Improving the performance and functionality of Mondrian open-source OLAP systems

Pablo Sendín-Raña¹, Francisco J. González-Castaño¹, *, †, Enrique Pérez-Barros², Pedro S. Rodríguez-Hernández¹, Felipe Gil-Castiñeira¹ and José M. Pousada-Carballo¹

¹Departamento de Ingeniería Telemática, Universidad de Vigo, Campus, 36310 Vigo, Spain
²SAEC-DATA SA, C. Gran Vía 94-1, 36203 Vigo, Spain

SUMMARY

For a long time, the design of relational databases has focused on the optimization of atomic transactions (insert, select, update or delete). Currently, relational databases store tactical information of data warehouses, mainly for select-like operations. However, the database paradigm has evolved, and nowadays on-line analytical processing (OLAP) systems handle strategic information for further analysis. These systems enable fast, interactive and consistent information analysis of data warehouses, including shared calculations and allocations. OLAP and data warehouses jointly allow multidimensional data views, turning raw data into knowledge. OLAP allows ‘slice and dice’ navigation and a top-down perspective of data hierarchies. In this paper, we describe our experience in the migration from a large relational database management system to an OLAP system on top of a relational layer (the data warehouse), and the resulting contributions in open-source ROLAP optimization. Existing open-source ROLAP technologies rely on summarized tables with materialized aggregate views to improve system performance (in terms of response time). The design and maintenance of those tables are cumbersome. Instead, we intensively exploit cache memory, where key data reside, yielding low response times. A cold start process brings summarized data from the relational database to cache memory, subsequently reducing the response time. We ensure concurrent access to the summarized data, as well as consistency when the relational database updates data. We also improve the OLAP functionality, by providing new features for automating the creation of calculated members. This makes it possible to define new measures on the fly using virtual dimensions, without re-designing the multidimensional cube. We have chosen the XML/A de facto standard for service provision. Copyright © 2008 John Wiley & Sons, Ltd.

Received 20 November 2007; Revised 30 June 2008; Accepted 1 July 2008

KEY WORDS: OLAP; data warehouses; XML/A

*Correspondence to: Francisco J. González-Castaño, Departamento de Ingeniería Telemática, Universidad de Vigo, Campus, 36310 Vigo, Spain.
†E-mail: javier@det.uvigo.es

Copyright © 2008 John Wiley & Sons, Ltd.
1. INTRODUCTION

1.1. Motivation

On-line analytical processing (OLAP) [1,2] is a formal data mining approach for multidimensional data frameworks. Unlike on-line transaction processing in relational databases (OLTP), OLAP provides quick answers to analytical queries. It deals with bulk data, and operations are generally read-only. OLAP systems return answers to complex queries in 0.1% of the time that they would take in OLTP systems.

In 1993, Codd published the OLAP white paper [3], with the well-known 12 rules describing desirable OLAP features. Pendse summarized these rules and coined the term fast analysis of shared multidimensional information (FASMI), as an alternative definition of OLAP. This definition is now widely accepted. It states that an OLAP system satisfies:

- **Fastness**: To deliver most responses to end users in 5 s (1 s for simple analyses, few answers reaching 20 s).
- **Analysis capability**: The system must adapt itself to any business logic and statistical analysis, without affecting user perception.
- **Resource sharing**: Providing security and locking concurrent updates at an appropriate level.
- **Multidimensionality (the key requirement)**: The system must provide a multidimensional view of data. Multidimensionality is defined by multiple dimensions, hierarchies and levels.
- **Information generation**: Meaning both data and the related information (axis, level, members, etc.).

OLAP systems offer intuitive views of large data sets. They often rely on a data warehouse [4] that manages the information. It supports OLAP, and its maintenance is independent of the operational database. Query throughput and response times are key issues [5,6].

OLAP solutions provide ‘slice and dice’ navigation operations (selection and projection), rollup (progressive decreasing of the level of detail), drill-down (detail increasing) and pivot (multidimensional axis re-definition, data view modification).

The driving application in our research has been the Galician Technological Fishing Platform (also called Pesca Fresca). This platform is a joint collaboration between the Spanish Ministry of Agriculture, Fishing and Maritime Affairs and the Regional Fishing and Maritime Affairs Council of Xunta de Galicia (Galician Regional Government). It holds statistical information on fishery sales (daily prices, auction data), fishing boat reports and other commercial information (Galicia has the most important fishing port in Europe, Vigo). The platform relies on a large relational database (Pesca Fresca database). Pesca Fresca analyses are currently severely limited, as they lack summarized and consolidated data. OLAP seems a natural approach to support the required reporting, enabling users to extract the knowledge themselves.

Currently, a reasonably cost-effective solution for our needs is an OLAP system keeping the information in a relational database (Relational OLAP or ROLAP). This allows an easy migration from the legacy structure to a multidimensional analysis system. Decision-support systems (DSS) (such as OLAP) involve complex aggregate queries over large volumes of data, and the underlying data warehouse must typically build many summarized tables (materialized aggregate views) [7–9]. However, the design of these tables is complex, their maintenance is cumbersome and it is necessary...
to keep them consistent when storing or updating information in the data warehouse [10]. Thus, we have chosen to employ a cache of aggregate views instead.

Consequently, we have developed a tailored Mondrian solution that summarizes information views in a memory cache, with a fast recovery from accidental shutdowns (by recovering the previous state of the aggregate views from the cache). This solution allows fast responses after system startup (processing time drops to $\sim 10\%$ of the base time—without any summarized technique), and it fulfills FASMI rules. As current open-source OLAP systems do not implement cold starts (which are necessary when using caches), we have adapted our Mondrian OLAP server to support them. We have also introduced a new functionality to define special virtual dimensions (calculated dimensions), to dynamically create calculated members. It avoids the definition of redundant information in each multidimensional query or the definition of new calculated members when the cube changes. To the best of our knowledge, no such functionality for calculated dimensions has been previously proposed.

### 1.2. State of the technology

There exist different approaches to analytical processing:

- **Multidimensional OLAP (MOLAP):** This array-based model provides immediate indexing of the information for cube queries. The data are stored in a multidimensional structure.
- **ROLAP:** The records are stored in standard relational tables.
- **Hybrid OLAP (HOLAP):** It seeks to combine the advantages of MOLAP and ROLAP technologies, using an array-based model for summary-type information, and a relational database for detailed information.

MOLAP scales worse than ROLAP, as the relational database can hold a large amount of data [11]. Therefore, ROLAP is a common developer’s choice.

Many data warehouse queries require summarized data (using aggregate views), for the response times to be practical [12]. The challenges of materialized views are as follows:

- identification of the aggregate views;
- how to answer queries with materialized views;
- keeping data consistency while updating materialized views.

Currently there exist many related proposals, such as a Grid MOLAP [13]. Thalhammer and Schrefl [14] proposed a new category of DSS, which automate decision making and extend conventional data warehouses with analysis rules. However, the common approach is a conventional schema combining a passive data warehouse and OLAP tools [15].

In ROLAP servers, each multidimensional query is parsed into several sentences for the underlying relational layer. Conversely, there are prototypes that parse relational queries into multidimensional sentences and obtain information from the resulting multidimensional views [16].

Interoperability is a key issue to communicate OLAP tools. Some proposals are MDAPI [17], OLE DB for OLAP [18], JOLAP [19] and others [20], but none of them has become a commonly accepted standard. Only XML for analysis (XML/A) [21–23] can be considered a de facto standard.
1.3. Available resources

Nowadays, business intelligence (BI) suites employ OLAP engines for analyzing large data sets. The most relevant commercial tools are SAP/BW, Hyperion Essbase and Microsoft Analysis Services.

SAP Business Information Warehouse (SAP/BW) has an OLAP engine that provides direct access to relational databases with DB Connect (therefore, SAP/BW may also rely on RDBMS). However, it has a proprietary design for multi-dimensional analysis. The SAP/BW star schema is established between the InfoCube and Master Data Tables.

The Hyperion Essbase product family includes the Hyperion Essbase OLAP Server, which takes advantage of sparsity to minimize the amount of physical memory and disk space to represent large multidimensional data. Essbase has a proprietary method to reduce the amount of physical memory without increasing processing time.

The storage modes in Microsoft Analysis Services depend on the model used, which can be defined as MOLAP, ROLAP or HOLAP. The analysis services architecture follows the unified dimensional model (UDM) concept, which provides a way to encapsulate accesses to multiple heterogeneous data sources into a single model. To attain high performance levels, UDM employs a MOLAP cache of underlying data (typically, relational databases).

The OLAP engines of these systems are based on closed, patented, proprietary solutions, which are optimized for high performance. Open-source systems achieve a reasonable performance level, even for large databases, and focus on flexibility. For example, Mondrian admits many add-ins (such as JPivot for web integration) that make it very useful. For these reasons, and given the preference of European public administrations in open-source tools, we focus on them. The following open-source OLAP systems are especially relevant:

- Mondrian [24] is a Java ROLAP server. It includes some development tools, such as Cube-Designer, and a workbench. Mondrian employs JPivot [25] (a JSP custom library) to render OLAP tables and to perform typical OLAP navigations.
- PALO [26] (Jedox application) is a 64-bit MOLAP server. It is a cell-oriented, memory-based, real-time system, specifically developed for spreadsheet data storage and analysis. PALO has a write-back feature. It offers an API in various languages (C, PHP, .NET and Java) and it has an EXCEL plug-in for data access. It depends on Microsoft software and it cannot deal with large databases.
- OpenOLAP [27] is a Java ROLAP/MOLAP server. It does not support some relevant relational databases, such as SQL server.

The following table shows the strengths and the weaknesses of these resources.

<table>
<thead>
<tr>
<th>Issues</th>
<th>Mondrian</th>
<th>Palo</th>
<th>OpenOLAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open source</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>RDBM-independence</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Well documented</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>ROLAP server</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>MOLAP server</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Supports large databases</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
As our choice is a ROLAP system, we have selected Mondrian (Pentaho Project since 2005, as part of a full open-source BI suite). Version 2.3.2 was the latest one at the time this paper was written, providing efficient cache flushing. Mondrian is free, portable and easy to integrate, and it supports many relational database management systems (RDBMS), such as Oracle, SQL Server, mySQL and others. As we have explained, Mondrian is not directly comparable to commercial tools. However, we found it interesting to test Mondrian against Microsoft Analysis Services in the context of this paper. Specifically, we tested the latter on the same data cube that we had created for Mondrian. The following facts are relevant:

- In our tests, Mondrian and MS analysis services relied on mySQL and SQL server, respectively.
- MS analysis services would be typically used in MOLAP or HOLAP mode, rather than in ROLAP mode.
- MS analysis services interact directly with the database, whereas in our implementation the queries were submitted to Mondrian via XML/A, before getting to the database.

Keeping that in mind, we found that MS analysis services was four times faster than our Mondrian-based implementation. This is tolerable if other advantages of open source are considered. In addition, our contributions are compatible with future Mondrian releases, and it has happened in the past that the gap between commercial software and open-source equivalents has dropped (MS Office vs OpenOffice, for example).

Getting back to Mondrian, in addition to its higher scalability, the ROLAP paradigm has extra benefits over MOLAP. It requires neither any information preprocessing nor storage, and it directly accesses data from relational databases such as Pesca Fresca (it generates SQL queries to obtain the information).

A Mondrian schema defines a multidimensional database over an underlying relational database. The schema comprises a logical model and a mapping onto a physical model. The logical model consists of cubes, dimensions, hierarchies, levels and members. It accepts queries in MDX language [28–30], which allow OLAP systems to obtain multidimensional information (on the other hand, RDBMS employ the SQL). MDX queries yield intensive workloads, with access to millions of records and many join operations and aggregations.

Mondrian schemas are written as XML files. The OLAP cubes in the logical model (also called multidimensional cubes or hypercubes) define measures (sets of numeric data) categorized by dimensions. These dimensions are composed of hierarchies, levels and members. The dimensions are derived from dimension tables, fact tables or in-line definitions, and the measures are defined from records in the fact table. All tables are relational in our case. Cube metadata follow either a star or a snowflake schema.

OLAP performance typically depends on the aggregation mechanism, which consists in building measures with different granularities (level dimension, for example) from the fact table. The combination of all possible aggregations and the database provides the answers to every MDX query.

Mondrian itself does not store data on the disk. It works directly with RDBMS data, and it stores only pieces of data in cache (Mondrian cache) after accessing them. This limits the performance when working with huge data sets. In order to speed up query processing, the underlying relational layer will usually hold many summarized tables (materialized aggregate views of the database) along various dimensions†. Aggregation data are also stored in the Mondrian cache (a specialized...

†Mondrian defines materialized views with lost and collapsed dimensions.
buffer in the memory) when they are read from the relational database. The memory cache is optimized for fast answers to subsequent requests. Mondrian cache control keeps integrity, preventing the system from inconsistent query results.

As previously mentioned, aggregate tables are difficult to design and maintain. Furthermore, materialized aggregate views are redundant data tables. The database administrator must ensure consistency when the information changes, and aggregation tables and store procedures may lead to data integrity problems.

1.4. Extensibility

Mondrian is highly flexible. It is possible to add new modules and integrate other tools. The user can include aggregate functions himself, as a part of specific Mondrian plug-ins.

Mondrian can be configured in an XML/A provider mode, allowing for OLAP interoperability. In 2001, Microsoft and Hyperion announced the XML/A specification [19–21], enabling a standardized OLAP API. Most OLAP systems endorsed it, and XML/A became the de facto standard. XML/A employs MDX as the query language to obtain multidimensional data. XML/A communication messages follow web standards—HTTP [31], SOAP [32] and XML [33]. A typical Mondrian installation includes a servlet engine to set an XML/A provider.

Figure 1 illustrates Mondrian operation. The user issues MDX queries to the Mondrian server using SOAP messages, using the XML/A protocol. If necessary, the server requests data from the relational database with SQL queries, or it consults cache memory otherwise. OLAP results are also wrapped in SOAP answers. Mondrian configuration files (defining Mondrian schemas, XML/A data sources and others) are declared in XML.

Figure 1. Mondrian operation.
The XML/A specification defines two methods:

- **Discover**: To find the data and metadata that are necessary for MDX queries from the web service or the provider.
- **Execute**: To send the MDX queries.

### 1.5. Contributions

In this paper we describe our *tailored*, improved Mondrian server, with a specialized cache and additional functionality to obtain calculated dimensions. We provide solutions for

- preventing inconsistencies using aggregate data of the Mondrian cache. We load the pre-aggregated data as soon as possible at system startup (*cold start*);
- dynamically obtaining calculated members from cube measures with calculated dimensions, by means of a simple data relation.

In cold starts, it is necessary to build the aggregate cache as quickly as possible. As a ROLAP server, Mondrian obtains the data from the underlying relational database in the case of cache miss. Usually, data access following a miss takes too long, as a request encompasses millions of record accesses. By loading all the corresponding data into an aggregate cache, the answers will typically be sooner available. Our implementation includes a fast cache load in the case of unexpected system shutdowns.

The remainder of this paper is organized as follows. Section 2 provides some background about the *Pesca Fresca* project and multidimensional design on top of relational databases. Section 3 explains the concept of calculated dimensions, which makes it possible to create multidimensional calculated members at runtime. Section 4 proposes a solution for the cold start workload problem to bring aggregate views to the memory cache. Section 6 concludes the paper.

### 2. MULTIDIMENSIONAL DESIGN

We take the original relational database of the *Pesca Fresca* platform, whose structure was not optimized for multidimensional access, as a starting point. Figure 2 shows a simplified model of the relational structure. The fact table (*sep_gdia*) is linked to the dimension tables, *sep_fishery* and *sep_species*. Tables *sep_fishery* and *sep_species* allow defining the geographical location and fishing species dimensions, respectively. The remaining dimensions (time dimension and data state dimension) result from fact table columns (so-called *degenerate dimensions*). Thus, the relations are as follows:

- Fact table (*sep_gdia*).
- Time dimension, defined from column *date* in the fact table.
- Data state dimension, defined from column *data_state* in the fact table.
- Geographic location dimension, defined in *sep_fishery* and linked to the fact table.
- Fishing species dimension, defined in *sep_species* (joined with *sep_species_group* and *sep_fao*) and linked to the fact table.
Figure 2. Model of the underlying structure.

The fact table contains the following measures:
- sales notes (column notes);
- quantity (column quantity);
- sales (column price);
- minimum price (column pmin);
- maximum price (column pmax);
- average price (calculated measure, which results from measures quantity and amount);
- number of days (measure from column data, obtained with different count aggregators);
- number of fishing species (measure from column species_id, obtained with different count aggregators);
- number of fisheries (measure from column fishery_id, obtained with different count aggregators).

In the Mondrian schema that defines the multidimensional cube, we declare the aggregate functions for each measure, as well as the dimensions in the schema below. In it, we deliberately replace some hierarchy, level and SQL definitions by notes, as they are not necessary to understand the concept:

```xml
<?xml version='1.0'?>
<Schema name='PescaFrescaSchema' measuresCaption='Data Analysis'>
  <Cube name='PescaFresca'>
    <Table name='sep_gdia'/>
    <Dimension name='Time' type='TimeDimension' caption='Time Dimension'>
      <Hierarchy and Level definitions/>
    </Dimension>
    <Dimension name='Time.byQuarters' type='TimeDimension' caption='Time Dimension by Quarters '>
      <Hierarchy and Level definitions/>
    </Dimension>
    <Dimension name='Time.byMonths' type='TimeDimension' caption='Time Dimension by Months'>
      <Hierarchy and Level definitions/>
    </Dimension>
    <Dimension name='Time.byDays' type='TimeDimension' caption='Time Dimension by Days'>
      <Hierarchy and Level definitions/>
    </Dimension>
    <Dimension name='Geographic location ' type='GeographicDimension' caption='Geographic location '>
      <Hierarchy name='byRegions' hasAll='true' allMemberName='All regions' primaryKey='fishery_id'>
        <Level definitions/>
      </Hierarchy>
    </Dimension>
  </Cube>
</Schema>
```

DOI: 10.1002/spe
<Hierarchy name="byZones" hasAll="true" allMemberName="All zones" primaryKey="fishery_id">
  Level definitions
</Hierarchy>
</Dimension>
<Dimension name="Fishing species" foreignKey="species_id" caption="Fishing Species Dimension">
  <Hierarchy name="Fishing Species" hasAll="true" allMemberName="All species" primaryKey="species_id"
    primaryKeyTable="species_table">
    SQL definition
    <Level name="Group" column="group_description" uniqueMembers="true"/>
    <Level name="FaoSpecie" column="fao" uniqueMembers="true"/>
    <Level name="Species" column="species_description" uniqueMembers="false"/>
  </Hierarchy>
</Dimension>
<Dimension name="Data State" caption="Data State Dimension">
  Hierarchy and Level definitions
</Dimension>
<Dimension name="Calculated Dimension" caption="Calculated Dimension" foreignKey="gdia_id">
  <Hierarchy hasAll="true" memberReaderClass="mondrian.rolap.MiscHierarchy">
    <Level name="Calculated Dimension" uniqueMembers="true"/>
  </Hierarchy>
</Dimension>
<Measure name="Sales Notes" column="notes" aggregator="sum" datatype="Numeric" formatString="#,###.##'/
  <Measure name="Quantity" column="quantity" aggregator="sum" datatype="Numeric" formatString="#,###.##'/
  <Measure name="Sales" column="price" aggregator="sum" datatype="Numeric" formatString="#,###.##'/
  <Measure name="Maximum Price" column="pmax" aggregator="max" datatype="Numeric" formatString="#,###.##'/
  <Measure name="Minimum Price" column="pmin" aggregator="min" datatype="Numeric" formatString="#,###.##'/
  <Measure name="Days Number" column="date" aggregator="distinct count" datatype="Numeric" formatString="#,###'/
  <Measure name="Species Number" column="species_id" aggregator="distinct count" datatype="Numeric" formatString="#,###.##'/
  <Measure name="Fisheries Number" column="fishery_id" aggregator="distinct count" datatype="Numeric" formatString="#,###'/
  <CalculatedMember name="Average Price" dimension="Measures" caption="Average Price">
    <CalculatedMemberProperty name="FORMAT_STRING" value="#,###.##'/
  </CalculatedMember>
</Cube>
</Schema>

Some notes about this cube definition:

- The definition of the time dimension is not optimal. Although this time structure usually relies on a
dimension table, the initial requirements forced us to employ this alternative.
- The time dimension is defined as four hierarchies. With this configuration, we allow requests with a two-axis time dimension. For example, a sales reporting with years in the vertical axis and months in the horizontal axis.
- We define two hierarchies for the geographical location dimension, with different levels. This allows us to access dimension members from diverse paths.
- The fishing species dimension must define an SQL sentence to create levels. We follow this procedure to obtain dimension data due to the underlying data model. We cannot use the usual Mondrian join tags.

Each measure corresponds to a Mondrian aggregate function (sum, avg, distinct count, etc.), except for the calculated measure (average price measure, defined with a formula).

### 3. IMPROVING MONDRIAN FUNCTIONALITY: CALCULATED DIMENSION

Our aim is to provide a solution for calculated members in Mondrian. Calculated members are special ones whose values are declared in MDX sentences (or in the Mondrian schema) and
calculated at runtime. Normal members directly obtain their value from aggregate data, but calculated members use a formula for that purpose. The definition of the formula is the only stored item (except for aggregate values). Calculated members allow adding measures to multidimensional cubes without increasing their size, as well as defining powerful expressions. For example, they can be used to calculate percentages or more complex operations. If the user wants to define several calculated members for each measure of the multidimensional cube, the Mondrian schema grows exponentially and maintenance becomes a nightmare, or, if the user defines the calculated members in each MDX query, it is necessary to declare the formula every time. Furthermore, when we add new measures, we must declare new calculated members again.

We propose to introduce calculated dimensions. This contribution enables the dynamic creation of calculated members at runtime. This enhancement improves user experience, providing an intuitive way to build queries. Mondrian engine performance is not affected by this contribution.

By calculated dimensions we mean a new type of dimensions of the Mondrian cube. In Mondrian schemas, every dimension results from dimension tables (from the relational database, inline or generated by a Java class), and they are linked to data (measures). A calculated dimension does not have these relations. It only specifies the behavior when a measure (or several ones) crosses with its members. This cross automatically produces a set of calculated members; thus, it is unnecessary to define them with an MDX query in each request. The measures usually have several associated common expressions. We customize them and create a dimension whose members are defined as interfaces, so that, when crossed with any measure, the result is a calculated member that implements the corresponding rules.

Currently, there are two ways to define calculated members in Mondrian:

- In MDX expressions, by means of with member statements. The expression defines the required measures and dimension members.
- With the CalculatedMember tag in the Mondrian schema. We can define the name of the calculated member and the formula to obtain its value.

The average price measure (defined in the previous schema) is an example of calculated member (applied to the measure dimension). A calculated dimension automatically allows creating calculated members when we cross it with any measure of the Mondrian star.

Let us consider the multidimensional cube schema that follows:

```xml
<Dimension name="Calculated Dimension" caption="Calculated Dimension" foreignKey="gdia(_id)"
    <Hierarchy hasAll="true" memberReaderClass="mondrian.rolap.MiscHierarchy">
        <Level name="Calculated Dimension" uniqueMembers="true"/>
    </Hierarchy>
</Dimension>
```

Calculated Dimension results from the MiscHierarchy class, a MemberReader subclass of Mondrian with a single level, as its data do not reside in the RDBMS. The MiscHierarchy class is a custom member reader, and it provides the member list of the dimension. Unlike common dimensions, this type of members is useful only when combined with measures in the MDX declaration. Without this mixing, they do not return any result. The MiscHierarchy class defines only the semantics of the dimension.

The MemberReader class implements common operations to retrieve members from a hierarchy. However, these members are quite restrictive for our purposes. Therefore, we employ an MDX pre-filtering technique.
Let us describe an application. For example, if we demand 2007 sales, as well as their variation and the variation percentage regarding the previous year, we use the following MDX sentence:

```
SELECT
Crossjoin ( {[Measures].[Sales]},
            {[Calculated Dimension].[Data], [Calculated Dimension].[Absolute Variation],
             [Calculated Dimension].[Relative Variation], [Calculated Dimension].[Percent]} ) ON COLUMNS,
NON EMPTY {[Time.byYears].Members} ON ROWS
FROM [PescaFresca]
```

It can be noted that we do not have to define calculated members. If we need more calculated members from any measure, we only have to include it in the `crossjoin` function. Otherwise, we would have to declare three new calculated members exclusively for that measure, and the MDX query would become the following:

```
WITH
MEMBER [Measures].[Data absolute variation]
AS '[Measures].[Sales] - ([Measures].[Sales], [Time.byYears].CurrentMember.PrevMember)',
MEMBER [Measures].[Data relative variation]
AS '([Measures].[Sales] - ([Measures].[Sales], [Time.byYears].CurrentMember.PrevMember)) / ([Measures].[Sales], [Time.byYears].CurrentMember.PrevMember)',
FORMAT_STRING='#.###.##%'
MEMBER [Measures].[Data percent]
AS '[Measures].[Sales] / ([Measures].[Sales], [Geographic location.byRegions].CurrentMember.Parent)',
FORMAT_STRING='#.###.##%'
SELECT
Crossjoin ( {[Measures].[Sales], [Measures].[Data absolute variation],
             [Measures].[Data relative variation], [Measures].[Data percent]} ) ON COLUMNS,
NON EMPTY {[Time.byYears].Members} ON ROWS
FROM [PescaFresca]
```

The `RolapConnection` class from the Mondrian project includes a method to parse MDX queries. We have developed a subclass (`RolapConnectionSIP`) with a pre-filtering method to build the necessary calculated members. Crossing any measure with any member of the calculated dimension (in the MDX sentence) produces a calculated member. The pattern to build these calculated members is always the same. Therefore, `RolapConnectionSIP` loads the configuration values for the calculated dimension functionality from an XML configuration document when Mondrian starts. Later, it analyzes the MDX sentence from the user request, and detects if this MDX sentence contains calculated dimensions. In that case, it parses the MDX sentence to build an equivalent, valid MDX query, with the resulting calculated members.

An XML document states the calculated dimension, its members and the necessary elements to parse the special cross function. For example, consider the following XML configuration for the `PescaFresca` cube, defining six members:

- all members;
- data;
- past;
- absolute variation;
- relative variation and
- percentage.
The XML configuration document for the PescaFresca cube is

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE calculatedDimension SYSTEM "dimconfiguration.dtd">
<calculatedDimension>
  <baseMember>BASE_MEMBER</baseMember>
  <baseSliceDimension>BASE_SLIDE_DIMENSION</baseSliceDimension>
  <members>
    <member isAll="true" name="All"/>
    <member name="Data"/>
    <member name="Past">(BASE_MEMBER,[Time].CurrentMember.PrevMember) </member>
    <member name="Absolute variation">
      (BASE_MEMBER-(BASE_MEMBER,[Time].CurrentMember.PrevMember))
    </member>
    <member name="Relative variation" formatString="#,###.##%">
      (BASE_MEMBER-(BASE_MEMBER,[Time].CurrentMember.PrevMember))/
      (BASE_MEMBER,[Time].CurrentMember.PrevMember)
    </member>
    <member name="Percentage" formatString="#,###.##%">
      BASE_MEMBER/(BASE_MEMBER BASE_SLIDE_DIMENSION)
    </member>
  </members>
  <relatedDimensions>
    <dimension>Fishing Species. Fishing Species</dimension>
    <dimension>Geographic location.byRegions</dimension>
    <dimension>Time.byYears</dimension>
  </relatedDimensions>
</calculatedDimension>
```

The special member All member designates the members of the dimension. The isAll attribute allows defining it (like common Mondrian dimensions). Data refers to the measure that is involved in the crossjoin function. The rest of the members have an associated function, which is used when the parser creates the calculated members. If the formatString attribute is applied to these members, the evaluation of its expression determines the format of the cell value.

Elements baseMember and baseSliceDimension are useful constants to parse the MDX sentence. They allow creating the calculated members in the RolapConnectionSIP class. Similarly, relatedDimensions allows parsing the MDX query when other dimensions are involved in the expression. It has multiple dimension elements, corresponding to all possible values for slicing.

The following sentence produces a report with three measures and every possible associated calculated member:

```sql
SELECT
crossjoin {([Measures].[Sales Notes], [Measures].[Quantity], [Measures].[Sales]),
  ([Calculated Dimension].Members) ON COLUMNS,
NON EMPTY {[Time.byYears].Members} ON ROWS
FROM [PescaFresca]
```

Without the calculated dimension, we would need 12 new dimensions to obtain the same report. Figure 3 shows the report (using Jpivot) for measure Sales.

The Mondrian server checks the received MDX queries. If the calculated dimension belongs to a crossjoin function with one measure at least, it calls a specific method to parse and generate the calculated members. The configuration file defines the parsing process, and the resulting calculated
members are added to the original MDX query, thus reformulating the sentence. This modified query is subsequently processed and the response conveys the report with the calculated values to the user.

It can be noted that this functionality can be applied to any Mondrian solution. The Mondrian administrator has to define the XML configuration document stating the calculated dimensions, as he has to design the XML configuration document to define the Mondrian cube.

4. IMPROVING THE PERFORMANCE OF COLD STARTS

Usually, each row in the underlying relational fact table has columns for every dimension and measure. The summarized tables of the data warehouse and the cache (the last element of the aggregation strategy) support the pre-computed aggregates for large data sets. The adaptive Mondrian cache holds pre-computed aggregations in the memory; hence, subsequent queries can access them avoiding the relational database.

Mondrian follows an aggregate view strategy to improve performance, as typical ROLAP systems do. The multidimensional cube automatically uses the fact table or the aggregate tables to obtain the information from the relational database. If the aggregate tables directly supply aggregate data (without the fact table), the answers to SQL queries are faster. We only have to define some rules in the multidimensional schema.

The aggregate tables are defined in the relational database, and they impose some requirements:

• The design of adequate summarized tables to improve Mondrian performance.
• When the fact table changes (due to insert or update actions), the summarized tables may need an update, depending on their data. Usually, the changes are periodically loaded (for example, once a day) by store procedures. Meanwhile, the data warehouse becomes unavailable.
• If the data warehouse must be permanently available, the changes in the fact table and the diverse aggregate tables (update operations) may compromise the consistency of the data model. It may happen that an MDX query (producing multiple SQL statements) accesses the updated fact table but the aggregate tables do not change.

As there are plenty of summarized tables to maintain, a critical issue in data warehousing is efficient aggregate table maintenance [34]. Moreover, update time is also critical (typically this time is limited). Our design requirements stated that all data had to be permanently available. However,
updates may take place while Mondrian delivers MDX queries from the users. The aggregation views depend on the fact tables, and before the store procedures update aggregate data in the summarized tables, the user may issue an MDX request after the fact table has been modified. This situation produces an inconsistent answer (the report combines changed with unchanged data).

In order to cope with these situations, in typical Mondrian layouts there are replicated servers with live and backup configurations, behind a front load balancer. When new data become available, the backup server enters the offline state, uploads the data into the fact and aggregate tables, and sets them online. The same operations take place with the live server. Nevertheless, this approach is unacceptable for the Pesca Fresca platform. It is necessary to use the original server hosting the relational database, which contains all the multidimensional data for the OLAP cube without any replication. The administrator may introduce new information or manage the relational database at any time. The data warehouse must always be available, and any change in the dimension or the fact tables must respect consistency.

As we do not use summarized views of the relational database, we avoid consistency problems and data unavailability. Our approach consists in employing the Mondrian cache to summarize the fact table into aggregations. The original purpose of the Mondrian cache is to keep aggregate data of recent MDX queries in the memory, adapting itself to user interaction. We extend this mechanism to summarized views in relational databases. With aggregate tables, creation and updates are subject to time constraints, and the same constraints hold for the summarized view in the Mondrian cache. However, there is no aggregate information in the cache when the Mondrian server starts. Therefore, a process must load the summarized data (cold start process), as quickly as possible.

The cache is empty on Mondrian initialization. When Mondrian starts processing queries, it issues SQL calls (which result from MDX parsing) to the relational database, getting member lists and determining cardinality. Then it loads segments in the cache. When Mondrian shuts down and reboots, its cache is empty again and it is necessary to reload it. This takes a significant time depending on the cube size (the relational database must perform calculations to obtain the aggregations). For example, in one of our tests, a query on a 4 GB cube took 6 min before returning the results.

Figure 4 shows the improvement due to cached aggregated data. Without it, the response times of complex SQL queries are unacceptable. Remember that these queries usually involve many SQL ones, and each of them requests aggregate operations involving thousands or millions of records from the relational database.

Our solution for cold starts is fast and has a negligible impact on maintenance. Aggregate data reside in the Mondrian cache, and data refresh operations use specific control functions, which ensures data integrity. The solution comprises two steps:

- To create, store and load a parallel cache, which contains all the data for the cold start and generated data for user queries. The parallel cache (or secondary-level cache) is the data source for the cold start process.
- To create a script of MDX queries to initialize the Mondrian cache. At startup, Mondrian loads the parallel cache and executes the script in a background thread. Every MDX query finds the data in the parallel cache, and Mondrian builds its own cache automatically and transparently. This method is faster than obtaining the data from the relational database.

This solution requires slight changes in the Mondrian cache code, specifically in segment generation (to obtain cache segments, i.e. multidimensional areas defined by one or more members, from
We retain the potential to add new Mondrian cache features. The last version (2.3) has a cache control API, providing fine-grained control of cache data and keeping data consistency in concurrent requests. The new cache control uses cache segments and it is possible to refresh specific cache regions (in previous versions, only complete flushes were possible). Thus, in the case of any change in the relational database, we can simply refresh the cache segments that contain the corresponding aggregate data, keeping valid data in the Mondrian cache for future MDX queries.

We must also refresh the parallel cache data in Mondrian flushing actions. If any Mondrian cache segment is flushed, we must erase the corresponding element.

We also considered other solutions, as Mondrian cache serializations or segment cache mapping, but we found several problems:

- The load and store procedures took a long time and consumed many resources, usually raising Java heap size exceptions.
- The generated files were too large, in the GB range. The times to load and store those files were unacceptable.
- The modification of the original Mondrian source was too intrusive.

The cold start process executes multiple MDX queries to build the Mondrian cache. The MDX queries are specified by XML document `mdxgenerator.xml`, an MDX script.

Basically, besides changing cache segment generation, we add two new classes to the Mondrian project: `ServletColdStart` and `CacheStore`. The servlet class is used to:

- initialize the cold start;
- start the cold start;
- load the parallel cache from the disk, using the `CacheStore` class;
- store the parallel cache in the disk, using the `CacheStore` class;
- flush the Mondrian cache using the cache control API, and refresh the parallel cache.

A Java web server (for example, Tomcat [35]) receives the `ServletColdStart` customized request and executes the corresponding actions.
CacheStore is a wrapper class for the parallel cache. It allows adding and removing entries, as well as loading/storing the parallel cache from/to the disk. This wrapper class has the following attributes:

- **Rowset list**, which contains the segment data from every Mondrian cache segment.
- **idSegment list**, with the segment identifiers of the Mondrian cache.
- **listHashQueries list**, a hash code of each SQL query to obtain the cache segments.

These attributes model the parallel cache. The store process serializes and saves them. The load process follows the opposite procedure.

The ServletColdStart dispatcher takes its parameters (recognizing the action and other variables) from the GET requests. There are four possible actions:

- **Start (cold start)**: At the initialization, the XML document with the MDX sentences to generate the Mondrian cache is loaded. Mondrian executes the queries and generates its cache. Other queries can be executed in parallel. If the requested data correspond to consistent data in the Mondrian cache, the query does not need the relational database.
- **Load (from the disk)**: The parallel cache is loaded from the disk into the CacheStore class.
- **Store (in the disk)**: The parallel cache (attributes from the CacheStore class) is saved to disk.
- **Flush**: The elements to be flushed are defined with input parameters. They are necessary to build cache segments definitions, and these cache regions are flushed using the cache control API. Then, we erase the corresponding elements from the parallel cache. Finally, we launch the cold start process to load the new data from the relational database into the Mondrian cache (and the parallel cache), and we store the updated data in the disk with the store process.

We illustrate these actions in Figure 5.

The parallel cache is created in two ways:

- reading the data from the disk;
- reading the data from the Mondrian cache.

![ServletColdStart actions.](image-url)
The second one is the natural way to build the parallel cache, and the only one when there is no information in the disk. Mondrian has a complex structure to manage the cache, with aggregation and segment classes, and methods to obtain the information. In the case of cache miss, it queries the RDBMS with SQL sentences. We modify this as follows: when Mondrian gets data from the relational database and creates the cache segment, we add the segment data to the parallel cache, with a segment identifier and the SQL hash code. This assists the queries of the cold start process, but it also applies to normal user queries (the parallel cache also contains data from user reporting).

After loading the parallel cache, if it contains data and the Mondrian cache is empty, the cold start process does not issue SQL queries to build the Mondrian cache segments. Instead, the segments are created from parallel cache data, and processing time drops. In Figure 6, the first read obtains data from the Mondrian cache, but the second fails and Mondrian gets data from the parallel cache.

When the XML/A Mondrian server starts, the parallel cache loads data from the disk and the cold start process runs in background mode, building the Mondrian cache. The user can access the Mondrian server in concurrent mode and execute MDX queries to obtain multidimensional data. If neither the Mondrian nor the parallel caches have any data, they are obtained from the relational database, and saved in the parallel cache for subsequent requests. The Mondrian administrator may call a store process to save the parallel cache to the disk at any time. A possible extension could define periodic tasks to automatically carry the management procedures out (e.g. to store the parallel cache weekly).

This technique considerably improves the performance of the cold start. It reduces the time to execute the MDX script by an order of magnitude. In the absence of a cache, the execution of the MDX script takes from hours to days. With the parallel cache, the process just takes minutes.

Figure 7 shows cold start elapsed times with and without a parallel cache.

Memory consumption is a key issue. Mondrian needs a large amount of memory for the cache, and in our case for the parallel cache. If we use Mondrian jointly with a Tomcat server (or JBoss [36]), we must correctly dimension the Java heap space (initial and maximum). We can take the ‘cache disk’ size as a reference.
ServletColdStart also has a cache-flushing functionality. It employs the Mondrian 2.3 cache control API to refresh cache segments (they can be erased or modified). The Mondrian administrator chooses a temporal axis to define updated data in the fact table (RDBMS). The servlet takes the following actions to refresh the Mondrian cache:

- It flushes the segments from the Mondrian cache (defined by the temporal axis).
- It deletes parallel cache entries (only those flushed in the Mondrian cache).
- It launches the cold start process to load flushed cache segments. If the response to the MDX query is already available in the Mondrian cache (or in the parallel cache), nothing changes. Otherwise, Mondrian obtains fresh data from the relational database and inserts them into the Mondrian cache and the parallel cache.
- It stores the refreshed parallel cache in the disk.

All these procedures are executed by background threads, ensuring concurrent access to the data. The Mondrian cache API avoids inconsistency problems. It provides atomic flush from the user’s point of view (database reads are consistent). The parallel cache also provides concurrent access to consolidated data.

5. CONCLUSIONS

In this paper we present a solution to implement an efficient open-source ROLAP server, so that users may perform complex analyses that are unfeasible with classical systems. Our implementation is an evolution of Mondrian, an open-source ROLAP server.

Mondrian is an independent open platform, simple to use, with drill-through capabilities, multilingual definition, role definition, access control functions and extensive RDBMS support (for mySQL, SQL server, Oracle, Ingres, and any other ODBC or OLE DB data sources). An efficient OLAP system needs some type of data pre-aggregation technique to achieve FASMI features.
Traditionally, ROLAP servers achieve high analysis performance by means of a proper design of aggregated tables and configuration tuning. For really large multidimensional cubes, performance mainly depends on hardware and database tuning (requiring a complex design of aggregate tables, which is difficult to maintain), rather than on the ROLAP server. We avoid the design, maintenance and data consistency problems of aggregate tables: We have developed a Mondrian cache extension to provide fast answers at any time, keeping data integrity. Mondrian is scalable, in the sense that it can handle large databases and dimensions, as well as many users, but data load performance is low for large databases. We employ a cold start process to load the pre-aggregate views in a parallel cache, achieving satisfactory response times. This technique reduces response times by an order of magnitude, compared with the case without summarizing methods (i.e. directly accessing the base information). The cache control ensures data consistency while updating the relational database. The combination of the Mondrian cache, the parallel cache and the data warehouse allows fast recovery from server failures, guaranteeing a high quality of service. Moreover, the tailored Mondrian solution is still a real-time OLAP, because new data that reside in the relational server are updated in the secondary cache.

This contribution is non-intrusive. The persistent parallel cache, or secondary level cache, allows an easy migration to new Mondrian releases.

The second contribution is a functionality enhancement: the inclusion of a virtual dimension called calculated dimension. A Mondrian manager simply has to define the calculated dimensions, as he defines the multidimensional cube. With this functionality, new calculated measures can be created on the fly from intuitive cross operations between members of a calculated dimension and previously defined measures. This avoids repetitive declarations in MDX queries and Mondrian schemas and therefore improves user experience. The OLAP user defines new measures using a graphical interface, so that it is possible to obtain complex reports without re-designing the multidimensional cube or introducing large calculated members in each MDX sentence. Mondrian performance is not affected with this new functionality, and the MDX sentences still follow the standard specification.

As future work, we plan to enhance web integration, providing a new graphical interface for multidimensional analysis.

REFERENCES


31. HTTP. http://www.w3.org/protocols/ [July 2008].


