OLAP Authentication and Authorization via Query Re-writing

Todd Eavis
Department of Computer Science
Concordia University
Montreal, Canada
Email: eavis@cs.concordia.ca

Ahmad Altamimi
Department of Computer Science
Concordia University
Montreal, Canada
Email: a_alta@cs.concordia.ca

***Abstract*—Online Analytical Processing (OLAP) has become an increasingly important and prevalent component of Decision Support Systems. OLAP is associated with a data model known as a cube, a multi-dimensional representation of the core measures and relationships within the associated organization. While numerous cube generation and processing algorithms have been presented in the literature, little effort has been made to address the unique security and authentication requirements of the model. In particular, the hierarchical nature of the cube allows users to bypass - either intentionally or unintentionally - partial constraints defined at alternate aggregation levels. In this paper, we present an authentication framework that builds upon an algebra designed specifically for OLAP domains. It is Object-Oriented in nature and uses query re-writing rules to ensure consistent data access across all levels of the conceptual model. The process is largely transparent to the user, though notification is provided in cases in which a subset of the original request is returned. We demonstrate the scope of our framework with a series of common OLAP queries. The end result is an intuitive but powerful approach to database authentication that is uniquely tailored to the OLAP domain.**

***Keywords*-Data warehouses; Data security; Query processing**

I. Introduction

Data warehousing (DW) and On-Line Analytical Processing (OLAP) play a pivotal role in modern organizations. Designed to facilitate the reporting and analysis required in decision making environments [1], OLAP builds upon a multi-dimensional data model that intuitively integrates the vast quantities of transactional level data collected by contemporary organizations. Ultimately, this data is used by managers and decision makers in order to extract and visualize broad patterns and trends that would otherwise not be obvious to the user.

One must note that while the data warehouse serves as a repository for all collected data, not all of its records should be universally accessible. Specifically, DW/OLAP systems almost always house confidential and sensitive data that must, by definition, be restricted to authorized users. The administrator of the warehouse is ultimately responsible for defining roles and privileges for each of the possible end users. In fact, a number of general warehouse security models have been proposed in the literature [2]–[5]. Several authors define frameworks that are likely too restrictive for production warehouses. For instance, Rosenthal et al. [4] discuss a security model based on *authorization views*. The views are created for specific users so as to allow access only to selected data. However, the administration of these views becomes quite complex when a security policy is added, changed, or removed. Moreover, complex roles can be difficult to implement in practice, and models of this type tend not to scale well with a large number of users. Conversely, other researchers have focused on the design process itself. For example, Fernndez-Medina et al. [2] propose a Unified Modeling Language (UML) profile for the definition of security constraints. Here, however, the physical implementation of the underlying authentication system remains undefined.

In this paper, we present an authentication model for OLAP environments that is based on a query rewriting technique. The model enforces distinct data security policies that, in turn, may be associated with user populations of arbitrary size. In short, our framework rewrites queries containing unauthorized data access to ensure that the user only receives the data that he/she is authorized to see. Rewriting is accomplished by adding or changing specific conditions within the query according to a set of concise but robust transformation rules. Because our methods specifically target the OLAP domain, the query rules are directly associated with the conceptual properties and elements of the OLAP data model itself. A primary advantage of this approach is that by manipulating the conceptual data model, we are able to apply query restrictions not only on direct access to OLAP elements, but also on certain forms of indirect access.

The remainder of this paper is organized as follows. In Section II, we present an overview of related work. Section III describes the core OLAP data model and associated algebra, and includes a discussion of the object-oriented query structure for which the proposed security model has been designed. The OLAP query rewriting model and its associated transformation rules are then presented in detail in Section IV. Final conclusions are offered in Section V.

II. Related work

As noted above, a number of security models that restrict warehouse access have been proposed in the literature, including those that focus strictly on the design process.

Extensions to the Unified Modeling Language to allow for the specification of multi-dimensional security constraints has been one approach that has been suggested [2]. In fact, a number of researchers have looked at similar techniques for setting access constraints at an early stage in the OLAP design process [6], [7]. Such models have great value of course, particularly if one has the option to create the warehouse from scratch. That being said, their focus is not on authentication algorithms per se, but rather on design methodologies that would most effectively use existing technologies.

In terms of true authentication models, several researchers have attempted to augment the core Database Management System (DBMS) with authorizations views [4], [8]. Typically, alternate views of data are defined for each distinct user or user group. The end result is often the generation of a large number of such views, all of which must be maintained manually by the system administrator. Clearly, this approach does not scale terribly well, and would be impractical in a huge, complex DW environment.

Query rewriting has also been explored in DBMS environments in a variety of ways, with search and optimization being common targets [9]. Beyond that, however, rewriting has also been utilized to provide fine grained access control in Relational databases [10]. Oracle's Virtual Private Database (VPD) [11], for example, limits access to row level data by appending a predicate clause to the user's SQL statement. Here, the security policy is encoded as policy functions defined for each table. These functions are used to return the predicate, which is then appended to the query. This process is done in a manner that is entirely transparent to the user. That is, whenever a user accesses a table that has a security policy, the policy function returns a predicate, which is appended to the user's query before it is executed.

In the Truman model [10], on the other hand, the database administrator defines a *parameterized* authorization view for each relation in the database. Note that parameterized views are normal views augmented with session-specific information, such as the user-id, location, or time. The query is modified transparently by substituting each relation in the query by the corresponding parameterized view to make sure that the user does not get to see anything more than his/her own view of the database. In this model, the user can also write queries on base relations by plugging in the values of session parameters such as user-id or time before the modified query is executed.

We note, however, that the mechanisms discussed above (e.g., Oracle's VPD) are not tailored specifically to the OLAP domain and, as such, either have limited ability to provide fine grained control of the elements in the conceptual OLAP data model or, at the very least, would make such constraints exceedingly tedious to define. Some commercial tools, such as Microsoft's Analysis Services [12], do in fact provide such mechanisms. However, even here, authentication controls are quite direct in that they must be explicitly associated with any and all affected elements of the model. This is in contrast to the work discussed in this paper, where the primary contribution is a query rewriting technique that transparently supports *indirect* authentication.

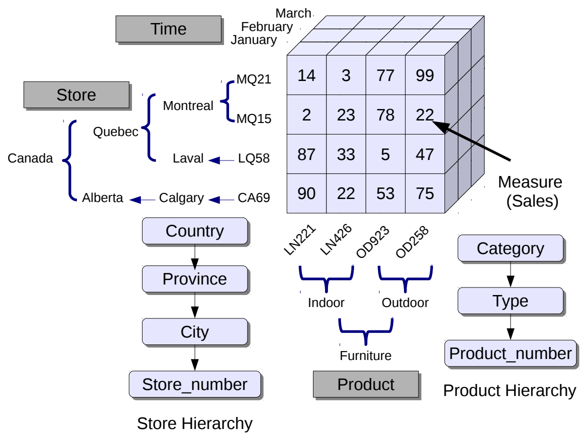

Figure 1. A simple three dimensional data cube

III. The Conceptual Data Model

We consider analytical environments to consist of one or more *data cubes*. Each cube is composed of a series of *d* dimensions — sometimes called *feature* attributes — and one or more *measures* [13]. The dimensions can be visualized as delimiting a *d*-dimensional hyper-cube, with each axis identifying the members of the parent dimension (e.g., the days of the year). Cell values, in turn, represent the aggregated measure (e.g., sum) of the associated members. Figure 1 provides an illustration of a very simple three dimensional cube on `Store`, `Time` and `Product`. Here, each unique combination of dimension members represents a unique aggregation on the measure. For example, we can see that Product OD923 was purchased 78 times at Store MQ15 in January (assuming a `Count` measure).

Note, as well, that each dimension is associated with a distinct aggregation hierarchy. Stores, for instance, are organized in `Country` $\rightarrow$ `Province` $\rightarrow$ `City` groupings. Referring again to Figure 1, we see that `Product Number` is the lowest or *base* level in the Product dimension. In practice, data is physically stored at the base level so as to support run-time aggregation to coarser hierarchy levels. Moreover, the attributes of each dimension are partially ordered by the dependency relation $\preceq$ into a dependency lattice [14]. For example, `Product Number` $\preceq$ `Type` $\preceq$ `Category` within the Product dimension. More formally, the dependency lattice is expressed in Definition 1.

Definition 1: A dimension hierarchy H_i of a dimension D_i, can be defined as $H_i = (L_0, L_1, \ldots, L_j)$ where L_0 is the lowest level and L_j is the highest. There is a functional dependency between L_{h-1} and L_h such that $L_{h-1} \preceq L_h$ where (0 ≤ h ≤ j).

Finally, we note that there are in fact many variations on the form of OLAP hierarchies [15] (e.g., symmetric, ragged, non-strict). Regardless of the form, however, traversal of these aggregation paths — typically referred to as *rollup and drill down* — is perhaps the single most common query form. It is also central to the techniques discussed in this paper.

A. Native Language Object Oriented OLAP Queries

The cube representation, as described above, is common to most OLAP query environments and represents the user's conceptual view of the data repository. That being said, it can be difficult to implement the data cube using standard relational tables alone and, even when this is possible, performance is usually sub-par as relational DBMSs have been optimized for transactional processing. As a result, most OLAP server products either extend conventional relational DBMSs or build on novel, domain specific indexes and algorithms.

In our own case, the authentication methods described in this paper are part of a larger project whose focus is to design, implement and optimize an OLAP-specific DBMS server. A key design target of this project is the integration of the conceptual cube model into the DBMS itself. This objective is accomplished, in part, by the introduction of an OLAP-specific algebra that identifies the core operations associated with the cube (SELECT, PROJECT, DRILL DOWN, ROLL UP, etc). In turn, these operations are accessible to the client side programmer by virtue of an Object Oriented API in which the elements of the cube (e.g., cells, dimensions, hierarchies) are represented in the *native* client language as simple OOP constructs. (We note that our prototype API uses Java but any contemporary OO language could be used). To the programmer, the cube and all of its data — which is physically stored on a remote server and may be Gigabytes or Terabytes in size — appears to be nothing more than a local in-memory object. At compile time, a fully compliant Java pre-parser examines the source code, creates a parse tree, identifies the relevant OLAP objects, and re-writes the original source code to include a native DBMS representation of the query. At run-time, the pre-compiled queries are transparently delivered to the back end analytics server for processing. Results are returned and encapsulated within a proxy object that is exposed to the client programmer.

As a concrete example, Listing 1 illustrates a simple SQL query that summarizes the total sales of Quebec's stores in 2011 for the data cube depicted in Figure 1. Typically, this query would be embedded within the application source code (e.g., wrapped in a JDBC call). Conversely, Listing 2 shows how this same query could be written in an Object-Oriented manner by a client-side Java programmer. Note that each algebraic operation is encapsulated within its own method (in this case, SELECT and PROJECT), while the logic of the operation is consolidated within the `return` statement. It is the job of the pre-parser to identify the relevant query methods and then extract and re-write the logic of the `return` statement(s). Again, it is important to understand that the original source code will never be executed directly. Instead, it is translated into the native operations of the OLAP algebra and sent to the server at run-time.

```
Select Store.province, SUM(sales)
From  Store, Time, Sales
Where Store.store_ID = Sales.store_ID AND
 Time.time_ID = Sales.time_ID AND
 Time.year = 2011 AND
 Store.province = 'Quebec'
Group by  Store.province
```

Listing 1. Simple SQL OLAP Query

```
Class SimpleQuery extends OLAPQuery{
 Public boolean select(){
  Store store = new Store();
  DateDimension time = new TimeDimension();
  return (time.getYear() == 2011 &&
      store.getProvince() == 'Quebec');
 }
 Public Object[] project(){
  Store store = new Store();
  Measure measure = new Measure();
  Object[] projections = {
   store.getProvince(),
   measure.getSales()};
  return projections;
 }}
```

Listing 2. An Object Oriented OLAP Query

While it is outside the scope of this paper to discuss the motivation for native language OLAP programming (a detailed presentation can be found in a recent submission [16]), we note that such an approach not only simplifies the programming model, but adds compile time type checking, robust re-factoring, and OOP functionality such as query inheritance and polymorphism. Moreover, query optimization is considerably easier on the backed as the DBMS natively understands the OLAP operations sent from the client side. In the context of the current paper, however, the significance of the query transformation process is that the authentication elements (e.g. roles and permissions) will be directly associated with the operations of the algebra. In fact, it is this algebraic representation that forms the input to the authentication module presented in the remainder of the paper.

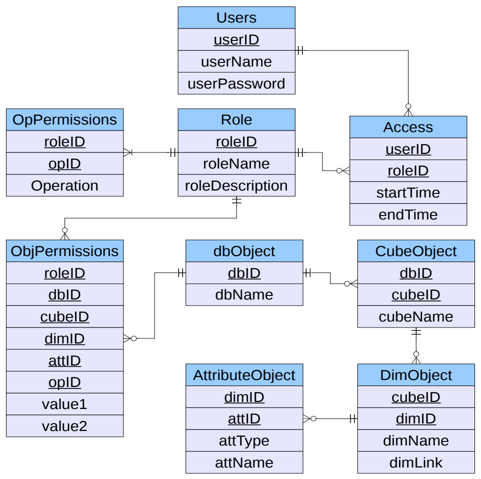

Figure 2. The Authentication DB.

IV. Authentication and Authorization

Without sufficient security countermeasures, open access to the OLAP repository becomes a powerful tool in the hands of malicious or unethical users. *Access Control* is the process that restricts unauthorized users from compromising protected data. This process can be thought of as occurring in two basic phases: **Authentication** and **Authorization**. Authentication is a form of *identity verification* that attempts to determine whether or not a user has valid credentials to access the system. In contrast, Authorization refers to the process of determining if the user has permission to access a specific data resource. In this section, we will describe our general framework, giving a detailed description of its two primary components and the relationship between them.

A. The Authentication Module

The authentication component is responsible for verifying user credentials against a list of valid accounts. These accounts are provided by the system administrator and are kept — along with their constituent permissions — in a backend database (i.e., the Authentication DB). The Authentication DB consists of a set of tables (`users`, `permissions`, and `objects`) that collectively represent the meta data required to authenticate and authorize the current user. For example, the `users` table stores basic user credentials (e.g., name, password), while the `permissions` table records the fact that a given user(s) may or may not access certain controlled objects. Figure 2 illustrates a slightly simplified version of the Authentication DB schema. In the current prototype, storage and access to the Authentication DB is provided by the SQLite toolkit [17]. SQLite is a small, open source C language library that is ideally suited to tasks that require basic relational query facilities to be embedded within a larger software stack.

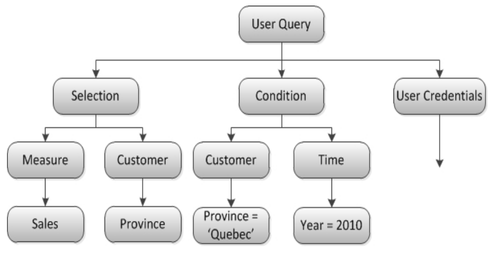

Figure 3. A small Parse Tree fragment.

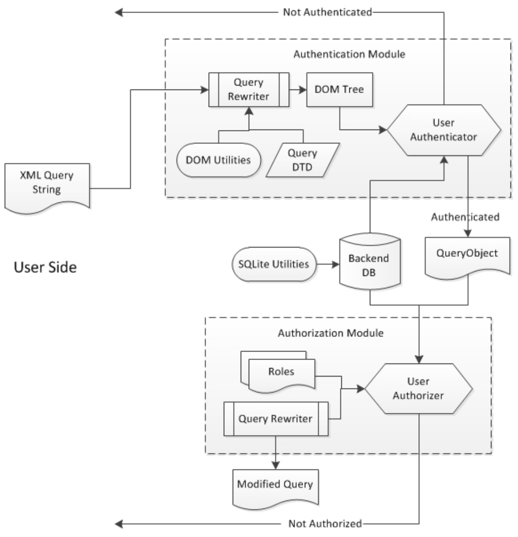

Figure 4. Authentication and Authorization.

Internally, the user's transformed OLAP query is represented in XML format (embedded within the re-written source code). Of course, in order to properly authenticate the query, it must first be decomposed into its algebraic components. The DBMS backend parser uses standard DOM utilities to parse the received query and extract the query elements (user credentials, dimensions, hierarchies, etc). Figure 3 shows the XML tree corresponding to the query depicted in Listing 2. Once the parsing is completed, the Authentication module extracts the user credentials to verify them against the Authentication DB. If the verification is successful, the DBMS proceeds with the authorization process. Otherwise, the query is rejected and the user/programmer is notified. In the case of successful authentication, the query elements are loaded into a memory-resident `QueryObject` structure for Authorization checking. The upper part of Figure 4 depicts the processing logic of the Authentication module.

B. The Authorization Module

The second — and more significant — phase is authorization, the process of determining if the user has permission to access specific data elements. Specifically, when a user requests access to a particular resource, the request is validated against the *permitted resource list* assigned to that user in the backend database. If the requested resource produces a valid match, the user request is allowed to execute as originally written. Otherwise, the query will either be rejected outright or modified according to a set of flexible transformation rules. To decide if the query will be modified or not, we rely on a set of *authorization objects* against which the rules will be applied. The rules themselves will be discussed in Section IV-D. The lower portion of Figure 4 graphically illustrates the Authorization module and indicates its interaction with the Authentication component.

C. Specifying Authorization Objects

Authorization is the granting of a right or privilege that enables a subject (e.g., users or user groups) to execute an action on an object. In order to make authorization decisions, we must first define the authorization objects. Note that the *objects* in the OLAP domain are different from those in the relational context. In a relational model, objects include logical elements such as tables, records within those tables, and fields within each record. In contrast, OLAP objects are elements of the more abstract conceptual model and include the dimensions of the multi-dimensional cube, the hierarchies within each dimension, and the aggregated cells (or facts). In practice, this changes the logic or focus of the authentication algorithm. For instance, a user in a relational environment may be allowed direct access to a specific record (or field in that record), while an OLAP user may be given permission to dynamically aggregate measure values at/to a certain level of detail in one or dimension hierarchies. Anything below this level of granularity would be considered too *sensitive*, and hence should be protected. In fact, the existence of aggregation hierarchies is perhaps the most important single distinction between the authentication logic of the OLAP domain versus that of the relational world.

We note that in the discussion that follows, we assume an *open world* policy, where only prohibitions are specified. In other words, permissions are implied by the absence of any explicit prohibition. We use the open world policy mainly for practical reasons, as the sheer number of possible prohibitions in an enterprise OLAP environment would be overwhelming.

Before discussing the authorization rules themselves, we first look at a pair of examples that illustrate the importance of proper authorization services in the OLAP domain. We begin with the definition of a policy for accessing a specific aggregation level in a data cube dimension hierarchy.

Example 1: An employee, Alice, is working in the Montreal store associated with the cube of Figure 1. The policy is simple: Alice should not know the sales totals of the individual provinces.

Clearly, Alice is prohibited from reading or aggregating data at the provincial level in the Store dimension hierarchy.

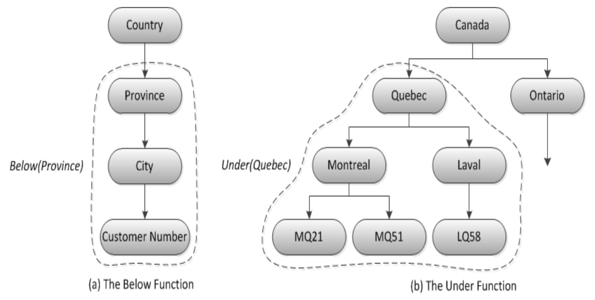

Figure 5. The Below and Under functions.

However, in the absence of any further restrictions, it would still be possible for her to compute the restricted values from the lower hierarchies levels (e.g., City or Store_Number). Ideally, the warehouse administrator should not be responsible for identifying and manually ensuring that all *implied* levels be included in the policy. Instead, our model assumes this responsibility and can, if necessary, restrict access to all *child* levels through the use of the *Below* function. As the name implies, this function returns a list consisting of the specified level L_i and all the lower levels of the associated dimension hierarchy. Figure 5(a) illustrates an example using a *Below(Province)* instantiation. Here, all levels surrounded by the dashed line are considered to be Authorization Objects, and thus should be protected. The formalization of the *Below* function is given by Definition 2.

Definition 2: In any dimension D_i with hierarchy H_i, the function Below(L_i) is defined as Below(L_i) = $\{L_j$: such that $L_j \preceq L_i$ holds$\}$, where L_i is the prohibited dimension level.

As shown in Example 1, a policy may restrict the user from accessing *any* of the values of a given level or levels. However, there are times when this approach is too coarse. Instead, we would like to also have a less restrictive mechanism that would only prevent the user from accessing a specific value *within* a level(s). For instance, suppose we want to alter the policy in Example 1 to make it more specific. The new policy might look like the following:

Example 2: Alice should not know the sales total for the province of Quebec.

In Example 2, we see that Alice may view sales totals for all provinces other than Quebec. However, Alice can still compute the Quebec sales by summing the sales of individual Quebec cities, or by summing the sales of Quebec's many stores. In other words, she can use the values of the lower levels to compute the prohibited value. Hence, all these values should also be protected. To determine the list of restricted member values, our model adds the *Under* function. Figure 5 (b) provides an example using *Under(Quebec)*. Here, all the values surrounded by the dashed line should be protected.

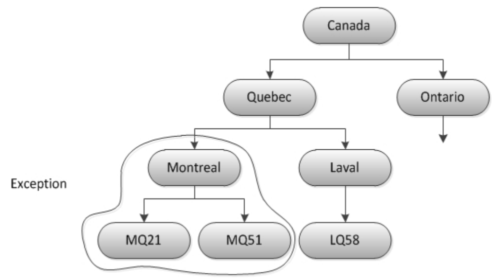

Figure 6. An Authorization Exception.

Finally, it is also possible that *exceptions* to the general authorization rule are required. For instance, Alice should not know the sales of stores in the province of Quebec except for the stores in the city/region she manages (e.g., Montreal). Figure 6 graphically illustrates this policy. In this case, the circled members represent the values associated with the exception that would, in turn, be contained within a larger encapsulating restriction. Note that a user may have one or more exceptions on a given hierarchy. The formalization of the exception object is given in in Definition 3.

Definition 3: For any prohibited level L_i, there may be an Exception E such that E contains a set Ev of values belonging to Under(L_i). That is, Ev $\in$ values of Under(L_i).

To summarize, authorization objects consist of the values of the prohibited level and all the levels below it, excluding zero or more exception value(s). We formalize the concept of the *Authorization Object* in Definition 4.

Definition 4: An Authorization Object O = $\{$v : v $\in$ Under(L_i) - Ev$\}$, where L_i is the prohibited level, and Ev is the exception value.

D. Authorization Rules

We now turn to the query authorization process itself. As noted above, pre-compiled queries are encoded internally in XML format. For the sake of simplicity (and space constraints), we will depict the received queries in a more compact form in this section. For example, Listing 3 represents the same query shown in Listing 2. Note that the query is divided into three elements: the SELECTION element, the CONDITION element, and the FROM element. The SELECTION element lists all attributes and measures the user wants to retrieve. The CONDITION element, in turn, limits or filters the data we fetch from the cube. Finally, the FROM element indicates the cube from which data is to be retrieved.

In the discussion that follows, we will assume the existence of a cube corresponding to Figure 1. That is, the cube has three dimensions (Product, Store, and Time). Dimension hierarchies include Product_Number $\preceq$ Type $\preceq$ Category for Product, Store_Number $\preceq$ City $\preceq$ Province $\preceq$ Country for Store, and Month $\preceq$ Year for Time. Selection

```
Selection:
    Store.Province, SUM(sales)
Condition:
    Time.year = 2011 AND
    Store.Province = 'Quebec'
From:
    Sales
```

Listing 3. A Query in Simple Form

operations correspond to the identification of one or more cells associated with some combination of hierarchy levels.

One of the advantages of building directly upon the OLAP conceptual model and its associated algebra is that it becomes much easier to represent, and subsequently assess, authorization policies. Specifically, we may think of policy analysis in terms of Restrictions, Exceptions, and Level Values that form a bridge between the algebra and the Authentication DB. There are in fact four primary *policy classes*, as indicated in the following list:

1) L_i Restriction + No Exception
2) L_i Restriction + Exception
3) Restriction on a specific value P of level L_i + no Exception
4) Restriction on a specific value P of level L_i + Exception

As mentioned, the query must be validated before execution. If validation is successful, then it can be executed as originally specified. Otherwise, the query is either rejected or rewritten according to a set of *transformation rules*. In the remainder of this section, we describe the four policy classes and the processing logic relevant to each.

1) ***Policy Class 1****: L_i Restriction + No Exception:* If a user is prohibited from accessing level L_i and the user has no exception(s), then the authorization objects consist of the values of level L_i and all the levels below it. In short, this means that if the user query specifies level L_i or any of its children in the SELECTION element, then the query should simply be rejected. Moreover, if any *value* belonging to the L_i level or any of its children is specified in the CONDITION element of the query, the query should also be rejected. The formalization of the rule and an illustrative example is given below.

Rule 1. *If a user is prohibited from accessing the values of level L_i, and there is no exception, then the Authorization Objects (O) = $\{v : v \in$ Below(L_i) $\}$.*

Example 3: If Alice sends the query depicted in Listing 4, which summarizes the total sales of Canada's stores in 2011, the query should be rejected.

Why is this query rejected? Recall that Alice is restricted from accessing provincial sales. Consequently, we see that an implicitly prohibited child level (i.e., City) is a component of the SELECTION element. So, if we allow this query, Alice

```
Selection:
   Store.City, SUM(sales)
Condition:
   Time.year = 2011 AND
   Store.Country = 'Canada'
From:
   Sales
```

Listing 4. Rule 1 example

```
Selection:
   Store.province, SUM(sales)
Condition:
   Time.year = 2011 AND
   Store.City = 'Montreal'
From:
   Sales
```

Listing 5. Rule 4 example

```
Selection:
 Store.City, SUM(sales)
Condition:
 Time.Year = 2011
From:
 Sales
```

Listing 6. Simple OLAP Query 2

```
Selection:
 Store.City, SUM(sales)
Condition:
 Time.Year = 2011 AND
 Store.Provice = 'Quebec'
From:
 Sales
```

Listing 7. Rule 5 example

can in fact compute the provincial sales by summing the associated city sales.

2) ***Policy Class 2***: *L_i Restriction + Exception:* In this case, the authorization objects that should be protected consist of the prohibited level value and all values below it, *except* of course for the value of the exception or any value under it. Let us first formalize this case, before proceeding with a detailed description.

Rule 2. *If a user is restricted from accessing the values of level L_i, and the user has an exception E, then the Authorization Objects (O) = $\{v : v \in$ Below(L_i) - Under(Ev) $\}$.*

As such, when a user is prohibited from accessing the L_i level — excluding the *exception* values — then the query can be (i) allowed to execute, or (ii) modified before its execution. Let's look at these two cases now.

Rule 3. *The query will be allowed to execute without modification* if *the prohibited level value Lv or any of its more granular level values in (Below(L_i)) exists in the* CONDITION *element AND is equal to the exception value (Ev) or any of its implied values in (Under(Ev)).*

Example 4: Suppose that we have the following policy: Alice is restricted from accessing provincial sales *except* the sales for Canadian provinces. If Alice resubmits the query in Listing 3, it will now be executed without modification because the prohibited value (e.g., Quebec) is *under* the exception value (e.g., Under(Canada)).

But what if Alice has an exception value only for a more detailed child level of L_i (e.g., the city of Montreal)? In this case, if Alice submits the previous query, it should now be modified by replacing the restricted value (e.g., Quebec) in the CONDITION element with the exception value (e.g., Montreal). In this example, Alice gets only the values that she is allowed to see. The modified query is depicted in Listing 5. Rule 4 gives the formalization of this case.

Rule 4. *If the prohibited level value Lv or any of its more granular level values (Under(Lv)) exists in the* CONDITION *element, and the exception value belongs to this set of values, then the query should be modified by replacing the prohibited value with the exception value.*

In addition to the scenario just described, the query can also be modified by adding a new predicate to the CONDITION element when the prohibited level or any of its child levels exists in the SELECTION element only.

Rule 5. *If the prohibited level Lv or any of its more granular levels (Below(L_i)) exists in the* SELECTION *element only, then the query should be modified by adding the exception E as a new predicate to the query.*

Example 5: Suppose that Alice sends the query depicted in Listing 6. In this case, the query will be modified by adding a new predicate (i.e., Store.Province = 'Quebec'), because the prohibited level (i.e., City) exists in the SELECTION element. After the modification, Alice will see only the cities of Quebec. The modified query is depicted in Listing 7.

The complete processing logic for Policy Class 2 (i.e., Rule 3, Rule 4, and Rule 5) is encapsulated in Algorithm 1. Essentially, the algorithm takes the prohibited level L_i and the exception E as input and produces as output an authorization decision to execute or modify the query. The process is divided into two main parts or conditions. In the first case, we are looking at situations whereby the prohibited level L_j exists in the query CONDITION element. Here, the query can either be allowed to execute directly or further modified. It executes directly if the prohibited value *Lv* is equal to the exception value *Ev* or any value under *Ev*. However, if the exception value *Ev* is equivalent to any value under *Lv*, then the query is modified by replacing the prohibited level with the exception level AND the prohibited level value with the exception value.

In the second case, we target the scenario whereby the prohibited level L_j exists in the SELECTION element only. Here, we modify the original query by adding the exception *E* as a new condition.

```
input  : The prohibited level $L_i$ and the exception $E$
output: Decision to directly execute or modify
Let Ev = $E$ value;
foreach level $L_j \in$ Below($L_i$) do
    if $L_j$ exists in the query CONDITION element then
        Let Lv = $L_j$ value;
        if Lv == Ev OR Lv $\in$ Under(Ev) then
            Allow the query to execute without modification;
        end
        else if Ev $\in$ Under(Lv) then
            Replace $E$ by $L_j$, and Ev by Lv, then inform the user, and allow the query to execute;
        end
    end
    else if $L_j$ exists only in the query SELECTION element then
        Add $E$ as new condition to the user query, inform the user, and allow the query to execute;
    end
end
```

Algorithm 1:

*3) **Policy Class** 3: Restriction on a specific value P of level L_i + no Exception:* We now turn to the classes in which specific values at a given level are restricted, as opposed to all members at a given level. We begin with the simplest scenario.

Rule 6. *If a user is prohibited from accessing a specific value P of level L_i, and the user has no exceptions, then the Authorization Objects(O)= $\{v : v \in P \cup$ Under(P) where P is the prohibited value$\}$.*

Here, the prohibited value *P*, or some value under *P*, exists in the query CONDITION element. As per Rule 6, the query should simply be rejected. But what if L_i exists in the SELECTION element only? In this case, the query should be modified by adding the prohibited value as a new predicate to the query CONDITION element. Let's look at the following example.

Example 6: Suppose that Alice is restricted from accessing Quebec's sales. If Alice sends the query depicted in Listing 8, the query should be modified as shown in Listing 9.

```
Selection:
    Store.Province, SUM(sales)
Condition:
    Time.year = 2011
From:
    Sales
```

Listing 8. Simple OLAP Query 3

```
Selection:
    Store.Province, SUM(sales)
Condition:
    Time.year = 2011 AND
    Store.Province != 'Quebec'
From:
    Sales
```

Listing 9. Rule 7 example

The associated query summarizes the sales of provinces in 2011. As noted, the SELECTION element contains the prohibited level (Province), so instead of rejecting the query we modify it by adding a new predicate to the condition. The modified query returns only the sales that Alice is allowed to see. The logic is formalized in Rule 7 below.

Rule 7. *If the prohibited level L_i exists in the* SELECTION *element only, then the query should be modified by adding a new predicate to the query* CONDITION *element.*

*4) **Policy Class** 4: Restriction on a specific value P of level L_i + Exception:* Finally, we add an exception to the queries described by Class 3. Here, the relevant authorization objects consist of the prohibited value *(P)*, *minus* the exception values.

Rule 8. *If a user is restricted from accessing a value P of level L_i, and the user has an exception E, then the Authorization Objects(O)= $\{v : v \in$ (P $\cup$ Under(P)) - (Ev $\cup$ Under(Ev))$\}$ where P is the prohibited value and E is the exception.*

In this scenario, the query can either be allowed to execute or modified according to the following associated rules.

Rule 9. *The query will be allowed to execute, if the prohibited value Lv exists in the* CONDITION *element AND is equal to the exception value Ev or any value Under(Ev).*

```
Selection:
    Store.City, SUM(sales)
Condition:
    Store.City = 'Montreal'
From:
    Sales
```

Listing 10. Rule 9 example

Example 7: Suppose that Alice is restricted from accessing the sales of Canadian provinces, *except* for the sales of Quebec. If Alice sends the Query depicted in Listing 10, the query will be allowed to execute since the prohibited value (i.e., Montreal) is under the exception value (i.e., Quebec).

Rule 10. *If the prohibited level L_i exists in the query* SELECTION *element only, the query will be modified by adding the exception E as a new predicate. In principle, this rule is similar to Rule 4.*

Rule 11. *When Lv exists in the query* CONDITION *element AND Lv is under Ev, the query is modified by replacing the prohibited level L_i by the exception level E AND the prohibited level value Lv by the exception value Ev.*

Algorithm 2 illustrates the full processing logic for Policy Class 4 (Rule 8, Rule 9, Rule 10, and Rule 11). In short, the authentication module takes the prohibited level value *P* and the exception *E* as input and gives as output an authorization decision to execute or modify the query. The algorithm is again divided into two main parts. The first component targets the case whereby the prohibited value *P* exists in the query CONDITION element. Here, the query can be modified or executed directly. If the prohibited value belongs to the set of values under *E* , the query is modified by replacing the condition that contains the prohibited value by a new one containing the exception. Conversely, the query is allowed to execute directly if the prohibited level value *Lv* belongs to the values *Under(P)* AND *Lv* is equal to the exception value *Ev* OR *Ev* belongs to the values *Under(Lv)*.

In the second case, a new condition (exception *E*) is added to the query CONDITION element when the prohibited level *Lv* or any level below it *Below(Lv)* exists in the SELECTION element only.

```
input  : The prohibited value P of level $L_i$ and the
         exception E
output: Decision to directly execute or modify

Let Ev = E value;
foreach level $L_j \in$ Below($L_i$) do
    if $L_j$ exists in the query CONDITION element
    then
        Let Lv = $L_j$ value;
        if (Lv == P) AND (P $\in$ Under(Ev)) then
            Add E as a new condition instead of the
            condition that contains $L_j$, inform the
            user, and allow the query to execute;
        end
        else if (Lv $\in$ Under(P)) AND (Lv == Ev
        OR Ev $\in$ Under(Lv)) then
            Allow the query to execute without
            modification;
        end
    end
    else if $L_j$ exists only in the query SELECTION
    element then
        Add E as new condition to the user query,
        inform the user, and allow the query to
        execute;
    end
end
```

Algorithm 2:

E. Authorization Rule Summary

The preceding sections have formalized the authentication framework in terms of four policy classes and their associated transformation rules. Below, we summarize the authorization decision in terms of its three possible outcomes — **Execute**, **Modify**, **Reject**:

1) The query is allowed to execute without modification in two situations:
   - Level L_i is restricted and there is an exception *E*:
     a) If any upper level exists in the SELECTION or PROJECTION query element, OR
     b) If the L_i value or any value from the levels below it exists in the CONDITION element AND this value is equal to the exception value *Ev* or any value under it.
   - A specific value of L_i is restricted and there is an exception *E*:
     a) If the prohibited value *Lv* or any value under it exists in the CONDITION element AND it is equal to the exception value *Ev* OR any value under it.
2) The query is modified in one situation:
   - A level L_i is restricted and there is an exception *E*:
     a) If level L_i or any value from the levels below it exists in the query SELECTION element only, then we add the exception *E* as a new condition, OR
     b) If the exception value *Ev* belongs to the values under *Lv*, then we replace the prohibited level in the CONDITION element by the exception *E*.
3) The query is rejected in two situations:
   - A level L_i is restricted, and there is no exception:
     a) If level L_i or any value from a lower level exists in the SELECTION element only, OR
     b) If level L_i or any value from the levels below it exists in the CONDITION element.
   - A specific value *P* is restricted, and there is no exception:
     a) If *P* or any value under it exists in the CONDITION element.

F. A note on Performance

As noted above, the authorization framework has been incorporated into a DBMS prototype specifically designed for OLAP storage and analysis. In practice, the authorization logic has a negligible impact on performance (less than a few milliseconds for the queries presented in this paper). Specifically, the decomposition of the user query into its algebraic components (and conditions) is performed by the underlying query engine; the framework simply *borrows* the result as input to the authorization process. Moreover, the analysis of policy classes is based upon a fixed set of IF/ELSE cases that, in turn, manipulate a small in-memory Authentication Database. The run-time impact of this analysis is completely dominated by the cost of answering the (validated) query.

V. Conclusions and Future Work

In this paper, we have discussed a query re-writing model to provide access control in multi-dimensional OLAP environments. We began by defining a conceptual model that focused on the data cube and its constituent dimension hierarchies. From there we introduced the notion of authorization objects designed to identify and constrain the relationships between parent/child aggregation levels. We then presented a series of rules that exploited the authorization objects to decide whether user queries should be rejected, executed directly, or dynamically and transparently transformed. In the latter case, we identified a set of minimal changes that would allow queries to proceed against a subset of the requested data.

We note that while the current authentication and authorization framework has been integrated into a prototype DBMS that provides OLAP-specific indexing and storage, we believe that the general principles are broadly applicable to any DBMS product that understands the fundamental data cube model. Exploiting the proposed framework would allow such systems to significantly simplify the process of designing and enforcing OLAP security policies by associating authorization decisions with an intuitive conceptual model rather than the low level logical model of relational DBMSs.

Finally, it is important to point out that the framework presented in this paper cannot block all attempts to access restricted data. In particular, it is possible for a user possessing some degree of external knowledge to combine the results of multiple *valid* queries to obtain data that is itself meant to be protected. We refer to such exploits as *inference* attacks. We are currently working on inference detection mechanisms that will piggy back on top of the core authentication and authorization framework to provide an even greater level of security for OLAP data.

References

[1] T. H. Davenport and J. G. Harris, "Competing on analytics: The new science of winning," in *Harvard Business School Press*, 2007.

[2] E. Fernández-Medina, J. Trujillo, R. Villarroel, and M. Piattini, "Developing secure data warehouses with a UML extension," *Information Systems*, vol. 32, pp. 826–856, 2007.

[3] J. Trujillo, E. Soler, E. Fernandez-Medina, and M. Piattini, "A UML 2.0 profile to define security requirements for data warehouses," *Computer Standards & Interfaces*, vol. 31, pp. 969–983, 2009.

[4] A. Rosenthal and E. Sciore, "View security as the basic for data warehouse security," in *International Workshop on Design and Management of Data Warehouse*, 2000, pp. 8.1–8.8.

[5] C. Blanco, E. Fernandez-Medina, J. Trujillo, and M. Piattini, "How to implement multidimensional security into OLAP tools," *International Journal of Business Intelligence and Data Mining*, vol. 3, pp. 255–276, 2008.

[6] C. Blanco, I. G.-R. de Guzman, D. Rosado, E. Fernandez-Medina, and J. Trujillo, "Applying QVT in order to implement secure data warehouses in SQL Server Analysis Services," *Journal of Research and Practice in Information Technology*, vol. 41, pp. 135–154, 2009.

[7] J. Trujillo, E. Soler, E. Fernández-Medina, and M. Piattini, "An engineering process for developing secure data warehouses," *Information and Software Technology*, vol. 51, pp. 1033–1051, 2009.

[8] N. Katic, G. Quirchmay, J. Schiefer, M. Stolba, and A. Tjoa, "A prototype model for data warehouse security based on metadata," in *DEXA*, 1998, pp. 300–308.

[9] A. Deshpande, Z. Ives, and V. Raman, "Adaptive query processing," *Foundations and Trends in Databases*, vol. 1, pp. 1–140, 2007.

[10] S. Rizvi, A. O. Mendelzon, S. Sudarshan, and P. Roy, "Extending query rewriting techniques for fine-grained access control," in *ACM SIGMOD*, 2004, pp. 551–562.

[11] "The Virtual Private Database," June 2011, http://www.oracle.com/technetwork/database/security/index-088277.html.

[12] "Microsoft Analysis Services," June 2011, http://www.microsoft.com/sqlserver/2008/en/us/analysis-services.aspx.

[13] J. Gray, A. Bosworth, A. Layman, D. Reichart, and H. Pirahesh, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," *Data Mining and Knowledge Discovery*, vol. 1, pp. 29–53, 1997.

[14] V. Harinarayan, A. Rajaraman, and J. Ullman, "Implementing data cubes efficiently," in *ACM SIGMOD*, 1996, pp. 205–227.

[15] E. Malinowski and E. Zimányi, "Hierarchies in a multidimensional model: from conceptual modeling to logical representation," *Data and Knowledge Engineering*, vol. 59, pp. 348–377, 2006.

[16] T. Eavis, H. Tabbara, and A. Taleb, "The NOX framework: native language queries for business intelligence applications," in *DaWak*, 2010, pp. 172–189.

[17] "SQL database engine," June 2011, http://www.sqlite.org.