MarkLogic Server

SQL Data Modeling Guide

MarkLogic 9
May, 2017

Last Revised: 9.0-2, July, 2017

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 an XML document in the Schemas database and consists of a unique name (which must be unique in the context of a particular schema) and a sequence of column specifications. There are two types of views: *template views* and *range views*.
- 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 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 in 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, SQL datatype, and a value that identifies a particular document element or property.
- 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), to documents in a particular directory, or to documents in a particular collection.
- *Template Driven Extraction* (TDE) is the method used to map documents in a MarkLogic database to SQL views.

  Note: You must have the `tde-admin` and `any-uri` roles to create template views and the `view-admin` role to create range views.

1.2 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. 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 an index in the content database, as described in “Example Template View” on page 9. User access to each view is controlled by a set of permissions, as described in “Template View Security” on page 8.

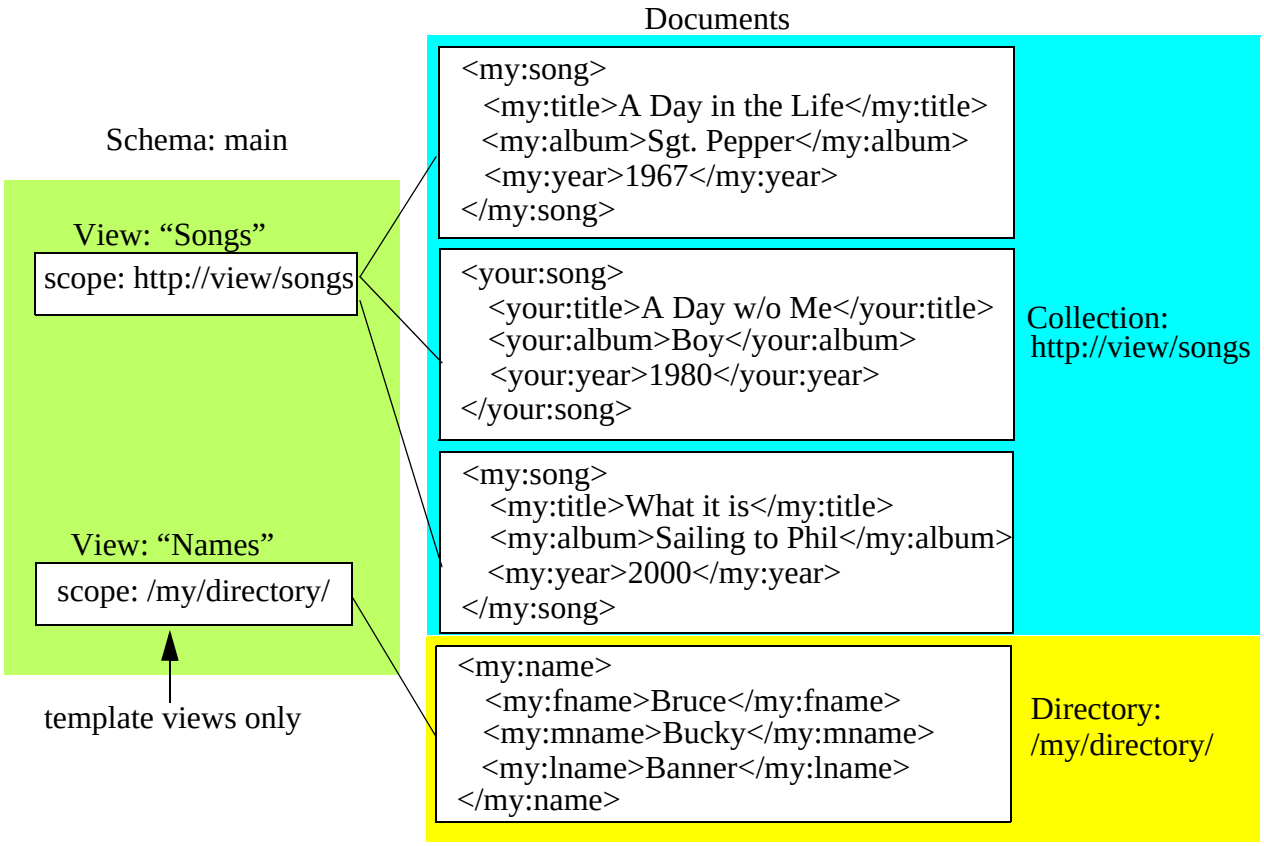
There are two types of views:

- template views: Views that are created by Template Driven Extraction (TDE *templates*). template views are inserted as documents into the schema database associated with the content database. When inserted into a schema database, template views automatically creates triple data in the content database for each column defined in the template and all of the documents are reindexed. Template views can also be created to extract existing triples in documents, rather than elements.
- range views: Views that are based on range indexes and fields. Each column in a view is based on a range index or field in the content database. You must create the range indexes and fields in the content database before creating a range view. Unlike template views, range views allow you to add and remove columns on the view.

  Note: In most situations, you will want to create a template view. Though a range view may be preferable to a template view in some situations, such as for a database already configured with range indexes, they are supported mostly for backwards compatibility with previous versions of MarkLogic. For this reason, most of the dicussion in this guide will be on the use of template views. For details on range views, see “Creating Range Views” on page 50.

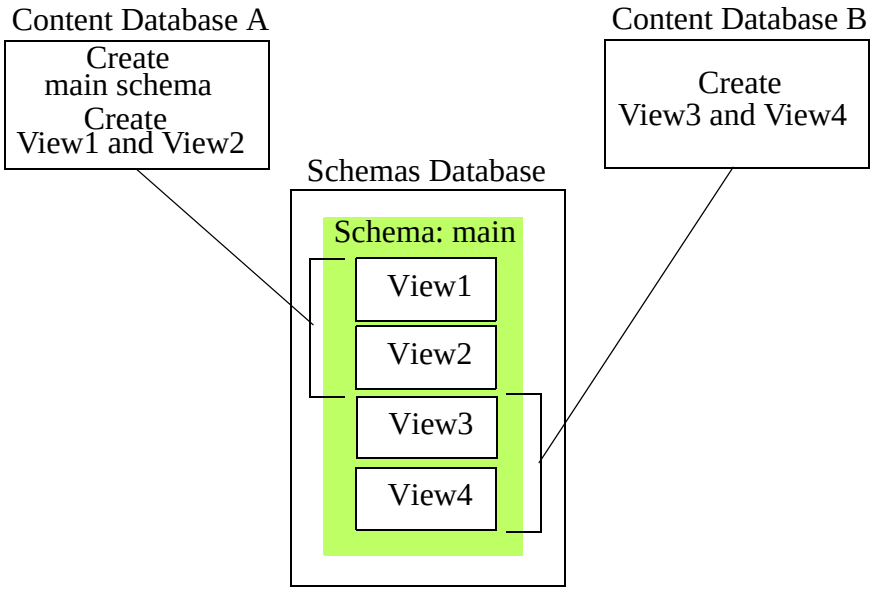
A schema is a naming context for a set of views and user access to each schema can be controlled with a different set of permissions. Each view in a schema must have a unique name. However, you can have multiple views of the same name in different schemas. For example, you can have three views, named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

Each view has a scope that defines the documents from which it reads the column data. The view scope constrains the view to documents located in a particular directory (template views only), or to documents in a particular collection. The figure below shows a schema called ‘main’ that contains two views, each with a different view scope. The view “Songs” is constrained to documents that are in the `http://view/songs` collection and the view “Names” is constrained to documents that are located in the `/my/directory/` directory.

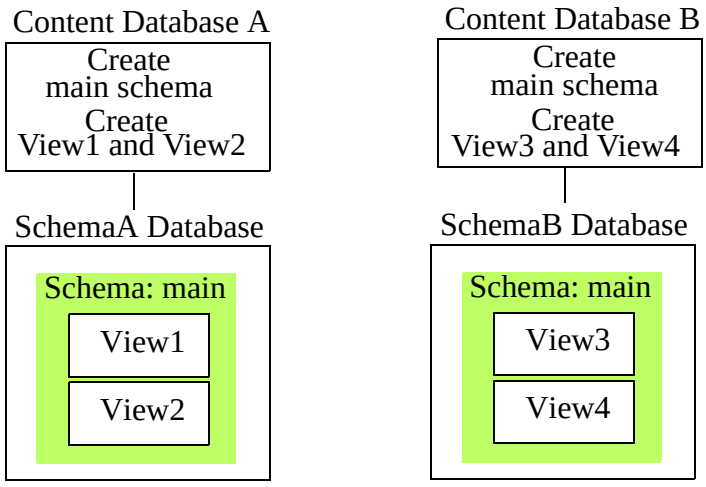


As described above, schemas and views are stored as documents in the schema database associated with the content database for which they are defined. The default schema database is named ‘Schemas.’ If multiple content databases share a single schema database, each content database will have access to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



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.



1.3 Template View Security

The `tde-admin` and `any-uri` roles are required in order to insert a template document into the schema database.

The `tde-view` role is required to access a template view. Access to views can be further restricted by setting additional permissions on the template document that defines the view. Since the same view can be declared in multiple templates loaded with different permissions, the access to views should be controlled at the column level.

Column level read permissions are implicit and are derived from the read permissions set on the template documents. Permissions on a column are not required to be identical and are ORed together. A user with a role that has at least one of the read permissions set on a column will be able to see the column.

If a user does not have permissions on any of the view's columns, the view itself is not visible.

For example, there are two views, as illustrated below.

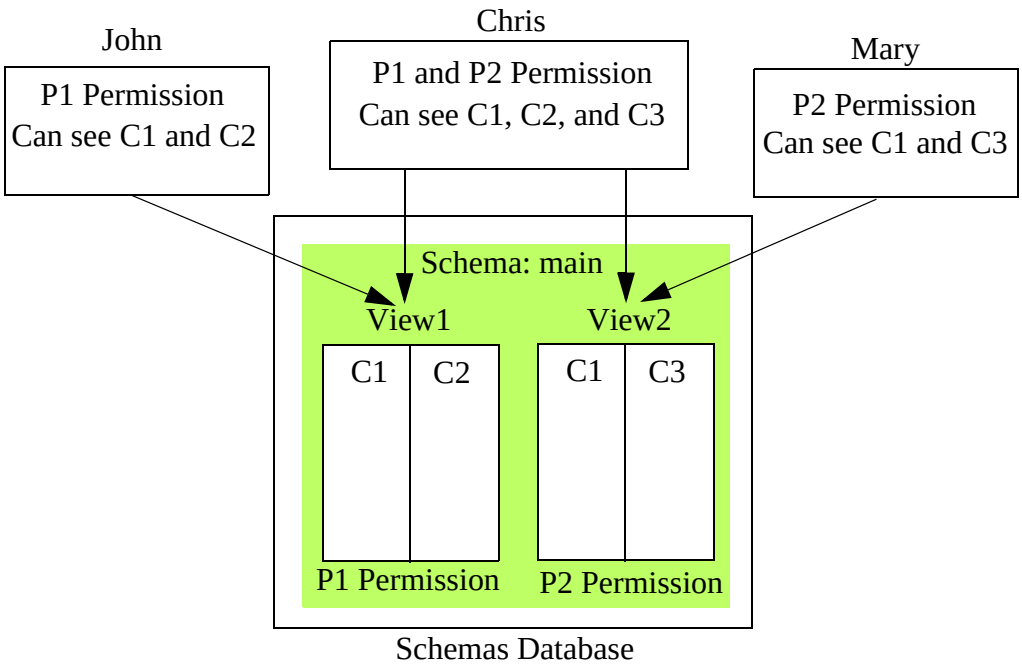
- The View1 template document is configured for Columns C1 and C2 was loaded with P1 Permissions.
- The View2 template document is configured for Columns C1 and C3 was loaded with P2 Permissions.

John has P1 Permissions, so he can see Columns C1 and C2.

Chris has both P1 and P2 Permissions, so he can see Columns C1, C2, and C3.

Mary has P2 Permissions, so she can see Columns C1 and C3.

For details on how to set document permissions, see Protecting Documents in the *Security Guide*.



Note: The MarkLogic SQL engine does not support documents that make use of element-level security. Any document containing protected elements will be skipped by the indexer.

1.4 Example Template View

This section provides an example document and a template view used to extract data from the document and present it in the form of a view.

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>
```

The following template extracts each element and presents it as a column in a view, named ‘book’ in the ‘main’ schema.

```
<template xmlns="http://marklogic.com/xdmp/tde">
  <context>/book</context>
  <rows>
    <row>
      <schema-name>main</schema-name>
      <view-name>book</view-name>
      <columns>
        <column>
         <name>title</name>
          <scalar-type>string</scalar-type>
          <val>title</val>
        </column>
        <column>
         <name>pubyear</name>
          <scalar-type>date</scalar-type>
          <val>pubyear</val>
        </column>
         <column>
          <name>keyword</name>
          <scalar-type>string</scalar-type>
          <val>keyword</val>
        </column>
         <column>
          <name>author</name>
          <scalar-type>string</scalar-type>
          <val>author/name</val>
        </column>
         <column>
          <name>university</name>
          <scalar-type>string</scalar-type>
          <val>author/university</val>
        </column>
         <column>
          <name>cephalopod</name>
          <scalar-type>string</scalar-type>
          <val>body/name[@type="cephalopod"]</val>
        </column>
         <column>
          <name>scombridae</name>
          <scalar-type>string</scalar-type>
          <val>body/name[@type="scombridae"]</val>
        </column>
         <column>
          <name>echinoderm</name>
          <scalar-type>string</scalar-type>
          <val>body/name[@type="echinoderm"]</val>
        </column>
       </columns>
    </row>
```

```
    </rows>
  </template>
```

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:

2.1 Setup MarkLogic Server

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

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:

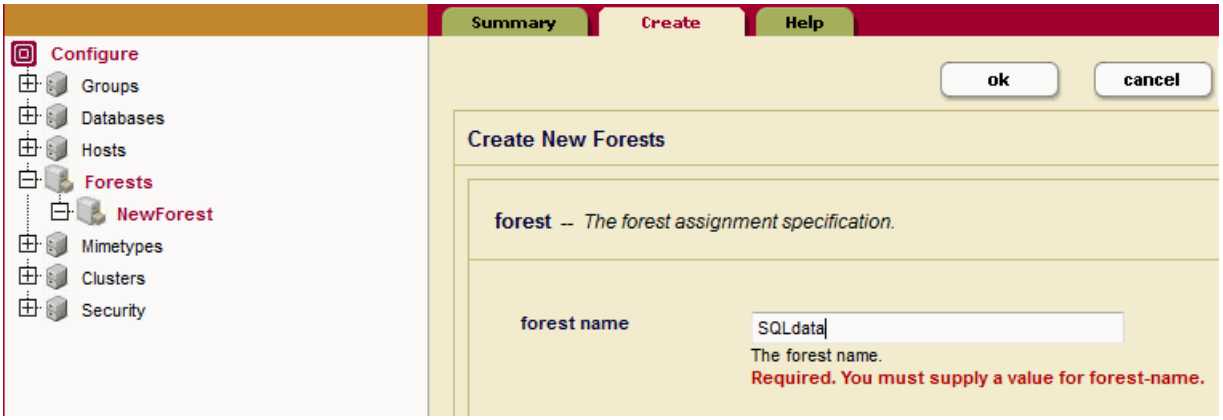
   ```
   http://hostname:8001
   ```

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

2. Click the Forests icon in the left tree menu.

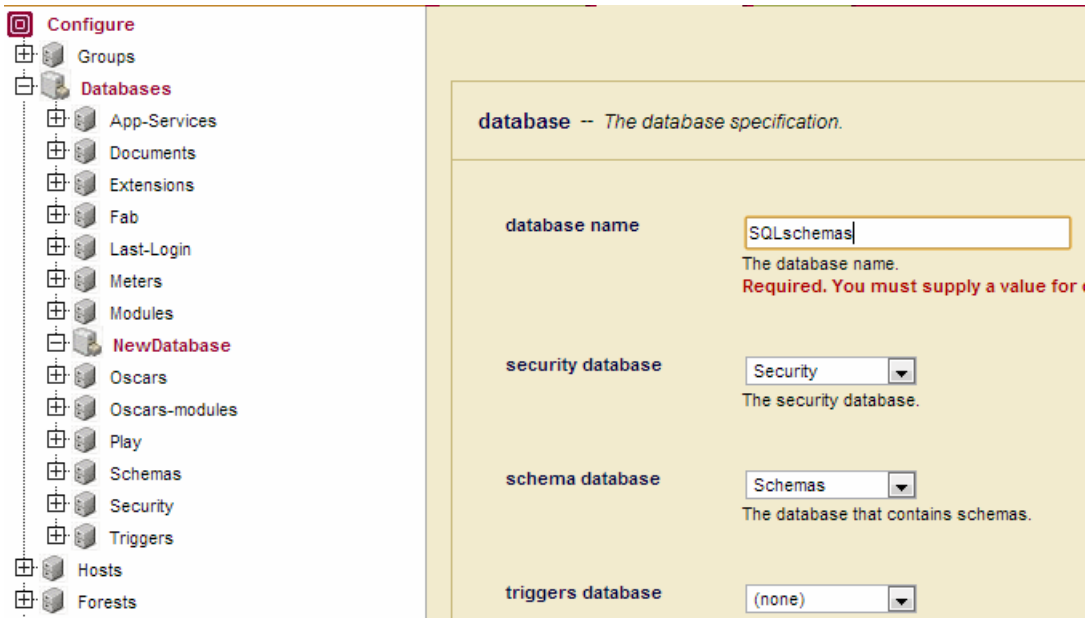
3. Click the Create tab at the top right. The Create Forest page displays. Enter ‘SQLschemas’ as the name of your forest in the Forest Name textbox. Click OK.

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.



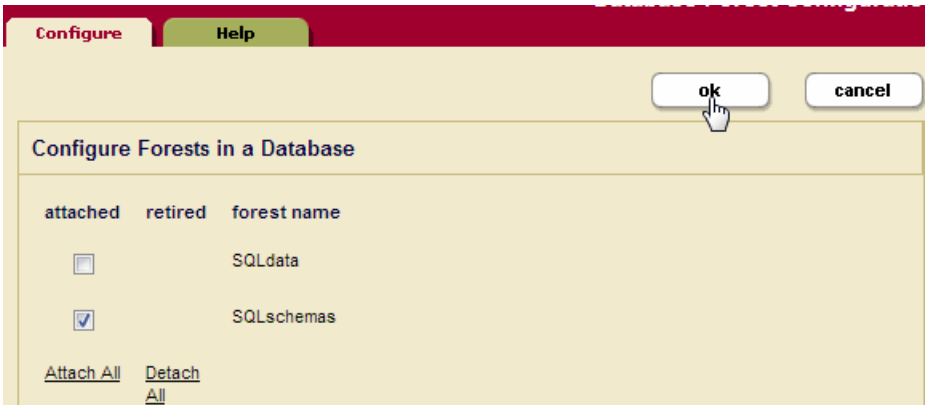
5. Click the Databases icon in the left tree menu.

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:



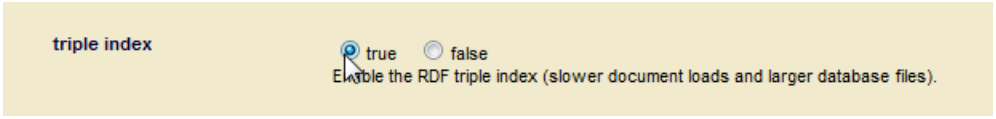
7. At the top of the page click Database->Forests

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.

10. Scroll down the Create Database page to the Triple Index setting and click ‘true’ to enable triple indexing. Click Ok:



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

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

2.1.2 Create an ODBC App Server

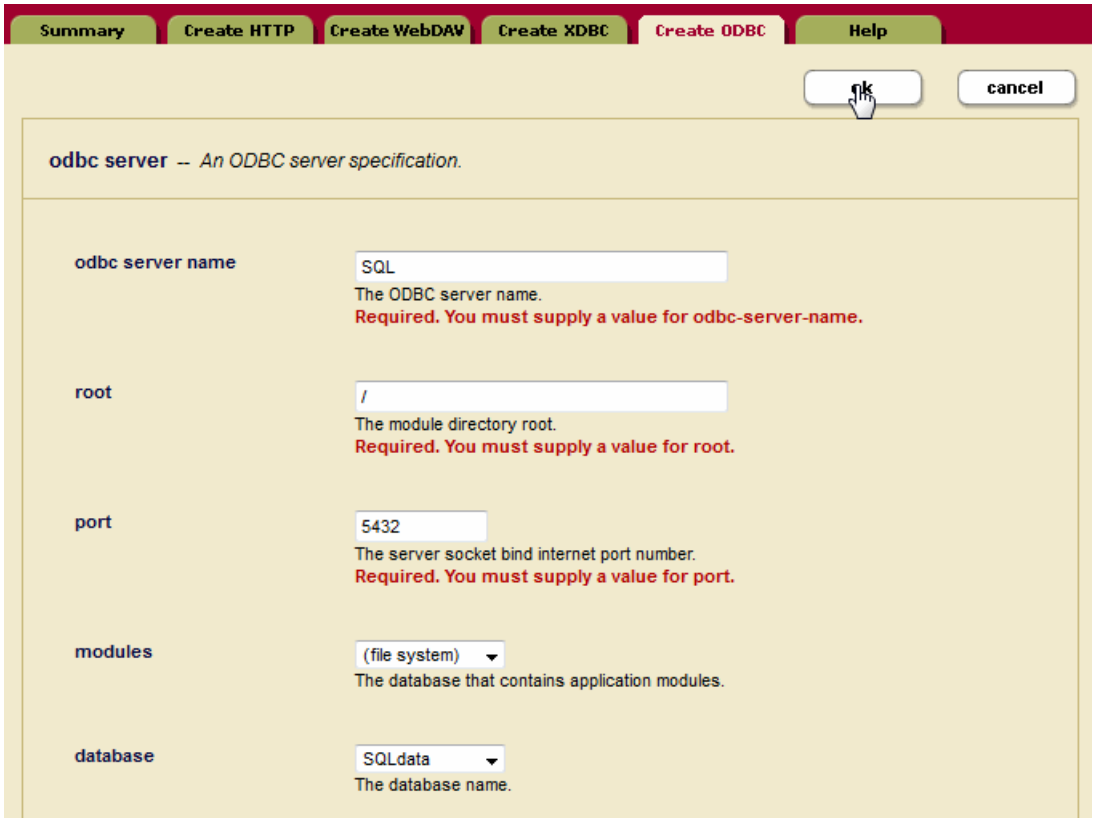
Schemas and views represent content stored in a MarkLogic Server database. Each content database used by a SQL client is managed by an ODBC App Server that accepts SQL queries from the SQL client and responds by returning MarkLogic Server data in tuple form. An ODBC App Server can manage only one content 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 tree menu.
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 12.

2.2 Load the Data

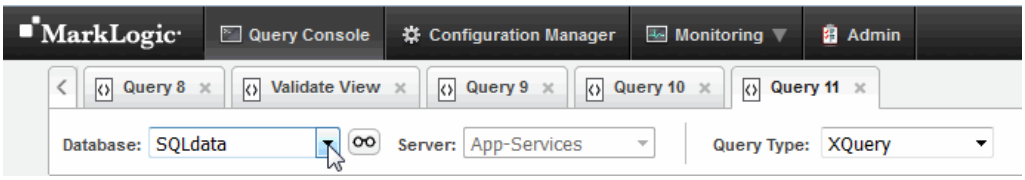
This section describes the procedure for loading the sample documents.

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:

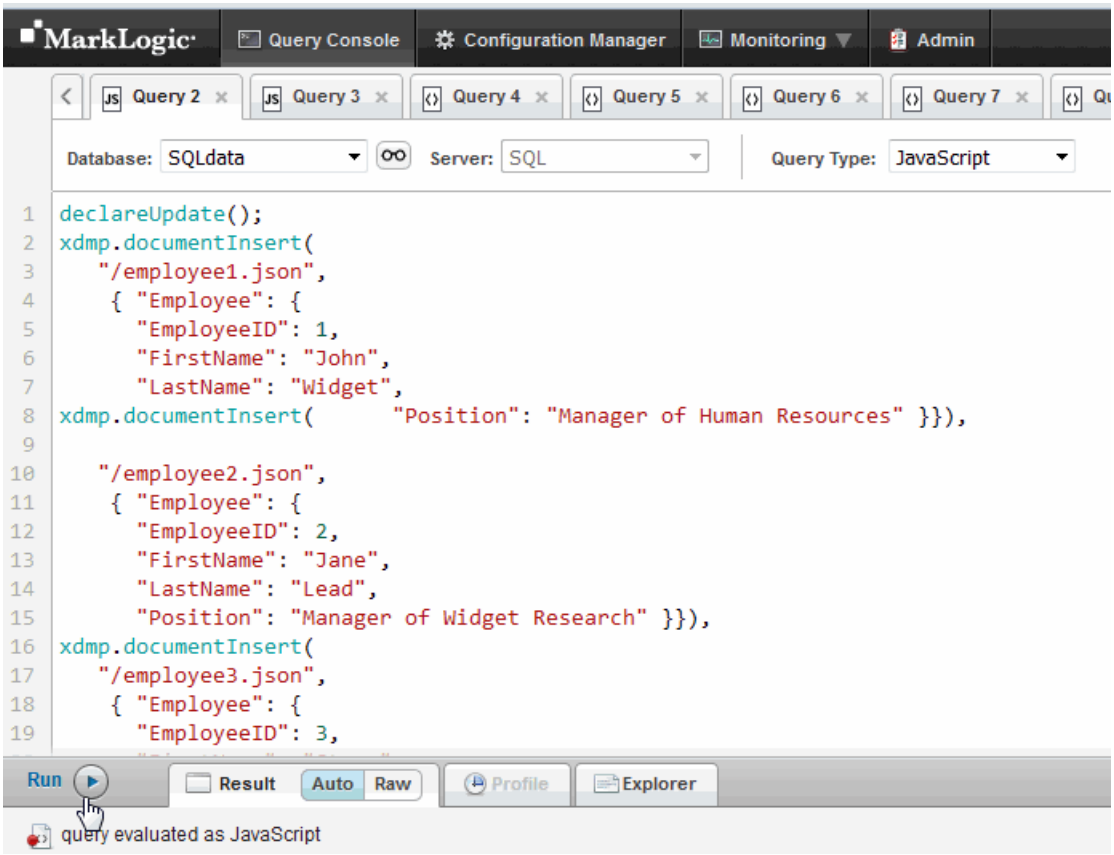
```
declareUpdate();
xdmp.documentInsert(
   "/employee1.json",
    { "Employee": {
      "ID": 1,
      "FirstName": "John",
      "LastName": "Widget",
      "Position": "Manager of Human Resources" }}),
xdmp.documentInsert(
   "/employee2.json",
    { "Employee": {
      "ID": 2,
      "FirstName": "Jane",
      "LastName": "Lead",
      "Position": "Manager of Widget Research" }}),
xdmp.documentInsert(
   "/employee3.json",
    { "Employee": {
      "ID": 3,
      "FirstName": "Steve",
      "LastName": "Manager",
      "Position": "Senior Technical Lead" }}),
xdmp.documentInsert(
   "/employee4.json",
    { "Employee": {
      "ID": 4,
      "FirstName": "Debbie",
      "LastName": "Goodall",
      "Position": "Senior Widget Researcher" }}),
xdmp.documentInsert(
```

```
   "/employee5.json",
    { "Employee": {
      "ID": 14,
      "FirstName": "Lori",
      "LastName": "Baker",
      "Position": "Senior Wingnut" }}),
xdmp.documentInsert(
   "/employee6.json",
    { "Employee": {
      "ID": 15,
      "FirstName": "Steve",
      "LastName": "Lostit",
      "Position": "Mad Scientist" }}),
xdmp.documentInsert(
   "/employee7.json",
    { "Employee": {
      "ID": 16,
      "FirstName": "Donald",
      "LastName": "Putin",
      "Position": "Power Couple" }}),
xdmp.documentInsert(
   "/expense1.json",
    { "Expenses": {
         "EmployeeID": 1,
         "Date": "2012-06-27",
         "Amount": 131.02,
         "Purchase": {
            "Category": "Lodging",
            "Vendor": "Hyatt Hotels",
            "Description": "Exec. King Room"}}}),
 xdmp.documentInsert(
   "/expense2.json",
    { "Expenses": {
         "EmployeeID": 2,
         "Date": "2012-06-27",
         "Amount": 155.22,
         "Purchase": {
            "Category": "Transportation",
            "Vendor": "Alaska",
            "Description": "SFO > SEA"}}}),
xdmp.documentInsert(
   "/expense3.json",
    { "Expenses": {
         "EmployeeID": 1,
         "Date": "2012-08-03",
         "Amount": 59.95,
         "Purchase": {
            "Category": "Meals",
            "Vendor": "Doug's Dinner",
            "Description": "Dinner"}}}),
xdmp.documentInsert(
   "/expense4.json",
    { "Expenses": {
         "EmployeeID": 3,
```

```
            "Date": "2012-05-07",
            "Amount": 162.95,
            "Purchase": {
               "Category": "Lodging",
               "Vendor": "Hilton Hotels",
               "Description": "Exec. Suite"}}}),
xdmp.documentInsert(
   "/expense5.json",
    { "Expenses": {
            "EmployeeID": 3,
            "Date": "2012-05-30",
            "Amount": 120.00,
            "Purchase": {
               "Category": "Lodging",
               "Vendor": "Kingsman Motel",
               "Description": "Reg Room"}}}),
xdmp.documentInsert(
   "/expense6.json",
    { "Expenses": {
            "EmployeeID": 4,
            "Date": "2012-03-23",
            "Amount": 155.55,
            "Purchase": {
               "Category": "Lodging",
               "Vendor": "Waterfront Hotel",
               "Description": "Queen Room"}}}),
xdmp.documentInsert(
   "/expense7.json",
    { "Expenses": {
            "EmployeeID": 4,
            "Date": "2012-06-05",
            "Amount": 104.29,
            "Purchase": {
               "Category": "Meals",
               "Vendor": "Good Eats",
               "Description": "Client Lunch"}}}),
xdmp.documentInsert(
   "/GoodEats.json",
    { "ApprovedVendor": {
            "Name": "Good Eats",
            "Address": {
               "Street": "707 Oxford Rd.",
               "City": "Ann Arbor",
               "Region": "MI",
               "PostalCode": "48104",
               "PostalCode": "USA",
               "Phone": "(313) 555-5735"}}}),
xdmp.documentInsert(
   "/WaterfrontHotel.json",
    { "ApprovedVendor": {
            "Name": "Waterfront Hotel",
            "Address": {
               "Street": "1000 Coast Rd.",
               "City": "Santa Cruz",
```

```
            "Region": "CA",
            "PostalCode": "94330",
            "PostalCode": "USA",
            "Phone": "(831) 745-8913"}}}),
xdmp.documentInsert(
   "/KingsmanMotel.json",
    { "ApprovedVendor": {
         "Name": "Kingsman Motel",
         "Address": {
            "Street": "4832 Frankster St.",
            "City": "Renor",
            "Region": "NV",
            "PostalCode": "88660",
            "PostalCode": "USA",
            "Phone": "(702) 436-3785"}}}),
xdmp.documentInsert(
   "/Hilton.json",
    { "ApprovedVendor": {
         "Name": "Hilton Hotels",
         "Address": {
            "Street": "555 Market St.",
            "City": "San Francisco",
            "Region": "CA",
            "PostalCode": "94033",
            "PostalCode": "USA",
            "Phone": "(415) 540-8732"}}}),
xdmp.documentInsert(
   "/Hyatt.json",
    { "ApprovedVendor": {
         "Name": "Hyatt Hotels",
         "Address": {
            "Street": "9023 Caterberry Ave.",
            "City": "Seattle",
            "Region": "WA",
            "PostalCode": "56445",
            "PostalCode": "USA",
            "Phone": "(206) 321-3152"}}}),
xdmp.documentInsert(
   "/MealLimit.json",
    { "ExpenseLimit": {
         "Category": "Meals",
         "Limit": 100}}),
xdmp.documentInsert(
   "/LodgingLimit.json",
    { "ExpenseLimit": {
         "Category": "Lodging",
         "Limit": 300}}),
xdmp.documentInsert(
   "/TransLimit.json",
    { "ExpenseLimit": {
         "Category": "Transportation",
         "Limit": 200}})
```

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



2.3 Create Template Views

This section describes how to use the XQuery API to create the template views used by SQL queries.

1. Create a template view in the `main` schema, named `employees`. Specify the `Employee` element as the context and columns for `EmployeeID`, `FirstName`, `LastName`, and `Position`. Use `tde:template-insert` to insert the template document into the `SQLschemas` database as `/employees.xml`. Run the script with `SQLdata` selected in the Database menu.

```
xquery version "1.0-ml";

import module namespace tde = "http://marklogic.com/xdmp/tde"
       at "/MarkLogic/tde.xqy";

let $employees :=
<template xmlns="http://marklogic.com/xdmp/tde">
  <context>/Employee</context>
  <rows>
    <row>
      <schema-name>main</schema-name>
      <view-name>employees</view-name>
      <columns>
        <column>
          <name>EmployeeID</name>
          <scalar-type>int</scalar-type>
          <val>ID</val>
        </column>
        <column>
          <name>FirstName</name>
          <scalar-type>string</scalar-type>
          <val>FirstName</val>
        </column>
        <column>
          <name>LastName</name>
          <scalar-type>string</scalar-type>
          <val>LastName</val>
        </column>
        <column>
          <name>Position</name>
          <scalar-type>string</scalar-type>
          <val>Position</val>
        </column>
       </columns>
    </row>
  </rows>
</template>

return tde:template-insert("/employees.xml", $employees)
```

2. Create a second view in the `main` schema, named `expenses`, with a scope on the `Expenses` element as the context and columns for `EmployeeID`, `Date`, and `Amount`. Use `tde:template-insert` to insert the template document into the `SQLschemas` database as `/expenses.xml`.

```
xquery version "1.0-ml";

import module namespace tde = "http://marklogic.com/xdmp/tde"
       at "/MarkLogic/tde.xqy";

let $expenses :=
<template xmlns="http://marklogic.com/xdmp/tde">
  <context>/Expenses</context>
  <rows>
    <row>
      <schema-name>main</schema-name>
      <view-name>expenses</view-name>
      <columns>
        <column>
          <name>EmployeeID</name>
          <scalar-type>int</scalar-type>
          <val>EmployeeID</val>
        </column>
        <column>
          <name>Date</name>
          <scalar-type>date</scalar-type>
          <val>Date</val>
        </column>
        <column>
          <name>Category</name>
          <scalar-type>string</scalar-type>
          <val>Purchase/Category</val>
        </column>
        <column>
          <name>Vendor</name>
          <scalar-type>string</scalar-type>
          <val>Purchase/Vendor</val>
        </column>
        <column>
          <name>Amount</name>
          <scalar-type>decimal</scalar-type>
          <val>Amount</val>
        </column>
       </columns>
    </row>
  </rows>
</template>

return tde:template-insert("/expenses.xml", $expenses)
```

3. Create a two more views in the `main` schema, named `approvedvendor` and `expenselimit` as follows.

```
xquery version "1.0-ml";

import module namespace tde = "http://marklogic.com/xdmp/tde"
       at "/MarkLogic/tde.xqy";

let $vendors :=
<template xmlns="http://marklogic.com/xdmp/tde">
  <context>ApprovedVendor</context>
  <rows>
    <row>
      <schema-name>main</schema-name>
      <view-name>approvedvendor</view-name>
      <columns>
        <column>
          <name>Vendor</name>
          <scalar-type>string</scalar-type>
          <val>Name</val>
        </column>
        <column>
          <name>City</name>
          <scalar-type>string</scalar-type>
          <val>Address/City</val>
        </column>
       </columns>
    </row>
  </rows>
</template>

return tde:template-insert("/vendors.xml", $vendors);
```

```
xquery version "1.0-ml";

import module namespace tde = "http://marklogic.com/xdmp/tde"
       at "/MarkLogic/tde.xqy";

let $limits :=
<template xmlns="http://marklogic.com/xdmp/tde">
  <context>ExpenseLimit</context>
  <rows>
    <row>
      <schema-name>main</schema-name>
      <view-name>expenselimit</view-name>
      <columns>
        <column>
          <name>Category</name>
          <scalar-type>string</scalar-type>
          <val>Category</val>
        </column>
        <column>
          <name>Limit</name>
          <scalar-type>decimal</scalar-type>
          <val>Limit</val>
        </column>
       </columns>
    </row>
  </rows>
</template>

return tde:template-insert("/limits.xml", $limits)
```

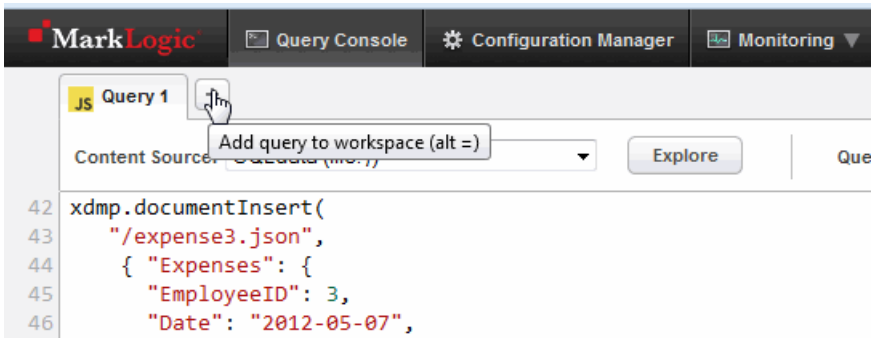
4. List the views that you just created.

```
tde:get-view("main","employees"),
tde:get-view("main","expenses"),
tde:get-view("main","approvedvendor"),
tde:get-view("main","expenselimit")
```

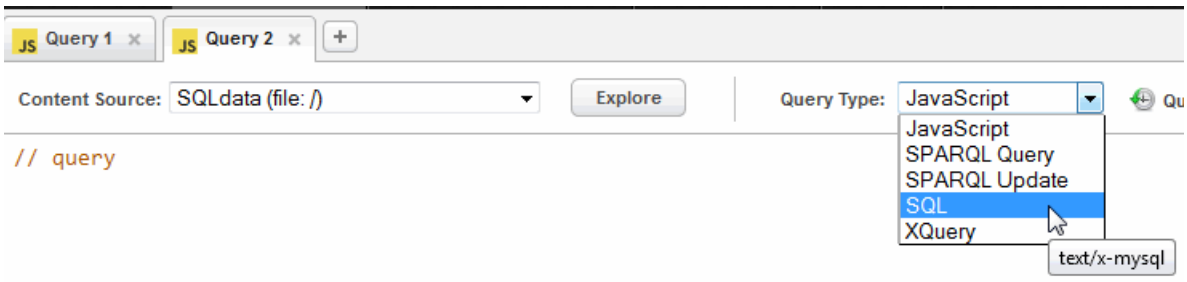
Note: If you change a template view, you must reindex your content database.

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:

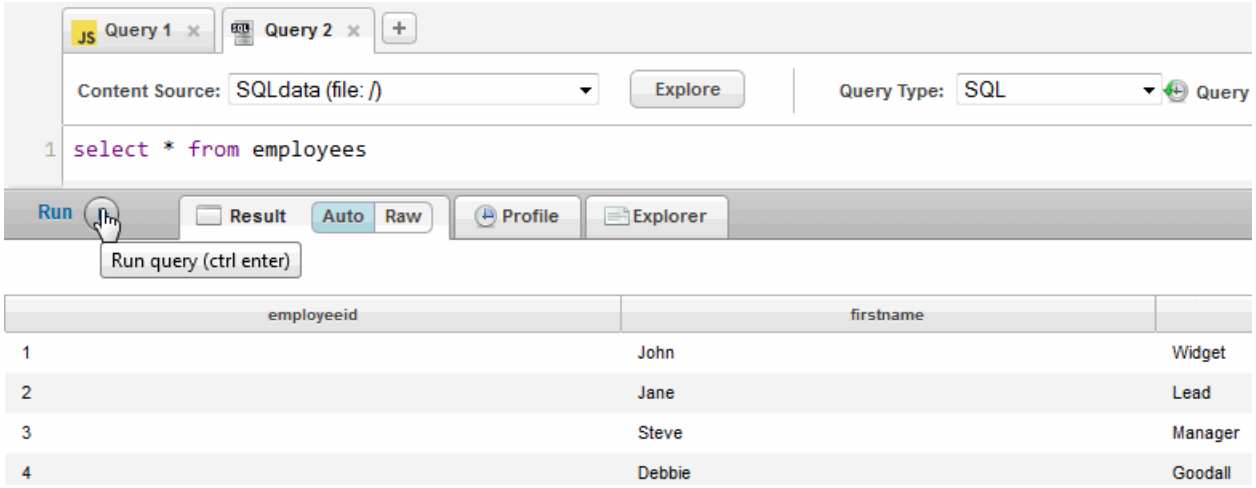


Enter the following query:

```
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 "Configuring the ODBC Driver on Linux" on page 73.

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 main.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 Qlik to MarkLogic Server” on page 84 and “Connecting Tableau to MarkLogic Server” on page 75.

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 Creating Template Views

MarkLogic allows you to define a template view that specifies which parts of the document make up a row in a view, and then query that view from a server-side program with `xdmp:sql`, `mlsql`, or ODBC. You can also query that view server-side from the MarkLogic Optic API, which is a fluent JavaScript and XQuery interface with the ability to perform joins and aggregates on views over documents. Template views are a simple, powerful way to specify a relational lens over documents, making parts of your document data accessible via SQL. The Optic API gives developers idiomatic JavaScript and XQuery access to relational operations over rows, combined with rich document search. The Optic API is described in the Optic API for Relational Operations chapter in the *Application Developer's Guide*.

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

The focus of this chapter is on the template elements that are specific to creating views. The Template Driven Extraction (TDE) chapter in the *Application Developer's Guide* describes the template elements that are common to all types of data-extraction templates.

This chapter contains the following topics:

3.1 Template View Elements

A template view contains the following elements and their child elements:

Element	Description
`description`	Optional description of the template.

`collections`
`  collection`
`  collections-and`
`  collection`
	Optional collection scopes. Multiple collection scopes can be ORed or ANDed. For details, see Collections in the *Application Developer's Guide*.

`directories` `  directory`	Optional directory scopes. Multiple directory scopes are ORed together. For details, see Directories in the *Application Developer's Guide*.
`vars` `  var`	Optional intermediate variables extracted at the current context level. For details, see Variables in the *Application Developer's Guide*.

`rows`
`  row`
`  schema-name`
`  view-name`
`  view-layout`
`  sparse`
`  identical`
`  columns`
`  column`
`  name`
`  scalar-type`
`  val`
`  nullable`
`  default`
`  invalid-values`
`  ignore`
`  reject`
`  reindexing`
`  hidden`
`  visible`
`  collation`
	`rows` is a sequence of `row` descriptions and mappings, as described in "Row" on page 36.
	
	`columns` is sequence of `column` descriptions and mappings, as described in "Columns" on page 38.
	
	`scalar-type` is the type for the `val`. See Type Casting in the *Application Developer's Guide* for details.

Element	Description
`templates` `  template`	Optional sequence of sub-templates. “Creating Views from Multiple Templates” on page 44 and “Creating Views from Nested Templates” on page 47. **Note:** You cannot use `tde:template-insert` to insert multiple templates created by means of the `<templates>` element. You must use `xdmp:document-insert` instead.
`path-namespaces` `  path-namespace`	Optional sequence of namespace bindings. For details, see path-namespaces in the *Application Developer's Guide*..
`context`	The lookup node that is used for template activation and data extraction. For details, see Context in the *Application Developer's Guide*.
`enabled`	A boolean that specifies whether the template is enabled (`true`) or disabled (`false`). Default value is `true`.

The `context`, `vars`, and `columns` identify XQuery elements or JSON properties by means of path expressions. Path expressions are based on XPath, which is described in XPath Quick Reference in the *XQuery and XSLT Reference Guide* and Traversing JSON Documents Using XPath in the *Application Developer's Guide*.

3.1.1 Row

A row definition contains:

- A unique `schema-name` to which the view belongs.

  Note: This schema cannot contain Range Views.

- A unique `view-name` specifies the target view. Extracted rows under a `row` section are added to its target view. Multiple templates can reference the same target view.
- A `view-layout` element:
  - If its value is set to `identical` (default), the view declaration must be consistent between templates; same column names, column data types, and column nullability.
  - A value of `sparse` is useful when you use more than one template to define a view. For example, you may want a view that has multiple contexts and an optional column that matches some, but not all, of the documents in the database.target view declaration can have other nullable columns not listed under the current `row`.

For example, if a view is referenced in template T1 using columns (A,B,C) and in template T2 using columns (A,B,D), the resulting view will have all 4 columns (A,B,C,D). Column A and B are present in both T1 and T2 and can be declared as non nullable. However, columns C and D must be nullable. For an example, see "Creating Views from Multiple Templates" on page 44.

- A sequence of column descriptions each specifying a column `name`, data type (`scalar-type`) and data mapping (`val`). See "Columns" on page 38. The `scalar-type` is the type for the `val`. See Type Casting in the *Application Developer's Guide*.

For example:

```
<row>
  <schema-name>main</schema-name>
  <view-name>expenses</view-name>
  <columns>
    <column>
      <name>EmployeeID</name>
      <scalar-type>int</scalar-type>
      <val>EmployeeID</val>
    </column>
    <column>
      <name>Date</name>
      <scalar-type>date</scalar-type>
      <val>Date</val>
    </column>
    <column>
      <name>Amount</name>
      <scalar-type>decimal</scalar-type>
      <val>Amount</val>
    </column>
   </columns>
</row>
```

3.1.2 Columns

A column definition contains:

- The column `name`. A column is uniquely identifiable by its schema, view, and column names.
- The last data projection into the column described inside the `val` element. The simplest form of projection is a child node under the current context, like `EmployeeID` in the example above `<val>EmployeeID</val>`. See Template Dialect and Data Transformation Functions in the *Application Developer's Guide* for the types of expressions allowed in a `val`.
- The column's SQL datatype `scalar-type`. The result of the `val` expression is automatically casted to the specified scalar type. Users do not have to explicitly create the result in the target datatype. See Type Casting in the *Application Developer's Guide*.
- By default, a column is not nullable. However, you can allow a column to have no values by adding `<nullable>true</nullable>` to the corresponding column element. You can specify a default value for a column by adding `<default>value</default>`. A null value will be replaced by the default value.
- An `invalid-values` element that controls the behavior when cell values cannot be coerced to their datatype:
  - If `invalid-values` is set to `reject` (default). The server should error out and indexing should stop.
  - If `invalid-values` is set to `ignore`, the entire row is skipped if any non-nullable column has a non-castable value. For nullable columns, a cell with a non-castable value is set to null.

The following table describes the results from the possible combinations of `ignore` and `reject` on `nullable` and non-nullable (`<nullable>false</nullable>`) columns. The Default Value column specifies whether or not a `default` value is specified for the column. The Invalid Input column describes what happens when the cell value cannot be coerced to the specified datatype. The Missing Input column describes what happens when there is no value available to populate the column.

Invalid Values	nullability	Default Value	Invalid Input	Missing Input
ignore	nullable	no default	skip cell	skip cell
		default value	default value	default
	non-nullable	no default	skip row	skip row
		default value	default value	default value

Invalid Values	nullability	Default Value	Invalid Input	Missing Input
reject	nullable	no default	rejected	skip cell
		default value	rejected	default value
	non-nullable	no default	rejected	rejected
		default value	rejected	rejected

For example:

```
<column>
  <name>EmployeeID</name>
  <scalar-type>int</scalar-type>
  <val>EmployeeID</val>
</column>

<column>
  <name>EmployeeID</name>
  <scalar-type>int</scalar-type>
  <val>EmployeeID</val>
  <nullable>true</nullable>
  <invalid-values>ignore</invalid-values>
</column>

<column>
  <name>Brand</name>
  <scalar-type>string</scalar-type>
  <val>Brand</val>
  <default>generic</default>
  <invalid-values>reject</invalid-values>
</column>

<column>
  <name>SSN</name>
  <scalar-type>string</scalar-type>
  <val>id[@root='2.16.840.1.113883.4.1']/@extension</val>
</column>

<column>
  <name>Name</name>
  <scalar-type>string</scalar-type>
  <val>concat(patient/name/given[1],' ',patient/name/family)</val>
</column>
```

3.1.3 Defining View Scope

The scope of the view is used to constrain the view to the documents in particular collections or directories. The scope is optional, so do not specify a scope if you elect not to set the scope of the view.

For details on defining a template view scope, see Collections and Directories in the Template Driven Extraction (TDE) chapter in the *Application Developer's Guide*

3.2 Example Documents

The template views described below are written to extract data from documents, like the XML medical document shown below

In XQuery, insert the following document:

```
let $med :=
<Citation Status="Completed">
  <ID>69152893</ID>
  <PMID>5717905</PMID>
  <Article>
    <Journal>
      <ISSN>0043-5341</ISSN>
      <JournalIssue>
        <Volume>118</Volume>
        <Issue>49</Issue>
        <PubDate>
          <Year>1968</Year>
          <Month>12</Month>
          <Day>7</Day>
        </PubDate>
      </JournalIssue>
    </Journal>
    <ArticleTitle>
       The Influence of Calcium on Cholesterol in Human Serum
    </ArticleTitle>
    <AuthorList>
      <Author>
        <LastName>Doe</LastName>
        <ForeName>John</ForeName>
      </Author>
      <Author>
        <LastName>Smith</LastName>
        <ForeName>Jane</ForeName>
      </Author>
    </AuthorList>
  </Article>
</Citation>

return xdmp:document-insert("med1.xml", $med)
```

In JavaScript, insert the following document:

```
declareUpdate();

xdmp.documentInsert(
   "med2.json",
    { "Journal": {
    "Issue": 103,
    "Title": "Bone Density Studies",
    "Date": "8/1/2009",
    "Author": "John Simson" }})
```

3.3 Example View Templates

This section shows two templates, one in XML and one in JSON, that define a view on the document in “Example Documents” on page 40. The view templates are:

- XML View Template
- JSON View Template

3.3.1 XML View Template

The following XML view template creates a “Publications” view in the “Medical” schema.

```
xquery version "1.0-ml";
import module namespace tde = "http://marklogic.com/xdmp/tde"
        at "/MarkLogic/tde.xqy";

let $ClinicalView :=
<template xmlns="http://marklogic.com/xdmp/tde">
  <description>populates patients' data</description>
  <context>/Citation/Article</context>
  <rows>
    <row>
      <schema-name>Medical</schema-name>
      <view-name>Publications</view-name>
      <columns>
        <column>
          <name>ID</name>
          <scalar-type>long</scalar-type>
          <val>../ID</val>
        </column>
        <column>
          <name>ISSN</name>
          <scalar-type>string</scalar-type>
          <val>Journal/ISSN</val>
        </column>
        <column>
          <name>Volume</name>
          <scalar-type>string</scalar-type>
          <val>Journal/JournalIssue/Volume</val>
          <nullable>true</nullable>
        </column>
        <column>
          <name>Date</name>
          <scalar-type>string</scalar-type>
          <val>Journal/JournalIssue/PubDate/Year||'-'
             ||Journal/JournalIssue/PubDate/Month||'-'
             ||Journal/JournalIssue/PubDate/Day</val>
          <nullable>true</nullable>
        </column>
      </columns>
    </row>
  </rows>
</template>
return tde:template-insert("Template.xml", $ClinicalView)
```

3.3.2 JSON View Template

The following JSON view template creates a “Publications” view in the “Medical” schema.

```
declareUpdate();
var tde = require("/MarkLogic/tde.xqy");
var ClinicalView = xdmp.toJSON(

{
  "template":{
    "context":"/Citation/Article",
    "rows":[
      {
        "schemaName":"Medical",
        "viewName":"Publications",
        "columns":[
          {
            "name":"ID",
            "scalarType":"long",
            "val":"../ID"
          },
          {
            "name":"ISSN",
            "scalarType":"string",
            "val":"Journal/ISSN"
          },
          {
            "name":"Volume",
            "scalarType":"string",
            "val":"Journal/JournalIssue/Volume"
          },
          {
            "name":"Date",
            "scalarType":"string",
            "val":"Journal/JournalIssue/PubDate/Year||'-' \
                 ||Journal/JournalIssue/PubDate/Month||'-' \
                 ||Journal/JournalIssue/PubDate/Day"
          }
        ]
      }
    ]
  }
}
);

tde.templateInsert("Template.json", ClinicalView);
```

3.4 Creating Views from Multiple Templates

You can create a single view from multiple templates. For example, if you want to create a view to support more than one context or scope. The templates below create a Publications view with columns that are scoped for the two different documents shown in “Example Documents” on page 40. The result is that a single query on the Publications view will populate the relavent columns from each document. For example, the ‘Title’ column will be populated with the value of the `<ArticleTitle>` element in the `med1.xml` file and the `Title` property in the `med2.json` file.

Note that both templates have the `view-layout` set to `strict` and that the `ISSN`, and `Author` columns are flagged as `nullable`.

Insert the first version of the Pulications view template as follows:

```
xquery version "1.0-ml";
import module namespace tde = "http://marklogic.com/xdmp/tde"
        at "/MarkLogic/tde.xqy";

let $ClinicalView :=
<template xmlns="http://marklogic.com/xdmp/tde">
  <description>populates patients' data</description>
  <context>/Citation/Article</context>
  <rows>
    <row>
      <schema-name>Medical</schema-name>
      <view-name>Publications</view-name>
      <view-layout>sparse</view-layout>
      <columns>
        <column>
          <name>ISSN</name>
          <scalar-type>string</scalar-type>
          <val>Journal/ISSN</val>
          <nullable>true</nullable>
        </column>
        <column>
          <name>Title</name>
          <scalar-type>string</scalar-type>
          <val>ArticleTitle</val>
        </column>
        <column>
          <name>Volume</name>
          <scalar-type>string</scalar-type>
          <val>Journal/JournalIssue/Volume</val>
        </column>
        <column>
          <name>Date</name>
          <scalar-type>string</scalar-type>
          <val>Journal/JournalIssue/PubDate/Month||'/'
             ||Journal/JournalIssue/PubDate/Day||'/'
             ||Journal/JournalIssue/PubDate/Year</val>
          <nullable>true</nullable>
        </column>
```

```
      </columns>
    </row>
  </rows>
</template>
return tde:template-insert("Template.xml", $ClinicalView)
```

Insert the second version of the Pulications view template as follows:

```
xquery version "1.0-ml";
import module namespace tde = "http://marklogic.com/xdmp/tde"
        at "/MarkLogic/tde.xqy";

let $ClinicalView :=
<template xmlns="http://marklogic.com/xdmp/tde">
  <description>populates patients' data</description>
  <context>/Journal</context>
  <rows>
    <row>
      <schema-name>Medical</schema-name>
      <view-name>Publications</view-name>
      <view-layout>sparse</view-layout>
      <columns>
        <column>
          <name>Volume</name>
          <scalar-type>string</scalar-type>
          <val>Issue</val>
        </column>
        <column>
          <name>Title</name>
          <scalar-type>string</scalar-type>
          <val>Title</val>
        </column>
        <column>
          <name>Date</name>
          <scalar-type>string</scalar-type>
          <val>Date</val>
          <nullable>true</nullable>
        </column>
        <column>
          <name>Author</name>
          <scalar-type>string</scalar-type>
          <val>Author</val>
          <nullable>true</nullable>
        </column>
      </columns>
    </row>
  </rows>
</template>
return tde:template-insert("Template2.xml", $ClinicalView)
```

To see the combined results, enter:

```
select * FROM Medical.Publications
```

The results should look like:

ISSN	Title	Volume	Date	Author
null	Bone Density Studies	103	8/1/2009	John Simson
0043-5341	The Influence of Calcium on Cholesterol in Human Serum	118	12/7/1968	null

3.5 Creating Views from Nested Templates

You can nest template views. The example in this section, though not based on a credible use case, does show how to nest three template views for “medical” documents so that each child view is within the context of its parent view. The example also shows how variables can be defined in a parent template and then used in child templates. There is no limit to the nesting of template views.

The context for each nested view is as follows:

View	Context
Publication	/Citation
JournalIssue	/Citation/Article/Journal/JournalIssue
PubDate	/Citation/Article/Journal/JournalIssue/PubDate

```
<template xmlns="http://marklogic.com/xdmp/tde">
  <description>Views of the medical data set</description>
  <context>/Citation</context>
  <!-- Variables extracted at the current context level -->
  <vars>
    <var>
      <name>ID</name>
      <val>./ID</val>
    </var>
    <var>
      <name>Status</name>
      <val>@Status</val>
    </var>
  </vars>
  <rows>
    <row>
      <schema-name>medical</schema-name>
      <view-name>Publication</view-name>
      <view-layout>sparse</view-layout>
          <columns>
             <column>
                <name>ArticleTitle</name>
                 <scalar-type>string</scalar-type>
                 <val>Article/ArticleTitle</val>
             </column>
             <column>
                <name>ISSN</name>
                 <scalar-type>string</scalar-type>
                 <val>Article/Journal/ISSN</val>
             </column>
          </columns>
    </row>
```

```
</rows>
<templates>
  <template>
    <!-- Nested child template -->
    <!-- context path relative to the parent context: /Citation -->
    <context>Article/Journal/JournalIssue</context>
      <rows>
         <row>
             <schema-name>medical</schema-name>
             <view-name>JournalIssue</view-name>
             <view-layout>sparse</view-layout>
             <columns>
                <column>
                    <name>MedID</name>
                    <scalar-type>long</scalar-type>
                    <val>$ID</val>
                    <!-- referencing context var ID -->
                </column>
                <column>
                    <name>Volume</name>
                    <scalar-type>long</scalar-type>
                    <val>Volume</val>
                </column>
                <column>
                    <name>Issue</name>
                    <scalar-type>long</scalar-type>
                    <val>Issue</val>
                    <nullable>true</nullable>
                </column>
             </columns>
         </row>
      </rows>
      <templates>
        <template>
        <!-- Nested child template -->
        <!-- context path relative to the parent context:
              /Article/Journal/JournalIssue -->
        <context>PubDate</context>
        <rows>
           <row>
             <schema-name>medical</schema-name>
             <view-name>PubDate</view-name>
             <view-layout>sparse</view-layout>
             <columns>
                 <column>
                    <name>Status</name>
                    <scalar-type>string</scalar-type>
                    <val>$Status</val>
                    <!-- referencing context var Status -->
                 </column>
                 <column>
                    <name>Year</name>
                    <scalar-type>long</scalar-type>
                    <val>Year</val>
```

```
                    </column>
                    <column>
                       <name>Month</name>
                       <scalar-type>string</scalar-type>
                       <val>Month</val>
                    </column>
                    <column>
                       <name>Day</name>
                       <scalar-type>long</scalar-type>
                       <val>Day</val>
                    </column>
                 </columns>
              </row>
           </rows>
        </template>
     </templates>
  </template>
</templates>
</template>
```

4.0 Creating Range Views

This chapter describes how to configure MarkLogic Server and create range views to model your MarkLogic data for access by SQL. Range views can also be created using the Views API described in *MarkLogic XQuery and XSLT Function Reference*. You must have the `admin` role on MarkLogic Server to complete the procedures described in this chapter.

> **Note:** In most situations, you will want to create a template view, as described in"Creating Template Views" on page 34. Though a range view may be preferable to a template view in some situations, such as for a database already configured with range indexes, they are supported mostly for backwards compatibility with previous versions of MarkLogic.

The main topics are:

4.1 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.

4.2 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 57.

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.

4.3 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

4.3.1 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"
```

4.3.2 Creating and Setting the Schema

As described in "Schemas and Views" on page 5, a schema is a naming context for a set of views. Each view must belong to a schema. A schema created in this manner can support both Range Views and Template Views.

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.

4.3.3 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 *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)

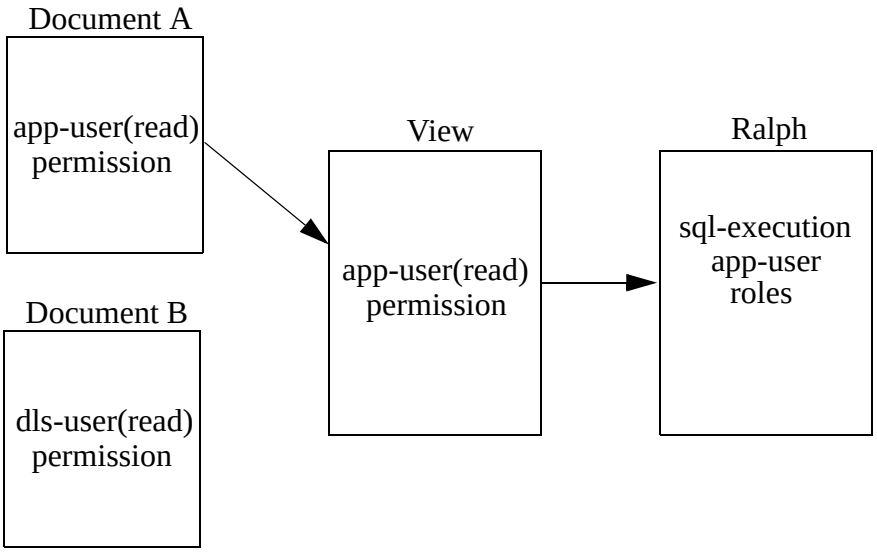
> **Note:** Unlike other documents, user default permissions are not assigned to the view or schema.

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.

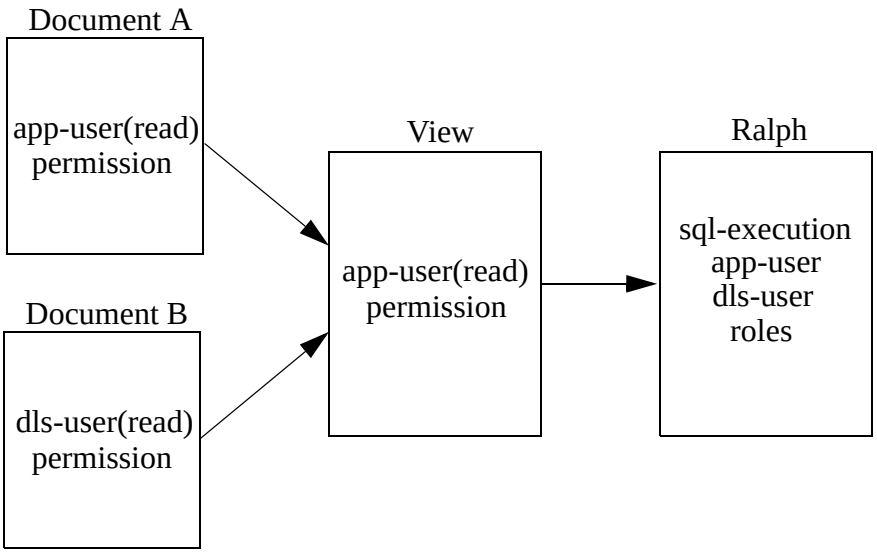
You can use the view API to set additional permissions on a schema or view to further restrict which users can access the schema or view. You set permissions on a schema by calling `view:schema-set-permissions` on an existing schema or by calling `view:schema-create` when creating a new schema. You set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



4.3.4 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.

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
```

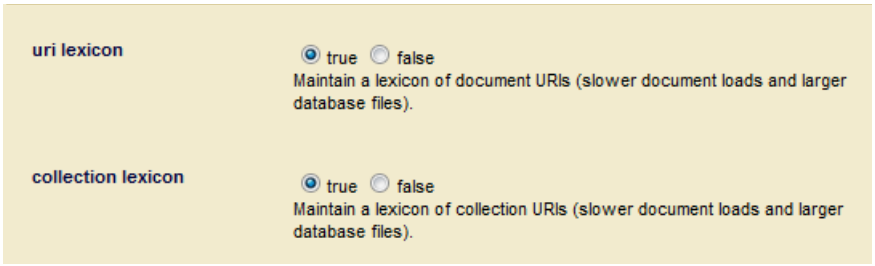
For example, to specify the "`subject`" column as nullable:

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

4.3.5 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}
```

4.3.6 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 51.

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"
}]
```

4.3.7 Defining View Scope

The scope of the view is used to constrain the view to the 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.

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

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

4.4 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.

4.4.1 The Email Data

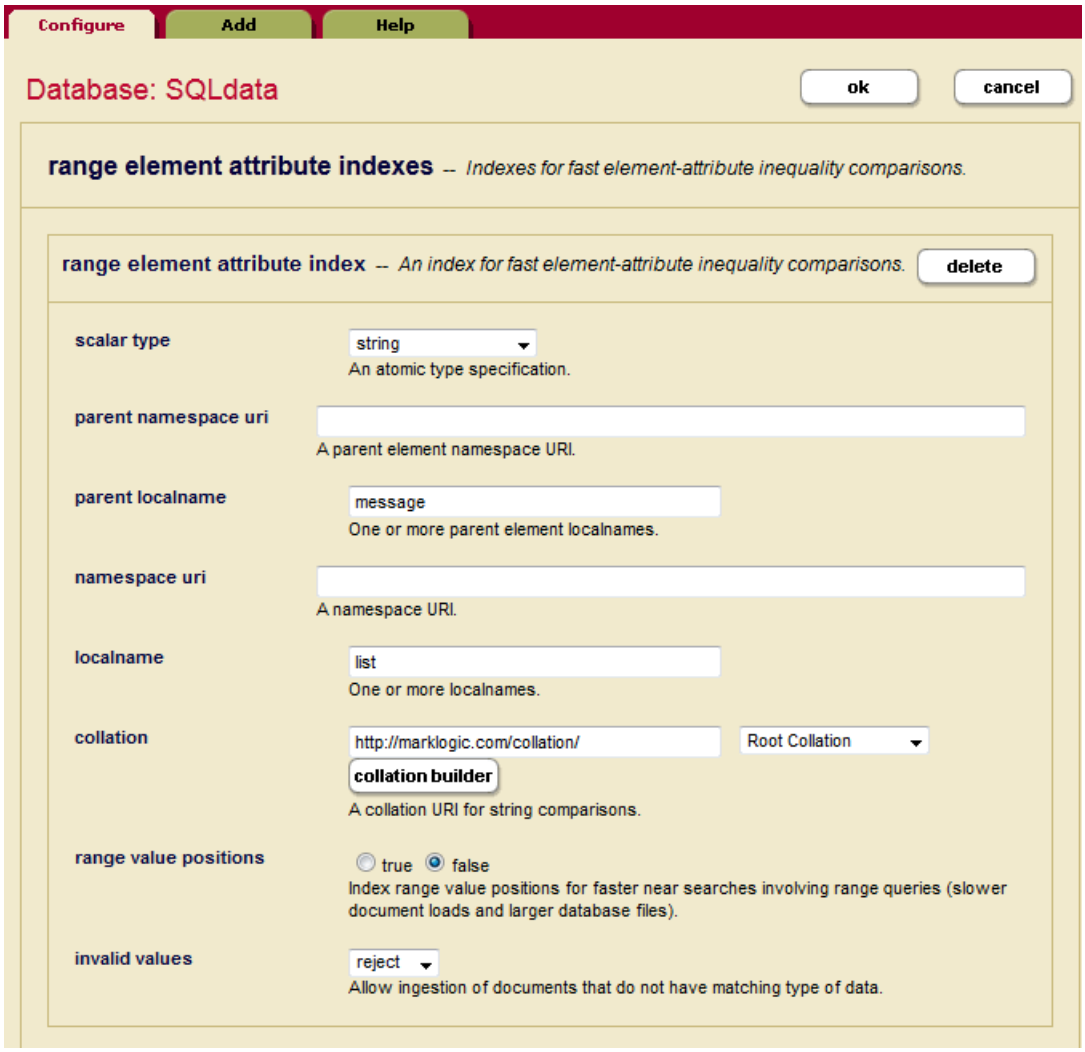
The procedures in this section assume you are loading documents like the following in your content database. The elements highlighted in yellow are those to be modeled as columns in the view.

```
<?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>
```

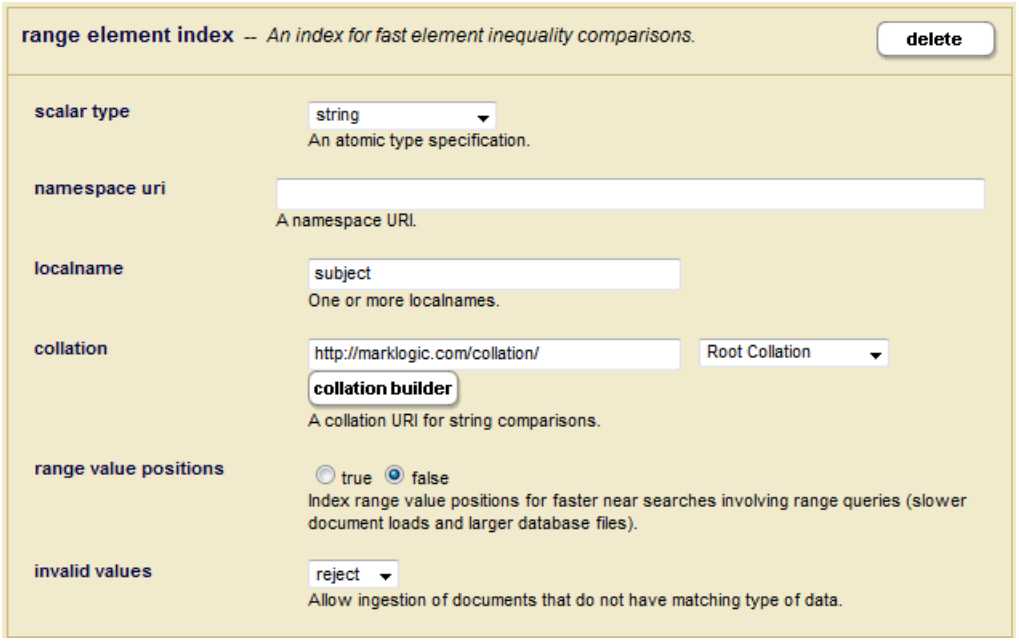
4.4.2 The Range Indexes

This section describes how to create the range indexes for the view described in “The View” on page 62.

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

4.4.3 The View

Create a view, named ‘mail.’ 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 59.

```
curl -X POST  --anyauth --user admin:admin \
--header "Content-Type:application/json" \
-d '{
     "view-schema-name": "mail",
     "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
```

4.5 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 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..."}
}
```

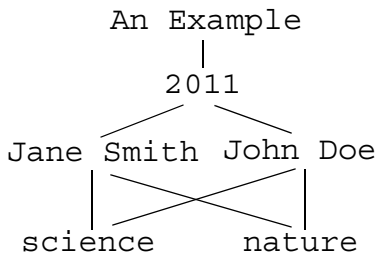
Because this document contains two `author` and `keyword` elements at the same level, it violates the first data modeling rule listed above. As a result, a `select *` on this view will produce multiple rows for the single document:

```
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
```

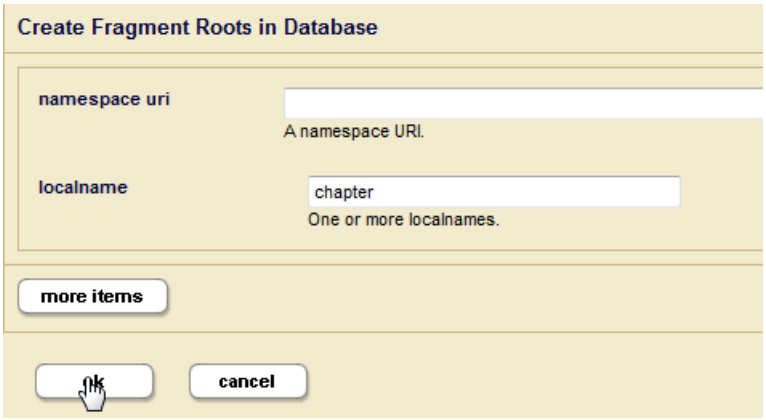
In other circumstances, you might want to set a root fragment on the database. For example, your document data is structured as follows:

```
<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	Columns

`meta`	`title`
	`pubyear`
	`author`
	`keyword`

`chapter`	`title` `section`

4.6 Limitations to SQL Support

The SQL supported by MarkLogic Server is SQL92 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.

4.7 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
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 to the log.
SQL Listing	Equivalent to executing "pragma vdbe_listing=1" which dumps the compiled 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.

5.0 Installing and Configuring the MarkLogic Server ODBC Driver

Tableau, Qlik, 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/odbc

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

- Configuring the ODBC Driver on Windows
- Configuring the ODBC Driver on Linux

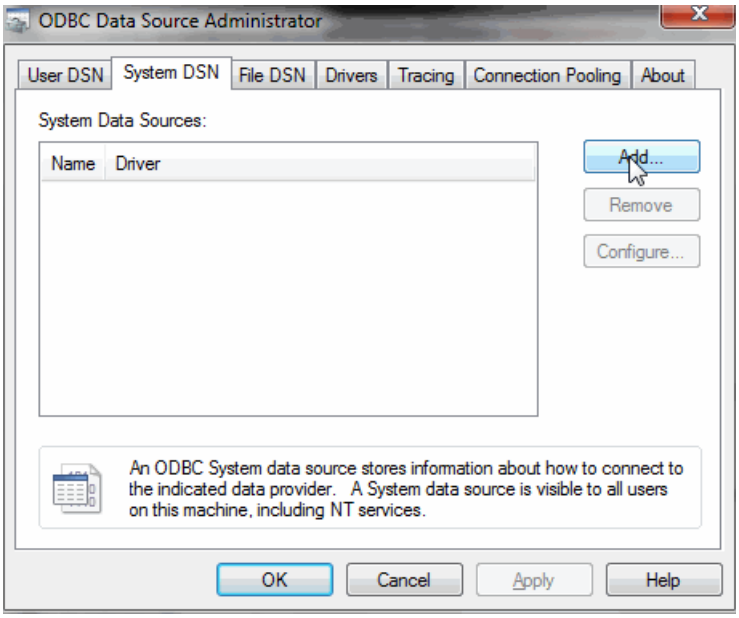
5.1 Configuring the ODBC Driver on Windows

Tableau and Qlik 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.

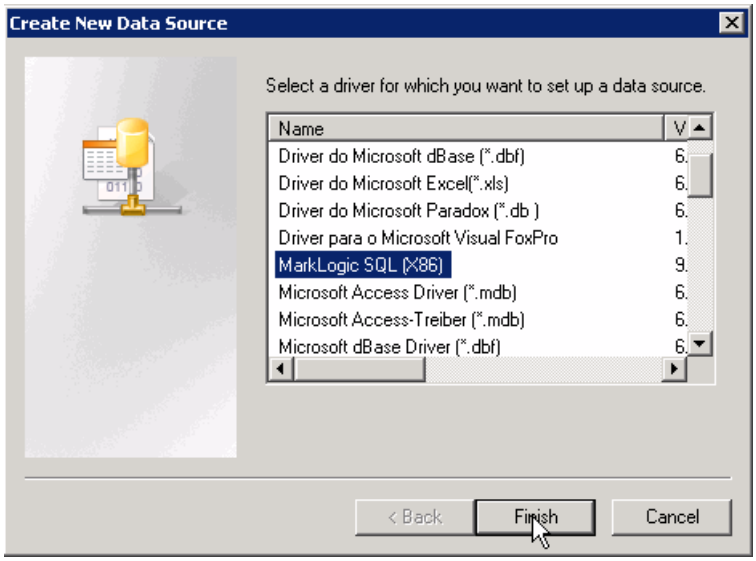
1. To launch the ODBC Data Source Administrator, open your Control Panel and navigate to:

   System and Security > Administrative Tools > Data Sources (ODBC)

2. Click the System DSN tab and click Add:

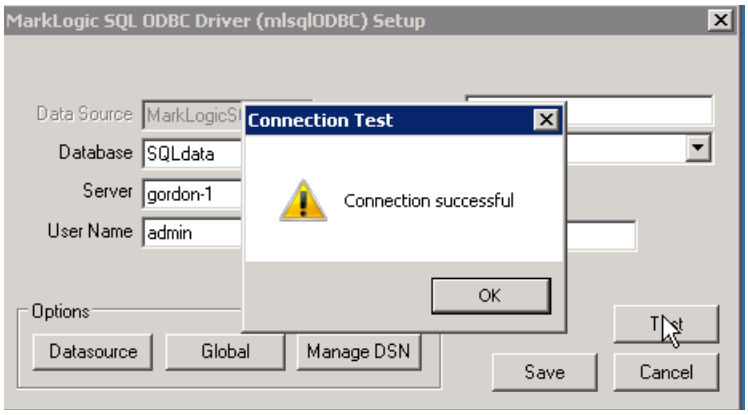


3. Select either the `MarkLogic SQL` (64-bit) or `MarkLogic SQL (x86)` (32-bit) driver and click Finish:

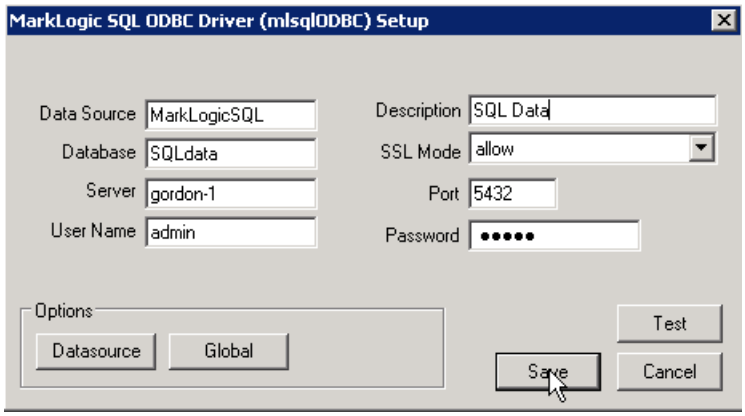


4. Set up an ODBC app server, as described in “Create an ODBC App Server” on page 17.

5. 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.



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



5.2 Configuring the ODBC Driver on Linux

Dependencies:openssl and unixODBC.

The following procedure describes how to install the MarkLogic ODBC driver on Linux. The unixODBC tool is only needed for testing your connection to MarkLogic via the mlsqlodbc driver. If you don't think it is necessary to test your ODBC connection, skip to Step 3.

1. Obtain a copy of unixODBC (2.3.4). 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-2.3.4.tar.gz
cd /tmp/unixODBC-2.3.4
./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-1.4-20170317.x86_64.rpm` in this example):

```
rpm -i mlsqlodbc-1.4-20170317.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 want 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.

6.0 Connecting Tableau to MarkLogic Server

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

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

Note: This chapter describes how to connect and configure Tableau version 10. The procedures described here may be different if you are using an older version of Tableau.

6.1 Install Tableau

Install Tableau, as described in the Tableau documentation.

Tableau has `select *` as the default query for the initial connection. To improve performance between Tableau and MarkLogic, follow the instructions below.

In the path shown below, save the following xml structure as `odbc-marklogic.tdc` (the name does not matter, but extension has to be `.tdc`):

```
C:\Users\**\Documents\My Tableau Repository\Datasources
```

If installing the 32-bit ODBC driver, the contents of the `odbc-marklogic.tdc` file should look like the following:

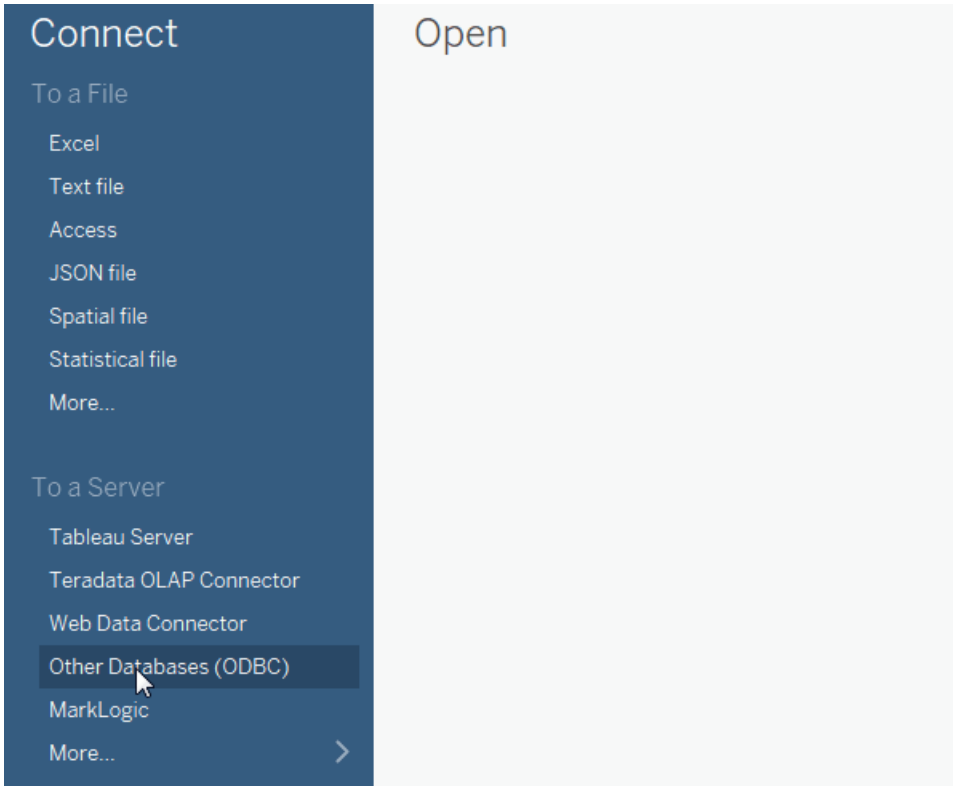
```
<?xml version='1.0' encoding='utf-8' ?>

<connection-customization class='genericodbc' enabled='true' version='10.2'>
   <vendor name='MarkLogic' />
   <driver name='MarkLogic SQL (X86)' />
 <customizations>
   <customization name='CAP_QUERY_TOP_0_METADATA' value='yes' />
   <customization name='CAP_ODBC_METADATA_SUPPRESS_SELECT_STAR' value='yes' />
   <customization name='CAP_QUERY_WHERE_FALSE_METADATA' value='no' />
   <customization name='CAP_CREATE_TEMP_TABLES' value='no' />
 </customizations>
</connection-customization>
```

6.2 Connect Tableau to MarkLogic Server

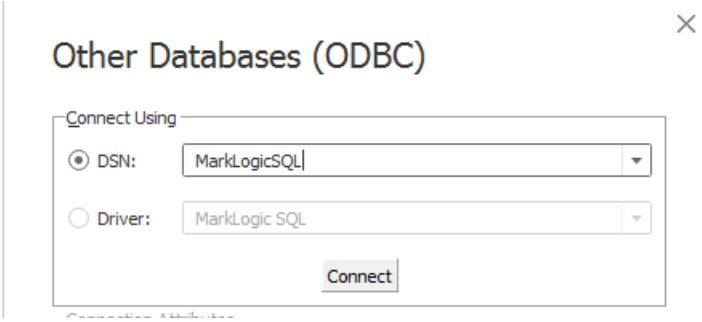
This section describes how to connect Tableau to MarkLogic Server.

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 70.

1. Open Tableau and click `Other Databases (ODBC)`:

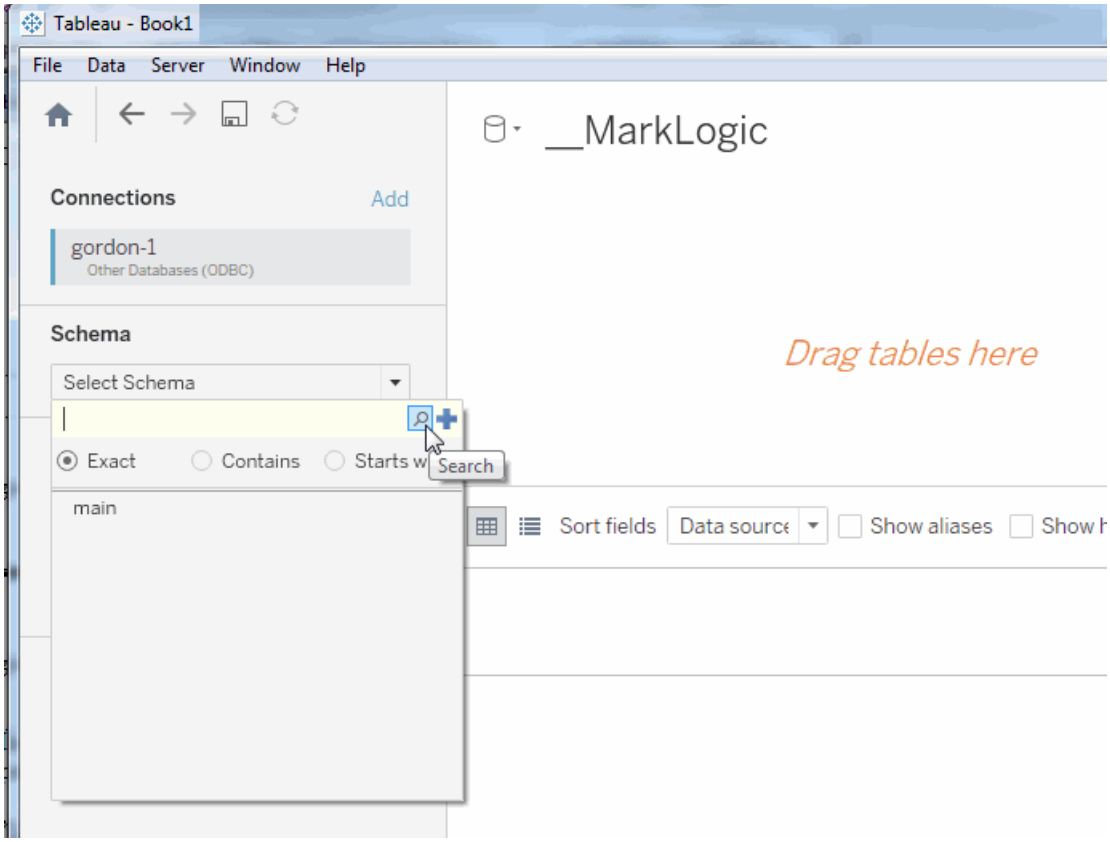


2. In the Other Databases (ODBC) window, select MarkLogicSQL from the DSN pulldown window. Click `Connect`:



3. In the Connection Attributes portion of the Other Databases (ODBC) window, enter the name of your MarkLogic server, the port number of your ODBC App Server (5432, in this example), and login credentials (the `Database` field is ignored). Click `Sign In`:

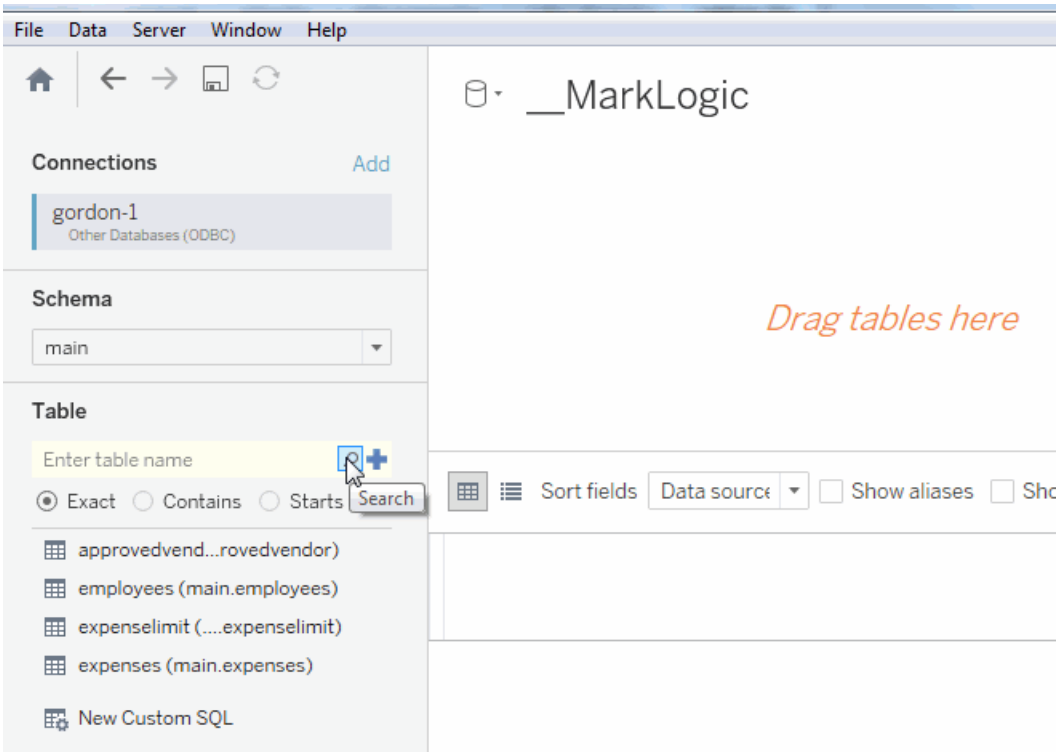
4. In the Data Source window, click Select Schema, then on the magnifying glass icon and select your schema (in this example, 'main') from the menu:



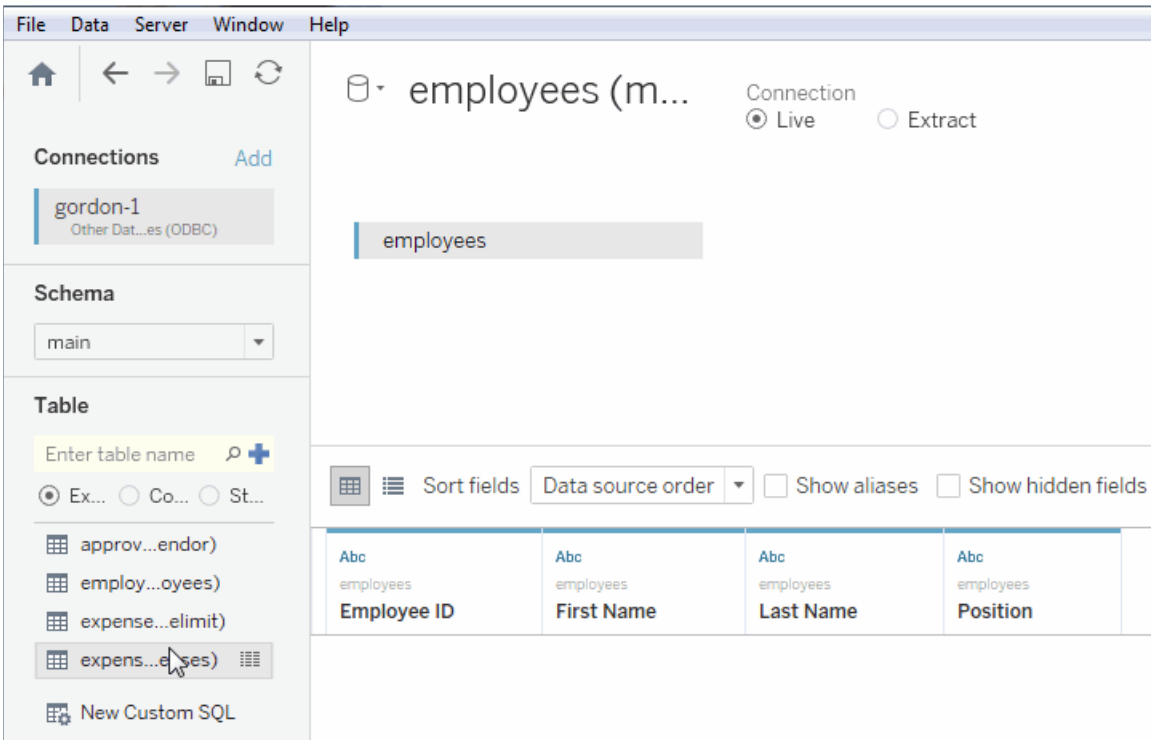
6.3 Add Tables to Tableau Workbook

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

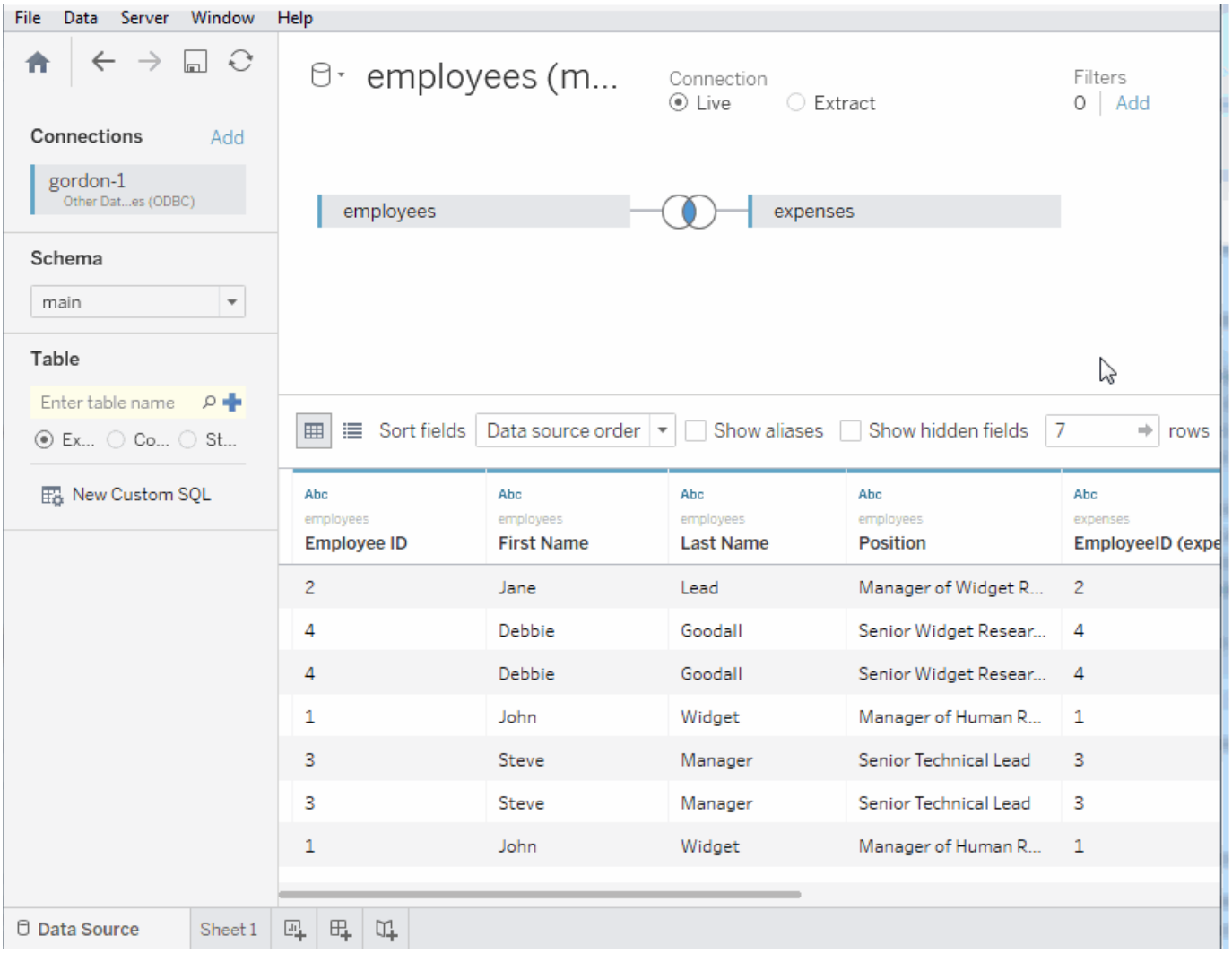
1. In the Data Source window, click Select Table, then on the magnifying glass icon to populate the menu below with the views from your schema:



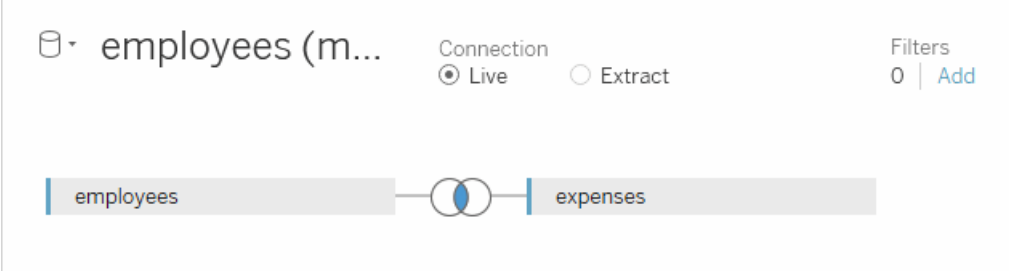
2. Drag the views in the Tables menu to the canvas. In this example, the `employees` view is dragged to the workbook:



3. Drag another view to the workbook. In this example, the `expenses` view is dragged to the canvas and joined with `employees` with an inner join. The results are shown below.

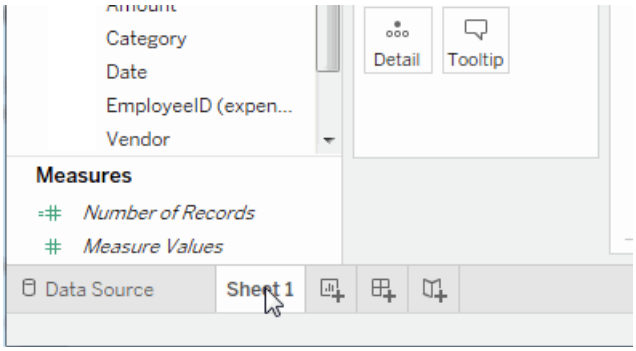


4. At the top of your canvas, select the desired Connection type:

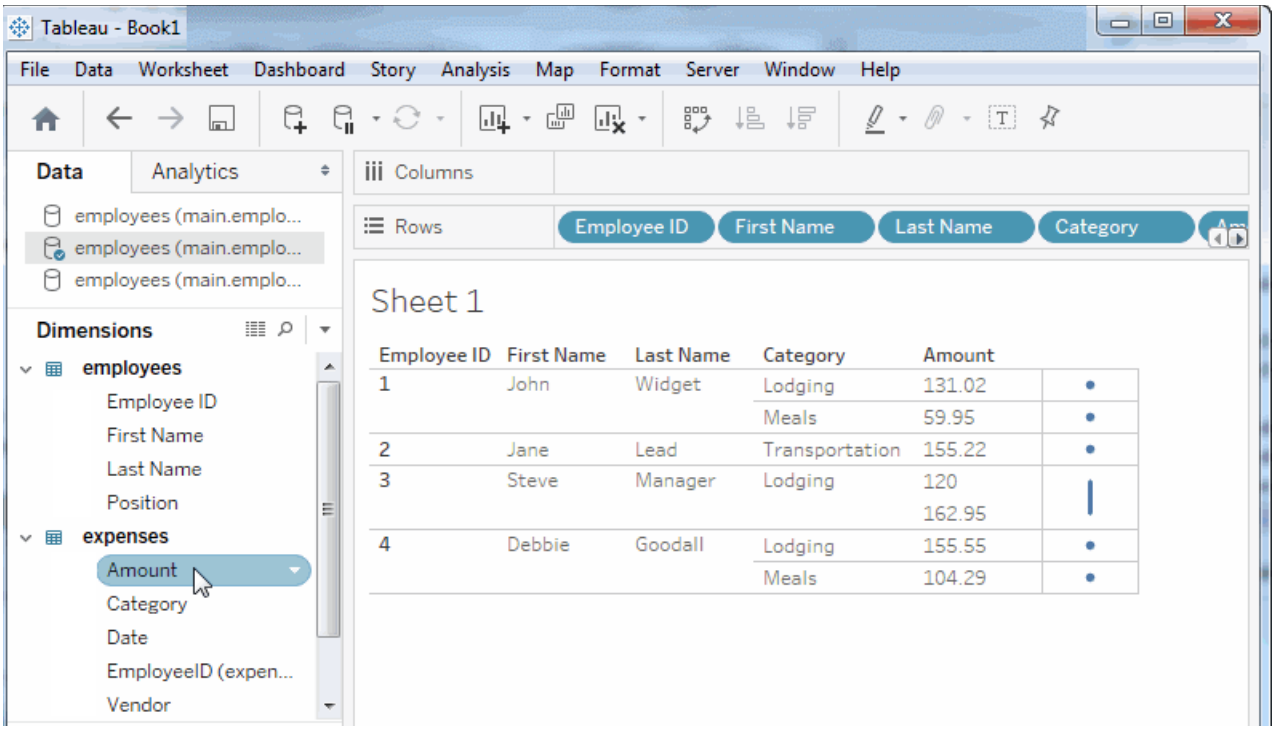


Connection Type	Description
Live	Creates a direct connection to your data. As reports are generated, data is pulled live from the data source. The speed of your data source will determine performance.
Extract	Imports the entire data source into Tableau's fast data engine as an extract. The extract is saved with the workbook.

5. At the bottom of the Tableau window, navigate to `Sheet1`.



6. In the `Sheet1` window, drag `EmployeeID`, `FirstName` and `LastName` from the `employees` view in the `Dimensions` pane to the `Rows` field. Drag `Category` and `Amount` from the `expenses` view to the `Rows` field. You should see your data stored in MarkLogic Server displayed in table on the right.

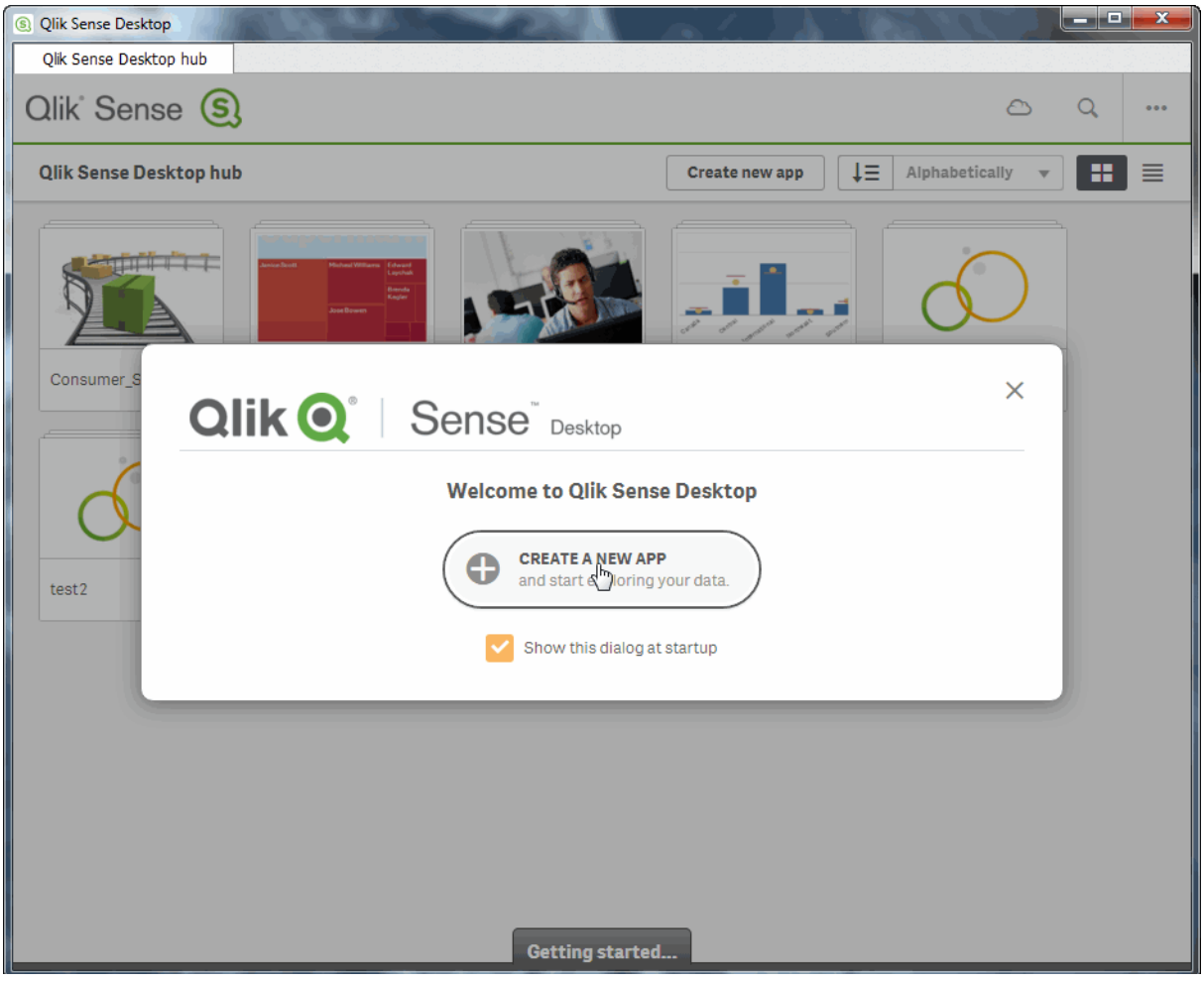


7.0 Connecting Qlik to MarkLogic Server

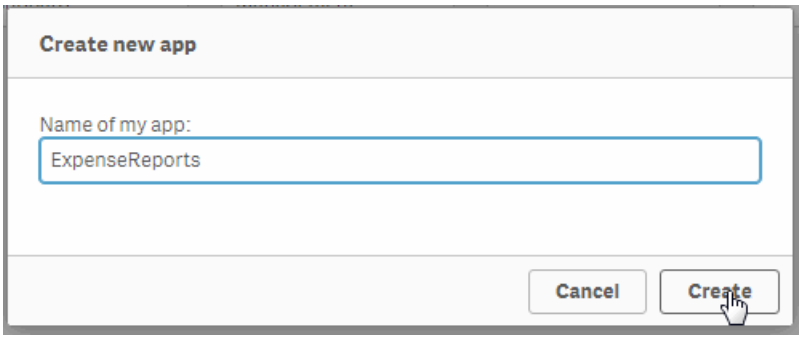
This chapter describes how to connect and configure Qlik for use with MarkLogic Server. The procedure described in this chapter assumes you have first installed the MarkLogic ODBC driver and configured it as an ODBC data source on the client server, as described in “Installing and Configuring the MarkLogic Server ODBC Driver” on page 70.

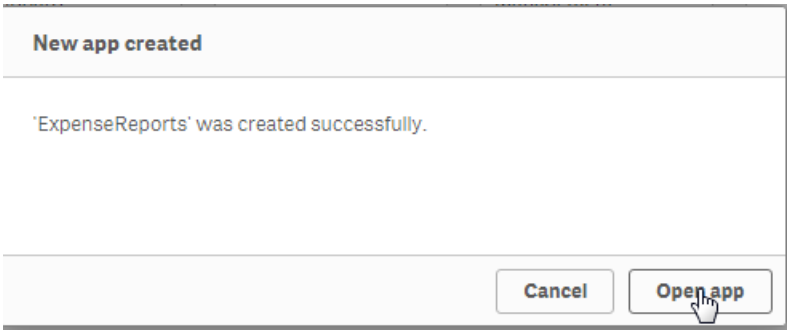
1. Open Qlik Desktop

2. Click on CREATE A NEW APP.

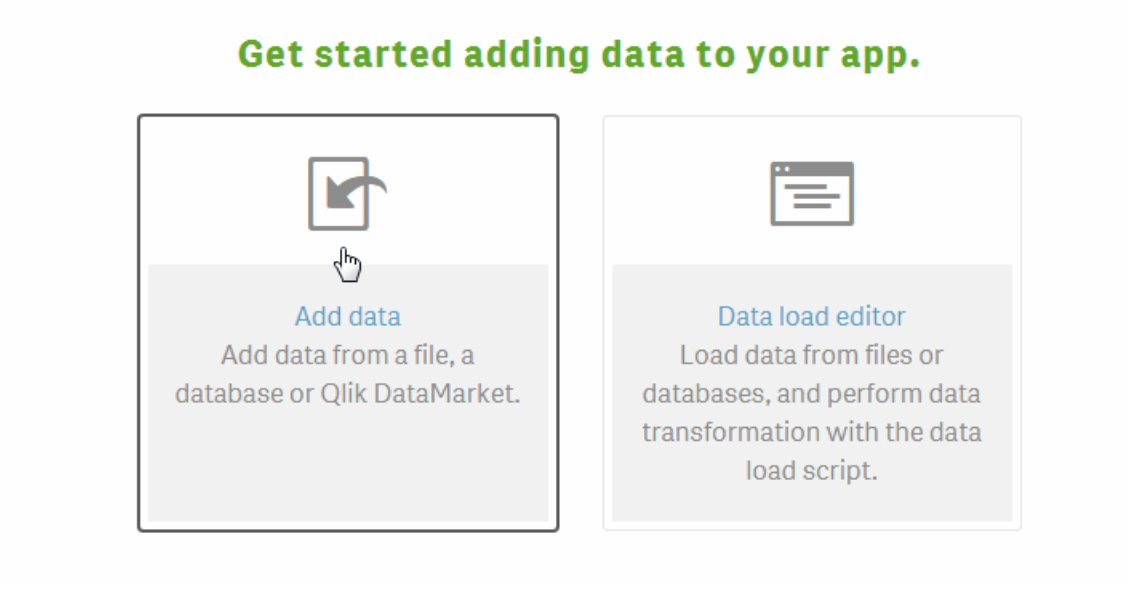


3. Provide a name for the new app and select `Open app`.

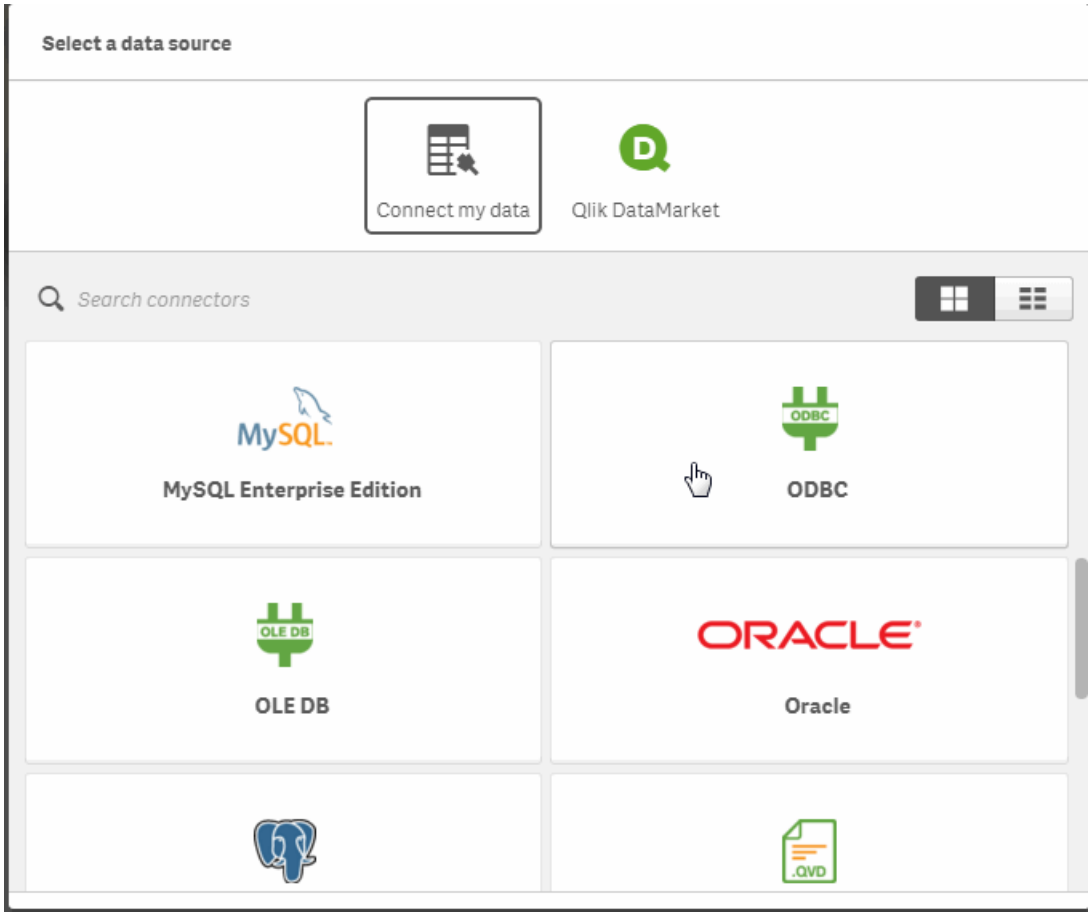




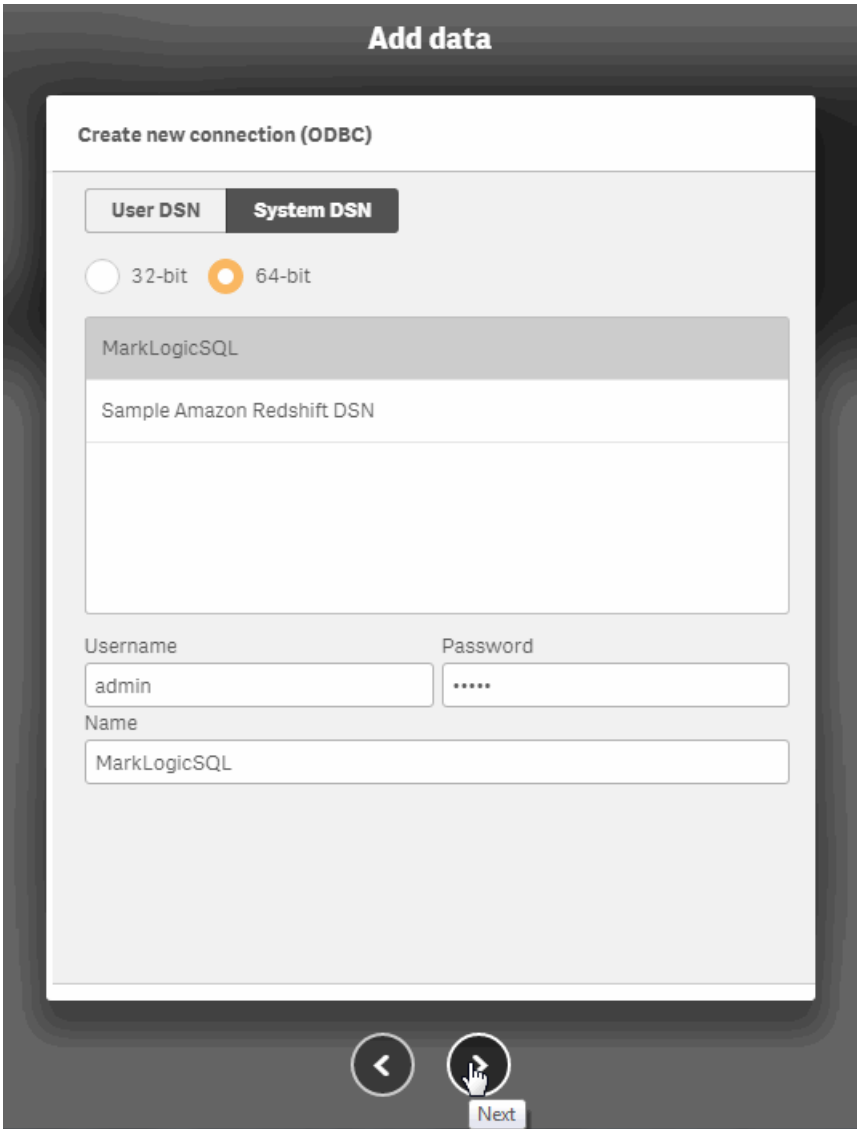
4. Select `Add data`.



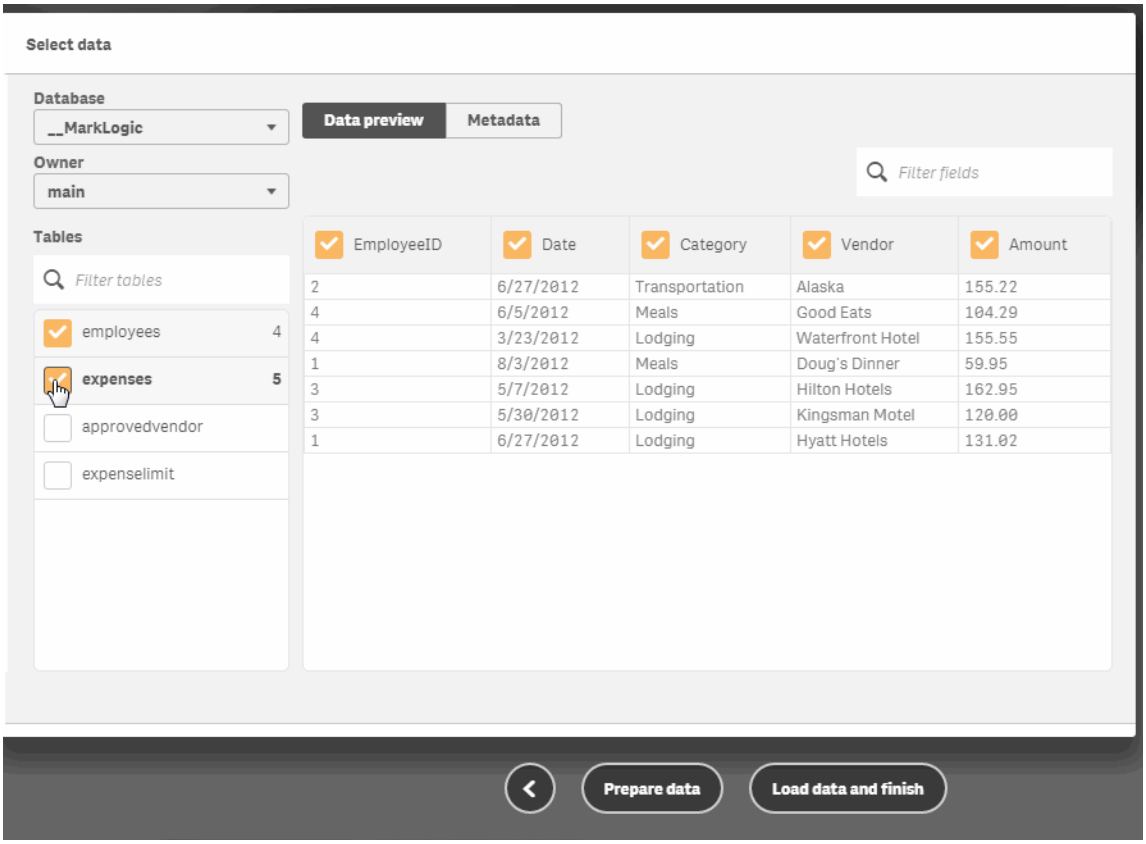
5. In the `Select a data source` window, select `ODBC`.



6. In the `Add data` window, select `System DSN`, the name of the MarkLogic ODBC driver (`MarkLogicSQL`), enter your MarkLogic Server login credentials. (You must have the view-admin role on MarkLogic Server). Click forward button at the bottom of the window.



7. In the `Tables` pane, select the tables you want to use. This will populate the `Data preview` with the view data.



8.0 SQL Syntax

In general, MarkLogic supports the syntax from the SQL92 standard. This chapter describes some of the SQL syntax that are unique to MarkLogic Server.

8.1 Supported SQL Statements, Functions and Types

This section describes the SQL statements and functions supported in MarkLogic. The topics are:

8.1.1 Supported Statements

MarkLogic SQL does not support updates, so only the SQL statements in the table below are supported.

SQL Statement	Notes
EXPLAIN	Produces an execution plan, as described in “Execution Plan” on page 103.
SELECT	The following SELECT options are not supported: FULL OUTER JOIN, BLOB types, and correlated subqueries containing a GROUP BY.
CREATE VIEW	Creates a view. Views created by this statement only exist for the duration of the SQL connection.
DROP VIEW	Drops a view created by CREATE VIEW.

8.1.2 Supported Functions

MarkLogic supports the SQL functions in the SQL92 standard. In addition, MarkLogic supports SQL functions that are not part of the SQL92 standard, as shown in the table below. The SQL functions are listed along with the MarkLogic builtin functions that support them. The syntax for the SQL function is the same as that of the respective builtin function.

You can also call any MarkLogic builtin function in a SQL query, as described in "Calling Built-in Functions from SQL" on page 96.

SQL Function	MarkLogic Builtin
acos	math:acos
ascii	fn:string-to-codepoints
asin	math:asin
atan	math:atan
atan2	math:atan2
bit-length	sql:bit-length
ceiling	fn:ceiling
char	fn:codepoints-to-string
character-length	fn:string-length
char-length	fn:string-length
concat	fn:concat
cos	math:cos
cot	math:cot
current-date	fn:current-date
current-time	fn:current-time
current-timestamp	fn:current-dateTime
current-user	fn:get-current-user
curdate	fn:current-date
curtime	fn:current-time
datepart	sql:datepart
datediff	sql:datediff
dateadd	sql:dateadd

SQL Function	MarkLogic Builtin
day	sql:day
dayname	sql:dayname
dayofmonth	sql:day
dayofweek	sql:weekday
dayofyear	sql:yearday
degrees	math:degrees
exp	math:exp
floor	fn:floor
hour	sql:hours
initcap	xdmp:initcap
insert	sql:insert
left	sql:left
localtime	fn:current-time
localtimestamp	fn:current-dateTime
locate	xdmp:position
log	math:log
log10	math:log10
minute	sql:minutes
mod	math:fmod
month	sql:month
monthname	sql:monthname
now	fn:current-time
octet-length	sql:octet-length
pi	math:pi
position	xdmp:position
power	math:pow
quarter	sql:quarter
radians	math:radians
rand	sql:rand

SQL Function	MarkLogic Builtin
random	sql:rand
repeat	sql:repeat
right	sql:right
sign	sql:sign
sin	math:sin
second	sql:seconds
session-user	fn:get-current-user
space	sql:space
sqrt	math:sqrt
strpos	xdmp:position
substring	fn:substring
tan	math:tan
timestampadd	sql:timestampadd
timestampdiff	sql:timestampdiff
truncate	math:trunc
trunc	math:trunc
user	xdmp:get-current-user
week	sql:week
year	sql:year

8.1.3 Supported Types

The table below lists all of the supported SQL types in MarkLogic, along with the mapping from the SQL types to XML Schema (or MarkLogic) types. MarkLogic also supports a number of SQL type that go beyond those supported by the SQL92 standard, as well as some vendor specific types.

Note: Limits on datatypes are not enforced. For example, if you enter `DECIMAL(p,s)`, the precision and scale are ignored.

SQL Type	XML Schema Type	Range Index (Scalar) Type	Notes
CHAR(ACTER)	xs:string		Fixed length unenforced. CHARACTER SET must be "UTF-8" if specified.
CHAR(ACTER) VARYING / VARCHAR / TEXT	xs:string	string, anyURI	Maximum length unenforced. CHARACTER SET must be "UTF-8" if specified.
NATIONAL CHAR(ACTER) / NCHAR	xs:string		Fixed length not enforced.
NATIONAL CHAR(ACTER) VARYING / NCHAR VARYING / NVARCHAR	xs:string		Maximum length not enforced.
NUMERIC / DEC(IMAL)	xs:decimal	decimal	Precision and scale not enforced.
INT(EGER) / MEDIUMINT / INT4	xs:int	int	

SQL Type	XML Schema Type	Range Index (Scalar) Type	Notes
UNSIGNED INT(EGER) / UNSIGNED MEDIUMINT / UNSIGNED INT4	xs:unsignedInt	unsignedInt	
TINYINT / INT1	xs:byte		
UNSIGNED TINYINT / UNSIGNED INT1	xs:unsignedByte		
SMALLINT / INT2	xs:short		
UNSIGNED SMALLINT / UNSIGNED INT2	xs:unsignedShort		
BIGINT / INT8	xs:long	long	
UNSIGNED BIGINT / UNSIGNED INT8	xs:unsignedLong	unsignedLong	
FLOAT(X) with X<24 / REAL	xs:float	float	
FLOAT(X) with 24<=X<=52 / DOUBLE (PRECISION)	xs:double	double	
BOOLEAN	xs:boolean		Not in SQL92
DATE	xs:date	date	DATE does not support a timezone
TIME	xs:time	time	
TIMESTAMP	xs:dateTime	dateTime, gYearMonth, gYear, gMonth, gDay	Oracle converts the g* datatypes to TIMESTAMP WITH TIMEZONE

SQL Type	XML Schema Type	Range Index (Scalar) Type	Notes
INTERVAL YEAR / INTERVAL MONTH / INTERVAL YEAR TO MONTH	xs:yearMonthDuration	yearMonthDuration	For INTERVAL types with only year and/or month specified.
INTERVAL DAY / INTERVAL HOUR / INTERVAL DAY TO SECOND etc.	xs:dayTimeDuration	dayTimeDuration	For INTERVAL types with only day / hour / minute / second specified
INTERVAL	xs:duration		For all other INTERVAL types

8.2 System Tables

Data dictionaries consists of a series of tables that are created in the SYS schema. These system tables are listed in the table below.

System Table	Description
sys_schemas	Lists all of the available schemas.
sys_tables	Lists all of the available tables.
sys_columns	Lists all of the available columns.
sys_functions	Lists all of the available functions.
sys_collations	Lists all of the available collations.

To see the full contents of a system table, do a `select *`. For example:

```
select * from sys_tables
```

8.3 System Columns __content and __docid

Each view has two system columns:

Column Name	Description
`__docid`	Identifies the fragment ID of each document that matches the view(s).
`__content`	Returns the content of document that matches the view(s).

Note: The `__docid` and `__content` system columns are preceeded by two underscores.

For example: The following returns the fragment ID for each document that matches the `employees` view:

```
select __docid from employees
```

The following returns the contents of each document that matches the `employees` view:

```
select __content from employees
```

8.4 Calling Built-in Functions from SQL

You can call MarkLogic built-in functions from inside a SELECT statement, as long as the parameter types match the column types. You cannot call aggregate functions from SQL.

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(EmployeeID) from employees
```

Do some geospatial:

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

Return the first five values of the FirstName column, starting with the third character:

```
select fn_substring(FirstName,3) from employees limit 5
```

8.5 MATCH Operator

The MATCH operates differently on range views and template views. You can MATCH column names when using range views, but not template views. You can MATCH on tables created by both range and template views.

When the MATCH operator is used with range views, column names are bound to their corresponding 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 index to the values of another. These queries are executed in unfiltered mode.

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

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

8.5.1 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. The field or column text must have the correct case. For example, `'Position:Manager'` is not the same as `'position:Manager'`. Because you cannot specify fields in a template view, you cannot MATCH on field names.

8.5.2 Examples

The following queries will work on both range views and template views:

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

SELECT * FROM employees WHERE employees MATCH 'J*'

SELECT employeeid, firstname, lastname, position FROM employees
    WHERE employees MATCH 'Steve OR John OR Goodall'

SELECT employeeid, firstname, lastname, position FROM employees
    WHERE employees MATCH 'Steve AND Manager'

SELECT * from employees WHERE firstname MATCH 'John OR Jane'
    AND lastname MATCH 'Lead'
```

The following queries will work on range views only:

```
SELECT * FROM employees WHERE employees MATCH 'position:Manager'

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

SELECT employeeid, firstname, lastname, position FROM employees
    WHERE firstname MATCH 'Steve OR John OR Goodall'

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

8.6 SET/SHOW Statements

The MarkLogic ODBC driver supports Postgres SET and SHOW run-time configuration parameters, as well as some parameters that are specific to MarkLogic Server. These parameters only work when accessing MarkLogic through an ODBC driver, as is the case with `mlsql`. They do not work when accessing MarkLogic through `xdmp:sql` or the Query Console.

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 102.

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

8.6.1 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'
```

8.6.2 statement_timeout

Sets the timeout for statement execution (milliseconds).

For example:

```
SET statement_timeout 5000
```

8.6.3 lc_messages

Sets the locale for error messages.

For example:

```
SET lc_messages 'en_US'
```

8.6.4 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'
```

8.6.5 lc_numeric

Sets the locale for formatting numeric values.

For example:

```
set lc_numeric 'de_DE'
```

8.6.6 lc_time

Sets the locale for formatting date/time values.

For example:

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

8.6.7 DateType

Sets the output format for dates.

For example:

```
SET DateType 'ISO'
```

8.6.8 extra_float_digits

Sets the number of digits displayed for floating point types.

For example:

```
SET extra_float_digits 2
```

8.6.9 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'
```

8.6.10 coordinate_system

Set the default coordinate system for geospatial operations.

For example:

```
SET coordinate_system 'wgs84/double'
```

For more details, see The Governing Coordinate System and Controlling Coordinate System and Precision in the *Search Developer's Guide*.

8.6.11 SCHEMA or search_path

Sets the default schema referenced by names in SQL statements.

For example:

```
SET search_path 'main'
```

8.6.12 mls_default_xquery

Set the default XQuery version.

For example:

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

8.6.13 mls_redundant_check

Enable or disable the 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;
```

8.7 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.

8.8 Best Practices and Performance Considerations

MarkLogic SQL does not have a default/implicit limit for the rows returned. Queries that return large result sets, such as tens of thousands of rows, may perform poorly. Should you experience performance problems it is a best practice to page the results using the `LIMIT` statement.

9.0 Execution Plan

This section describes how to interpret an execution plan output by the SQL `EXPLAIN` statement, the Optic `AccessPlan.prototype.explain` method, or the `xdmp:sql-plan` function.

9.1 Generating an Execution Plan

You can use the EXPLAIN statement or `xdmp:sql-plan` function to generate the query execution plan for a SQL query. For example, the following produces and execution plan for the SELECT query:

```
EXPLAIN SELECT employees.FirstName, employees.LastName,
SUM(expenses.Amount) AS ExpensesPerEmployee
FROM employees, expenses
WHERE employees.EmployeeID = expenses.EmployeeID
GROUP BY employees.FirstName, employees.LastName
```

Outputs the following execution plan:

```
<plan:plan xmlns:plan="http://marklogic.com/plan">
  <plan:select>
    <plan:project order="">
      <plan:column name="employees.FirstName"
            column-index="0" static-type="STRING">
      </plan:column>
      <plan:column name="employees.LastName"
            column-index="1" static-type="STRING">
      </plan:column>
      <plan:column name="ExpensesPerEmployee"
            column-index="2" static-type="DOUBLE">
      </plan:column>
      <plan:hash-group order="">
        <plan:order-spec descending="false"
            column="main.employees.FirstName" column-index="1">
        </plan:order-spec>
        <plan:order-spec descending="false"
            column="main.employees.LastName" column-index="2">
        </plan:order-spec>
        <plan:aggregate column="ExpensesPerEmployee"
            column-index="2" name="sum" distinct="false">
          <plan:column-ref name="main.expenses.Amount"
            column-index="5">
          </plan:column-ref>
        </plan:aggregate>
        <plan:aggregate column="employees.FirstName"
            column-index="0" name="sample" distinct="false">
          <plan:column-ref name="main.employees.FirstName" column-index="1">
          </plan:column-ref>
        </plan:aggregate>
        <plan:aggregate column="employees.LastName"
            column-index="1" name="sample" distinct="false">
          <plan:column-ref name="main.employees.LastName"
```

```
      column-index="2">
    </plan:column-ref>
  </plan:aggregate>
  <plan:parallel-hash-join order="3,2">
    <plan:hash left="4" right="0" operator="=">
    </plan:hash>
    <plan:sort-merge-join order="6,4">
      <plan:hash left="6" right="6" operator="=">
      </plan:hash>
      <plan:triple-index order="6,5" permutation="PSO">
        <plan:subject>
          <plan:column name="main.expenses.rowid"
                column-index="6" static-type="UNKNOWN">
          </plan:column>
        </plan:subject>
        <plan:predicate>
          <plan:value column="main.expenses.Amount"
                 columnID="14904495488947884968">
          </plan:value>
        </plan:predicate>
        <plan:object>
          <plan:column name="main.expenses.Amount"
                column-index="5" static-type="DECIMAL">
          </plan:column>
        </plan:object>
      </plan:triple-index>
      <plan:triple-index order="6,4" permutation="PSO">
        <plan:subject>
          <plan:column name="main.expenses.rowid"
                column-index="6" static-type="UNKNOWN">
          </plan:column>
        </plan:subject>
        <plan:predicate>
          <plan:value column="main.expenses.EmployeeID"
                 columnID="3887479265206160521">
          </plan:value>
        </plan:predicate>
        <plan:object>
          <plan:column name="main.expenses.EmployeeID"
                column-index="4" static-type="INT">
          </plan:column>
        </plan:object>
      </plan:triple-index>
    </plan:sort-merge-join>
    <plan:hash-join order="3,2">
      <plan:hash left="3" right="3" operator="=">
      </plan:hash>
      <plan:sort-merge-join order="3,1">
        <plan:hash left="3" right="3" operator="=">
        </plan:hash>
        <plan:triple-index order="3,0" permutation="PSO">
          <plan:subject>
            <plan:column name="main.employees.rowid"
                column-index="3" static-type="UNKNOWN">
```

```
          </plan:column>
        </plan:subject>
        <plan:predicate>
          <plan:value column="main.employees.EmployeeID"
               columnID="4691838910292433538">
          </plan:value>
        </plan:predicate>
        <plan:object>
          <plan:column name="main.employees.EmployeeID"
               column-index="0" static-type="INT">
          </plan:column>
        </plan:object>
      </plan:triple-index>
      <plan:triple-index order="3,1" permutation="PSO">
        <plan:subject>
          <plan:column name="main.employees.rowid"
              column-index="3" static-type="UNKNOWN">
          </plan:column>
        </plan:subject>
        <plan:predicate>
          <plan:value column="main.employees.FirstName"
               columnID="2346001466860406442">
          </plan:value>
        </plan:predicate>
        <plan:object>
          <plan:column name="main.employees.FirstName"
               column-index="1" static-type="STRING">
          </plan:column>
        </plan:object>
      </plan:triple-index>
    </plan:sort-merge-join>
    <plan:triple-index order="3,2" permutation="PSO">
      <plan:subject>
        <plan:column name="main.employees.rowid"
              column-index="3" static-type="UNKNOWN">
        </plan:column>
      </plan:subject>
      <plan:predicate>
        <plan:value column="main.employees.LastName"
               columnID="3470857143136371394">
        </plan:value>
      </plan:predicate>
      <plan:object>
        <plan:column name="main.employees.LastName"
              column-index="2" static-type="STRING">
        </plan:column>
      </plan:object>
    </plan:triple-index>
  </plan:hash-join>
  <plan:join-filter op="=">
    <plan:column name="main.employees.EmployeeID"
              column-index="0" static-type="UNKNOWN">
    </plan:column>
    <plan:column name="main.expenses.EmployeeID"
```

```
                    column-index="4" static-type="UNKNOWN">
            </plan:column>
          </plan:join-filter>
        </plan:parallel-hash-join>
      </plan:hash-group>
    </plan:project>
  </plan:select>
</plan:plan>
```

9.2 Parsing an Execution Plan

This section breaks down and describes each portion of the execution plan.

Notice that new column numbers are assigned to everything that gets used in the query. For example, column numbers are reassigned after a group-by, so that column 0 is something different inside the group-by compared to outside of it.

```
<plan:plan xmlns:plan="http://marklogic.com/plan">
  <plan:select>
    <plan:project order="">
```

Column names and numbers (three columns total):

```
      <plan:column name="employees.FirstName"
            column-index="0" static-type="STRING">
      </plan:column>
      <plan:column name="employees.LastName"
            column-index="1" static-type="STRING">
      </plan:column>
      <plan:column name="ExpensesPerEmployee"
            column-index="2" static-type="DOUBLE">
      </plan:column>
```

The output order of the `employees.FirstName` and `employees.LastName` columns:

```
      <plan:hash-group order="">
        <plan:order-spec descending="false"
            column="main.employees.FirstName" column-index="1">
        </plan:order-spec>
        <plan:order-spec descending="false"
            column="main.employees.LastName" column-index="2">
        </plan:order-spec>
```

The aggregation sequence for calculating `SUM(expenses.Amount) AS ExpensesPerEmployee FROM employees, expenses`. The results are identified as `column-index="5"`

```
        <plan:aggregate column="ExpensesPerEmployee"
            column-index="2" name="sum" distinct="false">
          <plan:column-ref name="main.expenses.Amount"
            column-index="5">
          </plan:column-ref>
        </plan:aggregate>
```

```
<plan:aggregate column="employees.FirstName"
    column-index="0" name="sample" distinct="false">
  <plan:column-ref name="main.employees.FirstName" column-index="1">
  </plan:column-ref>
</plan:aggregate>
<plan:aggregate column="employees.LastName"
    column-index="1" name="sample" distinct="false">
  <plan:column-ref name="main.employees.LastName" column-index="2">
  </plan:column-ref>
</plan:aggregate>
```

The columns are joined and the `triple-index` elements indicate which triples are accessed for the data. The `permutation` indicates how the results from a triple is ordered. For example, `PSO` indicates an order of `predicate`, `subject`, and `object`.

The `order` attribute details the known natural order of the result of the operators. For example, `order="6,4"` indicates that the result is ordered first by column 6 (`ascending`) and then by column 4 (`ascending`). Ascending is implied if `descending` is not shown.

```
<plan:parallel-hash-join order="3,2">
  <plan:hash left="4" right="0" operator="=">
  </plan:hash>
  <plan:sort-merge-join order="6,4">
    <plan:hash left="6" right="6" operator="=">
    </plan:hash>
    <plan:triple-index order="6,5" permutation="PSO">
      <plan:subject>
        <plan:column name="main.expenses.rowid"
              column-index="6" static-type="UNKNOWN">
        </plan:column>
      </plan:subject>
      <plan:predicate>
        <plan:value column="main.expenses.Amount"
               columnID="14904495488947884968">
        </plan:value>
      </plan:predicate>
      <plan:object>
        <plan:column name="main.expenses.Amount"
              column-index="5" static-type="DECIMAL">
        </plan:column>
      </plan:object>
    </plan:triple-index>
    <plan:triple-index order="6,4" permutation="PSO">
      <plan:subject>
        <plan:column name="main.expenses.rowid"
              column-index="6" static-type="UNKNOWN">
        </plan:column>
      </plan:subject>
      <plan:predicate>
        <plan:value column="main.expenses.EmployeeID"
               columnID="3887479265206160521">
        </plan:value>
```

```
      </plan:predicate>
      <plan:object>
        <plan:column name="main.expenses.EmployeeID"
              column-index="4" static-type="INT">
        </plan:column>
      </plan:object>
    </plan:triple-index>
  </plan:sort-merge-join>
  <plan:hash-join order="3,2">
    <plan:hash left="3" right="3" operator="=">
    </plan:hash>
    <plan:sort-merge-join order="3,1">
      <plan:hash left="3" right="3" operator="=">
      </plan:hash>
      <plan:triple-index order="3,0" permutation="PSO">
        <plan:subject>
          <plan:column name="main.employees.rowid"
              column-index="3" static-type="UNKNOWN">
          </plan:column>
        </plan:subject>
        <plan:predicate>
          <plan:value column="main.employees.EmployeeID"
               columnID="4691838910292433538">
          </plan:value>
        </plan:predicate>
        <plan:object>
          <plan:column name="main.employees.EmployeeID"
               column-index="0" static-type="INT">
          </plan:column>
        </plan:object>
      </plan:triple-index>
      <plan:triple-index order="3,1" permutation="PSO">
        <plan:subject>
          <plan:column name="main.employees.rowid"
              column-index="3" static-type="UNKNOWN">
          </plan:column>
        </plan:subject>
        <plan:predicate>
          <plan:value column="main.employees.FirstName"
               columnID="2346001466860406442">
          </plan:value>
        </plan:predicate>
        <plan:object>
          <plan:column name="main.employees.FirstName"
               column-index="1" static-type="STRING">
          </plan:column>
        </plan:object>
      </plan:triple-index>
    </plan:sort-merge-join>
    <plan:triple-index order="3,2" permutation="PSO">
      <plan:subject>
        <plan:column name="main.employees.rowid"
              column-index="3" static-type="UNKNOWN">
        </plan:column>
```

```
              </plan:subject>
              <plan:predicate>
                <plan:value column="main.employees.LastName"
                       columnID="3470857143136371394">
                </plan:value>
              </plan:predicate>
              <plan:object>
                <plan:column name="main.employees.LastName"
                      column-index="2" static-type="STRING">
                </plan:column>
              </plan:object>
            </plan:triple-index>
          </plan:hash-join>
```

The following is the execution pipeline for the `WHERE` clause.

```
          <plan:join-filter op="=">
            <plan:column name="main.employees.EmployeeID"
                      column-index="0" static-type="UNKNOWN">
            </plan:column>
            <plan:column name="main.expenses.EmployeeID"
                      column-index="4" static-type="UNKNOWN">
            </plan:column>
          </plan:join-filter>
        </plan:parallel-hash-join>
      </plan:hash-group>
    </plan:project>
  </plan:select>
</plan:plan>
```

10.0 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.

11.0 Copyright

MarkLogic Server 9.0 and supporting products.
Last updated: April 28, 2017

COPYRIGHT

This technology is protected by U.S. Patent No. 7,127,469B2, U.S. Patent No. 7,171,404B2, U.S. Patent No. 7,756,858 B2, and U.S. Patent No 7,962,474 B2, US 8,892,599, and US 8,935,267.

The MarkLogic software is protected by United States and international copyright laws, and incorporates certain third party libraries and components which are subject to the attributions, terms, conditions and disclaimers set forth below.

For all copyright notices, including third-party copyright notices, see the Combined Product Notices.