Securing Access to Data in Business Intelligence Domains

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

Todd Eavis
Department of Computer Science
Concordia University
Montreal, Canada
Email: eavis@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 authorization 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 authorization 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. For the most part, the process is largely transparent to the user. We demonstrate the scope of our framework with a series of common OLAP query case studies, as well as an experimental performance analysis using a common OLAP benchmark. The end result is an intuitive but powerful approach to database authorization 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, 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 — identification information, medical data or even religious beliefs or ideologies — that must, by definition, be restricted to authorized users. As a result, various pieces of legislation designed to protect individual privacy have been proposed. One can consider, for example, the United States HIPAA-Health Insurance Portability and Accountability Act, which regulates the privacy of personal health care information, the GLBA (Gramm-Leach-Bliley Act, also known as the Financial Modernization Act), the Sarbanes-Oxley Act, and the EU's Safe Harbour Law. These laws usually require strict technical security measures for guaranteeing privacy, with a failure to comply possibly leading to significant penalties. In this context, organizations must be able to guarantee the correct administration, security and confidentiality of the information they collect and store.

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. Several authors define frameworks that are likely too restrictive for production warehouses. For example, security models have been based upon the notion of user-specific *authorization views* that 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, including the use of Unified Modeling Language (UML) profiles for the definition of security constraints. Here, however, the physical implementation of the underlying authorization system remains undefined.

In a recent paper, we presented an authorization model for OLAP environments that is based on a query rewriting technique [1]. 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.

In the current paper, we expand upon the original work in two ways. First, we discuss the data structures and algorithms utilized by the functions that manipulate the hierarchical elements of the conceptual data model. The performance of the transformation process is closely associated with these mechanisms. To underscore the practical viability of the proposed methods, we have also added an experimental section that highlights the processing overhead relative to the execution costs of the underlying query. In addition to these core enhancements, we update the paper with a deeper treatment of the internal representation of the intermediate query, as well as a broader discussion of the work related to this research domain. Finally, an appendix has been included in order to provide the reader with a clear description of the query test cases.

The 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, including the extended section on query representation and hierarchy processing, are then presented in detail in Section IV. Experimental results are discussed in Section V, with final conclusions offered in Section VI.

II. Related work

The need for strong security mechanisms has long been recognized in the context of relational database management systems. A variety of *Access Control* techniques have in fact been proposed to restrict access to the appropriate authorized users. Each such technique aims to limit users and/or processes to performing only those table or column operations (i.e., read, write, or execute) for which they are actually authorized. The relevant control then either allows or disallows the execution of the specific operation to be performed.

During the early stages of database security research, the primary focus was on *Discretionary Access Controls* (DACs) [2]. The basic form of DAC authorization consists of a triple (*s, o, a*), such that a set of security *subjects s* can execute *actions a* on a set of security *objects o*. The earliest DAC model was the Access Matrix, whereby authorization is represented in an $|s| * |o|$ matrix in which rows are subjects, columns are objects and the mapping of subject and object pairs results in the set of rights the subject *s* has over the object *o*. A primary benefit associated with the use of a DAC is that it can be implemented relatively easily. However, in practice, large organizations give rise to extremely large access matrices. Maintaining matrix contents can be difficult as the matrix needs to be updated with each update to the subjects (e.g., addition of users) or objects (e.g., addition of columns).

In the 1980's the focus moved to *Mandatory Access Controls* (MACs) [3]. The most common form of MAC is the multilevel security policy, which secures data by assigning security labels to subjects and objects, and subsequently compares these labels to the level of sensitivity at which a user is operating. The access controls in MACs restrict subjects from accessing information labeled with a higher level. In other words, a user can access the data in his/her security level or in a lower security level(s) but not in a higher level(s). MAC is relatively straightforward from a design perspective and is considered a good model either for systems in which confidentiality is a primary access control concern, or in which the objects being protected are valuable. That being said, MAC systems can also be expensive to implement due to the necessity for applications to be rewritten to adhere to MAC labels and properties. Also, MACs do not provide each user with a distinct authorization context (i.e., access to only their own data address), nor fine-grained least privilege mechanisms.

An alternative approach was introduced in the 1990's [4]. This new model is known as *Role Based Access Control* (RBAC). RBAC consists of roles, permissions, and users. Roles are created for various job functions, with permissions for specific operations then assigned to these roles. Users are assigned particular roles, and through those role assignments acquire permissions to perform particular operations. The consolidation of access control for many users into a single role entry allows for much easier management of the overall system and much more effective verification of security policies. However, in large systems, role inheritance — and the need for finer-grained customized privileges— makes administration potentially unwieldy. Additionally, it is inappropriate for multi-dimensional data modeling due to the fact that it is based on relational concepts (i.e., tables, columns, rows, and cells), and thus, cannot be implemented directly on top of the multi-dimensional model.

In contrast to the Access Control paradigm, a number of security models that restrict data warehouse access have also 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 [5]. 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]. Others have developed security requirements for the entire Data Warehouse life cycle [8]–[10]. In this case, they first propose a model (agent-goal-decision-information) to support the early and late requirements for the development of DWs, then extend that model to capture security aspects in order to prevent illegitimate attempts to access the warehouse. 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 authorization algorithms

per se, but rather on design methodologies that would most effectively use existing technologies, including the Model Driven Architecture (MDA) and the standard Software Process Engineering Metamodel Specification (SPEM) from the Object Management Group (OMG).

In terms of true authorization models, several researchers have attempted to augment the core Database Management System (DBMS) with authorizations views [11]–[13]. Typically, alternate views of data are defined for each distinct user or user group. A query Q is inferred to be authorized if there is an equivalent query Q' which uses only authorized views. 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 [14]. Beyond that, however, rewriting has also been utilized to provide fine grained access control in Relational databases [15]. Oracle's Virtual Private Database (VPD) [16], 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 [15], 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 [17], do in fact provide some support for OLAP-level security specification. Here, however, there is virtually no formal basis for the application of authorization logic and little can be said about the actual scope or limitations of the relevant subsystems. This is in contrast to the work discussed in this paper, where the primary contribution is a query rewriting technique that not only transparently supports *indirect* authorization, but does so on the basis of an explicit policy/rule model. Moreover, the mechanisms are not tightly connected to a specific DBMS product but can be applied to virtually any standard data management system.

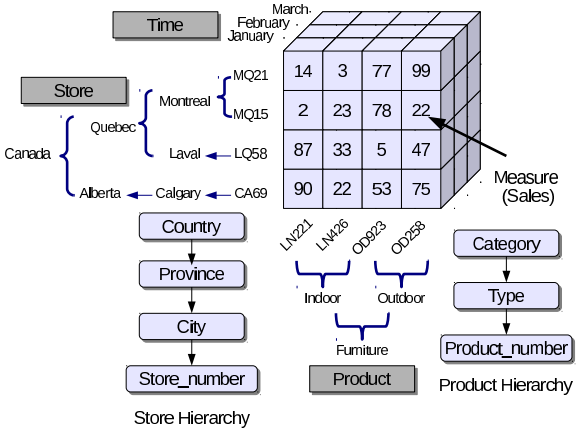

Figure 1. A simple three dimensional data cube

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* [18]. 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 [19]. 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 $(0 \leq h \leq j)$.

Finally, we note that there are in fact many variations on the form of OLAP hierarchies [20] (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 authorization 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

```
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

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.

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 [21]), 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 authorization 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 authorization module presented in the remainder of the paper.

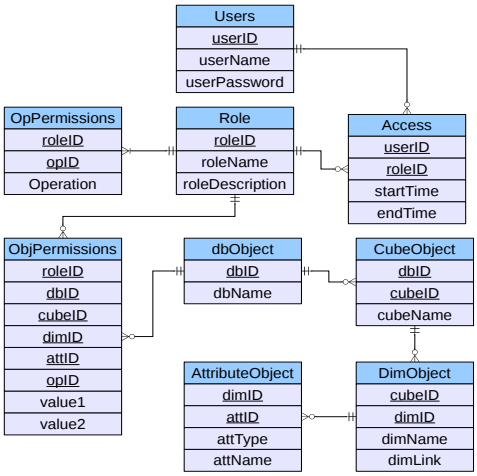

Figure 2. The Authorization DB.

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 Authorization DB). The Authorization 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 Authorization DB schema. In the current prototype, storage and access to the Authorization DB is provided by the SQLite toolkit [22]. 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.

```
<?xml version='1.0' encoding='UTF-8'?>
<DOCTYPE QUERY SYSTEM "ClientQuery.dtd" []>
<QUERY><DATA_QUERY>
  <CUBE_NAME>Furniture Sales</CUBE_NAME>

 <OPERATION_LIST>
  <OPERATION><PROJECTION>
    <MEASURE_LIST><MEASURE>Sales</MEASURE></MEASURE_LIST>
    <ATTRIBUTE_LIST>
     <PROJECTION_DIMENSION><DIMENSION_NAME>Store</DIMENSION_NAME>
       <ATTRIBUTE>Province</ATTRIBUTE>
       .....
  <OPERATION> <SELECTION>
    <DIMENSION_LIST><COMPOUND_DIMENSION> <DIMENSION_LIST>
      <DIMENSION><DIMENSION_NAME>Store</DIMENSION_NAME>
       <EXPRESSION><RELATIONAL_EXP><BASIC_EXP><SIMPLE_EXP><EXP_VALUE>
         <ATTRIBUTE>Province</ATTRIBUTE></EXP_VALUE></SIMPLE_EXP>
          <COND_OP><RELATIONAL_OP>EQUALS</RELATIONAL_OP></COND_OP>
          <SIMPLE_EXP><EXP_VALUE><CONSTANT>Quebec</CONSTANT> .....
     <LOGICAL_OP>AND</LOGICAL_OP>
     <DIMENSION><DIMENSION_NAME>Time</DIMENSION_NAME>
      <EXPRESSION><RELATIONAL_EXP><BASIC_EXP><SIMPLE_EXP><EXP_VALUE>
        <ATTRIBUTE>Year</ATTRIBUTE></EXP_VALUE></SIMPLE_EXP>
         <COND_OP><RELATIONAL_OP>EQUALS</RELATIONAL_OP></COND_OP>
           <SIMPLE_EXP><EXP_VALUE><CONSTANT>2011</CONSTANT>
            .....
   </OPERATION_LIST>
 <USER_CREDENTIALS>
  <USER_NAME>John</USER_NAME>
  <PASSWORD>J86mn</PASSWORD>
 </USER_CREDENTIALS>
</QUERY>
```

Figure 3. An XML query segment.

Internally, the user's transformed OLAP query is represented in XML format (embedded within the re-written source code). To validate the received XML query, the system relies on a Document Type Declaration (DTD) grammar [23] that is used to describe the structure of the expected XML query (We note that the somewhat more expressive XMLSchema can also be used for this purpose). The grammar itself is quite large but, ultimately, its purpose is to represent the functionality of the analytics queries one would expect to see in a Business Intelligence context. Figure 3 shows an XML-encoded segment of the query depicted in Listing 2. With a little effort one can see how the "total sales in 2011 for Quebec stores" is captured by the sequence of nested XML

The user query itself can be divided into three main parts: CUBE NAME, OPERATION LIST, and USER CREDENTIALS. As one would expect, the CUBE NAME element simply indicates the cube from which data is to be retrieved (the DBMS would likely store multiple cubes). The OPERATION LIST element contains one or more OPERATION elements, with PROJECTION and SELECTION being by far the most common (other analytics operations include CHANGE LEVEL, CHANGE BASE, PIVOT, DRILL ACROSS, UNION, DIFFERENCE, and INTERSECTION). In short, the PROJECTION element lists all attributes and measures the user wants to retrieve (e.g., Store.Province, and SUM(Sales)). The SELECTION element, in turn, limits or filters the data fetched from the data cube. Each SELECTION element consists of one or more criteria combined by

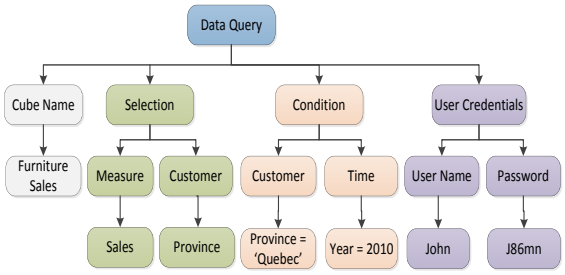

Figure 4. A small Parse Tree fragment.

a LOGICAL OP element (e.g., Store.Province = 'Quebec' AND Time.Year = 2011). Finally, the USER CREDENTIALS element, as the name indicates, contains the user's authentication identifiers (i.e., the user name and password).

Of course, in order to properly authenticate the query, it must first be parsed and decomposed into its algebraic components. In fact, the parsing is done in two phases. First, the DOM parser utility is used to produce a DOM tree that represents the raw contents of the XML document. In this phase, the parser not only builds the tree but also verifies that the received query has valid syntax corresponding to the DTD query grammar. An XML document is considered as valid if it contains only those elements defined in the DTD. If the query is syntactically valid, the query proceeds to the second phase. Otherwise, a parsing error message is returned to the user.

Figure 4 shows the node tree corresponding to the query depicted in Figure 3. We can easily see that the content of this parse tree is equivalent to the OLAP query represented in the XML format. Specifically, it is executed against the cube Furniture Sales and consists of two OLAP operations (Projection and Selection). The projection operation returns the dimension attribute Customer.Province, as well as one measure attribute — Sales. The Selection operation filters the returned information via two conditions on the dimensions Customer (i.e., Province = Quebec) and Time (Year = 2010). The user name "John" and the password "J86mn" represent the user credentials.

In the second phase of the process, the DOM tree is converted into a simplified data structure. This "Query Object" is cached in memory and contains all the query elements (i.e., returned attributes, query conditions along with its dimensions and attributes, and user credentials). The purpose of this final conversion process is to transform the user query into a simple, minimal data structure that represents the query in a compact but expressive form.

Once the parsing is completed, the Authentication module extracts the user credentials to verify them against a valid account stored in the Authorization DB. If the verification is successful, the DBMS proceeds with the authorization process. Otherwise, the query is rejected and the user/programmer is notified. The upper part of Figure 5 depicts the processing logic of the Authentication module. As a final note, we add that the prototype for the authentication and authorization modules has been designed as a third party component that can interact with existing DBMS products. As such, it does not maintain connection-oriented session data and thus requires authentication information to be provided for each query. That being said, this has a very limited impact on performance as the bulk of the processing logic is associated with the authorization module, which must assess user privileges on a query by query basis.

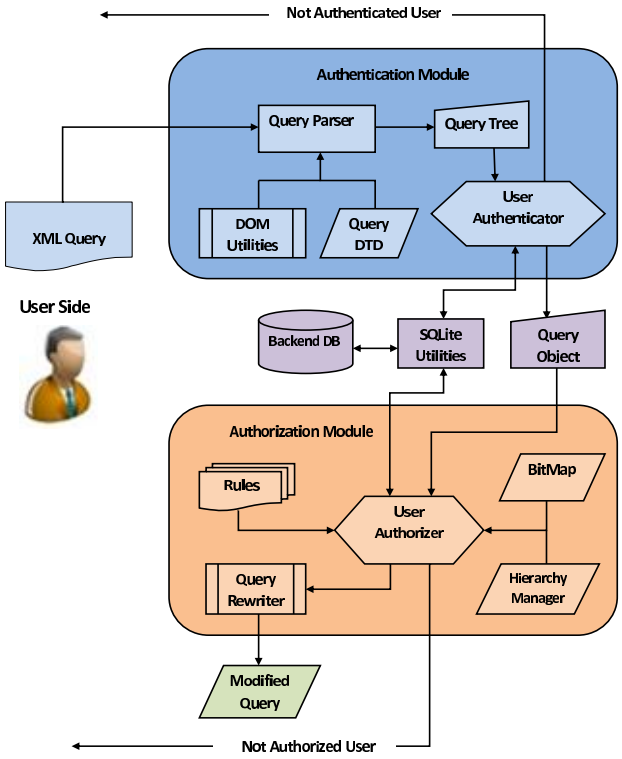

Figure 5. Authentication and Authorization.

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-E. The lower portion of

Figure 5 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 authorization 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 authorization 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 approach for the simple reason that, in contrast to the users in operational database settings, OLAP users are typically drawn from a relatively small pool of enterprise decision makers. As such, these more senior employees generally require broad access to data. It therefore makes sense to use an open world policy that defines a relatively small set of constraints, rather than a closed world approach that would require extensive "positive" privileges to be defined. That being said, there is no theoretical barrier to the use of a closed world strategy.

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. 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).

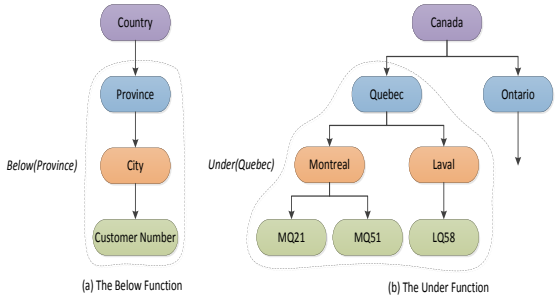

Figure 6. The Below and Under functions.

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 6(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, which is formalized in Definition 3. Figure 6 (b) provides an example using *Under(Quebec)*. Here, all the values surrounded by the dashed line should be protected.

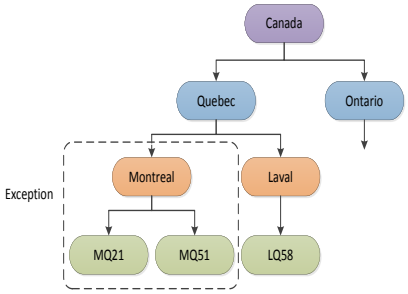

Figure 7. An Authorization Exception.

Definition 3: For any dimension hierarchy level L_i, and any attribute value V_i, the function Under(L_i, V_i) is defined as Under(L_i, V_i) = $\{V_j$: such that $V_j \preceq V_i$ holds$\}$, where L_i is the prohibited dimension level and V is the root value of the restriction.

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 7 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 4.

Definition 4: 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 5.

Definition 5: 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. Implementing Below and Under Functions

To efficiently implement Below and Under functions, a number of additional algorithms and data structures are needed in order to manipulate dimension hierarchies and to retrieve attribute values. These structures are initialized once the server receives a query and are subsequently exploited by the DBMS engine during query resolution. Below, we describe the core structures, along with the methods required to implement the associated functions efficiently.

1) Implementation of the Below Function: We begin by giving a brief description of the primary data structures utilized during function execution. The mapGraph is a suite

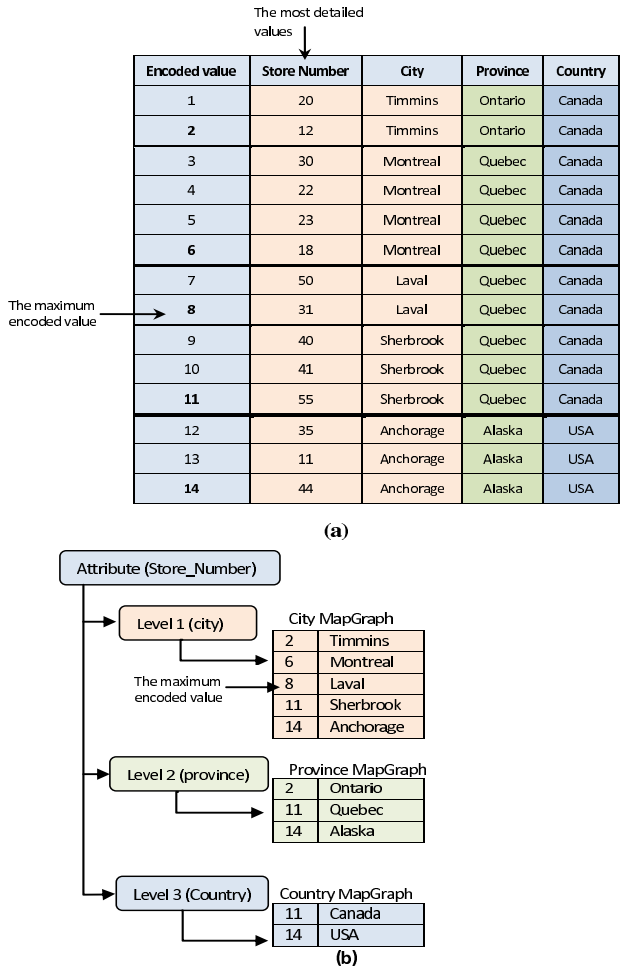

Figure 8. (a) The sorted data of the Store Dimension Table, (b) The corresponding mapGraph.

of algorithms and data structures for the manipulation of attribute hierarchies in "real time" [24]. mapGraph builds upon the notion of *hierarchy linearity* [25]. Briefly, a hierarchy is considered linear if there is a contiguous range of values R_j on dimension attribute A_j that may be aggregated into a contiguous range R_i. Informally, this implies that the totals for a range of values within a child aggregation level are equivalent to those of some range of values at the parent level. As a concrete example, the combined sales totals for the individual *months* of January, February, and March would be exactly equivalent to those of the first *quarter* of the calendar year. To establish the linearity of each dimension hierarchy a sorting technique is employed, with data subsequently stored at the finest level of granularity. If a Time hierarchy is present, for instance, transactional data would be stored at the Day level rather than at the Year level. A compact, in-memory lookup structure is then used to support efficient real time transformations between arbitrary levels of the dimension hierarchy. For example, Figure 8(a)

```
Select Product.Name, Store.province,
    Sum(sales)
From Product, Store, Sales
Where Product.product_ID = Sales.product_ID
  AND Store.store_ID = Sales.store_ID AND
   Store.Contry = 'Canada' AND
   (Product.Name = 'LN*' AND
    Product.price >= 24000)
Group by Product.Name, Store.province
Order by Product.Name, Store.province
```

Listing 3. Simple SQL OLAP Query

depicts the sorted data of the Store dimension table for the data cube depicted in Figure 1, while Figure 8(b) illustrates the corresponding mapGraph for the Store dimension hierarchy.

Each record in the mapGraph consists of two values — a native attribute representation (e.g., values of attribute Type in the Store dimension) and an integer value that represents the corresponding maximum encoded value in the primary attribute. We will look at a concrete example. While the city of Timmins has two stores, Store 1 and Store 2, the city of Montreal has four stores, Store 3 through Store 6. Using this structure, one can easily, and efficiently, perform a mapping from the most detailed encoded level value (i.e., Store_Number) to the corresponding sub-attribute value (i.e., attribute level values), and vice versa. For instance, Store 13 is located in the city of Anchorage and, as a consequence, in the State of Alaska in the USA (Alaska and the USA have a maximum Store_Number = 14).

While a number of commercial products and several research papers do support hierarchical processing for simple hierarchies, specifically those that can be represented as a balanced tree, mapGraph is unique in that it can enforce linearity on unbalanced hierarchies (i.e., optional nodes), as well as hierarchies defined by many-to-many parent/child relationships. The end result is that users may intuitively manipulate complex cubes at arbitrary granularity levels and can navigate easily through dimension levels.

Now recall the policy in Example 1. Suppose that Alice sent the query in Listing 3, which summarizes the total sales of stores in Canada for products of price 24K or more, and whose names start with "LN". To define the authorization objects, the Below function is invoked, taking the prohibited level (i.e., Store.Province) as an argument and using the mapGraph to retrieve a list consisting of the specified level and all the lower levels of the associated dimension hierarchy (i.e., Province, City, and Store Number). Clearly, the prohibited level is in the returned list, and as a result the query should be rejected.

2) Implementation of the Under Function: The Under function is invoked when the policy is less restrictive, as is the case in Example 2. Suppose that Alice now resends the query in Listing 3, assuming this less restrictive policy.

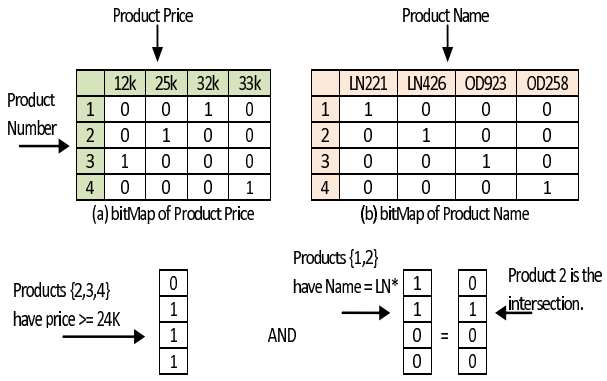

Figure 9. The bitMap of Product Price and Product Name.

To answer or reject the query, we have to determine if the user has requested access to the authorization objects. We note that the user query has two dimension conditions, the first on Product (Product.Name = 'LN*' AND Product.price $\geq$ 24000) and the second on Store (i.e., Store.Country = 'Canada'). The first condition will be ignored, since there is no restriction on the Product dimension in the current policy. For the second condition, we need to determine if the province of Quebec is in Canada (i.e., if Quebec is *Under* Canada). If so, we can say that the user has attempted to access a restricted data element and, as a consequence, the query should be rejected. By using the Under function, we retrieve the encoded values of Canada and Quebec from the mapGraph structure. If there is an intersection between the two, we know that Quebec is Under Canada. In our example, Canada has stores encoded with identifiers 1 through 11, and Quebec has stores encoded as 3 through 11. Clearly, there is an intersection between them, which means that the user has requested access to restricted data.

Definition 6: If there is an intersection between Under(L_i) values — where L_i is the prohibited level — and Under(E_j) values — where E_j is the requested level — then the query should not be executed directly.

As noted, the mapGraph is very useful when hierarchical attribute levels are involved in the OLAP query. However, in some cases, it is a non-hierarchical attribute that is restricted (e.g., the Name or Price attributes of Product). In this case, the FastBit [26] bitmap index structure allows us to easily find those records that contain specific values on a given attribute in the dimension. For example, suppose the Product dimension has four records (i.e., four products), numbered 1 through 4, and a non-hierarchy attribute (Product Price) is added to the Product dimension attributes. The bitmap index for the Product Price attribute is illustrated in Figure 9(a), while Figure 9(b) illustrates the bitmap index for the Product Name. Each index consists of four bit strings (number of products), each of length four. In each string, the 1's indicate the encoded values for the primary key.

```
Selection:
     Product.Name, Store.province, Sum(sales)
Condition:
     Store.Province = 'Quebec' AND
     (Product.Name = 'LN*' AND
      Product.price >= 24000)
From:
     Sales
```

Listing 4. A Query in Simple Form

Now, suppose that Alice is restricted from accessing all products whose names start with "LN". Further, we will assume that she resends the query in Listing 3. Since the Product Price and the Product Name are non-hierarchical attributes, we use their bitmap indexes to retrieve the base level numbers for those products, and then determine if there is an intersection between the two. Figure 9 illustrates how to identify those products whose Name starts with "LN" AND whose price $\geq$ 24K. The array at the lower left represents the products of price $\geq$ 24K, in this case Products 2, 3, and 4. The array in the center represents the products with names starting with "LN". Products 1 and 2 are identified in this case. The AND operator determines the intersection between them, with the final result shown in the last array. As we can see, there is in fact a non-empty intersection (i.e., Product_Number 2 has a price $\geq$ 24K and a name starts with "LN"); thus, the query should be rejected.

Algorithm 1 summarizes the logic of the checking process. In short, we determine if the attributes within the selection predicate(s) are hierarchical in nature. For example, a restriction on a time value (e.g., Day-Month-Year) would be hierarchical in nature, while a restriction on an attribute such as colour or weight would not be associated with any form of hierarchy. Based upon this understanding, query values would be analyzed relative to the information contained in the mapGraph data structure (which stores hierarchical relationship information) or bitmap indexes.

E. 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 4 represents the same query shown in Listing 3. 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

input : The policy condition S, and the Query Q
output: Returns True if Q is valid, False otherwise

Initialize the mapGraph (hM) and the bitMap (fB) if they have not already been initialized;
Let QA be the query attributes;
if *S has a hierarchy attribute* **then**
  Let SR be the range of S using hM;
end
else
  Let SR be the range of S using fB;
end
foreach *attribute* a_i *in QA* **do**
  if a_i *is hierarchy attribute* **then**
    Get the range of a_i QR using hM;
    if $QR \cap SR \neq \emptyset$ **then**
      Return False;
    end
  end
  else if a_i *is non-hierarchy attribute* **then**
    Get the range of a_i QR using fB;
    if $QR \cap SR \neq \emptyset$ **then**
      Return False;
    end
  end
end
Return True;

Algorithm 1: The procedure of Policy Class 2

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 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 Authorization 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

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

Listing 5. Authorization Strategy as per Rule 1

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

Listing 6. Authorization Strategy as per Rule 4

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 5, which summarizes the total sales of Canada's stores in 2011 for furnitures products, and she is restricted from accessing/reading provincial sales, 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 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 4, 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 6. 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 7. 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 8.

The complete processing logic for Policy Class 2 (i.e., Rule 3, Rule 4, and Rule 5) is encapsulated in Algorithm 2. 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

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

Listing 7. Simple OLAP Query 2

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

Listing 8. Authorization Strategy as per Rule 5

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.

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 9, the query should be modified as shown in Listing 10.

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 ∈ Under(Ev)* **then**
      Allow the query to execute without modification;
    end
    else if *Ev* ∈ 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 2: The procedure of Policy Class 2

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

Listing 9. Simple OLAP Query 3

The associated query summarizes the sales of provinces in 2011 for outdoor products. 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*

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

Listing 10. Authorization Strategy as per Rule 7

```
Selection:
    Store.City, Product.Type, SUM(sales)
Condition:
    Store.City = 'Montreal' AND
    Product.Type = 'Indoor' AND
    Time.Year = 2011
From:
    Sales
```

Listing 11. Authorization Strategy as per Rule 9

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).*

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 11, 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 3 illustrates the full processing logic for Policy Class 4 (Rule 8, Rule 9, Rule 10, and Rule 11). In short, the authorization 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 ∈ 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 ∈ Under(P)) AND (Lv == Ev OR Ev ∈ 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 3: The procedure of Policy Class 4

F. Authorization Rule Summary

The preceding sections have formalized the authorization 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.

V. EXPERIMENTAL RESULTS

Because of the potential to impact overall query resolution time, considerable effort has been made to ensure the efficiency of the authorization logic, including the exploitation of compact data structures such as mapGraph and the FastBit bitmap indexes. Moreover, the analysis of policy classes is based primarily upon a restricted set of IF/ELSE cases that, in turn, manipulate a small in-memory Authentication Database. Given the motivation to include OLAP-aware authorization mechanisms within fully functional database management systems, however, it is important to actually verify that our checking approach does not in fact seriously degrade query performance. As noted earlier, the authorization framework has been incorporated into a DBMS prototype specifically designed for OLAP storage and analysis. For testing purposes, however, this integrated environment is not necessarily ideal as it is difficult for the reader to determine if the balance between checking and execution is reflective of current systems. Furthermore, it may not be obvious that our authorization model has the potential for integration with standard database servers.

For this reason, we have coupled our framework with MonetDB, a popular open source database management system [27]. MonetDB is a column store DBMS, as opposed to the more familiar row-based systems. Column stores are particularly well suited to OLAP workloads as the ability to efficiently extract only the columns of interest can significantly improve IO performance. Note that in this case, the job of MonetDB is simply to provide execution services — all authorization services are provided by the subsystems defined in this paper. In the current case, we utilize the Star Schema Benchmark (SSB) [28], a variation of the original TPC-H benchmark augmented for OLAP settings. In short, SSB consist of a central Fact Table and four dimension tables, with a set of 13 analytics queries executed against the data. Queries are divided into four query categories, with each category providing increasingly sophisticated restrictions on the associated dimensions. A full listing of the queries can be found in the Appendix. The SSB is particularly valuable in the current context as it provides a common mechanism by which to assess the kinds of queries — in terms of both form and complexity — that one would actually expect to encounter in OLAP settings. As a final note, we stress that Monet does not provide an internal OLAP-aware conceptual model. To ensure compatibility with the mechanisms described throughput this paper, it was necessary to develop SQL conversion middleware, a significant research effort of its own. The details of the middleware architecture are the subject of an upcoming submission.

For the following tests, we have used the SSB generator (with default settings) to produce a Fact table of 180 million records, with each dimension housing between 60,000 and one million records. The experiments themselves were run on a 12-core AMD Opteron server with a CPU core frequency of 2100 MHz, L1/L2 cache size of 128K and 512K respectively, and a shared 12MB L3 cache. The server was equipped with 24 GB of RAM, and eight 1TB Serial ATA hard drives in a RAID 5 configuration. The supporting OS was CentOS Linux Release 6.0. All OS and DBMS caches were cleaned between runs.

A set of four simple but typical authorization policies was created, as follows. We generated one constraint across a full dimension (i.e., the Product.Part is restricted), a second constraint on an attribute, along with an exception (i.e., the attribute s_region is restricted with an s_province exception), a third constraint on an attribute value with an exception value (i.e., d_year $<$ 2009 is restricted except d_year = 2005 or 2006), and the last constraint prohibits access to a cuboid as a whole. Essentially, policies were designed in keeping with the logic of Section IV, but adapted to the specific attributes of the SSB schema.

In terms of the results, we have isolated each of the four query classes and show authorization processing versus the subsequent query execution time in Figure 10, Figure 11, Figure 12, and Figure 13. We note that all queries violated one or more security policies and that these violations were identified and appropriately processes by the authorization module (each authorization decision was manually verified

for correctness). For those that were not candidates for re-writing (i.e., they were simply rejected), the query execution time is still listed so as to give the reader a better sense of the relative balance between checking and execution. A few additional points are also worth noting. First, the ratio of checking time to execution time varies considerably, depending on the specification of the underlying query. In particular, many OLAP queries are *very* expensive to execute, given the amount of sorting and aggregation involved. In this case, Query Classes 1 and 3 have restrictive selection constraints (with the exception of the Query 3.1), thereby reducing the size of intermediate results and, in turn, dramatically limiting aggregation costs. Overall, execution times range from about half a second for Query Class 3 to more than 30 seconds for Query Class 2, where large intermediate results produce massive aggregation costs. As the database gets larger, of course, these execution times will continue to grow.

Second, the checking costs are quite modest, in the range of 100-400 milliseconds. More importantly, the size of the underlying database has no effect upon the checking costs, as only the cube meta data is inspected. In other words, it does not matter that 180 million records exist in the database as authorization decisions are not based upon this data. Rather, only schema information (e.g., cubes, dimensions, hierarchies) and policy specifications (i.e., restrictions and exceptions) are required during this process. In practice, it is extremely unlikely that, from an end user's perspective, authorization costs would have a tangible impact on database access and analysis.

As a final point, we re-iterate that column stores are well suited to this environment, given their ability to minimize I/O costs. The execution times for traditional row store database servers can be one to two orders of magnitude larger [29]. The authors have, in fact, evaluated the current test cases on the open source row-based PostgreSQL DBMS and validated these ratios. In such environments, the ratio of checking to execution costs would be far more extreme, with execution costs being dozens or even hundreds of times larger than checking costs.

VI. 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.

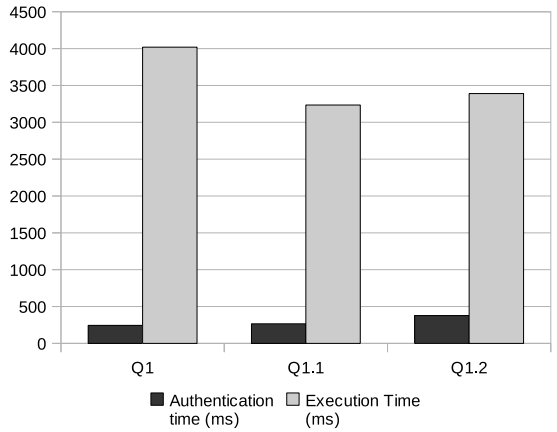

Figure 10. Performance for SSB schema, Query 1 category

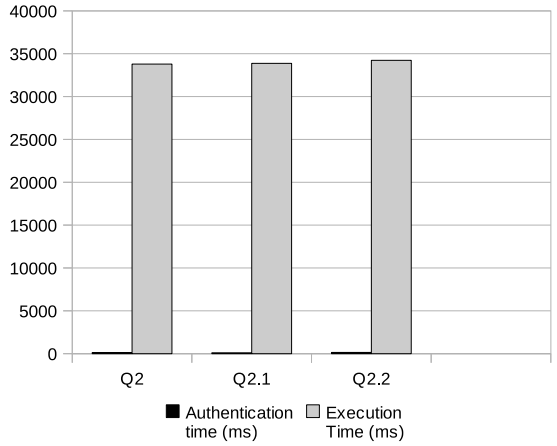

Figure 11. Performance for SSB schema, Query 2 category

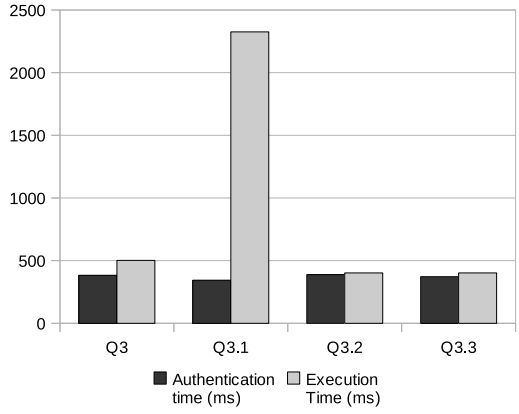

Figure 12. Performance for SSB schema, Query 3 category

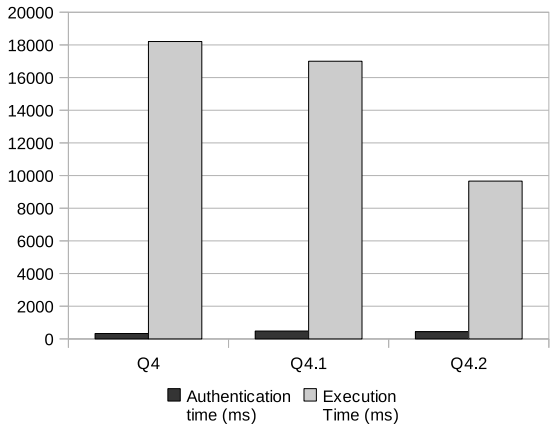

Figure 13. Performance for SSB schema, Query 4 category

While the 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 contemporary DBMS product. To this end, we combined the framework with MonetDB, an open source DBMS that provides efficient column oriented services. Using the Star Schema Benchmark, we showed that for common OLAP queries, authentication and authorization services represent a negligible impact on overall query execution and, in fact, that there is no relationship between authorization and execution costs. For this reason, we believe that our methods are viable for not only OLAP-specific database management systems, but more conventional platforms as well.

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. Eavis and A. Altamimi, "OLAP authentication and authorization via query re-writing," in *The Fourth International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA)*, 2012, pp. 130–139.

[2] P. P. Griffiths and B. W. Wade, "An authorization mechanism for a relational database system," *ACM Transactions on Database Systems*, vol. 1, no. 3, pp. 242–255, Sep. 1976.

[3] Biba, "Integrity considerations for secure computer systems," *MITRE Co., technical report ESD-TR 76-372*, 1977.

[4] R. Sandhu, D. Ferraiolo, and R. Kuhn, "The NIST model for role-based access control: towards a unified standard," in *Proceedings of the fifth ACM workshop on Role-based access control*, ser. RBAC '00, 2000, pp. 47–63.

[5] 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.

[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] K. Khajaria and M. Kumar, "Modeling of security requirements for decision information systems," *SIGSOFT Software Engineering Notes*, vol. 36, no. 5, pp. 1–4, Sep. 2011.

[9] M. Kumar, A. Gosain, and Y. Singh, "Stakeholders driven requirements engineering approach for data warehouse development," *JIPS*, vol. 6, no. 3, pp. 385–402, 2010.

[10] Y. Singh, A. Gosain, and M. Kumar, "From early requirements to late requirements modeling for a data warehouse," *Networked Computing and Advanced Information Management, International Conference on*, vol. 0, pp. 798–804, 2009.

[11] 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.

[12] 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.

[13] ——, "Administering permissions for distributed data: factoring and automated inference," in *Proceedings of the fifteenth annual working conference on Database and application security*, ser. Das'01, 2002, pp. 91–104.

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

[15] S. Rizvi, A. O. Mendelzon, S. Sudarshan, and P. Roy, "Extending query rewriting techniques for fine-grained access control," in *ACM Special Interest Group on the Management of Data*, ser. SIGMOD '04, 2004, pp. 551–562.

[16] "The Virtual Private Database," June 2012, http://www.oracle.com/technetwork/database/security/index-088277.html.

[17] "Microsoft Analysis Services," June 2012, http://www.microsoft.com/sqlserver/2008/en/us/analysis-services.aspx.

[18] 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.

[19] V. Harinarayan, A. Rajaraman, and J. Ullman, "Implementing data cubes efficiently," in *ACM Special Interest Group on the Management of Data*, ser. SIGMOD '96, 1996, pp. 205–227.

[20] 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.

[21] T. Eavis, H. Tabbara, and A. Taleb, "The NOX framework: native language queries for business intelligence applications," in *Data Warehousing and Knowledge Discovery (DaWak)*, 2010, pp. 172–189.

[22] "SQL database engine," June 2012, http://www.sqlite.org.

[23] "Definition of the XML document type declaration from Extensible Markup Language (XML) 1.0 (Fifth Edition)," June 2012, http://www.w3.org/TR/xml/.

[24] T. Eavis and A. Taleb, "Mapgraph: efficient methods for complex olap hierarchies," in *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, ser. CIKM '07, 2007, pp. 465–474.

[25] V. Markl, R. Bayer, B. Forschungszentrum, and R. Bayer, "Processing relational OLAP queries with UB-Trees and multidimensional hierarchical clustering," in *In Proceedings of DMDW 2000*, 2000, pp. 5–6.

[26] M. Zaker, S. Phon-amnuaisuk, and S. cheng Haw, "An adequate design for large data warehouse systems: Bitmap index versus B-tree index," 2008.

[27] "MonetDB column store database engine," June 2012, http://www.monetdb.org.

[28] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak, "Performance evaluation and benchmarking," R. Nambiar and M. Poess, Eds., 2009, ch. The Star Schema Benchmark and Augmented Fact Table Indexing, pp. 237–252.

[29] D. J. Abadi, S. R. Madden, and N. Hachem, "Column-stores vs. row-stores: how different are they really?" in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '08, 2008, pp. 967–980.

APPENDIX

Below, we provide a listing of the 13 queries found in the Star Schema Benchmark.

1. **select** sum(lo_extendedprice*lo_discount) as revenue
   from lineorder, date
   where lo_orderdate = d_datekey and
   d_year = 1993 and
   lo_discount between 1 and 3
   and lo_quantity < 25

   1. 1) **select** sum(lo_extendedprice*lo_discount) as revenue
      from lineorder, date
      where lo_orderdate = d_datekey and
      d_yearmonthnum = 199401 and
      lo_discount between 4 and 6 and
      lo_quantity between 26 and 35

   1. 2) **select** sum(lo_extendedprice*lo_discount) as revenue
      from lineorder, date
      where lo_orderdate = d_datekey and
      d_weeknuminyear = 6 and
      d_year = 1994 and
      lo_discount between 5 and 7 and
      lo_quantity between 26 and 35

2. **select** sum(lo_revenue), d_year, p_brand1
   from lineorder, date, part, supplier
   where lo_orderdate = d_datekey
   and lo_partkey = p_partkey and
   lo_suppkey = s_suppkey and
   p_category = 'MFGR#12' and
   s_region = 'AMERICA'
   group by d_year, p_brand1
   order by d_year, p_brand1

   2. 1) **select** sum(lo_revenue), d_year, p_brand1
      from lineorder, date, part, supplier
      where lo_orderdate = d_datekey and
      lo_partkey = p_partkey and
      lo_suppkey = s_suppkey and
      p_brand1 between 'MFGR#2221' and
      'MFGR#2228' and
      s_region = 'ASIA'
      group by d_year, p_brand1
      order by d_year, p_brand1

   2. 2) **select** sum(lo_revenue), d_year, p_brand1
      from lineorder, date, part, supplier
      where lo_orderdate = d_datekey and
      lo_partkey = p_partkey and
      lo_suppkey = s_suppkey and
      p_brand1 = 'MFGR#2221' and
      s_region = 'EUROPE'
      group by d_year, p_brand1
      order by d_year, p_brand1

3. **select** c_city, s_city, d_year, sum(lo_revenue) as revenue
   from customer, lineorder, supplier, date
   where lo_custkey = c_custkey and
   lo_suppkey = s_suppkey and
   lo_orderdate = d_datekey and
   c_nation = 'UNITED STATES' and
   s_nation = 'UNITED STATES' and

d_year >= 1992 and
d_year <= 1997
where c_city, s_city, d_year
where d_year asc, revenue desc

3. 1) **select** c_nation, s_nation, d_year, sum(lo_revenue) as revenue
from customer, lineorder, supplier, date
where lo_custkey = c_custkey and
lo_suppkey = s_suppkey and
lo_orderdate = d_datekey and
c_region = 'ASIA' and
s_region = 'ASIA' and
d_year >= 1992 and
d_year <= 1997
group by c_nation, s_nation, d_year
order by d_year asc, revenue desc

3. 2) **select** c_city, s_city, d_year, sum(lo_revenue) as revenue
from customer, lineorder, supplier, date
where lo_custkey = c_custkey and
lo_suppkey = s_suppkey and
lo_orderdate = d_datekey and
(c_city='UNITED KI1' or c_city='UNITED KI5') and
(s_city='UNITED KI1' or s_city='UNITED KI5') and
d_year >= 1992 and
d_year <= 1997
group by c_city, s_city, d_year
group by d_year asc, revenue desc

3. 3) **select** c_city, s_city, d_year, sum(lo_revenue) as revenue
from customer, lineorder, supplier, date
where lo_custkey = c_custkey and
lo_suppkey = s_suppkey and
lo_orderdate = d_datekey and
(c_city='UNITED KI1' or c_city='UNITED KI5') and
(s_city='UNITED KI1' or s_city='UNITED KI5') and
d_yearmonth = 'Dec1997'
group by c_city, s_city, d_year
order by d_year asc, revenue desc

4. **select** d_year, s_nation, p_category, sum(lo_revenue - lo_supplycost) as profit
from date, customer, supplier, part, lineorder
where lo_custkey = c_custkey and
lo_suppkey = s_suppkey and
lo_partkey = p_partkey and
lo_orderdate = d_datekey and
c_region = 'AMERICA' and
s_region = 'AMERICA' and
(d_year = 1997 or d_year = 1998) and
(p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2')
group by d_year, s_nation, p_category
order by d_year, s_nation, p_category

4. 1) **select** d_year, c_nation, sum(lo_revenue - lo_supplycost) as profit
from date, customer, supplier, part, lineorder
where lo_custkey = c_custkey and
lo_suppkey = s_suppkey and
lo_partkey = p_partkey and
lo_orderdate = d_datekey and
c_region = 'AMERICA' and
s_region = 'AMERICA' and
(p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2')
group by d_year, c_nation
order by d_year, c_nation

4. 2) **select** d_year, s_city, p_brand1, sum(lo_revenue - lo_supplycost) as profit
from date, customer, supplier, part, lineorder
where lo_custkey = c_custkey and
lo_suppkey = s_suppkey and
lo_partkey = p_partkey and
lo_orderdate = d_datekey and
c_region = 'AMERICA' and
s_nation = 'UNITED STATES' and
(d_year = 1997 or d_year = 1998) and
p_category = 'MFGR#14'
group by d_year, s_city, p_brand1
order by d_year, s_city, p_brand1