Olap4j Introduction - An end-user perspective

Version 0.1
July 2009 - September 2010

Prélude : General information and aims

Target Audience

This document is aimed at business intelligence developers and business analysts. Most concepts enumerated here are complex and imply a prior understanding of multidimensional databases and OLAP systems. Obviously, since olap4j is a Java API, understanding this document also requires basic Java development skills. We will not constraint this tutorial to any specific development environment nor any development tools. You will need to be able to write Java code, compile and execute it by yourself.

Olap4j was designed to be Java-platform agnostic, so you have the freedom to choose whatever platform you prefer. Olap4j main distribution is compatible with both Java versions 5 and 6. It supports JDBC specifications 3 and 4 as well. We also provide a Java 1.4 compatible binary, specially compiled through Retroweaver.

Olap4j is an OLAP API. That said, it is also a specification for database driver implementations. As of the moment this tutorial is written, there are currently two known and actively supported implementations of the driver component of olap4j. The first driver implementation is a generic XMLA web service consumer. It is capable of connecting to many database engines on the market via SOAP-style web services. The second implementation is provided by the Mondrian OLAP engine. Mondrian is a ROLAP engine, an OLAP database backed by a relational database, which requires both a properly structured relational database and a mapping schema, in XML format. Creating the mentioned database and schema file is not among the aims of this tutorial, and is covered in great lengths on the web. Resources and starting points are provided in annex to this tutorial.

To complete this tutorial, you will therefore need one of the following :

- A remote OLAP database, configured and ready to use, which exposes a XMLA service end-point. Many database platforms support this kind of access, whether they are open source or proprietary. As of June 2010, the olap4j project is constantly tested against Mondrian and Microsoft SQL Server 2005-2008. There is active development to make it compatible with SAP BW and Palo.

- A properly structured relational database schema and data, and a corresponding Mondrian schema file. Mondrian being a ROLAP type OLAP server, using it requires both these things to operate. The examples in this document are based on Mondrian's FoodMart database. You can find a copy of it along with instructions on how to set it up in Mondrian's documentation.

Roadmap

Throughout this tutorial, you will learn to create queries against OLAP databases via Java code. Olap4j is a vast and powerful API offering many different ways to achieve your data analysis goals. We will start by establishing connections to your existing OLAP databases, or for those who don't have one ready for use, show you how to create ad-hoc ones. We will then followup with basic query building, using the Query Model abstraction layer, which we will analyze, execute and observe their results. Once you have acquired those skills, we will move on to more advanced topics, such as excluding members, sorting results and

manipulating the query parse tree.

First steps and connection

The obvious first step will be to establish a connection to your OLAP database. We will offer you two different ways of establishing a connection to an OLAP database. You do not need to master both of those connection techniques in order to complete this tutorial. Olap4j being an abstraction layer, the specific way in which you connect to your OLAP database is of no importance once you obtain a connection object. Creating queries and executing them will be done in the exact same way. Without regards to the underlying connection implementation, creating simple queries and executing more advanced arbitrary MDX queries is the same for all olap4j drivers.

MDX?!

MDX is to OLAP databases as SQL is to relational ones. It is an incredibly powerful query language that was created with multidimensional database operations in mind. We will not cover the actual syntax nor elements of the MDX language in this tutorial. Sadly, few good documentation exists about it, and even less good teachers and classes. But fear not, using olap4j does not require any MDX skills whatsoever. Thanks to olap4j query model, you will be able to create basic queries without even thinking about MDX and it's implications. Of course, more advanced uses of olap4j does require knowledge of MDX, but you can postpone these more advanced topics for a later day. For further details about MDX, please refer to this document's annex.

Let's get back to connections.

As mentioned before, for those of you who don't have the luxury of a ready-to-use database, we will instruct you on how to create ad-hoc OLAP databases, thanks to the ever useful Mondrian implementation of the olap4j driver specification. In order to take that path, you will need a relational database properly structured for OLAP use, along with a Mondrian XML schema file. Those two requirements are not within the scope of this document. Refer to the annex of this tutorial for web resources. If you do not have a proper database and schema ready for use, there are half a dozen of freely distributed Mondrian schemas and databases on the web. We also provide links to them in this document's annex. Please take note that this document will show examples based on Mondrian's FoodMart database.

The XMLA implementation is faster to use and more lightweight on your development platform, but requires a remote server, fully configured and functional. Getting one of these online and ready to answer your OLAP queries is not part of this document. The Mondrian project distributes a web application, deployable in any Java application server, preconfigured for XMLA service. Please refer to the annex of this document for links to this resource.

Ad-hoc OLAP database

Mondrian basics

Mondrian is a ROLAP engine. Simply put, it requires a properly structured relational database and a mapping schema. The relational database contains the data and their tables. It will be in constant collaboration with your Mondrian engine instance and provide assistance with it's calculation power and pre-caching capabilities. The Mondrian schema is an XML file that instructs Mondrian how to exploit the underlying relational database. It will tell Mondrian where the data lies and how to obtain it. The schema file can be edited at will with any standard text editor. It is therefore very easy to tweak to your needs.

System wise, ROLAP engines have one particular requirement. They are relatively memory intensive software. It is recommended to allocate Mondrian at least one gigabyte of active memory space for it to function properly and efficiently. The amount of space varies obviously with the actual database size and structure, yet as a rule of thumb, one gigabyte is often enough for testing purposes. Deploying a ROLAP engine in a production environment obviously requires a more careful analysis and fine grained adjustments at both the applicative and operating system levels.

Creating an in-process Mondrian engine instance

This part of the tutorial will cover ad-hoc OLAP database connections, provided by the Mondrian implementation of the olap4j driver specification. First thing first. We need to initialize the Mondrian olap4j driver in the Java virtual machine classloader. In a Java runtime environments prior to version 6, this needs to be done manually in your code, although you are required to do it only once during your Java program. Subsequent calls will not have any effect and are thus useless. Java 6 has a mechanism that allows the Mondrian driver to initialize itself without a user's explicit intervention.

```
1: Class.forName("mondrian.olap4j.MondrianOlap4jDriver");
```

This previous snippet tells your Java program to read the contents of the MondrianOlap4jDriver class file. By doing so, the Mondrian driver registers itself with the Java's DriverManager, as per JDBC convention. You are now ready to create a connection.

```
1: Connection connection =
2:     DriverManager.getConnection(
3:         "jdbc:mondrian:" +
4:         "JdbcDrivers=sun.jdbc.odbc.JdbcOdbcDriver;" +
5:         "Jdbc=jdbc:odbc:MondrianFoodMart;" +
6:         "Catalog=file:/mondrian/demo/FoodMart.xml;");
```

The first line of the code above declares an object placeholder for the connection you want to create. The second line is a call to Java's JDBC API object, the DriverManager, responsible of creating connections according to the String object contents you provide to him.

The third line is very important. The `jdbc:mondrian` characters sequence is associated with Mondrian's olap4j driver implementation and by convention, it tells the DriverManager to pass the whole configuration string to it. All the subsequent lines are of no importance whatsoever to the DriverManager and are interpreted by Mondrian's driver. The actual mechanism with which this association is established is a tad more complicated than that, but it is not relevant here.

The fourth line therefore provides Mondrian with a crucial information. It tells it what JDBC driver it must use to connect to it's underlying relational database. In this example, it tells Mondrian that in order to communicate with the said relational backend, it must use the Java's built-in ODBC driver. It could have been any JDBC compatible driver, really. It does not make any importance,, as long as the driver is both JDBC compatible and is present in your classpath. One could decide to use MySQL, Oracle, or SQL Server.

The fifth line is again Mondrian specific. It tells it that in order to use the driver class specified at line four, it must initialize it with the provided string. The actual contents of this string is dictated by the driver class specified at line four. In this example, it is a very simple and concise parameter value, since ODBC drivers are very simple in their nature.

The last line relates to Mondrian and does not concern the underlying relational database anymore. It tells Mondrian where to find the schema file, also known as a catalog descriptor. The value of this parameter is quite flexible. In this example, a physical path to a local disk is used. Mondrian is also capable of fetching catalogs over the web, in which case the parameter would take a value resembling `http://example.com/my.catalog.file.xml`. It is even capable of fetching one that is placed in your Java program classpath; in a Jar file for example. In this later example, one would use a value similar to `res:com/examples/my.catalog.file.xml`.

Each olap4j driver implementation has it's specific configuration parameters. A lot of them can be included in the initial connection URL. The following table lists the most important ones which concern the Mondrian driver, but does not include all of them. For a complete list, please refer to the Mondrian olap4j driver documentation and it's list of supported parameters.

You now have rendered available to your program a fully configured Mondrian olap4j connection. By creating it, you have spawned an instance of the Mondrian OLAP engine. This instance will live for as long as your program is running and that you don't sever the link to the connection object. In the later case, the JVM garbage collector would eventually destroy it and free the previously occupied heap memory. It is also possible to explicitely destroy a connection by invoking it's `close()` method.

Remote access to an OLAP server

In order to establish a connection to a remote server, thus not instantiated by the connection object itself, olap4j contains a built-in XMLA driver implementation. Before getting into the details of connecting through this mechanism, one must understand what XMLA is and how it operates.

XMLA 101

Communications with a remote OLAP server are performed via the HTTP transport protocol. It is a very simple, well known and widely used means of communication between clients and servers. Chances are, you are using it as you read these lines. The XMLA specification leverages the power of HTTP, and the actual message payload has to conform to the SOAP message syntax. We will not go in great lengths over the structure of XMLA SOAP messages, for suffice to say that it is the means used by the XMLA driver. It is therefore required that an OLAP engine, in order to be reachable by the XMLA driver, be exposed properly over a network and is reachable by your Java program. Firewall software and similar other devices often get in the way of such communications. Database administrators have the good habit of protecting their servers from external access. You might therefore be required to collaborate with your enterprise DBA, or in the worst case scenario (from a non techie point of view, that is, us geeks see these constraints as a motivation anyway) resort to installing your own XMLA server.

XMLA also leverages HTTP authentication as a means of security. It uses BASIC authentication to do so. BASIC security, as it's name implies, is very basic in the assurances it offers, yet it is sufficient to provide a minimal authentication mechanism.

XMLA has two main SOAP methods. `Discover`, used to query the remote OLAP server about what elements are present in it's schemas, catalogs and cubes, and `Execute`. This second function serves a unique purpose; executing a MDX query.

Connecting to an XMLA server

As with the Mondrian in-process driver, it is required that the JVM be aware of the existence of the XMLA driver. Additionally to it's presence in the program classpath, Java versions prior to Java 6 require us to explicitly trigger it's initialization through the ClassLoader.

```
1: Class.forName("org.olap4j.driver.xmla.XmlaOlap4jDriver");
```

We should now be able to ask the JDBC DriverManager to establish a connection with our remote OLAP server.

```
1: Connection connection =
2:     DriverManager.getConnection(
3:         "jdbc:xmla:" +
4:         "Server=http://example.com/applicationContext/xmla",
5:         "username",
```

```
6:          "password");
```

The first two lines are exactly the same as with a Mondrian in-process connection. We create a variable to hold on to our resulting connection object, which will be provided by the `DriverManager.getConnection()` invocation.

The third line tells the DriverManager that we require a connection from the XMLA driver. The `jdbc:xmla` prefix to the connection string is associated with the XMLA driver, and instructs the DriverManager that this connection string is to be relayed to the XMLA driver class.

The fourth line is specific to the XMLA driver. It provides it with the URL at which it will be able to reach the remote XMLA server.

The fifth and sixth lines are quite different from our previous Mondrian in-process connection example. As a matter of fact, they are not even part of the connection string. They are parameters of the `getConnection()` method that were not used in the previous example. Since most XMLA servers require authentication, we find it convenient here to talk briefly about how exactly the XMLA driver passes it's authentication credentials to the remote server.

By using this method signature, the XMLA driver knows explicitly what credentials it should use. Note that those parameters are optional, and we could as well send `null` values, or omit them altogether if no authentication is required by the server. In the above snippet, we tell the XMLA driver to personify the user named `username`, and that this user's password is the very hard to find and oh so very cryptic password; `password`. Before being sent over the network, the credentials will be encoded as a Base64 hashed value, in conformity with HTTP's BASIC authentication standard.

It is also possible to send the credentials as part of the `Server` parameter. In this case, the server end-point parameter would look something like this.

```
Server=http://username:password@example.com/applicationContext/xmla
```

The URL is totally dependent on your XMLA server. The "`xmla`" part in the above example is totally arbitrary and does not reflect at all any standard or naming convention. Please refer to your server's documentation for the exact URL to use.

XMLA connections can be further customized. The two most commonly used parameters are `Catalog` and `Datasource`. They tell the driver manager to pin the created connection object to a specific catalog and datasource names. The complete list of supported parameters can be found in the XMLA driver javadoc.

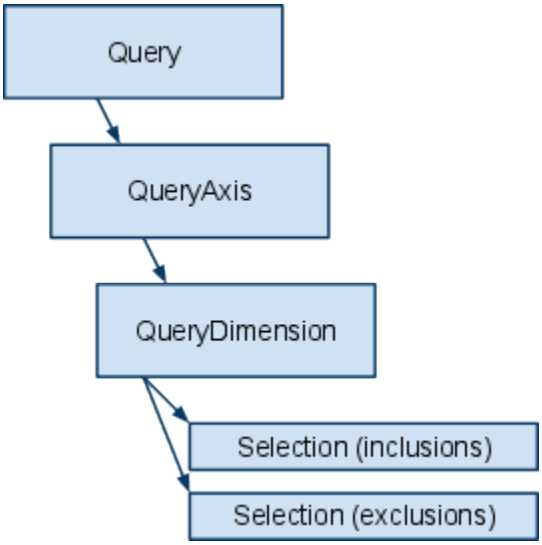
First simple query using the Query Model API

This section will cover the basics of building a very simple query against your database. In order to do that, we will be using the Query Model API (QM), which is part of olap4j. There are many ways of building queries with olap4j. Some require knowledge of MDX, some don't. The QM is part of the later, so it is a very good starting point for all of you readers.

The QM does not support the full range of possible MDX operations, and it never was it's aim. The QM allows users to create simple queries and/or build graphical interfaces on it with minimal effort. e will be covering more advanced queries later on in this document.

Anatomy of the Query Model

The QM was designed to represent what a user might want to do with an OLAP cube, and therefore represent all those operations with a simple set of objects and methods.



At the root level we find the Query object itself. It contains a list of `QueryAxis`. Each `QueryAxis` correspond to either the *columns*, the *rows*, the *filter* or the *unused* axis. Although the first three axis are pretty obvious in their representation, the last one, *unused*, needs further explaining. We will get to that later on.

A `QueryAxis` contains from zero to N number of `QueryDimension` objects. Each `QueryDimension` objects corresponds to an actual dimension from your source cube.

`QueryDimension` objects contains from zero to N `Selection` objects, that you can either include in or exclude from the query.

`Selection` objects are a meta representation of member sets. We will cover them later on in this document.

Creating a query

The first step is to put our hands on an object that represents a *cube* in our database. Cubes are specific to OLAP databases, so the JDBC standard had to be extended from this point on. In the previous section, you obtained a Connection object. This object is specific to JDBC *relational* connections and obviously cannot represent all the intricate particularities of their next-gen cousin, OLAP databases. We therefore need to unwrap the connection to gain access to it's full set of functionalities.

```
1: Connection rConnection =
2:     (OlapWrapper) DriverManager.getConnection(
3:         "jdbc:xmla:" +
4:         "Server=http://example.com/applicationContext/xmla",
5:         "username",
6:         "password");

7: OlapConnection oConnection =
8:     rConnection.unwrap(OlapConnection.class);
```

Notice the `OlapWrapper` casting? The JDBC standard has undergone important changes between versions 5 and 6, one of them being the ability to unwrap a connection object to gain access to it's true connection API. Olap4j provides this nifty wrapper class as a means for Java 5 code to mimic this Java 6 feature. In the listing above, the `oConnection` object now gives us access to the OLAP concepts which were not accessible previously.

```
1: OlapConnection oConnection =
2:     rConnection.unwrap(OlapConnection.class);

3: NamedList<Cube> cubes =
4:     connection.getSchema().getCubes();

5: Cube salesCube = cubes.get("Sales");

6: Query myQuery =
7:     new Query("SomeArbitraryName", salesCube);
```

In the previous listing, at line 3, we declare a variable of class `NamedList`. The `NamedList` interface is proper to olap4j and is a simple extension to the regular `List`, the difference being that we can obtain directly the element it contains and which has the arbitrary supplied name. In this case, we asked for a `Cube` object named *Sales*, as seen in line 5.

Lines 6 and 7 create the `Query` itself. A `Query` needs an arbitrary name and a `Cube` object to query against. Notice that we didn't need to instantiate anything other than the root `Query`. Olap4j will setup initialize everything for you and make sure that the `Query` is in an operational state.

Building a query

The first step is to place the dimensions on their axis.

```
1: Query myQuery =
2:     new Query("SomeArbitraryName", salesCube);

3: QueryDimension productDim =  myQuery.getDimension("Product");
4: QueryDimension storeDim =    myQuery.getDimension("Store");
5: QueryDimension timeDim =     myQuery.getDimension("Time");

6: myQuery.getAxis(Axis.COLUMNS).addDimension(productDim);
7: myQuery.getAxis(Axis.ROWS).addDimension(storeDim);
8: myQuery.getAxis(Axis.FILTER).addDimension(timeDim);
```

The listing above is pretty straight forward. We took three different dimensions of the underlying cube and placed them on three different axis. It is also possible to place more than one dimension on the same axis. In the later case, the QM will automatically create a Crossjoin function between the two dimensions. The order in which the dimensions are placed on the axis can be specified, either when placing it on the axis or afterwards, like demonstrated by the following code listing.

```
1: QueryDimension productDim =  myQuery.getDimension("Product");
2: QueryDimension storeDim =    myQuery.getDimension("Store");

3: myQuery.getAxis(Axis.COLUMNS).addDimension(productDim);
4: myQuery.getAxis(Axis.COLUMNS).addDimension(storeDim);

5: myQuery.getAxis(Axis.COLUMNS).addDimension(storeDim);
6: myQuery.getAxis(Axis.COLUMNS).addDimension(0, productDim);

7: myQuery.getAxis(Axis.COLUMNS).addDimension(storeDim);
8: myQuery.getAxis(Axis.COLUMNS).addDimension(productDim);
9: myQuery.getAxis(Axis.COLUMNS).pushDown(0);
```

In the code above, the sets of lines 3-4, 5-6 and 7-9 all have the exact same effect and will produce the exact same query. They are just different ways to place the dimensions on the axis. By default, dimensions are placed on the axis in the order that the code has placed them. It is also possible to specify the index at which to insert the dimensions. The result conforms to the `java.util.List` behavior; objects that occupied indexes subsequent to the arbitrary supplied one will be *right shifted*. The last example shows how to *push down* or *pull up* a dimension on the axis, or simply put, augment or diminish it's zero based index.

Now is the time to look at the output of the QM.

Validating a query

`Query` objects expose a very practical method to validate the current MDX representation of your work. The following code shows how to display the current state of our query as a MDX text.

```
1:  Query myQuery =
2:      new Query("SomeArbitraryName", salesCube);

3:  QueryDimension productDim =  myQuery.getDimension("Product");
4:  QueryDimension storeDim =    myQuery.getDimension("Store");
5:  QueryDimension timeDim =     myQuery.getDimension("Time");

6:  myQuery.getAxis(Axis.COLUMNS).addDimension(productDim);
7:  myQuery.getAxis(Axis.ROWS).addDimension(storeDim);
8:  myQuery.getAxis(Axis.FILTER).addDimension(timeDim);

9:  myQuery.validate();

10:  System.out.println(
11:     myQuery.getSelect().toString());
```

The code above would produce the following output.

```
SELECT
{[Product].[All Products]} ON COLUMNS,
{[Store].[All Stores]} ON ROWS
FROM [Sales]
WHERE ([Time].[1997])
```

The QM API has figured out that default inclusions have to be performed for the query to be valid and executable. If no inclusions were performed on the `QueryDimension` objects, as it is the case above, the QM classes will perform inclusions on the default hierarchies and members defined in your cube structure. In this example, the *Sales* cube defined that the year 1997 was the default member for the *Time* dimension.

The default inclusions were performed during the call at line 9. The `validate()` method does three things for you.

- **Performs default selections**
  Iterates over all the dimensions that are not on the `UNUSED` axis. If no selections are included in the `QueryDimension` object, the default member of the default hierarchy is selected and included.
- **Validates the presence of at least one dimensions on both `ROWS` and `COLUMNS` axis**

Verifies that at least one `QueryDimension` is present on each of the `ROWS` and `COLUMNS QueryAxis`. If none are present, it throws an `OlapException`.

- **Tries to build a MDX parse tree**
  In order to make sure that the `Query` object is in a valid state, the QM API will try to represent your current QM structure as a full MDX parse tree. If this operation fails, it is most likely that the current structure is not valid and an `OlapException` will be thrown explaining what exactly is wrong with your current query.

Creating selections to include or exclude from a query

The next step will be to perform inclusions or exclusions on members of our query dimensions. As we said earlier, in the QM realm, we are allowed to include or exclude selections of members. The selections are not composed of explicit lists of members, but rather leverage the power of tree operations that the MDX syntax offers to us. A `Selection` object is therefore composed of the two following attributes.

- A reference member. In the previous example, `[Time].[1997]` would be our reference member unique name.
- An operator relative to the reference member. In the previous example, the `Member` operator was implicitly used and represents the member itself. Other valid operators could be `CHILDREN`, `SIBLINGS`, or `ANCESTORS`. There are many more available, so consult the [Selection.Operator enumeration javadoc] for the complete list.

The following code listing shows different ways to perform inclusions or exclusions of members.

```
1:  Query myQuery =
2:      new Query("SomeArbitraryName", salesCube);

3:  QueryDimension productDim =  myQuery.getDimension("Product");
4:  QueryDimension storeDim =    myQuery.getDimension("Store");
5:  QueryDimension timeDim =     myQuery.getDimension("Time");

6:  myQuery.getAxis(Axis.COLUMNS).addDimension(productDim);
7:  myQuery.getAxis(Axis.ROWS).addDimension(storeDim);
8:  myQuery.getAxis(Axis.FILTER).addDimension(timeDim);

9:  Member year1997 = salesCube.lookupMember("Time", "1997");
10: timeDim.include(year1997);

11: productDim.include(
12:     Selection.Operator.CHILDREN, "Product", "Drink",
"Beverages");

13: productDim.exclude(
14:     "Product", "Drink", "Beverages", "Carbonated Beverages");

15:  myQuery.validate();

16:  System.out.println(
```

```
17:     myQuery.getSelect().toString());
```

The resulting MDX statement would be the following. (The indentation was modified from the actual output to facilitate reading.)

```
SELECT
{
    Except(
        {[Product].[All Products].[Drink].[Beverages].Children},
        {[Product].[All Products].[Drink].[Beverages].[Carbonated
Beverages]}
    )
} ON COLUMNS,
{
    [Store].[All Stores]
} ON ROWS
FROM [Sales]
WHERE ([Time].[1997])
```

As you can see, the exclusions have precedence on the inclusions, so careful thinking is required in order to obtain the desired results.

Executing a query

We are finally ready to execute our first query. The execution itself is very easy to trigger.

```
1:  CellSet results = myQuery.execute();
```

This call to the `execute()` method will parse the current QM attributes and generate an object model representing it. This object model will then be passed to the same OLAP connection we used to create the query. The returned object is a `CellSet`. `CellSet` is to OLAP queries as `ResultSet` is to relational ones.

Iterating over the results

In a relational ResultSet, we would iterate over rows of data and then iterate over cells contained in those rows. In the OLAP world, since a query could have as many axis as we see fit, the same paradigm cannot apply anymore. We will therefore talk about `Positions` along `Axis`. Consequently, a cell is found by specifying a tuple of positions along every axis. Based on our previous examples, since only two axis were used (the slicer axis is not considered an axis *per se*) we can iterate over our `CellSet` with the following code.

```
1:  for (Position axis_0 :
results.getAxes().get(Axis.ROWS.axisOrdinal()).getPositions()) {
2:      for (Position axis_1 :
results.getAxes().get(Axis.COLUMNS.axisOrdinal()).getPositions()) {
3:          Cell currentCell = results.getCell(axis_0, axis_1);
4:          Object value = currentCell.getValue();
5:      }
6:  }
```

If we had built a query with more than two axis, using a different object model than the QM, the `results.getCell()` function would have needed as many Position objects as arguments as there are axis present in the query. The resulting object of an OLAP query is always a `CellSet`, whatever means were used to build the query.

Displaying the results

Olap4j makes available a very handy class to display the results of an OLAP query in a human-readable and practical format. The following code listing will parse and display a `CellSet` object on the system console.

```
1:  CellSetFormatter formatter =
2:      new RectangularCellSetFormatter(false);

3:  formatter.format(
4:      cs,
5:      new PrintWriter(System.out, true));
```

The above code will output to the console something similar to this.

```
|                       | All Stores |
+------+----------------+------------+
| Food | Baked Goods    |      7,870 |
|      | Baking Goods   |     20,245 |
|      | Breakfast Foods |      3,317 |
|      | Canned Foods   |     19,026 |
|      | Canned Products |      1,812 |
|      | Dairy          |     12,885 |
|      | Deli           |     12,037 |
|      | Eggs           |      4,132 |
|      | Frozen Foods   |     26,655 |
|      | Meat           |      1,714 |
|      | Produce        |     37,792 |
|      | Seafood        |      1,764 |
|      | Snack Foods    |     30,545 |
```

```
|      | Snacks          |      6,884 |
|      | Starchy Foods   |      5,262 |
```

Sorting axis and dimensions

Now that we can execute our queries and visualize their results, we will learn how to customize them. The QM allows us to sort our query's axis and dimensions according to different parameters, but first thing first. We must understand what exactly it means to sort elements in the OLAP world.

Sorting an axis

Sorting an axis will reorder all the inclusions according to the parameters we specify. The following code listing shows how to build a very basic query and sort the products according to their sales volume. The example below will be used throughout the current section about sorting.

```
1:  Query myQuery =
2:      new Query("SomeArbitraryName", salesCube);

3:  myQuery.getAxis(Axis.ROWS).addDimension(
4:      myQuery.getDimension("Product"));
5:  myQuery.getAxis(Axis.ROWS).addDimension(
6:      myQuery.getDimension("Store Type"));
7:  myQuery.getAxis(Axis.COLUMNS).addDimension(
8:      myQuery.getDimension("Time"));

9:  myQuery.getDimension("Store Type").include(
10:     Selection.Operator.CHILDREN,
11:     "Store Type", "All Store Types");
12: myQuery.getDimension("Store Type").exclude(
13:     "Store Type", "HeadQuarters");
14: myQuery.getDimension("Product").include(
15:     "Product", "Food", "Seafood");
16: myQuery.getDimension("Product").include(
17:     "Product", "Food", "Meat");

18: myQuery
19:     .getAxis(Axis.ROWS)
20:     .sort(SortOrder.DESC);

21: myQuery.validate();

22: System.out.println(
23:     myQuery.getSelect().toString());

24: CellSet cs = myQuery.execute();
```

```
25: CellSetFormatter formatter = new
RectangularCellSetFormatter(true);
26: formatter.format(
27:     cs,
28:     new PrintWriter(System.out, true));
```

As you can observe at lines 18-20, we've defined a sort order to the rows axis. The `QueryAxis.sort()` method takes a value of the `SortOrder` enumeration as it's parameter. Before covering those in detail, let's take a look at the previous code output.

```
SELECT
{[Time].[1997]} ON COLUMNS,
Order(
    CrossJoin(
        {[Product].[All Products].[Food].[Seafood],
        [Product].[All Products].[Food].[Meat]},
        {Except(
            {[Store Type].[All Store Types].Children},
            {[Store Type].[All Store Types].[HeadQuarters]}
        )}
    ),
    [Measures].[Unit Sales],
    DESC
) ON ROWS
FROM [Sales]

|                                     | 1997  |
+------+---------+---------------------+-------+
| Food | Seafood | Supermarket         | 1,032 |
|      |         | Deluxe Supermarket  |   476 |
|      |         | Gourmet Supermarket |   164 |
|      |         | Mid-Size Grocery    |    61 |
|      |         | Small Grocery       |    31 |
|      | Meat    | Supermarket         |   919 |
|      |         | Deluxe Supermarket  |   495 |
|      |         | Gourmet Supermarket |   159 |
|      |         | Mid-Size Grocery    |    88 |
|      |         | Small Grocery       |    53 |
```

Our result table rows are now sorted in descending order, relative to the value of the `[Measures].[Unit Sales]` member. Olap4j's query model had to guess the actual values to sort onto, and by default will use the cube's default measure. There are overrides to the sort method that allows you to explicitly define by what measure or expression literal the axis should be sorted. Using those would allow you to sort by calculated members or any literal expression literal you fancy. Please consult the QueryAxis API for more details.

Interestingly enough, the resulting table shows that our results are indeed sorted in

descending order, but only in the scope of their respective hierarchy level. This is normal and in conformity with MDX standard behaviour. In order to sort all the rows, regardless of their hierarchies, MDX provides a second set of sort operators. They are also included in the `SortOrder` enumeration provided by olap4j. In the example code above, using the SortOrder.BDESC value at line 20 would modify the output like this.

```
SELECT
{[Time].[1997]} ON COLUMNS,
Order(
    CrossJoin(
        {[Product].[All Products].[Food].[Seafood],
        [Product].[All Products].[Food].[Meat]},
        {Except(
            {[Store Type].[All Store Types].Children},
            {[Store Type].[All Store Types].[HeadQuarters]}
        )}
    ),
    [Measures].[Unit Sales],
    BDESC
) ON ROWS
FROM [Sales]
```

```
|                                       | 1997  |
+------+---------+---------------------+-------+
| Food | Seafood | Supermarket         | 1,032 |
|      | Meat    | Supermarket         |   919 |
|      |         | Deluxe Supermarket  |   495 |
|      | Seafood | Deluxe Supermarket  |   476 |
|      |         | Gourmet Supermarket |   164 |
|      | Meat    | Gourmet Supermarket |   159 |
|      |         | Mid-Size Grocery    |    88 |
|      | Seafood | Mid-Size Grocery    |    61 |
|      | Meat    | Small Grocery       |    53 |
|      | Seafood | Small Grocery       |    31 |
```

Our results table rows are now sorted properly, regardless of inherent hierarchy.

Sorting a dimension's members

Olap4j's query model also allows you to sort the actual dimension members by their names. There are many use cases for this, one of them being a long list of clients in a given shop. One might expect the list of clients to be sorted alphabetically in order to simplify browsing. Sorting a dimension is done in the same way as sorting an axis.

We will create a new query that lists all the customers of the Sales cube.

```
18: myQuery
19:     .getAxis(Axis.ROWS)
20:     .sort(SortOrder.DESC);
```

Final thoughts on olap4j's Query Model

As you can see, building queries is very simple with olap4j's Query Model. With a strict minimum of code, we can generate the MDX query to fulfill many analytical needs. We have to keep in mind though that what it gains in simplicity came as a trade-off on flexibility. Olap4j's QM cannot leverage the full power of the MDX query language; it never was meant to do that in the first place.

There are more advanced ways to build queries with olap4j, and the following sections of this document will explore them.

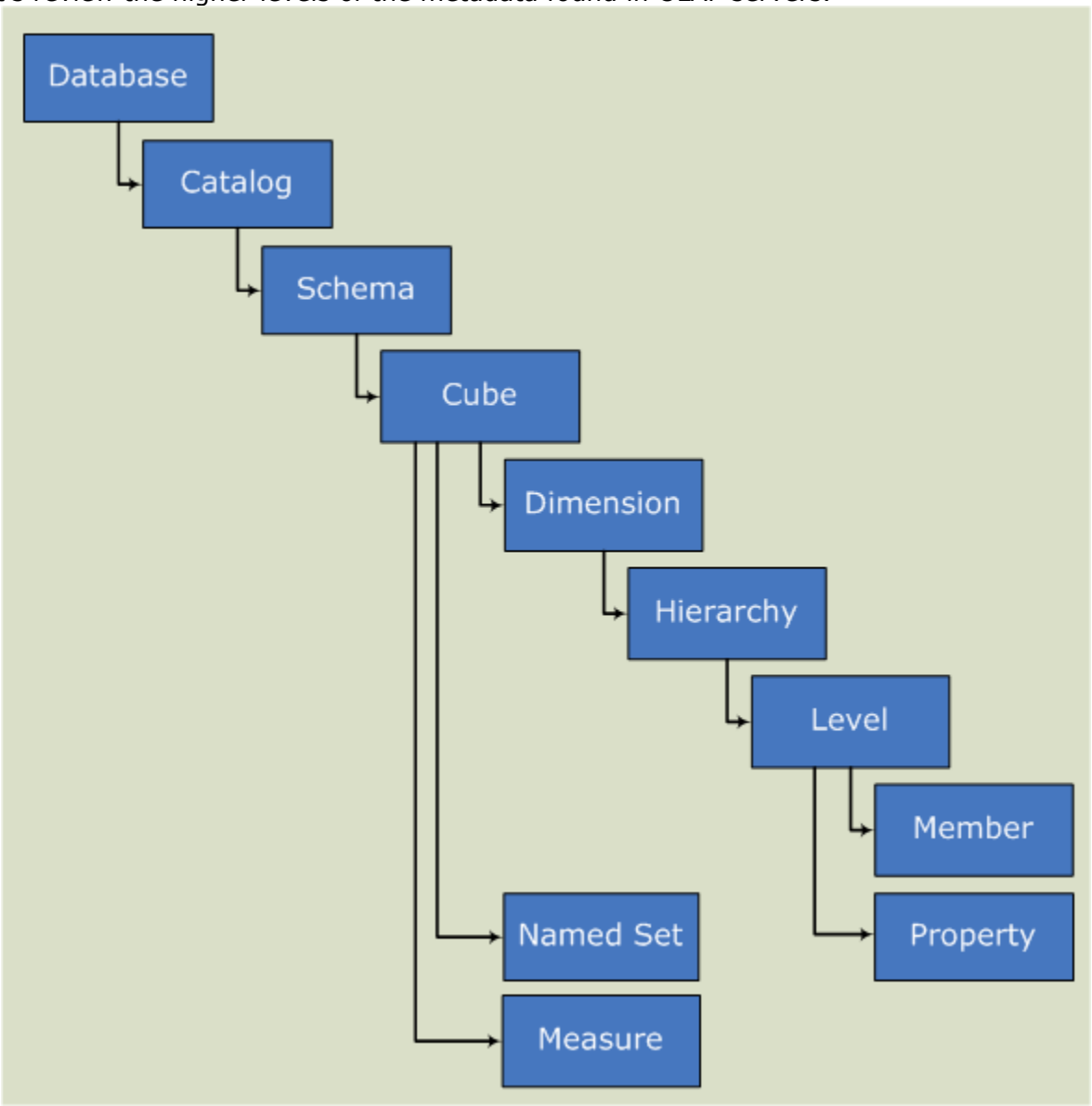
Advanced queries

Olap4j metadata anatomy

Building your own MDX queries allows you to harness the full power of the MDX query language. Before we start writing MDX queries, let's review the metadata structure we will have to deal with.

Overview of the metadata higher levels

Let's review the higher levels of the metadata found in OLAP servers.



The schema above illustrates the hierarchical organization of an OLAP server's metadata. Olap4j replicates this hierarchy and uses it to represent all the elements which are available in the remote OLAP servers. Below, you will find a brief explanation of each of these components.

Element	Description
Database	The database node is the root of your server connection. Some servers might expose more than one database root objects to bind a connection to. Olap4j needs to know which one to use. This can be configured via either a JDBC URL specific parameter, or via `OlapConnection.setDatabase(String)`. If you establish a connection without specifying which database to bind to, olap4j connections are expected to use the first one available. To browse the list of available databases, use `OlapDatabaseMetaData.getDatabases()`.
Catalog	The catalog is a collection of schemas. An olap4j connection needs to know which catalog to use. This can be configured via either a JDBC URL specific parameter, or via `OlapConnection.setCatalog(String)`. If no catalog name was specified, olap4j drivers are expected to use the first one available on the OLAP server. To browse the list of available catalogs, use `OlapDatabaseMetaData.getCatalogs()`. To obtain a list of Catalog Java objects from the server, use `OlapDatabaseMetaData.getOlapCatalogs()`.
Schema	The schema is rarely used in OLAP servers, yet always present. Some servers, SQL Server for example, won't even give it a name. It's literally a second class citizen in the OLAP hierarchy. To browse the list of available schemas, use `OlapDatabaseMetaData.getSchemas()`.
Cube	The cubes are ensembles of data against which you build MDX queries.

Browsing the metadata programmatically

Olap4j exposes to programmers the complete set of server metadata. It does so using a subclass of `java.util.List` called `NamedList`. This extension allows you to obtain list elements by their name as well as their index. This allows us to explore the metadata in the following way. Let's say we want a list of all the dimensions in a cube named 'Sales', starting from the root of the metadata. This requirement would translate in the following code.

```
1:  olapConnection
2:    .getMetaData()
3:      .getOlapCatalogs()
4:        .get("Catalog Name")
```

```
5:           .getSchemas()
6:             .get("Schema name")
7:               .getCubes()
8:                 .get("Sales")
9:                   .getDimensions()
```

The metadata is very easy to browse, and returns domain specific objects which represent available resources on the OLAP server. We will not cover the entire OLAP metadata structure here. If you would like to obtain the specific API for those, please consult olap4j's metadata API.

Arbitrary MDX queries

Now that we know how to browse the metadata, let's explore how queries are represented by olap4j and how we can programmatically create new queries. This section assumes that you are already familiar with MDX queries and know how to write them. Through this section, you will learn how to parse a query, how to validate it against your server and lastly how queries are represented and how you can alter this representation in order to generate new queries.

Olap4j's connections expose to you their internal parser and validator. These two classes are immensely useful for OLAP application developers, for they allow you to analyze and validate user defined MDX queries before they are even executed. Imagine a user interface where users write their own MDX queries. One way to sanitize the inputs would be to take advantage of olap4j's query parser and validator in order to know up front if the query is likely to succeed. Should any of those two utility classes detect a problem with the query, they will provide a detailed error message in order to help the user fix his query. Another main advantage is that faulty queries will be detected before they make it to the execution stage, thus preventing the server from being overloaded by half baked queries which it cannot execute anyways.

Both the parser and validator of a connection are accessible through the `OlapConnection.getMdxParserFactory`(). The MdxParserFactory exposes two methods; `createMdxParser()` and `createMdxValidator()`. Let's explore those two objects in detail.

Parsing a query with the MdxParser

An OLAP connection MDX parser is useful for validating the syntax of a query and detecting errors which prevent olap4j from understanding what the query means. Should any error be detected, the parser will tell you where in the query the error is happening and why it has happened. The following code illustrates how to use a connection's parser.

```
1:  MdxParserFactory pFactory = olapConnection.getParserFactory();
2:  MdxParser parser = Mdx4jParserFactory.createMdxParser();

3:  try {
4:      SelectNode parsedObject = parser.parseSelect("SELECT {} on
Bacon FROM [MyCube]");
5:  } catch (OlapException e) {
6:      System.out.println(e.getMessage());
7:  }
```

Obviously, there is no such axis with the name 'Bacon'. The code above would there fore result in the following output.

```
Syntax error at [1:14, 1:18], token 'Bacon'
```

Validating a query with the MdxValidator

Now that we know if a query is grammatically correct, this does not ensure that the query is actually valid. Take the following query for example.

```
SELECT
    {
        [Product].[All Products].[Drink].[Beverages].Children
    }
    ON COLUMNS

FROM [Sales]

WHERE ([Time].[2001])
```

Although the query syntax is valid, it won't return any result and will fail once executed. The filter axis is set to `[Time].[2001]`, which does not exist in the cube. In order to detect this type of error, you would have to execute the query on the server, thus augmenting the server work load. Olap4j's connection and metadata caching abilities can take care of the validation through the MdxValidator interface and validate the identifiers of your query on the client side. The following code demonstrates how.

```
1:  MdxParserFactory pFactory = olapConnection.getParserFactory();
2:  MdxParser parser =
Mdx4jParserFactory.createMdxParser(olapConnection);

3:  SelectNode parsedObject = parser.parseSelect("SELECT {} on
Bacon FROM [Sales]");

4:  MdxValidator validator =
pFactory.createMdxValidator(olapConnection);
5:  try {
6:      validator.validateSelect(parsedObject);
7:  } catch (OlapException e) {
8:      System.out.println(e.getMessage());
9:  }
```

MDX Object model

Now that you know how to transform a textual query into an object model using a connection parser, we will explore the returned objects. MDX queries are represented as nodes of the class `org.olap4j.mdx.ParseTreeNode`. Nodes can have children of the same class. All the parts of the query are represented as nodes; keyword literals, parenthesis, MDX built-in functions, etc. The root node is of the specialized type

`org.olap4j.mdx.SelectNode`. Let's examine some of the methods of the `SelectNode`.

Method	Description
`getAxisList() : List<AxisNode>`	Returns a list of the axis nodes present in this query.
`getFilterAxis() : AxisNode`	Helper method which returns the slicer axis defined by the WHERE clause of the query. Might be empty.
`getWithList() : List<ParseTreeNode>`	Returns a list of calculated members and sets defined as the WITH clause of this SelectNode.
`setFrom(ParseTreeNode node) : void`	Sets the FROM clause of this SELECT statement. Useful to change a query's cube name.

There are more methods available to play with. There are also many available node types which cover the full complexity of the MDX language. Everything is encapsulated in the `org.olap4j.mdx` package. Let's take a basic query and see how we interact with it. We will start by parsing an empty query template and adding nodes to it.

```
1:  MdxParserFactory pFactory = olapConnection.getParserFactory();
2:  MdxParser parser =
Mdx4jParserFactory.createMdxParser(olapConnection);

3:  SelectNode parsedObject = parser.parseSelect("SELECT {} ON ROWS
FROM [Sales]");

4:  List<IdentifierNode.Segment> segments =
5:      IdentifierNode.parseIdentifier("[Product]");

6:  selectNode
7:      .getAxisList()
8:          .get(Axis.COLUMNS.axisOrdinal())
9:              .setExpression(
10:                  new IdentifierNode(segments));

11: System.out.println(
12:     selectNode.toString());
```

In the above code, we created a list of identification segments which correspond to the Product dimension of the Sales cube. We then selected the axis list at line 7, focused on the `COLUMNS` axis at line 8, and inserted a new `IdentifierNode` object at lines 9 and 10. The `IdentifierNode` is a specialized node subclass which represents literal identifiers. It is a quick and easy way to insert nodes in the parse tree, but there are drawbacks to consider. An `IdentifierNode` does not convey information on what exactly it identifies, other than a list of name segments. We cannot therefore determine what type of MDX metadata object is represented. Is it a Level? A Hierarchy? Calling `ParseTreeNode.getType()` would not help us because the type returned would be "IdentifierNode". One way to find out would be to

resolve the identifier back against the cube. The correct way, yet more complex, is to use specialized nodes to represent the Product dimension. By using the specialized nodes subclass, we will be able to take advantage of `ParseTreeNode.getType()` in order to identify the nodes in the tree.

```
1:  MdxParserFactory pFactory = olapConnection.getParserFactory();
2:  MdxParser parser =
Mdx4jParserFactory.createMdxParser(olapConnection);

3:  SelectNode parsedObject = parser.parseSelect("SELECT {} ON ROWS
FROM [Sales]");

4:  final Dimension productDim =
5:          olapConnection
6:              .getMetaData()
7:                  .getOlapCatalogs()
8:                      .get("FoodMart")
9:                          .getSchemas()
10:                             .get("FoodMart")
11:                                 .getCubes()
12:                                     .get("Sales")
13:                                         .getDimensions()
14:                                             .get("Product");

15: selectNode
16:     .getAxisList()
17:         .get(Axis.COLUMNS.axisOrdinal())
18:             .setExpression(
19:                 new CallNode(
20:                     null,
21:                     "{}",
22:                     Syntax.Braces,
23:                     new DimensionNode(
24:                         null,
25:                         productDim)));

26: System.out.println(
27:     selectNode.toString());
```

The lines 4 to 14 are used to locate the Product dimension object from the connection's metadata. The 15 to 18 are the same as the previous example. Lines 19-25 are new though. What they do is create a brackets node (implemented by a `CallNode` object with the `Braces` syntax flavor) to wrap the dimension, then inserts a `DimensionNode` for a child. This way of playing with the MDX parse tree is more robust, for when we read back the object tree, each node reports a more useful and accurate node type identifier.

Scenarios, writeback and statistical simulations

One of the lesser known features of olap4j is the writeback capabilities. In the olap4j toolkit, it is called Scenarios. Scenarios allow users to change values of the result of a query and observe the results. Let's take a basic example using the following query.

```
SELECT
  { [Product].[All Products].[Drink].[Beverages].Children } ON
COLUMNS,
  { [Store].[All Stores].[USA], [Store].[All Stores].[USA].Children
} ON ROWS
FROM
  [Sales]
WHERE
  ([Time].[1997])
```

The query above selects the children of the Beverages member on the columns axis and places the USA stores followed by the children of USA on the rows axis. Displaying the resulting data would look something like this.

```
|          |
Drink
|
|          |
Beverages
|
|          | Carbonated Beverages | Drinks    | Hot Beverages |
Pure Juice Beverages |
+-----+----+----------------------+-----------+---------------+----
------------------+
| USA |    |                3,407 |     2,469 |         4,301
|                3,396 |
|     | CA |                  923 |       766 |         1,208
|                  989 |
|     | OR |                  858 |       632 |         1,078
|                  817 |
|     | WA |                1,626 |     1,071 |         2,015
|                1,590 |
```

Now, let's explore how olap4j allows you to modify the data and simulate different statistical scenarios.

```
1:  final Scenario scenario =
2:    olapConnection.createScenario();

3:  olapConnection.setScenario(scenario);

4:  final String query =
5:    "SELECT { [Product].[All
Products].[Drink].[Beverages].Children } ON COLUMNS, "
6:    + "{ [Store].[All Stores].[USA], [Store].[All
Stores].[USA].Children } ON ROWS FROM [Sales] WHERE ([Time].[1997],
"
7:    + "[Scenario].[" + scenario.getId() + "])";

8:  final CellSet cellSetBefore =
9:    olapConnection
10:     .prepareOlapStatement(query)
11:     .executeQuery();

12: cellSetBefore
13:   .getCell(
14:     Arrays.asList(3, 0))
15:   .setValue(
16:     1000000,
17:     AllocationPolicy.EQUAL_ALLOCATION);

18: final CellSet cellSetAfter =
19:   olapConnection
20:     .prepareOlapStatement(query)
21:     .executeQuery();
```

The code above does the following. Lines 1 and 2 create a scenario object. This scenario is provided by the connection object. We then set the scenario as the active one of that particular connection. Multiple scenarios can live simultaneously without any concurrency issues. Note that the query now has one supplemental element, at line 7, to the WHERE clause. The OLAP server maintains the scenario data on a seperate axis for ease of use. This design is database dependant. Mondrian decided to implement it this way. Other database engines are free to implement it however they wish.

Now that the query is created and has returned results (lines 8-11), we can modify the values of the cell. The call to Cell.setValue() (lines 15-17) takes two parameters. A new value, obviously, and an allocation policy. Because the data is organized in a hierarichal way, modifying the value of a cell might impact other values based on the children of the tuples of the cell whose result you modified. To maintain the required data integrity, using a scenario requires you to instruct olap4j how to distribute the data across the cell children. The allocation policies are described below.

Allocation Policy	**Description**
EQUAL_ALLOCATION	Every atomic cell that contributes to the updated cell will be assigned an equal value that is:

	<atomic cell value> = <value> / Count(atomic cells contained in <tuple>)
EQUAL_INCREMENT	Every atomic cell that contributes to the updated cell will be changed according to: *<atomic cell value> = <atomic cell value> + (<value> - <existing value>) / Count(atomic cells contained in <tuple>)*
WEIGHTED_ALLOCATION	Every atomic cell that contributes to the updated cell will be assigned an equal value that is: *<atomic cell value> = <value> * <weight value expression>* Takes an optional argument, weight_value_expression. If weight_value_expression is not provided, the following expression is assigned to it by default: *<weight value expression> = <atomic cell value> / <existing value>* The value of weight value expression should be expressed as a value between 0 and 1. This value specifies the ratio of the allocated value you want to assign to the atomic cells that are affected by the allocation. It is the client application programmer's responsibilffity to create expressions whose rollup aggregate values will equal the allocated value of the expression.
WEIGHTED_INCREMENT	Every atomic cell that contributes to the updated cell will be changed according to: *<atomic cell value> = <atomic cell value> + (<value> - <existing value>) * <weight value expression>* Takes an optional argument, weight_value_expression. If weight_value_expression is not provided, the following expression is assigned to it by default: *<weight value expression> = <atomic cell value> / <existing value>* The value of weight value expression should be expressed as a value between 0 and 1. This value specifies the ratio of the allocated value you want to assign to the atomic cells that are affected by the allocation. It is the client application programmer's responsibility to create expressions whose rollup aggregate values will equal the allocated value of the expression.

Executing the query again with the modified cell value, using the EQUAL_ALLOCATION policy, should therefore give out the following results.

```
|          |
Drink
|
|          |
Beverages
|
|          | Carbonated Beverages | Drinks    | Hot Beverages |
Pure Juice Beverages |
+-----+----+----------------------+-----------+---------------+----
------------------+
| USA |    |                3,407 |     2,469 |         4,301
|            1,000,000 |
|     | CA |                  923 |       766 |         1,208
|              285,584 |
|     | OR |                  858 |       632 |         1,078
|              241,788 |
|     | WA |                1,626 |     1,071 |         2,015
|              472,628 |
```

Annex

Online Resources

olap4j

Project home
http://olap4j.org

Project management
http://sourceforge.net/projects/olap4j/

Latest API
http://olap4j.org/head/api/index.html

Code samples, including a fully configured Eclipse project with the Foodmart database
https://code.google.com/p/olap4j-demo/

Mondrian

Project home
http://www.pentaho.com/products/analysis/

Project releases
http://sourceforge.net/projects/mondrian/

Project management
http://jira.pentaho.com/browse/MONDRIAN

MDX

Microsoft's MDX language reference
http://msdn.microsoft.com/en-us/library/ms145506.aspx

Nice introduction to MDX. The tutorial was never finished, but it is a great resource none the less.
http://www.mosha.com/msolap/articles/MDXForEveryone.htm