A meta-language for MDX queries in eLog
Business Solution

Sonia Bergamaschi #1, Matteo Interlandi #2, Mario Longo #3, Laura Po #4, Maurizio Vincini #5

#1 Department of Information Engineering, University of Modena and Reggio Emilia
via Vignolese 905, 41125 Modena, Italy
sonia.bergamaschi@unimore.it
#2 matteo.interlandi@unimore.it
#3 laura.po@unimore.it
#4 maurizio.vincini@unimore.it
#5 eBilling S.p.A.
Viale Virgilio 54/a, 41123 Modena, Italy
mlongo@ebilling.it

Abstract—The adoption of business intelligence technology in industries is growing rapidly. Business managers are not satisfied with ad hoc and static reports and they ask for more flexible and easy to use data analysis tools. Recently, application interfaces that expand the range of operations available to the user, hiding the underlying complexity, have been developed. The paper presents eLog, a business intelligence solution designed and developed in collaboration between the database group of the University of Modena and Reggio Emilia and eBilling, an Italian SME supplier of solutions for the design, production and automation of documentary processes for top Italian companies.

eLog enables business managers to define OLAP reports by means of a web interface and to customize analysis indicators adopting a simple meta-language. The framework translates the user’s reports into MDX queries and is able to automatically select the data cube suitable for each query.

Over 140 medium and large companies have exploited the technological services of eBilling S.p.A. to manage their documents flows. In particular, eLog services have been used by the major media and telecommunications Italian companies and their foreign annex, such as Sky, Mediaset, H3G, Tim Brazil etc. The largest customer can provide up to 30 millions mail pieces within 6 months (about 200 GB of data in the relational DBMS). In a period of 18 months, eLog could reach 150 millions mail pieces (1 TB of data) to handle.

I. INTRODUCTION

During the last decade Data Warehousing and Business Intelligence become key technologies for storing, analyzing and navigating business processes, providing managers and decision makers with OLAP [9] tools. OLAP is the most popular way to exploit information in a data warehouse, and it gives to end-users the opportunity to analyze and explore data interactively on the basis of the multidimensional model. Many commercial OLAP tools use languages, such as MDX (MultiDimensional eXpressions) [11]. An MDX expression returns a multidimensional result set that the user can navigate according to different viewpoints and at different levels of detail. MDX is very powerful and effective for IT managers, but it is unfeasible for unskilled users. On the other side, static predefined reports are easy to understand for business managers but are not flexible.

The approach we follow in eLog is intermediate between static reporting and OLAP, it is called semi-static reporting. In a semi-static report, users can follow a limited set of navigation paths. This approach provides some unquestionable advantages: (1) users need less skills to use data models and analysis tools than the ones needed for OLAP systems; (2) it is avoided the risk of achieving inconsistent or incorrect analysis results because of any misuse of aggregation operators; (3) posing constraints on the allowed analysis makes it possible to prevent users from unwillingly slowing down the system whenever they formulate heavy-demanding queries [7].

eLog is a Decision Support System (DSS) provided as a Software as a Service (SaaS) specifically developed for the document management traceability, optimization and analysis. The first version of eLog allowed business managers to select the required report starting from a predefined list, and, in case, to export the result in spreadsheet format. This paper outlines the improvements reached by the eLog new version, which allows to move from a transactional system to a multidimensional approach. With the new version, the user can dynamically create, save and share with other users his own reports. The MDXGenerator component acts as the middle layer between the framework and the OLAP engine, converting the reports requested by the business manager through a GUI, into MDX queries executed on the OLAP engine. In addition, eLog provides a simple and intuitive meta-language that defines new indicators or new aggregate functions for custom queries. Another functionality implemented in eLog is the selection, among all the cubes available in the data warehouse, of the optimal cube for the execution of the query. Using eLog, the decision maker is autonomous in the process of creating and managing new analysis, based on the entire set of measures and dimensions available in the underneath fact tables.

The paper is structured as follow. Section II presents a general overview of eLog. Section III contains the eLog case study we designed to explain how our system works. Section IV describes the Data Analysis service of eLog and Section...
V depicts the conceptualization of schemas and queries we designed in order to decouple the GUI and the back-end. Section VI describes the MDXGenerator. Section VII gives the meta-language definition, Section VIII presents the algorithm modelled for the optimal cube selection and Section IX shows some tests we performed on the system. Section X depicts some related work, whereas conclusion and possible future evolvements of eLog are depicted in Section XI.

II. OVERVIEW OF THE eLOG SYSTEM

eBilling is a company that handles the entire life cycle of the document delivery process, from the generation of files starting from the raw data supplied by the client, until the electronic or hard copy delivery of the documents to the end user.

The eLog application is one of the modules embedded within the BCPortal® developed by eBilling to meet the demands of the production process and document generation traceability.

eLog controls the production of spools and documents, the sending of emails or faxes, the communication to the printing providers and the delivery of printed documents. The framework stores low level granularity information related to operative production processes, i.e. documents transfer between servers and enterprises, in order to calculate the Key Performance Indicators (KPIs).

KPIs make possible to assess the adequacy of the expected and contractually specified services with respect to the Service Level Agreement (SLA). eLog traces the evolution and generates table reports and statistical graphics per type of flow, thus making it possible to keep track of the business communication process performance.

As shown in Figure 1, data coming from the client’s document generation system is traced during three stages of the document production process: receiving, formatting and sending of documents. Through eLog, customers have vision of both the internal and external document production processes, and all the estimated and actual events (pdf file generation, printing, delivery etc.).

The new version of eLog has been developed to meet the emerging needs of customer companies’ business managers. The requirements that the new application had to meet can be summarized as follows:

1) Simple GUI - users can easily and effectively formulate multidimensional queries since all the complexities are masked by the underlying system.

2) Personalization - personalization for eLog customers is a key issue, as different companies would like to define their own indicators. The meta-language has been specifically designed for users that lacks of technical knowledge on query languages in order to facilitate the definition of even complex expressions.

Further improvements introduced by the eLog’s new version are:

1) Technological improvement - the previous version was a web application developed in Apache Struts which allows users to extract static reports from the database by a web form. The database system was queried through SQL or OQL statements and the result sets were exported as excel files and therefore analyzed locally. The new version, instead, offers to customers an on-line service with an easy-to-use interface implemented in Adobe Flex. Users can take advantage of a portable (it requires just a browser with Flash player) and efficient (all the data is maintained remotely, hence issues such as occasionally data loss are avoided) tool.

2) Data conceptualization - in order to conceptualise the description of data and queries and to decouple the GUI and the OLAP server, a set of XSD schema has been defined for the description of cubes, attributes and user generated queries. The XSD schema also provides the ability to share or exchange custom attributes or queries among users.

3) Modular and reusable software architecture - MDXGenerator is a stand-alone java library, OLAP engine independent. Beside being used in the context of eLog, MDX-Generator can be plugged into any other frameworks compliant with the data conceptualization implemented in the XSDs.

Figure 2 summarizes the architecture of eLog. The Core
component provides general services related to eLog project setup, search services and monitoring. The Web Application component communicates with the Core component and provides high level services to the user. The Analysis Core component communicates with the ROLAP component and manipulates the results returned from multidimensional queries, in order to improve the data rendering operations of the Data Analysis Web Application component. The Analysis Core component permits to analyze reports expressed through the GUI and converts them into multidimensional queries. To accomplish this task, it is supported by the MDXGenerator: a stand-alone library that provides functionalities for automatic multidimensional queries generation starting from user inserted configurations. The Data Analysis Web Application component is responsible for presenting and validating the analysis reports. The DB component is responsible for processing and maintaining large amounts of data acquired from the outside world, applying rules of extraction, transformation, loading and data aggregation in order to satisfy the Data Analysis features requirements. For the ETL process, we take advantage from our previous work on semantic ETL [1].

III. eLog CASE STUDY

In order to explain in detail the features provided by the MDXGenerator and the meta-language, firstly it is useful to introduce a simple explicative Dimensional Fact Model (DFM [7]); we will use this fact model to design our examples appearing in the next sections. The model has been taken from the real data warehouse model of eLog. In particular we took out 5 dimensions over 25 and 3 measures over 19.

Our example represents the monitoring activity of the SLA related to some mail pieces, called missives. The conceptual representation of our example has been depicted in Figure 3. The measures we took into consideration are the total number of missives in SLA (SumMovINSLA), and the total and average number of missives for which the SLA has been expired (respectively SumMovOUTSLA and AvgMovOUTSLA). The fact has been modeled using five different dimensions, namely: Time, Geographic, Business Process, Society and Client. The last three dimensions are simple as they contain just one element in the hierarchy, while, in the Time dimension we represented the day of interest, the month, the quarter, the semester and the year. Similarly, the Geographic dimension has been divided in city, county, region and state. Two descriptive attributes, cod and type, have been, respectively, inserted in the Client and Business Process dimensions, in order to specify a property upon the related hierarchy.

IV. eLog DATA ANALYSIS

The eLog Data Analysis provides trend analysis functionalities. The web application interface (see Figure 4) contains the list of the already created reports on the left part. In the middle it is visualized the pivot table, which is rendered once the query result is available. The right part of the interface contains on the top all the attributes (predefined and custom) that are available for selection, and, on the bottom, the four quadrants the user can fills with attributes in order to create the desired query.

The user can create his own reports starting from a list of attributes. Attributes can be moved on one of the four quadrants: filters, columns, rows and values. Attributes disposed on the values quadrant will be aggregated using one of the available aggregate functions. Each attribute placed on the rows defines the level of aggregation and it can also define specific properties. For example the attribute Client contains the property cod (see Figure 3) that can be selected by the user if he decides to visualize this information.

Some predefined KPIs are provided by eLog: the user has the opportunity to adopt these predefined attributes or to define new indicators, called custom attributes. A custom attribute is the result of a meta-language expression and its operands are attributes already present in the analysis domain or alphanumeric constants.

In order to highlight the occurrence of certain events, the user may define formatting rules over the cells in the values area. The user can specify a boolean expression and the action to be taken if the expression is evaluated true. For example, a cell background can be changed in red if the contained number is below a certain threshold. Moreover, the web interface provides advanced querying features, e.g. to select the top-k values or to filter results by measure values.

The Data Analysis provides also save and load functionalities which allow to store the created custom attributes, the formatting settings, and the entire pivot table configuration if desired. We define pivot table configuration as the output query resulting from the user interaction with the data analysis interface, i.e. the selected attributes for each of the four quadrants and the list of predefined and custom attributes.
V. XSDs for High Level Conceptualization

Each pivot configuration is codified in a set of XML files that will be the input of the MDXGenerator component. Each XML configuration conforms to a high level conceptualization implemented in XML Schema (XSD) [15]. This approach allows us to maintain a logical separation between the interface used to define pivot configurations and the system that actually performs the queries on the data warehouse. On one side, all the information of the data warehouse schema are not encoded directly in the business logic (i.e. the web interface is not tight to a particular data model), on the other side, the backend system is lightly coupled with the web interface, since the only communication means are the pivot configurations that are compliant with the XSDs. For these reasons, the MDXGenerator together with the XSDs’ conceptualizations, can be seen as a bridge between the user interface and the OLAP server. Moreover, since the MDXGenerator receives as input just pivot configurations in XML, it can be considered not only in the context of eLog, but also as an external library that can be used from other applications.

A generic pivot configuration can be described by means of nine XSD schemas, each of which models a particular part of the pivot configuration. eLogPivot is the root element and it contains the general structure of every pivot configuration. This general structure contains six subsections: eLogCube specifies the cubes upon which the analysis should be executed; eLogData contains all the attributes that can be used inside the configuration; eLogFilters, eLogColumns, eLogRows and eLogAggregateValues describe specific properties for attributes placed on quadrants. Inside eLogData two types of attributes can be specified: predefined attributes and custom attribute, respectively described by eLogPredefinedAttribute and eLogCustomAttribute.

VI. MDX Generator

MDXGenerator is the java library able to generate a MDX query starting from a XML pivot configuration. MDXGenerator is also responsible for the syntactic verification of the pivot configuration and the semantic correctness of the MDX expression.

The syntactic verification examines the input pivot configuration so that it contains all the attributes and the information needed for the correct creation of the query, and that, in addition, the information conforms to the XSD schemas described in Section V.

Once the input has been verified to be syntactically well-formed, it is divided in subsections of interest and parsed. During the parsing phase, six different structures are filled with the elements extracted from the pivot configuration. The first structure, called Data Cube, contains all the cubes that should be taken into consideration for the query execution, and hence contained in the eLogCube section of the input configuration. Section VIII describes in details how the Data Analysis component chooses the more suitable cube for execution on the OLAP server among the provided list. Another structure, namely Data Attributes, is allocated for the attributes - predefined or custom - directly involved in the query. The system is therefore able to work just on a subset of all the attributes contained in the eLogData section of the configuration.
For every custom attribute instantiated, the system performs on-the-fly translation of the meta-language expression into MDX. After that, the semantic correctness of the MDX expression is checked. The MDXGenerator verifies that the elements contained in the configuration can be mapped into the data schema on the OLAP server.

During the meta-language translation, we need to define a solve order sequence for all the selected custom attributes. This sequence is automatically computed by using the position in the quadrant chosen by the user. For instance, if the column quadrant contains two custom attributes, the attribute that has been firstly selected by the user, will be solved as first, meanwhile the second attribute will be solved after the first attribute. The other structures are used to store the elements contained in the quadrant, hence one structure for each quadrant.

Once the whole configuration has been parsed, MDXGenerator starts the composition of the final query: this is a Java object containing three different strings, one for each section of a MDX query, namely SELECT, FROM, and WHERE. The FROM string contains a list of cubes; it is the AnalysisCore’s task to chose the best cube and consequently the reassembling of the query (see section VIII for a detailed description). The SELECT and WHERE strings are generated starting from the structures already filled, wrapping the attributes information with MDX language constructs.

Listing 1 shows a simple example of pivot configuration (with the omission of some specific details), while Listing 2 shows the MDX query generated by the library. As pointed out in Section 4, users can specify conditions and actions to be performed when conditions are verified.

In Listing 1, the condition “SumMsvINSLA>1000” on the aggregate value SumMsvINSLA has been set. The condition is verified on each single cell, once the condition occurs the format action is executed. In listing 1, this results in a red colored cell. Other similar features like for instance, the selection of top-k values, or the filtering by measure values, are managed directly by the web interface.

Listing 2. MDX Query resulting from the pivot configuration of listing 1

```
A. MDX Generator Repository
```

Users can create a multitude of different reports using our system, however we expect that, once a new report is created, users continue to adopt it over time. Therefore not only the Data Analysis service supports the features of saving
and loading reports, but also MDXGenerator implements a repository where already parsed configurations can be stored, thus decreasing the query generation time. The repository basically works as a cache memory: if a pivot configuration is stored in memory, there is no need to resolve the configuration, in fact the data can be retrieved directly from the cache. In order to maximize the reuse of cached information, we store each of the six data structures in the repository independently. If two pivot configurations differs just for a quadrant, we can use the already filled data structures for all the configuration sections that do not differ. MDXGenerator allows the reuse of the repository among different sessions thanks to its ability to save and load the cache in persistent memory. From the MDXGenerator configuration file, it can be specified if the repository must be made persistent, and how (so far, the repository can be saved on the file system or creating an entry in a database using JDBC).

VII. METALANGUAGE AND CUSTOM ATTRIBUTES

The system allows users to combine predefined attributes in order to analyze some specific event, but the user may also create a custom attribute, specifying an expression which defines the new attribute. Since MDX can be considered too expressive and complex for the target users the application has been developed for, we designed a meta-language specifically tailored for our domain. This meta-language is a simplified version of the MDX language both in semantics and syntax.

In the next section we describe the meta-language, while in Section VII-B we briefly show how custom attributes can be created through a couple of examples.

A. Meta-language Description

The main features introduced by the meta-languages are:
(a) a list of functions to combine and manage attributes values;
(b) the hierarchy management: it is possible to create new members inside a hierarchy or new hierarchies embracing a subset of members of a hierarchy.

The following elements have been introduced: Number, Constant, MDX Entity, Expression. We define as Number any element containing just numbers, while Constant is an element containing any sequence of characters and (potentially) numbers. A MDX Entity is any member belonging to a dimension inside a cube: it is specified by using the MDX notation ([Entity] for example). An Expression can be of three types: Arithmetic Expression, composed by two arithmetic operands and one of the usual arithmetic operators (+, -, *, /); Comparison Expression where the two operands are arithmetic and the permitted operators are the relational ones (>, <, =, >=, <= e <>); Logic Expression, described by two or more logic operands separated by one of the “AND”, “OR”, “XOR” and “NOT” operators. The strings “true”, “false”, and the Search Function are considered as logic operands.

The list of Functions we implemented in our meta-language in order to combine and manage values within the measures (and dimensions) is the following: format, if, concat, contains, current, previous, following.

1) Format Function: The format function accepts as parameters the element to be formatted and the formatting string, and returns as a result the string containing the element in the new format. The elements that can be passed as input are: an MDX Entity, an Expression or another Function. The formatting string is used to specify how the input element should be represented, and is composed by an optional value character followed by any sequence of numeric or alphabetic character. An example of format function and related MDX translation is reported in Listing 3 and 4.

```
FORMAT ([Price]; '"$#,###.0")
```
Listing 3. Example of Format Function

```
WITH MEMBER [Measures].[Price in Dollars] AS [Measures].[Price], FORMAT_STRING = '"$#,###.0")
```
Listing 4. MDX translation of Listing 3

2) If Function: In the if function three parameters have to be specified: the condition that must be assessed, the value that has to be returned if the condition is true and the value that has to be returned otherwise. The accepted condition parameters are a Comparison or a Logic Expression. The permitted result elements are: Constant, Arithmetic Operand, Expression and Function. An example of if function and the related translation in MDX is reported in Listing 5 and 6.

```
IF ([SumMsvINSLA] >= 0; IF ([SumMsvINSLA] <= 100; '"Low"; '"High") ; '"Lost")
```
Listing 5. Example of If Function

```
WITH MEMBER [Measures].[SumMsvINSLA Margin] AS IF ([Measures].[SumMsvINSLA] >= 0; IF ([Measures].[SumMsvINSLA] <= 1; '"Low"; '"High") ; '"Lost")
```
Listing 6. MDX translation of Listing 5

3) Concat Function: The concat function can be used to concatenate two input elements in one string. The elements allowed as input of the function are: Constant, MDX Entity, another concat function or current, previous and following functions. Listing 7 and 8 show an example of use of a concat function and its MDX translation.

```
CONCAT ([Time].[Month]; CONCAT ("\"\"; [Time].[Day])
```
Listing 7. Example of Concat Function

```
WITH MEMBER [Time].[Complete Data] AS [Time].[Month] || "\"\" || [Time].[Day]
```
Listing 8. MDX translation of Listing 7

4) Contains Function: The contain function searches into the first parameter the value of the second parameter: if the value is present the function returns “true”, “false” otherwise. Therefore the contain function accepts two parameters: the element where the research must be accomplished and the value of interest. The input can be a Constant, an MDX Entity, current, previous or following functions. Listing 9
and 10 show an example and the related MDX translation of a contain function.

\[
\text{IF (CONTAIN ((File name); "zip"); "Compressed"; "Not compressed")}
\]

Listing 9. Example of Contain Function

\[
\text{WITH MEMBER [File].[Compression] AS IIF (Cbool (InStr (Str ([File].[File name]), "zip")), "Compressed", "Not compressed")}
\]

Listing 10. MDX translation of Listing 9

5) Current, Previous and Following Functions: These functions are the equivalent of CurrentMember, PrevMember and NextMember in MDX. current/previous/following considers the current/previous/following member during the iteration inside a hierarchy or dimension. Listing 11 and 12 describe an example of a previous function.

\[
\text{PREVIOUS ([Sales].[2008])}
\]

Listing 11. Example of Previous Function

\[
\text{WITH MEMBER [Sales].[2007] AS [Sales].[2008].PrevMember}
\]

Listing 12. MDX translation of Listing 11

6) Create Hierarchy Function: To create and manage new hierarchies, a specific function has been introduced. This function creates a new hierarchy joining members from an existing hierarchy. The created hierarchy is therefore a subset of the original. The input parameters are a set of members belonging to the same hierarchy. An example of this function and related translation are reported in Listing 13 and 14.

\[
\text{CREATEHIERARCHY ([Coca-Cola]; [Red Bull]; [Pepsi])}
\]

Listing 13. Example of Create Hierarchy Function

\[
\text{WITH SET [Drink Companies] AS \{[Society].[Coca-Cola] , [Society].[Red Bull] , [Society].[Pepsi]\}}
\]

Listing 14. MDX translation of Listing 13

For a complete EBNF definition of the meta-language we developed, refer to the appendix.

B. Examples

Basically, the meta-language can be used to create two types of MDX elements: new members inside a hierarchy and new hierarchies embracing a subset of elements of a hierarchy already present in the cube. We introduce two simple examples.

In the first example, we want to compute for every client and for the cities of Naples and Rome, the number of missives in SLA and the total number of missives that are both in SLA and outside the SLA. This for the first and the second quarter of each year. Considering that the system does not contain an attribute specifying both the missive in SLA and out SLA, we have to define a new custom attribute expressing the sum between the predefined attributes \text{SumMsvINSLA} and \text{SumMsvOUTSLA}. To create the new custom member, starting from the GUI shown in Figure 5, the user enters in the field Name, the desired name for the new attribute (SumMsvIN/OUT), in Type the data type of the new attribute (Categorical Nominal, Categorical Ordinal, Numeric or Temporal), and in Analysis Domain the domain the new attribute belongs to.

```
<CustomAttributeData AttributeName="SumMsvIN/OUT"
   UserId="admin" AttributeType="N"
   AnalysisDomain="Production" SolveOrder="1"/>
<Expression Definition="[SumMsvINSLA]+[SumMsvOUTSLA]" Localization="en_US"/>
</CustomAttributeData>
```

Listing 15. Custom attribute definition for the first example

Further, the user is facilitate in the task of creating a meta-language expression defining the meaning of the custom attribute by the “Attribute Library” and the “Operator” lists that appear in the right part of the interface. The XML describing the new member SumMsvIN/OUT and compliant to the XSD depicted in Section V is reported in Listing 15.

For a custom attribute, the XML contains in AttributeName the name of the new member or hierarchy, in UserId the id of the user that creates it, in AttributeType the type of attribute (N for numeric, T for Temporal, CO for Categorical Ordinal and CN for Categorical Nominal), in AnalysisDomain the domain the new attribute belongs to, in SolveOrder the order in which the custom element must be solved by the OLAP server, and in DimensionName the dimension that contains the custom attribute (measures by default).

\[
\text{WITH MEMBER [Measures].[SumMsvIN/OUT] AS [Measures].[SumMsvINSLA]+[Measures].[SumMsvOUTSLA], SOLVE_ORDER=3}
\]

MEMBER [Client].[Total Client] AS Sum (EXCEPT([[Client].[L DES_CLIENT].Members], [Client].[All].CurrentMember))

MEMBER [Geographic].[Total Geographic] AS Sum ([[Geographic].[City].[Rome], [Geographic].[City].[Naples]])
SELECT NON EMPTY { Union ([Measures], [SumMsvINSLA], Measures, [SumMsvIN/OUT]) ON COLUMNS, NON EMPTY [ Union (Except ([Client], [1_DES_CLIENT]), Members), [Client], [ (All) ], CurrentMember ] AS [ Client ], [ Total Client ] } * [ [ Geographic ], [ City ], [ Rome ], [ Geographic ], [ City ], [ Naples ], [ Geographic ], [ Total Geographic ] ] ON ROWS FROM CUBE T1 WHERE [ ([Acq_Time], [Quarter], 11] , [Acq_Time], [Quarter], 12]}

Listing 16. MDX query generated from the pivot configuration of Table I.

Since the meta-language has been developed in multiple localizations (Italian and English so far) in order to make users more familiar with it, each custom attribute contains one or more Expression, one for each localization. In each expression element, the attribute Definition contains the meta-language expression defining the new element, meanwhile Localization contains the localization (en_US for English, it_IT for Italian). The pivot configuration from which the query will be generated, is described in Table I. The MDX query produced by the MDXGenerator is shown in Listing 16.

The second example shows how new hierarchies can be created. Let us suppose we want to know the number of misses in SLA and the average number of misses out SLA. The results must be grouped by some particular business process and by certain society of interest.

The XML containing the definition of the new hierarchies is contained in Listing 17.

As it can be noticed, in the Digital hierarchy, the property TotalWithoutOverlapping is true. This property (false by default) specifies how the repeated values must be treated in the total: if the property is set to true, the repeated values are counted just one time, in the other case the total considers also duplicated values. The pivot configuration is shown in Table II, meanwhile the related MDX query is contained in Listing 18.

Listing 17. Custom hierarchies definition for the second example.
Table III: DBMS Configuration

<table>
<thead>
<tr>
<th>DBMS_Vendor</th>
<th>Oracle</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBMS_Distribution</td>
<td>Oracle 10g Standard Edition</td>
</tr>
<tr>
<td>DBMS_HostSO</td>
<td>Red Hat Enterprise Linux Server Release 5.3</td>
</tr>
<tr>
<td>DBMS_HostRAM</td>
<td>16Gb</td>
</tr>
<tr>
<td>DBMS_HostDisk</td>
<td>2 Disks SAS RAID1 10.000 rpm</td>
</tr>
<tr>
<td>DBMS_HostCPU</td>
<td>2 Processors 4 Core 3Ghz</td>
</tr>
<tr>
<td>DBMS_HostSwap</td>
<td>16Gb</td>
</tr>
</tbody>
</table>

Table IV: Application Server Configuration

<table>
<thead>
<tr>
<th>AS_Vendor</th>
<th>JBOSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS_Version</td>
<td>JBOSS 4.2.0</td>
</tr>
<tr>
<td>AS_JDK</td>
<td>1.5 (32bit)</td>
</tr>
<tr>
<td>AS_HeapSize</td>
<td>2Gb</td>
</tr>
<tr>
<td>AS_HostRAM</td>
<td>16Gb</td>
</tr>
<tr>
<td>DBMS_HostDisk</td>
<td>2 Disks SAS RAID1 10.000 rpm</td>
</tr>
<tr>
<td>DBMS_HostCPU</td>
<td>2 Processors 4 Core 3Ghz</td>
</tr>
<tr>
<td>DBMS_HostSwap</td>
<td>16Gb</td>
</tr>
</tbody>
</table>

Table V: Client Configuration

<table>
<thead>
<tr>
<th>Client_RAM</th>
<th>3Gb DDR2 400Mhz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client_Disk</td>
<td>Dell SATA2 5400rpm</td>
</tr>
<tr>
<td>Client_CPU</td>
<td>Intel Core 2 Duo P9600@2.66 Ghz</td>
</tr>
<tr>
<td>Client_Network</td>
<td>Ethernet 1Gigabit</td>
</tr>
</tbody>
</table>

Listing 18. MDX query generated from the pivot configuration of Table II

```
FROM CUBE_T1
ON ROWS
```

VIII. CUBE SELECTION

The MDXGenerator only deals with the generation of the query. In case more that one cube is specified in the query, it does not investigate which cube is the more suitable for the query execution. For this reason, the query object created by the MDXGenerator contains in the FROM field all the cubes that might be used. In order to determine the cube that best fit the query, we developed an algorithm that chooses among all the cubes contained in the pivot configuration the optimal one for the execution of the query.

The algorithm has been designed to be as general as possible and completely independent on the input cubes. In such a way, it is not mandatory that the input cubes share the same semantic structure. In eLog each of the input cubes takes into consideration different portion of the schema and therefore it may contain data at different granularity. For example, the phase-level cube contains data at an higher granularity than the contract-level cube (since a contract collects more phases).

In our algorithm we consider a schema $S(C, D)$, where $C$ is the set containing all the cubes and $D$ the set of all dimensions. Given a query $Q$ derived from the pivot table configuration, we define $Q_{C'}$ as the query that involves the non empty set $C'$ of hierarchical levels of dimensions, where $C' \subseteq C$ and $C$ is the set of all the distinct levels belonging to at least one dimension contained in the set $D$. We denote $C_{l'}[l]$ as the not empty subset of $C$ containing the cubes where the dimension of the level $l \in C$ is used. We denote $|c|$ as the number of dimensions in the cube $c$. In addition, among all the cubes, we consider $c^0$ as the cube containing all the levels with lowest granularity.

```
Input: C, Q_{C'}
Output: c \in C
begin
  C' = \cap_{l \in C} C_{l'}[l]
  if |C'| = 0 then c = c^0
  else
    if |C'| = 1 then c = c^1
    else
      c = c^* where c^* \in C', \exists c' \in C', c' \neq c^*, |c'| < |c^*|
end
```

Listing 19. Algorithm for the selection of the cube starting from a pivot table configuration

Given the query $Q_{C'}$, we define as optimal cube, $c \in C$, the one that contains all the dimensions of the levels $C'$ involved in the query and has the minimal cardinality $|c|$. Listing 19 summarizes our algorithm.

IX. TEST

In order to evaluate the performance of the new version of eLog, we arranged preliminary tests on the systems. Analyzing real data gather from the first version of eLog a use case has been created. On this use case, we generated 15 typical queries, ordered by complexity, i.e. by the number of dimension involved (from a minimum of 3 to a maximum of 8 different dimensions). Table VI shows the queries summary, containing the number of dimensions used, the total amount of data returned for each query request and a brief description of the query. Even if our data warehouse consists of 25 dimensions, we did not adopt more than 8 dimensions per query, since it has been observed that a report containing such number of dimensions is enough complex for a typical pivot analysis. The number of facts we considered for our test are 8,974,662, and are related to 8 months of eLog’s data production. We used Btree index for the foreign keys on the fact table. We simulated the concurrent access for up to 25 users. The software and hardware configurations we adopted for the DBMS, the Application Server and the Client are reported in Tables III, IV and V.

We made three different tests. In the first test, we measured
Among the commercial tools landscape, Microsoft Excel PivotTable provides an expressive interface but does not support many dynamic analysis functionalities. Pivot tables are commonly criticized for disgraceful handling of large data sets and inefficiency in solving non-trivial analytical tasks, such as recognizing patterns, discovering trends, identifying outliers, etc.

State-of-the-art commercial OLAP tools, such as Business Objects, Cognos BI, Tableau and Target, enhance the pivot table view by providing very flexible data manipulation capabilities, often as vendors proprietary visualizations. A most important system, called Polaris [14], extends the Pivot Table interface by offering a combination of a variety of displays and tools for visual specification of analysis tasks. Polaris is the predecessor of a commercial product called Tableau Software [2]. ProClarity was the first to enhance business intelligence with Decomposition Trees [12] for visual node-by-node disaggregation of data cubes. XMLA enriches the idea of hierarchical disaggregation by arranging the decomposed subtotals of each parent value into a nested chart (Bar- and Pie-Chart Trees) in its Report Portal OLAP client [16]. Visual Insights has developed a family of tools, called ADVIZOR, with an intuitive framework for parallel exploration of multiple measures [5]. Each of these systems adopt a proprietary data model and query language, while our framework uses the multidimensional model [7] and the MDX query language.

While most vendors tend to limit the scope of supported visual layouts to popular and proven ones, researchers propose to employ novel visualization techniques to take full advantage of multidimensional and hierarchical properties of data. [10] concentrates on the problem of loosing the aggregates computed at preceding query steps while changing the level of detail and propose to use hierarchical layouts for capturing the results of multiple decompositions within the same display using the Enhanced Decomposition Tree technique. An advanced exploration framework for OLAP based on coordinated views of dimension hierarchies is proposed in [3]. Each dimension hierarchy, with qualifying fact entries attached as the bottom-level nodes, is presented using a space-filling nested tree layout. Drilling-down and rolling-up is performed implicitly by zooming within each dimension view. A new visual interactive exploration technique for an analysis of multidimensional databases is proposed in [6]: users can gain both overviews and refine views on any particular region of interest of data cubes through the combination of interactive tools and navigational functions such as drilling down, rolling up, and slicing.

Our approach overcomes the limitations of conventional interfaces by abstracting various visualization options into a common presentation model and providing algorithms implemented into MDXGenerator for mapping user interactions (providing MDXGenerator for mapping user interactions) as well as mapping query results to a specified visual layout.

X. RELATED WORK

Nowadays, managers and decision makers wish to have a simple and powerful tool to perform data analysis. Many OLAP servers [4] allow the definition of queries among multidimensional databases by means of languages, such as MDX, that are very effective but hard to be used by business managers. On the other side, commercial tools for semi-static reporting provide a higher usability but lack in:

1) consistency with multidimensional operations,
2) proved theoretical foundations,
3) a middle layer implementing a conceptual view over the multidimensional element stored underneath and
4) analysis coherence [13].

The time the MDXGenerator takes to elaborate each of the 15 queries without exploiting the caching functionality. In the second test, instead, we measured the performance of the MDXGenerator for the same set of queries but exploiting also the cache. Figure 6 represents the performance comparison between these two tests. As can be easily notice, exploiting the cache outperforms by 7 times the performance of the MDXGenerator.

In the third test, we measured the total response time of the system, starting from the time the user delivers the request, and ending when the client has completed the data loading from the server (hence taking into consideration also the MDXGenerator elaboration time using the cache). All the requests are submitted in parallel by all the 25 users and the response time, charted in Figure 7, is the average of the users response time. Each query has been executed without using the caching mechanism of the OLAP server.

![Average Query Generation Time](image1.png)

**Fig. 6.** Performance comparison between the MDX Generator with and without the cache functionality

![Average Response Time](image2.png)

**Fig. 7.** Response time of the system

**Insights** has developed a family of tools, called ADVIZOR, with an intuitive framework for parallel exploration of multiple measures [5]. Each of these systems adopt a proprietary data model and query language, while our framework uses the multidimensional model [7] and the MDX query language.
<table>
<thead>
<tr>
<th>Query</th>
<th>Dim.</th>
<th>Ret. Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>9758B</td>
<td>Number of completed, expected, unfinished missives, in SLA and out SLA missives, grouped by BILLING, OUTBOUND, RESEND ELIBRARY projects, for the customer MEDIASET and related to the acquisition time 1st quarter 2010.</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>11136B</td>
<td>Number of completed, expected, unfinished missives, in SLA and out SLA missives, grouped by all projects related to the customer H3G, and to the acquisition time 1st and 2nd quarter 2010.</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>29290B</td>
<td>Number of completed, expected, unfinished missives, in SLA and out SLA missives, grouped by EMISSION and ACQUISITION LOT, received in the fist quarter 2010, and belonging to the project BILLS POST CONSUMER of the customer H3G.</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>81576B</td>
<td>Number of completed, expected, unfinished missives, in SLA and out SLA missives, grouped by PRICE DESTINATION and AREA, received in the 1st quarter 2010, belonging to the custom H3G.</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>10710248B</td>
<td>Number of completed, expected, unfinished missives, in SLA and out SLA missives, grouped by PRICE DESTINATION and CAP, received in the 1st quarter 2010, belonging to the customer H3G.</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>34240B</td>
<td>Number of completed missives, partitioned by SLA range, grouped by ACQUISITION LOT code, received in the 1st quarter 2010 and belonging to the customer H3G.</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>57435B</td>
<td>Number of completed missives, partitioned by SLA range, grouped by EMISSION, ACQUISITION LOT and WORKFLOW, related to the acquisition time 2010 and belonging to the customer SKY.</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>298874B</td>
<td>Number of missives, partitioned by SLA range, grouped by EMISSION, ACQUISITION LOT and WORKFLOW, related to the acquisition time June 2010 and belonging to the customer FASTWEB.</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>189957B</td>
<td>Number of completed missives, partitioned by SLA range, grouped by EMISSION, ACQUISITION LOT and WORKFLOW, related to the acquisition time June 2010 and belonging to the customer FASTWEB.</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
<td>14467B</td>
<td>Number of completed missives, partitioned by delivery state, grouped by PRICE DESTINATION AM-CP-EU and weight range, related to the acquisition lots received in the 3rd quarter 2010 and belonging to the customer FASTWEB.</td>
</tr>
<tr>
<td>11</td>
<td>6</td>
<td>2249665B</td>
<td>Number of completed missives, partitioned by delivery state, grouped by PRICE DESTINATION AM-CP-EU, related to the acquisition lots received in the 3rd quarter 2010 and belonging to the customer FASTWEB.</td>
</tr>
<tr>
<td>12</td>
<td>7</td>
<td>2865182B</td>
<td>Number of completed missives, partitioned by delivery state, grouped by PRICE DESTINATION AM-CP-EU, AREA and CAP, related to the lots received in the 3rd quarter 2010 and belonging to the customer FASTWEB.</td>
</tr>
<tr>
<td>13</td>
<td>6</td>
<td>17033B</td>
<td>Number of completed missives, partitioned by delivery state, grouped by PRICE DESTINATION AM-CP-EU and AREA, related to the lots received in the 2nd quarter 2010 and belonging to the customer MEDIASET.</td>
</tr>
<tr>
<td>14</td>
<td>6</td>
<td>77427B</td>
<td>Number of completed missives, partitioned by delivery state and SLA range, grouped by PRICE DESTINATION AM-CP-EU and AREA, related to the acquisition lots received in the 1st quarter 2010 and belonging to the customer H3G.</td>
</tr>
<tr>
<td>15</td>
<td>8</td>
<td>8375465B</td>
<td>Number of completed missives, partitioned by delivery state, grouped by PRICE DESTINATION AM-CP-EU, AREA and CAP, related to the lots received in the 3rd quarter 2010 and belonging to the customer FASTWEB.</td>
</tr>
</tbody>
</table>

XI. CONCLUSION

We presented eLog: the business intelligence solution of eBilling for document management traceability, optimization and analysis. We outlined the improvements reached by the eLog new version, which allows to move from a transactional system to a multidimensional approach. eLog exploits the functionalities of the MDXGenerator library and the data model conceptualization provided by XSD schemas. MDXGenerator acts as the middle layer between the framework and the OLAP engine, converting the reports requested by the business manager through a GUI, into MDX queries executed on the OLAP engine. In order to customize data analysis indicators, eLog provides a simple and intuitive meta-language for unskilled users. Moreover, to optimize the MDX query execution on the OLAP server, we developed an algorithm for the optimal cube identification among all the cubes available in the data warehouse. The eLog architecture has been implemented in Adobe Flex for the web interface and in Java for the MDXGenerator. We plan to build a new version of MDXGenerator to extend the compliance of MDX language also for Microsoft SQL Server (so far, it is compliant with Mondrian technology).

The preliminary tests of eLog system produced promising results: the time needed to process even complex queries is acceptable. We plan more experiments on the applicative use of eLog system in the next future.

Starting from the assumption that MDX queries can be written in several ways maintaining the same semantics and obtaining the same result sets (e.g. operators like except, sum,
intersect are interchangeable) we will investigate, in the future, how MDXGenerator can be enhanced in order to optimize the query generation process, following the approach presented in [8].

**APPENDIX**

(*) TOKENS DECLARATION *)

(*) CHAR TOKENS *)
letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
digit = "0123456789";
currency = "$";
separater = ",

(*) OPERATOR TOKENS *)
compOperator = "\<" | "\>" | "\=" | "\<\=" | "\>=" | "\<\>";
logicOperator = "\&\&" | "||" | "\^";
signOperator = "\+-";
mulOperator = "\*" | "/";
not = "!";

(*) BOOLEAN OPERATOR TOKENS *)
booleanValue = "true" | "false";

(*) STRING AND NUMBERS TOKENS *)
string = { letter | digit } letter { letter | digit }.
constant = "" ( string | number ) "" | ( string | number ) ""
mdxEntity = "\[" string | number { " " string | number } ""]"
number = digit { digit } [ separator { digit } ];

(*) FUNCTION TOKENS *)
if = "IF";
format = "FORMAT";
concat = "CONCAT";
contain = "CONTAIN";
previous = "PREVIOUS";
following = "FOLLOWING";
createHierarchy = "CREATEHIERARCHY";

(*) META-LANGUAGE DEFINITION *)
GetMDX = Expression | Function | FormatFunction | AritmOperand | constant.

(*) Expressions *)
Expression = AritmExpression | BooleanExpression.

BooleanExpression = CompExpression | LogicExpression | not | InStrFunction.
CompExpression = ("\(\ AritmExpression \)\)" | aritmOperand | compOperator ("\(\ AritmExpression \)\)" | aritmOperand).
LogicExpression = { not } LogicFactor LogicOperator { not } LogicFactor.
LogicFactor = LogicOperand | ("\(\ CompExpression \)\)".
LogicOperand = booleanValues | InStrFunction.
AritmOperand = signOperator | mulOperator.

(*) FormatFunction *)

FormatFunction = format ("\(\ Expression \)\) | Function | mdxEntity") | ; | formatString ").

(* Remaining Functions *)
Function = IfFunction | ConcatFunction | GetMemberFunction.
GetMemberFunction = CurrentFunction | PrevMemberFunction | NextMemberFunction.
IfFunction = if ("\(\ BoolExpression \)\) | ("\(\ Expression \)\) | Function | AritmOperand | constant") | ; | ("\(\ Expression \)\) | Function | AritmOperand | constant")

CurrentFunction = current ("\(\ mdxEntity \)\)".
ConcatFunction = concat ("\(\ mdxEntity \)\) | constant | mdxEntity)
 ConcatFunction | GetMemberFunction") | ; | ("\(\ mdxEntity \)\) | constant | GetMemberFunction | GetMemberFunction")

PrevMemberFunction = previous ("\(\ mdxEntity \)"
NextMemberFunction = following ("\(\ mdxEntity \)"

CreateHierarchyFunction = createHierarchy ("\(\ mdxEntity \)\) | mdxEntity ["." mdxEntity | mdxEntity]]

**REFERENCES**


