olap4j Functional Specification

Author(s): Julian Hyde
Version: 0.5 (draft)
Last modified: September 1st, 2006.

Introduction

olap4j is an open Java API for building OLAP applications.

In essence, olap4j is to multidimensional data 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.

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.

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 scalar 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.

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.

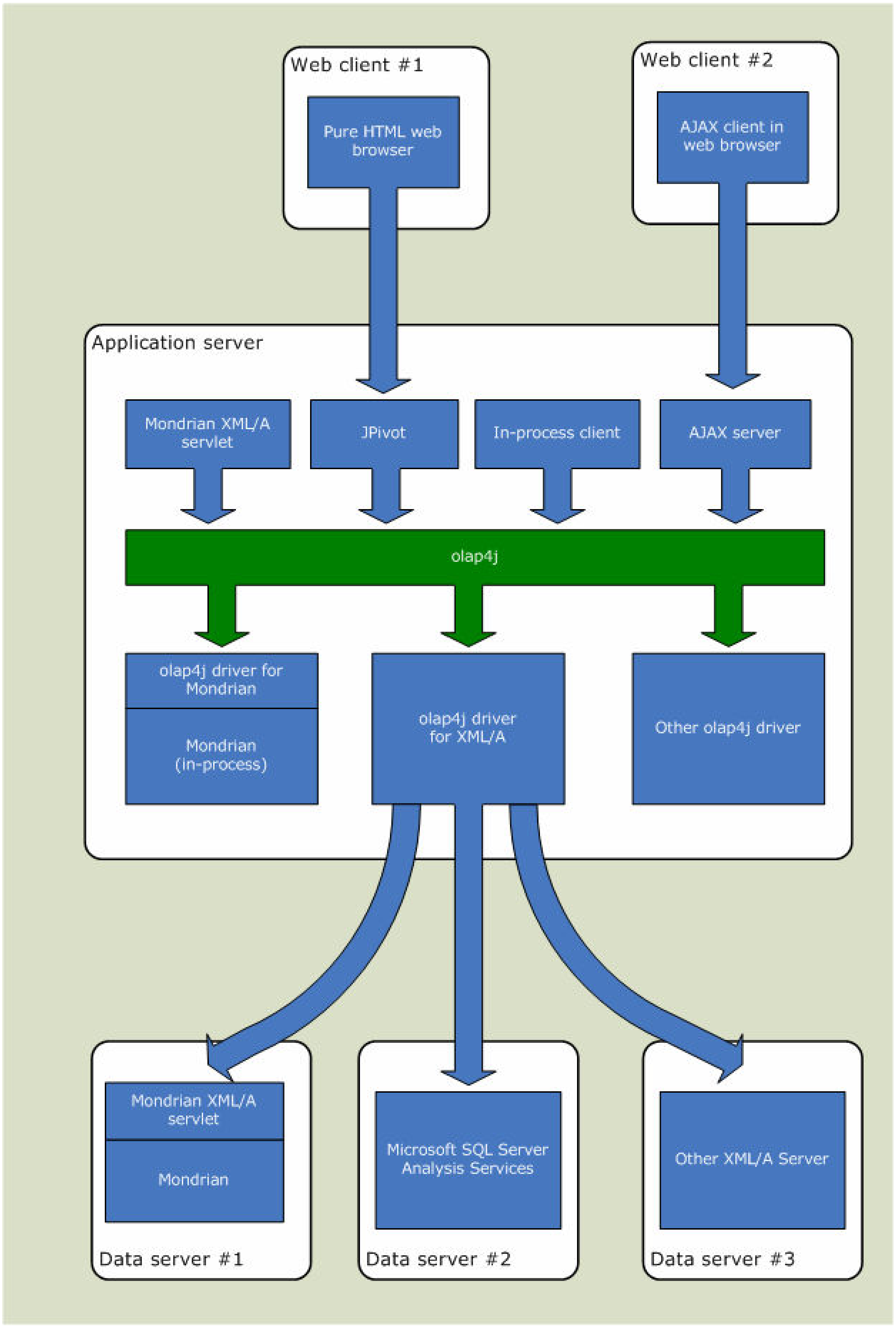
Benefits of a standard Java API for OLAP

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.

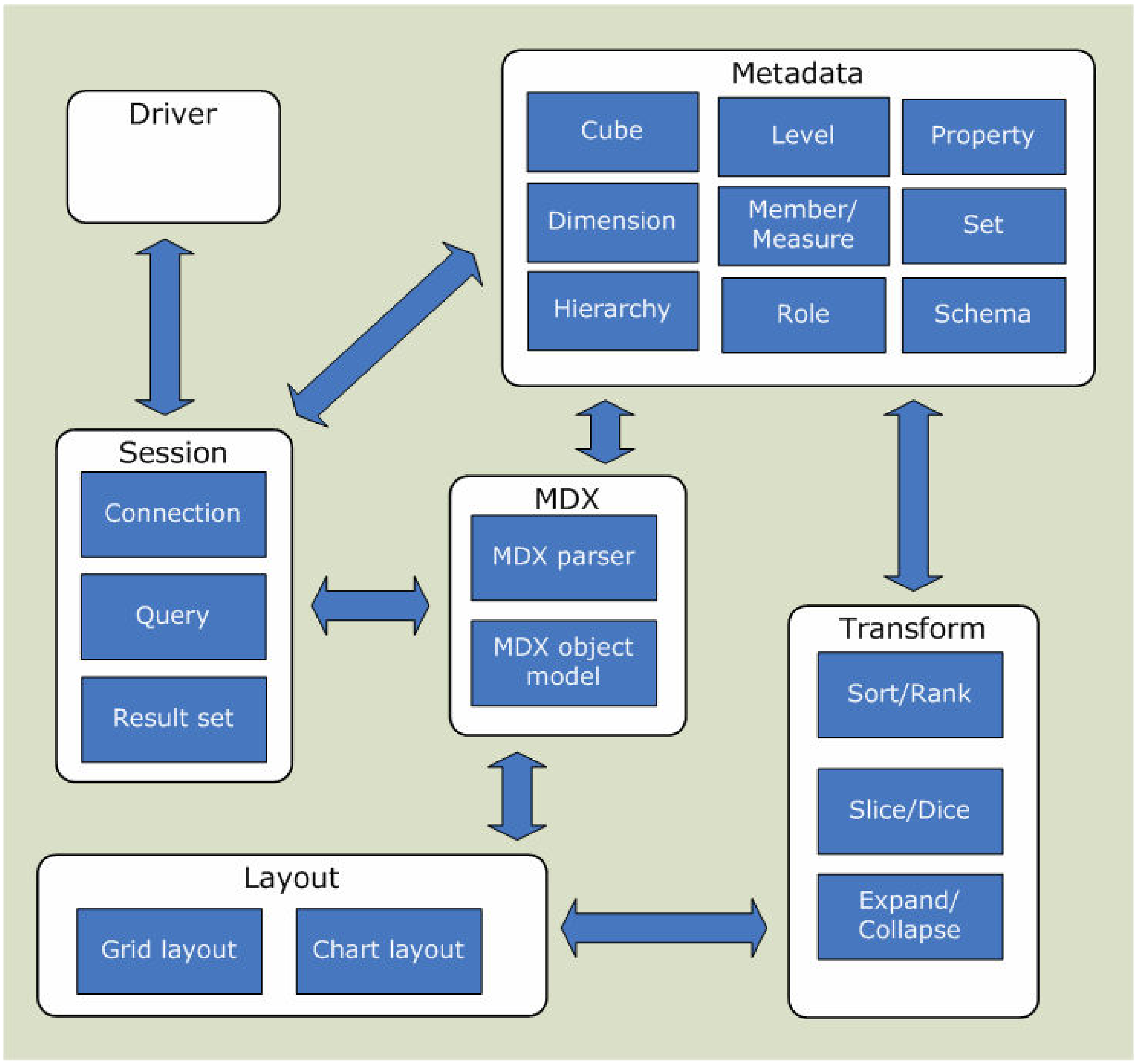
Architecture of olap4j

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
Data server #1
Microsoft SQL Server Analysis Services
Data server #2
Other XML/A Server
Data server #3

Components of the API

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



Driver

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

Session

olap4j's session 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 org.olap4j.*;

Class.forName("mondrian.olap4j.Driver");
OlapConnection connection = (OlapConnection)
    DriverManager.createConnection(
        "jdbc:mondrian:local:Jdbc=jdbc:odbc:MondrianFoodMart;" +
        "Catalog=/WEB-INF/queries/FoodMart.xml;" +
        "Role='California manager'");
OlapStatement statement = connection.createOlapStatement();

OlapResult result =
    statement.execute(
        "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 besides the driver class and connect string, the code is identical.

```
import org.olap4j.*;

Class.forName("olap4j.impl.xmla.Driver");
OlapConnection connection = (OlapConnection)
    DriverManager.createConnection(
        "jdbc:olap4jxmla:Server=http://localhost/xmla/msxisapi.dll;" +
        "Catalog=FoodMart");
OlapStatement statement = connection.createOlapStatement();

OlapResult result =
    statement.execute(
        "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.

Package name: `org.olap4j`

Major classes:

- OlapConnection (extends java.sql.Connection)
- Statement (extends java.sql.Statement)
- Parameter describes the parameters of a statement
- ResultSet
- ResultAxis
- Position

There is no support for JDBC's PreparedStatement.

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 *unvalidated* and *validated* state. In the unvalidated 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,

```
// Create a query model.
OlapConnection connection;
Query query = new Query();
query.setFrom("Sales");
query.getAxes().add(
    new Axis(
        "ROWS",
        false,
        new UnresolvedFunCall(
            "{}",
            Syntax.Special,
            new Id(new String[] {"Measures", "Unit Sales"})));

// Create a statement based upon the query model.
OlapStatement stmt;
```

```
try {
    stmt = connection.createOlapStatement(query);
} catch (OlapException e) {
    System.out.println("Validation failed: " + e);
    return;
}

// Execute the statement.
ResultSet rset;
try {
    rset = stmt.execute();
} catch (OlapException e) {
    System.out.println("Execution failed: " + e);
}
```

Package name: `org.olap4j.mdx`

Classes:

- Query
- Axis
- FunCall
- UnresolvedFunCall
- Id
- Literal
- MemberExpr
- LevelExpr
- HierarchyExpr
- DimensionExpr
- ParserFactory
- Parser

MDX type model

Represents the types of nodes in an MDX query model.

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.

Package name: `org.olap4j.mdx.type`

Classes:

- BooleanType
- CubeType
- DecimalType
- DimensionType
- HierarchyType
- LevelType
- MemberType
- NumericType
- ScalarType
- SetType
- StringType
- SymbolType
- TupleType
- Type
- TypeUtil

Metadata

The components of the OLAP model are available as read-only Java classes.

Access control

A user's view of metadata may be subject to access control. For example, a user may not have read access to certain hierarchies within a cube, or to certain members within a hierarchy. The API methods must behave consistently with access control.

Example: If Fred does not have access to the `[Nation]` level of the `[Store]` hierarchy, 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.

Package name: `org.olap4j.metamodel`

Classes:

- Schema
- Cube
- Dimension
- Hierarchy
- Level
- Member
- Measure
- Property
- Set
- Role

Transform

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]
```

Package name: `org.olap4j.transform`

Classes: TBD

Layout

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

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.

Test suite

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

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.

Non-functionality

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

- Schema reader parses an XML file to create a schema
- Cache management functions
- Ability to create/modify schema dynamically
- Definitions of MDX functions
- 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

Related projects

Mondrian provider

The Mondrian project will contain an implementation of the olap4j API based on the Mondrian OLAP server.

JPivot

JPivot currently based on two data sources: the Mondrian server, and a generic XML/A data source.

We intend to convert JPivot to run solely on the olap4j API. Connectivity to Mondrian and XML/A sources will be achieved by choosing the appropriate olap4j provider.

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.

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.

Date and Time types

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

(Richard Emberson, 2006/8/14)

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.

JDK version

See forum thread: olap4j, JDK 1.5 and generics. I am proceeding on the assumption that we are targeting JDK 1.5, and running retroweaver for backward compatibility for JDK 1.4.