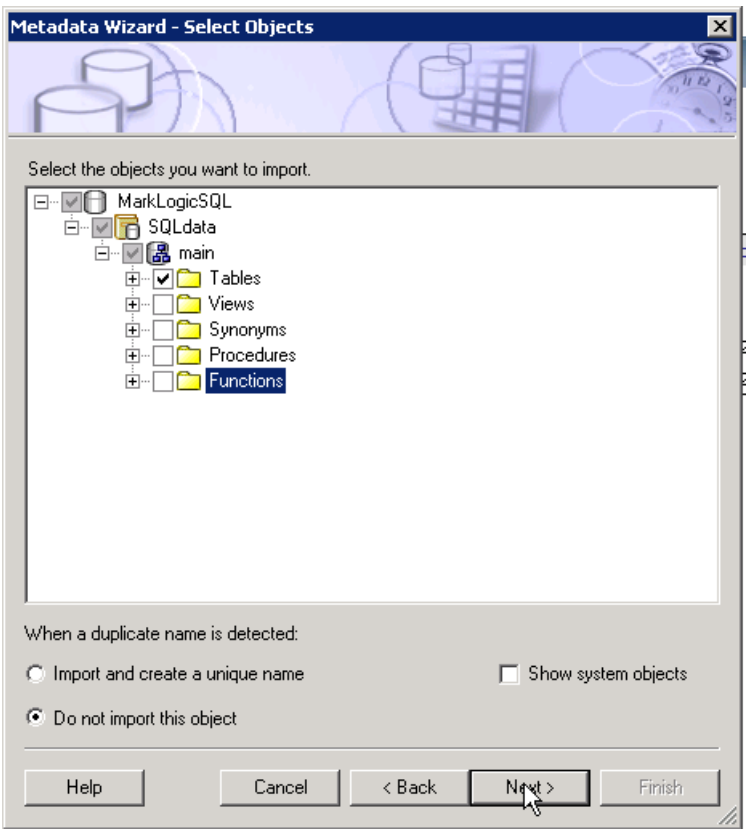
5. In the Metadata Wizard, select Data Sources and click Next.



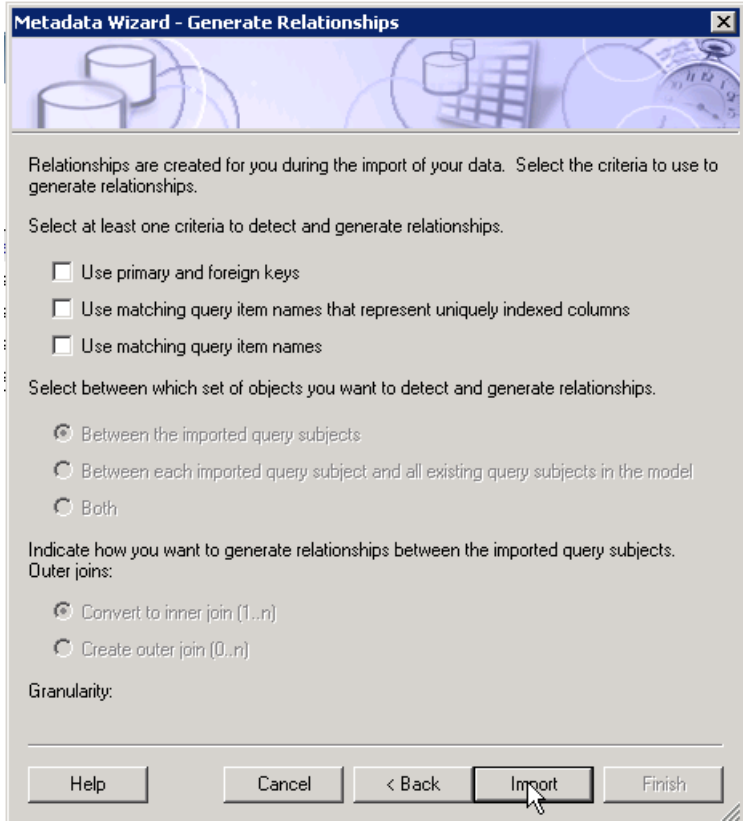
6. Select the data source you created in “Connect Cognos to your ODBC Data Source” on page 66.



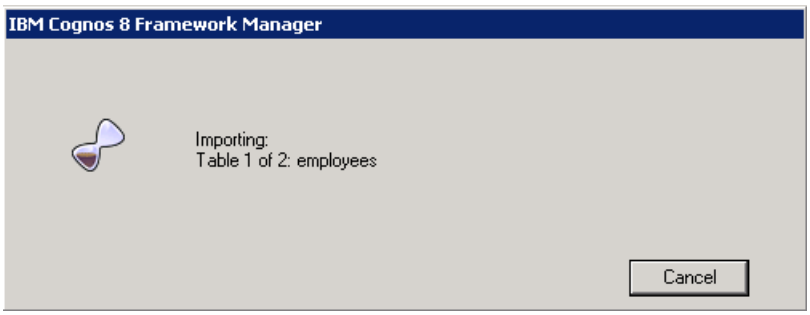
7. In the Select Objects dialog, select your data source and uncheck Views, Synonyms, Procedures, and Functions, leaving only Tables as the selected object under your schema name.



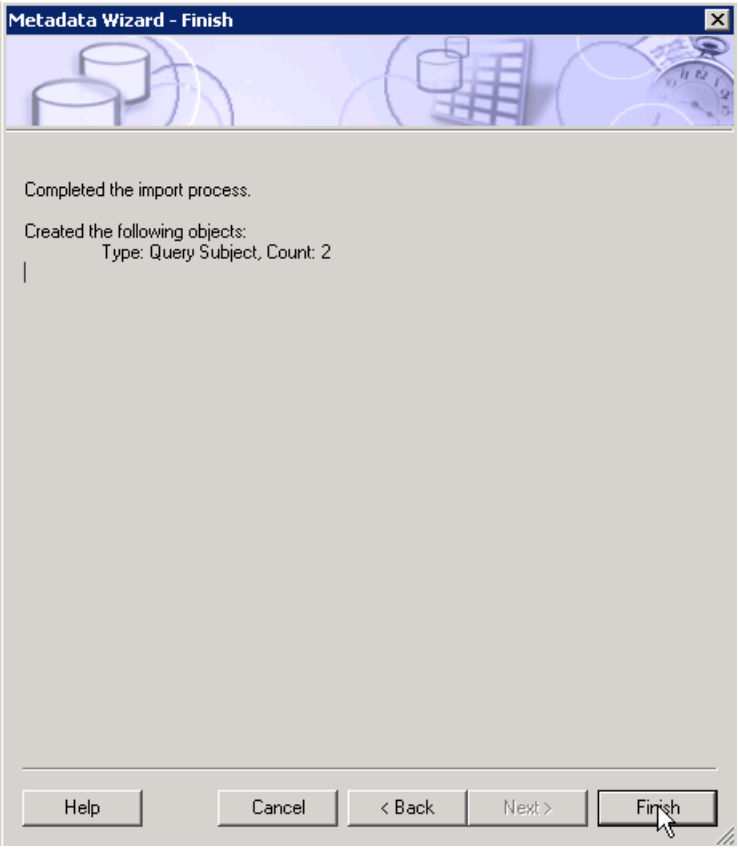
8. In the Generate Relationships dialog, uncheck 'Use primary and foreign keys'. Click Import



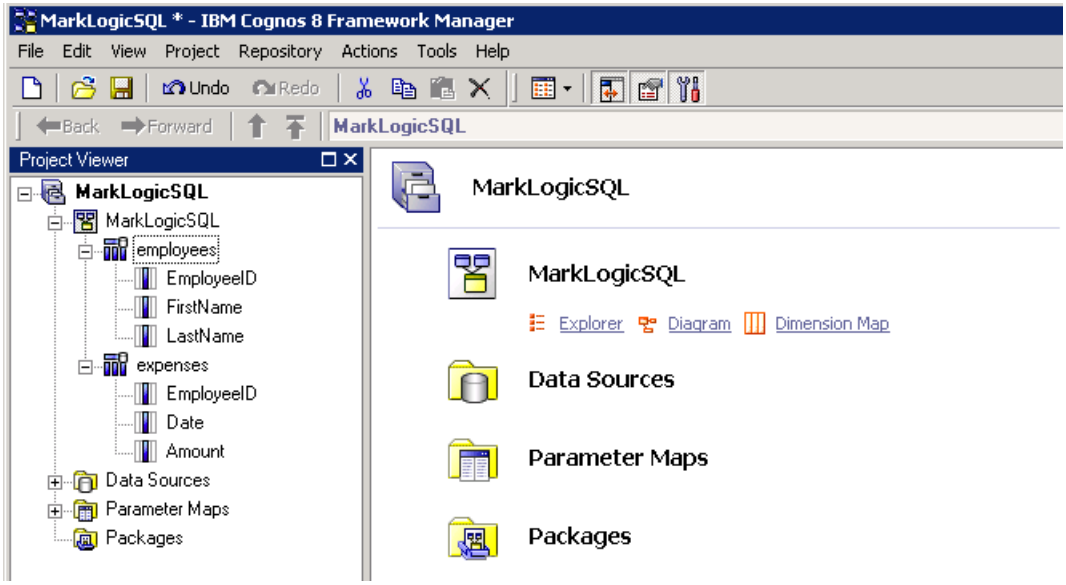
9. You should see an ‘Importing’ pop-up window.



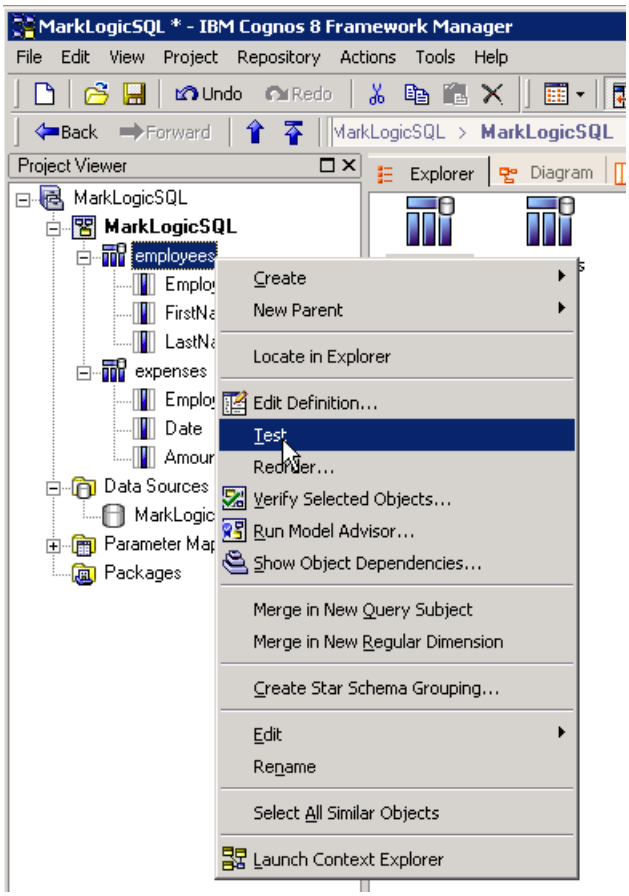
10. When all of the views and data have been imported, you will see a Finish dialog. Click Finish.



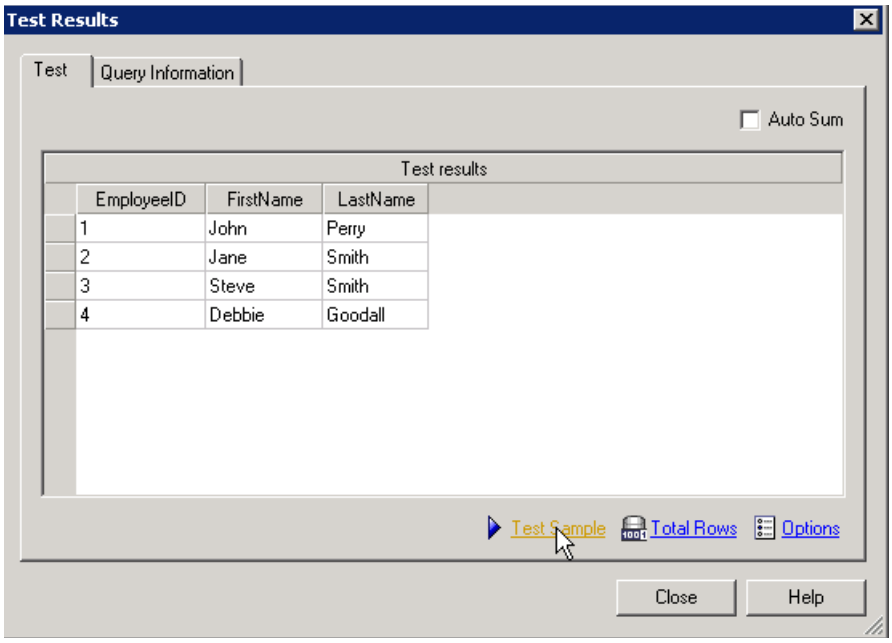
11. The Project Viewer will display the contents of your data source.



12. Right click on a table and select Test from the pop-up menu.



13. Click on Test Sample. If you get results, everything is working correctly.



7.0 SQL Syntax

This chapter describes some of the SQL syntax that are unique to MarkLogic Server.

7.1 DESCRIBE Statement

The DESCRIBE statement can be used to return the contents of a schema or view.

The syntax is:

```
DESCRIBE [SCHEMA] name
```

For example, to return a list of views in the ‘main’ schema, enter:

```
DESCRIBE SCHEMA main
```

To return a list of the columns in the ‘customers’ view, enter:

```
DESCRIBE customers
```

7.1.1 Using DESCRIBE to List Supported Functions

Many MarkLogic functions, such as `xdmp:database-name`, `fn:collection`, `cts:similar-query`, `math:exp`, can be executed in SQL statements. The SQL names of these functions take to form of:

```
namespace_function_name
```

For example, the `xdmp:database-name` function is expressed in SQL as `xdmp_database_name`.

The supported functions are too numerous to list here, but you can obtain a list of all supported functions by entering:

```
DESCRIBE all functions
```

You can narrow the list to scalar or aggregate functions by entering either:

```
DESCRIBE scalar functions
DESCRIBE aggregate functions
```

The following are some examples of the use of MarkLogic functions in SQl statements:

Provide the version of MarkLogic Server and hardware information:

```
select xdmp_version(), xdmp_platform(), xdmp_architecture()
```

Trace the performance of a query:

```
select xdmp_elapsed_time, t1.this, t2.that from t1, t2
    where t1.key=t2.ref group by t1.this
```

Do some trigonometry:

```
select math_cos(radian) from angles
```

Do some geospatial:

```
select cts_distance(town.center, building.location) from town, building
```

Get the MarkLogic home page:

```
select xdmp_http_get("http://www.marklogic.com/")
```

7.2 MATCH Operator

The MarkLogic SQL interface includes a MATCH operator that can be used to perform full-text queries against either a column or a field that is bound to a view, as described in “Creating a View” on page 32.

When the MATCH operator is applied to the whole content, column names are bound to their corresponding range index references and searchable fields are bound to their field names. When the MATCH operator is applied to individual columns, all names are unbound, as it doesn't make sense to constrain searches against one range index to the values of another. These queries are executed in unfiltered mode.

The search expression following the MATCH operator must be contained inside quotes.

> **Note:** Field names, like view and schema names, are treated as case-insensitive for the purposes of duplicate detection and lookup.

7.2.1 Search Grammar

The following table lists the search grammar that can be used by the MATCH operator.

Type	Token
Wildcards*	? % *
Boolean Operators	AND, OR, NOT, NOT_IN, NEAR/*integer*)
Comparison Operators	EQ, NE, LT, LE, GT, GE
Name Binding**	*<field_name>*:*<value>*, *<column_name>*:*<value>*

* To use wildcards in a search expression, you must enable `trailing wildcard searches` and `word lexicons` (`codepoint collation`) on your database.

** Searches are constrained to the named field or column values.

7.2.2 Examples

```
SELECT * FROM employees WHERE employees MATCH "Manager"

SELECT * FROM employees WHERE employees MATCH "position:Manager"

SELECT firstname, lastname FROM employees WHERE employees
    MATCH "employeeid LE 3"

SELECT * FROM messages WHERE messages
    MATCH "cause:numeric AND text:expression"

SELECT * FROM messages WHERE text MATCH "invalid OR wrong OR incorrect"

SELECT * FROM messages WHERE messages
    MATCH "(cause:operand NEAR/10 cause:incompatible)
    AND (correct AND expect)"

SELECT * FROM resumes WHERE resumes
    MATCH 'firstname:M* and (BA or BS)
    NEAR/15 (Pomona NOT_IN "Cal Poly Pomona")'

SELECT * FROM employees WHERE employees MATCH 'firstname:J*'

SELECT * FROM employees WHERE firstname MATCH "J*"
```

7.3 SET/SHOW Statements

MarkLogic Server supports Postgres SET and SHOW run-time configuration parameters, as well as some parameters that are specific to MarkLogic Server. For details on the Postgres parameters, see:

- http://www.postgresql.org/docs/9.1/static/sql-set.html
- http://www.postgresql.org/docs/9.1/static/sql-show.html

All SET parameters are good for the duration of the SQL session in which they are set. Some parameters are read-only and can only be specified by the SHOW statement. These are described in “Read-only SHOW Parameters” on page 85.

> **Note:** All SET string values must be specified in single quotes (`SET parameter 'value'`).

7.3.1 timezone or time zone

Sets the timezone offset to that for the given timezone name. The standard permitted formats and keywords can be used.

For example, to set the timezone to UTC, enter:

```
SET timezone 'UTC'
```

7.3.2 statement_timeout

Sets the timeout for statement execution (milliseconds).

For example:

```
SET statement_timeout 5000
```

7.3.3 lc_messages

Sets the locale for error messages.

For example:

```
SET lc_messages 'en_US'
```

7.3.4 lc_collate

Sets the default collation in the dynamic environment.

The form we will see from the Postgres client is:

```
SET lc_collate 'en_US.utf8'
```

This maps to the collation: http://marklogic.com/collation/en_US

You can also specify a full collation string:

```
SET lc_collation 'http://marklogic.com/collation/en_US/S1/MO'
```

7.3.5 lc_numeric

Sets the locale for formatting numeric values.

For example:

```
set lc_numeric 'de_DE'
```

7.3.6 lc_time

Sets the locale for formatting date/time values.

For example:

```
set lc_time 'en_US.UTF-8'
```

7.3.7 DateType

Sets the output format for dates.

For example:

```
SET DateType 'ISO'
```

7.3.8 extra_float_digits

Sets the number of digits displayed for floating point types.

For example:

```
SET extra_float_digits 2
```

7.3.9 client_encoding or NAMES

Declares the encoding of data coming from the client.

For example:

```
SET client_encoding 'UTF8'
```

SET NAMES is the standard syntax for the same thing.

```
SET NAMES 'UTF8'
```

7.3.10 SCHEMA or search_path

Sets the default schema referenced by names in SQL statements.

For example:

```
SET search_path 'main'
```

7.3.11 mls_default_xquery

Set the default XQuery version.

For example:

```
SET mls_default_xquery '1.0-ml'
```

7.3.12 mls_redundant_check

Enable or disable the SQLITE redundant check on normal (on full-text) query constraints on rows. Value is 1 (enable) or 0 (disable). The default is 0.

For example:

```
SET mls_redundant_check 1;
SELECT title, year FROM songs WHERE year=1991
```

7.4 Read-only SHOW Parameters

The following parameters can be obtained via the SHOW statement but they are read-only and cannot be set via the SET statement.

Parameter	Description
ALL	Return values for all the variables with descriptions (columns=name, setting, description).
lc_ctype	Return the locale for character classifications. For us this is fixed at `zxx.utf8`.
max_function_args	The limit on the number of function arguments. This will be the value of SQLITE_MAX_FUNCTION_ARG, by default `127`.
max_identifier_length	The limit on the length of a name. This will be fixed at `64`.
max_index_keys	The limit on the number of keys in an index. This will be the value of SQLITE_MAX_COLUMN, by default `2000`.
integer_datetimes	Whether the server supports 64-bit date/time values. Fixed at `1`.
server_encoding	The encoding the server uses. Fixed at `UTF-8`.
server_version	The version of MarkLogic Server.
server_version_num	The version of the server expressed as a single integer.

8.0 Technical Support

MarkLogic provides technical support according to the terms detailed in your Software License Agreement or End User License Agreement.

We invite you to visit our support website at http://help.marklogic.com to access information on known and fixed issues, knowledge base articles, and more. For licensed customers with an active maintenance contract, see the Support Handbook for instructions on registering support contacts and on working with the MarkLogic Technical Support team.

Complete product documentation, the latest product release downloads, and other useful information is available for all developers at http://developer.marklogic.com. For general questions, join the general discussion mailing list, open to all MarkLogic developers.

9.0 Copyright

MarkLogic Server 8.0 and supporting products.

NOTICE