Citra Pivot
Developer's Manual

version 1.0

Table of Contents

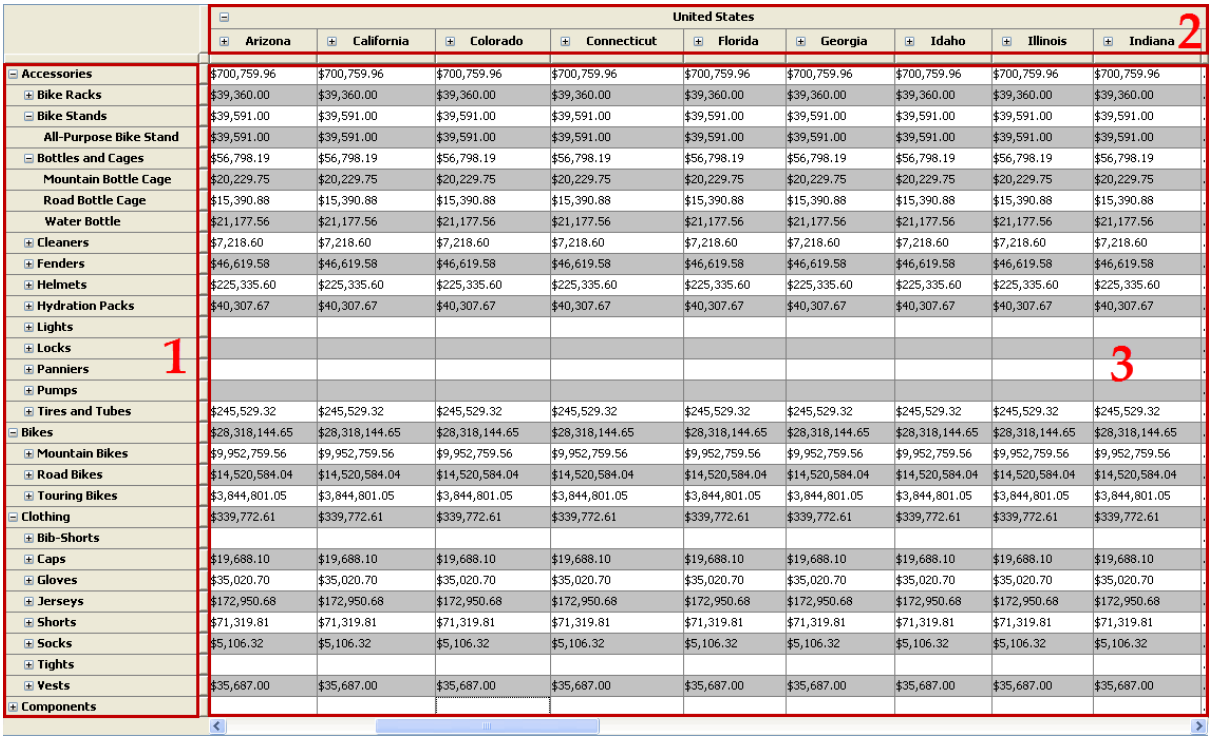
1 Preface

A pivot table is a component for summarizing, organizing and analyzing huge amounts of data. It is the most important visualization interface of an OLAP (OnLine Analytical Processing) system. There are hundreds of pivot table implementations, in various programming languages and architectures, such as C++, C#, Java, Eclipse, Delphi, Excel, Flash, Silverlight etc. Moreover, expensive solutions are offered by big companies, like Microsoft and Oracle. Lately, with the emergence of Rich Internet Applications (RIA), web-based pivot tables represent a large share of the market. Despite this fact, there is still a growing demand for traditional desktop pivot tables, especially in the Java Swing framework. We hope and feel that our Citra Pivot library, a powerful yet affordable pivot table solution, completely developed under Java/Swing, will successfully fill this requirement.

This manual intends to introduce Citra Pivot to developers, to describe and explain its main concepts. No matter how hard we tried to write it, we realize that there are still pieces missing, therefore under no circumstances should it be considered to be a complete description of all Citra Pivot's features and capabilities. The API documentation complements this manual, as well as our developer's forum at http://www.citra-tech.com/forum. For technical questions, we can also be reached via e-mail at support@citra-tech.com.

2 Pivot table

PivotTable is a JTable subclass that allows the pivoting of multi-dimensional data. Its accompanying table header is **PivotTableHeader**. The pivot table is split in three areas, the row, column and data area. Row and column areas are represented by a **PivotRowAdapter** and a **PivotColumnAdapter** respectively, the data area with a **PivotDataModel**, while all three areas are encapsulated in a **PivotTableModel**. The figure below shows a typical pivot table.



	United States								
	Arizona	California	Colorado	Connecticut	Florida	Georgia	Idaho	Illinois	Indiana
Accessories	$700,759.96	$700,759.96	$700,759.96	$700,759.96	$700,759.96	$700,759.96	$700,759.96	$700,759.96	$700,759.96
Bike Racks	$39,360.00	$39,360.00	$39,360.00	$39,360.00	$39,360.00	$39,360.00	$39,360.00	$39,360.00	$39,360.00
Bike Stands	$39,591.00	$39,591.00	$39,591.00	$39,591.00	$39,591.00	$39,591.00	$39,591.00	$39,591.00	$39,591.00
All-Purpose Bike Stand	$39,591.00	$39,591.00	$39,591.00	$39,591.00	$39,591.00	$39,591.00	$39,591.00	$39,591.00	$39,591.00
Bottles and Cages	$56,798.19	$56,798.19	$56,798.19	$56,798.19	$56,798.19	$56,798.19	$56,798.19	$56,798.19	$56,798.19
Mountain Bottle Cage	$20,229.75	$20,229.75	$20,229.75	$20,229.75	$20,229.75	$20,229.75	$20,229.75	$20,229.75	$20,229.75
Road Bottle Cage	$15,390.88	$15,390.88	$15,390.88	$15,390.88	$15,390.88	$15,390.88	$15,390.88	$15,390.88	$15,390.88
Water Bottle	$21,177.56	$21,177.56	$21,177.56	$21,177.56	$21,177.56	$21,177.56	$21,177.56	$21,177.56	$21,177.56
Cleaners	$7,218.60	$7,218.60	$7,218.60	$7,218.60	$7,218.60	$7,218.60	$7,218.60	$7,218.60	$7,218.60
Fenders	$46,619.58	$46,619.58	$46,619.58	$46,619.58	$46,619.58	$46,619.58	$46,619.58	$46,619.58	$46,619.58
Helmets	$225,335.60	$225,335.60	$225,335.60	$225,335.60	$225,335.60	$225,335.60	$225,335.60	$225,335.60	$225,335.60
Hydration Packs	$40,307.67	$40,307.67	$40,307.67	$40,307.67	$40,307.67	$40,307.67	$40,307.67	$40,307.67	$40,307.67
Lights									
Locks									
Panniers									
Pumps									
Tires and Tubes	$245,529.32	$245,529.32	$245,529.32	$245,529.32	$245,529.32	$245,529.32	$245,529.32	$245,529.32	$245,529.32
Bikes	$28,318,144.65	$28,318,144.65	$28,318,144.65	$28,318,144.65	$28,318,144.65	$28,318,144.65	$28,318,144.65	$28,318,144.65	$28,318,144.65
Mountain Bikes	$9,952,759.56	$9,952,759.56	$9,952,759.56	$9,952,759.56	$9,952,759.56	$9,952,759.56	$9,952,759.56	$9,952,759.56	$9,952,759.56
Road Bikes	$14,520,584.04	$14,520,584.04	$14,520,584.04	$14,520,584.04	$14,520,584.04	$14,520,584.04	$14,520,584.04	$14,520,584.04	$14,520,584.04
Touring Bikes	$3,844,801.05	$3,844,801.05	$3,844,801.05	$3,844,801.05	$3,844,801.05	$3,844,801.05	$3,844,801.05	$3,844,801.05	$3,844,801.05
Clothing	$339,772.61	$339,772.61	$339,772.61	$339,772.61	$339,772.61	$339,772.61	$339,772.61	$339,772.61	$339,772.61
Bib-Shorts									
Caps	$19,688.10	$19,688.10	$19,688.10	$19,688.10	$19,688.10	$19,688.10	$19,688.10	$19,688.10	$19,688.10
Gloves	$35,020.70	$35,020.70	$35,020.70	$35,020.70	$35,020.70	$35,020.70	$35,020.70	$35,020.70	$35,020.70
Jerseys	$172,950.68	$172,950.68	$172,950.68	$172,950.68	$172,950.68	$172,950.68	$172,950.68	$172,950.68	$172,950.68
Shorts	$71,319.81	$71,319.81	$71,319.81	$71,319.81	$71,319.81	$71,319.81	$71,319.81	$71,319.81	$71,319.81
Socks	$5,106.32	$5,106.32	$5,106.32	$5,106.32	$5,106.32	$5,106.32	$5,106.32	$5,106.32	$5,106.32
Tights									
Vests	$35,687.00	$35,687.00	$35,687.00	$35,687.00	$35,687.00	$35,687.00	$35,687.00	$35,687.00	$35,687.00
Components									

The following regions are identified:

1: Row header area
2: Column header area
3: Data area

The appearance of the pivot table can be customized with the help of specialized renderers, **PivotRowHeaderRenderer** for the row area and **PivotColumnHeaderRenderer** for the column area, as well as with a **PivotStyleModel**. Localization is also possible via **PivotResourceManager**.

2.1 Row and column areas

The row and column areas of a pivot table are defined with **PivotRowAdapter** and **PivotColumnAdapter**, respectively, the first being a TableModel, while the second a TableColumnModel. In the JTable framework, a TableModel is used as the table's data model and a TableColumnModel as the table's header column model. However, this applies to a two-dimensional data model. In the multi-dimensional case, this notion is extended by chaining models (adapters) together.

Data structures pivoted against a horizontal and vertical axis represent dimensions. The first dimension is specified by a single adapter, the second with a number of adapters equal to the number of entities in the first adapter, the third dimension with a number of adapters equal to the number of entities in the second adapter, and so on. The data model of each adapter contains a tree structure, extending the **TreeModel** interface. Therefore, in dimensions other than the first, adapters are created for every node in the previous adapter. In order to retrieve the next dimension's adapter at a given position, the following methods are used:

public PivotRowAdapter getAdapter(TreePath path): for the row header
public PivotColumnAdapter getAdapter(TreePath path): for the column header

Also, the adapter depth is defined as the total number of dimensions (and thus adapters) below the one examined. The following method retrieves an adapter's depth:

public int getAdapterDepth()

In addition, each node has a so-called pivot type, that specifies whether it is a normal, subtotal or grand total node. The type is used by visualization classes, such as Pivot Table's SpanDrawer, that merges summary cells together, as needed. The type is retrieved with:

public int getPivotType(TreePath path)

Concrete implementations used throughout the library are DefaultPivotRowAdapter and DefaultPivotColumnAdapter, that are constructed with a number of TreeTableModels and TreeTableColumnModels respectively, that each define the tree data structure to be used for each dimension.

2.2 Data area

The data area is represented by a **PivotDataModel**. In a two-dimensional TableModel, a cell is defined by only two parameters, the row and column index. In the multi-dimensional case of pivot tables, as many parameters are required as there are models (dimensions) in the pivot. Since, in PivotDataModel, models are effectively TreeModels, the row and column indices are replaced with TreePath arrays. Therefore, the method that determines a cell's value changes accordingly.

More specifically:

TableModel

dimensions: two
parameters: row and column index
method: public Object getValueAt(int rowIndex, int columnIndex)

PivotDataModel

dimensions: multiple
parameters: row and column TreePath arrays
method: public Object getValueAt(TreePath[] rowPaths, TreePath[] columnPaths)

The TreePath arguments completely identify a cell in the data model. The length of each path array matches the number of models in each area (row and column).

Example: Identifying a cell in a PivotDataModel

For a cell with row and columns indexes **row** and **column** respectively, in the **dataModel**:

```
//get the first row adapter
PivotRowAdapter rowAdapter = dataModel.getPivotRowAdapter();

//get the row paths for the row index
TreePath[] rowPaths = PivotUtils.getPathsForRow(
                                        row, dataModel.getPivotRowAdapter());

//iterate over the row values
for (int i=0;i<rowPaths.length;i++) {
     //get the node for this path
     Object node = rowPaths[i].getLastPathComponent();
     //get the value of the model at this location
     Object value = rowAdapter.getTreeTableModel().getValueAt(node, 0);
     //get the next adapter
     rowAdapter = rowAdapter.getAdapter(rowPaths[i]);
}

//get the first column adapter
PivotColumnAdapter columnAdapter = dataModel.getPivotColumnAdapter();

//get the column paths for the column index
TreePath[] columnPaths = PivotUtils.getPathsForColumn(
                                        column, dataModel.getPivotColumnAdapter());

//iterate over the column values
for (int i=0;i<columnPaths.length;i++) {
     //get the node for this path
     Object node = columnPaths[i].getLastPathComponent();
```

```
    //get the value of the model at this location
    Object value = columnAdapter.getTreeTableColumnModel().
                                        getColumn(node).getHeaderValue();

    //get the next adapter
    columnAdapter = columnAdapter.getAdapter(columnPaths[i]);
}
```

2.3 PivotTableModel

PivotTableModel is Pivot Table's data model. There are three areas used in pivoting, the row header, column header and data area. PivotTableModel consists of three sub-models, each corresponding to these areas:

PivotRowAdapter: the model of the table's row header
PivotColumnAdapter: the model of the table's column header
DataModel: the model of the data area

DefaultPivotTableModel is the default PivotTableModel implementation that requires a PivotDataModel in its constructor.

2.4 Appearance

The table appearance can be easily customized. The cells of the row and column areas are rendered with specialized renderers, **PivotRowHeaderRenderer** and **PivotColumnHeaderRenderer**. These renderers return suitable components, showing expand/collapse icons that allow the discovery of the tree structures being displayed. The component for a PivotRowHeaderRender is returned with:

```
public Component getTreeTableCellRendererComponent(
        JTable table,
        Object value,
        boolean isSelected,
        boolean hasFocus,
        int row,
        int column,
        boolean isLeaf,
        boolean isExpanded,
        PivotRowAdapter adapter,
        TreePath path,
        TableCellRenderer cellRenderer)
```

A similar method is used in the case of a PivotColumnHeaderRenderer:

```
public Component getTreeTableCellRendererComponent(
            JTable table,
```

Object value,
boolean isSelected,
boolean hasFocus,
int row,
int column,
boolean isLeaf,
boolean isExpanded,
TreeTableColumnModelAdapter adapter,
TreePath path,
TableCellRenderer cellRenderer,
int level)

In addition to renderers, a **PivotStyleModel** changes typical component attributes, such as the font and the background and foreground colors. Its default implementation is a DefaultPivotStyleModel that uses a PivotStyle for the aforementioned purpose. Unlike renderers, the pivot style is applied to all three pivot areas with the methods:

```
public void applyDataStyle(Component c, PivotTable table,
                           PivotRowAdapter[] rowAdapters, TreePath[] rowPaths,
                           PivotColumnAdapter[] columnAdapters, TreePath[] columnPaths,
                           int row, int column)

public void applyColumnHeaderStyle(Component c, PivotTable table,
                           PivotColumnAdapter adapter, TreePath path,
                           int level, int column)

public void applyRowHeaderStyle(Component c, PivotTable table,
                           PivotRowAdapter adapter, TreePath path,
                           int row, int column)
```

2.5 Pivot components

PivotTable is the component that allows the pivoting of multi-dimensional data. Its accompanying JTableHeader is **PivotTableHeader**. In a pivot table, there are three distinct areas. These are the column area (the horizontal axis), the row area (the vertical axis) and the intersection of the two, represented by the data area. These three areas are completely specified by the table's model, which is a **PivotTableModel**. As PivotTable extends JTable, and also our Citra Table component, AdvancedJTable, it will inherit all methods and properties of both superclasses.

PivotTable creates and uses a single PivotRowHeaderRenderer for each row header cell. It is created with PivotTable's method:

protected PivotRowHeaderRenderer createDefaultPivotRowRenderer()

Also, PivotStyleModel is a property of PivotTable that is created with:

protected PivotStyleModel createDefaultPivotStyleModel()

PivotTableHeader creates and manages a number of PivotColumnHeaderRenderer instances, each for every level in the column header area. The renderers are created with PivotTableHeader's method:

protected PivotColumnHeaderRenderer createDefaultPivotColumnRenderer()

2.6 Localization

Localization in our pivot table library is materialized through PivotResourceManager. PivotResourceManager uses a ResourceBundle instance which contains String key/value pairs for the various texts used throughout the library. By default, a resource bundle is created with the current locale, which loads the accompanying property file, named CitraPivotBundle. properties. By modifying the contents of this file, or by creating a new one in the format CitraPivotBundle_[locale].properties, developers may customize the language of the various Citra Pivot components, like OlapAssistant and OlapFilterPanel. The resource bundle can be assigned with:

public static void setResourceBundle(String resource)
public static void setResourceBundle(String resource, Locale locale)

Assigning a new resource bundle, will have no effect on the components that are currently displayed in the application. For updating the various text strings, the UI needs to be updated and the PivotResourceManager's **updateComponents** variable needs to be set to true.

Example: Update the application with a new resource bundle

```
//assign a new resource bundle
PivotResourceManager.setResourceBundle("c:\\resources\\MyResource.properties");

//set update components variable to true
PivotResourceManager.setUpdateComponents(true);

//get the application frame
Frame frame = JOptionPane.getFrameForComponent(pivotTable);

//update the frame's UI
SwingUtilities.updateComponentTreeUI(frame);

//set update components variable back to false - optional
PivotResourceManager.setUpdateComponents(false);
```

3 Olap Metadata

Citra Pivot's function is to display multidimensional data. Before an attempt is made to show how this is achieved, it is important to describe what is meant by multidimensional data, and their relation to each other. In fact, data in an OLAP source carry descriptive information about the OLAP structure, and are thus called metadata. These structural metadata are organized in cubes, dimensions, hierarchies, levels, members and measures, all of which define the OLAP schema. Next, each of these entities will be described in detail.

3.1 OlapObject

OlapObject is the base class for all olap metadata objects. It defines methods for retrieving the name, id, caption and description of the object under consideration. More specifically:

getID(): returns the object's unique identifier. This value is unique in the whole schema.
getName(): returns the object's name.
getCaption(): returns the object's caption. This value is used by default when pivoting hierarchies.
getDescription(): returns the object's description.

3.2 OlapSchema

OlapSchema is the top container for cubes and dimensions. There are two methods defined:

List getCubes(): returns a list of cubes
List getDimensions(): returns a list of dimensions

The dimensions returned by the schema are shared between its cubes.

3.3 OlapCube

OlapCube is part of a schema and consists of a number of dimensions and measures. It contains the following methods:

List getDimensions(): returns the dimensions of the cube
OlapDimension getMeasureDimension(): returns the measure dimension
List getMeasures(): returns the measures of the cube
OlapSchema getSchema(): returns the cube’s schema
List getSupportedLocales(): returns a list of the supported Locales

3.4 OlapDimension

OlapDimension represents an organized data structure. Data belonging to the same dimension have similar properties, e.g. a Customer dimension may contain information about a

company's clientele, like country, city, town etc. The way data are categorized within a dimension, is called a hierarchy. A dimension must have at least one hierarchy and also a default one. A special type of dimension, called the measure dimension, is part of every cube, and contains the cube's measures. The following methods are defined:

OlapHierarchy getDefaultHierarchy(): returns the default hierarchy
List getHierarchies(): returns the hierarchies of the dimension
boolean isMeasureDimension(): returns true if the dimension is the measure dimension

3.5 OlapHierarchy

OlapHierarchy represents an organization of data within a dimension. Each hierarchy contains a number of unique categories, called levels. Members within the same level have the same properties and attributes. For example, a Year level contains years. At least one level must exist and also a default member must be defined. OlapHierarchy defines the following methods:

OlapMember getDefaultMember(): returns the default member of the hierarchy
OlapDimension getDimension(): returns the dimension of the hierarchy
List getLevels(): returns the levels of the hierarchy
boolean hasAll(): returns true if the hierarchy has a single member at the root level

3.6 OlapLevel

OlapLevel is a data structure that contains similar members, each at the same depth from the root of the hierarchy. There is only one method defined:

OlapHierarchy getHierarchy(): returns the hierarchy of the level

3.7 OlapMember

OlapMember represents a data value in a level, hierarchy or dimension. Its methods are:

OlapLevel getLevel(): returns the level of the member
int getMemberCount(): returns the number of children this member has

3.8 OlapMeasure

OlapMeasure is a special case of an OlapMember, that represents a calculation within a cube. Measures belong to a special type of dimension, called the measure dimension. The type of measure depends on the evaluation method used in order to produce it. It defines the following methods:

OlapType getDataType(): returns the olap type of the measure's value

Class getJavaDataType(): returns the type of the measure's value as a Java Class object

3.9 OlapType

OlapType describes the type of a given measure value. There are nine different data types that relate to the datatypes defined in the OLE DB system. They are:

INTEGER
DOUBLE
CURRENCY
BOOLEAN
VARIANT
UNSIGNED_SHORT
UNSIGNED_INTEGER
LARGE_INTEGER
STRING

There is one method defined:

String getName(): returns the type’s name in the OLE DB specification

4 Interfacing with olap data

Interfacing with OLAP servers is materialized with classes found in the **com.citra.pivot.olap.data** package. In fact, **DataSource** represents a link to a multidimensional storage, be it a traditional OLAP database, like Microsoft SQL Analysis Server, or a in-memory repository. Queries are made by executing an **OlapSelection** from a **Connection** instance that is retrieved from the datasource. The data that is returned is encapsulated into various types, such as OlapTuple, OlapSet, OlapCell, OlapCellSet and TabularSet, according to the query. Furthermore, an **OlapCursor** can be used to iterate over the queried data. Finally, in the case of errors, a **DataSourceException** is thrown.

4.1 DataSource

In order to create a link to an OLAP server, a DataSource is used. DataSource is a Java interface that, at the time of writing, has two concrete implementations: Olap4jDataSource, for connecting to OLAP servers with the olap4j driver using the XML/A protocol, and TableDataSource, which is a in-memory multidimensional storage, based on a two-dimensional tabular model. The schema of the datasource is retrieved with:

public OlapSchema getSchema()

Also, for making queries, a datasource connection is required which is created with the method:

public Connection createConnection()

4.2 Connection

A Connection instance is used to access the OLAP server's data and metadata. In order to do so, the following method should be called:

public Object execute(OlapSelection selection) throws DataSourceException

Depending on the type and parameters of the OlapSelection, different objects are returned. OlapSelection is described in the subsequent section.

Once a connection has been created, it will stay in memory until it is closed explicitly with the method:

public void close() throws DataSourceException

Therefore, always remember to close the connection after it is no longer needed, so that resources can be released.

Other Connection methods are:

public OlapCube getCube(): returns the cube the connection will run queries against
public void setCube(OlapCube cube): assigns the cube the connection will run queries against
public Locale getLocale(): returns the current java locale that is applied to the queried data
public void setLocale(Locale locale): assigns the current java locale
public OlapSchema getSchema(): returns the connection's schema

4.3 OlapSelection

OlapSelection is an interface that is used for selecting objects from a DataSource. The selection can return a list of members, data values, query results or any Java Object. There is only one method defined for that purpose:

public Object resolveSelection(Connection connection) throws DataSourceException

Depending on the return object, OlapSelections are further categorized into sub-interfaces that also include an additional method for returning a more specific object. These are:

AxisSelection: returns an AxisOlapSet
LogicalSelection: returns a Boolean
MemberSelection: returns an OlapMember
QuerySelection: returns an OlapCellSet
SetSelection: returns an OlapSet
StringSelection: returns a String
TabularSelection: returns a TabularSet
TupleSelection: returns an OlapTuple

When constructing an OlapSelection, a number of arguments is supplied that determine the selection output. For example, SingleSetSelection takes an OlapMember and an operator as arguments and returns its children, siblings, parent, descendants, ancestors or leaf members, according to the operator. OlapSelections can also be nested in order to produce more complex results. As is the case with OrderSetSelection, that takes a SetSelection, a sort operator and a measure as arguments and returns an ordered set. For a complete list and a description of each OlapSelection implementation, please refer to Citra Pivot's API documentation.

4.4 CustomSelection

For creating custom user-defined selections, a CustomSelection should be used, which is an OlapSelection sub-interface. If a selection is constructed merely by implementing OlapSelection, then a DataSourceException will be thrown when the DataSource attempts to resolve it.

4.5 OlapTuple

OlapTuple is a container for OlapMembers. A tuple may contain any number of members, with the only limitation that each belongs to a different dimension. The tuple's dimensionality defines the number of members in the tuple, and thus the number of unique dimensions:

public int getDimensionality()

The list of members is retrieved with:

List getMembers()

OlapTuple's concrete implementation is **DefaultOlapTuple**, that contains methods for adding members to it.

4.6 OlapSet

OlapSet is a container of OlapTuples. A set may contain any number of tuples, however each tuple must have the same dimensionality. In order to retrieve a set's dimensionality:

public int getDimensionality()

Also, the set's tuples are retrieved with:

public List getTuples()

OlapSet's concrete implementation is **DefaultOlapSet**, that contains methods for adding tuples to it.

4.7 OlapCell

OlapCell represents a calculated value in a cube. A value is produced for every possible combination between the cube's dimensions. Typically, OLAP servers precompute this calculation after the cube's creation. OlapCell encapsulates this information, which can be retrieved with the methods:

public Object getValue(): returns the calculated value
public String getFormattedValue(): returns the calculated value as this has been formatted into a string
public OlapMeasure getMeasure(): returns the measure with which the value is associated

Note that OlapCell's concrete implementation is **DefaultOlapCell**.

4.8 Cursors

Cursors are used in order to iterate through query results. OlapCursor is the base cursor class, TupleCursor is used for OlapTuples, ValueCursor for OlapCells and TabularCursor for two dimensional data.

4.8.1 OlapCursor

OlapCursor defines methods for iterating over sets of data:

public int getPosition(): gets the current cursor position
public int getPositionCount(): gets the total number of the cursor's positions
public boolean next(): moves to the next position and returns true if there are more results
public void reset(): moves the cursor to the beginning
public void setPosition(int): moves the cursor to a given position

4.8.2 TupleCursor

TupleCursor is a cursor that retrieves and assigns olap tuples at a given position. It defines the following methods:

public OlapTuple getCurrentTuple(): returns the tuple at the current position
public void setCurrentTuple(OlapTuple): assigns the tuple at the current position

4.8.3 ValueCursor

ValueCursor is a cursor that retrieves and assigns olap cell values at a given position. It has the methods:

public OlapCell getCurrentValue(): returns the cell at the current position
public void setCurrentValue(OlapCell): assigns the cell at the current position

4.8.4 TabularCursor

TabularCursor is a cursor that retrieves data in tabular form. It has the methods:

public int getColumnCount(): returns the number of columns in the set
public String getColumnName(int index): returns the column name at a given location
public Object getValue(int index): returns the column value at a given location of the current row in the set

4.9 AxisOlapSet

AxisOlapSet contains tuples that are assigned to a given axis. The axis index or ordinal is retrieved with:

public int getAxisOrdinal()

For iterating over the tuples, a TupleCursor is used which is retrieved with:

public TupleCursor getTupleCursor()

Note that the concrete AxisOlapSet implementation is **DefaultAxisOlapSet**.

Example: Iterate over an AxisOlapSet to discover the returned tuples

Having **axis** as the AxisOlapSet:

```
//get the tuple cursor
TupleCursor tupleCursor = axis.getTupleCursor();

//iterate over tuples
while (tupleCursor.next()) {
        //get the current tuple
        OlapTuple tuple = tupleCursor.getCurrentTuple();
        //print out
        System.out.println("members:" + tuple.getMembers());
}
```

4.10 OlapCellSet

OlapCellSet is the result of an olap query, that contains a number of OlapCells, as well as metadata information about the values returned. When building a query, tuples are assigned to different axes. The result from the query contains the same axis assignment, which is encoded in an AxisOlapSet. These axes are retrieved with:

public List getAxisSets(): returns a list of AxisOlapSets

Every axis set can be probed individually, via its TupleCursor, in order to discover the returned tuples on each of them.

The query may optimally include a filter, which will also be present in the result. To find out the filter members, the appropriate tuple cursor can be retrieved:

public TupleCursor getFilterCursor(): returns the cursor that contains the filter members, if any

Finally, the values are retrieved using a ValueCursor, with the method:

public ValueCursor getValueCursor(): returns the cursor that contains the values of the query result

Note that OlapCellSet's concrete implementation is **DefaultCellSet**.

Example: Iterating over an OlapCellSet to discover the returned values

Having **result** as the OlapCellSet:

```
//get the value cursor
ValueCursor values = result.getValueCursor();

//get the filter cursor
TupleCursor filter = result.getFilterCursor();

//get filter tuple
OlapTuple filterTuple = null;
if (filter.next()) {
        filterTuple = filter.getCurrentTuple();
}

//iterate over values
while (values.next()) {
        //get the current cell
        OlapCell cell = values.getCurrentValue();
        //get the current tuple
        OlapTuple tuple = values.getCurrentTuple();
        //compile a list of members
        List members = new ArrayList();
        members.addAll(tuple.getMembers());
        if (filterTuple != null) members.addAll(filterTuple.getMembers());
        //print out
        System.out.println("value:" + cell.getValue() + ", formattedValue:" +
                                cell.getFormattedValue() + ", members:" + members);
}
```

4.11 TabularSet

TabularSet is a container for a two-dimensional set of data. The values of the set can be iterated over with a TabularCursor:

public TabularCursor getTabularCursor()

TabularSet's concrete implementation is **DefaultTabularSet**.

Example: Iterating over a tabular set

Having **set** as the TabularSet:

```
//get the tabular cursor
TabularCursor cursor = set.getTabularCursor();
```

```
//get the number of columns
int columnsNum = cursor.getColumnCount();

//add the column names to a vector
Vector columns = new Vector();
for (int i=0;i<columnsNum;i++) {
        columns.add(cursor.getColumnName(i));
}

//create a vector for the rows
Vector rows = new Vector();

//iterate over the cursor to get the rows
while (cursor.next()) {
        Vector row = new Vector();
        for (int i=0;i<columnsNum;i++) {
                row.add(cursor.getValue(i));
        }
        //add the row to the rows vector
        rows.add(row);
}

//create a table model from the rows and columns vector
DefaultListTableModel model = new DefaultListTableModel(rows, columns);
```

The model can then be displayed in a table.

4.12 Exceptions

DataSource operations throw a DataSourceException, that identifies the error in the underlying data connection or query arguments. In addition, OlapCursors throw a OlapCursorException. The cause of these exceptions is also assigned for further inspection of the problem.

5 Pivoting olap data

The **com.citra.pivot.olap** package provides an olap implementation of PivotTable. It contains a number of objects that are extensions of those found in the **com.citra.pivot** package. These extended objects include methods that refer to olap data and metadata. Thus, **OlapRowAdapter** extends PivotRowAdapter, **OlapColumnAdapter** extends PivotColumnAdapter, **OlapPivotTable** extends PivotTable, **OlapDataModel** is a PivotDataModel implementation, **OlapCellRenderer** is a TableCellRenderer for olap cells and **DefaultOlapStyleModel** is a PivotStyleModel based on OlapCells. OlapRowAdapter's and OlapColumnAdapter's model is an **OlapTableModel** that extends the TreeTableModel interface.

OlapDataModel depends on an **OlapProvider**, that creates the pivoted OlapTableModels, as well as the row, column and data areas. The way a cube is pivoted is described by an **OlapDefinition**, that defines the pivot area and order of a given olap hierarchy. Via the concrete OlapProvider implementation, **DefaultOlapProvider**, certain common olap operations are possible. In fact, filtering is defined with an **OlapFilter**, sorting with an **OlapSort**, getting the top or bottom values in a hierarchy can be achieved with **OlapTopBottom**. The provider also has more functions, such as showing empty/non-empty cells, visual totals, grand and subtotals, swapping the axes, member drilling and more.

5.1 OlapTableModel

OlapTableModel is the model that is used for pivoting olap data. It defines essentially a tree structure with nodes that contain an OlapMember and/or an OlapMeasure. OlapTableModel represents members belonging to the same hierarchy. The hierarchy can be retrieved with:

public OlapHierarchy getOlapHierarchy()

The value that is displayed on each node is determined and assigned with:

public String getName(Object node)
public void setName(String name, Object node)

Each node has a pivot type, depending on the nature of the member at that node. There are three different types, taken from **PivotConstants**:

NORMAL_TYPE: the default behaviour
SUBTOTAL_TYPE: identifies that a member is showing a subtotal
GRAND_TOTAL_TYPE: identifies that a member is showing a grand total

The type can be retrieved with the **getOlapType(Object node)** method.

The olap member at a given node is determined with:

public OlapMember getOlapMember(Object node)

In addition to an olap member, a node can also contain a measure. In that case, the default measure in the pivot table is overridden. For example, the node with member Beverages gives the value of **Beverages** with the default measure. On the other hand, a node with member **Beverages** and measure **Max Sales**, yields the value of **Beverages** with the **Max Sales** measure. The measure defined on a node is retrieved with:

public OlapMeasure getOlapMeasure(Object node)

DefaultOlapTableModel is the default OlapTableModel implementation. The model can be constructed automatically, given a DataSource's Connection and a hierarchy or member, or manually by supplying only the model's name or hierarchy. In the first case, nodes will be discovered on demand, as they are needed, by retrieving each node's children members. In the second, developers should add each node programmatically. For doing so, the various **addEntry** and **insertEntry** methods may be used. The model is being discovered by taking each node's children, starting from the root. The root member will be excluded from the view, and therefore, in order to show the 'all' member of a hierarchy, this needs to be attached to a special root member, called **OlapRootMember**, which is taken with the static method **OlapRootMember.getRootMember(OlapHierarchy hierarchy)**. Finally, In a DefaultOlapTableModel, the member, measure, name and type of a node is stored in an OlapEdge.

Example: Creating olap models

Having **connection** as the datasource's connection:

```
//get a member to use it as the model's root member
OlapMember time2000 = OlapUtils.retrieveMember(
                                new String[] {"Time", "All Years", "2000"}, connection);

//create a model having time2000 as root
DefaultOlapTableModel model_1 = new DefaultOlapTableModel(time2000, connection);

//create a model showing the whole hierarchy tree
DefaultOlapTableModel model_2 = new DefaultOlapTableModel(
                        OlapRootMember.getRootMember(time2000.getLevel().getHierarchy(),
                        connection);

//create a model of the time hierarchy by hand
DefaultOlapTableModel model_3 = new DefaultOlapTableModel(
                        OlapRootMember.getRootMember(time2000.getLevel().getHierarchy());

//add the '2000' year to the tree
MutableTreeNode time2000node = model_3.addEntry(time2000);

//get the first quarter
OlapMember timeQ1 = OlapUtils.retrieveMember(
```

```
                    new String[] {"Time", "All Years", "2000", "Q1-2000"}, connection);

//insert the quarter as a child of the 2000 node
model_3.insertEntry(timeQ1, time2000node);
```

5.2 Olap adapters

Pivot adapters for olap data are OlapRowAdapter for the row area, and OlapColumnAdapter for the column area. Olap adapters have a more suitable model for olap data than a TreeModel, OlapTableModel. In addition, they define a method for retrieving their own type at a given tree path:

```
public OlapRowAdapter getOlapAdapter(TreePath path)
public OlapTableModel getOlapModel()
```

5.3 OlapDefinition

OlapDefinition describes the location and order of a cube's hierarchy on a pivot table. The location is defined by a pivot type. Valid locations are the row, column and filter area, each having a pivot type of ROW_TYPE, COLUMN_TYPE and FILTER_TYPE respectively, whereas a non-pivoted hierarchy has a type of UNDEF_TYPE. In addition, OlapDefinition maintains an index for each hierarchy, identifying its position in the pivot area. There are methods for changing and retrieving the pivot type and index of a hierarchy.

Example: Creating and managing an OlapDefinition

```
//create an olap definition given an olap cube
OlapDefinition def = new OlapDefinition(cube);

//find the time hierarchy
OlapHierarchy time = OlapUtils.findHierarchyByName(cube, “Time”);

//pivot the time hierarchy in the column area
def.setColumnHierarchy(time);

//find the products hierarchy
OlapHierarchy products = OlapUtils.findHierarchyByName(cube, “Products”);

//pivot the products hierarchy in the row area
def.setRowHierarchy(products);

//find the customers hierarchy
OlapHierarchy customers = OlapUtils.findHierarchyByName(cube, “Customers”);

//pivot the customers hierarchy in the row area, before products
def.setRowHierarchy(products, 0);
```

5.4 OlapProvider

OlapProvider supplies all the information that is needed to display a cube on a pivot table, based on olap data. It constructs and maintains the row and column areas that are returned with the **getRowAdapter** and **getColumnAdapter** methods respectively. The cube it represents is returned with the **getCube** method. In order to build the data models and adapters, a DataSource's connection is required, that is returned with the **getConnection** method. In addition, it defines a filter selection, retrieved with **getSlicingMembers**. Finally, a default measure is used, when the measure dimension is not present in the pivot table, returned with **getDefaultMeasure**. Its default implementation, widely used in the library, is DefaultOlapProvider, that will be described next.

5.4.1 DefaultOlapProvider

DefaultOlapProvider is the default OlapProvider implementation. It takes care of creating the OlapRowAdapters and OlapColumnAdapters required by OlapDataModel. The way a cube is pivoted is described by an OlapDefinition. DefaultOlapProvider includes a number of methods that modify the data and structure of the pivot table. These methods may be accessed programmatically or via the graphical olap components of the **com.citra.pivot.olap.gui** package. Data is loaded on demand, as they are needed, depending on the cells that are being displayed. DefaultOlapProvider makes an effort to minimize query calls, by requesting an optimal amount of values with a single database call.

5.4.1.1 Pivoting

Hierarchies can be pivoted in one of the three areas, the row, column or filter area. The pivoting of hierarchies is defined in an OlapDefinition. The olap definition currently in use is retrieved with:

public OlapDefinition getCompiledOlapDefinition()

This definition will change only when a new one is assigned with:

public void setOlapDefinition(OlapDefinition olapDefinition)
throws DataSourceException

In order to modify the pivoted hierarchies, the current definition should first be retrieved with either the **getCompiledOlapDefinition** or the **getOlapDefinition** methods. After modification, the altered definition can be applied with **setOlapDefinition**. If an error occurs, a DataSourceException will be thrown.

Example: Altering an olap definition

```
//retrieve the current olap definition
OlapDefinition def = provider.getOlapDefinition();
```

```
//get the time hierarchy
OlapHierarchy timeHier = OlapUtils.findHierarchyByName(def.getCube(), “Time”);

//pivot the time hierarchy in the row area
def.setRowHierarchy(timeHier, OlapDefinition.ROW_TYPE);

//get the products hierarchy
OlapHierarchy productsHier = OlapUtils.findHierarchyByName(def.getCube(), “Products”);

//pivot the products hierarchy in the column area
def.setColumnHierarchy(productsHier, OlapDefinition.COLUMN_TYPE);

//apply the new definition
try {
  provider.setOlapDefinition(def);
} catch (DataSourceException e) {
  e.printStackTrace();
}
```

5.4.1.2 Sorting

Olap members in a hierarchy can be sorted either in ascending or descending order, based on the values of a measure or on their names in the hierarchy. The method used for that purpose is:

public void setSort(OlapLevel level, int mode, OlapMeasure measure, OlapTuple tuple)

Sorting information is encapsulated in an **OlapSort** object. The method arguments are:

level: the olap level in the hierarchy that we would like to sort
mode: constant in **OlapSort** specifying the sort mode, one of **NO_SORT**, **ASC_SORT** or **DESC_SORT**
measure: the measure based on which the members are sorted. If this is null, members are sorted using their names.
tuple: a reference tuple on which the sort is additionally based, only valid if measure argument is not null

Example: Sorting in a time hierarchy

```
//in a time hierarchy, get the second level, i.e. year level
OlapLevel yearLevel = (OlapLevel ) time.getLevels().get(1);

//sort in descending order based on the natural names of the year level members
provider.setSort(yearLevel, OlapSort.DESC_SORT, null, null);

//sort in ascending order based on the sales values of the year level members
provider.setSort(yearLevel, OlapSort.ASC_SORT, sales, null);

//construct a tuple having (Beverages, Greece) as its members
```

```
DefaultOlapTuple tuple = new DefaultOlapTuple();
tuple.addMember(beverages);
tuple.addMember(greece);

//sort in ascending order based on the sales values of the year level members
//AND the (Beverages, Greece) tuple
provider.setSort(yearLevel, OlapSort.ASC_SORT, sales, tuple);

//remove sorting for the year level
provider.setSort(yearLevel, OlapSort.NO_SORT, null, null);
```

5.4.1.3 Filtering

A filtering function is applied per hierarchy with the method:

public void setFilter(OlapFilter filter, OlapHierarchy hierarchy)

OlapFilter is an interface that has methods for defining the filtering process:

public boolean accept(OlapTableModel model, Object parent, Object child): returns true if child is accepted
public boolean hasFilterDescendants(OlapTableModel model, Object node): returns true if node has one or more children nodes filtered out.

In order to remove a filter for a hierarchy, the **setFilter** method should be called, with a null OlapFilter.

Example: Creating an OlapFilter for filtering out the “1997” year member in a time hierarchy

```
//create the filter
MyOlapFilter filter = new OlapFilter() {
  public boolean accept(OlapTableModel model, Object parent, Object child) {
    OlapMember member = model.getOlapMember(child);
    if (member.getName().equals(“1997”)) return false;
    return true;
  }
  public boolean hasFilterDescendants(OlapTableModel model, Object node) {
    OlapMember member = model.getOlapMember(node);
    if (member.getName().equals(“All Years”)) return true;
    return false;
  }
};

//set the new filter
provider.setFilter(filter, timeHier);
```

5.4.1.4 Drilling

A drill operation on a given cell can be performed with the method:

public void drill(OlapTableModel model, TreePath path, boolean up)

The drill operation replaces the visible root member of a hierarchy to the one supplied. The method arguments are:

model: the model for which drilling should occur
path: the path identifying the member in the model
up: boolean variable indicating the direction of the drill

Example: Drilling

Having **timeModel** as the DefaultOlapTableModel of a time hierarchy

```
//get the path to the "1997" member
Object rootNode = timeModel.getRoot();
Object allYearsNode = timeModel.getChild(rootNode, 0);
Object year97 = timeModel.getChild(allYearsNode, 0);
TreePath path = new TreePath(new Object[] {rootNode, allYearsNode, year97});

//drill down to 1997
provider.drill(timeModel, path, false);

//drill up to the root node
path = new TreePath(rootNode);
provider.drill(timeModel, path, true);
```

5.4.1.5 Top/Bottom

Selecting the top or bottom cells in a hierarchy, based on their natural order or on a measure, represents a common olap operation. This is achieved with the method:

public void setTopBottom(OlapLevel level, int mode, OlapMeasure measure, int count)

The information of the operation is encapsulated in an **OlapTopBottom** object. The method arguments are:

level : the level for which the top/bottom operation is applied
mode: constant in **OlapTopBottom**, specifying the top/bottom mode. One of **NO_TOP_BOTTOM**, **TOP** or **BOTTOM**.
measure: the measure upon which the top/bottom operation is based. If the measure is null, then members are selected according to their natural order in the hierarchy.
count: the maximum number of members to select, only valid if the mode is other than NO_TOP_BOTTOM.

Example: Applying top/bottom operation to a hierarchy

```
//get the year level
OlapLevel yearLevel = (OlapLevel) timeHier.getLevels().get(1);

//select the top (first) 2 rows in the year level of the time hierarchy
provider.setTopBottom(yearLevel, OlapTopBottom.TOP, null, 2);

//select the bottom 2 rows in the year level, based on the values of the sales measures
provider.setTopBottom(yearLevel, OlapTopBottom.BOTTOM, sales, 2);

//remove the top/bottom from the year level
provider.setTopBottom(yearLevel, OlapTopBottom.NO_TOP_BOTTOM, null, -1);
```

5.4.1.6 Root members

Usually, hierarchies have a single member that is defined as the root. The visibility of this root member is controlled with:

public void setShowRootMembers(boolean show)

This setting will apply globally to all hierarchies in the provider.

5.4.1.7 Subtotals

Subtotal visibility for a given level is controlled with:

public void setShowSubTotal(OlapLevel level, boolean show)

5.4.1.8 Grand totals

Row and column grand totals can be shown/hidden with appropriate methods:

public void setShowRowGrandTotal(boolean): shows/hides grand total for the row area
public void setShowColumnGrandTotal(boolean): shows/hides grand total for the column area
public void setShowGrandTotals(boolean): shows/hides grand totals for both the row and columns areas

5.4.1.9 Swapping axes

Swapping axes between the row and column area can be useful sometimes. This is achieved with:

public void swapAxes()

When calling this method, row hierarchies are pivoted to the column area and column

hierarchies to the row area.

5.4.1.10 Empty/Non-empty cells

Hiding empty cells from the pivot table can be extremely useful, especially in the case where there are a lot of them. The following methods control this behaviour:

public void setShowEmptyColumnCells(boolean): shows/hides empty column cells
public void setShowEmptyRowCells(boolean): shows/hides empty row cells
public void setShowEmptyCells(boolean): shows/hides both row and column empty cells

The default behaviour is to show all cells.

5.4.1.11 Visual Totals

With visual totals, dynamic totals are generated for a parent member, according to the visibility of its children. They are controlled with:

public void setUseVisualTotals(boolean visual)

This setting will apply globally to all the hierarchies currently pivoted. Note that if visual totals are enabled, a member's value will depend on its actual children that are shown. Values may differ, when filtering and/or top/bottom operations are present.

5.5 Olap components

The pivot functionality in our library is complemented with a few components that perform additional olap functions. In particular, drag and drop support is achieved with **HeaderDndSupport**, that allows users to move pivoted hierarchies between different regions, such as the row and column area, with the mouse. In addition, **OlapCubePanel** is able to display the dimension, hierarchies and measures of a given cube and **OlapFilterPanel** is a JPanel that allows users to install filter conditions, thus creating a **slice** of the current cube. Finally, **OlapAssistant** provides pivot tables, via a popup menu, with many common olap operations, such as sorting, filtering, drilling, top/bottom, showing of non-empty cells, axis swapping etc.

Example: Creating olap components

```
//get the sales cube
OlapCube cube = OlapUtils.findCubeByName(dataSource.getSchema(), "Sales");

//create a default olap provider
DefaultOlapProvider provider = new DefaultOlapProvider(dataSource, cube);

//create a pivot table
PivotTable pivotTable = new OlapPivotTable(provider);
```

```
//create an OlapAssistant
OlapAssistant assist = new OlapAssistant(pivotTable, provider);

//create an OlapCubePanel
OlapCubePanel cubePanel = new OlapCubePanel(provider);

//create an OlapFilterPanel
OlapFilterPanel filterPanel = new OlapFilterPanel(provider);

//add header dnd support to pivot table
HeaderDndSupport hdnd = new HeaderDndSupport(
                                pivotTable.getPivotTableHeader(), provider);
```

5.6 OlapDataModel

The default PivotDataModel for olap is OlapDataModel. Its row and column areas are constructed and maintained by an OlapProvider, which also stores and retrieves the data values. In order to create an OlapDataModel, only the OlapProvider needs to be supplied:

```
//create an olap provider from a datasource and cube
OlapProvider provider = new DefaultOlapProvider(dataSource, salesCube);

//create the OlapDataModel
OlapDataModel dataModel = new OlapDataModel();
```

5.7 Appearance

The appearance of the pivot table is easily customized. Citra Pivot includes a specialized TableCellRenderer, named **OlapCellRenderer**, which is suitable for rendering the cells in the data area of the table. OlapCellRenderer uses the value or the formatted value of an OlapCell as the table cells' text. It also delegates the rendering to the default renderer for the class of the value being returned. In addition, the style of all three areas of the pivot table, can be further improved with a DefaultOlapStyleModel, a PivotStyleModel based on OlapCells. Please see the javadoc for more information.

5.8 OlapPivotTable

OlapPivotTable is a PivotTable subclass suitable for olap data. It installs a TableCellRenderer and a PivotStyleModel for olap cells, OlapCellRenderer and DefaultOlapStyleModel respectively. An OlapPivotTable can be constructed given an OlapProvider:

```
public OlapPivotTable(OlapProvider provider)
```

Example: Creating an OlapPivotTable

Given a DataSource:

```
//get the sales cube
OlapCube cube = OlapUtils.findCubeByName(dataSource.getSchema(), "Sales");

//create olap provider
DefaultOlapProvider provider = new DefaultOlapProvider(dataSource, cube);

//create the table
OlapPivotTable table = new OlapPivotTable(provider);

//add olap assistant's functionality
OlapAssistant assist = new OlapAssistant(table, provider);

//apply dnd support to header
HeaderDndSupport hdnd = new HeaderDndSupport(table.getPivotTableHeader(), provider);

//create filter panel
JPanel filterPanel = new OlapFilterPanel(provider);

//create cube browser panel
JPanel cubePanel = new OlapCubePanel(provider);
```

6 XML/A compatibility

Citra Pivot allows connectivity to OLAP servers that support the XML/A specification. Connections are established with a 3rd party driver, **olap4j**. Olap4j has been successfully tested with Pentaho Analysis (Mondrian server), Microsoft SQL Server Analysis Services, Palo and SAP BW. For more information on olap4j, please follow the link: http://www.olap4j.org

The DataSource that connects to OLAP servers using the olap4j driver, is **Olap4jDataSource** , found in the **com.citra.pivot.olap4j** package. Developers do not have to be familiar with olap4j, minimal knowledge is only needed in order to create an olap4j connection, and pass it as argument to the sole Olap4jDataSource constructor.

6.1 Creating an olap4j connection

Before constructing an Olap4jDataSource, an olap4j connection needs to be created. The olap4j connection is represented by a **org.olap4j.OlapConnection** object. At the time of writing, there are two olap4j drivers available for connecting to OLAP servers with XML/A. The first is the generic XML/A driver, that resides in the **org.olap4j.driver.xmla** package. The second is the mondrian driver, applicable only for Mondrian servers (http://mondrian.pentaho.com). In order to create the OlapConnection, the driver needs to be loaded with a call to the **Class.forName(driverClassString)** method, with a subsequent call to **DriverManager.getConnection(connectionString)**, which will yield a **java.sql.Connection** instance. Usually, the OLAP server provider will supply the correct connection string to use. Next, the connection needs to be unwrapped and the OlapConnection instance is retrieved. Please find some connection examples for common OLAP servers below.

Example 1: Connection to Mondrian server using generic olap4j driver

```
//load the driver
Class.forName("org.olap4j.driver.xmla.XmlaOlap4jDriver");

//create sql connection
java.sql.Connection connection = DriverManager.getConnection(
                "jdbc:xmla:Server=http://localhost:8080/mondrian/xmla;
Catalog=FoodMart;");

//unwrap connection
OlapWrapper wrapper = (OlapWrapper) connection;
OlapConnection olapConnection = (OlapConnection) wrapper.unwrap(OlapConnection.class);
```

Example 2: Connection to Mondrian server using mondrian driver

```
//load the driver
Class.forName("mondrian.olap4j.MondrianOlap4jDriver");
```

```
//create sql connection
java.sql.Connection connection = DriverManager.getConnection(
                        "jdbc:mondrian:Jdbc=jdbc:odbc:MondrianFoodMart;
                        Catalog=c:\\mondrian\\FoodMart.xml;");

//unwrap connection
OlapWrapper wrapper = (OlapWrapper) connection;
OlapConnection olapConnection = (OlapConnection) wrapper.unwrap(OlapConnection.class);
```

Example 3: Connection to Microsoft SQL Server Analysis Services using generic olap4j driver

```
//load the driver
Class.forName("org.olap4j.driver.xmla.XmlaOlap4jDriver");

//create sql connection
OlapConnection connection = (OlapConnection) DriverManager.getConnection(
                            "jdbc:xmla:Server=http://localhost/olap/msmdpump.dll;");

//unwrap connection
OlapWrapper wrapper = (OlapWrapper) connection;
OlapConnection olapConnection = (OlapConnection) wrapper.unwrap(OlapConnection.class);
```

Notes for the examples above:

-In example 1, the j2ee server (e.g. tomcat) that hosts Mondrian, listens at port 8080.
-In example 2, an ODBC data source named MondrianFoodMart needs to created beforehand. Please see Mondrian's documentation for more information.
-In example 3, the SQL Server listens at port 80.

6.2 Library dependencies

While the generic olap4j driver is included in the Citra Pivot distribution, the mondrian driver is not. The mondrian driver should be downloaded separately from Pentaho, if developers wish to use this instead. In order to use the generic olap4j driver, the following files, residing in the /lib/ folder, need to be added to the classpath:

-olap4j-1.0.1.500.jar: olap4j's core API and driver specification
-olap4j-xmla-1.0.1.500.jar: olap4j's driver implementation for XML/A data sources
-javacup.jar: java parser required by olap4j driver
-xercesImpl.jar: xml parser required by olap4j driver

Note that depending on the mondrian driver version used, additional files will need to be added to the classpath. Please refer to the Mondrian server documentation for more information.

6.3 Olap4jDataSource

After the OlapConnection has been created, an Olap4jDataSource can be constructed with the connection as argument. Namely:

Olap4jDataSource dataSource = new Olap4jDataSource(olapConnection);

The datasource can then be used for querying and pivoting. The original OlapConnection can also be retrieved with the method:

public OlapConnection getOlap4jConnection()

In addition to the queries performed with the Citra Pivot API, the olap4j API can also be used to make queries in the mdx format, via the OlapConnection. For example:

```
OlapStatement statement = olapConnection.createStatement();

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

Please refer to the olap4j documentation for more information on mdx queries using the olap4j API.

7 TableDataSource

TableDataSource is a in-memory DataSource implementation that creates an olap cube from a two-dimensional data model. The input data model is a **javax.swing.table.TableModel**, with which Java Swing developers are already familiar. The TableModel may be created from a variety of sources, such as from an SQL database or a file (csv, xml etc).

In a TableDatasource, the entire schema, including cubes, hierarchies, levels and measures need to be created programmatically. The framework contains objects such as **TableSchema**, **TableCube**, **TableDimension**, **TableHierarchy**, **TableLevel**, **TableMeasure**, **TableMember**, each referring to the corresponding olap metadata. The schema is created by calling appropriate methods and constructors of these objects.

Once a schema has been defined, the datasource can be compiled, so that it becomes available for querying, as well as for pivoting on a PivotTable. All data resides in memory, however compiled datasources may be serialized to an ObjectOutputStream, and saved to a file, so that they can be loaded at a later time. TableDataSource also supports incremental data loading, which can be very useful for large data quantities.

The structure of an olap hierarchy is created with the use of a **TableContext**, which also defines the captions and descriptions of its members for a given Locale.

The OlapMeasure representation in the TableDataSource framework is a **TableMeasure**. Measures are distinguished in two types, **StandardTableMeasure** and **DerivedTableMeasure**. The first does not require another measure for its definition and evaluation, while the second does. Measures are evaluated with a **TableAggregator**, **StandardMeasureAggregators** are used for StandardTableMeasures, while **DerivedMeasureAggregators** for DerivedTableMeasures.

TableQuery and **TableTuple** are used to make direct queries to the datasource, and are also employed when evaluating measures by TableAggregators.

Finally, the measure values retrieved from the datasource are formatted with the use of a **TableFormatter**.

Developers wishing to use TableDataSource should first familiarize themselves with the notions of TableContext and TableAggregator that will be described next.

7.1 TableContext

A TableContext is specified when creating a TableLevel. It is responsible for creating the level members, and therefore the whole hierarchy structure.

Given that the input model is a two dimensional table, TableContext will extract member values from each row of the table. This is achieved with the **extractMemberValue(List row)**

method. The member values that were extracted are then compared to each other in the **compareMemberValues(Object memberValue1, Object memberValue2)** method. This method returns an integer as is the case with java.util.Comparator, that determines the comparison result and also the relative location of the members to each other. The member name, description and caption are also retrieved with the **getName(Object memberValue, List row)**, **getDescription(Object memberValue, List row, Locale locale)** and **getCaption(Object memberValue, List row, Locale locale)** methods respectively.

There exist several TableContexts, that reside in the **com.citra.pivot.table.context** package. Usually, a **SingleColumnContext** will be used for most purposes. This context extracts member values, as well as names, captions and descriptions from a single column in the data model. For creating an 'all' member in a hierarchy, a **SingleGroupContext** can be used. Other contexts in the package deal with dates, and they are:

YearContext: for years
QuarterContext: for quarters
MonthContext: for months
WeekContext: for weeks
DayContext: for days

Please refer to the javadoc documentation for more information on these objects.

7.2 TableAggregator

TableAggregators are used for aggregating data and are an essential part of every measure. There are two types of aggregators, StandardMeasureAggregator for standard measures and DerivedMeasureAggregator for derived ones. All three are interfaces, concrete implementations of which are found in the **com.citra.pivot.table.aggregator** package. This package also contains a number of commonly used functions, such as sum, max, min, average etc, encapsulated in the Functions class. Developers may create and use their own TableAggregators or subclass existing ones.

7.2.1 StandardMeasureAggregator

Aggregation for a standard measure is a two-step process. First, we need to know the members (usually the children members) that the current aggregated member depends on. This is achieved with the **getAggregatedMembers(TableMember currentMember, TableConnection connection, TableMeasure measure)** method. Next, the aggregated value is calculated with the **getAggregateValue(List queries, TableTuple query, TableConnection connection, TableMeasure measure)** method. The first list argument (queries) passed to that method, contains the aggregated values of its dependent members.

For example, in a time hierarchy with a year and a quarter level, the aggregated members of the "1997" year member are its quarter members, i.e. “Q1-1997”, “Q2-1997”, “Q3-1997” and “Q4-1997”, which are returned from the **getAggregatedMembers** method. When summing,

the **getAggregateValue** method sums over all quarter member values passed as argument to it.

The com.citra.pivot.table.aggregator package contains most commonly used aggregators. For example:

SumAggregator: used for summation
AvgAggregator: used for averaging
MaxAggregator: produces aggregates that are the maximum value
MinAggregator: produces aggregates that are the minimum value
ScaledSumAggregator: sums over values with a weight to each value
FirstValueAggregator: selects the first non-value

Please refer to the Citra Pivot javadoc for a complete list.

7.2.2 DerivedMeasureAggregator

DerivedMeasureAggregators are used for calculating aggregated values for a derived measure. The only method defined is **aggregate(TableQuery query, TableConnection connection, TableMeasure measure, TableMeasure inputMeasure)**, which depends on a single query, instead of a list, like is the case in a StandardMeasureAggregator. Typically, a DerivedMeasureAggregator produces aggregate values based on a standard measure, however it can also be based on a derived one.

At the time of writing of this manual, there are three DerivedMeasureAggregators in the com.citra.pivot.table.aggregator package. These are:

RankAggregator: a ranking for a given measure in the context of a dimension
IndexAggregator: the ratio of the value defined by a list of members to that of another measure
MovingTotalAggregator: a moving total aggregation of a given measure in the context of a dimension

7.3 Defining the schema

A schema can be easily created using simple public API methods. Usually, an olap schema is composed of a number of cubes. An olap cube consists of a number of dimensions and measures. An olap dimension itself consists of a number of hierarchies. However, because of the fact that TableDataSource keeps all data in memory, and because of memory limitations, the number of cubes and hierarchies per dimension are currently restricted to one.

In order to create a schema, we need to define the dimensions, hierarchies, levels and measures. This process will be described in the following pages.

7.3.1 TableSchema

TableSchema has methods for adding and removing cubes and dimensions to itself. In order to create a schema, we first need to give it a name. Cubes and dimensions that are created from a schema, are automatically added to it.

Example: Creating a TableSchema, adding/removing dimensions

```
//create a schema named e-shop
TableSchema schema = new TableSchema("e-shop");

//create a dimension named products
TableDimension productsDim = schema.createDimension(“Products”);

//create a cube named sales
TableCube salesCube = schema.createCube(“Sales”);

//remove the products dimension
schema.removeDimension(productsDim);

//remove the sales cube
schema.removeCube(salesCube);
```

7.3.2 TableCube

TableCube is a dynamically created OlapCube. It has methods for creating measures, both standard and derived. Dimensions and supported locales may also be added to the cube.

Standard measures are calculated from values that originate in the input TableModel, typically from a single column. These are called leaf values, and are the starting point for the measure aggregation. TableDataSource needs to know how to retrieve and calculate these leaf values. For this purpose, a TableLeafAggregator is used. The default leaf aggregator is DefaultTableLeafAggregator, which takes data from a single column, summing over the found values. After the leaf values are calculated, the remaining aggregations are performed with StandardMeasureAggregators. For a more detailed description of standard measures, see the chapter on TableMeasure. There are several ways to create a standard measure, the simpler of which is to supply the measure's name, java class type and the values column in the TableModel. For a complete measure definition, the name, java class type, olap type, TableLeafAggregator and StandardMeasureAggregator should be given.

Derived measures depend on another measure, typically a standard one, however a derived measure can also be used. Instead of supplying a StandardMeasureAggregator, derived measures use a DerivedMeasureAggregator.

Example: Creating measures (standard and derived)

Having **cube** as a TableCube previously created from a TableSchema:

```
//create a sales measure using sum aggregator
TableMeasure sales = cube.createMeasure("Net Sales", Double.class, salesColumn),
                                                    Aggregators.getSum());

//create a sales measure using max aggregator
TableMeasure maxSales = cube.createMeasure("Max Sales", Double.class, salesColumn),
                                                    Aggregators.getMax());

//create a derived measure for a moving total of the sales measure over the time dimension
DerivedTableMeasure moving = cube.createDerivedMeasure("Moving", Double.class,
                                                    new MovingTotalAggregator(timeDim));
```

7.3.3 TableDimension

TableDimension is an OlapDimension that is part of a TableSchema and of a TableCube. TableDimension has methods for creating and removing hierarchies.

Example: Creating/removing hierarchies

Having **productsDim** as a TableDimension previously created from a TableSchema:

```
//create a hierarchy named products
TableHierarchy productsHier = productsDim.createHierarchy("Products");

//remove the products hierarchy
productsDim.removeHierarchy(productsHier);
```

7.3.4 TableHierarchy

TableHierarchy is an OlapHierarchy that defines methods for adding and removing TableLevels. The default member can also be assigned. A TableLevel is added by supplying its name and TableContext. A column index in the TableModel may also be given instead of the TableContext, in which case, a SingleColumnContext is created having the column index as argument to its constructor.

Example: Creating/removing levels

Having **productsHier** and **timeHier** as hierarchies previously created from a TableDimension:

```
//create a level named categories
TableLevel categoryLevel = productsHier.createLevel("Categories", categoryColumn);

//remove the category level
productsHier.removeLevel(categoryLevel);

//add All Years level
timeHier.createLevel("All Years", new SingleGroupContext("All Years"));
```

```
//add Years level
TableLevel yearLevel = timeHier.createLevel(“Year”, new YearContext(dateColumn));

//add Quarter level
timeHier.createLevel ("Quarter", new QuarterContext(dateColumn));

//add Month level
timeHier.createLevel ("Month", new MonthContext(dateColumn));
```

7.3.5 TableLevel

TableLevel represents an OlapLevel that is created from a TableHierarchy. Its context may be assigned and retrieved with suitable methods. More specifically:

public TableContext getContext(): retrieve the current context
public void setContext(TableContext context): assign the context

7.3.6 TableMeasure

A TableMeasure is distinguished in two types, StandardTableMeasure and DerivedTableMeasure. The first is calculated from values that directly originate in the input TableModel, while the second is "derived" from the values of another measure. Additionally, data in standard measures are aggregated with a StandardMeasureAggregator, while derived ones with a DerivedMeasureAggregator. However more importantly, there is another critical distinction between a standard and a derived measure. A standard measure is aggregated over each dimension in a cube, one dimension at a time, with an aggregator that may be different for each dimension. The order with which dimensions will be aggregated is important, and it can produce different results. For example, in a cube with two dimensions, both of which use an average aggregator, the order matters. Derived measures, on the other hand, have only one aggregator and, therefore, there is no notion of order.

In the TableCube description above, it was shown how to create a standard measure. TableCube's **createMeasure** method, accepts only one StandardMeasureAggregator (when there is no argument for an aggregator, a SumAggregator is used). The aggregator specified in the create method, will be used for all the cube's dimensions in an arbitrary order. However, StandardTableMeasure includes methods for assigning different aggregators to dimensions and also for changing the order in which the aggregators will be applied. In particular, this is accomplished with the following StandardTableMeasure methods:

```
public void setAggregator(StandardMeasureAggregator aggregator,
                                                TableDimension dimension)

public void setAggregator(StandardMeasureAggregator aggregator,
                                                TableDimension dimension, int priority)

public void setPriority(TableDimension dimension, int priority)
```

The priority argument specifies the aggregation order. Dimensions with higher priority will be processed before others, while those with the same priority, will be processed in a random order.

Example: Creating a standard "inventory" measure

```
//create a standard measure with a SumAggregator (no-aggregator argument)
//remainingProductsColumn contains the number of items remaining in stock after each order

StandardTableMeasure inventory = cube.createMeasure("Inventory", Integer.class,
                                                    remainingProductsColumn);

//set a last value aggregator over the time dimension
inventory.setAggregator(Aggregators.getLast(), timeDim);
```

Derived measures are aggregated with a DerivedMeasureAggregator, which usually contains parameters that dictate the aggregation process. For example, in MovingTotalAggregator, the dimension over which the moving total will be evaluated is defined, as well as the offsets to lead or the lag the current member under evaluation.

Example: Creating a derived measure

```
//create a derived measure for a moving total of the sales measure over the time dimension
DerivedTableMeasure moving = cube.createDerivedMeasure("Moving", Double.class,
                                                    new MovingTotalAggregator(timeDim));
```

7.3.7 Example: Creating a schema

Having **model** as a DefaultTableModel:

```
//start with creating a schema called sales
TableSchema schema = new TableSchema("sales");

//create products dimension
TableDimension productsDim = schema.createDimension("Products");

//create the default hierarchy for the products dimension
TableHierarchy productsHier = productsDim.createHierarchy("Products");

//add levels to the products hierarchy
productsHier.createLevel("All Products", new SingleGroupContext("All Products"));
productsHier.createLevel("Categories", model.findColumn("Category"));
productsHier.createLevel("Products", model.findColumn("Product"));

//create time dimension
TableDimension timeDim = schema.createDimension("Time");

//create the default hierarchy for the time dimension
TableHierarchy timeHier = timeDim.createHierarchy("Time");
```

```
//add levels to the time hierarchy
timeHier.createLevel("All Years", new SingleGroupContext("All Years"));
timeHier.createLevel("Year", new YearContext(model.findColumn("Order Date")));
timeHier.createLevel("Quarter", new QuarterContext(model.findColumn("Order Date")));
timeHier.createLevel("Month", new MonthContext(model.findColumn("Order Date")));

//create customer dimension
TableDimension customerDim = schema.createDimension("Customer");

//create the default hierarchy for the customer dimension
TableHierarchy customerHier = customerDim.createHierarchy("Customer");

//add levels to the customer hierarchy
customerHier.createLevel("All Customers", new SingleGroupContext("All Customers"));
customerHier.createLevel("Customer Country", model.findColumn("Customer Country"));
customerHier.createLevel("Customer City", model.findColumn("Customer City"));
customerHier.createLevel("Customer", model.findColumn("Customer"));

//create a cube called sales
TableCube cube = schema.createCube("sales");

//add dimensions to the cube
cube.addDimension(productsDim);
cube.addDimension(timeDim);
cube.addDimension(customerDim);

//find the column in the model representing the "Net Sales" values
int sales = model.findColumn("Net Sales");

//create some measures
TableMeasure sales = cube.createMeasure("Net Sales", Double.class, sales,
                                        Aggregators.getSum());

TableMeasure maxSales = cube.createMeasure("Max Sales", Double.class, sales,
                                        Aggregators.getMax());

NormalTableMeasure orders = cube.createMeasure("# of Orders", Integer.class, sales,
                                        Aggregators.getSum());

orders.setLeafAggregator(new DefaultTableLeafAggregator(sales,
                                        Functions.getCountFunction()));
```

7.4 Creating a TableDataSource

In order to create a TableDataSource, a schema and a TableModel is needed. We already saw how to create a schema in the previous pages. As for the TableModel, this may originate from a variety of sources, such as from an SQL database, a file (csv, xml) etc. While any TableModel would do, our product, Citra Table, that deals exclusively with tables in Java, contains a number of TableModel subclasses, such as ObjectTableModel and

DefaultDatabaseTableModel, that facilitate the model creation.

Once the schema and TableModel have been constructed, we can proceed with creating the TableDataSource:

```
//create the TableDataSource
TableDataSource dataSource = new TableDataSource(schema, dataModel);
```

Next, the datasource needs to be compiled in order to become available for querying and pivoting:

```
//compile the data source
try {
        dataSource.compile();
} catch (DataSourceException e) {
        e.printStackTrace();
}
```

Note that a TableDataSource may also be created using the no-argument constructor, and the schema and TableModel assigned to it at a later time. For example:

```
//create the TableDataSource
TableDataSource dataSource = new TableDataSource();

//assign the schema
dataSource.setSchema(schema);

//assign the model
dataSource.setModel(dataModel);
```

The datasource still needs to be compiled after the schema and model have been assigned.

During compilation, TableDataSource will discover and create the hierarchies and its members. The data aggregation that will take place depends on the so-called "pre-compute mode" that has been assigned. Valid values for a pre-compute mode are:

PRECOMPUTE_NOTHING: no aggregations should be performed
PRECOMPUTE_ALL: all data should be aggregated
PRECOMPUTE_LEAVES: only the leaf members should be aggregated
PRECOMPUTE_TOP: only the first level member values should be aggregated

By default, TableDataSource uses a mode of **PRECOMPUTE_TOP**. When the pre-compute mode is other than **PRECOMPUTE_ALL**, data will be aggregated on demand, according to the queries being made. Also, after the datasource has been compiled, changing the pre-compute mode, has no effect what-so-ever: it only applies BEFORE the compilation.

NOTE: Since all data in a TableDataSource resides in memory, memory issues may be a problem. You might want to increase the maximum memory used by the java application

using the -Xmx switch. For example:

java -Xmx600m MyApplication

will increase the maximum memory used to 600 megabytes.

7.5 Saving/loading

Discovering a cube's hierarchies as well as data aggregation may be a time-consuming process (depending on the schema and data), which also takes up a lot of memory resources. For this reason, TableDataSource supports saving and loading of the schema and associated aggregates. After compilation, the TableDataSource may be serialized to an ObjectOutputStream, and piped to a file, so that it can be loaded at a later time. The datasource will serialize the whole schema, including its cubes, hierarchies and measures, and whatever data had been aggregated so far. In order to save more data than those that have already been aggregated, a pre-compute mode may also be specified, such as PRECOMPUTE_ALL, which will effectively include everything.

Example: Saving to a file the schema and whatever aggregates are evaluated so-far

```
//create the output stream
ObjectOutputStream out = new ObjectOutputStream(
                                new FileOutputStream(new File("c:\\mydata.bin")));

//save file to the stream by pre-computing all aggregated values first
dataSource.save(out);

//the above call is the same as: dataSource.save(out, PRECOMPUTE_NOTHING);

//close the stream
out.close();
```

Example: Saving to a file the schema and all aggregated data

```
//create the output stream
ObjectOutputStream out = new ObjectOutputStream(
                                new FileOutputStream(new File("c:\\mydata.bin")));

//save file to the stream by pre-computing all aggregated values first
dataSource.save(out, TableDataSource.PRECOMPUTE_ALL);

//close the stream
out.close();
```

The output that was serialized using the **save** method above, may be loaded into the TableDataSource.

Example: Loading a serialized file to the datasource

```
//create the datasource
TableDataSource dataSource = new TableDataSource();
//create the input stream

ObjectInputStream in = new ObjectInputStream(
                                    new FileInputStream( new File("c:\\mydata.bin")));

//load the datasource
dataSource.load(in);

//close the stream
in.close();
```

7.6 Incremental aggregation

TableDataSource also supports incremental loading of data. This may prove extremely useful for large data quantities, when memory limitations are present. For example, a TableModel consisting of 2 billion rows may be impossible to exist in memory, let alone loaded into the datasource. We could however chunk the model into smaller pieces, and perform incremental data loading and aggregation. The datasource could then be saved to an external file, so that the whole process need not be repeated. See below for an example.

Example: Incremental data loading

```
//create a data source
TableDataSource source = new TableDataSource(schema);

//compile the data source
source.compile();

//get first chunk of data
TableModel data1 = getModel(0, 10000); //user-defined method for retrieving first 10,000 rows

//add the model to the datasource
source.setModel(data1, false); //a value of 'false' can be omitted here since it is the first batch

//get second chunk of data
TableModel data2 = getModel(10000, 20000); //user-defined method for retrieving next 10,000
rows

//add the model to the datasource
source.setModel(data2, false); //a value of 'false' is needed so that data can be added instead of
replaced

//get second chunk of data
TableModel data3 = getModel(20000, 30000); //user-defined method for retrieving next 10,000
rows
```

```
//add the model to the datasource
source.setModel(data3, false); //a value of 'false' is needed so that data can be added instead of replaced

//...and so on
```

7.7 Making queries

Although data are retrieved from a datasource's connection with **execute(OlapSelection selection)**, values may also be acquired directly from the TableDataSource, by calling the **getValue(TableMeasure measure, TableQuery query)** method. The input to the **getValue** method should completely identify an OlapCell. TableQuery is a placeholder for a TableTuple , which, in turn, contains the members that define the olap cell.

Example: Making a query

Having **source** as our TableDataSource:

```
//create a new connection - the connection will be used to discover the members below
Connection connection = source.createConnection();

//retrieve members by specifying their full path
OlapMember m1997 = OlapUtils.retrieveMember(
                        new String[] {"Time", "Time", "1997"}, connection);

OlapMember mBev = OlapUtils.retrieveMember(
                        new String[] {"Products", "Products", "Beverages"}, connection);

//close the connection
connection.close();

//create a list to hold the query members
List members = new ArrayList();

//add the members to the list
members.add(m1997);
members.add(mBev);

//construct the TableTuple
TableTuple tuple = new TableTuple(members);

//construct the query
TableQuery query = new TableQuery(tuple);

//fetch the sales measure
OlapCube salesCube = OlapUtils.findCubeByName(source.getSchema(), "Sales");
TableMeasure sales = OlapUtils.findMeasureByName(salesCube, "Net Sales");

//retrieve the value
```

Object value = source.getValue(sales, query);

7.8 Formatting values

Values that are returned from a query need to be formatted to a string, so that they can be encapsulated in an OlapCell. In TableDataSource, this is achieved with a TableFormatter. TableFormatter has only one method that defines the way a given value is transformed into a String:

public String format(Object value, TableMeasure measure, Locale locale)

By default, TableDataSource uses a DefaultTableFormatter, that returns string representations of number values using various number formats. A new formatter may be assigned with the **setTableFormatter(TableFormatter tableFormatter)** method.

8 Remote Operations

Olap operations are sometimes time consuming. As a result, the UI freezes until these operations are completed. In order to get around this problem, a **RemoteOlapModel** can be used. RemoteOlapModel will asynchronously retrieve the values from an OlapDataModel. These values are retrieved from a separate thread outside the Event Dispatch Thread (EDT), thus ensuring that the pivot table does not freeze while data operations take place.

While RemoteOlapModel may work on its own, there are also a few components that show the underlying data retrieval activity. These are **RemoteOlapPanel** and **RemoteOlapStyle**. A value that has not been fetched yet is translated into a RemotePendingValue. Finally, RemoteOlapModel uses a **RemoteOlapListener** to indicate when it will start or stop retrieving the underlying data, through a **RemoteOlapEvent**.

8.1 RemoteOlapPanel

RemoteOlapPanel is a JPanel that shows the current status of a RemoteOlapModel. It contains a label and an indicator that update themselves according to the data retrieval activity. The remote panel will receive an event each time an olap operation is started or stopped and its text and indicator will be updated accordingly.

The indicator is a RemotePendingIndicator, a component that sweeps a circular path, while data is being retrieved.

8.2 RemoteOlapStyle

RemoteOlapStyle paints pending cells with a background color. It will be applied only if the value of a cell is an instance of RemotePendingValue. In order to take effect, the style should be added to pivot table's style model. By default, the style uses a yellow background color. The background color can be assigned with the method:

public void setPendingBackgroundColor(Color pendingBackgroundColor)

The background color is also determined with the method:

public Color getPendingBackgroundColor()

8.3 RemotePendingValue

RemotePendingValue is an interface, extending com.citra.pivot.olap.data.OlapCell, that represents values that have not yet been retrieved by RemoteOlapModel. Its default implementation in this package is **DefaultRemotePendingValue**, with a text value equal to an empty string.

8.4 RemoteOlapListener

RemoteOlapListener is a listener that is notified each time RemoteOlapModel starts or stops querying the underlying olap data model. RemoteOlapModel will fire a suitable RemoteOlapEvent. RemoteOlapListener has one notifying method:

public void remoteActionPerformed(RemoteOlapEvent e)

Also, RemoteOlapEvent identifies the affected cell and type of remote operation which will be propagated to the listeners:

public TreePath[] getRowPaths() : the row path part to retrieve or that was retrieved
public TreePath[] getColumnPaths() : the column path part to retrieve or that was retrieved
public int getType() : either RemoteOlapEvent.STARTED or RemoteOlapEvent.STOPPED, indicating a start or stop activity respectively

8.5 Usage

Example: How to use RemoteOlapModel

Having olapDataModel as the underlying model:

```
//create a RemoteOlapModel
RemoteOlapModel remoteModel = new RemoteOlapModel(olapDataModel);

//create pivot model
PivotTableModel pivotModel = new DefaultPivotTableModel(remoteModel);

//create pivot table
PivotTable pivotTable = new OlapPivotTable(pivotModel);

//add remote olap style to pivot table
RemoteOlapStyle style = new RemoteOlapStyle();
pivotTable.getStyleModel().addStyle(style);

//create remote olap panel – this panel can be added anywhere in the application
RemoteOlapPanel remotePanel = new RemoteOlapPanel();

//add the panel as listener to the remote olap model
remoteModel.addRemoteListener(remotePanel);
```