

Mondrian 2.2.2 Technical Guide

Developing OLAP solutions with Mondrian

March 2007

Table of Contents

License and Copyright	4
Introduction.....	9
Mondrian and OLAP	10
Online Analytical Processing	10
Mondrian Architecture	12
Layers of a Mondrian system	12
API	15
How to Design a Mondrian Schema	17
What is a schema?.....	17
Schema files.....	17
Logical model	17
Cube.....	18
Measures	19
Dimensions, Hierarchies, Levels.....	20
An example	20
Mapping dimensions and hierarchies onto tables	21
The 'all' member.....	21
Time dimensions	21
Order and display of levels	22
Multiple hierarchies.....	23
Degenerate dimensions.....	23
Inline tables	24
Member properties and formatters.....	25
Approximate level cardinality	25
Star and snowflake schemas	25
Shared dimensions	26
Join optimization	27
Advanced logical constructs.....	27
Member properties	30
Calculated members	31
Named sets.....	32
Plug-ins.....	34
Member reader.....	37
Internationalization.....	39
Aggregate tables	41
Access-control	42
XML elements.....	44
MDX.....	47
What is the syntax of MDX?.....	47
Mondrian-specific MDX.....	47
Configuration Guide	51
Properties that control execution	51
Property list.....	51
Connect strings	58
Optimizing Mondrian Performance.....	60
Introduction	60
A generalized tuning process for Mondrian	60
Recommendations for database tuning	61
Aggregate Tables, Materialized Views and Mondrian	61
AggGen.....	62
Aggregate Tables.....	63
Introduction	63

What are aggregate tables?	64
A simple aggregate table.....	65
Another aggregate table	66
Defining aggregate tables	67
Building aggregate tables.....	68
How Mondrian recognizes Aggregate Tables.....	74
Aggregate tables and parent-child hierarchies	79
How Mondrian uses aggregate tables.....	82
Tools for designing and maintaining aggregate tables.....	85
Properties which affect aggregates	86
Aggregate Table References.....	88
Mondrian CmdRunner.....	89
What is CmdRunner?	89
Building.....	89
Usage	89
Properties File	90
Command line arguments	91
CmdRunner Commands.....	91
AggGen: Aggregate SQL Generator.....	95
Mondrian FAQs	99
Why doesn't Mondrian use a standard API?.....	99
How does Mondrian's dialect of MDX differ from Microsoft Analysis Services?	99
How can Mondrian be extended?.....	99
Can Mondrian handle large datasets?.....	99
How do I enable tracing?	99
How do I enable logging?.....	100
Where can I find out more?.....	100
OLAP Modeling	101
Performance.....	102
Results Caching – The key to performance	104
Segment	105
Member set.....	105
Schema.....	105
Star schemas.....	105
Learning more about Mondrian	106
How Mondrian generates SQL.....	106
Logging Levels and Information.....	107
Default aggregate table recognition rules	108
Snowflakes and the DimensionUsage level attribute.....	113
Appendix A – MDX Function List.....	117

License and Copyright

This manual is derived from content published as part of the Mondrian open source project at <http://mondrian.pentaho.org>, <https://sourceforge.net/projects/mondrian> and https://sourceforge.net/project/showfiles.php?group_id=35302.

This content is published under the Common Public License Agreement version 1.0 (the "CPL", available at the following URL: <http://www.opensource.org/licenses/cpl.html>) - the same license as the the original content.

Copyright is retained by the individual contributors note on the various sections of this document.

Common Public License - v 1.0

THE ACCOMPANYING PROGRAM IS PROVIDED UNDER THE TERMS OF THIS COMMON PUBLIC LICENSE ("AGREEMENT"). ANY USE, REPRODUCTION OR DISTRIBUTION OF THE PROGRAM CONSTITUTES RECIPIENT'S ACCEPTANCE OF THIS AGREEMENT.

1. DEFINITIONS

"Contribution" means:

- a) in the case of the initial Contributor, the initial code and documentation distributed under this Agreement, and
- b) in the case of each subsequent Contributor:
 - i) changes to the Program, and
 - ii) additions to the Program;where such changes and/or additions to the Program originate from and are distributed by that particular Contributor. A Contribution 'originates' from a Contributor if it was added to the Program by such Contributor itself or anyone acting on such Contributor's behalf. Contributions do not include additions to the Program which: (i) are separate modules of software distributed in conjunction with the Program under their own license agreement, and (ii) are not derivative works of the Program.

"Contributor" means any person or entity that distributes the Program.

"Licensed Patents " mean patent claims licensable by a Contributor which are necessarily infringed by the use or sale of its Contribution alone or when combined with the Program.

"Program" means the Contributions distributed in accordance with this Agreement.

"Recipient" means anyone who receives the Program under this Agreement, including all Contributors.

2. GRANT OF RIGHTS

- a) Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free copyright license to reproduce, prepare derivative works of, publicly display, publicly perform, distribute and sublicense the Contribution of such Contributor, if any, and such derivative works, in source code and object code form.
- b) Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free patent license under Licensed Patents to make, use, sell, offer to sell, import and otherwise transfer the Contribution of such Contributor, if any, in source code and object code form. This patent license shall apply to the combination of the Contribution and the Program if, at the time the Contribution is added by the Contributor, such addition of the Contribution causes such combination to be covered by the Licensed Patents. The patent license shall not apply to any other combinations which include the Contribution. No hardware per se is licensed hereunder.
- c) Recipient understands that although each Contributor grants the licenses to its Contributions set forth herein, no assurances are provided by any Contributor that the Program does not infringe the patent or other intellectual property rights of any other entity. Each Contributor disclaims any liability to Recipient for claims brought by any other entity based on infringement of intellectual property rights or otherwise. As a condition to exercising the rights and licenses granted hereunder, each Recipient hereby assumes sole responsibility to secure any other intellectual property rights needed, if any. For example, if a third party patent license is required to allow

Recipient to distribute the Program, it is Recipient's responsibility to acquire that license before distributing the Program.

d) Each Contributor represents that to its knowledge it has sufficient copyright rights in its Contribution, if any, to grant the copyright license set forth in this Agreement.

3. REQUIREMENTS

A Contributor may choose to distribute the Program in object code form under its own license agreement, provided that:

- a) it complies with the terms and conditions of this Agreement; and
- b) its license agreement:
 - i) effectively disclaims on behalf of all Contributors all warranties and conditions, express and implied, including warranties or conditions of title and non-infringement, and implied warranties or conditions of merchantability and fitness for a particular purpose;
 - ii) effectively excludes on behalf of all Contributors all liability for damages, including direct, indirect, special, incidental and consequential damages, such as lost profits;
 - iii) states that any provisions which differ from this Agreement are offered by that Contributor alone and not by any other party; and
 - iv) states that source code for the Program is available from such Contributor, and informs licensees how to obtain it in a reasonable manner on or through a medium customarily used for software exchange.

When the Program is made available in source code form:

- a) it must be made available under this Agreement; and
- b) a copy of this Agreement must be included with each copy of the Program.

Contributors may not remove or alter any copyright notices contained within the Program.

Each Contributor must identify itself as the originator of its Contribution, if any, in a manner that reasonably allows subsequent Recipients to identify the originator of the Contribution.

4. COMMERCIAL DISTRIBUTION

Commercial distributors of software may accept certain responsibilities with respect to end users, business partners and the like. While this license is intended to facilitate the commercial use of the Program, the Contributor who includes the Program in a commercial product offering should do so in a manner which does not create potential liability for other Contributors. Therefore, if a Contributor includes the Program in a commercial product offering, such Contributor ("Commercial Contributor") hereby agrees to defend and indemnify every other Contributor ("Indemnified Contributor") against any losses, damages and costs (collectively "Losses") arising from claims, lawsuits and other legal actions brought by a third party against the Indemnified Contributor to the extent caused by the acts or omissions of such Commercial Contributor in connection with its distribution of the Program in a commercial product offering. The obligations in this section do not apply to any claims or Losses relating to any actual or alleged intellectual property infringement. In order to qualify, an Indemnified Contributor must: a) promptly notify the Commercial Contributor in writing of such claim, and b) allow the Commercial Contributor to control, and cooperate with the Commercial Contributor in, the defense and any related settlement negotiations. The Indemnified Contributor may participate in any such claim at its own expense.

For example, a Contributor might include the Program in a commercial product offering, Product X. That Contributor is then a Commercial Contributor. If that Commercial Contributor then makes performance claims, or offers warranties related to Product X, those performance claims and warranties are such Commercial Contributor's responsibility alone. Under this section, the Commercial Contributor would have

to defend claims against the other Contributors related to those performance claims and warranties, and if a court requires any other Contributor to pay any damages as a result, the Commercial Contributor must pay those damages.

5. NO WARRANTY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, THE PROGRAM IS PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OR CONDITIONS OF TITLE, NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Each Recipient is solely responsible for determining the appropriateness of using and distributing the Program and assumes all risks associated with its exercise of rights under this Agreement, including but not limited to the risks and costs of program errors, compliance with applicable laws, damage to or loss of data, programs or equipment, and unavailability or interruption of operations.

6. DISCLAIMER OF LIABILITY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, NEITHER RECIPIENT NOR ANY CONTRIBUTORS SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE PROGRAM OR THE EXERCISE OF ANY RIGHTS GRANTED HEREUNDER, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. GENERAL

If any provision of this Agreement is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this Agreement, and without further action by the parties hereto, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

If Recipient institutes patent litigation against a Contributor with respect to a patent applicable to software (including a cross-claim or counterclaim in a lawsuit), then any patent licenses granted by that Contributor to such Recipient under this Agreement shall terminate as of the date such litigation is filed. In addition, if Recipient institutes patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Program itself (excluding combinations of the Program with other software or hardware) infringes such Recipient's patent(s), then such Recipient's rights granted under Section 2(b) shall terminate as of the date such litigation is filed.

All Recipient's rights under this Agreement shall terminate if it fails to comply with any of the material terms or conditions of this Agreement and does not cure such failure in a reasonable period of time after becoming aware of such noncompliance. If all Recipient's rights under this Agreement terminate, Recipient agrees to cease use and distribution of the Program as soon as reasonably practicable. However, Recipient's obligations under this Agreement and any licenses granted by Recipient relating to the Program shall continue and survive.

Everyone is permitted to copy and distribute copies of this Agreement, but in order to avoid inconsistency the Agreement is copyrighted and may only be modified in the following manner. The Agreement Steward reserves the right to publish new versions (including revisions) of this Agreement from time to time. No one other than the Agreement Steward has the right to modify this Agreement. IBM is the initial Agreement Steward. IBM may assign the responsibility to serve as the Agreement Steward to a suitable separate entity. Each new version of the Agreement will be given a distinguishing version number. The

Program (including Contributions) may always be distributed subject to the version of the Agreement under which it was received. In addition, after a new version of the Agreement is published, Contributor may elect to distribute the Program (including its Contributions) under the new version. Except as expressly stated in Sections 2(a) and 2(b) above, Recipient receives no rights or licenses to the intellectual property of any Contributor under this Agreement, whether expressly, by implication, estoppel or otherwise. All rights in the Program not expressly granted under this Agreement are reserved.

This Agreement is governed by the laws of the State of New York and the intellectual property laws of the United States of America. No party to this Agreement will bring a legal action under this Agreement more than one year after the cause of action arose. Each party waives its rights to a jury trial in any resulting litigation.

Introduction

This document summarizes in one place the available documentation from the Mondrian open source project, version 2.2.2. The contents are derived from documentation in the Mondrian code distribution.

The aim of this document is to provide a guide to the use of Mondrian, covering:

- Mondrian overview and architecture
- Developing OLAP schemas
- Querying cubes with MDX
- Tools and techniques for managing data and tuning query performance
- Integrating Mondrian into applications

The audience of this document is intended to be people creating and managing Mondrian based OLAP environments and developers who are integrating Mondrian into their applications.

Mondrian and OLAP

Copyright (C) 2002-2006 Julian Hyde

Mondrian is an Online Analytical Processing (OLAP) engine written in Java. It executes queries written in the MDX language, reading data from a relational database (RDBMS), and presents the results in a multidimensional format via a Java API. Let's go into what that means.

Online Analytical Processing

OLAP means analysing large quantities of data in real-time. Unlike Online Transaction Processing (OLTP), where typical operations read and modify individual and small numbers of records, OLAP deals with data in bulk, and operations are generally read-only. The term 'online' implies that even though huge quantities of data are involved — typically many millions of records, occupying several gigabytes — the system must respond to queries fast enough to allow an interactive exploration of the data. As we shall see, that presents considerable technical challenges.

OLAP employs a technique called Multidimensional Analysis. Whereas a relational database stores all data in the form of rows and columns, a multidimensional dataset consists of axes and cells. Consider the dataset

<i>Year</i>	2000		2001		Growth	
<i>Product</i>	Dollar sales	Unit sales	Dollar sales	Unit sales	Dollar sales	Unit sales
Total	\$7,073	2,693	\$7,636	3,008	8%	12%
— Books	\$2,753	824	\$3,331	966	21%	17%
— Fiction	\$1,341	424	\$1,202	380	-10%	-10%
— Non-fiction	\$1,412	400	\$2,129	586	51%	47%
— Magazines	\$2,753	824	\$2,426	766	-12%	-7%
— Greetings cards	\$1,567	1,045	\$1,879	1,276	20%	22%

The rows axis consists of the members 'All products', 'Books', 'Fiction', and so forth, and the columns axis consists of the cartesian product of the years '2000' and '2001', and the calculation 'Growth', and the measures 'Unit sales' and 'Dollar sales'. Each cell represents the sales of a product category in a particular year; for example, the dollar sales of Magazines in 2001 were \$2,426.

This is a richer view of the data than would be presented by a relational database. The members of a multidimensional dataset are not always values from a relational column. 'Total', 'Books' and 'Fiction' are members at successive levels in a hierarchy, each of which is rolled up to the next. And even though it is alongside the years '2000' and '2001', 'Growth' is a calculated member, which introduces a formula for computing cells from other cells.

The dimensions used here — products, time, and measures — are just three of many dimensions by which the dataset can be categorized and filtered. The collection of dimensions, hierarchies and measures is called a cube.

Although some multidimensional databases store the data in multidimensional format, I shall argue that it is simpler to store the data in relational format.

Mondrian Architecture

Copyright (C) 2001-2002 Kana Software, Inc.

Copyright (C) 2001-2007 Julian Hyde

Layers of a Mondrian system

A Mondrian OLAP System consists of four layers; working from the eyes of the end-user to the bowels of the data center, these are as follows: the presentation layer, the dimensional layer, the star layer, and the storage layer. (See [figure 1.](#))

The presentation layer determines what the end-user sees on his or her monitor, and how he or she can interact to ask new questions. There are many ways to present multidimensional datasets, including pivot tables (an interactive version of the table shown above), pie, line and bar charts, and advanced visualization tools such as clickable maps and dynamic graphics. These might be written in Swing or JSP, charts rendered in JPEG or GIF format, or transmitted to a remote application via XML. What all of these forms of presentation have in common is the multidimensional 'grammar' of dimensions, measures and cells in which the presentation layer asks the question is asked, and OLAP server returns the answer.

The second layer is the dimensional layer. The dimensional layer parses, validates and executes MDX queries. A query is evaluated in multiple phases. The axes are computed first, then the values of the cells within the axes. For efficiency, the dimensional layer sends cell-requests to the aggregation layer in batches. A query transformer allows the application to manipulate existing queries, rather than building an MDX statement from scratch for each request. And metadata describes the the dimensional model, and how it maps onto the relational model.

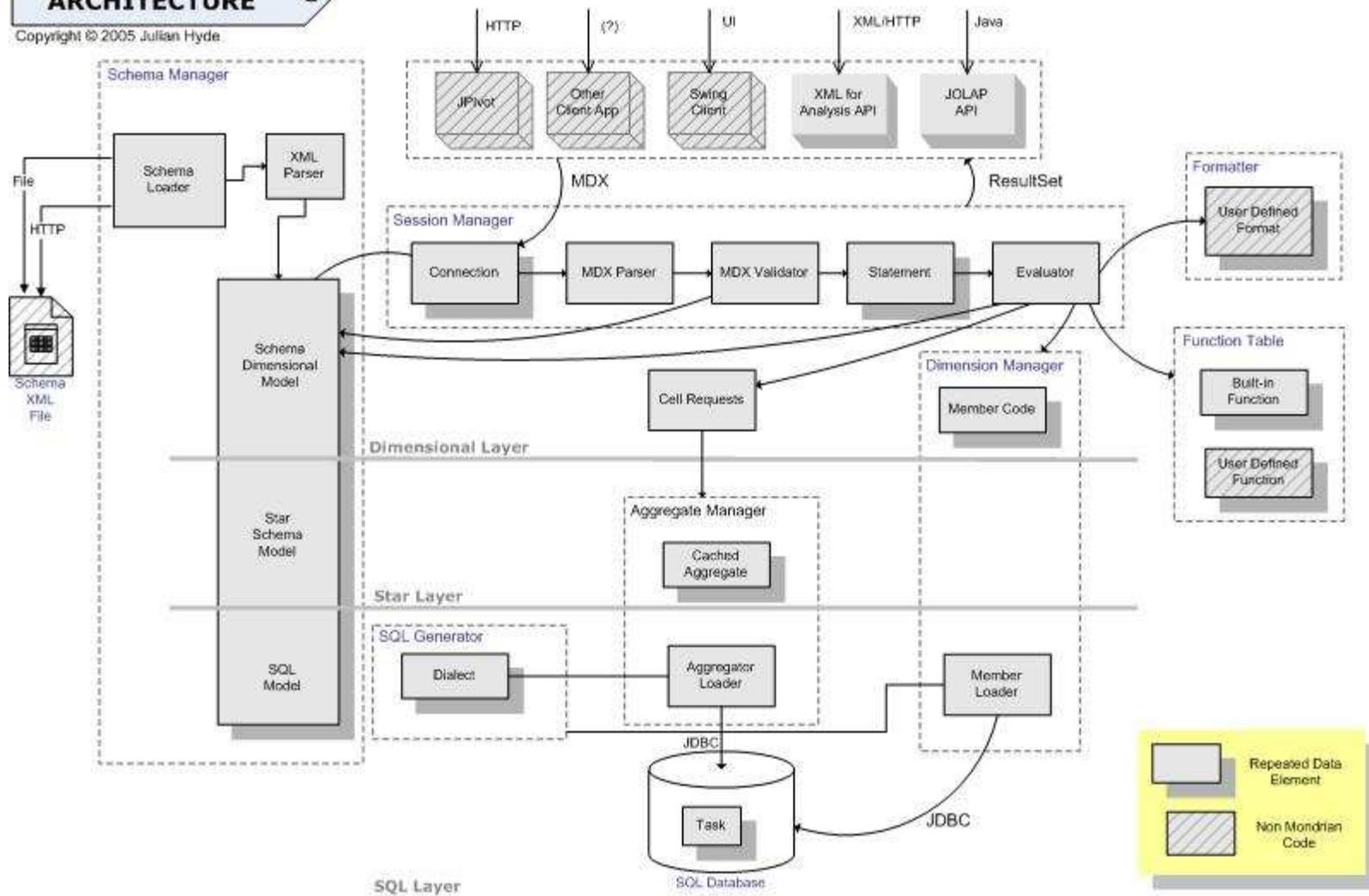
The third layer is the star layer, and is responsible for maintaining an aggregate cache. An aggregation is a set of measure values ('cells') in memory, qualified by a set of dimension column values. The dimensional layer sends requests for sets of cells. If the requested cells are not in the cache, or derivable by rolling up an aggregation in the cache, the aggregation manager and sends a request to the storage layer.

The storage layer is an RDBMS. It is responsible for providing aggregated cell data, and members from dimension tables. I describe [below](#) why I decided to use the features of the RDBMS rather than developing a storage system optimized for multidimensional data.

These components can all exist on the same machine, or can be distributed between machines. Layers 2 and 3, which comprise the Mondrian server, must be on the same machine. The storage layer could be on another machine, accessed via remote JDBC connection. In a multi-user system, the presentation layer would exist on each end-user's machine (except in the case of JSP pages generated on the server).

MONDRIAN ARCHITECTURE

Copyright © 2005 Julian Hyde



Storage and aggregation strategies

OLAP Servers are generally categorized according to how they store their data:

- A MOLAP (multidimensional OLAP) server stores all of its data on disk in structures optimized for multidimensional access. Typically, data is stored in dense arrays, requiring only 4 or 8 bytes per cell value.
- A ROLAP (relational OLAP) server stores its data in a relational database. Each row in a fact table has a column for each dimension and measure.

Three kinds of data need to be stored: fact table data (the transactional records), aggregates, and dimensions.

MOLAP databases store fact data in multidimensional format, but if there are more than a few dimensions, this data will be sparse, and the multidimensional format does not perform well. A HOLAP (hybrid OLAP) system solves this problem by leaving the most granular data in the relational database, but stores aggregates in multidimensional format.

Pre-computed aggregates are necessary for large data sets, otherwise certain queries could not be answered without reading the entire contents of the fact table. MOLAP aggregates are often an image of the in-memory data structure, broken up into pages and stored on disk. ROLAP aggregates are stored in tables. In some ROLAP systems these are explicitly managed by the OLAP server; in other systems, the tables are declared as materialized views, and they are implicitly used when the OLAP server issues a query with the right combination of columns in the group by clause.

The final component of the aggregation strategy is the cache. The cache holds pre-computed aggregations in memory so subsequent queries can access cell values without going to disk. If the cache holds the required data set at a lower level of aggregation, it can compute the required data set by rolling up.

The cache is arguably the most important part of the aggregation strategy because it is adaptive. It is difficult to choose a set of aggregations to pre-compute which speed up the system without using huge amounts of disk, particularly those with a high dimensionality or if the users are submitting unpredictable queries. And in a system where data is changing in real-time, it is impractical to maintain pre-computed aggregates. A reasonably sized cache can allow a system to perform adequately in the face of unpredictable queries, with few or no pre-computed aggregates.

Mondrian's aggregation strategy is as follows:

- Fact data is stored in the RDBMS. Why develop a storage manager when the RDBMS already has one?
- Read aggregate data into the cache by submitting group by queries. Again, why develop an aggregator when the RDBMS has one?
- If the RDBMS supports materialized views, and the database administrator chooses to create materialized views for particular aggregations, then Mondrian will use them implicitly. Ideally, Mondrian's aggregation manager should be aware that these

materialized views exist and that those particular aggregations are cheap to compute. If should even offer tuning suggestions to the database administrator.

The general idea is to delegate unto the database what is the database's. This places additional burden on the database, but once those features are added to the database, all clients of the database will benefit from them. Multidimensional storage would reduce I/O and result in faster operation in some circumstances, but I don't think it warrants the complexity at this stage.

A wonderful side-effect is that because Mondrian requires no storage of its own, it can be installed by adding a JAR file to the class path and be up and running immediately. Because there are no redundant data sets to manage, the data-loading process is easier, and Mondrian is ideally suited to do OLAP on data sets which change in real time.

API

Mondrian provides an API for client applications to execute queries.

Since there is no widely universally accepted API for executing OLAP queries, Mondrian's primary API proprietary; however, anyone who has used JDBC should find it familiar. The main difference is the query language: Mondrian uses a language called MDX ('**M**ulti-**D**imensional **eX**pressions') to specify queries, where JDBC would use SQL. MDX is described in more detail [below](#).

The following Java fragment connects to Mondrian, executes a query, and prints the results:

```
import mondrian.olap.*;
import java.io.PrintWriter;

Connection connection = DriverManager.getConnection(
    "Provider=mondrian;" +
    "Jdbc=jdbc:odbc:MondrianFoodMart;" +
    "Catalog=/WEB-INF/FoodMart.xml;",
    null,
    false);
Query query = connection.parseQuery(
    "SELECT {[Measures].[Unit Sales], [Measures].[Store Sales]} on columns," +
    " {[Product].children} on rows " +
    "FROM [Sales] " +
    "WHERE ([Time].[1997].[Q1], [Store].[CA].[San Francisco])");
Result result = connection.execute(query);
result.print(new PrintWriter(System.out));
```

A [Connection](#) is created via a [DriverManager](#), in a similar way to JDBC. A [Query](#) is analogous to a JDBC [Statement](#), and is created by parsing an MDX string. A [Result](#) is analogous to a JDBC [ResultSet](#); since we are dealing with multi-dimensional data, it consists of axes and cells, rather than rows and columns. Since OLAP is intended for data exploration, you can modify the parse tree contained in a query by operations such as [drillDown](#) and [sort](#), then re-execute the query.

The API also presents the database schema as a set of objects: [Schema](#), [Cube](#), [Dimension](#), [Hierarchy](#), [Level](#), [Member](#). For more information about the Mondrian API, see [the javadoc](#).

To comply with emerging standards, we are adding two APIs to Mondrian:

- [JOLAP](#) is a standard emerging from the JSR process, and it will become part of J2EE sometime in 2003. We have a few simple JOLAP queries running in [class mondrian.test.JolapTest](#).
- [XML for Analysis](#) is a standard for accessing OLAP servers via SOAP (Simple Object Access Protocol). This will allow non-Java components like Microsoft Excel to run queries against Mondrian.

How to Design a Mondrian Schema

Copyright (C) 2001-2002 Kana Software, Inc.
Copyright (C) 2002-2007 Julian Hyde and others

What is a schema?

A schema defines a multi-dimensional database. It contains a logical model, consisting of cubes, hierarchies, and members, and a mapping of this model onto a physical model.

The logical model consists of the constructs used to write queries in MDX language: cubes, dimensions, hierarchies, levels, and members.

The physical model is the source of the data which is presented through the logical model. It is typically a star schema, which is a set of tables in a relational database; later, we shall see examples of other kinds of mappings.

Schema files

Mondrian schemas are represented in an XML file. An example schema, containing almost all of the constructs we discuss here, is supplied as `demo/FoodMart.xml` in the mondrian distribution. The dataset to populate this schema is [also in the distribution](#).

Currently, the only way to create a schema is to edit a schema XML file in a text editor. The XML syntax is not too complicated, so this is not as difficult as it sounds, particularly if you use the FoodMart schema as a guiding example.

NOTE: The order of XML elements is important. For example, [<UserDefinedFunction>](#) element has to occur inside the [<Schema>](#) element after all collections of [<Cube>](#), [<VirtualCube>](#), [<NamedSet>](#) and [<Role>](#) elements. If you include it before the first [<Cube>](#) element, the rest of the schema will be ignored.

Logical model

The most important components of a schema are cubes, measures, and dimensions:

- A *cube* is a collection of dimensions and measures in a particular subject area.
- A *measure* is a quantity that you are interested in measuring, for example, unit sales of a product, or cost price of inventory items.
- A *dimension* is an attribute, or set of attributes, by which you can divide measures into sub-categories. For example, you might wish to break down product sales by their color, the gender of the customer, and the store in which the product was sold; color, gender, and store are all dimensions.

Let's look at the XML definition of a simple schema.

```
<Schema>  
<Cube name="Sales">  
<Table name="sales_fact_1997"/>  
<Dimension name="Gender" foreignKey="customer_id">
```

```

<Hierarchy hasAll="true" allMemberName="All Genders" primaryKey="customer_id">
  <Table name="customer"/>
  <Level name="Gender" column="gender" uniqueMembers="true"/>
</Hierarchy>
</Dimension>
<Dimension name="Time" foreignKey="time_id">
  <Hierarchy hasAll="false" primaryKey="time_id">
    <Table name="time_by_day"/>
    <Level name="Year" column="the_year" type="Numeric" uniqueMembers="true"/>
    <Level name="Quarter" column="quarter" uniqueMembers="false"/>
    <Level name="Month" column="month_of_year" type="Numeric" uniqueMembers="false"/>
  </Hierarchy>
</Dimension>
<Measure name="Unit Sales" column="unit_sales" aggregator="sum"
formatString="#,###"/>
  <Measure name="Store Sales" column="store_sales" aggregator="sum"
formatString="#,###.##"/>
  <CalculatedMember name="Profit" dimension="Measures" formula="[Measures].
[Store Sales]-[Measures].[Store Cost]">
    <CalculatedMemberProperty name="FORMAT_STRING" value="$#,##0.00"/>
  </CalculatedMember>
</Cube>
</Schema>

```

This schema contains a single cube, called "Sales". The Sales cube has two dimensions, "Time", and "Gender", and two measures, "Unit Sales" and "Store Sales".

We can write an MDX query on this schema:

```

SELECT {[Measures].[Unit Sales], [Measures].[Store Sales]} ON COLUMNS,
  {[Time].[1997].[Q1].descendants} ON ROWS
FROM [Sales]
WHERE [Gender].[F]

```

This query refers to the Sales cube ([Sales]), each of the dimensions [Measures], [Time], [Gender], and various members of those dimensions. The results are as follows:

[Time]	[Measures].[Unit Sales]	[Measures].[Store Sales]
[1997].[Q1]	0	0
[1997].[Q1].[Jan]	0	0
[1997].[Q1].[Feb]	0	0
[1997].[Q1].[Mar]	0	0

Now let's look at the schema definition in more detail.

Cube

A cube (see [<Cube>](#)) is a named collection of measures and dimensions. The one thing the measures and dimensions have in common is the fact table, here "sales_fact_1997". As we

shall see, the fact table holds the columns from which measures are calculated, and contains references to the tables which hold the dimensions.

```
<Cube name="Sales">
<Table name="sales_fact_1997"/>
...
</Cube>
```

The fact table is defined using the [<Table>](#) element. If the fact table is not in the default schema, you can provide an explicit schema using the "schema" attribute, for example

```
<Table schema=" dmart" name="sales_fact_1997"/>
```

You can also use the [<View>](#) and [<Join>](#) constructs to build more complicated SQL statements.

Measures

The Sales cube defines several measures, including "Unit Sales" and "Store Sales".

```
<Measure name="Unit Sales" column="unit_sales"
aggregator="sum" datatype="Integer" formatString="#,###"/>
<Measure name="Store Sales" column="store_sales"
aggregator="sum" datatype="Numeric" formatString="#,###.00"/>
```

Each measure (see [<Measure>](#)) has a name, a column in the fact table, and an aggregator. The aggregator is usually "sum", but "count", "mix", "max", "avg", and "distinct count" are also allowed; "distinct count" has some limitations if your cube contains a [parent-child hierarchy](#).

The optional `datatype` attribute specifies how cell values are represented in Mondrian's cache, and how they are returned via XML for Analysis. The `datatype` attribute can have values "String", "Integer" and "Numeric". The default is "Numeric", except for "count" and "distinct-count" measures, which are "Integer".

An optional `formatString` attribute specifies how the value is to be printed. Here, we have chosen to output unit sales with no decimal places (since it is an integer), and store sales with two decimal places (since it is a currency value). The ',' and '.' symbols are locale-sensitive, so if you were running in Italian, store sales might appear as "48.123,45". You can achieve even more wild effects using [advanced format strings](#).

A measure can have a caption attribute to be returned by the [Member.getCaption\(\)](#) method instead of the name. Defining a specific caption does make sense if special letters (e.g. Σ or Π) are to be displayed:

```
<Measure name="Sum X" column="sum_x" aggregator="sum" caption="Σ X"/>
```

Rather than coming from a column, a measure can use a [cell reader](#), or a measure can use a SQL expression to calculate its value. The measure "Promotion Sales" is an example of this.

```
<Measure name="Promotion Sales" aggregator="sum"
formatString="#,###.00">
```

```

<MeasureExpression>
<SQL dialect="generic">
(case when sales_fact_1997.promotion_id =
0 then 0 else sales_fact_1997.store_sales end)
</SQL>
</MeasureExpression>
</Measure>

```

In this case, sales are only included in the summation if they correspond to a promotion sales. Arbitrary SQL expressions can be used, including subqueries. However, the underlying database must be able to support that SQL expression in the context of an aggregate. Variations in syntax between different databases is handled by specifying the dialect in the SQL tag.

In order to provide a specific formatting of the cell values, a measure can use a [cell formatter](#).

Dimensions, Hierarchies, Levels

Some more definitions:

- A *member* is a point within a dimension determined by a particular set of attribute values. The gender hierarchy has the two members 'M' and 'F'. 'San Francisco', 'California' and 'USA' are all members of the store hierarchy.
- A *hierarchy* is a set of members organized into a structure for convenient analysis. For example, the store hierarchy consists of the store name, city, state, and nation. The hierarchy allows you form intermediate sub-totals: the sub-total for a state is the sum of the sub-totals of all of the cities in that state, each of which is the sum of the sub-totals of the stores in that city.
- A *level* is a collection of members which have the same distance from the root of the hierarchy.
- A *dimension* is a collection of hierarchies which discriminate on the same fact table attribute (say, the day that a sale occurred).

For reasons of uniformity, measures are treated as members of a special dimension, called 'Measures'.

An example

Let's look at a simple dimension.

```

<Dimension name="Gender" foreignKey="customer_id">
  <Hierarchy hasAll="true" primaryKey="customer_id">
    <Table name="customer"/>
    <Level name="Gender" column="gender" uniqueMembers="true"/>
  </Hierarchy>
</Dimension>

```

This dimension consists of a single hierarchy, which consists of a single level called `Gender`. (As we shall see [later](#), there is also a special level called [(All)] containing a grand total.)

The values for the dimension come from the `gender` column in the `customer` table. The "gender" column contains two values, 'F' and 'M', so the Gender dimension contains the members `[Gender].[F]` and `[Gender].[M]`.

For any given sale, the gender dimension is the gender of the customer who made that purchase. This is expressed by joining from the fact table "sales_fact_1997.customer_id" to the dimension table "customer.customer_id".

Mapping dimensions and hierarchies onto tables

A dimension is joined to a cube by means of a pair of columns, one in the fact table, the other in the dimension table. The `<Dimension>` element has a `foreignKey` attribute, which is the name of a column in the fact table; the `<Hierarchy>` element has `primaryKey` attribute.

If the hierarchy has more than one table, you can disambiguate using the `primaryKeyTable` attribute.

The `uniqueMembers` attribute is used to optimize SQL generation. If you know that the values of a given level column in the dimension table are unique across all the other values in that column across the parent levels, then set `uniqueMembers="true"`, otherwise, set to "false". For example, a time dimension like `[Year].[Month]` will have `uniqueMembers="false"` at the Month level, as the same month appears in different years. On the other hand, if you had a `[Product Class].[Product Name]` hierarchy, and you were sure that `[Product Name]` was unique, then you can set `uniqueMembers="true"`. If you are not sure, then always set `uniqueMembers="false"`. At the top level, this will always be `uniqueMembers="true"`, as there is no parent level.

The 'all' member

By default, every hierarchy contains a top level called '(All)', which contains a single member called '(All {*hierarchyName*})'. This member is parent of all other members of the hierarchy, and thus represents a grand total. It is also the default member of the hierarchy; that is, the member which is used for calculating cell values when the hierarchy is not included on an axis or in the slicer. The `allMemberName` and `allLevelName` attributes override the default names of the all level and all member.

If the `<Hierarchy>` element has `hasAll="false"`, the 'all' level is suppressed. The default member of that dimension will now be the first member of the first level; for example, in a Time hierarchy, it will be the first year in the hierarchy. Changing the default member can be confusing, so you should generally use `hasAll="true"`.

Time dimensions

Time dimensions based on year/month/week/day are coded differently in the Mondrian schema due to the MDX time related functions such as:

- `ParallelPeriod([level[, index[, member]])]`
- `PeriodsToDate([level[, member]])]`
- `WTD([member])]`

- MTD([member])
- QTD([member])
- YTD([member])
- LastPeriod(index[, member])

Time dimensions have `type="TimeDimension"`. The role of a level in a time dimension is indicated by the level's `levelType` attribute, whose allowable values are as follows:

`levelType` value **Meaning**

TimeYears	Level is a year
TimeQuarters	Level is a quarter
TimeMonths	Level is a month
TimeDays	Level represents days

Here is an example of a time dimension:

```
<Dimension name="Time" type="TimeDimension">
  <Hierarchy hasAll="true" allMemberName="All Periods"
primaryKey="dateid">
  <Table name="datehierarchy"/>
  <Level name="Year" column="year" uniqueMembers="true"
levelType="TimeYears" type="Numeric"/>
  <Level name="Quarter" column="quarter"
uniqueMembers="false" levelType="TimeQuarters" />
  <Level name="Month" column="month" uniqueMembers="false"
ordinalColumn="month" nameColumn="month_name"
levelType="TimeMonths" type="Numeric"/>
  <Level name="Week" column="week_in_month"
uniqueMembers="false" levelType="TimeWeeks" />
  <Level name="Day" column="day_in_month"
uniqueMembers="false" ordinalColumn="day_in_month"
nameColumn="day_name" levelType="TimeDays" type="Numeric"/>
  </Hierarchy>
</Dimension>
```

Order and display of levels

Notice that in the time hierarchy example above the `ordinalColumn` and `nameColumn` attributes on the `<Level>` element. These effect how levels are dispysyed in a result. The `ordinalColumn` attribute specifies a column in the Hierarchy table that provides the order of the members in a given Level, while the `nameColumn` specifies a column that will be displayed.

For example, in the Month Level above, the `datehierarchy` table has `month` (1 .. 12) and `month_name` (January, February, ...) columns. The column value that will be used internally within MDX is the `month` column, so valid member specifications will be of the form: `[Time].[2005].[Q1].[1]`. Members of the `[Month]` level will displayed in the order January, February, etc.

Multiple hierarchies

A dimension can contain more than one hierarchy:

```
<Dimension name="Time" foreignKey="time_id">
  <Hierarchy hasAll="false" primaryKey="time_id">
    <Table name="time_by_day"/>
    <Level name="Year" column="the_year" type="Numeric"
      uniqueMembers="true"/>
    <Level name="Quarter" column="quarter"
      uniqueMembers="false"/>
    <Level name="Month" column="month_of_year"
      type="Numeric" uniqueMembers="false"/>
  </Hierarchy>
  <Hierarchy name="Time Weekly" hasAll="false"
    primaryKey="time_id">
    <Table name="time_by_week"/>
    <Level name="Year" column="the_year" type="Numeric"
      uniqueMembers="true"/>
    <Level name="Week" column="week"
      uniqueMembers="false"/>
    <Level name="Day" column="day_of_week" type="String"
      uniqueMembers="false"/>
  </Hierarchy>
</Dimension>
```

Notice that the first hierarchy doesn't have a name. By default, a hierarchy has the same name as its dimension, so the first hierarchy is called "Time".

These hierarchies don't have much in common — they don't even have the same table! — except that they are joined from the same column in the fact table, "time_id". The main reason to put two hierarchies in the same dimension is because it makes more sense to the end-user: end-users know that it makes no sense to have the "Time" hierarchy on one axis and the "Time Weekly" hierarchy on another axis. If two hierarchies are the same dimension, the MDX language enforces common sense, and does not allow you to use them both in the same query.

Degenerate dimensions

A *degenerate dimension* is a dimension which is so simple that it isn't worth creating its own dimension table. For example, consider following the fact table:

product_id	time_id	payment_method	customer_id	store_id	item_count	dollars
55	20040106	Credit	123	22	3	\$3.54
78	20040106	Cash	89	22	1	\$20.00
199	20040107	ATM	3	22	2	\$2.99
55	20040106	Cash	122	22	1	\$1.18

and suppose we created a dimension table for the values in the `payment_method` column:

payment_method

Credit

Cash

ATM

This dimension table is fairly pointless. It only has 3 values, adds no additional information, and incurs the cost of an extra join.

Instead, you can create a degenerate dimension. To do this, declare a dimension without a table, and Mondrian will assume that the columns come from the fact table.

```
<Cube name="Checkout">
  <!-- The fact table is always necessary. -->
  <Table name="checkout">
    <Dimension name="Payment method">
      <Hierarchy hasAll="true">
        <!-- No table element here.
        Fact table is assumed. -->
        <Level name="Payment method"
        column="payment_method" uniqueMembers="true" />
      </Hierarchy>
    </Dimension>
    <!-- other dimensions and measures -->
  </Cube>
```

Note that because there is no join, the `foreignKey` attribute of `Dimension` is not necessary, and the `Hierarchy` element has no `<Table>` child element or `primaryKey` attribute.

Inline tables

The `<InlineTable>` construct allows you to define a dataset in the schema file. You must declare the names of the columns, the column types ("String" or "Numeric"), and a set of rows. As for `<Table>` and `<View>`, you must provide a unique alias with which to refer to the dataset.

Here is an example:

```
<Dimension name="Severity">
  <Hierarchy hasAll="true" primaryKey="severity_id">
    <InlineTable alias="severity">
      <ColumnDefs>
        <ColumnDef name="id" type="Numeric"/>
        <ColumnDef name="desc" type="String"/>
      </ColumnDefs>
      <Rows>
        <Row>
          <Value column="id">1</Value>
          <Value column="desc">High</Value>
        </Row>
        <Row>
          <Value column="id">2</Value>
```

```

    <Value column="desc">Medium</Value>
  </Row>
  <Row>
    <Value column="id">3</Value>
    <Value column="desc">Low</Value>
  </Row>
</Rows>
</InlineTable>
<Level name="Severity" column="id" nameColumn="desc"
uniqueMembers="true"/>
</Hierarchy>
</Dimension>

```

This has the same effect as if you had a table called 'severity' in your database:

id desc

- 1 High
- 2 Medium
- 3 Low

and the declaration

```

<Dimension name="Severity">
  <Hierarchy hasAll="true" primaryKey="severity_id">
    <Table name="severity"/>
    <Level name="Severity" column="id" nameColumn="desc"
uniqueMembers="true"/>
  </Hierarchy>
</Dimension>

```

To specify a NULL value for a column, omit the <Value> for that column, and the column's value will default to NULL.

Member properties and formatters

As we shall see later, a level definition can also define [member properties](#) and a [member formatter](#).

Approximate level cardinality

The [<Level>](#) element allows specifying the optional attribute "approxRowCount". Specifying approxRowCount can improve performance by reducing the need to determine level, hierarchy, and dimension cardinality. This can have a significant impact when connecting to Mondrian via XMLA.

Star and snowflake schemas

We saw earlier how to build a cube based upon a fact table, and dimensions in the fact table ("Payment method") and in a table joined to the fact table ("Gender"). This is the most common kind of mapping, and is known as a *star schema*.

But a dimension can be based upon more than one table, provided that there is a well-defined path to join these tables to the fact table. This kind of dimension is known as a snowflake, and is defined using the `<Join>` operator. For example:

```
<Cube name="Sales">
  ...
  <Dimension name="Product" foreignKey="product_id">
    <Hierarchy hasAll="true" primaryKey="product_id"
    primaryKeyTable="product">
      <Join leftKey="product_class_key" rightAlias="product_class"
      rightKey="product_class_id">
        <Table name="product"/>
        <Join leftKey="product_type_id" rightKey="product_type_id">
          <Table name="product_class"/>
          <Table name="product_type"/>
        </Join>
      </Join>
    <!-- Level declarations ... -->
  </Hierarchy>
</Dimension>
</Cube>
```

This defines a "Product" dimension consisting of three tables. The fact table joins to "product" (via the foreign key "product_id"), which joins to "product_class" (via the foreign key "product_class_id"), which joins to "product_type" (via the foreign key "product_type_id"). We require a `<Join>` element nested within a `<Join>` element because `<Join>` takes two operands; the operands can be tables, joins, or even queries.

The arrangement of the tables seems complex, the simple rule of thumb is to order the tables by the number of rows they contain. The "product" table has the most rows, so it joins to the fact table and appears first; "product_class" has fewer rows, and "product_type", at the tip of the snowflake, has least of all.

Note that the outer `<Join>` element has a `rightAlias` attribute. This is necessary because the right component of the join (the inner `<Join>` element) consists of more than one table. No `leftAlias` attribute is necessary in this case, because the `leftKey` column unambiguously comes from the "product" table.

Shared dimensions

When generating the SQL for a join, mondrian needs to know which column to join to. If you are joining to a join, then you need to tell it which of the tables in the join that column belongs to (usually it will be the first table in the join).

Because shared dimensions don't belong to a cube, you have to give them an explicit table (or other data source). When you use them in a particular cube, you specify the foreign key. This example shows the `Store Type` dimension being joined to the `Sales` cube using the `sales_fact_1997.store_id` foreign key, and to the `Warehouse` cube using the `warehouse.warehouse_store_id` foreign key:

```
<Dimension name="Store Type">
  <Hierarchy hasAll="true" primaryKey="store_id">
```

```

    <Table name="store"/>
    <Level name="Store Type" column="store_type" uniqueMembers="true"/>
  </Hierarchy>
</Dimension>

<Cube name="Sales">
  <Table name="sales_fact_1997"/>
  ...
  <DimensionUsage name="Store Type" source="Store Type"
foreignKey="store_id"/>
</Cube>

<Cube name="Warehouse">
  <Table name="warehouse"/>
  ...
  <DimensionUsage name="Store Type" source="Store Type"
foreignKey="warehouse_store_id"/>
</Cube>

```

Join optimization

The table mapping in the schema tells Mondrian how to get the data, but Mondrian is smart enough not to read the schema literally. It applies a number of optimizations when generating queries:

- If a dimension has a small number of members, Mondrian reads it into a cache on first use. See the [mondrian.rolap.LargeDimensionThreshold](#) property.
- If a dimension (or, more precisely, the level of the dimension being accessed) is in the fact table, Mondrian does not perform a join.
- If two dimensions access the same table via the same join path, Mondrian only joins them once. For example, [Gender] and [Age] might both be columns in the customers table, joined via `sales_1997.cust_id = customers.cust_id`.

Advanced logical constructs

Virtual cubes

Defined by the [<VirtualCube>](#) element. (To be continued...)

Parent-child hierarchies

A conventional hierarchy has a rigid set of levels, and members which adhere to those levels. For example, in the `Product` hierarchy, any member of the `Product Name` level has a parent in the `Brand Name` level, which has a parent in the `Product Subcategory` level, and so forth. This structure is sometimes too rigid to model real-world data.

A *parent-child hierarchy* has only one level (not counting the special 'all' level), but any member can have parents in the same level. A classic example is the reporting structure in the `Employees` hierarchy:

```

<Dimension name="Employees" foreignKey="employee_id">
  <Hierarchy hasAll="true" allMemberName="All Employees"
primaryKey="employee_id">
    <Table name="employee" />
    <Level name="Employee Id" uniqueMembers="true" type="Numeric"
      column="employee_id" nameColumn="full_name"
      parentColumn="supervisor_id" nullParentValue="0">
      <Property name="Marital Status" column="marital_status" />
      <Property name="Position Title" column="position_title" />
      <Property name="Gender" column="gender" />
      <Property name="Salary" column="salary" />
      <Property name="Education Level" column="education_level" />
      <Property name="Management Role" column="management_role" />
    </Level>
  </Hierarchy>
</Dimension>

```

The important attributes here are `parentColumn` and `nullParentValue`:

- The `parentColumn` attribute is the name of the column which links a member to its parent member; in this case, it is the foreign key column which points to an employee's supervisor. The `<ParentExpression>` child element of `<Level>` is equivalent to the `parentColumn` attribute, but allows you to define an arbitrary SQL expression, just like the `<Expression>` element. The `parentColumn` attribute (or `<ParentExpression>` element) is the only indication to Mondrian that a hierarchy has a parent-child structure.
- The `nullParentValue` attribute is the value which indicates that a member has no parent. The default is `nullParentValue="null"`, but since many database don't index null values, schema designers sometimes use values as the empty string, 0, and -1 instead.

Tuning parent-child hierarchies

There's one serious problem with the parent-child hierarchy defined above, and that is the amount of work Mondrian has to do in order to compute cell-totals. Let's suppose that the employee table contains the following data:

employee		
supervisor_id	employee_id	full_name
null	1	Frank
1	2	Bill
2	3	Eric
1	4	Jane
3	5	Mark
2	6	Carla

If we want to compute the total salary budget for Bill, we need to add in the salaries of Eric and Carla (who report to Bill) and Mark (who reports to Eric). Usually Mondrian generates a SQL `GROUP BY` statement to compute these totals, but there is no (generally available) SQL construct which can traverse hierarchies. So by default, Mondrian generates one SQL statement per supervisor, to retrieve and total all of that supervisor's direct reports.

This approach has a couple of drawbacks. First, the performance is not very good if a hierarchy contains more than a hundred members. Second, because Mondrian implements the "distinct count" aggregator by generating SQL, you cannot define a "distinct count" member in any cube which contains a parent-child hierarchy.

How can we solve these problems? The answer is to enhance the data so that Mondrian is able to retrieve the information it needs using standard SQL. Mondrian supports a mechanism called a *closure table* for this purpose.

Closure tables

A closure table is a SQL table which contains a record for every employee/supervisor relationship, regardless of depth. (In mathematical terms, this is called the 'reflexive transitive closure' of the employee/supervisor relationship. The `distance` column is not strictly required, but it makes it easier to populate the table.)

employee_closure		
supervisor_id	employee_id	distance
1	1	0
1	2	1
1	3	2
1	4	1
1	5	3
1	6	2
2	2	0
2	3	1
2	5	2
2	6	1
3	3	0
3	5	1
4	4	0
5	5	0
6	6	0

In the catalog XML, the `<Closure>` element maps the level onto a `<Table>`:

```
<Dimension name="Employees" foreignKey="employee_id">
  <Hierarchy hasAll="true" allMemberName="All Employees"
    primaryKey="employee_id">
    <Table name="employee"/>
    <Level name="Employee Id" uniqueMembers="true" type="Numeric"
      column="employee_id" nameColumn="full_name"
      parentColumn="supervisor_id" nullParentValue="0">
      <Closure parentColumn="supervisor_id" childColumn="employee_id">
        <Table name="employee_closure"/>
      </Closure>
    <Property name="Marital Status" column="marital_status"/>
    <Property name="Position Title" column="position_title"/>
  </Hierarchy>
</Dimension>
```

```

    <Property name="Gender" column="gender"/>
    <Property name="Salary" column="salary"/>
    <Property name="Education Level" column="education_level"/>
    <Property name="Management Role" column="management_role"/>
  </Hierarchy>
</Dimension>

```

This table allows totals to be evaluated in pure SQL. Even though this introduces an extra table into the query, database optimizers are very good at handling joins. I recommend that you declare both `supervisor_id` and `employee_id` NOT NULL, and index them as follows:

```

CREATE UNIQUE INDEX employee_closure_pk ON employee_closure (
    supervisor_id,
    employee_id
);
CREATE INDEX employee_closure_emp ON employee_closure (
    employee_id
);

```

The table needs to be re-populated whenever the hierarchy changes, and it is the application's responsibility to do so -- Mondrian does not do this! Here is an example of a stored procedure that computes a closure table.

```

CREATE PROCEDURE close_employee()
BEGIN
    DECLARE distance int;
    TRUNCATE TABLE employee_closure;
    SET distance = 0;
    -- seed closure with self-pairs (distance 0)
    INSERT INTO employee_closure (supervisor_id, employee_id, distance)
        SELECT employee_id, employee_id, distance
        FROM employee;

    -- for each pair (root, leaf) in the closure,
    -- add (root, leaf->child) from the base table
    REPEAT
        SET distance = distance + 1;
        INSERT INTO employee_closure (supervisor_id, employee_id, distance)
            SELECT employee_closure.supervisor_id, employee.employee_id,
distance
            FROM employee_closure, employee
            WHERE employee_closure.employee_id = employee.supervisor_id
            AND employee_closure.distance = distance - 1;
    UNTIL (ROW_COUNT() == 0)
    END REPEAT
END

```

Member properties

Member properties are defined by the `<Property>` element within a `<Level>`, like this:

```

<Level name="MyLevel" column="LevelColumn" uniqueMembers="true"/>
<Property name="MyProp" column="PropColumn"

```

```
formatter="com.acme.MyPropertyFormatter"/>
<Level/>
```

The `formatter` attribute defines a [property formatter](#), which is explained later.

Once properties have been defined in the schema, you can use them in MDX statements via the `member.Properties("propertyName")` function, for example:

```
SELECT {[Store Sales]} ON COLUMNS,
    TopCount(Filter([Store].[Store Name].Members,
        [Store].CurrentMember.Properties("Store Type") =
"Supermarket"),
    10,
    [Store Sales]) ON ROWS
FROM [Sales]
```

Mondrian deduces the type of the property expression, if it can. If the property name is a constant string, the type is based upon the type attribute ("String", "Numeric" or "Boolean") of the property definition. If the property name is an expression (for example `CurrentMember.Properties("Store " + "Type")`), Mondrian will return an untyped value.

Calculated members

Suppose you want to create a measure whose value comes not from a column of the fact table, but from an MDX formula. One way to do this is to use a `WITH MEMBER` clause, like this:

```
WITH MEMBER [Measures].[Profit] AS '[Measures].[Store Sales]-
[Measures].[Store Cost]',
    FORMAT_STRING = '$#,###'
SELECT {[Measures].[Store Sales], [Measures].[Profit]} ON COLUMNS,
    {[Product].Children} ON ROWS
FROM [Sales]
WHERE [Time].[1997]
```

But rather than including this clause in every MDX query of your application, you can define the member in your schema, as part of your cube definition:

```
<CalculatedMember name="Profit" dimension="Measures">
    <Formula>[Measures].[Store Sales] - [Measures].[Store Cost]</Formula>
    <CalculatedMemberProperty name="FORMAT_STRING" value="$#,##0.00"/>
</CalculatedMember>
```

You can also declare the formula as an XML attribute, if you prefer. The effect is just the same.

```
<CalculatedMember name="Profit" dimension="Measures"
    formula="[Measures].[Store Sales]-[Measures].[Store Cost]">
    <CalculatedMemberProperty name="FORMAT_STRING" value="$#,##0.00"/>
</CalculatedMember>
```

Note that the `<CalculatedMemberProperty >` (not `<Property>`) element corresponds to the `FORMAT_STRING = '$#,###'` fragment of the MDX statement. You can define other properties here too, but `FORMAT_STRING` is by far the most useful in practice.

The `FORMAT_STRING` property value can also be evaluated using an expression. When formatting a particular cell, first the expression is evaluated to yield a format string, then the format string is applied to the cell value. Here is the same property with a conditional format string:

```
<CalculatedMemberProperty name="FORMAT_STRING" expression="Iif(Value
< 0, '|($#,##0.00)|style=red', '|$#,##0.00|style=green)"/>
```

For more details about format strings, see the [MDX specification](#).

You can make a calculated member or a measure invisible. If you specify `visible="false"` (the default is "true") in the `<Measure>` or `<CalculatedMember>` element, user-interfaces such as JPivot will notice this property and hide the member. This is useful if you want to perform calculations in a number of steps, and hide intermediate steps from end-users. For example, here only "Margin per Sqft" is visible, and its factors "Store Cost", "Margin" and "Store Sqft" are hidden:

```
<Measure
  name="Store Cost"
  column="store_cost"
  aggregator="sum"
  formatString="# ,###.00"
  visible="false"/>
<CalculatedMember
  name="Margin"
  dimension="Measures"
  visible="false">
  <Formula>([Measures].[Store Sales] - [Measures].[Store Cost]) /
[Measures].[Store Cost]</Formula>
<CalculatedMember
  name="Store Sqft"
  dimension="Measures"
  visible="false">
  <Formula>[Store].Properties("Sqft")</Formula>
<CalculatedMember
  name="Margin per Sqft"
  dimension="Measures"
  visible="true">
  <Formula>[Measures].[Margin] / [Measures].[Store Cost]</Formula>
  <CalculatedMemberProperty name="FORMAT_STRING" value="$#,##0.00"/>
</CalculatedMember>
```

Named sets

The `WITH SET` clause of an MDX statement allows you to declare a set expression which can be used throughout that query. For example,

```
WITH SET [Top Sellers] AS
  'TopCount([Warehouse].[Warehouse Name].MEMBERS, 5,
```

```
[Measures].[Warehouse Sales]]'
SELECT
    {[Measures].[Warehouse Sales]} ON COLUMNS,
    {[Top Sellers]} ON ROWS
FROM [Warehouse]
WHERE [Time].[Year].[1997]
```

The WITH SET clause is very similar to the WITH MEMBER clause, and as you might expect, it has a construct in schema analogous to < [CalculatedMember](#)>. The <NamedSet> element allows you to define a named set in your schema as part of a cube definition. It is implicitly available for any query against that cube:

```
<Cube name="Warehouse">
    ...
    <NamedSet name="Top Sellers">
        <Formula>TopCount([Warehouse].[Warehouse Name].MEMBERS, 5,
[Measures].[Warehouse Sales])</Formula>
    </NamedSet>
</Cube>
SELECT
    {[Measures].[Warehouse Sales]} ON COLUMNS,
    {[Top Sellers]} ON ROWS
FROM [Warehouse]
WHERE [Time].[Year].[1997]
```

Warehouse	Warehouse Sales
Treehouse Distribution	31,116.37
Jorge Garcia, Inc.	30,743.77
Artesia Warehousing, Inc.	29,207.96
Jorgensen Service Storage	22,869.79
Destination, Inc.	22,187.42

A named set defined against a cube is not inherited by a virtual cubes defined against that cube. (But you can define a named set against a virtual cube.)

You can also define a named set as global to a schema:

```
<Schema>
    <Cube name="Sales" ... />
    <Cube name="Warehouse" ... />
    <VirtualCube name="Warehouse and Sales" .../>
    <NamedSet name="CA Cities" formula="{[Store].[USA].[CA].Children}"/>
    <NamedSet name="Top CA Cities">
        <Formula>TopCount([CA Cities], 2, [Measures].[Unit
Sales])</Formula>
    </NamedSet>
</Schema>
```

A named set defined against a schema is available in all cubes and virtual cubes in that schema. However, it is only valid if the cube contains dimensions with the names required to make the formula valid. For example, it would be valid to use [CA Cities] in queries against the [Sales] and [Warehouse and Sales] cubes, but if you used it in a query against the

[Warehouse] cube you would get an error, because [Warehouse] does not have a [Store] dimension.

Plug-ins

Sometimes Mondrian's schema language isn't flexible enough, or the MDX language isn't powerful enough, to solve the problem at hand. What you want to do is add a little of your own Java code into the Mondrian application, and a *plug-in* is a way to do this.

Each of Mondrian's extensions is technically a Service Provider Interface (SPI); in short, a Java interface which you write code to implement, and which Mondrian will call at runtime. You also need to register an extension (usually somewhere in your schema.xml file) and to ensure that it appears on the classpath.

Plug-ins include [user-defined functions](#); [cell](#), [member](#) and [property formatters](#); and [dynamic schema processors](#). There is incomplete support for [member readers](#) and [cell readers](#), and in future we may support pluggable [SQL dialects](#).

User-defined function

A user-defined function must have a public constructor and implement the [mondrian.spi.UserDefinedFunction](#) interface. For example,

```
package com.acme;

import mondrian.olap.*;
import mondrian.olap.type.*;
import mondrian.spi.UserDefinedFunction;

/**
 * A simple user-defined function which adds one to its argument.
 */
public class PlusOneUdf implements UserDefinedFunction {
    // public constructor
    public PlusOneUdf() {
    }

    public String getName() {
        return "PlusOne";
    }

    public String getDescription() {
        return "Returns its argument plus one";
    }

    public Syntax getSyntax() {
        return Syntax.Function;
    }

    public Type getReturnType(Type[] parameterTypes) {
        return new NumericType();
    }
}
```

```

public Type[] getParameterTypes() {
    return new Type[] {new NumericType()};
}

public Object execute(Evaluator evaluator, Exp[] arguments) {
    final Object argValue = arguments[0].evaluateScalar(evaluator);
    if (argValue instanceof Number) {
        return new Double(((Number) argValue).doubleValue() + 1);
    } else {
        // Argument might be a RuntimeException indicating that
        // the cache does not yet have the required cell value. The
        // function will be called again when the cache is loaded.
        return null;
    }
}

public String[] getReservedWords() {
    return null;
}
}

```

Declare it in your schema:

```

<Schema>
    ...
    <UserDefinedFunction name="PlusOne" class="com.acme.PlusOneUdf">
</Schema>

```

And use it in any MDX statement:

```

WITH MEMBER [Measures].[Unit Sales Plus One]
    AS 'PlusOne([Measures].[Unit Sales])'
SELECT
    {[Measures].[Unit Sales]} ON COLUMNS,
    {[Gender].MEMBERS} ON ROWS
FROM [Sales]

```

If a user-defined function has a public constructor with one string argument, Mondrian will pass in the function's name. Why? This allows you to define two or more user-defined functions using the same class:

```

package com.acme;

import mondrian.olap.*;
import mondrian.olap.type.*;
import mondrian.spi.UserDefinedFunction;

/**
 * A user-defined function which either adds one to or
 * subtracts one from its argument.
 */
public class PlusOrMinusOneUdf implements UserDefinedFunction {
    private final name;
    private final isPlus;
}

```

```

// public constructor with one argument
public PlusOneUdf(String name) {
    this.name = name;
    if (name.equals("PlusOne")) {
        isPlus = true;
    } else if (name.equals("MinusOne")) {
        isPlus = false;
    } else {
        throw new IllegalArgumentException("Unexpected name " +
name);
    }
}

public String getName() {
    return name;
}

public String getDescription() {
    return "Returns its argument plus or minus one";
}

public Syntax getSyntax() {
    return Syntax.Function;
}

public Type getReturnType(Type[] parameterTypes) {
    return new NumericType();
}

public Type[] getParameterTypes() {
    return new Type[] {new NumericType()};
}

public Object execute(Evaluator evaluator, Exp[] arguments) {
    final Object argValue = arguments[0].evaluateScalar(evaluator);
    if (argValue instanceof Number) {
        if (isPlus) {
            return new Double(((Number) argValue).doubleValue() +
1);
        } else {
            return new Double(((Number) argValue).doubleValue() -
1);
        }
    } else {
        // Argument might be a RuntimeException indicating that
        // the cache does not yet have the required cell value. The
        // function will be called again when the cache is loaded.
        return null;
    }
}

public String[] getReservedWords() {
    return null;
}
}

```

and register two the functions in your schema:

```
<Schema>
...
<UserDefinedFunction name="PlusOne"
class="com.acme.PlusOrMinusOneUdf">
  <UserDefinedFunction name="MinusOne"
class="com.acme.PlusOrMinusOneUdf">
</Schema>
```

If you're tired of writing duplicated User-defined Function declarations in schema files, you can pack your User-defined Function implementation classes into a jar file with a embedded resource file META-INF/services/mondrian.spi.UserDefinedFunction. This resource file contains class names of implementations of interface mondrian.spi.UserDefinedFunction, one name per line. For more details, you may look into src/main/META-INF/services/mondrian.spi.UserDefinedFunction in source ball and [Service Provider](#). User-defined Functions declared by this means are available to all mondrian schema in one JVM.

Caution: you can't define more than one User-defined Function implementations in one class when you declare User-defined Functions in this way.

Member reader

A *member reader* is a means of accessing members. Hierarchies are usually based upon a dimension table (an 'arm' of a star schema), and are therefore populated using SQL. But even if your data doesn't reside in an RDBMS, you can make it appear as a hierarchy by writing a Java class called a *custom member reader*.

Here are a couple of examples:

1. `DateSource` (to be written) generates a time hierarchy. Conventionally, data warehouse implementors generate a table containing a row for every date their system is ever likely to deal with. But the problem is that this table needs to be loaded, and as time goes by, they will have to remember to add more rows. `DateSource` generates date members in memory, and on demand.
2. `FileSystemSource` (to be written) presents the file system as a hierarchy of directories and files. Since a directory can have a parent which is itself a directory, it is a parent-child hierarchy. Like the time hierarchy created by `DateSource`, this is a virtual hierarchy: the member for a particular file is only created when, and if, that file's parent directory is expanded.
3. `ExpressionMemberReader` (to be written) creates a hierarchy based upon an expression.

A custom member reader must implement the [mondrian.rolap.MemberSource](#) interface. If you need to implement a larger set of member operations for fine-grained control, implement the derived [mondrian.rolap.MemberReader](#) interface; otherwise, Mondrian wrap your reader in a [mondrian.rolap.CacheMemberReader](#) object. Your member reader must have a public constructor which takes ([RolapHierarchy](#), [Properties](#)) parameters, and throws no checked exceptions.

Member readers are declared using the [<Hierarchy>](#) element's `memberReaderClass` attribute; any [<Parameter>](#) child elements are passed via the `properties` constructor parameter. Here is an example:

```
<Dimension name="Has bought dairy">
  <Hierarchy hasAll="true"
memberReaderClass="mondrian.rolap.HasBoughtDairySource">
  <Level name="Has bought dairy" uniqueMembers="true"/>
  <Parameter name="expression" value="not used"/>
  </Hierarchy>
</Dimension>
```

Cell reader

Not implemented yet. Syntax would be something like

```
<Measure name="name" cellReaderClass="com.acme.MyCellReader"/>
```

and the class "com.acme.MyCellReader" would have to implement the [mondrian.olap.CellReader](#) interface.

Cell formatter

A cell formatter modifies the behavior of [Cell.getFormattedValue\(\)](#). The class must implement the [mondrian.olap.CellFormatter](#) interface, and is specified like this:

```
<Measure name="name" formatter="com.acme.MyCellFormatter"/>
```

Member formatter

A member formatter modifies the behavior of [Member.getCaption\(\)](#). The class must implement the [mondrian.olap.MemberFormatter](#) interface, and is specified like this:

```
<Level column="column" name="name"
formatter="com.acme.MyMemberFormatter"/>
```

Property formatter

A property formatter modifies the behavior of [Property.getPropertyFormattedValue\(\)](#). The class must implement the [mondrian.olap.PropertyFormatter](#) interface, and is specified like this:

```
<Level name="MyLevel" column="LevelColumn" uniqueMembers="true"/>
<Property name="MyProp" column="PropColumn"
formatter="com.acme.MyPropertyFormatter"/>
</Level/>
```

Schema processor

A schema processor implements the [mondrian.rolap.DynamicSchemaProcessor](#) interface. It is specified as part of the connection string, like this:

```
Jdbc=jdbc:odbc:MondrianFoodMart; JdbcUser=ziggy; JdbcPassword=stardust;  
DynamicSchemaProcessor=com.acme.MySchemaProcessor
```

The effect is that when reading the contents of the schema from a URL, Mondrian turns to the schema processor rather than Java's default URL handler. This gives the schema reader the opportunity to run a schema through a filter, or even generate an entire schema on the fly.

Dynamic schemas are a very powerful construct. As we shall see, an important application for them is [internationalization](#).

Internationalization

An internationalized Mondrian application would have a schema for each language, where the caption of each object appears in the local language. For example, the [Product] dimension would have the caption "Product" in English and "Produit" in French.

It is unwise to translate the actual names of the schema objects, because then the MDX statements would need to be changed also. All that you need to change is the caption. Every schema object (schema, cube, dimension, level, measure) has a caption attribute, and user interfaces such as JPivot display the caption rather than the real name. Additionally:

- A hierarchy can have an `allMemberCaption` attribute as display value of the "All" member.
- For the schema we can set a display value of the "measures" dimension by the `measuresCaption` attribute.

One way to create an internationalized application is to create a copy of the schema file for each language, but these are difficult to maintain. A better way is to use the [LocalizingDynamicSchemaProcessor](#) class to perform dynamic substitution on a single schema file.

Localizing schema processor

First, write your schema using variables as values for `caption`, `allMemberCaption` and `measuresCaption` attributes as follows:

```
<Schema measuresCaption="%{foodmart.measures.caption}">  
  
  <Dimension name="Store"  
  caption="%{foodmart.dimension.store.caption}">  
    <Hierarchy hasAll="true" allMemberName="All Stores"  
    allMemberCaption = "%{foodmart.dimension.store.allmember.caption =All  
    Stores}" primaryKey="store_id">  
      <Table name="store"/>  
      <Level name="Store Country" column="store_country"
```

```

uniqueMembers="true" caption=
"%{foodmart.dimension.store.country.caption}"/>
  <Level name="Store State" column="store_state"
uniqueMembers="true" caption=
"%{foodmart.dimension.store.state.caption}"/>
  <Level name="Store City" column="store_city"
uniqueMembers="false" caption=
"%{foodmart.dimension.store.city.caption}"/>
  <Level name="Store Name" column="store_name" uniqueMembers="true"
caption= "%{foodmart.dimension.store.name.caption}">
    <Property name="Store Type" column="store_type" caption=
"%{foodmart.dimension.store.name.property_type.caption}"/>
    <Property name="Store Manager" column="store_manager" caption=
"%{foodmart.dimension.store.name.property_manager.caption}"/>
    <Property name="Store Sqft" column="store_sqft" type="Numeric"
caption= "%{foodmart.dimension.store.
name.property_storesqft.caption}"/>
    <Property name="Grocery Sqft" column="grocery_sqft"
type="Numeric"/>
    <Property name="Frozen Sqft" column="frozen_sqft"
type="Numeric"/>
    <Property name="Meat Sqft" column="meat_sqft" type="Numeric"/>
    <Property name="Has coffee bar" column="coffee_bar"
type="Boolean"/>
    <Property name="Street address" column="store_street_address"
type="String"/>
  </Level>
</Hierarchy>
</Dimension>

<Cube name="Sales" caption="%{foodmart.cube.sales.caption}">
  ...
  <DimensionUsage name="Store" source="Store" foreignKey="store_id"/>
  ...
  <Measure name="Unit Sales" column="unit_sales"
caption="%{foodmart.cube.sales.measure.unitsales}">

```

As usual, the default caption for any cube, measure, dimension or level without a `caption` attribute is the name of the element. A hierarchy's default caption is the caption of its dimension; for example, the `[Store]` hierarchy has no `caption` defined, so it inherits the `caption` attribute from its parent, the `[Store]` dimension.

Next, add the dynamic schema processor and locale to your connect string. For example,

```

Provider=mondrian; Locale=en_US; DynamicSchemaProcessor=
mondrian.i18n.LocalizingDynamicSchemaProcessor; Jdbc=
jdbc:odbc:MondrianFoodMart; Catalog= /WEB-INF/FoodMart.xml

```

Now, for each locale you wish to support, provide a resource file named `locale_{locale}.properties`. For example,

```

# locale.properties: Default resources
foodmart.measures.caption=Measures
foodmart.dimension.store.country.caption=Store Country
foodmart.dimension.store.name.property_type.column= store_type

```

```

foodmart.dimension.store.country.member.caption= store_country
foodmart.dimension.store.name.property_type.caption =Store Type
foodmart.dimension.store.name.caption =Store Name
foodmart.dimension.store.state.caption =Store State
foodmart.dimension.store.name.property_manager.caption =Store Manager
foodmart.dimension.store.name.property_storesqft.caption =Store Sq. Ft.
foodmart.dimension.store.allmember.caption =All Stores
foodmart.dimension.store.caption =Store
foodmart.cube.sales.caption =Sales
foodmart.dimension.store.city.caption =Store City
foodmart.cube.sales.measure.unitsales =Unit Sales

```

and

```

# locale_hu.properties: Resources for the 'hu' locale.
foodmart.measures.caption=Hungarian Measures
foodmart.dimension.store.country.caption=Orsz\u00E1g
foodmart.dimension.store.name.property_manager.caption
=\u00C1ruh\u00E1z vezet\u0151
foodmart.dimension.store.country.member.caption
=store_country_caption_hu
foodmart.dimension.store.name.property_type.caption =Tipusa
foodmart.dimension.store.name.caption =Megnevez\u00E9s
foodmart.dimension.store.state.caption =\u00C1llam/Megye
foodmart.dimension.store.name.property_type.column
=store_type_caption_hu
foodmart.dimension.store.name.property_storesqft.caption =M\u00E9ret
n.l\u00E9b
foodmart.dimension.store.allmember.caption =Minden \u00C1ruh\u00E1z
foodmart.dimension.store.caption =\u00C1ruh\u00E1z
foodmart.cube.sales.caption =Forgalom
foodmart.dimension.store.city.caption =V\u00E9ros
foodmart.cube.sales.measure.unitsales =Eladott db

```

Aggregate tables

Aggregate tables are a way to improve Mondrian's performance when the fact table contains a huge number of rows: a million or more. An aggregate table is essentially a pre-computed summary of the data in the fact table.

Let's look at a simple aggregate table.

```

<Cube name="Sales">
  <Table name="sales_fact_1997">
    <AggName name="agg_c_special_sales_fact_1997">
      <AggFactCount column="FACT_COUNT"/>
      <AggMeasure name="[Measures].[Store Cost]"
column="STORE_COST_SUM"/>
      <AggMeasure name="[Measures].[Store Sales]"
column="STORE_SALES_SUM"/>
      <AggLevel name="[Product].[Product Family]"
column="PRODUCT_FAMILY"/>
      <AggLevel name="[Time].[Quarter]" column="TIME_QUARTER"/>
      <AggLevel name="[Time].[Year]" column="TIME_YEAR"/>
      <AggLevel name="[Time].[Quarter]" column="TIME_QUARTER"/>
    </AggName>
  </Table>
</Cube>

```

```

    <AggLevel name="[Time].[Month]" column="TIME_MONTH"/>
  </AggName>
</Table>

<!-- Rest of the cube definition -->
</Cube>

```

The [<AggForeignKey>](#) element, not shown here, allows you to reference a dimension table directly, without including its columns in the aggregate table. It is described in the [aggregate tables guide](#).

In practice, a cube which is based upon a very large fact table may have several aggregate tables. It is inconvenient to declare each aggregate table explicitly in the schema XML file, and luckily there is a better way. In the following example, Mondrian locates aggregate tables by pattern-matching.

```

<Cube name="Sales">
  <Table name="sales_fact_1997">
    <AggPattern pattern="agg_*_sales_fact_1997"/>
    <AggExclude name="agg_c_14_sales_fact_1997"/>
    <AggExclude name="agg_lc_100_sales_fact_1997"/>
  </Table>
</Cube>

```

It tells Mondrian to treat all tables which match the pattern "agg_*_sales_fact_1997" as aggregate tables, except "agg_c_14_sales_fact_1997" and "agg_lc_100_sales_fact_1997". Mondrian uses rules to deduce the roles of the columns in those tables, so it's important to adhere to strict naming conventions. The naming conventions are described in the [aggregate tables guide](#).

The performance guide has advice on [choosing aggregate tables](#).

Access-control

OK, so now you've got all this great data, but you don't everyone to be able to read all of it. To solve this, you can define an access-control profile, called a *Role*, as part of the schema, and set this role when establishing a connection.

Defining a role

Roles are defined by [<Role>](#) elements, which occur as direct children of the [<Schema>](#) element, after the last [<Cube>](#). Here is an example of a role:

```

<Role name="California manager">
  <SchemaGrant access="none">
    <CubeGrant cube="Sales" access="all">
      <HierarchyGrant hierarchy="[Store]" access="custom"
topLevel="[Store].[Store Country]">
        <MemberGrant member="[Store].[USA].[CA]" access="all"/>
        <MemberGrant member="[Store].[USA].[CA].[Los Angeles]"
access="none"/>
      </HierarchyGrant>
    </CubeGrant>
  </SchemaGrant>

```

```

    <HierarchyGrant hierarchy="[Customers]" access="custom"
topLevel="[Customers].[State Province]"
bottomLevel="[Customers].[City]">
    <MemberGrant member="[Customers].[USA].[CA]" access="all"/>
    <MemberGrant member="[Customers].[USA].[CA].[Los Angeles]"
access="none"/>
    </HierarchyGrant>
    <HierarchyGrant hierarchy="[Gender]" access="none"/>
    </CubeGrant>
</SchemaGrant>
</Role>

```

A [<SchemaGrant>](#) defines the default access for objects in a schema. The `access` attribute can be "all" or "none"; this access can be overridden for specific objects. In this case, because `access="none"`, a user would only be able to browse the "Sales" cube, because it is explicitly granted.

A [<CubeGrant>](#) defines the access to a particular cube. As for [<SchemaGrant>](#), the access attribute can be "all" or "none", and can be overridden for specific sub-objects in the cube.

A [<HierarchyGrant>](#) defines access to a hierarchy. The access attribute can be "all", meaning all members are visible; "none", meaning the hierarchy's very existence is hidden from the user; and "custom". With custom access, you can use the `topLevel` attribute to define the top level which is visible (preventing users from seeing too much of the 'big picture', such as viewing revenues rolled up to the `Store Country` level); or use the `bottomLevel` attribute to define the bottom level which is visible (here, preventing users from invading looking at individual customers' details); or control which sets of members the user can see, by defining nested [<MemberGrant>](#) elements.

You can only define a [<MemberGrant>](#) element if its enclosing [<HierarchyGrant>](#) has `access="custom"`. Member grants give (or remove) access to a given member, and all of its children. Here are the rules:

1. **Members inherit access from their parents.** If you deny access to California, you won't be able to see San Francisco.
2. **Grants are order-dependent.** If you grant access to USA, then deny access to Oregon, then you won't be able to see Oregon, or Portland. But if you were to deny access to Oregon, then grant access to USA, you can effectively see everything.
3. **A member is visible if any of its children are visible.** Suppose you deny access to USA, then grant access to California. You will be able to see USA, and California, but none of the other states. The totals against USA will still reflect all states, however.
4. **Member grants don't override the hierarchy grant's top- and bottom-levels.** If you set `topLevel="[Store].[Store State]"`, and grant access to California, you won't be able to see USA.

In the example, the user will have access to California, and all of the cities in California except Los Angeles. They will be able to see USA (because its child, California, is visible), but no other nations, and not All Stores (because it is above the top level, `Store Country`).

Setting a connection's role

A role only has effect when it is associated with a connection. By default, connections have a role which gives them access to every cube in that connection's schema.

Most databases associate roles (or 'groups') with users, and automatically assign them when users log in. However, Mondrian doesn't have the notion of users, so you have to establish the role in a different way. There are two ways of doing this:

1. **In the connect string.** If you specify the `Role` keyword in the connect string, the connection will adopt that role. See [class DriverManager](#) for examples of connect string syntax.
2. **Programmatically.** Once your application has established a connection, call the method [Connection.setRole\(Role\)](#). You can create a Role programmatically (see [class Role](#) for more details), or look one up using the method [Schema.lookupRole\(String\)](#).

XML elements

Element	Description
<Schema>	Collection of Cubes, Virtual cubes, Shared dimensions, and Roles.
<i>Logical elements</i>	
<Cube>	A collection of dimensions and measures, all centered on a fact table.
<VirtualCube>	A cube defined by combining the dimensions and measures of one or more cubes.
<Dimension>	
<DimensionUsage>	Usage of a shared dimension by a cube.
<Hierarchy>	Hierarchy.
<Level>	Level of a hierarchy.
<Property>	Member property. The definition is against a hierarchy or level, but the property will be available to all members.
<Measure>	
<CalculatedMember>	A member whose value is derived using a formula, defined as part of a cube.
<NamedSet>	A set whose value is derived using a formula, defined as part of a cube.
<i>Physical elements</i>	
<Table>	Fact or dimension table.
<View>	Defines a 'table' using a SQL query, which can have different variants for different underlying databases.
<Join>	Defines a 'table' by joining a set of queries.

<InlineTable>	Defines a table using an inline dataset.
<Closure>	Maps a parent-child hierarchy onto a closure table.
<i>Aggregate Tables</i>	
<AggExclude>	Exclude a candidate aggregate table by name or pattern matching.
<AggName>	Declares an aggregate table to be matched by name.
<AggPattern>	Declares a set of aggregate tables by regular expression pattern.
<AggFactCount>	Specifies name of the column in the candidate aggregate table which contains the number of fact table rows.
<AggIgnoreColumn>	Tells Mondrian to ignore a column in an aggregate table.
<AggForeignKey>	Maps foreign key in the fact table to a foreign key column in the candidate aggregate table.
<AggMeasure>	Maps a measure to a column in the candidate aggregate table.
<AggLevel>	Maps a level to a column in the candidate aggregate table.
<i>Access control</i>	
<Role>	An access-control profile.
<SchemaGrant>	A set of rights to a schema.
<CubeGrant>	A set of rights to a cube.
<HierarchyGrant>	A set of rights to a hierarchy and levels within that hierarchy.
<MemberGrant>	A set of rights to a member and its children.
<i>Extensions</i>	
<UserDefinedFunction>	Imports a user-defined function.
<i>Miscellaneous</i>	
<Parameter>	Part of the definition of a Hierarchy; passed to a MemberReader, if present.
<CalculatedMemberProperty>	Property of a calculated member.
<Formula>	Holds the formula text within a <NamedSet> or <CalculatedMember>.
<ColumnDefs>	Holder for <ColumnDef> elements.
<ColumnDef>	Definition of a column in an <InlineTable> dataset.
<Rows>	Holder for <Row> elements.
<Row>	Row in an <InlineTable> dataset.
<Value>	Value of a column in an <InlineTable> dataset.

<[MeasureExpression](#)>

SQL expression used to compute a measure, in lieu of a column.

<[SQL](#)>

The SQL expression for a particular database dialect.

MDX

Copyright (C) 2005-2006 Julian Hyde

MDX is a language for querying multidimensional databases, in the same way that SQL is used to query relational databases. It was originally defined as part of the OLE DB for OLAP specification, and a similar language, mdXML, is part of the XML for Analysis specification.

MDX stands for 'multi-dimensional expressions'. It is the query language implemented by Mondrian. Microsoft proposed MDX as a standard, and its adoption among application writers and other OLAP providers is steadily increasing. Since you can read [the specification online](#) and there are [some great books on MDX available](#), I won't describe the full MDX language.

Mondrian extends MDX with parameters and modified builtin functions.

What is the syntax of MDX?

A basic MDX query looks like this:

```
SELECT {[Measures].[Unit Sales], [Measures].[Store Sales]} ON COLUMNS,  
{[Product].members} ON ROWS  
FROM [Sales]  
WHERE [Time].[1997].[Q2]
```

It looks a little like SQL, but don't be deceived! The structure of an MDX query is quite different from SQL.

Since MDX is a standard language, we don't cover its syntax here. (The Microsoft SQL Server site has an [MDX specification](#); there's also a [good tutorial](#) in Database Journal.) This specification describes the differences between Mondrian's dialect and the standard dialect of MDX.

Mondrian-specific MDX

StrToSet and StrToTuple

The StrToSet() and StrToTuple() functions take an optional parameter not present in the standard MDX versions of these functions, describing the hierarchy the result will belong to:

```
StrToSet(<String Expression>[, <Hierarchy>])  
StrToTuple(<String Expression>[, <Hierarchy>])
```

Parsing

Parsing is case-sensitive.

Parameters

Pseudo-functions `Param()` and `ParamRef()` allow you to create parameterized MDX statements. A parameter is a named variable embedded in an MDX query. Every parameter has a default value, but you can supply a different value when you run the query.

Parameters are declared and used by using a special function, `Parameter`:

```
Parameter(<name>, <type>, <defaultValue>[, <description>])
```

The arguments of `Parameter` are as follows:

- name is a string constant. It must be unique within the query.
- type is either `NUMERIC`, `STRING`, or the name of a hierarchy.
- `defaultValue` is an expression. The expression's type must be consistent with the type parameter; if type was a hierarchy, the expression must be a member of that hierarchy.
- description is an optional string constant.

If you want to use a parameter more than once in a query, use the `ParamRef` function:

```
ParamRef(<name>)
```

The name argument must be the name of a parameter declared elsewhere in the query by calling the `Parameter` function.

The following query shows the top 10 brands in California, but you could change the `Count` parameter to show the top 5, or the `Region` parameter to show sales in Seattle:

```
SELECT {[Measures].[Unit Sales]} on columns,  
  TopCount([Product].[Brand].members,  
    Parameter("Count", NUMERIC, 10, "Number of products to show"),  
    (Parameter("Region", [Store], [Store].[USA].[CA]),  
    [Measures].[Unit Sales])) on rows  
FROM Sales
```

You can list a query's parameters by calling [Query.getParameters\(\)](#), and change a parameter's value by calling [Query.setParameter\(String name, String value\)](#).

Cast operator

The Cast operator converts scalar expressions to other types. The syntax is

```
Cast(<Expression> AS <Type>)
```

where `<Type>` is one of:

- `BOOLEAN`
- `NUMERIC`
- `DECIMAL`
- `STRING`

For example,

```
Cast([Store].CurrentMember.[Store Sqft], INTEGER)
```

returns the value of the [Store Sqft] property as an integer value.

IN **and** NOT IN

IN and NOT IN are Mondrian-specific functions. For example:

```
SELECT {[Measures].[Unit Sales]} ON COLUMNS,  
       FILTER([Product].[Product Family].MEMBERS,  
             [Product].[Product Family].CurrentMember NOT IN  
             {[Product].[All Products].firstChild,  
              [Product].[All Products].lastChild}) ON ROWS  
FROM [Sales]
```

MATCHES **and** NOT MATCHES

MATCHES and NOT MATCHES are Mondrian-specific functions which compare a string with a [Java regular expression](#). For example, the following query finds all employees whose name starts with 'sam' (case-insensitive):

```
SELECT {[Measures].[Org Salary]} ON COLUMNS,  
       Filter([Employees].MEMBERS,  
             [Employees].CurrentMember.Name MATCHES '(?i)sam.*') ON ROWS  
FROM [HR]
```

Comments

MDX statements can contain comments. There are 3 syntactic forms for comments:

```
// End-of-line comment
```

```
-- End-of-line comment
```

```
/* Multi-line  
comment */
```

Comments can be nested, for example

```
/* Multi-line  
comment /* Comment within a comment */  
*/
```

Format Strings

Every member has a FORMAT_STRING property, which affects how its raw value is rendered into text in the user interface. For example, the query

```

WITH MEMBER [Measures].[Profit] AS '([Measures].[Store Sales] -
[Measures].[Store Cost])',
FORMAT_STRING = "$#,###.00"
SELECT {[Measures].[Store Sales], [Measures].[Profit]} ON COLUMNS,
{[Product].CurrentMember.Children} ON ROWS
FROM [Sales]

```

yields cells formatted in dollar and cent amounts.

Members defined in a schema file can also have format strings. Measures use the `formatString` attribute:

```

<Measure name="Store Sales" column="store_sales" aggregator="sum"
formatString="# ,###.00"/>

```

and calculated members use the `<CalculatedMemberProperty>` sub-element:

```

<CalculatedMember name="Profit" dimension="Measures"
formula="[Measures].[Store Sales] - [Measures].[Store Cost]">
  <CalculatedMemberProperty name="FORMAT_STRING" value="$#,##0.00"/>
</CalculatedMember>

```

Format strings use Visual Basic formatting syntax; see [class mondrian.olap.Format](#) for more details.

A measure's format string is usually a fixed string, but is really an expression, which is evaluated in the same context as the cell. You can therefore change the formatting of a cell depending upon the cell's value.

The format string can even contain 'style' attributes which are interpreted specially by JPivot. If present, JPivot will render cells in color.

The following example combines a dynamic formula with style attributes. The result is that cells are displayed with green background if they are less than \$100,000, or a red background if they are greater than \$100,000:

```

WITH MEMBER [Measures].[Profit] AS
  '([Measures].[Store Sales] - [Measures].[Store Cost])',
  FORMAT_STRING = Iif([Measures].[Profit] < 100000, '|#|style=green',
'|#|style=red')
SELECT {[Measures].[Store Sales], [Measures].[Profit]} ON COLUMNS,
{[Product].CurrentMember.Children} ON ROWS
FROM [Sales]

```

Configuration Guide

Copyright (C) 2006-2007 Julian Hyde and others

Properties that control execution

Mondrian has a properties file to allow you to configure how it executes. The `mondrian.properties` file is loaded when the executing Mondrian JAR detects it needs properties, but can also be done explicitly in your code. It looks in several places, in the following order:

1. In the directory where you started your JVM (Current working directory for JVM process, `java.exe` on Win32, `java` on *nix).
2. If there isn't `mondrian.properties` under current working directory of JVM process, Class `MondrianProperties`'s classloader will try to locate `mondrian.properties` in all of its classpaths. So you may put `mondrian.properties` under `/WEB-INF/classes` when you pack Mondrian into a Java web application. The demonstration web applications have this configuration.

These properties are stored as system properties, so they can be set during JVM startup via `-D<property>=<value>`.

Property list

The following properties in the `mondrian.properties` file effect the operations of Mondrian.

Not all of the properties in this table are of interest to the end-user. For example, those in the 'Testing' are only applicable if are running Mondrian's suite of regression tests.

Property	Type	Default value	Description
Miscellaneous			
<code>mondrian.foodmart.jdbcURL</code>	string	"jdbc:odbc:Mondrian FoodMart"	Property containing the JDBC URL of the FoodMart database. The default value is to connect to an ODBC data source called "MondrianFoodMart". Maximum number of simultaneous queries the system will allow.
<code>mondrian.query.limit</code>	int	40	Oracle fails if you try to run more than the 'processes' parameter in <code>init.ora</code> , typically 150. The throughput of Oracle and other databases will probably reduce long before you get to their limit.
<code>mondrian.jdbcDrivers</code>	string	See Description	A list of JDBC drivers to load automatically. Must be a comma-separated list of class names, and the classes must be on the class path.

Default drivers are:

mondrian.jdbcDrivers,
sun.jdbc.odbc .JdbcOdbcDriver,
org.hsqldb.jdbcDriver,
oracle.jdbc.Oracle Driver,
com.mysql.jdbc .Driver

mondrian.result .limit	int	0	If set to a value greater than zero, limits the maximum size of a result set.
mondrian.rolap. CachePool.costLimit	int	10,000	Obsolete.
mondrian.rolap. evaluate. MaxEvalDepth	int	10	Maximum number of passes allowable while evaluating an MDX expression. If evaluation exceeds this depth (for example, while evaluating a very complex calculated member), Mondrian will throw an error.
mondrian.rolap. LargeDimension Threshold	int	100	Determines when a dimension is considered "large". If a dimension has more than this number of members, Mondrian uses a smart member reader . The values of the <code>mondrian.rolap.SparseSegment ValueThreshold</code> (<i>countThreshold</i>) and <code>mondrian.rolap.SparseSegment DensityThreshold</code> (<i>densityThreshold</i>) properties determine whether to choose a sparse or dense representation when storing collections of cell values in memory. When storing collections of cell values in memory, Mondrian has to choose between a sparse and a dense representation, based upon the <i>possible</i> and <i>actual</i> number of values. The <i>density</i> is defined by the formula $density = actual / possible$ Mondrian uses a sparse representation if $possible - (countThreshold * actual) > densityThreshold$ For example, at the default values (<i>countThreshold</i> = 1000 and <i>densityThreshold</i> = 0.5), Mondrian use a dense representation for <ul style="list-style-type: none">• (1000 possible, 0 actual), or• (2000 possible, 500 actual), or
mondrian.rolap. SparseSegment ValueThreshold	int	1,000	

- (3000 possible, 1000 actual).

Any fewer actual values, or any more possible values, and Mondrian will use a sparse representation.

[mondrian.rolap.SparseSegmentDensityThreshold](#)

double 0.5

See `mondrian.rolap.SparseSegmentValueThreshold`.

Whether to notify the Mondrian system when a property value changes.

This allows objects dependent on Mondrian properties to react (that is, reload), when a given property changes via, say,

[mondrian.olap.triggers.enable](#)

boolean true

```
MondrianProperties .instance()
    .populate(null);
```

or

```
MondrianProperties .instance()
    .QueryLimit.set(50);
```

[mondrian.olap.case.sensitive](#)

boolean false

Controls whether the MDX parser resolves uses case-sensitive matching when looking up identifiers.

[mondrian.rolap.localePropFile](#)

string null

Name of locale property file.

Used for the [LocalizingDynamicSchemaProcessor](#); see [Internationalization](#) for more details.

`mondrian.rolap.queryTimeout`

int 0

If set to a value greater than zero, limits the number of seconds a query executes before it is aborted.

[mondrian.rolap.nonempty](#)

boolean false

If true, each query axis implicit has the NON EMPTY option set (and in fact there is no way to display empty cells).

`mondrian.rolap.ignoreInvalidMembers`

boolean false

If set to true, during schema load, invalid members are ignored and will be treated as a null member if they are later referenced in a query.

Tracing

The amount of tracing displayed.

[mondrian.trace.level](#)

int -

If trace level is above 0, SQL tracing will be enabled and logged as per the `out.file` below. This is separate from Log4j logging.

[mondrian.debug.out.file](#)

string System.out

The name of the file to which tracing is to be written.

[mondrian.rolap.
RolapResult.
printCacheables](#)

boolean false

Obsolete.

Testing

[mondrian.test. Name](#)

string null

Property which determines which tests are run. This is a [Java regular expression](#). If this property is specified, only tests whose names match the pattern in its entirety will be run.

[mondrian.test. Class](#)

string -

Property which determines which test class to run. This is the name of the class which either implements interface [junit .framework.Test](#) or has a method `public static junit.framework.Test suite()`.

[mondrian.test
.connectString](#)

string -

Property containing the connect string which regression tests should use to connect to the database.

See the [connect string specification](#) for more details.

[mondrian.test
.QueryFilePattern](#)

string -

(not documented)

[mondrian.test
.QueryFileDirectory](#)

string -

(not documented)

[mondrian.test .Iterations](#)

int 1

(not documented)

[mondrian.test .VUsers](#)

int 1

(not documented)

[mondrian.test .TimeLimit](#)

int 0

The time limit for the test run in seconds. If the test is running after that time, it is terminated.

[mondrian.test .Warmup](#)

boolean false

Whether this is a "warmup test".

[mondrian. catalogURL](#)

string -

The URL of the catalog to be used by [CmdRunner](#) and XML/A Test.

Whether to test operators' dependencies, and how much time to spend doing it.

[mondrian.test
.ExpDependencies](#)

int 0

If this property is positive, Mondrian's test framework allocates an expression evaluator which evaluates each expression several times, and makes sure that the results of the expression are independent of dimensions which the expression claims to be independent of.

Seed for random number generator used by some of the tests.

[mondrian.test .random.seed](#)

int 1234

Any value besides 0 or -1 gives deterministic behavior. The default value is 1234: most users should use this. Setting

the seed to a different value can increase coverage, and therefore may uncover new bugs.

If you set the value to 0, the system will generate its own pseudo-random seed.

If you set the value to -1, Mondrian uses the next seed from an internal random-number generator. This is a little more deterministic than setting the value to 0.

```
public final IntegerProperty TestSeed =
new IntegerProperty(
this, "", 1234);
```

mondrian.test.jdbcURL	string	-	Property containing the JDBC URL of a test database. It does not default.
mondrian.test.jdbcUser	string	-	Property containing the JDBC user of a test database. The default value is null, to cope with DBMSs that don't need this.
mondrian.test.jdbcPassword	string	-	Property containing the JDBC password of a test database. The default value is null, to cope with DBMSs that don't need this.

Aggregate tables

Whether to use aggregate tables.

If true, then Mondrian uses aggregate tables. This property is queried prior to each aggregate query so that changing the value of this property dynamically (not just at startup) is meaningful.

mondrian.rolap.aggregates.Use	boolean	false	
---	---------	-------	--

Aggregates can be read from the database using the `mondrian.rolap.aggregates.Read` property but will not be used unless this property is set to true. Whether to read aggregate tables.

mondrian.rolap.aggregates.Read	boolean	false	
--	---------	-------	--

If set to true, then Mondrian scans the database for aggregate tables. Unless `mondrian.rolap.aggregates.Use` is set to true, the aggregates found will not be used.

Whether to choose an aggregate tables based volume or row count.

mondrian.rolap.aggregates.ChooseByVolume	boolean	false	
--	---------	-------	--

If true, Mondrian uses the aggregate table with the smallest volume (number of rows multiplied by number of columns); if false,

			Mondrian uses the aggregate table with the fewest rows.
			Name of the file which defines the rules for recognizing an aggregate table.
mondrian.rolap.aggregates.rules	string	See Description	<p>Can be either a resource in the Mondrian jar or a URL. See aggregate table rules for details.</p> <p>Normally, this property is not set by a user.</p>
			Default: "/DefaultRules.xml" (which is in the <code>mondrian.rolap.aggmatcher</code> package in <code>mondrian.jar</code>)
mondrian.rolap.aggregates.rule.tag	string	default	<p>The AggRule element's tag value.</p> <p>Normally, this property is not set by a user.</p>
			Whether to print the SQL code generated for aggregate tables.
mondrian.rolap.aggregates.generateSql	boolean	false	<p>If set, then as each aggregate request is processed, both the lost and collapsed dimension create and insert sql code is printed. This is for use in the CmdRunner allowing one to create aggregate table generation sql.</p>

Caching

			Whether to clear a RolapStar's data cache after each query.
mondrian.rolap.star.disableCaching	boolean	false	<p>If true, RolapStar does not cache aggregate data from one query to the next: the cache is cleared after each query.</p> <p>Controls whether to use a cache for the results of frequently evaluated expressions.</p>
			With the cache disabled, an expression like:
mondrian.expCache.enable	boolean	true	<pre>Rank([Product]. CurrentMember, Order([Product] .MEMBERS, [Measures].[Unit Sales]))</pre> <p>would perform many redundant sorts.</p>
mondrian.rolap.RolapResult.flushAfterEachQuery	boolean	false	Obsolete.

SQL generation

mondrian.native.crossjoin.enable	boolean	true	If enabled, some NON EMPTY <code>CrossJoin</code> MDX statements will be computed in the database and not within Mondrian/Java
mondrian.native.topcount.enable	boolean	false	If enabled, some <code>TopCount</code> MDX statements will be computed in the database and not within Mondrian/Java
mondrian.native.filter.enable	boolean	false	If enabled, some <code>Filter()</code> MDX statements will be computed in the database and not within Mondrian/Java
mondrian.native.nonempty.enable	boolean	true	If enabled, some NON EMPTY MDX set operations like <code>member.children</code> , <code>level.members</code> and <code>member.descendants</code> will be computed in the database and not within Mondrian/Java
mondrian.rolap.generate.formatted.sql	boolean	false	Whether to pretty-print SQL generated statements.
mondrian.rolap.maxConstraints	int	1,000	If true, Mondrian generates SQL strings are generated in the log or output in pretty-print mode, formatted for ease of reading. Max number of constraints in a single `IN` SQL clause.

This value may be variant among database products and their runtime settings. Oracle, for example, gives the error "ORA-01795: maximum number of expressions in a list is 1000".

Recommended values:

- Oracle: 1,000
- DB2: 2,500
- Other: 10,000

XML/A

mondrian.xmla.drillthrough.TotalCount.enable	boolean	true	If enabled, first row in the result of an XML/A drill-through request will be filled with the total count of rows in underlying database.
mondrian.xmla.drillthrough.MaxRows	int	1,000	Limit on the number of rows returned by XML/A drill through request.

Connect strings

Connect string syntax

Mondrian connect strings are a connection of property/value pairs, of the form 'property=value;property=value;...'.

Values can be enclosed in single-quotes, which allows them to contain spaces and punctuation. See the the [OLE DB connect string syntax specification](#).

The supported properties are described below.

Connect string properties

Name	Required?	Description
Provider	Yes	Must have the value "Mondrian".
Jdbc	Exactly one	The URL of the JDBC database where the data is stored. You must specify either <code>DataSource</code> or <code>Jdbc</code> .
DataSource		The name of a data source class. The class must implement the javax.sql.DataSource interface. You must specify either <code>DataSource</code> or <code>Jdbc</code> .
JdbcDrivers	Yes	Comma-separated list of JDBC driver classes, for example, <code>JdbcDrivers=sun.jdbc.odbc.JdbcOdbcDriver,oracle.jdbc.OracleDriver</code>
JdbcUser	No	The name of the user to log on to the JDBC database. (If your JDBC driver allows you to specify the user name in the JDBC URL, you don't need to set this property.)
JdbcPassword	No	The name of the password to log on to the JDBC database. (If your JDBC driver allows you to specify the password in the JDBC URL, you don't need to set this property.)
Catalog	Exactly one	The URL of the catalog, an XML file which describes the schema: cubes, hierarchies, and so forth. For example, <code>Catalog=file:demo/FoodMart.xml</code>
CatalogContent		Catalogs are described in the Schema Guide . See also <code>CatalogContent</code> . An XML string representing the schema: cubes, hierarchies, and so forth. For example, <code>CatalogContent=<Schema name="MySchema"><Cube name="Cube1"> ... </Schema></code> Catalogs are described in the Schema Guide . See also <code>Catalog</code> .

CatalogName	No	Not used. If, in future, Mondrian supports multiple catalogs, this property will specify which catalog to use. See also <code>Catalog</code> .
PoolNeeded	No	<p>Tells Mondrian whether to add a layer of connection pooling.</p> <p>If the value "true" is specified, or no value is specified, Mondrian assumes that:</p> <ul style="list-style-type: none"> connections created via the <code>Jdbc</code> property are not pooled, and therefore need to be pooled; connections created via the <code>DataSource</code> are already pooled. <p>If the value "false" is specified, Mondrian does not apply connection-pooling to any connection.</p>
Role	No	The name of the role to adopt for access-control purposes. If not specified, the connection uses a role which has access to every object in the schema.
DynamicSchemaProcessor	No	<p>The name of a class which is called at runtime in order to modify the schema content. The class must implement the mondrian.rolap.DynamicSchemaProcessor interface. For example,</p> <pre>DynamicSchemaProcessor = mondrian.i18n.LocalizingDynamicSchemaProcessor</pre> <p>uses the builtin schema processor class mondrian.i18n.LocalizingDynamicSchemaProcessor to replace variables in the schema file, according to resource files and the current locale (see the <code>Locale</code> property).</p>
Locale	No	<p>The requested Locale for the current session. The locale determines the formatting of numbers and date/time values, and Mondrian's error messages.</p> <p>Example values are "en" (English), "en_US" (United States English), "hu" (Hungarian). If Locale is not specified, then the name of system's default will be used, as per java.util.Locale#getDefault().</p>

Connect string properties are also documented in the [RolapConnectionProperties](#) class.

Optimizing Mondrian Performance

Copyright (C) 2005-2006 Julian Hyde, Sherman Wood and others

Introduction

As with any data warehouse project, dealing with volumes is always the make or break issue. Mondrian has its own issues, based on its architecture and goals of being cross platform. Here are some experiences and comments.

From the Mondrian developer's mailing list in February, 2005 - an example of unoptimized performance:

When Mondrian initializes and starts to process the first queries, it makes SQL calls to get member lists and determine cardinality, and then to load segments into the cache. When Mondrian is closed and restarted, it has to do that work again. This can be a significant chunk of time depending on the cube size. For example in one test an 8GB cube (55M row fact table) took 15 minutes (mostly doing a group by) before it returned results from its first query, and absent any caching on the database server would take another 15 minutes if you closed it and reopened the application. Now, this cube was just one month of data; imagine the time if there was 5 years worth.

Since this time, Mondrian has been extended to use aggregate tables and materialized views, which have a lot of performance benefits that address the above issue.

From Julian:

I'm surprised that people can run 10m+ row fact tables on Mondrian at all, without using aggregate tables or materialized views.

From Sherman:

Our largest site has a cube with currently ~6M facts on a single low end Linux box running our application with Mondrian and Postgres (not an ideal configuration), without aggregate tables, and gets sub second response times for the user interface (JPivot). This was achieved by tuning the database to support the queries being executed, modifying the OS configuration to best support Postgres execution (thanks Josh!) and adding as much RAM as possible.

A generalized tuning process for Mondrian

The process for addressing performance of Mondrian is a combination of design, hardware, database and other configuration tuning. For really large cubes, the performance issues are driven more by the hardware, operating system and database tuning than anything Mondrian can do.

- Have a reasonable physical design for requirements, such as a data warehouse and specific data marts
- Architect the application effectively
 - Separate the environment where Mondrian is executing from the DBMS

- If possible: separate UI processing from the environment where Mondrian is caching
- Have adequate hardware for the DBMS
- Tune the operating system for the DBMS
- Add materialized views or aggregate tables to support specific MDX queries (see Aggregate Tables and AggGen below)
- Tune the DBMS for the specific SQL queries being executed: that is, indexes on both the dimensions and fact table
- Tune the Mondrian cache: the larger the better

Recommendations for database tuning

As part of database tuning process, enable SQL tracing and tail the log file. Run some representative MDX queries and watch which SQL statements take a long time. Tune the database to fix those statements and rerun.

- Indexes on primary and foreign keys
- Consider enabling foreign keys
- Ensure that columns are marked NOT NULL where possible
- If a table has a compound primary key, experiment with indexing subsets of the columns with different leading edges. For example, for columns (a, b, c) create a unique index on (a, b, c) and non-unique indexes on (b, c) and (c, a). Oracle can use such indexes to speed up counts.
- On Oracle, consider using bitmap indexes for low-cardinality columns. (Julian implemented the Oracle's bitmap index feature, and he's rather proud of them!)
- On Oracle, Postgres and other DBMSs, analyze tables, otherwise the cost-based optimizers will not be used

Mondrian currently uses 'count(distinct ...)' queries to determine the cardinality of dimensions and levels as it starts, and for your measures that are counts, that is, `aggregator="count"`. Indexes might speed up those queries -- although performance is likely to vary between databases, because optimizing count-distinct queries is a tricky problem.

Aggregate Tables, Materialized Views and Mondrian

The best way to increase the performance of Mondrian is to build a set of aggregate (summary) tables that coexist with the base fact table. These aggregate tables contain pre-aggregated measures build from the fact table.

Some databases, particularly Oracle, can automatically create these aggregations through materialized views, which are tables created and synchronized from views. Otherwise, you will have to maintain the aggregation tables through your data warehouse load processes, usually by clearing them and rerunning aggregating INSERTs.

Aggregate tables are introduced in the [Schema Guide](#).

Choosing aggregate tables

It isn't easy to choose the right aggregate tables. For one thing, there are so many to choose from: even a modest cube with six dimensions each with three levels has $6^4 = 1296$ possible

aggregate tables! And aggregate tables interfere with each other. If you add a new aggregate table, Mondrian may use an existing aggregate table less frequently.

Missing aggregate tables may not even be the problem. Choosing aggregate tables is part of a wider performance tuning process, where finding the problem is more than half of the battle. The real cause may be a missing index on your fact table, your cache isn't large enough, or (if you're running Oracle) the fact that you forgot to compute statistics. (See [recommendations](#), above.)

Performance tuning is an iterative process. The steps are something like this:

1. Choose a few queries which are typical for those the end-users will be executing.
2. Run your set of sample queries, and note how long they take. Now the cache has been primed, run the queries again: has performance improved?
3. Is the performance good enough? If it is, stop tuning now! If your data set isn't very large, you probably don't need any aggregate tables.
4. Decide which aggregate tables to create. If you turn on SQL tracing, looking at the GROUP BY clauses of the long-running SQL statements will be a big clue here.
5. Register the aggregate tables in your catalog, create the tables in the database, populate the tables, and add indexes.
6. Restart Mondrian, to flush the cache and re-read the schema, then go to step 2 to see if things have improved.

AggGen

AggGen is a tool that generates SQL to support the creation and maintenance of aggregate tables, and would give a template for the creation of materialized views for databases that support those. Given an MDX query, the generated create/insert SQL is optimal for the given query. The generated SQL covers both the "lost" and "collapsed" dimensions. For usage, see the documentation for [CmdRunner](#).

Aggregate Tables

Copyright (C) 2005-2006 Julian Hyde, Richard Emberson and others

Introduction

Unlike many OLAP servers, Mondrian does not store data on disk: it just works on the data in the RDBMS, and once it has read a piece of data once, it stores that data in its cache. This greatly simplifies the process of installing Mondrian, but it puts limits on Mondrian's performance when Mondrian is applied to a huge dataset.

Consider what happens when the CEO runs her Sales Report first thing on a Monday morning. This report contains a single number: the total sales of all products, in all regions, this year. In order to get this number, Mondrian generates a query something like this:

```
SELECT sum(store_sales)
FROM sales_fact,
     time
WHERE sales_fact.time_id = time.time_id
AND time.year = 2005
```

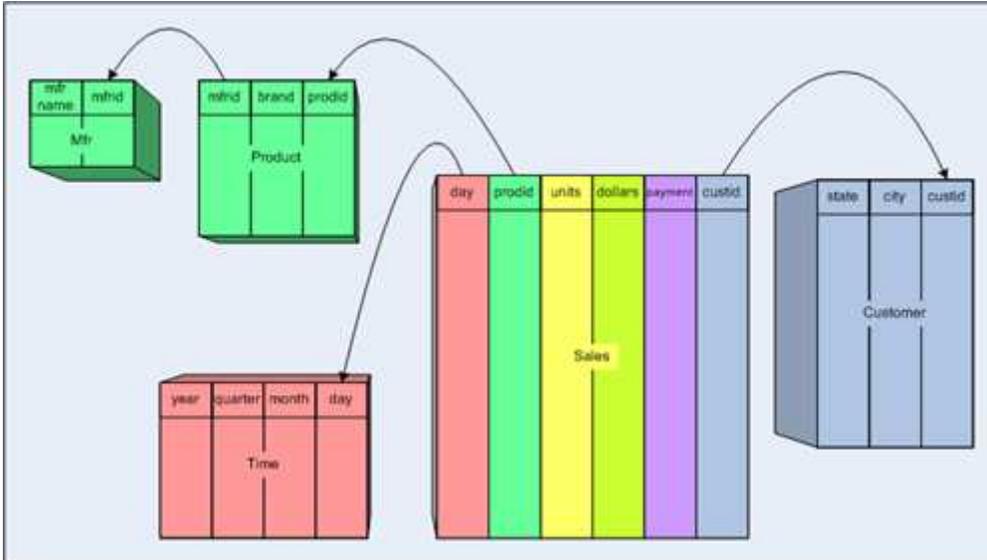
and sends it to the DBMS. The DBMS takes several minutes to execute it: which is understandable because the DBMS has to read all of this year's records in the fact table (a few million sales, say) and aggregate them into a single total. Clearly, what is needed in this case, and in others like it, is a pre-computed summary of the data: an aggregate table.

An *aggregate table* coexists with the base fact table, and contains pre-aggregated measures build from the fact table. It is registered in Mondrian's schema, so that Mondrian can choose to use whether to use the aggregate table rather than the fact table, if it is applicable for a particular query.

Designing aggregate tables is a fine art. There is extensive research, both empirical and theoretical, available on the web concerning different ways to structure aggregate tables and we will not attempt to duplicate any of it here.

What are aggregate tables?

To explain what aggregate tables are, let's consider a simple star schema.



The star schema has a single fact table *Sales*, two measure columns (*units* and *dollars*) and four dimension tables (*Product*, *Mfr*, *Customer*, *Time*, and *Customer*).

On top of this star schema, we create the following multidimensional model:

- Cube [*Sales*] has two measures [*Unit sales*] and [*Dollar sales*]
- Dimension [*Product*] has levels [*All Products*], [*Manufacturer*], [*Brand*], [*Prodid*]
- Dimension [*Time*] has levels [*All Time*], [*Year*], [*Quarter*], [*Month*], [*Day*]
- Dimension [*Customer*] has levels [*All Customers*], [*State*], [*City*], [*Custid*]
- Dimension [*Payment Method*] has levels [*All Payment Methods*], [*Payment Method*]

Note that the [*Product*] dimension is a 'snowflake dimension' (that is, it is spread across two tables *Product* and *Mfr*) and the [*Payment Method*] dimension is contained within the *payment* column in the fact table.

A simple aggregate table

Now let's create an aggregate table, Agg_1:

year	quarter	mfrid	brand	prodid	sum units	min units	max units	sum dollars	row count

See how the original star schema columns have been combined into the table:

- The Time dimension has been "collapsed" into the aggregate table, omitting the month and day columns.
- The two tables of the Product dimension has been "collapsed" into the aggregate table.
- The Customer dimension has been "lost".
- For each measure column in the fact table (units, dollars), there are one or more measure columns in the aggregate table (sum units, min units, max units, sum dollars).
- There is also a measure column, row count, representing the "count" measure.

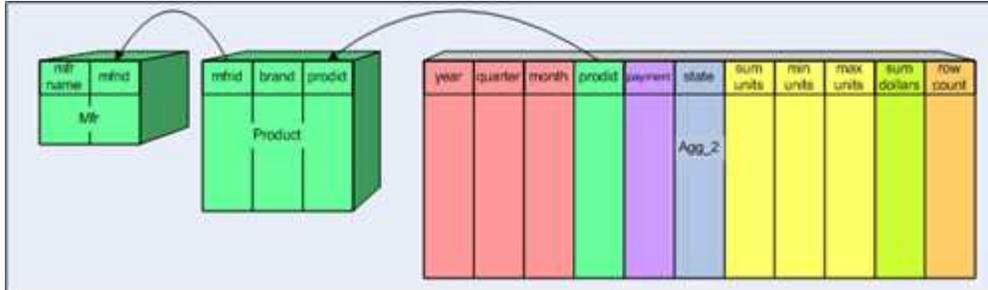
Agg_1 would be declared like this:

```
<Cube name="Sales">
  <Table name="sales">
    <AggName name="agg_1">
      <AggFactCount column="row count"/>
      <AggMeasure name="[Measures].[Unit Sales]" column="sum units"/>
      <AggMeasure name="[Measures].[Min Units]" column="min units"/>
      <AggMeasure name="[Measures].[Max Units]" column="max units"/>
      <AggMeasure name="[Measures].[Dollar Sales]" column="sum
dollars"/>
      <AggLevel name="[Time].[Year]" column="year"/>
      <AggLevel name="[Time].[Quarter]" column="quarter"/>
      <AggLevel name="[Product].[Mfrid]" column="mfrid"/>
      <AggLevel name="[Product].[Brand]" column="brand"/>
      <AggLevel name="[Product].[Prodid]" column="prodid"/>
    </AggName>
  </Table>

  <!-- Rest of the cube definition -->
</Cube>
```

Another aggregate table

Another aggregate table, Agg_2:



and the corresponding XML:

```
<Cube name="Sales">
  <Table name="sales">
    <AggName name="agg_1" ... />
    <AggName name="agg_2">
      <AggFactCount column="row count"/>
      <AggForeignKey factColumn="prodid" aggColumn="prodid"/>
      <AggMeasure name="[Measures].[Unit Sales]" column="sum units"/>
      <AggMeasure name="[Measures].[Min Units]" column="min units"/>
      <AggMeasure name="[Measures].[Max Units]" column="max units"/>
      <AggMeasure name="[Measures].[Dollar Sales]" column="sum
dollars"/>
      <AggLevel name="[Time].[Year]" column="year"/>
      <AggLevel name="[Time].[Quarter]" column="quarter"/>
      <AggLevel name="[Time].[Month]" column="month"/>
      <AggLevel name="[Payment Method].[Payment Method]"
column="payment"/>
      <AggLevel name="[Customer].[State]" column="state"/>
    </AggName>
  </Table>

  <Dimension name="Product">
    <Hierarchy hasAll="true" primaryKey="prodid"
primaryKeyTable="Product">
      <Join leftKey="mfrid" rightKey="mfrid">
        <Table name="Product"/>
        <Table name="Mfr"/>
      </Join>
      <Level name="Manufacturer" table="Mfr" column="mfrid"/>
      <Level name="Brand" table="Product" column="brand"/>
      <Level name="Name" table="Product" column="prodid"/>
    </Hierarchy>
  </Dimension>

  <!-- Rest of the cube definition -->
</Cube>
```

Several dimensions have been collapsed: [Time] at the [Quarter] level; [Customer] at the [State] level; and [Payment Method] at the [Payment Method] level. But the [Product] dimension has been retained in its original snowflake form.

The `<AggForeignKey>` element is used to declare that the column `prodid` links to the dimension table, but all other columns remain in the `Product` and `Mfr` dimension tables.

Defining aggregate tables

A fact table can have zero or more aggregate tables. Every aggregate table is associated with just one fact table. It aggregates the fact table measures over one or more of the dimensions. As an example, if a particular column in the fact table represents the number of sales of some product on a given day by a given store, then an aggregate table might be created that sums the information so that applies at a month level rather than by day. Such an aggregate might reasonably be 1/30th the size of the fact table (assuming comparable sales for every day of a month). Now, if one were to execute a MDX query that needed sales information at a month (or quarter or year) level, running the query against the aggregate table is faster but yields the same answer as if it were run against the base fact table.

Further, one might create an aggregate that not only aggregates at the month level but also, rather than at the individual store level, aggregates at the state level. If there were, say, 20 stores per state, then this aggregate table would be 1/600th the size of the original fact table. MDX queries interested only at the month or above and state or above levels would use this table.

When a MDX query runs, what aggregate should be used? This comes down to what measures are needed and with which dimension levels. The base fact table always has the correct measures and dimension levels. But, it might also be true that there is one or more aggregate tables that also have the measures and levels. Of these, the aggregate table with the lowest cost to read, the smallest number of rows, should be the table used to fulfill the query.

Mondrian supports two aggregation techniques which are called "lost" dimension and "collapsed" dimension. For the creation of any given aggregate table these can be applied independently to any number of different dimensions.

A "lost" dimension is one which is completely missing from the aggregate table. The measures that appear in the table have been aggregated across all values of the lost dimension. As an example, in a fact table with dimensions of time, location, and product and measure sales, for an aggregate table that did not have the location dimension that dimension would be "lost". Here, the sales measure would be the aggregation over all locations. An aggregate table where all of the dimensions are lost is possible - it would have a single row with the measure aggregated over everything - sales for all time, all locations and all products.

```
fact table
  time_id
  product_id
  location_id
  measure
```

```
lost (time_id) dimension table
  product_id
  location_id
```

```
measure (aggregated over time)
fact_count
```

```
fully lost dimension table
measure (aggregated over everything)
fact_count
```

Note the "fact_count" column in the aggregate table. This additional column is a general feature of aggregate tables. It is a count of how many fact table columns were aggregated into the one aggregate table row. As an example, if for a particular choice of product_id and location_id, the time_id occurred 5 times in the fact table, then in the aggregate table the fact_count column would contain 5 for that product_id/location_id pair (a given product was sold at a given location at 5 different times).

The second supported aggregation technique provides a finer level of control, the "collapsed" dimension technique. Recall that the dimension key in the fact table refers (more or less) to the lowest level in the dimension hierarchy. For a collapsed dimension, the dimension key in the aggregate table is replaced with a set of dimension levels; the dimension key column is replaced with a set of columns; a fully denormalized summary table for that dimension. As an example, if the time dimension with base fact table foreign key time_id had the levels: day, month, quarter and year, and in an aggregate it was collapsed to the month level, then the aggregate table would not have a time_id column but rather columns for month, quarter and year. The SQL generated for a MDX query for which this aggregate table can be used, would no longer refer to the time dimension's table but rather all time related information would be gotten from the aggregate table.

```
time dimension table
time_id
day
month
quarter
year
```

```
fact table
time_id
measure
```

```
collapsed dimension table
month
quarter
year
measure (aggregated to month level)
fact_count
```

In the literature, there are other ways of creating aggregate tables but they are not supported by Mondrian at this time.

Building aggregate tables

Aggregate tables must be built. Generally, they not real-time; they are rebuilt, for example, every night for use the following day by the analysts. Considering the lost and collapsed dimension technique for aggregate table definition, one can estimate that for a dimension with N levels, there are N+1 possible aggregate tables (N collapsed and 1 lost). Also, dimensions (with different

dimension tables) can be aggregated independently. For the FoodMart Sales cube there are 1400 different possible aggregate tables.

Clearly, one does not want to create all possible aggregate tables. Which ones to create depends upon two considerations. The first consideration is application dependent: the nature of the MDX queries that will be executed. If many of the queries deal with per month and per state questions, then an aggregate at those levels might be created. The second consideration is application independent: per dimension aggregating from the lowest level to the next lowest generally gives greater bang for the buck than aggregating from the N to the N+1 (N>1) level. This is because 1) a first level aggregation can be used for all queries at that level and above and 2) dimension fanout tends to increase for the lower levels. Of course, your mileage may vary.

In a sense, picking which aggregate tables to build is analogous to picking which indexes to build on a table; it is application dependent and experience helps.

The hardest part about the actually creation and population of aggregate tables is figuring out how to create the first couple; what the SQL looks like. After that they are pretty much all the same.

Four examples will be given. They all concern building aggregate tables for the sales_fact_1997 fact table. As a reminder, the sales_fact_1997 fact table looks like:

```
sales_fact_1997
  product_id
  time_id
  customer_id
  promotion_id
  store_id
  store_sales
  store_cost
  unit_sales
```

The first example is a lost time dimension aggregate table, the time_id foreign key is missing.

```
CREATE TABLE agg_l_05_sales_fact_1997 (
  product_id INTEGER NOT NULL,
  customer_id INTEGER NOT NULL,
  promotion_id INTEGER NOT NULL,
  store_id INTEGER NOT NULL,
  store_sales DECIMAL(10,4) NOT NULL,
  store_cost DECIMAL(10,4) NOT NULL,
  unit_sales DECIMAL(10,4) NOT NULL,
  fact_count INTEGER NOT NULL);

CREATE INDEX i_sls_97_cust_id ON agg_l_05_sales_fact_1997
(customer_id);
CREATE INDEX i_sls_97_prod_id ON agg_l_05_sales_fact_1997 (product_id);
CREATE INDEX i_sls_97_promo_id ON agg_l_05_sales_fact_1997
(promotion_id);
CREATE INDEX i_sls_97_store_id ON agg_l_05_sales_fact_1997 (store_id);

INSERT INTO agg_l_05_sales_fact_1997 (
  product_id,
  customer_id,
```

```

        promotion_id,
        store_id,
        store_sales,
        store_cost,
        unit_sales,
        fact_count)
SELECT
    product_id,
    customer_id,
    promotion_id,
    store_id,
    SUM(store_sales) AS store_sales,
    SUM(store_cost) AS store_cost,
    SUM(unit_sales) AS unit_sales,
    COUNT(*) AS fact_count
FROM
    sales_fact_1997
GROUP BY
    product_id,
    customer_id,
    promotion_id,
    store_id;

```

A couple of things to note here.

The above is in MySQL's dialect of SQL, and may not work for your database - but I hope the general idea is clear. The aggregate table "looks like" the base fact table except the time_id column is missing and there is a new fact_count column. The insert statement populates the aggregate table from the base fact table summing the measure columns and counting to populate the fact_count column. This done while grouping by the remaining foreign keys to the remaining dimension tables.

Next, some databases recognize star joins - Oracle for instance. For such database one should not create indexes, not on the fact table and not on the aggregate tables. On the other hand, databases that do not recognize star joins will require indexes on both the fact table and the aggregate tables.

For our purposes here, the exact name of the aggregate table is not important; the "agg_I_05_" preceding the base fact table's name sales_fact_1997. First, the aggregate table name must be different from the base fact table name. Next, the aggregate table name ought to be related to the base fact table name both for human eyeballing of what aggregate is associated with which fact table, but also, as described below, Mondrian employs mechanism to automatically recognize which tables are aggregates of others.

The following example is a collapsed dimension aggregate table where the time dimension has been rolled up to the month level.

```

CREATE TABLE agg_c_14_sales_fact_1997 (
    product_id INTEGER NOT NULL,
    customer_id INTEGER NOT NULL,
    promotion_id INTEGER NOT NULL,
    store_id INTEGER NOT NULL,
    month_of_year SMALLINT(6) NOT NULL,
    quarter VARCHAR(30) NOT NULL,

```

```

    the_year SMALLINT(6) NOT NULL,
    store_sales DECIMAL(10,4) NOT NULL,
    store_cost DECIMAL(10,4) NOT NULL,
    unit_sales DECIMAL(10,4) NOT NULL,
    fact_count INTEGER NOT NULL);

CREATE INDEX i_sls_97_cust_id ON agg_c_14_sales_fact_1997
(customer_id);
CREATE INDEX i_sls_97_prod_id ON agg_c_14_sales_fact_1997 (product_id);
CREATE INDEX i_sls_97_promo_id ON agg_c_14_sales_fact_1997
(promotion_id);
CREATE INDEX i_sls_97_store_id ON agg_c_14_sales_fact_1997 (store_id);

INSERT INTO agg_c_14_sales_fact_1997 (
    product_id,
    customer_id,
    promotion_id,
    store_id,
    month_of_year,
    quarter,
    the_year,
    store_sales,
    store_cost,
    unit_sales,
    fact_count)
SELECT
    BASE.product_id,
    BASE.customer_id,
    BASE.promotion_id,
    BASE.store_id,
    DIM.month_of_year,
    DIM.quarter,
    DIM.the_year,
    SUM(BASE.store_sales) AS store_sales,
    SUM(BASE.store_cost) AS store_cost,
    SUM(BASE.unit_sales) AS unit_sales,
    COUNT(*) AS fact_count
FROM
    sales_fact_1997 AS BASE, time_by_day AS DIM
WHERE
    BASE.time_id = DIM.time_id
GROUP BY
    BASE.product_id,
    BASE.customer_id,
    BASE.promotion_id,
    BASE.store_id,
    DIM.month_of_year,
    DIM.quarter,
    DIM.the_year;

```

In this case, one can see that the time_id foreign key in the base fact table has been replaced with the columns: month_of_year, quarter, and the_year in the aggregate table. There is, as always, the fact_count column. The measures are inserted as sums. And, the group by clause is over the remaining foreign keys as well as the imported time dimension levels.

When creating a collapsed dimension aggregate one might consider creating indexes for the columns imported from the dimension that was collapsed.

Below is another aggregate table. This one has two lost dimensions (`store_id` and `promotion_id`) as well as collapsed dimension on time to the quarter level. This shows how aggregate techniques can be mixed.

```
CREATE TABLE agg_lc_100_sales_fact_1997 (  
    product_id INTEGER NOT NULL,  
    customer_id INTEGER NOT NULL,  
    quarter VARCHAR(30) NOT NULL,  
    the_year SMALLINT(6) NOT NULL,  
    store_sales DECIMAL(10,4) NOT NULL,  
    store_cost DECIMAL(10,4) NOT NULL,  
    unit_sales DECIMAL(10,4) NOT NULL,  
    fact_count INTEGER NOT NULL);  
  
CREATE INDEX i_sls_97_cust_id ON agg_lc_100_sales_fact_1997  
(customer_id);  
CREATE INDEX i_sls_97_prod_id ON agg_lc_100_sales_fact_1997  
(product_id);  
  
INSERT INTO agg_lc_100_sales_fact_1997 (  
    product_id,  
    customer_id,  
    quarter,  
    the_year,  
    store_sales,  
    store_cost,  
    unit_sales,  
    fact_count)  
SELECT  
    BASE.product_id,  
    BASE.customer_id,  
    DIM.quarter,  
    DIM.the_year,  
    SUM(BASE.store_sales) AS store_sales,  
    SUM(BASE.store_cost) AS store_cost,  
    SUM(BASE.unit_sales) AS unit_sales,  
    COUNT(*) AS fact_count  
FROM sales_fact_1997 AS BASE,  
    time_by_day AS DIM  
WHERE  
    BASE.time_id = DIM.time_id  
GROUP BY  
    BASE.product_id,  
    BASE.customer_id,  
    DIM.quarter,  
    DIM.the_year;
```

In the above three examples, for the most part the column names in the aggregate are the same column names that appear in the fact table and dimension tables. These tables would all be recognized by the Mondrian [default](#) aggregate recognizer. It is possible to create an aggregate table and name the columns arbitrarily. For such an aggregate, an [explicit](#) Mondrian recognizer must be specified.

```

CREATE TABLE agg_c_special_sales_fact_1997 (
    PRODUCT_ID INTEGER NOT NULL,
    CUSTOMER_ID INTEGER NOT NULL,
    PROMOTION_ID INTEGER NOT NULL,
    STORE_ID INTEGER NOT NULL,
    TIME_MONTH SMALLINT(6) NOT NULL,
    TIME_QUARTER VARCHAR(30) NOT NULL,
    TIME_YEAR SMALLINT(6) NOT NULL,
    STORE_SALES_SUM DECIMAL(10,4) NOT NULL,
    STORE_COST_SUM DECIMAL(10,4) NOT NULL,
    UNIT_SALES_SUM DECIMAL(10,4) NOT NULL,
    FACT_COUNT INTEGER NOT NULL);

CREATE INDEX i_sls_97_cust_id ON agg_c_special_sales_fact_1997
(CUSTOMER_ID);
CREATE INDEX i_sls_97_prod_id ON agg_c_special_sales_fact_1997
(PRODUCT_ID);
CREATE INDEX i_sls_97_promo_id ON agg_c_special_sales_fact_1997
(PROMOTION_ID);
CREATE INDEX i_sls_97_store_id ON agg_c_special_sales_fact_1997
(STORE_ID);

INSERT INTO agg_c_special_sales_fact_1997 (
    PRODUCT_ID,
    CUSTOMER_ID,
    PROMOTION_ID,
    STORE_ID,
    TIME_MONTH,
    TIME_QUARTER,
    TIME_YEAR,
    STORE_SALES_SUM,
    STORE_COST_SUM,
    UNIT_SALES_SUM,
    FACT_COUNT)
SELECT
    BASE.product_id,
    BASE.customer_id,
    BASE.promotion_id,
    BASE.store_id,
    DIM.month_of_year,
    DIM.quarter,
    DIM.the_year,
    SUM(BASE.store_sales) AS STORE_SALES_SUM,
    SUM(BASE.store_cost) AS STORE_COST_SUM,
    SUM(BASE.unit_sales) AS UNIT_SALES_SUM,
    COUNT(*) AS FACT_COUNT
FROM
    sales_fact_1997 BASE, time_by_day DIM
WHERE
    BASE.time_id = DIM.time_id
GROUP BY
    BASE.product_id,
    BASE.customer_id,
    BASE.promotion_id,
    BASE.store_id,
    DIM.month_of_year,

```

```
DIM.quarter,  
DIM.the_year;
```

This aggregate table has column names that are not identical to those found in the base fact table and dimension table. It is still a valid aggregate but Mondrian has to be told how to map its columns into those of the base fact table.

Sometimes with multiple aggregate tables, one aggregate table is an aggregate of not only the base fact table but also another aggregate table; an aggregate table with lost time and product dimensions (no time_id and product_id foreign keys) is an aggregate of the base fact table and an aggregate which only has a lost time dimension (no time_id foreign key). In this case, one might first build the aggregate with only the lost time dimension and then build the aggregate with both lost time and product dimensions from that first aggregate - it will be faster (in some cases, much faster) to populate the second aggregate from the first rather than from the base fact table.

One last note, when creating aggregate tables from the base fact table pay attention to the size of the numeric columns - what might be big enough in the base fact table might not be big enough in an aggregate.

How Mondrian recognizes Aggregate Tables

Mondrian has to know about the aggregate tables in order to use them. You can either define an aggregate explicitly, or set up rules to recognize several aggregate tables at the same time.

How Mondrian recognizes aggregate table names and columns pretty much dictates how one must name those table names and columns when creating them in the first place!

Rules

Rules are templates, designed to work for all fact table names and their column names. These rules are templates of regular expressions that are instantiated with the names of a fact table and its columns. In order to describe the rule templates, a name that instantiate a rule are represented in a rule by have the name bracketed by "\${" and "}". As an example, "abc_\${name}_xyz" is a rule parameterized by "name". When name is "john" the template becomes "abc_john_xyz".

The regular expression engine used here and a definition of the allowed regular expression grammar is found in the Java regular expression Pattern class: java.util.regex.Pattern.

In order that a table be recognized as an aggregate table, Mondrian must be able to map from the fact table foreign key columns and measure columns and those in the aggregate table. In addition, Mondrian must identify the fact count column in the aggregate and possible level columns (which would appear in an aggregate table if it had a "collapsed" dimension). What follows is a description of the steps taken in the identification of aggregate tables by the default recognizer. If at any step, a match fails, the table is rejected as an aggregate table.

Starting off, the candidate aggregate table's name must comply with the aggregate table name rule. Represented as a template regular expression the rule is:

```
agg_._+_${fact_table_name}
```

which is parameterized with the fact table's name. (In addition, this rule is applied in "ignore case" mode.) This means that an aggregate table's name must start with "agg_" (ignoring character case), followed by at least one character, then the '_' character and, lastly, the name of the fact table. The "." in the template has special meaning in a regular expression - it matches one or more characters.

As an example of applying the aggregate table name rule, let the fact table be called `sales_fact_1997`, the `Sales` cube's fact table from the `FoodMart` schema. Applying the specific fact table name to the regular expression template creates the following regular expression:

```
agg_+sales_fact_1997
```

This will match the following table names:

- `agg_l_05_sales_fact_1997`
- `agg_c_14_sales_fact_1997`
- `agg_lc_100_sales_fact_1997`
- `agg_c_special_sales_fact_1997`
- `AGG_45_SALES_FACT_1997`
- `AGG_drop_time_id_sales_fact_1997`

The aggregate table name recognition mechanism has one additional programatic feature, one can specify that only a portion of the base fact table name be used as the basis of template name. For instance, if the DBA demanded that all fact tables begin with the string "fact_", e.g., "fact_sales_fact_1997", one would certainly not want that string to have to be part of each aggregate table's name. The aggregate table name recognition mechanism allows one to specify a regular expression with one and only one group clause (a group clause is a pattern bracketed by '(' and ')'). Whatever is matched by the contents of the group clause is taken to be the part of the fact table name to be used in the matching template. This regular expression containing the group clause is specified as the "basename" attribute. The default Mondrian aggregate table recognizer does not use this feature. For more information see the associated [developer's note link](#).

After the default recognizer determines that a table's name matches the aggregate table template regular expression for a given fact table, it then attempts to match columns. The first column tested for is the "fact count" column. Here the candidate aggregate table must have a column called "fact_count" (ignoring case) and this column's type must be numeric. The following examples would match as "fact count" columns.

```
fact_count  
FACT_COUNT  
fact_COUNT
```

Following matching the "fact count" column, the candidate aggregate table's columns are examined for possible foreign key matches. For each of the foreign key column names in the fact table it is determined if there are any character case independent matches of the aggregate table's columns. Those columns that match are noted. It is alright if no columns match; the aggregate might be a "collapsed" dimension aggregate with no fact table foreign keys remaining. If the fact table had foreign key columns "store_id" and "time_id", then the following aggregate table columns (for example) would match:

- time_id
- store_id
- TIME_ID
- STORE_ID
- time_ID
- STORE_id

At this point, matches are looked for the level and measure columns. Both of these matching rules are multi-part - has sub rules; each rule has more than one possible regular expression that might match where a match on any one is a match.

There are three sub rules for matching level columns. Each is a template which is parameterized with 1) the fact table's cube's dimension hierarchy's name, "hierarchy_name", 2) the fact table's cube's dimension hierarchy's level name, "level_name", 3) the dimension table's level column name, "level_column_name", and 4) a usage prefix, "usage_prefix", which in most cases is null":

- `${hierarchy_name}_${level_name}`
- `${hierarchy_name}_${level_column_name}`
- `${usage_prefix}${level_column_name}`
- `${level_column_name}`

The "usage_prefix" is the value of the `DimensionUsage`'s or private `Dimension`'s optional `usagePrefix` attribute. It can be the case that a "level_column_name", the name of a dimension's level column, is the same for more than one dimension. During aggregate recognition for collapsed dimension aggregates where the base fact table has two or more dimensions with common column names, the attempted recognition will fail unless in the schema catalog the `usagePrefix` attribute is used to disambiguate those column names. Of course, one must also remember to prefix the the column in the aggregate table with the same prefix.

As an example of `usagePrefix`, consider a fact table named `ORDERS` which has two `DimensionUsages`, one for the `CUSTOMER` dimension and the other for the `WHOLESALE` dimension where each dimension has a level column named `CUST_NM`. In this case, a collapsed aggregate table could not include a column named `CUST_NM` because there would be no way to tell which dimension to associate it with. But if in the `CUSTOMER`' `DimensionUsage` the `usagePrefix` had the value "CU_", while the `WHOLESALE`'s `usagePrefix` had the value "WS_", and the aggregate table column was named `WS_CUST_NM`, then the recognizer could associate the column with the `WHOLESALE` dimension.

In the case of a private `Dimension`, a `usagePrefix` need only be used if there is a public, shared `Dimension` that has the same name and has a "level_column_name" that is also the same. Without the `usagePrefix` there would be no way of disambiguating collapsed dimension aggregate tables.

If any of these parameters have space characters, ' ', these are mapped to underscore characters, '_', and, similarly, dot characters, '.', are also mapped to underscores. So, if the hierarchy_name is "Time", level_name is "Month" and level_column_name is month_of_year, the possible aggregate table column names are:

- time_month
- time_month_of_year

- month_of_year

For this rule, the "hierarchy_name" and "level_name" are converted to lower case while the "level_column_name" must match exactly.

Lastly, there is the rule for measures. There are three parameters to matching aggregate columns to measures: 1) the fact table's cube's measure name, "measure_name", 2) the fact table's cube's measure column name, "measure_column_name", and 3) the fact table's cube's measure's aggregator (sum, avg, max, etc.), "aggregate_name".

- \${measure_name}
- \${measure_column_name}
- \${measure_column_name}_\${aggregate_name}

where the measure name is converted to lower case and both the measure column name and aggregate name are matched as they appear. If the fact table's cube's measure name was, "Avg Unit Sales", the fact table's measure column name is "unit_sales", and, lastly, the fact table's cube's measure's aggregate name is "avg", then possible aggregate table column names that would match are:

- avg_unit_sales
- unit_sales
- unit_sales_avg

For Mondrian developers there are [additional notes](#) describing the default rule recognition schema.

Explicit aggregates

On a per cube basis, in a schema file a user can both include and exclude aggregate tables. A table that would have been include as an aggregate by the default rules can be explicitly excluded. A table that would not be include by the default rules can be explicitly included. A table that would have only been partially recognized by the default rules and, therefore, resulted in a warning or error message, can be explicitly include in rules specified in the cube's definition.

Below is an example for the FoodMart Sales cube with fact table sales_fact_1997. There are child elements of the Table element that deal with aggregate table recognition.

```
<Cube name="Sales">
  <Table name="sales_fact_1997">
    <AggExclude name="agg_c_14_sales_fact_1997" />
    <AggExclude name="agg_lc_10_sales_fact_1997" />
    <AggExclude name="agg_pc_10_sales_fact_1997" />

    <AggName name="agg_c_special_sales_fact_1997">
      <AggFactCount column="FACT_COUNT"/>
      <AggIgnoreColumn column="admin_one"/>
      <AggIgnoreColumn column="admin_two"/>
      <AggForeignKey factColumn="product_id" aggColumn="PRODUCT_ID"
    />

    <AggForeignKey factColumn="customer_id" aggColumn="CUSTOMER_ID"
```

```

/>
    <AggForeignKey factColumn="promotion_id"
aggColumn="PROMOTION_ID" />
    <AggForeignKey factColumn="store_id" aggColumn="STORE_ID" />
    <AggMeasure name="[Measures].[Unit Sales]"
column="UNIT_SALES_SUM" />
    <AggMeasure name="[Measures].[Store Cost]"
column="STORE_COST_SUM" />
    <AggMeasure name="[Measures].[Store Sales]"
column="STORE_SALES_SUM" />
    <AggLevel name="[Time].[Year]" column="TIME_YEAR" />

    <AggLevel name="[Time].[Quarter]" column="TIME_QUARTER" />
    <AggLevel name="[Time].[Month]" column="TIME_MONTH" />
</AggName>
<AggPattern pattern="agg_sales_fact_1997_.*">
....
<AggExclude name="agg_sales_fact_1997_olddata" />
<AggExclude pattern="agg_sales_fact_1997_test.*" />

</AggPattern>

</Table>
....
</Cube>

```

The `AggExclude` elements define tables that should not be considered aggregates of the fact table. In this case Mondrian is instructed to ignore the tables `agg_c_14_sales_fact_1997`, `agg_lc_10_sales_fact_1997` and `agg_pc_10_sales_fact_1997`. Following the excludes is the `AggName` element which identifies the name of an aggregate table table, `agg_c_special_sales_fact_1997`, and rules for mapping names from the fact table and cube to it. The two `AggIgnoreColumn` elements are used to specifically state to Mondrian that the columns `admin_one` and `admin_two` are known and should be ignored. If these columns were not so identified, Mondrian at the end of determining the fitness of the `agg_c_special_sales_fact_1997` table to be an aggregate of the `sales_fact_1997` fact table would complain that there were extra unidentified columns and that the mapping was incomplete. The `AggForeignKey` elements define mappings from the `sales_fact_1997` fact table foreign key column names into the `agg_c_special_sales_fact_1997` aggregate table column names.

Both the `AggMeasure` and `AggLevel` elements map "logical" name, names defined in the cube's schema, to the aggregate table's column names. An aggregate table does not have to have all of the measures that are found in the base fact table, so it is not a requirement that all of the fact table measures appear as `AggMeasure` mappings, though it will certainly be the most common case. The most notable exception are `distinct-count` measures; such a measure can be aggregated, but one can not in general aggregate further on the measure - the "distinctness" of the measure has been lost during the first aggregation.

The `AggLevel` entries correspond to collapsed dimensions. For each collapsed dimension there is a hierarchy of levels spanning from the top level down to some intermediate level (with no gaps).

The `AggName` element is followed by an `AggPattern` element. This matches candidate aggregate table names using a regular expression. Included as child elements of the `AggPattern` element are two `AggExclude` elements. These specifically state what table names should not be considered by this `AggPattern` element.

In a given `Table` element, all of the `AggExclude` are applied first, followed by the `AggName` element rules and then the `AggPattern` rules. In the case where the same fact table is used by multiple cubes, the above still applies, but its across all of the aggregation rules in all of the multiple cube's `Table` elements. The first "Agg" element, name or pattern, that matches per candidate aggregate table name has its associated rules applied.

Most of the time, the scope of these include/exclude statements apply only to the cube in question, but not always. A cube has a fact table and it is the characteristics of the fact table (like column names) against which some of the aggregate table rules are applied. But, a fact table can actually be the basis of more than one cube. In the FoodMart schema the `sales_fact_1997` fact table applies to both the `Sales` and the `Sales Ragged` cubes. What this means is that any explicit rules defined in the `Sales` cube also applies to the `Sales Ragged` cube and visa versa.

One feature of the explicit recognizer is very useful. With a single line in the cubes definition in the schema file, one can force Mondrian not to recognize any aggregate tables for the cube's fact table. As an example, for the FoodMart `Sales` cube the following excludes all aggregate tables because the regular expression pattern `".*"` matches all candidate aggregate table names.

```
<Table name="sales_fact_1997" >      <AggExclude pattern=".*" />
</Table>
```

During aggregate table recognition, rather than fail silently, Mondrian is rather noisy about things it can not figure out.

Aggregate tables and parent-child hierarchies

A [parent-child hierarchy](#) is a special kind of hierarchy where members can have arbitrary depth. The classic example of a parent-child hierarchy is an employee org-chart.

When dealing with parent-child hierarchies, the challenge is to roll up measures of child members into parent members. For example, when considering an employee Bill who is head of a department, we want to report not Bill's salary, but Bill's salary plus the sum of his direct and indirect reports (Eric, Mark and Carla). It is difficult to generate efficient SQL to do this rollup, so Mondrian provides a special structure called a [closure table](#), which contains the expanded contents of the hierarchy.

A closure table serves a similar purpose to an aggregate table: it contains a redundant copy of the data in the database, organized in such a way that Mondrian can access the data efficiently. An aggregate table speeds up aggregation, whereas a closure table makes it more efficient to compute hierarchical rollups.

Supposing that a schema contains a large fact table, and one of the hierarchies is a parent-child hierarchy. Is it possible to make aggregate tables and closure tables work together, to get better performance? Let's consider a concrete example.

Cube:
[Salary]

Dimensions:
[Employee], with level [Employee]
[Time], with levels [Year], [Quarter], [Month], [Day]

Fact table:
salary (employee_id, time_id, dollars)

Parent-child dimension table:
employee (employee_id, supervisor_id, name)

employee

supervisor_id employee_id name

null	1	Frank
1	2	Bill
2	3	Eric
1	4	Jane
3	5	Mark
2	6	Carla

Closure table:
employee_closure (employee_id, supervisor_id, depth)

employee_closure

supervisor_id employee_id distance

1	1	0
1	2	1
1	3	2
1	4	1
1	5	3
1	6	2
2	2	0
2	3	1
2	5	2
2	6	1
3	3	0
3	5	1
4	4	0
5	5	0
6	6	0

Regular dimension table:
time (year, month, quarter, time_id)

Aggregate tables at the leaf level of a parent-child hierarchy

The simplest option is to create an aggregate table which joins at the leaf level of the parent-child hierarchy. The following aggregate table is for leaf members of the [Employee] hierarchy, and the [Year] level of the [Time] hierarchy.

Aggregate table:

```
agg_salary_Employee_Time_Year (employee_id, time_year, sum_dollars)
```

```
INSERT INTO agg_salary_Employee_Time_Year
SELECT
    salary.employee_id,
    time.year AS time_year,
    sum(salary.dollars) AS sum_dollars
FROM salary,
    time
WHERE time.time_id = salary.time_id
GROUP BY salary.employee_id, time.year
```

Mondrian can use the aggregate table to retrieve salaries of leaf employees (without rolling up salaries of child employees). But because the aggregate table has the same foreign key as the salary fact table, Mondrian is able to automatically join salary.employee_id to either agg_salary_Employee_Time_Year.employee_id or agg_salary_Employee_Time_Year.supervisor_id to rollup employees efficiently.

Combined closure and aggregate tables

A more advanced option is to combine the closure table and aggregate table into one:

Aggregate table:

```
agg_salary_Employee$Closure_Time_Year (supervisor_id, time_year,
sum_dollars)
```

```
INSERT INTO agg_salary_Employee$Closure_Time_Year
SELECT
    ec.supervisor_id,
    time.year AS time_year,
    sum(salary.dollars) AS sum_dollars
FROM employee_closure AS ec,
    salary,
    time
WHERE ec.supervisor_id = salary.employee_id
AND ec.supervisor_id <> ec.employee_id
AND time.time_id = salary.time_id
GROUP BY ec.employee_id, ec.supervisor_id, time.year
```

The agg_salary_Employee\$Closure_Time_Year aggregate table contains the salary of every employee, rolled up to include their direct and indirect reports, aggregated to the [Year] level of the [Time] dimension.

The trick: How combined closure and aggregate tables work

Incidentally, this works based upon a 'trick' in Mondrian's internals. Whenever Mondrian sees a closure table, it creates a auxilliary dimension behind the scenes. In the case of the [Employee] hierarchy and its employee_closure table, the auxilliary dimension is called [Employee\$Closure].

```
Dimension [Employee$Closure], levels [supervisor_id], [employee_id]
```

When an MDX query evaluates a cell which uses a rolled up salary measure, Mondrian translates the coordinates of that cell in the [Employee] dimension into a corresponding coordinate in the [Employee\$Closure] dimension. This translation happens *before* Mondrian starts to search for a suitable aggregate table, so if your aggregate table contains the name of the auxiliary hierarchy (as agg_salary_Employee\$Closure_Time_Year contains the name of the [Employee\$Closure] hierarchy) it find and use the aggregate table in the ordinary way.

How Mondrian uses aggregate tables

Choosing between aggregate tables

If more than one aggregate table matches a particular query, Mondrian needs to choose between them.

If there is an aggregate table of the same granularity as the query, Mondrian will use it. If there is no aggregate table at the desired granularity, Mondrian will pick an aggregate table of lower granularity and roll up from it. In general, Mondrian chooses the aggregate table with the fewest rows, which is typically the aggregate table with the fewest extra dimensions. See property [mondrian.rolap.aggregates.ChooseByVolume](#).

Distinct count

There is an important exception for distinct-count measures: they cannot in be rolled up over arbitrary dimensions. To see why, consider the case of a supermarket chain which has two stores in the same city. Suppose that Store A has 1000 visits from 800 distinct customers in the month of July, while Store B has 1500 visits from 900 distinct customers. Clearly the two stores had a total of 2500 customer visits between them, but how many distinct customers? We can say that there were at least 900, and maybe as many as 1700, but assuming that some customers visit both stores, and the real total will be somewhere in between. "Distinct customers" is an example of a distinct-count measure, and cannot be deduced by rolling up subtotals. You have to go back to the raw data in the fact table.

There is a special case where it is acceptable to roll up distinct count measures. Suppose that we know that in July, this city's stores (Store A and B combined) have visits from 600 distinct female customers and 700 distinct male customers. Can we say that the number of distinct customers in July is 1300? Yes we can, because we know that the sets of male and female customers cannot possibly overlap. In technical terms, gender is *functionally dependent on* customer id.

The rule for rolling up distinct measures can be stated as follows:

A distinct count measure over key k can be computed by rolling up more granular subtotals only if the attributes which are being rolled up are functionally dependent on k .

Even with this special case, it is difficult to create enough aggregate tables to satisfy every possible query. When evaluating a distinct-count measure, Mondrian can only use an aggregate table if it has the same logical/level granularity as the cell being requested, or can be rolled up to that granularity only by dropping functionally dependent attributes. If there is no aggregate table of the desired granularity, Mondrian goes instead against the fact table.

This has implications for aggregate design. If your application makes extensive use of distinct-count measures, you will need to create an aggregate table for each granularity where it is used. That could be a lot of aggregate tables! (We hope to have a better solution for this problem in future releases.)

That said, Mondrian will rollup measures in an aggregate table that contains one or more distinct-count measures if none of the distinct-count measures are requested. In that respect an aggregate table containing distinct-count measures are just like any other aggregate table as long as the distinct-count measures are not needed. And once in memory, distinct-count measures are cached like other measures, and can be used for future queries.

When building an aggregate table that will contain a distinct-count measure, the measure must be rolled up to a logical dimension level, which is to say that the aggregate table must be a collapsed dimension aggregate. If it is rolled up only to the dimension's foreign key, there is no guarantee that the foreign key is at the same granularity as the lowest logical level, which is what is used by MDX requests. So for an aggregate table that only rolls the distinct-count measure to the foreign key granularity, a request of that distinct-count measure may result in further rollup and, therefore, an error.

Consider the following aggregate table that has lost dimensions `customer_id`, `product_id`, `promotion_id` and `store_id`.

```
INSERT INTO "agg_l_04_sales_fact_1997" (  
    "time_id",  
    "store_sales",  
    "store_cost",  
    "unit_sales",  
    "customer_count",  
    "fact_count"  
) SELECT  
    "time_id",  
    SUM("store_sales") AS "store_sales",  
    SUM("store_cost") AS "store_cost",  
    SUM("unit_sales") AS "unit_sales",  
    COUNT(DISTINCT "customer_id") AS "customer_count",  
    COUNT(*) AS "fact_count"  
FROM "sales_fact_1997"  
GROUP BY "time_id";
```

This aggregate table is useless for computing the `"customer_count"` measure. Why? The distinct-count measure is rolled up to the `time_id` granularity, the lowest level granularity of the physical database table `time_by_day`. Even a query against the lowest level in the `Time`

dimension would require a rollup from `time_id` to `month_of_year`, and this is impossible to perform.

Now consider this collapsed Time dimension aggregate table that has the same lost dimensions `customer_id`, `product_id`, `promotion_id` and `store_id`. The `time_id` foreign key is no longer present, rather it has been replaced with the logical levels `the_year`, `quarter` and `month_of_year`.

```
INSERT INTO "agg_c_10_sales_fact_1997" (  
    "month_of_year",  
    "quarter",  
    "the_year",  
    "store_sales",  
    "store_cost",  
    "unit_sales",  
    "customer_count",  
    "fact_count"  
) SELECT  
    "D"."month_of_year",  
    "D"."quarter",  
    "D"."the_year",  
    SUM("B"."store_sales") AS "store_sales",  
    SUM("B"."store_cost") AS "store_cost",  
    SUM("B"."unit_sales") AS "unit_sales",  
    COUNT(DISTINCT "customer_id") AS "customer_count",  
    COUNT(*) AS fact_count  
FROM  
    "sales_fact_1997" "B",  
    "time_by_day" "D"  
WHERE  
    "B"."time_id" = "D"."time_id"  
GROUP BY  
    "D"."month_of_year",  
    "D"."quarter",  
    "D"."the_year";
```

This aggregate table of the distinct-count measure can be used to fulfill a query as long as the query specifies the Time dimension down to the `month_of_year` level.

The general rule when building aggregate tables involving distinct-count measures is that there can be NO foreign keys remaining in the aggregate table - for each base table foreign key, it must either be dropped, a lost dimension aggregate, or it must be replaced with levels, a collapsed dimension aggregate. In fact, this rule, though not required, is useful to follow when creating any aggregate table; there is no value in maintaining foreign keys in aggregate tables. They should be replaced by collapsing to levels unless the larger memory used by such aggregate tables is too much for one's database system.

A better design for the aggregate table would include a few attributes which are functionally dependent on `customer_id`, the key for the distinct-count measure:

```
INSERT INTO "agg_c_12_sales_fact_1997" (  
    "country",  
    "gender",  
    "marital_status",  
    "month_of_year",  
    "quarter",  
    "the_year",  
    "store_sales",  
    "store_cost",  
    "unit_sales",  
    "customer_count",  
    "fact_count"  
) SELECT  
    "D"."month_of_year",  
    "D"."quarter",  
    "D"."the_year",  
    SUM("B"."store_sales") AS "store_sales",  
    SUM("B"."store_cost") AS "store_cost",  
    SUM("B"."unit_sales") AS "unit_sales",  
    COUNT(DISTINCT "customer_id") AS "customer_count",  
    COUNT(*) AS fact_count  
FROM  
    "sales_fact_1997" "B",  
    "time_by_day" "D",  
    "customer" "C"  
WHERE  
    "B"."time_id" = "D"."time_id"  
AND "B"."customer_id" = "C"."customer_id"  
GROUP BY  
    "C"."country",  
    "C"."gender",  
    "C"."marital_status",  
    "D"."month_of_year",  
    "D"."quarter",  
    "D"."the_year";
```

The added attributes are "country", "gender" and "marital_status". This table has only approximately 12x the number of rows of the previous aggregate table (3 values of `country` x 2 values of `gender` x 2 values of `marital_status`) but can answer many more potential queries.

Tools for designing and maintaining aggregate tables

Aggregate tables are difficult to design and maintain. We make no bones about it. But this is the first release in which aggregate tables have been available, and we decided to get the internals right rather than building a toolset to make them easy to use.

Unless your dataset is very large, Mondrian's performance will be just fine without aggregate tables. If Mondrian isn't performing well, you should first check that your DBMS is well-tuned: see our guide to [optimizing performance](#)). If decide to build aggregate tables anyway, we don't offer

any tools to help administrators design them, so unless you are blessed with superhuman patience and intuition, using them won't be smooth sailing.

Here are some ideas for tools we'd like to build in the future. I'm thinking of these being utilities, not part of the core runtime engine. There's plenty of room to wrap these utilities in nice graphical interfaces, make them smarter.

AggGen (aggregate generator)

AggGen is a tool that generates SQL to support the creation and maintenance of aggregate tables, and would give a template for the creation of materialized views for databases that support those. Given an MDX query, the generated create/insert SQL is optimal for the given query. The generated SQL covers both the "lost" and "collapsed" dimensions. For usage, see the documentation for [CmdRunner](#).

Aggregate table populator

This utility populates (or generates INSERT statements to populate) the agg tables.

For extra credit: populate the tables in topological order, so that higher level aggregations can be built from lower level aggregations. (See [[AAD+96](#)]).

Script generator

This utility generates a script containing CREATE TABLE and CREATE INDEX statements all possible aggregate tables (including indexes), XML for these tables, and comments indicating the estimated number of rows in these tables. Clearly this will be a huge script, and it would be ridiculous to create all of these tables. The person designing the schema could copy/paste from this file to create their own schema.

Recommender

This utility (maybe graphical, maybe text-based) recommends a set of aggregate tables. This is essentially an optimization algorithm, and it is described in the academic literature [[AAD+96](#)]. Constraints on the optimization process are the amount of storage required, the estimated time to populate the agg tables.

The algorithm could also take into account usage information. A set of sample queries could be an input to the utility, or the utility could run as a background task, consuming the query log and dynamically making recommendations.

Online/offline control

This utility would allow agg tables to be taken offline/online while Mondrian is still running.

Properties which affect aggregates

Mondrian has properties that control the behavior of its aggregate table sub-system. (You can find the full set of properties in the [Configuration Guide](#).)

Property	Type	Default Value	Description
mondrian.rolap.aggregates.Use	boolean	false	<p>If set to true, then Mondrian uses any aggregate tables that have been read. These tables are then candidates for use in fulfilling MDX queries. If set to false, then no aggregate table related activity takes place in Mondrian.</p>
mondrian.rolap.aggregates.Read	boolean	false	<p>If set to true, then Mondrian reads the database schema and recognizes aggregate tables. These tables are then candidates for use in fulfilling MDX queries. If set to false, then aggregate table will not be read from the database. Of course, after aggregate tables have been read, they are read, so setting this property false after starting with the property being true, has no effect. Mondrian will not actually use the aggregate tables unless the <code>mondrian.rolap.aggregates.Use</code> property is set to true.</p>
mondrian.rolap.aggregates.ChooseByVolume	boolean	false	<p>Currently, Mondrian support to algorithms for selecting which aggregate table to use: the aggregate with smallest row count or the aggregate with smallest volume (row count * row size). If set to false, then row count is used. If true, then volume is used.</p>
mondrian.rolap.aggregates.rules	resource or url	/Default Rules.xml	<p>This is a developer property, not a user property. Setting this to a url (e.g., <code>file:///c:/myrules.xml</code>) allows one to use their own "default" Mondrian aggregate table recognition rules. In general use this should never be changed from the default value.</p>
mondrian.rolap.aggregates.rule.tag	string	default	<p>This is also a developer property. It allows one to pick which named rule in the default rule file to use. In general use this should never be changed from the default value.</p>

Aggregate Table References

- [AAD⁺96] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In Proc. 22nd VLDB, pages 506-521, Mumbai, Sept. 1996. [[pdf](#)]
- [ABDGHLS99] J. Albrecht, A. Bauer, O. Deyerling, H. Gunze, W. Hummer, W. Lehner, L. Schlesinger. Management of Multidimensional Aggregates for Efficient Online Analytical Processing. Proceedings of International Database Engineering and Applications Symposium, 1999, pp. 156–164. [[pdf](#)]
- [GBLP96] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In Proc. 12th ICDE, pages 152-159, New Orleans, March 1996. [[pdf](#)]
- [HNSS95] P.J. Haas, J.F. Naughton, S. Seshadri, and L. Stokes. Sampling-based estimation of the number of distinct values of an attribute. Proceedings of the Eighth International Conference on Very Large Databases (VLDB), pages 311–322, Zurich, Switzerland, September 1995. [[pdf](#)]
- [Rittman05] M. Rittman. Compressed Composites (Oracle 10g Compression) Explained. Online article. [[html](#)]
- [SDNR96] Amit Shukla, Prasad Deshpande, Jeffrey F. Naughton, Karthikeyan Ramasamy. Storage Estimation for Multidimensional Aggregates in the Presence of Hierarchies. VLDB 1996, pp. 522–531. [[pdf](#)]

Mondrian CmdRunner

Copyright (C) 2005-2006 Julian Hyde, Richard Emberson and others

What is CmdRunner?

`CmdRunner` is a command line interpreter for Mondrian. From within the command interpreter or in a command file: properties can be set and values displayed, logging levels changed, built-in function usages displayed, parameter values displayed and set, per-cube attributes displayed and set, results and errors from the previous MDX command displayed and, of course, MDX queries evaluated.

For Mondrian developers new features can be quickly tested with `CmdRunner`. As an example, to test a new user-defined function all one need to is add it to the schema, add the location of the function's java class to the class path, point `CmdRunner` at the schema and execute a MDX query that uses the new function.

For MDX developers, `CmdRunner` lets one test a new MDX query or Mondrian schema without having to run Mondrian in a Webserver using JPivot. Rather, one can have the new MDX query in a file and point `CmdRunner` at it. Granted, the output is a list, possibly long, of row and column entries; but sometimes all one needs from `CmdRunner` is to know that the query runs and other times one can always post process the output into excel or gnuplot, etc.

Building

There are two ways to run the command interpreter. The first is to have a script create a class path with all of the needed mondrian and support jars in it and then have java execute the `CmdRunner` main method. The second is to build a jar that contains all of the needed classes and simply have java reference the jar using the `-jar` argument.

To build the `CmdRunner` combined jar from the shell command line execute the following build command:

```
mondrian> ./build.sh cmdrunner
```

This will create the jar `cmdrunner.jar` in the `MONDRIAN_HOME/lib` directory. For this build to create a jar that can actually be used it is important that the `JDBC` jar for your database be placed in the `MONDRIAN_HOME/testlib` directory prior to executing the build command.

What is useful about the `cmdrunner.jar` is that it can be executed without having to have the `MONDRIAN_HOME` directory around since it bundles up everything that is needed (other than the properties and schema files).

Usage

There are two ways to invoke `CmdRunner`: using the `cmdrunner.jar` or using a script that builds a class path of the required jars and then executes java with that class path. The former is an easy "canned" solution but requires building the `cmdrunner.jar` while the later is quicker if you are in a code, compile and test cycle.

To run CmdRunner using the cmdrunner.jar from the shell prompt execute:

```
somedir> java -jar cmdrunner.jar -p foodmart.properties
```

In the `MONDRIAN_HOME/bin` directory there are the shell scripts `cmdrunner.sh` and `cmdrunner.cmd` that can be used duplicating the above command:

```
mondrian> ./bin/cmdrunner.sh -p foodmart.properties
```

To run CmdRunner without first building the `cmdrunner.jar` there is the `run.sh` in the `MONDRIAN_HOME/bin` directory. This script creates a class path and includes all jars in the `MONDRIAN_HOME/testlib` directory where the jdbc jars are located.

```
mondrian> ./bin/run.sh -p foodmart.properties
```

Properties File

Below is an example properties file:

```
#####  
#####  
#  
# Example properties file  
#  
# $Id: //open/mondrian/doc/cmdrunner.html#10 $  
#####  
#####  
# Environment  
mondrian.catalogURL=file:///home/madonna/mondrian/FoodMartSchema.xml  
  
# mysql  
mondrian.foodmart.jdbcURL=jdbc:mysql://localhost/foodmart?user=foodmart  
&password=foodmart  
mondrian.jdbcDrivers=com.mysql.jdbc.Driver  
  
# Use MD5 based caching for the RolapSchema instance  
mondrian.catalog.content.cache.enabled=true  
  
# both read and use aggregate tables  
mondrian.rolap.aggregates.Use=true  
mondrian.rolap.aggregates.Read=true  
  
# generate aggregate sql (for every mdx query)  
#mondrian.rolap.aggregates.generateSql=true  
  
# pretty print sql (if log level for mondrian.rolap.RolapUtil is DEBUG)  
mondrian.rolap.generate.formatted.sql=true  
  
# by default the aggregate table with the smallest number of rows  
# (rather than rows times size of each row) is used  
#mondrian.rolap.aggregates.ChooseByVolume=true
```

Command line arguments

CmdRunner has the following command line options:

Option	Description
-h	Print help, the list of command line options.
-d	Enable CmdRunner debugging. This does not change this log level.
-t	Time each mdx query's execution.
-nocache	Regardless of the settings in the Schema file, set each Cube to no in-memory aggregate caching (caching is turned off so each query goes to the database).
-rc	Do not reload the connection after each query (the default is to reload the connection. Its safe to just ignore this.
-p property-file	Specify the Mondrian property file. This argument is basically required for any but the most trivial command interpreter commands. To execute a MDX query or request information about a function, the property file must be supplied. On the other hand, to have the CmdRunner print out its internal help, then the property file is not needed.
-f filename+	Specify the name of one or more files that contains CmdRunner commands. If this argument is not supplied, then the interpreter starting in the command entry mode. After the -f is seen, all subsequent arguments are interpreted as filenames.
-x xmla_filename+	Specify the name of one or more files that contains XMAL request that has no SOAP wrapper. After the -x is seen, all subsequent arguments are interpreted as XMLA filenames.
-xs soap_xmla_filename+	Specify the name of one or more files that contains XMAL request with a SOAP wrapper. After the -xs is seen, all subsequent arguments are interpreted as SOAP XMLA filenames.
-vt	Validate the XMLA response using XSLT transform. This can only be used with the -x or -xs flags.
-vx	Validate the XMLA response using XPath. This can only be used with the -x or -xs flags.
mdx_command	A string representing one or more CmdRunner commands.

CmdRunner Commands

The command interpreter has a fixed set of built in commands. When a line is read, if the first word of the line matches one of the commands, then the rest of the line is assumed to be arguments to that command. On the other hand, if the first word does not match a built in command, then all text until a ';' is seen or until a '=' is entered by itself on a command continuation line is seen will be passed to the Mondrian query engine.

help

```
> help <cr>
```

Prints help for all commands.

set

```
> set [ property[=value ] ] <cr>
```

With no args, prints all mondrian properties and values.

With "property" prints property's value.

With "property=value" set property to that value.

log

```
> log [ classname[=level ] ] <cr>
```

With no args, prints the current log level of all classes.

With "classname" prints the current log level of the class.

With "classname=level" set log level to new value.

file

```
> file [ filename | '=' ] <cr>
```

With no args, prints the last filename executed.

With "filename", read and execute filename.

With "=" character, re-read and re-execute previous filename.

list

```
> list [ cmd | result ] <cr>
```

With no arguments, list previous cmd and result

With "cmd" argument, list the last mdx query cmd.

With "result" argument, list the last mdx query result.

func

```
> func [ name ] <cr>
```

With no arguments, list all defined function names.

With "name" argument, display the functions: name, description, and syntax.

param

```
> param [ name[=value ] ] <cr>
```

With no arguments, all param name/value pairs are printed.

With "name" argument, the value of the param is printed.

With "name=value" sets the parameter with name to value. If name is null, then unsets all parameters. If value is null, then unsets the parameter associated with value.

cube

```
> cube [ cubename [ name [=value | command] ] ] <cr>
```

With no arguments, all cubes are listed by name.

With "cubename" argument, cube attribute name/values for: fact table (readonly) aggregate caching (readwrite) are printed.

With "cubename name=value", sets the readwrite attribute with name to value.

With "cubename command", executes the commands: clearCache.

error

```
> error [ msg | stack ] <cr>
```

With no arguments, both message and stack are printed.

With "msg" argument, the Error message is printed.

With "stack" argument, the Error stack trace is printed.

echo

```
> echo text <cr>
```

Prints text to standard out.

expr

```
> expr cubename expression <cr>
```

Evaluates an expression against a cube

=

```
> = <cr>
```

Re-executes previous MDX query.

~

```
> ~ <cr>
```

Clears any text entered so far for the current command.

exit

```
> exit <cr>
```

Exits the MDX command interpreter.

run an MDX query

```
> <mdx query> ( [ ';' ] <cr> | <cr> ( '=' | '~' ) <cr> )
```

Executes or cancels an MDX query.

An MDX query may span one or more lines. The continuation prompt is a '?'.
After the last line of the query has been entered, on the next line a single execute character, '=', may be entered followed by a carriage return. The lone '=' informs the interpreter that the query has has been entered and is ready to execute.

At anytime during the entry of a query the cancel character, '~', may be entered alone on a line. This removes all of the query text from the the command interpreter.

Queries can also be ended by using a semicolon ';' at the end of a line.

During general operation, Mondrian Property triggers are disabled. If you enable Mondrian Property triggers for a CmdRunner session, either in the property file read on startup or by explicitly using the `set` property command

```
> set mondrian.olap.triggers.enable=true <cr>
```

```
> set mondrian.olap.triggers.enable=true <cr>
```

then one can force a re-scanning of the database for aggregate tables by disabling and then re-enabling the use of aggregates:

```
> set mondrian.olap.aggregates.Read=false <cr>  
> set mondrian.olap.aggregates.Read=true <cr>
```

In fact, as long as one does not use the `-rc` command line argument so that a new connection is gotten every time a query is executed, one can edit the Mondrian schema file between MDX

query execute. This allows one to not only change what aggregates tables are in seen by Mondrian but also the definitions of the cubes within a given CmdRunner session.

Similarly, one can change between aggregate table partial ordering algorithm by changing the value of the associated property, `mondrian.olap.aggregates.ChooseByVolume` thus triggering internal code to reorder the aggregate table lookup order.

Within the command interpreter there is no ability to edit a previously entered MDX query. If you wish to iteratively edit and run a MDX query, put the query in a file, tell the CmdRunner to execute the file using the `file` command, re-execute the file using the `=` command, and in separate window edit/save MDX in the file.

There is also no support for a command history (other than the '=' command).

AggGen: Aggregate SQL Generator

Mondrian release 1.2 introduced [Aggregate Tables](#) as a means of improving performance, but aggregate tables are difficult to use without tools to support them.

CmdRunner includes a utility called `AggGen`, the Aggregate Table Generator. With it, you can issue an MDX query, and generate a script to create and populate the appropriate aggregate tables to support that MDX query. (The query does not actually return a result.)

In the property file provided to the `CmdRunner` at startup add the line:

```
mondrian.rolap.aggregates.generateSql=true
```

or from the `CmdRunner` command line enter:

```
> set mondrian.rolap.aggregates.generateSql=true <cr>
```

This instructs Mondrian whenever an MDX query is executed (and the cube associated with the query is not virtual) to output to standard out the `Sql` associated with the creation and population of both the "lost" dimension aggregate table and the "collapsed" dimension aggregate table which would be best suited to optimize the given MDX query. This `Sql` has to be edited to change the "`I_XXX`" in the "lost" dimension statements or "`c_XXX`" in the "collapsed" dimension statements to more appropriate table names (remembering to make sure that the new names can still be recognized by Mondrian as aggregates of the particular fact table).

As an example, if the following MDX is run against a `MySQL` system:

```
WITH MEMBER
    [Store].[Nat'l Avg] AS
    'AVG( { [Store].[Store Country].Members}, [Measures].[Units
Shipped])'
SELECT
    { [Store].[Store Country].Members, [Store].[Nat'l Avg] } ON
COLUMNS,
    { [Product].[Product Family].[Non-Consumable].Children } ON ROWS
FROM
    [Warehouse]
```

```
WHERE
    [Measures].[Units Shipped];
```

Then the following is written to standard output:

```
WARN [main] AggGen For RolapStar: "inventory_fact_1997" measure with
name, "warehouse_sales"- "inventory_fact_1997"."warehouse_cost", is not
a column
name. The measure's column name may be an expression and currently
AggGen does
not handle expressions. You will have to add this measure to the
aggregate table
definition by hand.
```

```
CREATE TABLE agg_l_XXX_inventory_fact_1997 (
    time_id INT,
    product_id INT NOT NULL,
    store_id INT,
    store_invoice DECIMAL(10,4),
    supply_time SMALLINT,
    warehouse_cost DECIMAL(10,4),
    warehouse_sales DECIMAL(10,4),
    units_shipped INT,
    units_ordered INT,
    fact_count INTEGER NOT NULL);

INSERT INTO agg_l_XXX_inventory_fact_1997 (
    time_id,
    product_id,
    store_id,
    store_invoice,
    supply_time,
    warehouse_cost,
    warehouse_sales,
    units_shipped,
    units_ordered,
    fact_count)
SELECT
    `inventory_fact_1997`.`time_id` AS `time_id`,
    `inventory_fact_1997`.`product_id` AS `product_id`,
    `inventory_fact_1997`.`store_id` AS `store_id`,
    SUM(`inventory_fact_1997`.`store_invoice`) AS `store_invoice`,
    SUM(`inventory_fact_1997`.`supply_time`) AS `supply_time`,
    SUM(`inventory_fact_1997`.`warehouse_cost`) AS `warehouse_cost`,
    SUM(`inventory_fact_1997`.`warehouse_sales`) AS `warehouse_sales`,
    SUM(`inventory_fact_1997`.`units_shipped`) AS `units_shipped`,
    SUM(`inventory_fact_1997`.`units_ordered`) AS `units_ordered`,
    COUNT(*) AS `fact_count`
FROM
    `inventory_fact_1997` AS `inventory_fact_1997`
GROUP BY
    `inventory_fact_1997`.`time_id`,
    `inventory_fact_1997`.`product_id`,
    `inventory_fact_1997`.`store_id`;

CREATE TABLE agg_c_XXX_inventory_fact_1997 (
```

```

product_family VARCHAR(30),
product_department VARCHAR(30),
store_country VARCHAR(30),
the_year SMALLINT,
store_invoice DECIMAL(10,4),
supply_time SMALLINT,
warehouse_cost DECIMAL(10,4),
warehouse_sales DECIMAL(10,4),
units_shipped INT,
units_ordered INT,
fact_count INTEGER NOT NULL);

INSERT INTO agg_c_XXX_inventory_fact_1997 (
    product_family,
    product_department,
    store_country,
    the_year,
    store_invoice,
    supply_time,
    warehouse_cost,
    warehouse_sales,
    units_shipped,
    units_ordered,
    fact_count)
SELECT
    `product_class`.`product_family` AS `product_family`,
    `product_class`.`product_department` AS `product_department`,
    `store`.`store_country` AS `store_country`,
    `time_by_day`.`the_year` AS `the_year`,
    SUM(`inventory_fact_1997`.`store_invoice`) AS `store_invoice`,
    SUM(`inventory_fact_1997`.`supply_time`) AS `supply_time`,
    SUM(`inventory_fact_1997`.`warehouse_cost`) AS `warehouse_cost`,
    SUM(`inventory_fact_1997`.`warehouse_sales`) AS `warehouse_sales`,
    SUM(`inventory_fact_1997`.`units_shipped`) AS `units_shipped`,
    SUM(`inventory_fact_1997`.`units_ordered`) AS `units_ordered`,
    COUNT(*) AS `fact_count`
FROM
    `inventory_fact_1997` AS `inventory_fact_1997`,
    `product_class` AS `product_class`,
    `product` AS `product`,
    `store` AS `store`,
    `time_by_day` AS `time_by_day`
WHERE
    `product`.`product_class_id` = `product_class`.`product_class_id`
and
    `inventory_fact_1997`.`product_id` = `product`.`product_id` and
    `inventory_fact_1997`.`store_id` = `store`.`store_id` and
    `inventory_fact_1997`.`time_id` = `time_by_day`.`time_id`
GROUP BY
    `product_class`.`product_family`,
    `product_class`.`product_department`,
    `store`.`store_country`,
    `time_by_day`.`the_year`;

```

There are a couple of things to notice about the output.

First, is the `WARN` log message. This appears because the `inventory_fact_1997` table has a measure with a column attribute `"warehouse_sales" - "inventory_fact_1997"."warehouse_cost"` that is not a column name, its an expression. The `AggGen` code does not currently know what to do with such an expression, so it issues a warning. A user would have to take the generated aggregate table `Sql` scripts and alter them to accommodate this measure.

There are two aggregate tables, `agg_l_XXX_inventory_fact_1997` the "lost" dimension case and `agg_c_XXX_inventory_fact_1997` the "collapsed" dimension case. The "lost" dimension table, keeps the foreign keys for those dimension used by the MDX query and discards the other foreign keys, while the "collapsed" dimension table also discards the foreign keys that are not needed but, in addition, rolls up or collapses the remaining dimensions to just those levels needed by the query.

There are no indexes creation `Sql` statements for the aggregate tables. This is because not all databases require indexes to achive good performance against star schemas - your mileage may vary so do some testing. (With `MySQL` indexes are a good idea).

If one is creating a set of aggregate tables, there are cases where it is more efficient to create the set of aggregates that are just above the fact tables and then create each subsequent level of aggregates from one of the preceeding aggregate tables rather than always going back to the fact table.

There are many possible aggregate tables for a given set of fact tables. `AggGen` just provides example `Sql` scripts based upon the MDX query run. Judgement has to be used when creating aggregate tables. There are tradeoffs such as which are the MDX queries that are run the most often? How much space does each aggregate table take? How long does it take to create the aggregate tables? How often does the set of MDX queries change? etc.

During normal Mondrian operation, for instance, with `JPivot`, it is recommended that the above `AggGen` property not be set to true as it will slow down Mondrian and generate a lot of text in the log file.

Mondrian FAQs

Copyright (C) 2002-2007 Julian Hyde

Why doesn't Mondrian use a standard API?

Because there isn't one. MDX is a component of Microsoft's OLE DB for OLAP standard which, as the name implies, only runs on Windows. Mondrian's API is fairly similar in flavor to ADO MD (ActiveX Data Objects for Multidimensional), a API which Microsoft built in order to make OLE DB for OLAP easier to use.

XML for Analysis is pretty much OLE DB for OLAP expressed in Web Services rather than COM, and therefore seems to offer a platform-neutral standard for OLAP, but take-up seems to be limited to vendors who supported OLE DB for OLAP already.

The other query vendors failed to reach consensus several years ago with the OLAP Council API, and are now encamped on the JOLAP specification.

We plan to provide a JOLAP API to Mondrian as soon as JOLAP is available.

How does Mondrian's dialect of MDX differ from Microsoft Analysis Services?

See [MDX language specification](#).

Not very much.

1. The `StrToSet()` and `StrToTuple()` functions take an extra parameter.
2. Parsing is case-sensitive.
3. Pseudo-functions `Param()` and `ParamRef()` allow you to create parameterized MDX statements.

How can Mondrian be extended?

See User-defined functions, Cell readers, Member readers

Can Mondrian handle large datasets?

Yes, if your RDBMS can. We delegate the aggregation to the RDBMS, and if your RDBMS happens to have materialized group by views created, your query will fly. And the next time you run the same or a similar query, that will really fly, because the results will be in the aggregation cache. See the Performance section in this document.

How do I enable tracing?

To enable tracing, set `mondrian.trace.level` to 1 in `mondrian.properties`. You will see text and execution time of each SQL statement, like this:

```
SqlMemberSource.getLevelMemberCount: executing sql [select count(*) as  
`c0` from (select distinct `store`.`store_country` as `c0` from `store`
```

```
as `store`) as `foo`], 110 ms
SqlMemberSource.getMembers: executing sql [select distinct
`store`.`store_sqft` as `c0` from `store` as `store` order by
`store`.`store_sqft`], 50 ms
```

Notes:

- If you are running mondrian from the command-line, or via Ant, `mondrian.properties` should be in the current directory.
- If you are running in Tomcat, `mondrian.properties` should be in `TOMCAT_HOME/bin`. Changes will only take effect when you re-start Tomcat. The output goes to the console from which you started Tomcat.

How do I enable logging?

Mondrian uses the [Apache Log4j logger](#). To build, test, and run Mondrian requires a `log4j.jar` file. A `log4j.jar` file is provided as part of the Mondrian distribution.

Also provided is a `log4j.properties` file. Such a file is needed when running Mondrian in standalone mode (such as when running the Mondrian junit tests or the `CmdRunner` utility). Generally, Mondrian is embedded in an application, such as a webserver, which may have their own `log4j.properties` file or some other mechanism for setting log4j properties. In such cases, the user must use those for controlling Mondrian's logging.

Mondrian follows Apache's guidance on what type of information is logged at what level:

- **FATAL:** A very severe error event that will presumably lead the application to abort.
- **ERROR:** An error event that might still allow the application to continue running.
- **WARN:** A potentially harmful situation.
- **INFO:** An informational message that highlight the progress of the application at a coarse-grained level.
- **DEBUG:** A fine-grained informational event that is most useful to debug an application.

It is recommended for general use that the Mondrian log level be set to `WARN`; arguably, its good to know when things are going South.

Where can I find out more?

[MDX Solutions with Microsoft SQL Server Analysis Services by George Spofford](#) is the best book I have found on MDX. Despite the title, principles it describes can be applied to any RDBMS.

[OLAP Solutions: Building Multidimensional Information Systems by Erik Thomsen](#) is a great overview of multidimensional databases, but does not deal with MDX.

The reference work on data warehousing is [The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling \(Second Edition\), by Ralph Kimball, Margy Ross](#). It covers the business process well, but the focus is more on star schemas and ROLAP than OLAP.

The [Microsoft Analysis Services online documentation](#) has excellent online documentation of MDX, including a [list of MDX functions](#).

OLAP Modeling

Measures not stored in the fact table

I am trying to build a cube with measures from 2 different tables. I have tried a virtual cube, but it does not seem to work - it only relates measures and dimensions from the same table. Is there a way to specify that a measure is not coming from the fact table? Say using SQL select?

Virtual cubes sound like the right approach. The way to do it is to first create a dummy cube on your lookup table, with dimensions for as many columns as are applicable. (A classic example of this kind of cube is an 'ExchangeRate' cube, whose only dimensions are time and currency.)

Then create a virtual cube of the dummy cube and the real cube (onto your fact table).

Note that you will need to use shared dimensions for the cubes to join implicitly.

How can I define my fact table based on an arbitrary SQL statement?

Use the <View> element INSTEAD OF the <Table> element. You need to specify the 'alias' attribute, which Mondrian uses as a table alias.

The XML 'CDATA' construct is useful in case there are strange characters in your SQL, but isn't essential.

```
<View alias="DFACD_filtered"> <SQL dialect="generic"> <![CDATA[select *
from DFACD where CSOC = '09']]> </SQL> </View>
```

Why can't Mondrian find my tables?

Consider this scenario. I have created some tables in Oracle, like this:

```
CREATE TABLE sales ( prodid INTEGER, day INTEGER, amount NUMBER);
```

and referenced it in my schema.xml like this:

```
<Cube name="Sales"> <Table name="sales"/> ... <Measure name="Sales"
column="amount" aggregator="sum"/> <Measure name="Sales count"
column="prodid" aggregator="count"/> </Cube>
```

Now I start up Mondrian and get an error ORA-00942: Table or view "sales" does not exist while executing the SQL statement SELECT "prodid", count(*) FROM "sales" GROUP BY "prodid". The query looks valid, and the table exists, so why is Oracle giving an error?

The problem is that table and column names are case-sensitive. You told Mondrian to look for a table called "sales", not "SALES" or "Sales".

Oracle's table and column names are case-sensitive too, provided that you enclose them in double-quotes, like this:

```
CREATE TABLE "sales" ( "prodid" INTEGER, "day" INTEGER, "amount"
NUMBER );
```

If you omit the double-quotes, Oracle automatically converts the identifiers to upper-case, so the first `CREATE TABLE` command actually created a table called "SALES". When the query gets run, Mondrian is looking for a table called "sales" (because that's what you called it in your schema.xml), yet Oracle only has a table called "SALES".

There are two possible solutions. The simplest is to change the objects to upper-case in your schema.xml file:

```
<Cube name="Sales"> <Table name="SALES"/> ... <Measure name="Sales"
column="AMOUNT" aggregator="sum"/> <Measure name="Sales count"
column="PRODID" aggregator="count"/> </Cube>
```

Alternatively, if you decide you would like your table and column names to be in lower or mixed case (or even, for that matter, to contain spaces), then you must double-quote object names when you issue `CREATE TABLE` statements to Oracle.

Performance

When I change the data in the RDBMS, the result doesn't change even if i refresh the browser. Why is this?

Mondrian uses a cache to improve performance. The first time you run a query, Mondrian will execute various SQL statements to load the data (you can see these statements by turning on tracing). The next time, it will use the information in the cache.

Cache control is primitive right now. If the data in the RDBMS is modified, Mondrian has no way to know, and does not refresh its cache. If you are using the JPivot web ui and refresh the browser, that will simply regenerate the web page, not flush the cache. The only way to refresh the cache is to call the following piece of code, which flushes the entire contents:

```
mondrian.rolap.CachePool.instance().flush();
```

See [caching design](#) for more information.

Tuning the Aggregate function

I am using an MDX query with a calculated "aggregate" member. It aggregates the values between Node A and Node B. The dimension that it is aggregating on is a Time dimension. This Time dimension has a granularity of one minute. When executing this MDX query, the performance seems to be fairly bad.

Here is the query:

```

WITH MEMBER [Time].[AggregateValues] AS
  'Aggregate([Time].[2004].[October].[1].[12].[10] :
[Time].[2004].[October].[20].[12].[10])'
SELECT [Measures].[Volume] ON ROWS,
  NON EMPTY {[Service].[Name]}
WHERE ([Time].[AggregateValues])

```

Is this normal behavior? Is there any way I can speed this up?

Answer:

The performance is bad because you are pulling 19 days * 1440 minutes per day = 27360 cells from the database into memory per cell that you actually display. Mondrian is a lot less efficient at crunching numbers than the database is, and uses a lot of memory.

The best way to improve performance is to push as much of the processing to the database as possible. If you were asking for a whole month, it would be easy:

```

WITH MEMBER [Time].[AggregateValues]
AS 'Aggregate({[Time].[2004].[October]})'
SELECT [Measures].[Volume] ON ROWS,
NON EMPTY {[Service].[Name]}
WHERE ([Time].[AggregateValues])

```

But since you're working with time periods which are not aligned with the dimensional structure, you'll have to chop up the interval:

```

WITH MEMBER [Time].[AggregateValues]
AS 'Aggregate({
[Time].[2004].[October].[1].[12].[10]
: [Time].[2004].[October].[1].[23].[59],
[Time].[2004].[October].[2]
: [Time].[2004].[October].[19],
[Time].[2004].[October].[20].[0].[00]
: [Time].[2004].[October].[20].[12].[10]})'
SELECT [Measures].[Volume] ON ROWS,
NON EMPTY {[Service].[Name]}
WHERE ([Time].[AggregateValues])

```

This will retrieve a much smaller number of cells from the database — 18 days + no more than 1440 minutes — and therefore do more of the heavy lifting using SQL's GROUP BY operator. If you want to improve it still further, introduce hourly aggregates.

Results Caching – The key to performance

Copyright (C) 2002-2006 Julian Hyde

The various subsystems of Mondrian have different memory requirements. Some of them require a fixed amount of memory to do their work, whereas others can exploit extra memory to increase their performance. This is an overview of how the various subsystems use memory.

Caching is a scheme whereby a component uses extra memory when it is available in order to boost its performance, and when times are hard, it releases memory with loss of performance but with no loss of correctness. A cache is the use of extra memory when times are good, use varying amounts of memory.

Garbage collection is carried out by the Java VM to reclaim objects which are unreachable from 'live' objects. A special construct called a *soft reference* allows objects to be garbage-collected in hard times.

The garbage collector is not very discriminating in what it chooses to throw out, so Mondrian has its own caching strategy. There are several caches in the system (described below), but they all of the objects in these caches are registered in the singleton instance of [class `mondrian.rolap.CachePool`](#) (currently there is just a single instance). The cache pool doesn't actually store the objects, but handles all of the events related to their life cycle in a cache. It weighs objects' cost (some function involving their size in bytes and their usefulness, which is based upon how recently they were used) and their benefit (the effort it would take to re-compute them).

The cache pool is not infallible — in particular, it can not adapt to conditions where memory is in short supply — so uses soft references, so that the garbage collector can overrule its wisdom.

Cached objects must obey the following contract:

1. They must implement [interface `mondrian.rolap.CachePool.Cacheable`](#), which includes methods to measure objects' cost, benefit, record each time they are used, and tell them to remove themselves from their cache.
2. They must call [CachePool.register\(Cacheable\)](#) either in their constructor or, in any case, before they are made visible in their cache.
3. They must call [CachePool.unregister\(Cacheable\)](#) when they are removed from their cache and in their `finalize()` method.
4. They must be dispensable: if they disappear, their subsystem will continue to work correctly, albeit slower. A subsystem can declare an object to be temporarily indispensable by calling [CachePool.pin\(Cacheable, Collection\)](#) and then unpin it a short time later.
5. Their cache must reference them via soft references, so that they are available for garbage collection.
6. Thread safety. Their cache must be thread-safe.

If a cached object takes a significant time to initialize, it may not be possible to construct it, register it, and initialize it within the same synchronized section without unacceptably reducing concurrency. If this is the case, you should use phased construction. First construct and register the object, but mark it 'under construction'. Then release the lock on the CachePool and the object's cache, and continue initializing the object. Other threads will be able to see the object,

and should be able to wait until the object is constructed. The method [Segment.waitUntilLoaded\(\)](#) is an example of this.

The following objects are cached.

Segment

A Segment ([class mondrian.rolap.agg.Segment](#)) is a collection of cell values parameterized by a measure, and a set of (column, value) pairs. An example of a segment is

(Unit sales, Gender = 'F', State in {'CA','OR'}, Marital Status = *anything*)

All segments over the same set of columns belong to an Aggregation, in this case

('Sales' Star, Gender, State, Marital Status)

Note that different measures (in the same Star) occupy the same Aggregation. Aggregations belong to the AggregationManager, a singleton.

Segments are pinned during the evaluation of a single MDX query. The query evaluates the expressions twice. The first pass, it finds which cell values it needs, pins the segments containing the ones which are already present (one pin-count for each cell value used), and builds a cell request ([class mondrian.rolap.agg.CellRequest](#)) for those which are not present. It executes the cell request to bring the required cell values into the cache, again, pinned. Then it evaluates the query a second time, knowing that all cell values are available. Finally, it releases the pins.

Member set

A member set ([class mondrian.rolap.SmartMemberReader.ChildrenList](#)) is a set of children of a particular member. It belongs to a member reader ([class mondrian.rolap.SmartMemberReader](#)).

Schema

Schemas ([class mondrian.rolap.RolapSchema](#)) are cached in [class mondrian.rolap.RolapSchema.Pool](#), which is a singleton (todo: use soft references). The cache key is the URL which the schema was loaded from.

Star schemas

Star schemas ([class mondrian.rolap.RolapStar](#)) are stored in the static member `RolapStar.stars` (todo: use soft references), and accessed via `RolapStar.getOrCreateStar(RolapSchema, MondrianDef.Relation)`.

Learning more about Mondrian

Copyright (C) 2005-2006 Julian Hyde, Richard Emberson and others

How Mondrian generates SQL

If you're feeling mystified where the various SQL statements come from, here's a good way to learn more. Give it a try, and if you have more questions I'll be glad to answer them.

In a debugger, put a break point in the [RolapUtil.executeQuery\(\)](#) method, and run a simple query. The easiest way to run a query is to run a junit testcase such as [BasicQueryTest.testSample0\(\)](#). The debugger will stop every time a SQL statement is executed, and you should be able to loop up the call stack to which component is executing the query.

I expect that you will see the following phases in the execution:

- One or two SQL queries will be executed as the `schema.xml` file is read (validating calculated members and named sets, resolving default members of hierarchies, and such)
- A few SQL queries will be executed to resolve members as the query is parsed. (For example, if a query uses `[Store].[USA].[CA]`, it will look all members of the `[Store Nation]` level, then look up all children of the `[USA]` member.)
- When the query is executed, the axes (slicer, columns, rows) are executed first. Expect to see more queries on dimension tables when expressions like `[Product].children` are evaluated.
- Once the axes are populated, the cells are evaluated. Rather than executing a SQL query per cell, Mondrian makes a pass over all cells building a list of cells which are not in the cache. Then it builds and executes a SQL query to fetch all of those cells. If it didn't manage to fetch all cell values, it will repeat this step until it does.

Remember that the purpose of these queries is to populate cache. There are two caches. The dimension cache which maps a member to its children, e.g.

```
[Store].[All Stores] □ { [Store].[USA], [Store].[Canada],  
[Store].[Mexico]}
```

The aggregation cache maps a tuple a measure value, e.g.

```
([Store].[USA], [Gender].[F], [Measures].[Unit Sales]) □ 123,456
```

Once the cache has been populated, the query won't be executed again. That's why I recommend that you restart the process each time you run this in the debugger.

Logging Levels and Information

Some of the Mondrian classes are instrumented with Apache Log4J Loggers. For some of these classes there are certain logging setting that provide information for not just the code developer but also for someone setting up a Mondrian installation. The following is a list of some of those log setting and the associated information.

Category	Level	Description
<code>mondrian.rolap.aggmatcher.AggregateTableManager</code>	INFO	A list of the RolapStar fact table names (aliases) and for each fact table, a list of all of its associated aggregate tables.
<code>mondrian.rolap.aggmatcher.AggregateTableManager</code>	DEBUG	A verbose output of all RolapStar fact tables, their measures columns, and dimension tables and columns, along with all of each fact table's aggregate tables, columns and dimension tables.
<code>mondrian.rolap.aggmatcher.DefaultDef</code>	DEBUG	For each candidate aggregate table, the Matcher regular expressions for matching: table name and the fact count, foreign key, level and measure columns. Helpful in finding out why an aggregate table was not recognized.
<code>mondrian.rolap.agg.AggregationManager</code>	DEBUG	For each aggregate Sql query, if an aggregate table can be used to fulfill the query, which aggregate it was along with bitKeys and column names.
<code>mondrian.rolap.RolapUtil</code>	DEBUG	Prints out all Sql statements and their execution time. If one set the Mondrian property, <code>mondrian.rolap.generate.formatted.sql</code> to true, then the Sql is pretty printed (very nice).
<code>mondrian.rolap.RolapConnection</code>	DEBUG	Prints out each MDX query prior to its execution. (No pretty printing, sigh.)
<code>mondrian.rolap.RolapSchema</code>	DEBUG	Prints out each Rolap Schema as it is being loaded.

There are more classes with logging, but their logging is at a lower, more detailed level of more use to code developers.

Log levels can be set in either a `log4j.properties` file or `log4j.xml` file. You have to make sure you tell Mondrian which one to use. For the `log4j.properties`, entries might look like:

```
log4j.category.mondrian.rolap.RolapConnection=DEBUG
log4j.category.mondrian.rolap.RolapUtil=DEBUG
```

while for the `log4.xml`:

```

<category name="mondrian.rolap.RolapConnection">
  <priority value="DEBUG"/>
</category>
<category name="mondrian.rolap.RolapUtil">
  <priority value="DEBUG"/>
</category>

```

Default aggregate table recognition rules

The default Mondrian rules for recognizing aggregate tables are specified by creating an instance of the rule schema found in the file:

MONDRIAN_HOME/src/main/rolap/aggmatcher/DefaultRulesSchema.xml. The instance of this schema that is built into the `mondrian.jar` after a build is in the same directory, `MONDRIAN_HOME/src/main/rolap/aggmatcher/DefaultRules.xml`.

There are six different default rules that are used to match and map a candidate aggregate table: table name, ignore column, fact count column, foreign key column, level column and measure column. All of these rules are defined by creating an instance of the `DefaultRulesSchema.xml` grammar. The `DefaultRulesSchema.xml` instance, the `DefaultRules.xml` file mentioned above, that by default is built as part of the `mondrian.jar` does not contain an ignore column rule. This grammar has base/supporting classes that are common to the above rules. In XOM terms, these are classes and super classes of the rule elements.

The first XOM class dealing with matching is the `CaseMatcher` class. This has an attribute "charcase" that takes the legal values of

```

"ignore" (default)
"exact"
"upper"
"lower"

```

When the value of the attribute is "ignore", then the regular expression formed by an element extending the `CaseMatcher` class will be case independent for both any parameters used to instantiate the regular expression template as well as for the text in the post-instantiated regular expression. On the other hand, when the "charcase" attribute take any of the other three values, it is only the parameter values themselves that are "exact", unchanged, "lower", converted to lower case, or "upper", converted to upper case.

The class `NameMatcher` extends the `CaseMatcher` class. This class has pre-template and post-template attributes whose default values is the empty string. These attributes are prepended/appended to a parameter to generate a regular expression. As an example, the `TableMatcher` element extends `NameMatcher` class. The parameter in this case is the fact table name and the regular expression would be:

```
pre-template-attribute${fact_table_name}post-template-attribute
```

For Mondrian, the builtin rule has the pre template value "agg_+_" and the post template attribute value is the default so the regular expression becomes:

```
agg_+_${fact_table_name}
```

Also, the `NameMatcher` has an attribute called `basename` which is optional. If set, then its value must be a regular expression with a single capture group. A capture group is a regular expression component surrounded by "(" and ")". As an example, "(.*)" is a capture group and if this was the total regular expression, then it would match anything and the single capture would match the same. On the other hand if the total regular expression was "RF_(.*)_TBL", then a name such as "RF_SHIPMENTS_TBL" would match the regular expression while the capture group would be "SHIPMENTS". Now, if the `basename` attribute is defined, then it is applied to each fact table name allowing one to strip away information and get to the "base" name. This might be needed because a DBA might prepend or append a tag to all of your fact table names and the DBA might wish to have a different tag prepend or append to all of your aggregate table names (RF_SHIPMENTS_TBL as the fact table and RA_SHIPMENTS_AGG_14 as an example aggregate name (the DBA prepended the "RA_" and you appended the "_AGG_14")).

Both the `FactCountMatch` and `ForeignKeyMatch` elements also extend the `NameMatcher` class. In these cases, the builtin Mondrian rule has no pre or post template attribute values, no regular expression, The `FactCountMatch` takes no other parameter from the fact table (the fact table does not have a fact count column) rather it takes a fact count attribute with default value "fact_count", and this is used to create the regular expression. For the `ForeignKeyMatch` matcher, its the fact table's foreign key that is used as the regular expression.

The `ignore`, `asdf` level and `measure` column matching elements have one or more `Regex` child elements. These allow for specifying multiple possible matches (if any match, then its a match). The `IgnoreMap`, `LevelMap` and `MeasureMap` elements extend the `RegexMapper` which holds an array of `Regex` elements. The `Regex` element extends `CaseMatcher` It has two attributes, `space` with default value '_' which says how space characters should be mapped, and `dot` with default value '.' which says how '.' characters should be mapped. If a name were the string "Unit Sales.Case" then (with the default values for the `space` and `dot` attributes and with `CaseMatcher` mapping to lower case) this would become "unit_sales_case".

The `IgnoreMap` element has NO template parameter names. Each `Regex` value is simply a regular expression. As an example (Mondrian by default does not include an `IgnoreMap` by default), a regular expression that matches all aggregate table columns then end with '_DO_NOT_USE' would be:

```
.*_DO_NOT_USE
```

One might want to use an `IgnoreMap` element to filter out aggregate columns if, for example, the aggregate table is a materialized view, since with each "normal" column of such a materialized view there is an associated support column used by the database which has no significance to Mondrian. In the process of recognizing aggregate tables, Mondrian logs a warning message for each column whose use can not be determined. Materialized views have so many of these support columns that if, in fact, there was a column whose use was desired but was not recognized (for instance, the column name is misspelt) all of the materialized view column warning message mask the one warning message that one really needs to see.

The `IgnoreMap` regular expressions are applied before any of the other column matching actions. If one sets the `IgnoreMap` regular expression to, for example,

```
.*
```

then all columns are marked as "ignore" and there are no other columns left to match anything else. One must be very careful when choosing `IgnoreMap` regular expressions not just for your current columns but for columns that might be created in the future. Its best to document this usage in your organization.

The following is what the element might look like in a `DefaultRules.xml` file:

```
<IgnoreMap id="ixx" >
  <Regex id="physical" charcase="ignore">
    .*_DO_NOT_USE
  </Regex>
</IgnoreMap>
```

The `LevelMap` element has the four template parameter names (hardcoded):

```
hierarchy_name
level_name
level_column_name
usage_prefix
```

These are names that can be used in creating template regular expressions. The builtin Mondrian default rules for level matching defines three `Regex` child elements for the `LevelMap` element. These define the template regular expressions:

```
${hierarchy_name}_${level_name}
${hierarchy_name}_${level_column_name}
${usage_prefix}${level_column_name}
${level_column_name}
```

Mondrian while attempting to match a candidate aggregate table against a particular fact table, iterates through the fact table's cube's hierarchy name, level name and level column names looking for matches.

The `MeasureMap` element has the three template parameter names (hardcoded):

```
measure_name
measure_column_name
aggregate_name
```

which can appear in template regular expressions. The builtin Mondrian default rules for measure matching defines three `Regex` child elements for the `MeasureMap` element. These are

```
${measure_name}
${measure_column_name}
${measure_column_name}_${aggregate_name}
```

and Mondrian attempts to match a candidate aggregate table's column names against these as it iterates over a fact table's measures.

A grouping of `FactCountMatch` , `ForeignKeyMatch` , `TableMatcher` , `LevelMap` , and `MeasureMap` make up a `AggRule` element, a rule set. Each `AggRule` has a `tag` attribute

which is a unique identifier for the rule. There can be multiple `AggRule` elements in the outer `AggRules` element. Each `AggRule` having its own `tag` attribute. When Mondrian runs, it selects (via the `mondrian.rolap.aggregates.rule.tag` property) which rule set to use.

One last wrinkle, within a `AggRule` the `FactCountMatch`, `ForeignKeyMatch`, `TableMatcher`, `LevelMap`, and `MeasureMap` child elements can be either defined explicitly within the `AggRule` element or by reference `FactCountMatchRef`, `ForeignKeyMatchRef`, `TableMatcherRef`, `LevelMapRef`, and `MeasureMapRef`. The references are defined as child elements of the top level `AggRules` element. With references the same rule element can be used by more than one `AggRule` (code reuse).

Below is an example of a default rule set with rather different matching rules.

```
<AggRules tag="your_mamas_dot_com">
  <AggRule tag="default" >
    <FactCountMatch id="fca" factCountName="FACT_TABLE_COUNT"
      charcase="exact" />
    <ForeignKeyMatch id="fka" pretemplate="agg_" />
    <TableMatch id="ta" pretemplate="agg_" posttemplate="_.+"/>
    <LevelMap id="lxx" >
      <Regex id="logical" charcase="ignore" space="_" dot="_">
        ${hierarchy_name}_${level_name}
      </Regex>
      <Regex id="mixed" charcase="ignore" >
        ${hierarchy_name}_${level_name}_${level_column_name}
      </Regex>
      <Regex id="mixed" charcase="ignore" >
        ${hierarchy_name}_${level_column_name}
      </Regex>
      <Regex id="usage" charcase="exact" >
        ${usage_prefix}${level_column_name}
      </Regex>
      <Regex id="physical" charcase="exact" >
        ${level_column_name}_.+
      </Regex>
    </LevelMap>
    <MeasureMap id="mxx" >
      <Regex id="one" charcase="lower" >
        ${measure_name}(_${measure_column_name})(_${aggregate_name})??
      </Regex>
      <Regex id="two" charcase="exact" >
        ${measure_column_name}(_${aggregate_name})?
      </Regex>
    </MeasureMap>
  </AggRule>
</AggRules>
```

First, all fact count columns must be called `FACT_TABLE_COUNT` exactly, no ignoring case. Next, foreign key columns match the regular expression

```
agg_${foreign_key_name}
```

that is, the fact table foreign key column name with "agg_" prepended such as `agg_time_id` .
The aggregate table names match the regular expression

```
agg_${fact_table_name}_.+
```

For the FoodMart `sales_fact_1997` fact table, an aggregate could be named,

```
agg_sales_fact_1997_01  
agg_sales_fact_1997_lost_time_id  
agg_sales_fact_1997_top
```

If the hierarchy, level and level column names were:

```
hierarchy_name="Sales Location"  
level_name="State"  
level_column_name="state_location"  
usage_prefix=null
```

then the following aggregate table column names would be recognizing as level column names:

```
SALES_LOCATION_STATE  
Sales_Location_State_state_location  
state_location_level.
```

If in the schema file the DimensionUsage for the hierarchy had a usagePrefix attribute,

```
usage_prefix="foo_"
```

then with the above level and level column names and usage_prefix the following aggregate table column names would be recognizing as level column names:

```
SALES_LOCATION_STATE  
Sales_Location_State_state_location  
state_location_level.  
foo_state_location.
```

In the case of matching measure columns, if the measure template parameters have the following values:

```
measure_name="Unit Sales"  
measure_column_name="m1"  
aggregate_name="Avg"
```

then possible aggregate columns that could match are:

```
unit_sales_m1  
unit_sales_m1_avg  
m1  
m1_avg
```

The intent of the above example default rule set is not that they are necessarily realistic or usable, rather, it just shows what is possible.

Snowflakes and the DimensionUsage level attribute

Mondrian supports dimensions with all of their levels lumped into a single table (with all the duplication of data that that entails), but also snowflakes. A snowflake dimension is one where the fact table joins to one table (generally the lowest) and that table then joins to a table representing the next highest level, and so on until the top level's table is reached. For each level there is a separate table.

As an example snowflake, below is a set of Time levels and four possible join element blocks, relationships between the tables making up the Time dimension. (In a schema file, the levels must appear after the joins.)

```
<Level name="Calendar Year" table="TimeYear" column="YEAR_SID"
  nameColumn="YEAR_NAME" levelType="TimeYears" uniqueMembers="true"/>
<Level name="Quarter" table="TimeQtr" column="QTR_SID"
  nameColumn="QTR_NAME" levelType="TimeQuarters" uniqueMembers="true"/>
<Level name="Month" table="TimeMonth" column="MONTH_SID"
  nameColumn="MONTH_ONLY_NAME" levelType="TimeMonths"
uniqueMembers="false"/>
<Level name="Day" table="TimeDay" column="DAY_SID"
nameColumn="DAY_NAME"
  levelType="TimeDays" uniqueMembers="true"/>

<Join leftAlias="TimeYear" leftKey="YEAR_SID"
  rightAlias="TimeQtr" rightKey="YEAR_SID" >
  <Table name="RD_PERIOD_YEAR" alias="TimeYear" />
  <Join leftAlias="TimeQtr" leftKey="QTR_SID"
    rightAlias="TimeMonth" rightKey="QTR_SID" >
    <Table name="RD_PERIOD_QTR" alias="TimeQtr" />
    <Join leftAlias="TimeMonth" leftKey="MONTH_SID"
      rightAlias="TimeDay" rightKey="MONTH_SID" >
      <Table name="RD_PERIOD_MONTH" alias="TimeMonth" />
      <Table name="RD_PERIOD_DAY" alias="TimeDay" />
    </Join>
  </Join>
</Join>

<Join leftAlias="TimeQtr" leftKey="YEAR_SID"
  rightAlias="TimeYear" rightKey="YEAR_SID" >
  <Join leftAlias="TimeMonth" leftKey="QTR_SID"
    rightAlias="TimeQtr" rightKey="QTR_SID" >
    <Join leftAlias="TimeDay" leftKey="MONTH_SID"
      rightAlias="TimeMonth" rightKey="MONTH_SID" >
      <Table name="RD_PERIOD_DAY" alias="TimeDay" />
      <Table name="RD_PERIOD_MONTH" alias="TimeMonth" />
    </Join>
    <Table name="RD_PERIOD_QTR" alias="TimeQtr" />
  </Join>
  <Table name="RD_PERIOD_YEAR" alias="TimeYear" />
</Join>
```

```

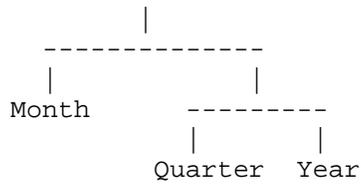
<Join leftAlias="TimeMonth" leftKey="MONTH_SID"
      rightAlias="TimeDay" rightKey="MONTH_SID" >
  <Join leftAlias="TimeQtr" leftKey="QTR_SID"
        rightAlias="TimeMonth" rightKey="QTR_SID" >
    <Join leftAlias="TimeYear" leftKey="YEAR_SID"
          rightAlias="TimeQtr" rightKey="YEAR_SID" >
      <Table name="RD_PERIOD_YEAR" alias="TimeYear" />
      <Table name="RD_PERIOD_QTR" alias="TimeQtr" />
    </Join>
    <Table name="RD_PERIOD_MONTH" alias="TimeMonth" />
  </Join>
  <Table name="RD_PERIOD_DAY" alias="TimeDay" />
</Join>

<Join leftAlias="TimeDay" leftKey="MONTH_SID"
      rightAlias="TimeMonth" rightKey="MONTH_SID" >
  <Table name="RD_PERIOD_DAY" alias="TimeDay" />
  <Join leftAlias="TimeMonth" leftKey="QTR_SID"
        rightAlias="TimeQtr" rightKey="QTR_SID" >
    <Table name="RD_PERIOD_MONTH" alias="TimeMonth" />
    <Join leftAlias="TimeQtr" leftKey="YEAR_SID"
          rightAlias="TimeYear" rightKey="YEAR_SID" >
      <Table name="RD_PERIOD_QTR" alias="TimeQtr" />
      <Table name="RD_PERIOD_YEAR" alias="TimeYear" />
    </Join>
  </Join>
</Join>

```


Mondrian reorders these join blocks into the canonical form and uses that to build subtables in the RolapStar.

In addition, if a cube had a `DimensionUsage` of this Time dimension with, for example, its `level` attribute set to `Month`, then the above tree is pruned



and the pruned tree is what is used to create the subtables in the RolapStar. Of course, the fact table must, in this case, have a `MONTH_SID` foreign key.

Note that the `Level` element's `table` attribute **MUST** use the table alias and **NOT** the table name.

Appendix A – MDX Function List

These are the functions implemented in the current Mondrian release.

Name	Description
\$AggregateChildren	Equivalent to 'Aggregate(<Hierarchy>.CurrentMember.Children); for internal use.
	Syntax
	<Numeric Expression> \$AggregateChildren(<Hierarchy>)
\$Cache	Evaluates and returns its sole argument, applying statement-level caching
	Syntax
	\$Cache(<<Exp>>)
()	Syntax
*	Multiplies two numbers.
	Syntax
	<Numeric Expression> * <Numeric Expression>
*	Returns the cross product of two sets.
	Syntax
	<Set> * <Set> <Member> * <Set> <Set> * <Member> <Member> * <Member>
+	Adds two numbers.
	Syntax
	<Numeric Expression> + <Numeric Expression>
-	Subtracts two numbers.
	Syntax
	<Numeric Expression> - <Numeric Expression>
-	Returns the negative of a number.
	Syntax
	- <Numeric Expression>

/

Divides two numbers.

Syntax

<Numeric Expression> / <Numeric Expression>

:

Infix colon operator returns the set of members between a given pair of members.

Syntax

<Member> : <Member>

<

Returns whether an expression is less than another.

Syntax

<Numeric Expression> < <Numeric Expression>

<

Returns whether an expression is less than another.

Syntax

<String> < <String>

<=

Returns whether an expression is less than or equal to another.

Syntax

<Numeric Expression> <= <Numeric Expression>

<=

Returns whether an expression is less than or equal to another.

Syntax

<String> <= <String>

<>

Returns whether two expressions are not equal.

Syntax

<Numeric Expression> <> <Numeric Expression>

<>

Returns whether two expressions are not equal.

Syntax

<String> <> <String>

=

Returns whether two expressions are equal.

Syntax

<Numeric Expression> = <Numeric Expression>

=	Returns whether two expressions are equal.
	Syntax
	<String> = <String>
>	Returns whether an expression is greater than another.
	Syntax
	<Numeric Expression> > <Numeric Expression>
>	Returns whether an expression is greater than another.
	Syntax
	<String> > <String>
>=	Returns whether an expression is greater than or equal to another.
	Syntax
	<Numeric Expression> >= <Numeric Expression>
>=	Returns whether an expression is greater than or equal to another.
	Syntax
	<String> >= <String>
AND	Returns the conjunction of two conditions.
	Syntax
	<Logical Expression> AND <Logical Expression>
AddCalculatedMembers	Adds calculated members to a set.
	Syntax
	<Set> AddCalculatedMembers(<Set>)
Aggregate	Returns a calculated value using the appropriate aggregate function, based on the context of the query.
	Syntax
	<Numeric Expression> Aggregate(<Set>) <Numeric Expression> Aggregate(<Set>, <Numeric Expression>)
AllMembers	Returns a set that contains all members, including calculated members, of the specified dimension.

Syntax

AllMembers <Dimension>.AllMembers
Returns a set that contains all members, including calculated members, of the specified hierarchy.

Syntax

AllMembers <Hierarchy>.AllMembers
Returns a set that contains all members, including calculated members, of the specified level.

Syntax

Ancestor <Level>.AllMembers
Returns the ancestor of a member at a specified level.

Syntax

Ascendants <Member> Ancestor(<Member>, <Level>)
<Member> Ancestor(<Member>, <Numeric Expression>)
Returns the set of the ascendants of a specified member.

Syntax

Avg <Set> Ascendants(<Member>)
Returns the average value of a numeric expression evaluated over a set.

Syntax

BottomCount <Numeric Expression> Avg(<Set>)
<Numeric Expression> Avg(<Set>, <Numeric Expression>)
Returns a specified number of items from the bottom of a set, optionally ordering the set first.

Syntax

BottomPercent <Set> BottomCount(<Set>, <Numeric Expression>, <Numeric Expression>)
<Set> BottomCount(<Set>, <Numeric Expression>)
Sorts a set and returns the bottom N elements whose cumulative total is at least a specified percentage.

Syntax

`<Set> BottomPercent(<Set>, <Numeric Expression>, <Numeric Expression>)`

BottomSum
Sorts a set and returns the bottom N elements whose cumulative total is at least a specified value.

Syntax

`<Set> BottomSum(<Set>, <Numeric Expression>, <Numeric Expression>)`

CalculatedChild
Returns an existing calculated child member with name `<String>` from the specified `<Member>`.

Syntax

`<Member> <Member>.CalculatedChild(<String>)`

Caption
Returns the caption of a dimension.

Syntax

`<Dimension>.Caption`

Caption
Returns the caption of a hierarchy.

Syntax

`<Hierarchy>.Caption`

Caption
Returns the caption of a level.

Syntax

`<Level>.Caption`

Caption
Returns the caption of a member.

Syntax

`<Member>.Caption`

Cast
Converts values to another type

Syntax

`Cast(<Expression> AS <Type>)`

Children
Returns the children of a member.

Syntax

`<Member>.Children`

ClosingPeriod	<p>Returns the last descendant of a member at a level.</p> <p>Syntax</p> <p><Member> ClosingPeriod() <Member> ClosingPeriod(<Level>) <Member> ClosingPeriod(<Level>, <Member>) <Member> ClosingPeriod(<Member>)</p>
CoalesceEmpty	<p>Coalesces an empty cell value to a different value. All of the expressions must be of the same type (number or string).</p> <p>Syntax</p> <p>CoalesceEmpty(<Value Expression>[, <Value Expression>...])</p>
Correlation	<p>Returns the correlation of two series evaluated over a set.</p> <p>Syntax</p> <p><Numeric Expression> Correlation(<Set>, <Numeric Expression>) <Numeric Expression> Correlation(<Set>, <Numeric Expression>, <Numeric Expression>)</p>
Count	<p>Returns the number of tuples in a set, empty cells included unless the optional EXCLUDEEMPTY flag is used.</p> <p>Syntax</p> <p><Numeric Expression> Count(<Set>) <Numeric Expression> Count(<Set>, <Symbol>)</p>
Count	<p>Returns the number of tuples in a set including empty cells.</p> <p>Syntax</p> <p><Set>.Count</p>
Cousin	<p>Returns the member with the same relative position under <ancestor member> as the member specified.</p> <p>Syntax</p> <p><Member> Cousin(<Member>, <Member>)</p>
Covariance	<p>Returns the covariance of two series evaluated over a set (biased).</p>

Syntax

CovarianceN
<Numeric Expression> Covariance(<Set>, <Numeric Expression>)
<Numeric Expression> Covariance(<Set>, <Numeric Expression>, <Numeric Expression>)
Returns the covariance of two series evaluated over a set (unbiased).

Syntax

Crossjoin
<Numeric Expression> CovarianceN(<Set>, <Numeric Expression>)
<Numeric Expression> CovarianceN(<Set>, <Numeric Expression>, <Numeric Expression>)
Returns the cross product of two sets.

Syntax

CurrentDateMember
<Set> Crossjoin(<Set>, <Set>)
Returns the member within the specified dimension corresponding to the current date, in the format specified by the format parameter.

Syntax

CurrentDateMember
<Member> CurrentDateMember(<Hierarchy>, <String>)
Returns the closest or exact member within the specified dimension corresponding to the current date, in the format specified by the format parameter.

Syntax

CurrentDateString
<Member> CurrentDateMember(<Hierarchy>, <String>, <Symbol>)
Returns the current date formatted as specified by the format parameter.

Syntax

CurrentMember
<String> CurrentDateString(<String>)
Returns the current member along a dimension during an iteration.

Syntax

<Dimension>.CurrentMember

CurrentMember Returns the current member along a hierarchy during an iteration.

Syntax

<Hierarchy>.CurrentMember

DataMember Returns the system-generated data member that is associated with a nonleaf member of a dimension.

Syntax

<Member>.DataMember

DefaultMember Returns the default member of a dimension.

Syntax

<Dimension>.DefaultMember

DefaultMember Returns the default member of a hierarchy.

Syntax

<Hierarchy>.DefaultMember

Descendants Returns the set of descendants of a member at a specified level, optionally including or excluding descendants in other levels.

Syntax

<Set> Descendants(<Member>)
<Set> Descendants(<Member>, <Level>)
<Set> Descendants(<Member>, <Level>, <Symbol>)
<Set> Descendants(<Member>, <Numeric
Expression>)
<Set> Descendants(<Member>, <Numeric
Expression>, <Symbol>)

Dimension Returns the dimension that contains a specified hierarchy.

Syntax

<Dimension>.Dimension

Dimension Returns the dimension that contains a specified hierarchy.

Syntax

<Hierarchy>.Dimension

Dimension Returns the dimension that contains a specified level.

Syntax

Dimension
<Level>.Dimension
Returns the dimension that contains a specified member.

Syntax

Dimensions
<Member>.Dimension
Returns the dimension whose zero-based position within the cube is specified by a numeric expression.

Syntax

Dimensions
<Dimension> Dimensions(<Numeric Expression>)
Returns the dimension whose name is specified by a string.

Syntax

Distinct
<Dimension> Dimensions(<String>)
Eliminates duplicate tuples from a set.

Syntax

DrilldownLevel
<Set> Distinct(<Set>)
Drills down the members of a set, at a specified level, to one level below. Alternatively, drills down on a specified dimension in the set.

Syntax

DrilldownMember
<Set> DrilldownLevel(<Set>)
<Set> DrilldownLevel(<Set>, <Level>)
Drills down the members in a set that are present in a second specified set.

Syntax

Except
<Set> DrilldownMember(<Set>, <Set>)
<Set> DrilldownMember(<Set>, <Set>, <Symbol>)
Finds the difference between two sets, optionally retaining duplicates.

Syntax

<Set> Except(<Set>, <Set>)
<Set> Except(<Set>, <Set>, <Symbol>)

Filter Returns the set resulting from filtering a set based on a search condition.

Syntax

<Set> Filter(<Set>, <Logical Expression>)

FirstChild Returns the first child of a member.

Syntax

<Member>.FirstChild

FirstQ Returns the 1st quartile value of a numeric expression evaluated over a set.

Syntax

<Numeric Expression> FirstQ(<Set>)
<Numeric Expression> FirstQ(<Set>, <Numeric Expression>)

FirstSibling Returns the first child of the parent of a member.

Syntax

<Member>.FirstSibling

Format Formats a number to string.

Syntax

<String> Format(<Member>, <String>)
<String> Format(<Numeric Expression>, <String>)

Generate Applies a set to each member of another set and joins the resulting sets by union.

Syntax

<Set> Generate(<Set>, <Set>)
<Set> Generate(<Set>, <Set>, <Symbol>)

Head Returns the first specified number of elements in a set.

Syntax

<Set> Head(<Set>)
<Set> Head(<Set>, <Numeric Expression>)

Hierarchize Orders the members of a set in a hierarchy.

Syntax

<Set> Hierarchize(<Set>)
<Set> Hierarchize(<Set>, <Symbol>)

Hierarchy	Returns a level's hierarchy.
	Syntax
	<Level>.Hierarchy
Hierarchy	Returns a member's hierarchy.
	Syntax
	<Member>.Hierarchy
IIf	Returns one of two numeric values determined by a logical test.
	Syntax
	<Numeric Expression> IIf(<Logical Expression>, <Numeric Expression>, <Numeric Expression>)
IIf	Returns one of two string values determined by a logical test.
	Syntax
	<String> IIf(<Logical Expression>, <String>, <String>)
IS	Returns whether an object is null
	Syntax
	<Member> IS <Null> <Level> IS <Null> <Hierarchy> IS <Null> <Dimension> IS <Null>
IN	Returns whether a member is contained in a set.
	Syntax
	<Member> IN <Set>
IS	Returns whether two objects are the same
	Syntax
	<Member> IS <Member> <Level> IS <Level> <Hierarchy> IS <Hierarchy> <Dimension> IS <Dimension> <Tuple> IS <Tuple>
Intersect	Returns the intersection of two input sets, optionally retaining duplicates.

Syntax

`<Set> Intersect(<Set>, <Set>, <Symbol>)`
`<Set> Intersect(<Set>, <Set>)`
IsEmpty
Determines if an expression evaluates to the empty cell value.

Syntax

`<Logical Expression> IsEmpty(<String>)`
`<Logical Expression> IsEmpty(<Numeric Expression>)`
Item
Returns a member from the tuple specified in `<Tuple>`. The member to be returned is specified by the zero-based position of the member in the set in `<Index>`.

Syntax

`<Member> <Tuple>.Item(<Numeric Expression>)`
Item
Returns a tuple from the set specified in `<Set>`. The tuple to be returned is specified by the zero-based position of the tuple in the set in `<Index>`.

Syntax

`<Member> <Set>.Item(<Numeric Expression>)`
Item
Returns a tuple from the set specified in `<Set>`. The tuple to be returned is specified by the member name (or names) in `<String>`.

Syntax

`<Set>.Item(<String> [, ...])`
Lag
Returns a member further along the specified member's dimension.

Syntax

`<Member> <Member>.Lag(<Numeric Expression>)`
LastChild
Returns the last child of a member.

Syntax

`<Member>.LastChild`
LastPeriods
Returns a set of members prior to and including a specified member.

Syntax

LastSibling
<Set> LastPeriods(<Numeric Expression>)
<Set> LastPeriods(<Numeric Expression>, <Member>)
Returns the last child of the parent of a member.

Syntax

Lead
<Member>.LastSibling
Returns a member further along the specified member's dimension.

Syntax

Level
<Member> <Member>.Lead(<Numeric Expression>)
Returns a member's level.

Syntax

Levels
<Member>.Level
Returns the level whose position in a hierarchy is specified by a numeric expression.

Syntax

Levels
<Level> <Hierarchy>.Levels(<Numeric Expression>)
Returns the level whose name is specified by a string expression.

Syntax

LinRegIntercept
<Level> Levels(<String>)
Calculates the linear regression of a set and returns the value of b in the regression line $y = ax + b$.

Syntax

LinRegPoint
<Numeric Expression> LinRegIntercept(<Set>, <Numeric Expression>)
<Numeric Expression> LinRegIntercept(<Set>, <Numeric Expression>, <Numeric Expression>)
Calculates the linear regression of a set and returns the value of y in the regression line $y = ax + b$.

Syntax

<Numeric Expression> LinRegPoint(<Numeric Expression>, <Set>, <Numeric Expression>)
<Numeric Expression> LinRegPoint(<Numeric Expression>, <Set>, <Numeric Expression>, <Numeric Expression>)

LinRegR2 Calculates the linear regression of a set and returns R2 (the coefficient of determination).

Syntax

<Numeric Expression> LinRegR2(<Set>, <Numeric Expression>)
<Numeric Expression> LinRegR2(<Set>, <Numeric Expression>, <Numeric Expression>)

LinRegSlope Calculates the linear regression of a set and returns the value of a in the regression line $y = ax + b$.

Syntax

<Numeric Expression> LinRegSlope(<Set>, <Numeric Expression>)
<Numeric Expression> LinRegSlope(<Set>, <Numeric Expression>, <Numeric Expression>)

LinRegVariance Calculates the linear regression of a set and returns the variance associated with the regression line $y = ax + b$.

Syntax

<Numeric Expression> LinRegVariance(<Set>, <Numeric Expression>)
<Numeric Expression> LinRegVariance(<Set>, <Numeric Expression>, <Numeric Expression>)

Max Returns the maximum value of a numeric expression evaluated over a set.

Syntax

<Numeric Expression> Max(<Set>)
<Numeric Expression> Max(<Set>, <Numeric Expression>)

MATCHES Returns whether an expression matches a regular expression.

Syntax

<String> MATCHES <String>

Median Returns the median value of a numeric expression evaluated over a set.

Syntax

<Numeric Expression> Median(<Set>)
<Numeric Expression> Median(<Set>, <Numeric Expression>)

Members Returns the set of members in a dimension.

	Syntax
Members	<Dimension>.Members Returns the set of members in a hierarchy.
	Syntax
Members	<Hierarchy>.Members Returns the set of members in a level.
	Syntax
Members	<Level>.Members Returns the member whose name is specified by a string expression.
	Syntax
Min	<Member> Members(<String>) Returns the minimum value of a numeric expression evaluated over a set.
	Syntax
Mtd	<Numeric Expression> Min(<Set>) <Numeric Expression> Min(<Set>, <Numeric Expression>) A shortcut function for the PeriodsToDate function that specifies the level to be Month.
	Syntax
NOT	<Set> Mtd() <Set> Mtd(<Member>) Returns the negation of a condition.
	Syntax
Name	NOT <Logical Expression> Returns the name of a dimension.
	Syntax
NOT IN	<Dimension>.Name Returns whether a member is not contained in a set.

	Syntax
	<Member> NOT IN <Set>
NOT MATCHES	Returns whether an expression does not match a regular expression.
	Syntax
	<String> NOT MATCHES <String>
Name	Returns the name of a hierarchy.
	Syntax
	<Hierarchy>.Name
Name	Returns the name of a level.
	Syntax
	<Level>.Name
Name	Returns the name of a member.
	Syntax
	<Member>.Name
NextMember	Returns the next member in the level that contains a specified member.
	Syntax
	<Member>.NextMember
NonEmptyCrossJoin	Returns the cross product of two sets, excluding empty tuples and tuples without associated fact table data.
	Syntax
	<Set> NonEmptyCrossJoin(<Set>, <Set>)
OR	Returns the disjunction of two conditions.
	Syntax
	<Logical Expression> OR <Logical Expression>
OpeningPeriod	Returns the first descendant of a member at a level.
	Syntax
	<Member> OpeningPeriod()
	<Member> OpeningPeriod(<Level>)

Order	<p><Member> OpeningPeriod(<Level>, <Member>) Arranges members of a set, optionally preserving or breaking the hierarchy.</p> <p>Syntax</p> <p><Set> Order(<Set>, <Value>, <Symbol>) <Set> Order(<Set>, <Value>)</p>
Ordinal	<p>Returns the zero-based ordinal value associated with a level.</p> <p>Syntax</p> <p><Level>.Ordinal</p>
ParallelPeriod	<p>Returns a member from a prior period in the same relative position as a specified member.</p> <p>Syntax</p> <p><Member> ParallelPeriod() <Member> ParallelPeriod(<Level>) <Member> ParallelPeriod(<Level>, <Numeric Expression>) <Member> ParallelPeriod(<Level>, <Numeric Expression>, <Member>)</p>
ParamRef	<p>Returns the current value of this parameter. If it is null, returns the default value.</p> <p>Syntax</p> <p><Value> ParamRef(<String>)</p>
Parameter	<p>Returns default value of parameter.</p> <p>Syntax</p> <p><String> Parameter(<String>, <Symbol>, <String>, <String>) <String> Parameter(<String>, <Symbol>, <String>) <Numeric Expression> Parameter(<String>, <Symbol>, <Numeric Expression>, <String>) <Numeric Expression> Parameter(<String>, <Symbol>, <Numeric Expression>) <Member> Parameter(<String>, <Hierarchy>, <Member>, <String>) <Member> Parameter(<String>, <Hierarchy>, <Member>)</p>
Parent	<p>Returns the parent of a member.</p> <p>Syntax</p>

PeriodsToDate	<p><Member>.Parent</p> <p>Returns a set of periods (members) from a specified level starting with the first period and ending with a specified member.</p> <p>Syntax</p> <p><Set> PeriodsToDate() <Set> PeriodsToDate(<Level>) <Set> PeriodsToDate(<Level>, <Member>)</p>
PrevMember	<p>Returns the previous member in the level that contains a specified member.</p> <p>Syntax</p> <p><Member>.PrevMember</p>
Properties	<p>Returns the value of a member property.</p> <p>Syntax</p> <p><Member>.Properties(<String Expression>)</p>
Qtd	<p>A shortcut function for the PeriodsToDate function that specifies the level to be Quarter.</p> <p>Syntax</p> <p><Set> Qtd() <Set> Qtd(<Member>)</p>
Rank	<p>Returns the one-based rank of a tuple in a set.</p> <p>Syntax</p> <p><Integer> Rank(<Tuple>, <Set>) <Integer> Rank(<Tuple>, <Set>, <Numeric Expression>) <Integer> Rank(<Member>, <Set>) <Integer> Rank(<Member>, <Set>, <Numeric Expression>)</p>
SetToStr	<p>Constructs a string from a set.</p> <p>Syntax</p> <p><String> SetToStr(<Set>)</p>
Siblings	<p>Returns the siblings of a specified member, including the member itself.</p> <p>Syntax</p> <p><Member>.Siblings</p>

Stddev	Alias for Stdev.
	Syntax
	<Numeric Expression> Stddev(<Set>) <Numeric Expression> Stddev(<Set>, <Numeric Expression>)
StddevP	Alias for StdevP.
	Syntax
	<Numeric Expression> StddevP(<Set>) <Numeric Expression> StddevP(<Set>, <Numeric Expression>)
Stdev	Returns the standard deviation of a numeric expression evaluated over a set (unbiased).
	Syntax
	<Numeric Expression> Stdev(<Set>) <Numeric Expression> Stdev(<Set>, <Numeric Expression>)
StdevP	Returns the standard deviation of a numeric expression evaluated over a set (biased).
	Syntax
	<Numeric Expression> StdevP(<Set>) <Numeric Expression> StdevP(<Set>, <Numeric Expression>)
StrToMember	Returns a member from a unique name String in MDX format.
	Syntax
	<Member> StrToMember(<String>)
StrToSet	Constructs a set from a string expression.
	Syntax
	StrToSet(<String Expression>)
StrToTuple	Constructs a tuple from a string.
	Syntax
	<Tuple> StrToTuple(<String>)
StripCalculatedMembers	Removes calculated members from a set.

Syntax

Subset <Set> StripCalculatedMembers(<Set>)
Returns a subset of elements from a set.

Syntax

Sum <Set> Subset(<Set>, <Numeric Expression>)
<Set> Subset(<Set>, <Numeric Expression>, <Numeric Expression>)
Returns the sum of a numeric expression evaluated over a set.

Syntax

Tail <Numeric Expression> Sum(<Set>)
<Numeric Expression> Sum(<Set>, <Numeric Expression>)
Returns a subset from the end of a set.

Syntax

ThirdQ <Set> Tail(<Set>)
<Set> Tail(<Set>, <Numeric Expression>)
Returns the 3rd quartile value of a numeric expression evaluated over a set.

Syntax

ToggleDrillState <Numeric Expression> ThirdQ(<Set>)
<Numeric Expression> ThirdQ(<Set>, <Numeric Expression>)
Toggles the drill state of members. This function is a combination of DrillupMember and DrilldownMember.

Syntax

TopCount <Set> ToggleDrillState(<Set>, <Set>)
<Set> ToggleDrillState(<Set>, <Set>, <Symbol>)
Returns a specified number of items from the top of a set, optionally ordering the set first.

Syntax

TopPercent <Set> TopCount(<Set>, <Numeric Expression>, <Numeric Expression>)
<Set> TopCount(<Set>, <Numeric Expression>)
Sorts a set and returns the top N elements whose

cumulative total is at least a specified percentage.

Syntax

<Set> TopPercent(<Set>, <Numeric Expression>, <Numeric Expression>)

TopSum

Sorts a set and returns the top N elements whose cumulative total is at least a specified value.

Syntax

<Set> TopSum(<Set>, <Numeric Expression>, <Numeric Expression>)

TupleToStr

Constructs a string from a tuple.

Syntax

<String> TupleToStr(<Tuple>)

Union

Returns the union of two sets, optionally retaining duplicates.

Syntax

<Set> Union(<Set>, <Set>)
<Set> Union(<Set>, <Set>, <Symbol>)

UniqueName

Returns the unique name of a dimension.

Syntax

<Dimension>.UniqueName

UniqueName

Returns the unique name of a hierarchy.

Syntax

<Hierarchy>.UniqueName

UniqueName

Returns the unique name of a level.

Syntax

<Level>.UniqueName

UniqueName

Returns the unique name of a member.

Syntax

<Member>.UniqueName

ValidMeasure

Returns a valid measure in a virtual cube by forcing inapplicable dimensions to their top level.

	Syntax
Value	<p><Numeric Expression> ValidMeasure(<Tuple>) Returns the value of a measure.</p>
	Syntax
Var	<p><Member>.Value Returns the variance of a numeric expression evaluated over a set (unbiased).</p>
	Syntax
VarP	<p><Numeric Expression> Var(<Set>) <Numeric Expression> Var(<Set>, <Numeric Expression>) Returns the variance of a numeric expression evaluated over a set (biased).</p>
	Syntax
Variance	<p><Numeric Expression> VarP(<Set>) <Numeric Expression> VarP(<Set>, <Numeric Expression>) Alias for Var.</p>
	Syntax
VarianceP	<p><Numeric Expression> Variance(<Set>) <Numeric Expression> Variance(<Set>, <Numeric Expression>) Alias for VarP.</p>
	Syntax
VisualTotals	<p><Numeric Expression> VarianceP(<Set>) <Numeric Expression> VarianceP(<Set>, <Numeric Expression>) Dynamically totals child members specified in a set using a pattern for the total label in the result set.</p>
	Syntax
Wtd	<p><Set> VisualTotals(<Set>) <Set> VisualTotals(<Set>, <String>) A shortcut function for the PeriodsToDate function that specifies the level to be Week.</p>

Syntax

XOR
<Set> Wtd()
<Set> Wtd(<Member>)
Returns whether two conditions are mutually exclusive.

Syntax

Ytd
<Logical Expression> XOR <Logical Expression>
A shortcut function for the PeriodsToDate function that specifies the level to be Year.

Syntax

_CaseMatch
<Set> Ytd()
<Set> Ytd(<Member>)
Evaluates various expressions, and returns the corresponding expression for the first which matches a particular value.

Syntax

_CaseTest
Case <Expression> When <Expression> Then
<Expression> [...] [Else <Expression>] End
Evaluates various conditions, and returns the corresponding expression for the first which evaluates to true.

Syntax

{ }
Case When <Logical Expression> Then <Expression>
[...] [Else <Expression>] End
Brace operator constructs a set.

Syntax

||
{<Member> [, <Member>...]}
Concatenates two strings.

Syntax

<String> || <String>