olap4j Specification

Table of Contents

olap4j Specification

Authors: Julian Hyde, Barry Klawans
Version: 1.0
Revision: $Id: olap4j_fs.html 250 2009-06-25 21:35:50Z jhyde $ (log)
Last modified: April 9th, 2011.

Contents

1. Introduction

olap4j is an open Java API for building OLAP applications.

In essence, olap4j is to multidimensional data what JDBC is for relational data. olap4j has a similar programming model to JDBC, shares some of its core classes, and has many of the same advantages. You can write an OLAP application in Java for one server (say Mondrian) and easily switch it to another (say Microsoft Analysis Services, accessed via XML for Analysis).

However, creating a standard OLAP API for Java is a contentious issue. To understand why, it helps to understand the history of OLAP standards.

1.1. A brief history of OLAP standards

History is strewn with attempts to create a standard OLAP API. First, the OLAP council's MDAPI (in two versions), then the JOLAP API emerged from Sun's Java Community Process. These all failed, it seems, because at some point during the committee stages, all of the OLAP server vendors concerned lost interest in releasing an implementation of the standard. The standards were large and complex, and no user-interface provider stepped forward with a UI which worked with multiple back-ends.

Meanwhile, Microsoft introduced OLE DB for OLAP (which works only between Windows clients and servers), and then XML/A (XML for Analysis, a web-services API). These standards were more successful, for a variety of reasons. First, since the standards (OLE DB for OLAP in particular) were mainly driven by one vendor, they were not a compromise attempting to encompass the functionality of several products. Second, there was a ready reference implementation, and Microsoft saw to it that there were sufficient OLAP clients to make these standards viable forums for competition and innovation. Third, there was the MDX query language. A query language is easier to explain than an API. It leaves unsolved the problem of how to construct queries to answer business questions, but application developers could solve that problem by embedding one of the off-the-shelf OLAP clients.

The Open Source community has been developing a taste for OLAP. First there was Mondrian, an open-source OLAP server; then there was JPivot, a client which first spoke to Mondrian, then also to XML/A; then there were more OLAP clients, and applications which wanted to use a particular client, but wanted to talk to a variety of servers; and companies using a particular OLAP server that wanted to get at it from several clients. It became clear the open-source OLAP tools needed a standard, and that standard would probably be suitable for other Java-based OLAP tools.

1.2. Overview of olap4j

An OLAP application interacts with an OLAP server by means of MDX statements belonging to connections. The statements are defined in terms of metadata and validated according to a type system, and some applications are built at a higher level, manipulating MDX parse trees, and defining complex queries in terms that a business user can understand. The olap4j API provides all of these facilities.

At the lowest level, olap4j has a framework for registering **drivers**, and managing the lifecycle of **connections and statements**. olap4j provides this support by extending the JDBC framework.

A key decision in the design of an OLAP API is whether to include a **query language**. Historically, it has been a contentious one. The previous standards fell into two camps: MDAPI and JOLAP had an API for building queries, while OLE DB for OLAP and XML/A had the MDX query language. The SQL query language is an essential component of relational database APIs such as ODBC and JDBC, and it makes similar sense to base an OLAP API on a query language such as MDX. But OLAP applications also need to **build and transform queries** as the end-user explores the data. So, olap4j embraces both approaches: you can create a query by parsing an MDX statement, you can build a query by manipulating an MDX parse tree, and an MDX parser library allows you to easily convert an MDX string to and from a parse tree.

Metadata is at the heart of olap4j. You can browse the cubes, dimensions, hierarchies, members in an OLAP schema, and an MDX parse tree and query result are tied back to the same metadata objects. There is also a **type system** for describing expressions.

olap4j makes it possible to write an OLAP client without starting from scratch. In addition to the MDX parser, and operations on the MDX parse tree, there is a higher-level query model, which includes **operations to transform queries** (also called 'navigations'), and facilities to layout multidimensional results as HTML tables.

There are experimental modules in olap4j for **scenarios** (also called 'what-if' analysis, or 'writeback') and **notifications**, pushed from the server to the client when the data set on the server changes.

1.3. Relationship to other standards

1.3.1. olap4j and XML/A

At this point, you may be saying: what about XML/A? XML/A was here first, is an open standard, and is supported by a number of servers. Is olap4j an attempt to replace XML/A? Isn't XML/A good enough for everyone?

olap4j certainly has some similarities with XML/A. Both APIs allow an application to execute OLAP queries, and to browse the metadata of an OLAP schema. But XML/A is a low-level web-services API which leaves a lot of work to the application writer. (Witness the fact that the majority of successful XML/A applications run only on Windows, where the ADOMD.NET is a high-level interface to XML/A servers.) The APIs are mostly complementary, because olap4j can be easily added to an XML/A back-end, and provides features which would be difficult or impossible to provide via a web-services API. These are functions for parsing MDX, building and transforming MDX query models, and mapping result sets into graphical layouts such as pivot tables.

If a web-services based application needs these functions, it can use the XML/A provider to connect to the underlying data source, execute queries, and browse metadata, but can still use olap4j's features for MDX parsing, query models and layout.

The metamodels of olap4j and XML/A are similar. Both contain schemas, cubes, dimensions, et cetera. Where possible, olap4j uses the same terminology for entity and attribute names. This simplifies the job of creating an XML/A driver for olap4j, and also makes possible an XML/A -to-olap4j bridge (a server that answers XML/A requests by querying an underlying olap4j data source).

1.3.2. olap4j is built on other standards

Where possible, olap4j leverages existing standards. This has several advantages. First, an end-user familiar with the existing standards can come up to speed with olap4j quickly. For instance, creating a connection and executing a statement should be straightforward to anyone familiar with JDBC connections, statements and result sets work.

If an OLAP server implementor has already implemented a driver for one standard, then it should be less work to implement an olap4j driver. This clearly applies to the MDX language (borrowed from XML/A and OLE DB for OLAP). Implementation schema result sets should be straightforward if the server already supports XML/A schema rowsets.

If olap4j is sufficiently similar to an existing standard, tools designed for use with that standard may be applicable to olap4j also. For instance, one goal of olap4j is that people will be able to use connection-pooling libraries such as Jakarta Commons DBCP, C3P0. (This presents some challenges because olap4j extends some of the JDBC interfaces, but we hope to solve them.)

Lastly, reusing an existing standard is less work for the authors of the new standard!

Sometimes the standards conflict. ADOMD exposes its metadata through an object model, whereas JDBC and XML/A expose metadata relationally, via what XML/A calls rowsets and what we and JDBC call result sets. In this case, we chose to do both, because of the diversity of needs of olap4j clients. Metadata objects allow you to integrate query results with metadata using much fewer code: positions can reference members, and you can navigate from a member to its hierarchy, and so forth. Likewise, metadata objects can be used in building MDX parse trees. But if a client tool wants to maintain its own metadata cache, schema rowsets are more flexible and efficient.

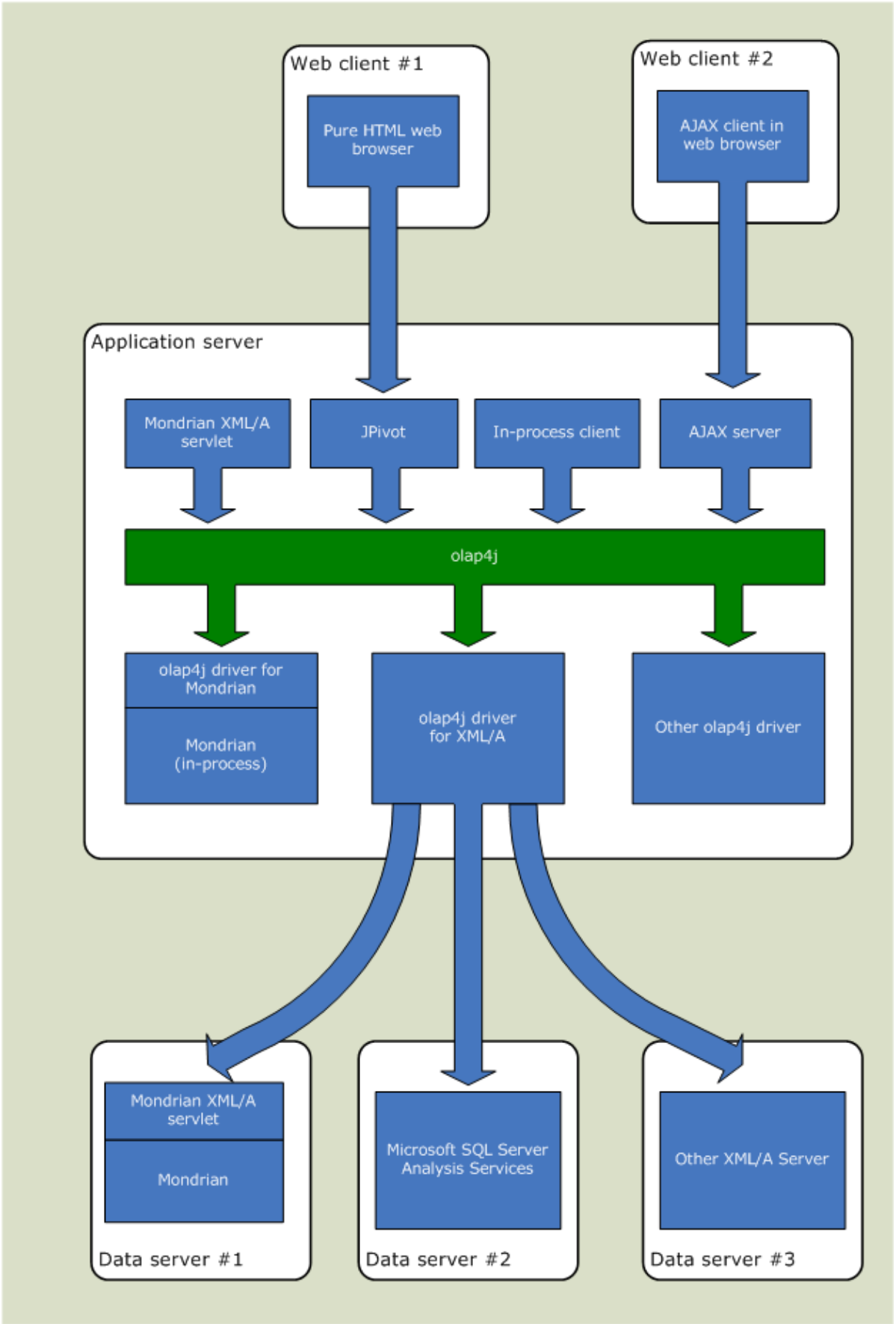
1.4. Benefits of a standard Java API for OLAP

Once the olap4j standard is in place, we can expect that the familiar benefits of an open standard will emerge: a larger variety of tools, better tools, and more price/feature competition between OLAP servers. These benefits follow because if a developer of OLAP tool can reach a larger audience, there is greater incentive to build new tools.

Eventually there will be olap4j providers for most OLAP servers. The server vendors will initially have little incentive to embrace a standard which will introduce competition into their market, but eventually the wealth of tools will compel them to write a provider; or, more likely, will tempt third-party or open-source efforts to build providers for their servers.

1.5. Architecture of olap4j

The following diagram shows how olap4j fits into an enterprise architecture.


Web client #1
Pure HTML web browser
Web client #2
AJAX client in web browser
Application server
Mondrian XML/A servlet
JPivot
In-process client
AJAX server
olap4j
olap4j driver for Mondrian
Mondrian (in-process)
olap4j driver for XML/A
Other olap4j driver
Mondrian XML/A servlet
Mondrian
Microsoft SQL Server Analysis Services
Other XML/A Server
Data server #1
Data server #2
Data server #3

1.6. Compatibility

olap4j requires JDK 1.5 or higher, in particular because it uses the generics and enum features introduced in JDK 1.5.0.

olap4j works best against JDK 1.6 or higher, because it makes use of the java.sql.Wrapper interface. In earlier Java versions, the same functionality is available by casting objects to the built-in OlapWrapper interface.

JDK 1.4 compatibility will be available on demand, using the Retroweaver utility. This will consist of a retrowoven JAR file, `olap4j-jdk1.4.jar` and retroweaver's runtime library `retroweaver-rt-1.2.4.jar`. (See design note.)

olap4j's JDBC support is consistent with JDBC version 3.0 (which was introduced in JDK 1.4 and is also in JDK 1.5) and also with JDBC version 4.0 (introduced in JDK 1.6).

1.7. Compliance levels

There are two compliance levels to which a driver can support olap4j. olap4j is a rich specification, and it takes a considerable effort to implement it fully, but applications can still be built on a driver which only implements the core parts of the specification. The lower level allows a vendor to a subset of the specification without the significant investment of a fully-compliant driver.

The compliance levels are as follows:

- **olap4j Core Compliance**. To comply with the olap4j core specification, it must implement all classes in the `org.olap4j` package, with the exception of the `get`*`Xxxx`*`s()` methods in OlapDatabaseMetaData that return result sets, and all classes in the `org.olap4j.metadata` package. The OLAP server must also implement the fundamental features of the MDX language (see below).
- **olap4j Full Compliance**. To fully comply with the olap4j specification, a driver must comply with the core requirements, plus implement the OlapDatabaseMetaData.`get`*`Xxxx`*`s()` methods, plus implements an MDX parser and validator (`org.olap4j.mdx.parser` package).

MDX language compliance

All olap4j drivers must implement the fundamental features of the MDX language, but olap4j compliance levels do not imply a particular degree of support for the MDX language.

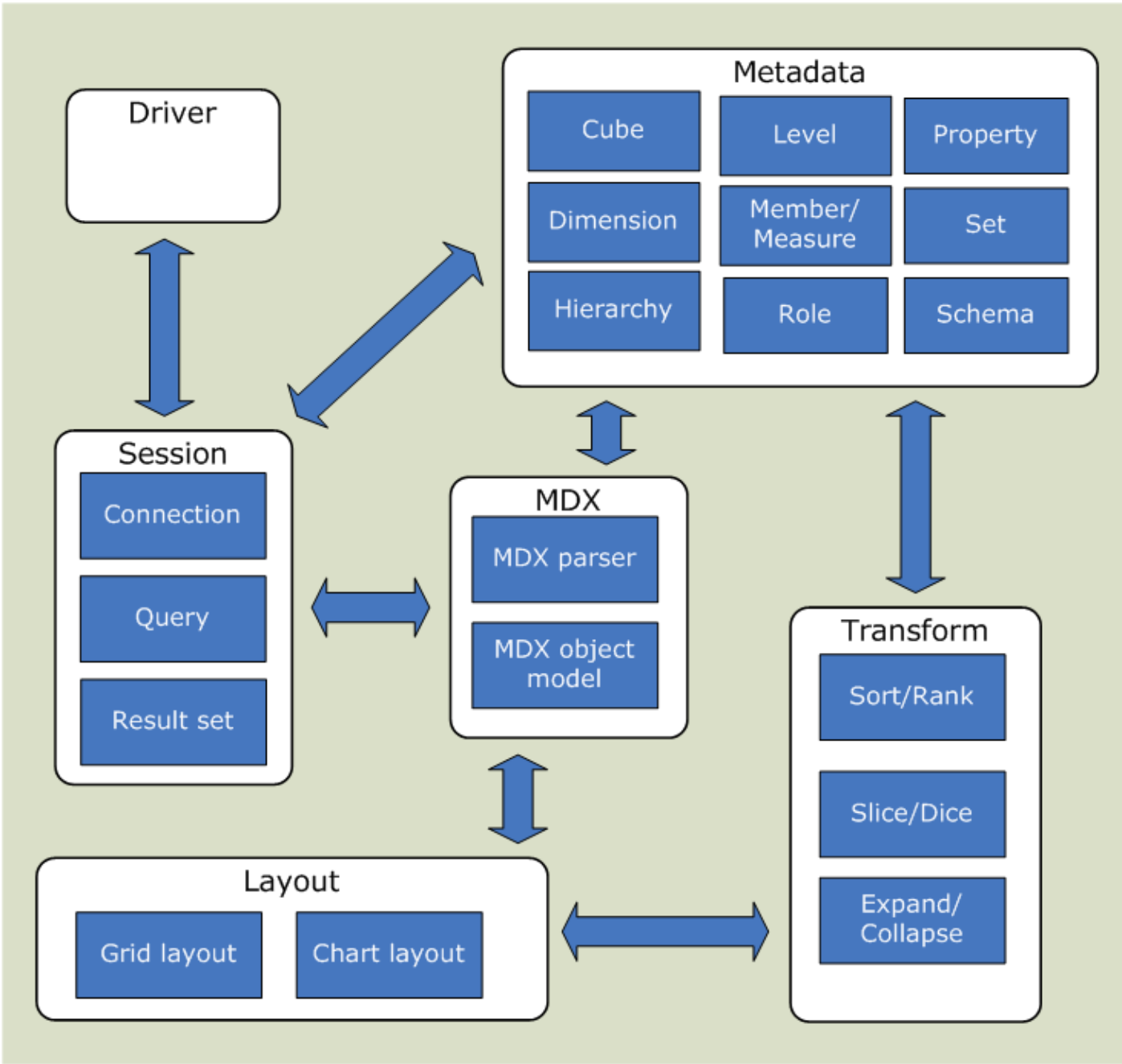
The fundamental features of MDX are:

- Queries of the form "SELECT ... FROM ... WHERE".
- The set constructor operator { ... }.
- Set functions CrossJoin(<Set>, <Set>), Filter(<Set>, <Condition>), Order(<Set>, <Expression>), Hierarchize(<Set>).
- Navigation operators <Member>.Children, <Level>.Members <Hierarchy>.Members, <Member>.Parent, Level, Hierarchy, Dimension
- Aggregation functions Aggregate
- Basic arithmetic and logical operators

Of course, a provider may implement a larger subset of MDX, and most do. A provider can describe its MDX compliance level by describing the additional features it supports (e.g. WITH MEMBER, WITH SET, NON EMPTY, and HAVING clauses in queries; virtual cubes; and calculated members and sets defined against cubes) and additional functions and operators implemented.

2. Components of the API

We now describe the olap4j API in more detail, by breaking it down into a set of functional areas.



2.1. Driver management

olap4j shares JDBC's driver management facilities. This allows olap4j clients to leverage the support for JDBC such as connection pooling, driver registration.

Classes:

- java.sql.Driver
- java.sql.DriverManager
- javax.sql.DataSource

2.1.1. The Driver class

Same functionality as JDBC.

Here is an example of registering an olap4j driver:

Class.forName("mondrian.olap4j.MondrianOlap4jDriver");
Note that this is the same as you would write for any JDBC driver. From JDBC 4.0 (JDK 1.6) onwards, compliant JDBC drivers register themselves automatically by creating an entry in their JAR file's `META-INF/services/java.sql.Driver` file. Compliant olap4j drivers register themselves in the same way.

2.1.2. The DriverManager class

Same functionality as JDBC.

2.1.3. The DataSource interface

Same functionality as JDBC.

See also: Database.

2.1.4. The OlapDataSource interface

Extension to `DataSource` that returns `OlapConnection` objects rather than mere `java.sql.Connection` objects.

2.1.5. The OlapException class

OlapException (extends java.sql.SQLException) describes an error which occurred while accessing an OLAP server.

Since olap4j extends JDBC, it is natural that OlapException should extend JDBC's SQLException. The implementation by an olap4j driver of a JDBC method which is declared to throw a SQLException may, if the driver chooses, throw instead an OlapException.

OlapException provides some additional information to help an OLAP client identify the location of the error. The `context` is the Cell or Position object where the error occurred. The `region` is an object representing the textual region in the MDX statement.

Methods:

- `Region getRegion()`
- `setRegion(Region region)`
- `Object getContext()`
- `void setContext(Object context)`

2.2. Connections

olap4j's connection management component manages connections to the OLAP server, statements.

Where possible, olap4j uses JDBC's session management facility. olap4j defines extensions to JDBC interfaces Connection and Statement.

For example, the following code registers a driver, connects to Mondrian and executes a statement:

```
import java.sql.*;
import org.olap4j.*;

Class.forName("mondrian.olap4j.MondrianOlap4jDriver");
Connection connection =
   DriverManager.getConnection(
      "jdbc:mondrian:Jdbc=jdbc:odbc:MondrianFoodMart;"
      + "Catalog=/WEB-INF/queries/FoodMart.xml;"
      + "Role='California manager'");
OlapWrapper wrapper = (OlapWrapper) connection;
OlapConnection olapConnection = wrapper.unwrap(OlapConnection.class);
OlapStatement statement = olapConnection.createStatement();

CellSet cellSet =
   statement.executeOlapQuery(
      "SELECT {[Measures].[Unit Sales]} ON COLUMNS,\n"
      + "  {[Product].Members} ON ROWS\n"
      + "FROM [Sales]");
```

Here's a piece of code to connect to Microsoft SQL Server Analysis Services™ (MSAS) via XML/A. Note that except for the driver class and connect string, the code is identical.

```
import java.sql.*;
import org.olap4j.*;

Class.forName("org.olap4j.driver.xmla.XmlaOlap4jDriver");
Connection connection =
   DriverManager.getConnection(
      "jdbc:xmla:Server=http://localhost/xmla/msxisapi.dll;"
      + "Catalog=FoodMart");
OlapWrapper wrapper = (OlapWrapper) connection;
OlapConnection olapConnection = wrapper.unwrap(OlapConnection.class);
OlapStatement statement = connection.createStatement();

CellSet cellSet =
```

```
    statement.executeOlapQuery(
        "SELECT {[Measures].[Unit Sales]} ON COLUMNS,\n"
        + "  {[Product].Members} ON ROWS\n"
        + "FROM [Sales]");
```

In the above examples, a statement was created from a string. As we shall see, a statement can also be created from an MDX parse tree.

2.2.1. Connection pooling

Look again at the code samples in the previous section. One would expect that it would be safe to downcast the result of a factory method to the desired result. For example, if you invoke an `OlapConnection`'s `createStatement()` method, the result should be an `OlapStatement`.

But if you you are using a connection-pooling library (common examples of which include Jakarta Commons DBCP and C3P0), this is not so. Every connection-pooling library tracks connections by wrapping them in another class, and this class will implement `java.sql.Connection` but not `OlapConnection`. To access methods of the `OlapConnection`, the client application must first strip away the wrapper object.

If you are using a connection-pooling library, olap4j provides the OlapWrapper interface with the method method `unwrap(Class)` to access the object underneath the wrapped connection. For instance, if you were using DBCP, you could define and use a pooling olap4j data source as follows:

```
import java.sql.*;
import org.olap4j.*;
import org.apache.commons.dbcp.*;

GenericObjectPool connectionPool =
    new GenericObjectPool(null);
ConnectionFactory connectionFactory =
    new DriverManagerConnectionFactory(
        "jdbc:mondrian:Jdbc=jdbc:odbc:MondrianFoodMart;"
        + "Catalog=/WEB-INF/queries/FoodMart.xml;"
        + "Role='California manager'",
        new Properties());
PoolableConnectionFactory poolableConnectionFactory =
    new PoolableConnectionFactory(
        connectionFactory, connectionPool, null, null, false, true);
DataSource dataSource =
    new PoolingDataSource(connectionPool);

// and some time later...

Connection connection = dataSource.getConnection();
OlapWrapper wrapper = (OlapWrapper) connection;
OlapConnection olapConnection = wrapper.unwrap(OlapConnection.class);
OlapStatement statement = olapConnection.createStatement();
```

The OlapStatement, PreparedOlapStatement, and CellSet interfaces also extend `OlapWrapper`, and can be accessed similarly.

If connection pooling is not being used, then the object returned by the driver will be an `OlapConnection` and will therefore trivially implement `OlapWrapper` (because the `OlapConnection` interface extends

`OlapWrapper`). If connection pooling is being used, the code will work provided that the implementer of the connection pool has ensured that the pooled connection object implements the `OlapWrapper` interface. This is a minor change to the connection pool, and we hope that popular connection pools will utilize this method in the near future.

If you are using JDBC 4.0 (which is part of JDK 1.6 and later), the java.sql.Connection class implements the java.sql.Wrapper interface introduced in JDBC 4.0, so the code can be simplified:

Connection connection = DriverManager.getConnection();
OlapConnection olapConnection = connection.unwrap(OlapConnection.class);
OlapStatement statement = olapConnection.createStatement();
Note that the `OlapWrapper` interface is not needed. This code will work with any JDBC 4.0-compliant connection pool.

Package name: org.olap4j

2.2.2. The OlapConnection interface

OlapConnection (extends java.sql.Connection) is a connection to an OLAP data source.

Methods:

- `String getCatalog()` // returns the name of the current catalog (inherited from Connection)
- `Catalog getOlapCatalog()` // returns the current catalog object
- `NamedList<Catalog> getOlapCatalogs()` // returns a list of all catalogs
- `Database getOlapDatabase()` // returns the current database
- `NamedList<Database> getOlapDatabases()` // returns a list of all databases
- `String getSchema()` // returns the name of the current schema
- `Schema getOlapSchema()` // returns the current schema object
- `NamedList<Schema> getOlapSchema()` // returns a list of all schemas
- `OlapDatabaseMetaData getMetaData()` // returns an object that contains metadata about the database
- `void setLocale(Locale locale)` // sets this connection's locale
- `Locale getLocale()` // returns this connection's locale
- `void setRoleName(String roleName)` // sets the name of the role in which access-control context this connection will execute queries
- `String getRoleName()` // returns the name of the role in which access-control context this connection will execute queries
- `PreparedOlapStatement prepareOlapStatement(String mdx)` // prepares a statement
- `createStatement createStatement(String mdx)` // creates a statement (overrides Connection method)
- `List<String> getAvailableRoleNames()` // returns a list of role names available in this connection
- `Scenario createScenario()` // creates a scenario
- `void setScenario(Scenario scenario)` // sets the current scenario for this connection
- `Scenario getScenario()` // returns the current scenario for this connection
- `MdxParserFactory getParserFactory()` // returns a factory to create MDX parsers

2.2.3. The OlapWrapper interface

OlapWrapper provides the ability to retrieve a delegate instance when the instance in question is in fact a proxy class.

`OlapWrapper` duplicates the functionality of the java.sql.Wrapper interface (introduced in JDBC 4.0), making this functionality available to olap4j clients running in a JDBC 3.0 environment. For code which will run only on JDBC 4.0 and later, `Wrapper` can be used, and `OlapWrapper` can be ignored.

Methods:

- `boolean isWrapperFor(Class<?> iface)` // returns true if this either implements the interface argument or is directly or indirectly a wrapper for an object that does
- `<T> T unwrap(Class<T>)` // returns an object that implements the given interface

2.2.4. The OlapDatabaseMetaData interface

OlapDatabaseMetaData (extends java.sql.DatabaseMetaData) provides information about an OLAP database.

Just as DatabaseMetaData provides a method to query the each kind of metadata element in a relational database (tables, columns, and so forth), returning the rows as a ResultSet, `OlapDatabaseMetaData` provides methods for OLAP metadata elements (cubes, dimensions, hierarchies, levels, members, measures).

These methods are described in the section "The `OlapDatabaseMetaData` interface and methods which return schema rowsets".

2.3. Statements

2.3.1. The OlapStatement interface

OlapStatement (extends java.sql.Statement) is an object used to execute a static MDX statement and return the result it produces.

It has methods to execute an MDX query represented both as a string and as a parse tree.

Methods:

- `CellSet executeOlapQuery(String mdx)` // executes an MDX statement
- `CellSet executeOlapQuery(SelectNode selectNode)` // executes an MDX statement expressed as a parse tree
- `OlapConnection getConnection()` // returns the current connection (overrides Statement method)
- `addListener(Granularity, CellSetListener)` // adds a listener to be notified of events to CellSets created by this statement

2.3.2. The PreparedOlapStatement interface

PreparedOlapStatement (extends java.sql.PreparedStatement) represents a precompiled MDX statement.

An MDX statement is precompiled and stored in a `PreparedOlapStatement` object. This object can

then be used to efficiently execute this statement multiple times.

The method PreparedStatement.getParameterMetaData() returns a description of the parameters, as in JDBC. The result is an OlapParameterMetaData.

To set values of parameters, use the `set`*`Type`*`(int, `*`type`*`)` methods. If a parameter is a member, use the `setObject(int, Object)` method; throws an exception if the object is not a member, or is a member of the wrong hierarchy.

Unlike JDBC, it is not necessary to assign a value to every parameter. This is because OLAP parameters have a default value. Parameters have their default value until they are set, and then retain their new values for each subsequent execution of this `PreparedOlapStatement`.

The getCube() method returns the cube (or virtual cube) the prepared statement relates to.

Methods:

- `CellSet executeQuery()`
- `Cube getCube()`
- `CellSetMetaData getMetaData()`
- `OlapParameterMetaData getParameterMetaData()`

2.3.3. The OlapParameterMetaData interface

OlapParameterMetaData (extends java.sql.ParameterMetaData) describes parameters of a `PreparedOlapStatement`.

Additional methods:

- `getName()`
- `getOlapType(int param)`

2.3.4. The CellSet interface

CellSet (extends java.sql.ResultSet) is the result of executing an OlapStatement or PreparedOlapStatement.

It extends `ResultSet`, but since most of these methods are concerned with rows and columns, only a few of the base class's methods are applicable. The following methods are applicable:

- clearWarnings()
- close()
- getConcurrency()
- getStatement()
- getType()
- getWarnings()

Additional methods to retrieve the axes of the multidimensional result:

- `List<CellSetAxis> getAxes()`
- `CellSetAxis getFilterAxis()`
- `Cell getCell(List<Integer> coordinates)`

An OlapStatement can have no more than one CellSet open. Closing an OlapStatement, or preparing or executing a new query, implicitly closes any previous CellSet.

2.3.5. The CellSetAxis interface

A CellSetAxis is an axis belonging to a CellSet.

A cell set has the same number of axes as the MDX statement which was executed to produce it. For example, a typical cell set, resulting from an MDX query with COLUMNS and ROWS expressions is two-dimensional, and therefore has two axes.

Each axis is an ordered collection of members or tuples. Each member or tuple on an axis is called a Position.

The positions on the cell set axis can be accessed sequentially or random-access. Use the `List<Position> getPositions()` method to return a list for random access, or the `Iterator<Position> iterate()` method to obtain an iterator for sequential access.

Methods:

- `Axis getOrdinal()`
- `CellSet getCellSet()`
- `CellSetAxisMetaData getAxisMetaData()`
- `List<Position> getPositions()`
- `int getPositionCount()`
- `ListIterator<Position> iterate()`

2.3.6. The Axis enum

Axis is an enumeration of axis types.

2.3.7. The CellSetAxisMetaData interface

A CellSetAxisMetaData is describes a CellSetAxis.

Methods:

- `Axis getAxis()`
- `List<Hierarchy> getHierarchies()`
- `List<Property> getProperties()`

2.3.8. The Position interface

Position is a position on a CellSetAxis.

An axis has a particular dimensionality, that is, a set of one or more dimensions which will appear on that axis, and every position on that axis will have a member of each of those dimensions. For example, in the MDX query

```
SELECT {[Measures].[Unit Sales], [Measures].[Store Sales]} ON COLUMNS,
  CrossJoin(
    {[Gender].Members},
```

{[Product].[Food], [Product].[Drink]}) ON ROWS
FROM [Sales]
the `COLUMNS` axis has dimensionality {`[Measures]`} and the `ROWS` axis has dimensionality {`[Gender]`, `[Product]`}. In the result of this query,

Gender	*Product*	Unit Sales	Store Sales
All Gender	**Food**	191,940	409,035.59
All Gender	**Drink**	24,597	48,836.21
F	**Food**	94,814	203,094.17
F	**Drink**	12,202	24,457.37
M	**Food**	97,126	205,941.42
M	**Drink**	12,395	24,378.84

each of the 5 positions on the `ROWS` axis has two members, consistent with its dimensionality of 2. The `COLUMNS` axis has two positions, each with one member.

Methods:

- `List<Member> getMembers()`
- `int getOrdinal()`

2.3.9. The Cell interface

A Cell is a cell returned from an CellSet.

Methods:

- `CellSet getCellSet()`
- `int getOrdinal()`
- `List<Integer> getCoordinateList()`
- `Object getPropertyValue(Property)`
- `boolean isError()`
- `boolean isNull()`
- `boolean isEmpty()`
- `double getDoubleValue()`
- `String getErrorText()`
- `Object getValue()`
- `String getFormattedValue()`

2.3.10. The CellSetMetaData interface

CellSetMetaData (extends java.sql.ResultSetMetaData) describes a CellSet.

Methods:

- `NamedList<Property> getCellProperties()`
- `Cube getCube()`
- `NamedList<CellSetAxisMetaData> getAxesMetaData()`

- `CellSetAxisMetaData getSlicerAxisMetaData()`

2.4. MDX query model

The MDX query model represents a parsed MDX statement.

An MDX query model can be created in three ways:

- The MDX parser parses an MDX string to create an MDX query model;
- Client code programmatically builds a query model by calling API methods;
- Code in the transform package manipulates query model in response to graphical operations.

An MDX query model can exist in an *un-validated* and *validated* state. In the un-validated state, identifiers and function calls exist as raw strings, and no type information has been assigned. During validation, identifiers are resolved to specific MDX objects (members, etc.), type information is assigned, and if a function exists in several overloaded forms, a specific instance is chosen based upon the types of its arguments.

Any MDX query model can be serialized to a string containing MDX text.

An MDX query model can be converted into a statement. For example,

```
import org.olap.*;
import org.olap4j.mdx.*;

// Create a query model.
OlapConnection connection;
SelectNode query = new SelectNode();
query.setFrom(
   new IdentifierNode(
      new IdentifierNode.NameSegment("Sales")));
query.getAxisList().add(
   new AxisNode(
      null,
      false,
      Axis.ROWS,
      new ArrayList<IdentifierNode>(),
      new CallNode(
         null,
         "{}",
         Syntax.Braces,
         new IdentifierNode(
            new IdentifierNode.NameSegment("Measures"),
            new IdentifierNode.NameSegment("Unit Sales")))));

// Create a statement based upon the query model.
OlapStatement stmt;
try {
   stmt = connection.createStatement();
} catch (OlapException e) {
```

```
    System.out.println("Validation failed: " + e);
    return;
}

// Execute the statement.
CellSet cset;
try {
    cset = stmt.executeOlapQuery(query);
} catch (OlapException e) {
    System.out.println("Execution failed: " + e);
}
```

Package name: org.olap4j.mdx

Parse tree classes:

- ParseTreeNode is a node in a parse tree representing a parsed MDX statement.
- SelectNode represents a `SELECT` statement, including `FROM` and `WHERE` clauses if present.
- AxisNode represents an axis expression.
- CallNode represents a call to a function or operator.
- IdentifierNode represents an identifier, such as `Sales` or `[Measures].[Unit Sales]`.
- LiteralNode represents a literal, such as `123` or `"Hello, world!"`.
- MemberNode represents a use of a member name in an expression.
- LevelNode represents a use of a level name in an expression.
- HierarchyNode represents a use of a hierarchy name in an expression.
- DimensionNode represents a use of a dimension name in an expression.
- WithMemberNode represents a `WITH MEMBER` clause defining a calculated member.
- WithSetNode represents a `WITH SET` clause defining a calculated set.
- PropertyValueNode represents *property* = *value* pair as part of a the declaration of a calculated member or set.

Other classes:

- enum Syntax describes the possible syntaxes for functions and operators (infix, prefix, function call, and so forth)

2.4.1 The ParseTreeWriter class

ParseTreeWriter is used in conjunction with the ParseTreeNode.unparse(ParseTreeWriter) method to convert a parse tree into MDX code.

2.5. MDX parser

Package name: org.olap4j.mdx.parser

Provides an MDX parser and validator.

Parser and validator are both allocated via a parser factory, which is obtained from a connection:

```
OlapConnection connection;
MdxParserFactory parserFactory =
```

```
    connection.getParserFactory();
MdxParser parser =
    parserFactory.createMdxParser(connection);
SelectNode select =
    parser.parseSelect("SELECT FROM [Sales]");
MdxValidator validator =
    parserFactory.createMdxValidator(connection);
select = validator.validate(select);
```

Parser and validator are not thread-safe (they cannot be used by more than one thread simultaneously) but they can be re-used for multiple statements.

One of the chief purposes of validation is to assign a type to every expression within the parse tree. Before validation, any node's ParseTreeNode.getType() method may throw an exception, but after validation the `getType()` method will return a type. Nodes which are not expressions do not have types, and will always return `null`.

Classes:

- MdxParserFactory
- MdxParser
- MdxValidator

2.6. MDX type model

Package name: org.olap4j.type

Represents the MDX type system.

Here are some examples:

Expression	Type
`1 + 2`	Integer
`[Store]`	Dimension
`[Store].[State]`	Level<dimension=[Store], hierarchy=[Store]>
`[Store].[USA].[CA]`	Member<dimension=[Store], hierarchy=[Store], level=[Store].[State], member=[Store].[USA].[CA]>
`[Store].[USA].Children(2)`	Member<dimension=[Store], hierarchy=[Store], level=[Store].[State]>

Since MDX is a late-binding language, some expressions will have unknown types, or only partial type information. For example, the expression

```
[Store].Levels("Sta" + "te")
```

will have type `Level<dimension=[Store], level=unknown>`. The validator knows that the `<hierarchy>.Levels(<string expr>)` function returns a level, but exactly which level is not known until the expression is evaluated at runtime.

Type is the base class for all types.

Scalar types:

- ScalarType represents the type of an expression which has a simple value such as a number or a string.
- BooleanType (extends ScalarType) represents an expression which can have values TRUE and FALSE.
- NumericType represents the type of a numeric expression.
- DecimalType (extends `NumericType`) represents a fixed-point numeric expression. It is a subclass of `NumericType`, and has precision and scale. An integer expression would have scale 0.
- StringType (extends ScalarType) represents the type of an expression which has a string value.
- SymbolType (extends ScalarType) represents the type of a symbol, or flag, argument to a built-in function. For example, the `ASC` keyword in the expression `Order(Gender.MEMBERS, Measures.[Unit Sales], ASC)` is a symbol. Symbol types are rarely used except if you are manipulating a parse tree.

Metadata types:

- CubeType represents the type of an expression whose value is a cube.
- DimensionType represents the type of an expression whose value is a dimension.
- HierarchyType represents the type of an expression whose value is a hierarchy.
- LevelType represents the type of an expression whose value is a level.
- MemberType represents the type of an expression whose value is a member.

A metadata type may be constrained to a particular part of the schema. For example, `LevelType(hierarchy=[Time])` indicates that the expression must evaluate to one of the levels of the `[Time]` hierarchy, that is, one of the values `[Time].[Year]`, `[Time].[Quarter]`, or `[Time].[Month]`.

Composite types:

- SetType represents the type of an expression which is a set. It has a component type, for example, the type of the expression `{[Store].[USA].Children}` is `Set(Member(level=[Store].[Store State])`.
- TupleType represents the type of an expression which consists of an n-tuple of members. It has a set of component types, each of which is a member type. For example, the type of the expression `CrossJoin({[Gender].[F], [Gender].[M]}, [Store].Members)` is `Set(Tuple(Member(level=[Gender].[Gender]), Member(hierarchy=[Store]))`.

2.7. Metadata

Package name: org.olap4j.metadata

Metadata are the objects which describe the structure of an OLAP schema: cubes, dimensions, members, properties and so forth.

olap4j exposes metadata in two very different ways:

- A metadata object is a Java object which represents a particular metadata class. For example, org.olap4j.metadata.Cube.
- A schema result set is a JDBC ResultSet which returns a record for each instance of a particular metadata class. There is a method in the OlapDatabaseMetaData interface to create a schema result

set for each metadata class. Some of these methods accept parameters to filter the rows returned. For example, OlapDatabaseMetaData.getCubes(String catalog, String schemaNamePattern, String cubeNamePattern).

2.7.1. Access control

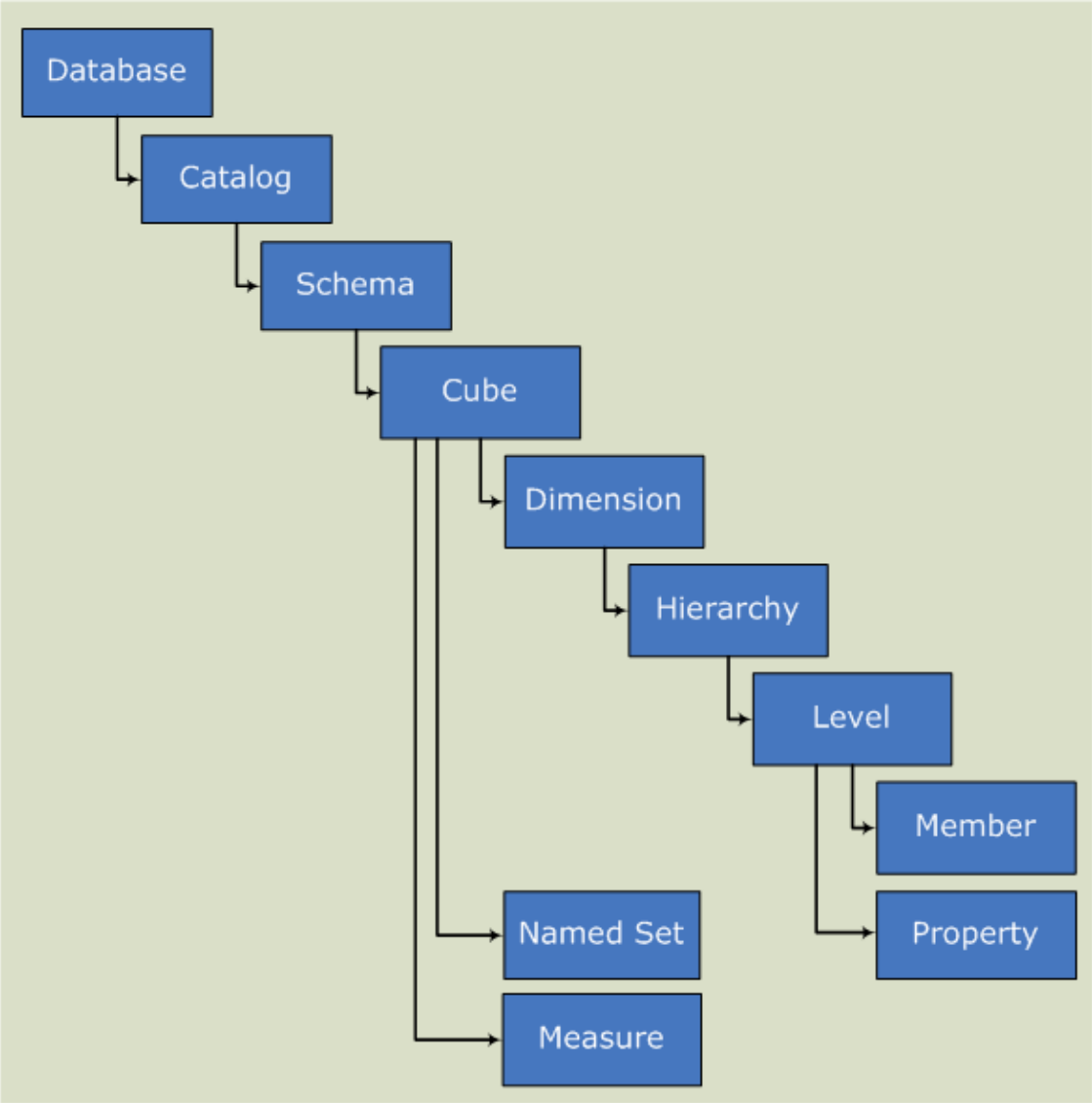
A user's view of metadata may be subject to access control. The precise rules for access control depend on the provider, and this specification does not say what those rules should be. But this specification requires that the API methods must behave consistently with the server's access control policy.

For example, in mondrian, users belong to roles, and roles may be granted or denied access to cubes, hierarchies, or members within hierarchies. Suppose that user Fred belongs to the "Sales Manager" role, which does not have access to the `[Nation]` level of the `[Store]` hierarchy, and the current connection has been opened in the "Sales Manager" role. Then the `Member.getParentMember()` method will return null if applied to `[Store].[USA].[CA]`, because the 'real' parent member `[Store].[USA]` is invisible to him; also, the `Hierarchy.getLevels()` and `OlapDatabaseMetaData.getLevels()` methods will omit the Nation level from the list of levels they return.

In olap4j, you can set a connection's role at connect time using the `Role` connect string property, or you can call the `OlapConnection.setRole(String roleName)` method at any point during the lifecycle of the connection. Setting the role name to `null` reverts to the default access-control context.

2.7.2. Metadata objects

The following diagram shows the metadata objects in an olap4j schema.



In the diagram, each arrow represents a collection of objects in a parent object; for example, a database is a collection of catalogs, each catalog is a collection of schemas, each schema is a collection of cubes, and so forth. Each object has a corresponding class in the org.olap4j.metadata package.

Most metadata objects extend the MetadataElement interface, which gives them `name` and `uniqueName` attributes, and localized `caption` and `description`.

When the API returns a list of metadata elements whose names must be unique (for example, the list of dimensions in a cube), the return type is the NamedList extension to java.util.List.

Providers are at liberty to implement metadata objects using a cache, and therefore over the course of time, different java objects may represent the same underlying metadata object. Always use equals(), not the == operator, when comparing metadata objects, and do not use IdentityHashMap.

2.7.2.1. The MetadataElement interface

A MetadataElement is an element which describes the structure of an OLAP schema.

Subtypes are Cube, Dimension, Hierarchy, Level, Member, Property. MetadataElement provides name and unique-name properties (not localized), and localized caption and description (see Internationalization).

- `String getName()` // name of this metadata element
- `String getUniqueName()` // unique name of this metadata element
- `String getCaption()` // localized caption of this metadata element
- `String getDescription()` // localized description of this metadata element

2.7.2.2. The Database interface

A Database is the highest level element in the hierarchy of metadata objects. A database contains one or more catalogs.

Some OLAP servers may only have one database. Mondrian is one such OLAP server.

To obtain the collection of databases in the current server, call the `OlapConnection.getDatabases()` method.

Methods:

- `OlapConnection()` // returns the current connection
- `String getName()` // returns the name of this database
- `NamedList<Catalog> getCatalogs()` // returns a list of catalogs in this database
- `String getURL()` // returns the redirection URL, if this database is a proxy to another server
- `String getDataSourceInfo()` // returns provider-specific information
- `String getProviderName()` // returns the name of the underlying OLAP provider
- `List<ProviderType> getProviderTypes()` // returns the types of data that are supported by this provider

2.7.2.3. The Catalog interface

A Catalog is the highest level element in the hierarchy of metadata objects. A catalog contains one or more schemas.

Some OLAP servers may only have one catalog. Mondrian is one such OLAP server; its sole catalog is always called "LOCALDB".

To obtain the collection of catalogs in the current server, call the `OlapConnection.getCatalogs()` method.

Methods:

- `String getName()` // returns the name of this catalog
- `NamedList<Schema> getSchemas()` // returns a list of schemas in this catalog
- `OlapDatabaseMetaData getMetaData()` // returns the metadata describing the OLAP server that this catalog belongs to
- `Database getDatabase()` // returns this catalog's parent database

2.7.2.4. The Schema interface

A Schema is a collection of database objects that contain structural information, or metadata, about a database.

It belongs to a catalog and contains a number of cubes and shared dimensions.

- `Catalog geCatalog()` // returns this schema's parent catalog
- `String getName()` // returns the name of this catalog
- `NamedList<Dimension> getSharedDimensions()`
- `NamedList<Cube> getCubes()`
- `Collection<Locale> getSupportedLocales()` (see Internationalization)

2.7.2.5. The Cube interface

A Cube is the central metadata object for representing multidimensional data.

It belongs to a schema, and is described by a list of dimensions and a list of measures. It may also have a collection of named sets, each defined by a formula.

- `NamedList<Dimension> getDimensions()`
- `NamedList<Hierarchy> getHierarchies()`
- `List<Measure> getMeasures()`
- `NamedList<NamedSet> getSets()`
- `Schema getSchema()`
- `String getName()`
- `List<Locale> getSupportedLocales()` (see Internationalization)
- `Member lookupMember(List<IdentifierSegment> nameParts)`
- `List<Member> lookupMembers(Set<TreeOp> treeOps, List<IdentifierSegment> nameParts)`

2.7.2.6. The Dimension interface

A Dimension (extends MetadataElement) is an organized hierarchy of categories, known as levels, that describes data in a cube.

Dimensions typically describe a similar set of members upon which the user wants to base an analysis.

A dimension must have at least one hierarchy, and may have more than once, but most have exactly one hierarchy.

- `String getName()`
- `NamedList<Hierarchy> getHierarchies()`
- `Dimension.Type getDimensionType()`

2.7.2.7. The Hierarchy interface

A Hierarchy (extends MetadataElement) is an organization of the set of members in a dimension and their positions relative to one another.

A hierarchy is a collection of levels, each of which is a category of similar members.

- `Dimension getDimension()`
- `String getName()`
- `NamedList<Level> getLevels()`
- `boolean hasAll()`
- `Member getDefaultMember()`
- `NamedList<Member> getRootMembers()`

2.7.2.8. The Level interface

A Level (extends MetadataElement) is a group of members in a hierarchy, all with the same attributes and at the same depth in the hierarchy.

- `int getDepth()`
- `Hierarchy getHierarchy()`
- `Level.Type getLevelType()`
- `NamedList<Property> getProperties()`
- `List<Member> getMembers()`
- `int getCardinality()`

2.7.2.9. The Member interface

A Member (extends MetadataElement) is a data value in an OLAP dimension.

- `String getName()`
- `NamedList<Member> getChildMembers()`
- `Member getParentMember()`
- `Level getLevel()`
- `Hierarchy getHierarchy()`
- `boolean isAll()`
- `boolean isChildOrEqualTo(Member member)`
- `boolean isCalculated()`
- `boolean isCalculatedInQuery()`
- `int solveOrder()`
- `List<Member> getAncestorMembers()`
- `Object getPropertyValue(Property property)`
- `String getPropertyFormattedValue(Property property)`
- `void setProperty(Property property, Object value)`
- `NamedList<Property> getProperties()`
- `int getOrdinal()`
- `boolean isHidden()`
- `Member getDataMember()`
- `int getChildMemberCount()`

2.7.2.10. The Measure interface

A Measure (extends Member) is a data value of primary interest to the user browsing the cube. It provides the value of each cell, and is usually numeric.

Every measure is a member of a special dimension called "Measures".

- `boolean isVisible()`

- `Aggregator getAggregator()`
- `Datatype getDataType()`

2.7.2.11. The Property interface

Property (extends MetadataElement) is the definition of a property of a member or a cell.

`Property` contains two enumerated types StandardMemberProperty and StandardCellProperty whose values are the built-in properties of members and cells. Because these types implement the Property interface, you can use them as properties; for example:

```
Member member;
Object o = member.getPropertyValue(
   Property.StandardMemberProperty.CATALOG_NAME);
```

Members:

- `Datatype getDatatype()`
- `Set<TypeFlag> getType()`
- `ContentType getContentType()`
- `enum TypeFlag { MEMBER, CELL, SYSTEM, BLOB }`
- `enum StandardMemberProperty implements Property { CATALOG_NAME, SCHEMA_NAME, CUBE_NAME, ... }`
- `enum StandardCellProperty implements Property { BACK_COLOR, CELL_EVALUATION_LIST, ... }`
- `enum ContentType { REGULAR, ID, RELATION_TO_PARENT, ... }`

2.7.2.12. The NamedSet interface

A NamedSet (extends MetadataElement) describes a set whose value is determined by an MDX expression. It belongs to a cube.

- `Cube getCube()`
- `Expression getExpression()`

2.7.2.13. The Datatype enum

The Datatype enum describes the type of property and measure values. Because olap4j drivers need to interoperate with OLE DB for OLAP and XMLA systems, Datatype values have the same ordinals as in the OLE DB specification, and we show here the name and description of the corresponding type in the OLE DB specification. The table shows the analogous Java type, if there is one.

Datatype	Java type	OLE DB type	Description
INTEGER	int	DBTYPE_I4	A four-byte, signed integer: INTEGER
DOUBLE	double	DBTYPE_R8	A double-precision floating-point value: Double
CURRENCY		DBTYPE_CY	A currency value: LARGE_INTEGER, Currency is a fixed-point number with four digits to the right of the decimal point. It is stored in an eight-byte signed integer, scaled by 10,000.
BOOLEAN	boolean	DBTYPE_BOOL	A Boolean value stored in the same way as in Automation: VARIANT_BOOL; 0 means false

			and ~0 (bitwise, the value is not 0; that is, all bits are set to 1) means true.
VARIANT	Object	DBTYPE_VARIANT	An Automation VARIANT
UNSIGNED_SHORT	-	DBTYPE_UI2	A two-byte, unsigned integer
UNSIGNED_INTEGER	-	DBTYPE_UI4	A four-byte, unsigned integer
LARGE_INTEGER	long	DBTYPE_I8	An eight-byte, signed integer: LARGE_INTEGER
STRING	String	DBTYPE_WSTR	A null-terminated Unicode character string: wchar_t[length]; If DBTYPE_WSTR is used by itself, the number of bytes allocated for the string, including the null-termination character, is specified by cbMaxLen in the DBBINDING structure. If DBTYPE_WSTR is combined with DBTYPE_BYREF, the number of bytes allocated for the string, including the null-termination character, is at least the length of the string plus two. In either case, the actual length of the string is determined from the bound length value. The maximum length of the string is the number of allocated bytes divided by sizeof(wchar_t) and truncated to the nearest integer.

2.7.3. The OlapDatabaseMetaData interface, and methods which return schema rowsets

OlapDatabaseMetaData (extends java.sql.DatabaseMetaData) contains methods which return schema result sets.

Schema result sets are specified as in [XML for Analysis specification]. Here is a table of the XML/A methods and the corresponding olap4j method and element type.

XML for Analysis schema rowset	Schema result set method	Metadata element
DBSCHEMA_CATALOGS	DatabaseMetaData.getCatalogs	Catalog
not supported	DatabaseMetaData.getSchemas	Schema
DBSCHEMA_COLUMNS	not supported	not supported
DBSCHEMA_PROVIDER_TYPES	not supported	not supported
DBSCHEMA_TABLES	not supported	not supported
DBSCHEMA_TABLES_INFO	not supported	not supported
DISCOVER_DATASOURCES	OlapDatabaseMetaData.getDatabases	Database
DISCOVER_ENUMERATORS	not supported	not supported
DISCOVER_KEYWORDS	OlapDatabaseMetaData.getMdxKeywords	not supported
DISCOVER_LITERALS	OlapDatabaseMetaData.getLiterals	not supported
DISCOVER_PROPERTIES	OlapDatabaseMetaData.getDatabaseProperties	not supported
DISCOVER_SCHEMA_ROWSETS	not supported	not supported
MDSCHEMA_ACTIONS	OlapDatabaseMetaData.getActions	not supported
MDSCHEMA_CUBES	OlapDatabaseMetaData.getCubes	Cube
MDSCHEMA_DIMENSIONS	OlapDatabaseMetaData.getDimensions	Dimension

MDSCHEMA_FUNCTIONS	OlapDatabaseMetaData.getFunctions	not supported
MDSCHEMA_HIERARCHIES	OlapDatabaseMetaData.getHierarchies	Hierarchy
MDSCHEMA_INPUT_DATASOURCES	not supported	not supported
MDSCHEMA_KPIS	not supported	not supported
MDSCHEMA_LEVELS	OlapDatabaseMetaData.getLevels	Level
MDSCHEMA_MEASURES	OlapDatabaseMetaData.getMeasures	Measure
MDSCHEMA_MEMBERS	OlapDatabaseMetaData.getMembers	Member
MDSCHEMA_PROPERTIES	OlapDatabaseMetaData.getProperties	Property
MDSCHEMA_SETS	OlapDatabaseMetaData.getSets	NamedSet

The rows returned in the result set returned from the metadata methods are structured according to the result set column layouts detailed in this section.

All columns noted in the following result sets are required, and they must be returned in the order shown. However, additional columns (which should be ignored by clients not expecting them) can be added at the end, and some columns can contain null data for info that does not apply.

The following sections describe the columns in each rowset. Each section includes a table that provides the following information for each column.

Column heading	Contents
Column name	The name of the column in the output rowset.
Type	A description of the data type for the column, and whether the column may be NULL.
Description	A brief description of the purpose of the column.

2.7.3.1. getDatabases

Specified by the `DISCOVER_DATASOURCES` XML for Analysis method.

Note that we use the name 'database' rather than 'data source' because 'data source' has a well-established and entirely different meaning (see interface javax.sql.DataSource) in the JDBC specification.

The returned result set contains the following columns.

Column name	Type	Description
DATA_SOURCE_NAME	String	The name of the data source, such as **FoodMart 2000**. Never null.
DATA_SOURCE_DESCRIPTION	String	A description of the data source, as entered by the publisher.
URL	String	The unique path that shows where to invoke the XML for Analysis methods for that data source.
DATA_SOURCE_INFO	String	A string containing any additional information required to connect to the data source. This can include the Initial Catalog property or other information for the provider. Example: `"Provider=MSOLAP;Data Source=Local;"`
PROVIDER_NAME	String	The name of the provider behind the data source.

		Example: `"MSDASQL"`
PROVIDER_TYPE	String	Comma-separated list of the types of data supported by the provider. May include one or more of the following types. Example follows this table. • **TDP**: tabular data provider. • **MDP**: multidimensional data provider. • **DMP**: data mining provider. A DMP provider implements the OLE DB for Data Mining specification.
AUTHENTICATION_MODE	String	Specification of what type of security mode the data source uses. Values can be one of the following, never null: • **Unauthenticated**: no user ID or password needs to be sent. • **Authenticated**: User ID and Password must be included in the information required for the connection. • **Integrated**: the data source uses the underlying security to determine authorization, such as Integrated Security provided by Microsoft Internet Information Services (IIS).

2.7.3.2. getDatabaseProperties

Returns information about the standard and provider-specific properties supported by an olap4j provider. Properties that are not supported by a provider are not listed in the return result set.

Specified by the `DISCOVER_PROPERTIES` XML for Analysis method, except that we rename the VALUE property to PROPERTY_VALUE because "VALUE" is a SQL:2003 reserved word.

The returned result set contains the following columns.

Column name	Type	Description
PROPERTY_NAME	String	The name of the property. Never null.
PROPERTY_DESCRIPTION	String	A localizable text description of the property.
PROPERTY_TYPE	String	The XML data type of the property.
PROPERTY_ACCESS_TYPE	String	Access for the property. The value can be Read, Write, or ReadWrite. Never null.
IS_REQUIRED	boolean	True if a property is required, false if it is not required.
PROPERTY_VALUE	String	The current value of the property. This property is named VALUE in XMLA.

2.7.3.3 getLiterals

Retrieves a list of information on supported literals, including data types and values.

Specified by the `DISCOVER_LITERALS` XML for Analysis method.

The returned result set contains the following columns.

Column name	Type	Description
LITERAL_NAME	String	The name of the literal described in the row. Never null. Example: DBLITERAL_LIKE_PERCENT.
LITERAL_VALUE	String	Contains the actual literal value. Example, if `LITERAL_NAME` is DBLITERAL_LIKE_PERCENT and the percent character (%) is used to match zero or more characters in a LIKE clause, this column's value would be "%".
LITERAL_INVALID_CHARS	String	The characters, in the literal, that are not valid. For example, if table names can contain anything other than a numeric character, this string would be "0123456789".
LITERAL_INVALID_ STARTING_CHARS	String	The characters that are not valid as the first character of the literal. If the literal can start with any valid character, this is null.
LITERAL_MAX_LENGTH	int	The maximum number of characters in the literal. If there is no maximum or the maximum is unknown, the value is -1.

2.7.3.4. getCubes

Describes the structure of cubes within a database.

Specified by the `MDSCHEMA_CUBES` XML for Analysis method.

The returned result set contains the following columns.

Column name	Type	Description
CATALOG_NAME	String	The name of the database.
SCHEMA_NAME	String	Not supported.
CUBE_NAME	String	The name of the cube or dimension. Dimension names are prefaced by a dollar sign ($) symbol.
CUBE_TYPE	String	The type of the cube. Valid values are: • **CUBE** • **DIMENSION**

CUBE_GUID	String	Not supported.
CREATED_ON	Timestamp	Not supported.
LAST_SCHEMA_UPDATE	Timestamp	The time that the cube was last processed.
SCHEMA_UPDATED_BY	String	Not supported.
LAST_DATA_UPDATE	Timestamp	The time that the cube was last processed.
DATA_UPDATED_BY	String	Not supported.
DESCRIPTION	String	A user-friendly description of the cube.
IS_DRILLTHROUGH_ENABLED	boolean	A Boolean that always returns true.
IS_LINKABLE	boolean	A Boolean that indicates whether a cube can be used in a linked cube.
IS_WRITE_ENABLED	boolean	A Boolean that indicates whether a cube is write-enabled.
IS_SQL_ENABLED	boolean	A Boolean that indicates whether SQL can be used on the cube.
CUBE_CAPTION	String	The caption of the cube.
BASE_CUBE_NAME	String	The name of the source cube if this cube is a perspective cube.
ANNOTATIONS	String	(Optional) A set of notes, in XML format.

The rowset is sorted on **CATALOG_NAME**, **SCHEMA_NAME**, **CUBE_NAME**.

2.7.3.5. getDimensions

Retrieves a result set describing the shared and private dimensions within a database.

Specified by the `MDSCHEMA_DIMENSIONS` XML for Analysis method.

The returned result set contains the following columns.

Column name	Type	Description
CATALOG_NAME	String	The name of the database.
SCHEMA_NAME	String	Not supported.
CUBE_NAME	String	The name of the cube.
DIMENSION_NAME	String	The name of the dimension. If a dimension is part of more than one cube or measure group, then there is one row for each unique combination of dimension, measure group, and cube.
DIMENSION_UNIQUE_NAME	String	The unique name of the dimension.
DIMENSION_GUID	String	Not supported.
DIMENSION_CAPTION	String	The caption of the dimension. This should be used when displaying the name of the dimension to the user, such as in the user interface or reports.
DIMENSION_ORDINAL	int	The position of the dimension within the cube.
DIMENSION_TYPE	int	The type of the dimension. Valid values include the values of the `xmlaOrdinal` attribute of the org.olap4j.Dimension.Type enum.

DIMENSION_CARDINALITY	int	The number of members in the key attribute.
DEFAULT_HIERARCHY	String	A hierarchy from the dimension. Preserved for backwards compatibility.
DESCRIPTION	String	A user-friendly description of the dimension.
IS_VIRTUAL	boolean	Always `false`.
IS_READWRITE	boolean	A Boolean that indicates whether the dimension is write-enabled. `true` if the dimension is write-enabled.
DIMENSION_UNIQUE_SETTINGS	int	A bitmap that specifies which columns contain unique values if the dimension contains only members with unique names. The following bit value constants are defined for this bitmap: • **MDDIMENSIONS_MEMBER_KEY_UNIQUE (1)**
DIMENSION_MASTER_ UNIQUE_NAME	String	Always `null`.
DIMENSION_IS_VISIBLE	boolean	Always `true`.

The result set is sorted on **CATALOG_NAME**, **SCHEMA_NAME**, **CUBE_NAME**, **DIMENSION_NAME**.

2.7.3.6. getFunctions

Retrieves a result set describing the functions available to client applications connected to the database.

Specified by the `MDSCHEMA_FUNCTIONS` XML for Analysis method.

The returned result set contains the following columns.

Column name	Type	Description
FUNCTION_NAME	String	The name of the function.
DESCRIPTION	String	A description of the function.
PARAMETER_LIST	String	A comma delimited list of parameters formatted as in Microsoft Visual Basic. For example, a parameter might be Name as String.
RETURN_TYPE	int	The **VARTYPE** of the return data type of the function.
ORIGIN	int	The origin of the function: • 1 for MDX functions. • 2 for user-defined functions.
INTERFACE_NAME	String	The name of the interface for user-defined functions The group name for Multidimensional Expressions (MDX) functions.
LIBRARY_NAME	String	The name of the type library for user-defined

		functions. `null` for MDX functions.
DLL_NAME	String	(Optional) The name of the assembly that implements the user-defined function. Returns `null` for MDX functions.
HELP_FILE	String	(Optional) The name of the file that contains the help documentation for the user-defined function. Returns `null` for MDX functions.
HELP_CONTEXT	int	(Optional) Returns the Help context ID for this function.
OBJECT	String	(Optional) The generic name of the object class to which a property applies. For example, the rowset corresponding to the <level_name>.Members function returns "**Level**". Returns `null` for user-defined functions, or non-property MDX functions.
CAPTION	String	The display caption for the function.

The rowset is sorted on **ORIGIN**, **INTERFACE_NAME**, **FUNCTION_NAME**.

2.7.3.7. getHierarchies

Retrieves a result set describing each hierarchy within a particular dimension.

Specified by the `MDSCHEMA_HIERARCHIES` XML for Analysis method.

The returned result set contains the following columns.

Column name	Type	Description
CATALOG_NAME	String	The name of the catalog to which this hierarchy belongs. `null` if the provider does not support catalogs.
SCHEMA_NAME	String	Not supported
CUBE_NAME	String	(Required) The name of the cube to which this hierarchy belongs.
DIMENSION_UNIQUE_NAME	String	The unique name of the dimension to which this hierarchy belongs. For providers that generate unique names by qualification, each component of this name is delimited.
HIERARCHY_NAME	String	The name of the hierarchy. Blank if there is only a single hierarchy in the dimension. This will always have a value in Microsoft SQL Server 2005 Analysis Services (SSAS).
HIERARCHY_UNIQUE_NAME	String	The unique name of the hierarchy.
HIERARCHY_GUID	String	Not supported
HIERARCHY_CAPTION	String	A label or a caption associated with the hierarchy. Used primarily for display purposes. If a caption does not exist, **HIERARCHY_NAME** is returned. If the dimension either does not contain a hierarchy or has just one hierarchy, this column will contain the name of the dimension.

DIMENSION_TYPE	int	The type of the dimension. Valid values include the values of the xmlaOrdinal attribute of .
HIERARCHY_CARDINALITY	int	The number of members in the hierarchy.
DEFAULT_MEMBER	String	The default member for this hierarchy. This is a unique name. Every hierarchy must have a default member.
ALL_MEMBER	String	The member at the highest level of the rollup.
DESCRIPTION	String	A human-readable description of the hierarchy. `null` if no description exists.
STRUCTURE	int	The structure of the hierarchy. Valid values include the following values: • **MD_STRUCTURE_FULLYBALANCED** (**0**) • **MD_STRUCTURE_RAGGEDBALANCED** (**1**) • **MD_STRUCTURE_UNBALANCED** (**2**) • **MD_STRUCTURE_NETWORK** (**3**)
IS_VIRTUAL	boolean	Always returns `false`.
IS_READWRITE	boolean	A Boolean that indicates whether the Write Back to dimension column is enabled. Returns `true` if the **Write Back to dimension** column that represents this hierarchy is enabled.
DIMENSION_UNIQUE_SETTINGS	int	Always returns **MDDIMENSIONS_MEMBER_KEY_UNIQUE** (**1**).
DIMENSION_MASTER_ UNIQUE_NAME	String	Always returns `null`.
DIMENSION_IS_VISIBLE	boolean	Always returns `true`. If the dimension is not visible, it will not appear in the schema rowset.
HIERARCHY_ORDINAL	int	The ordinal number of the hierarchy across all hierarchies of the cube.
DIMENSION_IS_SHARED	boolean	Always returns `true`.
HIERARCHY_IS_VISIBLE	boolean	A Boolean that indicates whether the hieararchy is visible. Returns `true` if the hierarchy is visible; otherwise, `false`.
HIERARCHY_ORIGIN	int	A bit mask that determines the source of the hierarchy: • **MD_USER_DEFINED** identifies user defined hierarchies, and has a value of **0x0000001**. • **MD_SYSTEM_ENABLED** identifies attribute hierarchies, and has a value of **0x0000002**. • **MD_SYSTEM_INTERNAL** identifies attributes with no attribute hierarchies, and has a value of **0x0000004**. A parent/child attribute hierarchy is both **MD_USER_DEFINED** and **MD_SYSTEM_ENABLED**.
HIERARCHY_DISPLAY_FOLDER	String	The path to be used when displaying the hierarchy in the user interface. Folder names will be separated by a semicolon (;).

		Nested folders are indicated by a backslash (\).
INSTANCE_SELECTION	int	A hint to the client application on how to show the hierarchy. Valid values include the following values: • **MD_INSTANCE_SELECTION_NONE** • **MD_INSTANCE_SELECTION_DROPDOWN** • **MD_INSTANCE_SELECTION_LIST** • **MD_INSTANCE_SELECTION_FILTEREDLIST** • **MD_INSTANCE_SELECTION_MANDATORYFILTER**

The rowset is sorted on **CATALOG_NAME**, **SCHEMA_NAME**, **CUBE_NAME**, **DIMENSION_UNIQUE_NAME**, **HIERARCHY_NAME**.

2.7.3.8. getLevels

Retrieves a result set describing each level within a particular hierarchy.

Specified by the `MDSCHEMA_LEVELS` XML for Analysis method.

The returned result set contains the following columns.

Column name	Type	Description
CATALOG_NAME	String	The name of the catalog to which this level belongs. `null` if the provider does not support catalogs.
SCHEMA_NAME	String	The name of the schema to which this level belongs. `null` if the provider does not support schemas.
CUBE_NAME	String	The name of the cube to which this level belongs.
DIMENSION_UNIQUE_NAME	String	The unique name of the dimension to which this level belongs. For providers that generate unique names by qualification, each component of this name is delimited.
HIERARCHY_UNIQUE_NAME	String	The unique name of the hierarchy. If the level belongs to more than one hierarchy, there is one row for each hierarchy to which it belongs. For providers that generate unique names by qualification, each component of this name is delimited.
LEVEL_NAME	String	The name of the level.
LEVEL_UNIQUE_NAME	String	The properly escaped unique name of the level.
LEVEL_GUID	String	Not supported.
LEVEL_CAPTION	String	A label or caption associated with the hierarchy. Used primarily for display purposes. If a caption does not exist, **LEVEL_NAME** is returned.
LEVEL_NUMBER	int	The distance of the level from the root of the hierarchy. Root level is zero (**0**).
LEVEL_CARDINALITY	int	The number of members in the level.
LEVEL_TYPE	int	Type of the level. Values are as allowed by the `xmlaOrdinal` field of the org.olap4j.Level.Type enum.
DESCRIPTION	String	A human-readable description of the level. null if no

		description exists.
CUSTOM_ROLLUP_SETTINGS	int	A bitmap that specifies the custom rollup options: • **MDLEVELS_CUSTOM_ROLLUP_EXPRESSION** (**0x01**) indicates an expression exists for this level. (Deprecated) • **MDLEVELS_CUSTOM_ROLLUP_COLUMN** (**0x02**) indicates that there is a custom rollup column for this level. • **MDLEVELS_SKIPPED_LEVELS** (**0x04**) indicates that there is a skipped level associated with members of this level. • **MDLEVELS_CUSTOM_MEMBER_PROPERTIES** (**0x08**) indicates that members of the level have custom member properties. • **MDLEVELS_UNARY_OPERATOR** (**0x10**) indicates that members on the level have unary operators.
LEVEL_UNIQUE_SETTINGS	int	A bitmap that specifies which columns contain unique values, if the level only has members with unique names or keys. The Msmd.h file defines the following bit value constants for this bitmap: • **MDDIMENSIONS_MEMBER_KEY_UNIQUE** (**1**) • **MDDIMENSIONS_MEMBER_NAME_UNIQUE** (**2**)
		The key is always unique in Microsoft SQL Server 2005 Analysis Services (SSAS). The name will be unique if the setting on the attribute is **UniqueInDimension** or **UniqueInAttribute**
LEVEL_IS_VISIBLE	boolean	A Boolean that indicates whether the level is visible. Always returns True. If the level is not visible, it will not be included in the schema rowset.
LEVEL_ORDERING_PROPERTY	String	The ID of the attribute that the level is sorted on.
LEVEL_DBTYPE	int	The **DBTYPE** enumeration of the member key column that is used for the level attribute. Null if concatenated keys are used as the member key column.
LEVEL_MASTER_ UNIQUE_NAME	String	Always returns null.
LEVEL_NAME_ SQL_COLUMN_NAME	String	The SQL representation of the level member names.
LEVEL_KEY_ SQL_COLUMN_NAME	String	The SQL representation of the level member key values.
	String	The SQL representation of the member unique names.

LEVEL_UNIQUE_NAME_ SQL_COLUMN_NAME		
LEVEL_ATTRIBUTE_ HIERARCHY_NAME	String	The name of the attribute hierarchy providing the source of the level.
LEVEL_KEY_CARDINALITY	int	The number of columns in the level key.
LEVEL_ORIGIN	int	A bit map that defines how the level was sourced: • **MD_ORIGIN_USER_DEFINED** identifies levels in a user defined hierarchy. • **MD_ORIGIN_ATTRIBUTE** identifies levels in an attribute hierarchy. • **MD_ORIGIN_KEY_ATTRIBUTE** identifies levels in a key attribute hierarchy. • **MD_ORIGIN_INTERNAL** identifies levels in attribute hierarchies that are not enabled.

The rowset is sorted on **CATALOG_NAME**, **SCHEMA_NAME**, **CUBE_NAME**, **DIMENSION_UNIQUE_NAME**, **HIERARCHY_UNIQUE_NAME**, **LEVEL_NUMBER**.

2.7.3.9. getMeasures

Retrieves a result set describing each measure within a cube.

Specified by the `MDSCHEMA_MEASURES` XML for Analysis method.

The returned result set contains the following columns.

Column name	Type	Description
CATALOG_NAME	String	The name of the catalog to which this measure belongs. `null` if the provider does not support catalogs.
SCHEMA_NAME	String	The name of the schema to which this measure belongs. `null` if the provider does not support schemas.
CUBE_NAME	String	The name of the cube to which this measure belongs.
MEASURE_NAME	String	The name of the measure.
MEASURE_UNIQUE_NAME	String	The Unique name of the measure. For providers that generate unique names by qualification, each component of this name is delimited.
MEASURE_CAPTION	String	A label or caption associated with the measure. Used primarily for display purposes. If a caption does not exist, **MEASURE_NAME** is returned.
MEASURE_GUID	String	Not supported.
MEASURE_AGGREGATOR	int	An enumeration that identifies how a measure was derived. Can be one of the values allowed by the `xmlaOrdinal` field of the org.olap4j.Measure.Aggregator enum.
DATA_TYPE	int	The data type of the measure.

NUMERIC_PRECISION	int	The maximum precision of the property if the measure object's data type is exact numeric. `null` for all other property types.
NUMERIC_SCALE	int	The number of digits to the right of the decimal point if the measure object's type indicator is **DBTYPE_NUMERIC** or **DBTYPE_DECIMAL**. Otherwise, this value is `null`.
MEASURE_UNITS	String	Not supported
DESCRIPTION	String	A human-readable description of the measure. **null** if no description exists.
EXPRESSION	String	An expression for the member.
MEASURE_IS_VISIBLE	boolean	A Boolean that always returns True. If the measure is not visible, it will not be included in the schema rowset.
LEVELS_LIST	String	A string that always returns `null`.
MEASURE_NAME_ SQL_COLUMN_NAME	String	The name of the column in the SQL query that corresponds to the measure's name.
MEASURE_UNQUALIFIED_ CAPTION	String	The name of the measure, not qualified with the measure group name.
MEASUREGROUP_NAME	String	The name of the measure group to which the measure belongs.
MEASURE_DISPLAY_FOLDER	String	The path to be used when displaying the measure in the user interface. Folder names will be separated by a semicolon. Nested folders are indicated by a backslash (\).
DEFAULT_FORMAT_STRING	String	The default format string for the measure.

The rowset is sorted on **CATALOG_NAME**, **SCHEMA_NAME**, **CUBE_NAME**, **MEASURE_NAME**.

2.7.3.10. getMembers

Retrieves a result set describing the members within a database.

Specified by the `MDSCHEMA_MEMBERS` XML for Analysis method.

The returned result set contains the following columns.

Column name	Type	Description
CATALOG_NAME	String	The name of the database to which this member belongs.
SCHEMA_NAME	String	The name of the schema to which this member belongs.
CUBE_NAME	String	The name of the cube to which this member belongs.
DIMENSION_UNIQUE_NAME	String	The unique name of the dimension to which this member belongs.
HIERARCHY_UNIQUE_NAME	String	The unique name of the hierarchy to which this member belongs.
LEVEL_UNIQUE_NAME	String	

		The unique name of the level to which this member belongs.
LEVEL_NUMBER	int	The distance of the member from the root of the hierarchy. The root level is zero (0).
MEMBER_ORDINAL	int	(Deprecated) Always returns **0**.
MEMBER_NAME	String	The name of the member.
MEMBER_UNIQUE_NAME	String	The unique name of the member.
MEMBER_TYPE	int	The type of the member, one of the values of the `ordinal` field of the org.olap4j.Member.Type enum. **FORMULA** takes precedence over **MEASURE**. For example, if there is a formula (calculated) member on the Measures dimension, it is listed as **FORMULA**.
MEMBER_GUID	String	The GUID of the member. `null` if no GUID exists.
MEMBER_CAPTION	String	A label or caption associated with the member. Used primarily for display purposes. If a caption does not exist, **MEMBER_NAME** is returned.
CHILDREN_CARDINALITY	int	The number of children that the member has. This can be an estimate, so consumers should not rely on this to be the exact count. Providers should return the best estimate possible.
PARENT_LEVEL	int	The distance of the member's parent from the root level of the hierarchy. The root level is zero (0).
PARENT_UNIQUE_NAME	String	The unique name of the member's parent. `null` is returned for any members at the root level.
PARENT_COUNT	int	The number of parents that this member has.
DESCRIPTION	String	Always returns `null`.
EXPRESSION	String	The expression for calculations, if the member is of type **MDMEMBER_TYPE_FORMULA**.
MEMBER_KEY	String	The value of the member's key column. Returns **null** if the member has a composite key.
IS_PLACEHOLDERMEMBER	boolean	A Boolean that indicates whether a member is a placeholder member for an empty position in a dimension hierarchy. It is valid only if the **MDX Compatibility** property has been set to 1.
IS_DATAMEMBER	boolean	A Boolean that indicates whether the member is a data member. Returns True if the member is a data member.
Zero or more additional columns	int	No properties are returned if the members could be returned from multiple levels. For example, if the Tree operator is **PARENT** and **SELF** for a non-parent child hierarchy, no member properties

	are returned. This applies to ragged hierarchies where tree operators could return members from different levels (for example, if the prior level contains holes and parent on members is requested).

The rowset is sorted on **CATALOG_NAME**, **SCHEMA_NAME**, **CUBE_NAME**, **DIMENSION_UNIQUE_NAME**, **HIERARCHY_UNIQUE_NAME**, **LEVEL_UNIQUE_NAME**, **LEVEL_NUMBER**, **MEMBER_ORDINAL**.

2.7.3.11. getProperties

Retrieves a list of descriptions of member and cell Properties.

Specified by the `MDSCHEMA_PROPERTIES` XML for Analysis method.

The returned result set contains the following columns.

Column name	Type	Description
CATALOG_NAME	String	The name of the database.
SCHEMA_NAME	String	The name of the schema to which this property belongs. `null` if the provider does not support schemas.
CUBE_NAME	String	The name of the cube.
DIMENSION_UNIQUE_NAME	String	The unique name of the dimension. For providers that generate unique names by qualification, each component of this name is delimited.
HIERARCHY_UNIQUE_NAME	String	The unique name of the hierarchy. For providers that generate unique names by qualification, each component of this name is delimited.
LEVEL_UNIQUE_NAME	String	The unique name of the level to which this property belongs. If the provider does not support named levels, it should return the **DIMENSION_UNIQUE_NAME** value for this field. For providers that generate unique names by qualification, each component of this name is delimited.
MEMBER_UNIQUE_NAME	String	The unique name of the member to which the property belongs. Used for data stores that do not support named levels or have properties on a member-by-member basis. If the property applies to all members in a level, this column is `null`. For providers that generate unique names by qualification, each component of this name is delimited.
PROPERTY_TYPE	int	A bitmap that specifies the type of the property: • **MDPROP_MEMBER** (**1**) identifies a property of a member. This property can be

		used in the DIMENSION PROPERTIES clause of the SELECT statement. • **MDPROP_CELL** (**2**) identifies a property of a cell. This property can be used in the CELL PROPERTIES clause that occurs at the end of the SELECT statement. • **MDPROP_SYSTEM** (**4**) identifies an internal property. • **MDPROP_BLOB** (**8**) identifies a property which contains a binary large object (blob).
PROPERTY_NAME	String	The name of the property. If the key for the property is the same as the name for the property, **PROPERTY_NAME** will be blank.
PROPERTY_CAPTION	String	A label or caption associated with the property, used primarily for display purposes. Returns **PROPERTY_NAME** if a caption does not exist.
DATA_TYPE	int	The data type of the property.
CHARACTER_ MAXIMUM_LENGTH	int	The maximum possible length of the property, if it is a character, binary, or bit type. Zero indicates there is no defined maximum length. Returns `null` for all other data types.
CHARACTER_OCTET_LENGTH	int	The maximum possible length (in bytes) of the property, if it is a character or binary type. Zero indicates there is no defined maximum length. Returns `null` for all other data types.
NUMERIC_PRECISION	int	The maximum precision of the property, if it is a numeric data type. Returns `null` for all other data types.
NUMERIC_SCALE	int	The number of digits to the right of the decimal point, if it is a **DBTYPE_NUMERIC** or **DBTYPE_DECIMAL** type. Returns `null` for all other data types.
DESCRIPTION	String	A human readable description of the property. `null` if no description exists.
PROPERTY_CONTENT_TYPE	int	The type of the property. Can be one of the values of the `xmlaOrdinal` field of the org.olap4j.Property.ContentType enum.
SQL_COLUMN_NAME	String	The name of the property used in SQL queries from the cube dimension or database dimension.
LANGUAGE	int	The translation expressed as an **LCID**. Only valid for property translations.
PROPERTY_ORIGIN	int	

		Identifies the type of hierarchy that the property applies to: • **MD_USER_DEFINED** (**1**) indicates the property is on a user defined hierarchy • **MD_SYSTEM_ENABLED** (**2**) indicates the property is on an attribute hierarchy • **MD_SYSTEM_DISABLED** (**4**) indicates the property is on an attribute hierarchy that is not enabled.
PROPERTY_ATTRIBUTE_HIERARCHY_NAME	String	The name of the attribute hierarchy sourcing this property.
PROPERTY_CARDINALITY	String	The cardinality of the property. Possible values include the following strings: • **ONE** • **MANY**
MIME_TYPE	String	The mime type for binary large objects (BLOBs).
PROPERTY_IS_VISIBLE	boolean	A Boolean that indicates whether the property is visible. `true` if the property is visible; otherwise, `false`.

This schema rowset is not sorted.

2.7.3.12. getSets

Retrieves a result set describing any sets that are currently defined in a database, including session-scoped sets.

Specified by the `MDSCHEMA_SETS` XML for Analysis method.

The returned result set contains the following columns.

Column name	Type	Description
CATALOG_NAME	String	The name of the database.
SCHEMA_NAME	String	Not supported.
CUBE_NAME	String	The name of the cube.
SET_NAME	String	The name of the set, as specified in the **CREATE SET** statement.
SCOPE	int	The scope of the set: • **MDSET_SCOPE_GLOBAL** (**1**) • **MDSET_SCOPE_SESSION** (**2**)
DESCRIPTION	String	Not supported.
EXPRESSION	String	The expression for the set.
DIMENSIONS	String	A comma delimited list of hierarchies included in the set.
SET_CAPTION	String	

		A label or caption associated with the set. The label or caption is used primarily for display purposes.
SET_DISPLAY_FOLDER	String	The path to be used by the user interface when displaying the set. Folder names are separated by a backslash (\), folders are separated by a semicolon (**;**).

The rowset is sorted on **CATALOG_NAME**, **SCHEMA_NAME**, **CUBE_NAME**.

2.7.4. Other methods

Method	Description
`getConnection()`	Returns the connection (overrides DatabaseMetaData method).
`getMdxKeywords()`	Returns the keywords of this dialect of MDX, as a comma-separated string.
`getSupportedCellSetListenerGranularities()`	Returns the granularity of changes to cell sets that the database is capable of providing.

2.8. Transform

NOTE: As of olap4j 1.0, this package is experimental and is subject to change in future releases.

A transform is an operation which maps a query model to a new query model. It is usually triggered by a gesture within the user-interface. For example, clicking on the *Unit Sales* column transforms the query

```
SELECT {[Measures].[Store Sales], [Measures].[Unit Sales]} ON COLUMNS,
  {[Product].Members} ON ROWS
FROM [Sales]
```

into one with sorting:

```
SELECT {[Measures].[Store Sales], [Measures].[Unit Sales]} ON COLUMNS,
  Order({[Product].Members}, [Measures].[Unit Sales], ASC) ON ROWS
FROM [Sales]
```

Transformations can only modify a query within a cube - it cannot be used to change the cube that the query is against or to join two cubes. Similarly, the transform package only supports modifying a MDX query model. For example, a "drill" transform can not be used to produce a SQL query that returns data outside of the cube.

Package name: org.olap4j.transform

Classes: (incomplete)

- Tuple
- Set
- CalculatedMember
- NamedSet
- Axis
- Slicer

2.8.1. Query Model Details

This section should probably be moved into Section 2.4.

The MDX query language uses a data model based on cubes, dimensions, tuples and sets. The transformation package allows direct manipulation of a query exploring a cube.

A tuple is a multidimensional member. It is a combination of members from one or more dimensions, with the limitation that only one member can be used from each dimension. A set is an ordered collection of tuples. An MDX query selects zero or more axes using a data slicer. (The axes loosely correspond to the "SELECT" clause in a SQL query, and the slicer to the "WHERE".)

2.8.2. Navigation Actions

The defined set of navigations can be divided into four categories: Slicing, Restructuring, Drilling, Scoping.

2.8.2.1. Slicing Navigations

setSlicer
: Secifies the slicer to use, replacing any current one.

getSlicer
: Retrieve the current slicer.

excludeEmpty
: Removes empty slices from the results.

setLimit
: Limits the results to the top/bottom n results for a specified measure.

2.8.2.2. Restructuring Navigations

Restructuring navigations change the axes of the returned cube.

getAxis
setAxis
: Specifies the Set to use for an Axis. Can be used to add or replace a axis.

deleteAxis
: Removes the specified Axis from the results.

addToAxis
: Appends a new Tuple or Set to an Axis.

moveTuple
: Moves a tuple from one Axis to another. If the tuple is not contained in the first axis this method behaves like addToAxis on the second axis.

reorderAxis
: Reorders the tuples in the axis.

addTotal
: Adds an aggregation (total, min, max, count, distinct) to the specified member. More than one aggregation can be added to a single member.

deleteTotal
: Deletes an aggregation from the specified member.

2.8.2.3. Drilling Navigations

Navigations that allow a user to move through the levels in a hierarchy. All drill navigations operate on a single Axis.

drill
: Moves the specified member up/down one level in the hierarchy. All members of the hierarchy are replaced by this action.

2.8.2.4. Scoping Navigations

Navigations that allow a user to expand/collapse sections of a result set. All scoping navigations operate on single Axis.

expand
: Expand the given measure to include both the current level and all the members one level down the hierarchy. Optionally expands a single measure or all measures at the level.

collapse
: Removes members at a given level of a hierarchy. Optionally collapses a single measure or all measures at the level.

2.8.2.5. Supporting Actions

Axis Operations

- getSet
- setSet
- addToAxis
- removeFromAxis

Set Operations

- getTuple
- setTuple
- addToSet - includes Tuples, ranges of Tuples, functions, properties
- removeFromSet

Tuple Operations

- addMember
- deleteMember
- getMembers

Open Issues

- Is this API at the right level, or is it too close to MDX?
- Do we want to include support for adding highlighting conditions?
- What about result formatting as part of the query?
- Should common operators such as CrossJoin(), Order() and Filter() be baked into the API as methods, or just treated as functions?

- How should we handle functions that return or modify Sets? We want to make it easy to wrap an entire axis in a function.
- Should we limit the Slicer to a Tuple or allow a Set? I believe the MDX spec allows a Set, but I don't know if anybody supports it.
- I think we should add some explicit time based functions, since time based analysis is so common, and so frequently done wrong. ie if the Axis is using a time based dimension you can use "setCompareToPreviousTimePeriod()" instead of having to add the previous time period as a member and calculate the change.

2.9. Layout

NOTE: As of olap4j 1.0, this package is experimental and is subject to change in future releases.

The layout package provides data models for graphical OLAP applications. In particular, the GridModel class provides, for OLAP data, what Swing's TableModel provides for SQL data.

Package name: org.olap4j.layout

Classes: TBD

2.10. Scenarios

NOTE: As of olap4j 1.0, this functionality is experimental and is subject to change in future releases.

Scenarios allow an application to change values of cells. When the value of a cell changes, values of related cells also change (parent cells, child and descendant cells, and calculated cells).

Scenarios can therefore be used to perform 'what-if' analysis, useful in budgeting or forecasting applications. This functionality is commonly called 'write-back' (or sometimes 'writeback' or 'writethrough'; see for instance the wikipedia article "Comparison of OLAP Servers"), but we avoid that term because this specification does not stipulate that a provider implements scenarios by writing the changed values to disk.

Each scenario has a different set of modifications. There is a base scenario where the values are unchanged from the star schema; in this scenario, cells cannot be modified.

A provider may provide a `[Scenario]` dimension for each cube for which scenarios are enabled. This dimension contains a member for each scenario that is visible in the current access-control context; the name of each member is the value returned by the `getId()` method. The default member of the Scenario dimension is the current scenario for the current connection (as set by the `OlapConnection.setScenario()` method).

The Scenario dimension behaves in the way you would expect. For example, if a query contains a slicer `WHERE [Scenario].[1]` then the cell values returned by that query will reflect their values under that scenario. Also, you can define cross-dimensional calculations, such as `WITH MEMBER [Gain] AS ([Time].[2011], [Scenario].[1] - [Scenario].[Default Scenario])`, to compare values under two or more scenarios.

A particular provider may provide a means to save a scenario. (Say, to modify the fact table, or save the scenario to disk in some other format.)

A particular provider may support access control to scenarios. (For example, a particular scenario is invisible to role A, visible but read-only to role B, and read-write to role C.)

Methods of the Scenario class:

- `String getId()` // returns the unique identifier of this scenario

Other methods relating to scenarios:

- `Scenario OlapConnection.createScenario()` // creates a scenario
- `void OlapConnection.setScenario(Scenario)` // sets the current scenario for this connection
- `Scenario OlapConnection.getScenario()` // returns this connection's current scenario
- `void Cell.setValue(Object value, AllocationPolicy allocationPolicy, Object... allocationArgs)` // sets the value of a cell

2.11. Notifications

NOTE: As of olap4j 1.0, this functionality is experimental and is subject to change in future releases.

The `CellSetListener` interface allows an application to receive events when the contents of a CellSet have changed.

The client can ask the server to provide the listener with a specific granularity of events, but the server can decline to provide that granularity.

Fine granularity deals with changes such as cell values changing (and reports the before and after value, before and after formatted value), positions being deleted, positions being changed.

When an atomic change happens on the server (say a cache flush, if the server is mondrian) then an event will arrive on the client containing all of those changes. Although `CellSetListener.CellSetChange.getCellChanges()` and `CellSetListener.CellSetChange.getAxisChanges()` return lists, the client should assume that all of the events in these lists occur simultaneously.

At any point, the server is free to throw up its hands and say 'there are too many changes' by sending null values for `getCellChanges` or `getAxisChanges`. This prevents situations where there are huge numbers of changes that might overwhelm the server, the network link, or the client, such as might happen if a large axis is re-sorted.

The client should always be ready for that to happen (even for providers that claim to provide fine granularity events), and should re-execute the query to get the cell set. In fact, we recommend that clients re-execute the query to get a new cellset whenever they get an event. Then the client can use the details in the event to highlight cells that have changed.

Methods on interface CellSetListener:

- `cellSetChanged(CellSetChange)` // invoked when a cell set has changed
- `cellSetClosed(CellSet)` // invoked when a cell set is closed
- `cellSetOpened(CellSet)` // invoked when a cell set is opened

Methods on interface CellSetChange:

- `CellSet getCellSet()` // returns the cell set affected by this change
- `List<CellChange> getCellChanges()` // returns a list of cells that have changed, or null if the server cannot provide detailed changes
- `List<AxisChange> getAxisChanges()` // returns a list of axis changes, or null if the server cannot provide detailed changes

Methods on interface AxisChange:

- `CellSetAxis getAxis()` // returns the axis affected by this change
- `Position getBeforePosition()` // returns the position before the change; null if the change created a new position
- `Position getAfterPosition()` // returns the position after the change; null if the change deleted a new position

Methods on interface CellChange:

- `Cell getBeforeCell()` // returns the cell before the change
- `Cell getAfterCell()` // returns the cell after the change

Other methods:

- `OlapDatabaseMetaData.getSupportedCellSetListenerGranularities()` // returns the granularity of changes to cell sets that the database is capable of providing
- `OlapStatement.addListener(Granularity, CellSetListener)` // adds a listener to be notified of events to CellSets created by this statement

Notes for implementors

The purpose of registering a listener before creating a cell set is to ensure that no events "leak out" between creating a cell set and registering a listener, or while a statement is being re-executed to produce a new cell set.

The `cellSetOpened(CellSet)` and `cellSetClosed(CellSet)` methods are provided so that the listener knows what is going on when a statement is re-executed. In particular, suppose a statement receives an change event decides to re-execute. The listener is attached to the statement, so receives notifications about both old and new cell sets. The driver implicitls closes the previous cell set and calls cellSetClosed, then calls `cellSetOpened` with the new cell set.

If changes are occurring regularly on the server, there will soon be a call to `cellSetChanged(CellSetChange)`. It is important to note that this event contains only changes that have occurred since the new cell set was opened.

The granularity parameter is provided to `OlapStatement.addListener(Granularity, CellSetListener)` for the server's benefit. If granularity is only `Granularity.COARSE`, the server may be able to store less information in order to track the cell set.

2.12. Drill through

olap4j provides two ways of drilling through to get the collection of atomic rows underlying a given cell.

- The `Cell.drillThrough()` method drills through a given cell in the cell set returned by a previously executed statement.
- Execute the `DRILLTHROUGH` MDX statement using the `OlapStatement.executeStatement(String sql)` method.

The `DRILLTHROUGH` statement is a more powerful approach, because it offers options `MAXROWS` to limit the number of rows returned, and `RETURN` to choose which attributes and measures are projected, but not all OLAP servers implement it.

Note that we call the `ResultSet Statement.executeStatement(String sql)` method, not `CellSet OlapStatement.executeOlapStatement(String mdx)`, because the result of drillthrough is relational (rows and columns), not a dimensional (axes and cells). A statement can be created by calling `OlapConnection.createStatement()`; even though this returns an `OlapStatement`, the OlapStatement is required to implement applicable methods of its `Statement` base class.

3. Other topics

In this section we discuss aspects of the design and usage of olap4j which pervade all of the components.

3.1. Internationalization

Metadata elements in olap4j can be localized. Unlike the tables and columns model of relational databases and JDBC, elements of an OLAP data model appear on the screen of the end-user, and the user expects these elements to appear in his or her own language.

A connection has a locale. For most drivers, this can be initialized using a connection parameter called `Locale`. The locale can be overridden by calling `OlapConnection.setLocale(Locale)`.

Metadata elements Cube, Dimension, Hierarchy, Level, Member and so forth have methods `getCaption` and `getDescription` (inherited from MetadataElement). The values returned from these methods depend on the locale of the connection.

Suppose one cube is available in English and French, and in French and Spanish, and both are shown in same portal. Clients typically say that seeing reports in a mixture of languages is confusing; the portal would figure out the best common language, in this case French. The Cube and Schema objects have `getSupportedLocales()` methods for this purpose.

3.2. Concurrency and thread-safety

The JDBC 4.0 specification describes the thread-safety requirements for drivers, and what modes of concurrency JDBC applications can assume that their drivers will support. Since the olap4j specification is an extension to the JDBC specification, an olap4j driver must comply with the JDBC specification in this regard.

3.3. Canceling statements

The JDBC specification provides the Statement.cancel() method, so that a statement which is executing in one thread may be safely terminated by another thread; and Statement.setQueryTimeout(int seconds), to request that a statement aborts itself after executing for a certain period of time.

4. Other components

The API described above is a set of interfaces which must be implemented by any compliant provider. The olap4j project also contains some components which are not part of the API.

4.1. Test suite

The olap4j project contains a TCK (Test Compatibility Kit). The TCK is a suite of tests which can be used to verify the compliance of an implementation of the API.

4.2. XML/A provider

The XML/A provider is an implementation of the olap4j API which talks to a generic XML/A provider.

Since there are many XML/A providers, and some of them require requests in a particular format and/or produce idiosyncratic responses, the XML/A provider will come in several flavors.

The XML/A provider is being developed in the same source-code repository as olap4j, in a Java package `org.olap4j.driver.xmla`, but is not part of the olap4j specification or release.

5. Non-functionality

Here are some of the areas of functionality which will *not* be part of olap4j:

- Schema reader parses an XML file to create a schema
- Cache management functions
- Ability to create/modify schema dynamically
- Definitions of MDX functions (such as the number and types of parameters)
- SPI to extend the system by creating user-defined functions and so forth
- XML/A bridge (to make an olap4j data source appear as an XML/A server)
- SchemaReader

6. Related projects

6.1. Mondrian provider

The Mondrian project contains an implementation of the olap4j API based on the Mondrian OLAP engine, namely the mondrian.olap4j.MondrianOlap4jDriver driver. It is the reference implementation of olap4j.

6.2. XML for Analysis provider

We intend to create an a driver which implements the olap4j API on top of any XML/A data source.

This code is currently being developed in the same source-code repository as olap4j, but will be spun off as a separate project at some point.

6.3. Other data sources

In principle, providers could be created to other OLAP data sources. This would be particularly straightforward for servers which already have a native Java API.

6.4 xmla4js

xmla4js is a JavaScript front-end to XML/A.

Appendix A. Opportunities for specification

The following are features which have been suggested for inclusion in the olap4j specification, but which are not part of the current version. They may be included in future revisions of the specification.

A.1. Date and Time types

Include support for Date and Time values. The package `org.olap4j.type` could have additional classes DateType and TimeType.

(Richard Emberson, 2006/8/14)

A.2. Schema notification

Add a mechanism for the client to detect that the schema has been modified (for instance, that a cube has been added). Not necessarily to find out what those changes are.

(Richard Emberson, 2006/8/15)

Appendix B. Feedback

Richard Emberson, email, 2006/8/15

"One thing we found about XMLA was that our users wanted all roles to be defined, stored, modified, and accessed though the same mechanism. With a large application with many areas that can be permissioned, it is important that olap4j let an application builder manage roles externally and apply them as part of an individual's execution context."

Appendix C. Open issues

These issues will be voted upon at the next meeting. If they are accepted, they will generally be put into the

spec.

(No issues are currently open.)

Appendix D. Miscellaneous

D.1. To be specified

[2006/10/20#3. Need to allow clients to access the members on a ResultAxis via a list (for convenience) and via an iterator. Iterators need to be restartable, but not bidirectional. Need to know the size of the axes, even if using the iterator interface.]

[2006/10/20#6. We discussed session support. It is necessary for write-back. JDBC's 'stateful session' is difficult to implement over a stateless protocol like HTTP. Michael suggested adding 'session name' as a parameter to 'execute' methods. Julian disagreed. No conclusion reached.]

[2006/10/20#7. We discussed the goals and intended audience of olap4j. The audience spans from a beginner's audience (only 2 hours experience with the API) who don't want to write a lot of code, to writers of clients (2 yrs experience with the API) who want performance and don't care how much code they need to write. Distributed clients (e.g. olap4j provider for XMLA) have bandwidth constraints. Mobile clients also have memory constraints.

ADOMD addressed beginners audience well, but used a lot of memory. Challenge is to support an object model (hence easy programming model) without increasing memory. No specific change to the specification, but decided to add memory efficiency as a design goal.]

D.2. Design notes

JDK

We are targeting JDK 1.5, and running retroweaver for backward compatibility for JDK 1.4. See forum thread: olap4j, JDK 1.5 and generics.

We also support JDK 1.6, and with it JDBC 4.0.

Result sets, random access, and memory usage

Should result sets return their axes as cursors or collections? Cursors require less memory, but collections provide an easier programming model.

Also on the subject of memory, how to represent the metadata? Schema result sets require less memory, are more flexible, and have better defined semantics in the presence of transactions and offline working; but an object model (Cube, Dimension, Level) provides an easier programming model.

Accessing cells

It would be possible to access cells in a result set (a) by ordinal; (b) by coordinates; (c) by the 'etchasketch' model determined by the position of the iterator along each axis, as used by JOLAP. We decided to support (a) and (b) but not (c). There are methods on CellSet to convert from ordinal to coordinates and vice versa.

If there is a huge number of cells, the client has limited memory, and bandwidth to the server is limited, random access to cells is costly. Michael suggested that we add a method `List<Cell> getCells(int startOrdinal, int endOrdinal)`, which matches XML/A behavior, but we declined to add it to the spec for now. John drew the analogy of a modern file system, implementing a serial access interface (streams) on top of random-access primitives. For now, we support only random access, but suggest that the provider looks for patterns of access.

Appendix E. References

1. XMLA: XML for Analysis Specification, version 1.1.

Appendix F. Change log

- Version 1.0.