A Comprehensive Data Security Framework for OLAP Domains

Ahmad Altamimi

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fullfilment of the Requirements
for the Degree of Doctor of Philosophy at
Concordia University
Montreal, Quebec, Canada

January, 2014

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: Ahmad Altamimi

Entitled: A Comprehensive Data Security Framework for OLAP Domains

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

signed by the final examining committee:

Deborah Dysart-Gale	Chair
Robert Wrembel	External Examiner
Dhrubajyoti Goswami	Examiner
Rajagopalan Jayakumar	Examiner
Benjamin Fung	Examiner
Todd Eavis	Supervisor

Approved By _____
Chair of Department or Graduate Program Director

_____
Dean of Faculty

ABSTRACT

A Comprehensive Data Security Framework for OLAP Domains

Ahmad Altamimi

Online Analytical Processing (OLAP) has become an increasingly important and prevalent component of enterprise Decision Support Systems. OLAP is associated with a data model known as a Cube, a multi-dimensional representation that allows for the extraction and intuitive visualization of broad patterns and trends that would otherwise not be obvious to the user. One must note, however, that not all of the collected data should be universally accessible. Specifically, DW/OLAP systems almost always house confidential and sensitive data — identification information, medical data, or even religious beliefs and ideologies — that must, by definition, be restricted to authorized users. In this thesis, we provide models and algorithms for protecting the data in multi-dimensional data cube spaces.

To this end, the thesis addresses three distinct but related themes. In the opening part of this study, we propose an authentication and authorization framework that builds upon an algebra designed specifically for OLAP domains. It relies on robust query re-writing rules to ensure consistent data access across all levels of the conceptual data cube model. In the second part, we present a framework for controlling malicious inferences caused by unprotected access to coarser level aggregations. Our framework prevents complicated inferences through a combination of initial query restrictions and the removal of the remaining inferences. In the final part, we enhance the core framework with an object-oriented security design model and client side language extensions that collectively produce a more intuitive and usable infrastructure.

The purpose of this study is to design a comprehensive end-to-end framework for OLAP security that is flexible, intuitive, and powerful. In short, the framework allows administrators to associate security policies with an intuitive conceptual model that maps directly to the model that users see. Restrictions then can be propagated transparently from users to all the hierarchical data. Moreover, the framework provides an automatic form of inference control that is fast enough in practice to not affect query time.

To ground our conceptual work, we have integrated our research themes on the top of an OLAP-specific DBMS server (Sidera). Sidera gives us the opportunity to explore performance and correctness issues that would not be possible without such direct access to a DBMS. In addition, we have evaluated its efficiency with a pair of common industrial DBMS, a row-based DBMS (PostgreSQL) and a column-store DBMS (MonetDB). The evaluation is done using two common benchmarks (e.g., SSB and APB). The results show the ratio of checking time to execution time varies considerable, depending on the specification of the underlying query. These times are acceptable, particularly given that checking costs do not grow with data set size.

ACKNOWLEDGEMENTS

I owe my foremost gratitude to my advisor, Todd Eavis, for providing tremendous guidance and support. He has always been a source of inspiration and motivation for me. He has excellent insight at suggesting research problems that are well-motivated and also theoretically challenging. My interactions with him have greatly shaped my perspectives about research. I am especially thankful for his support and encouragement during the initial stage of my PhD program.

I thank my committee members for their patience and guidance: Dr. Jayakumar Rajagopalan, Dr. Goswami Dhrubajyoti, and Dr. Benjamin Fung.

Also, I thank my parents and sisters who have always been very understanding and encouraging, showering unconditional love and affection upon me all the time. My gratitude to them is beyond expression.

Special thanks are extended to my father in law Dr. Ahmed Omari for his kindness, support and encouragement. He does not realize how much I have learned from him. I am really glad that I have one like him in my life.

Finally, I would like to take this opportunity to personally thank all of my colleagues and friends who have been so supportive and giving of their time especially Ahmad Taleb, Yasser Mahmood, Nour Eddine Caidi, Mohammad Solihat, and Adnan Abu Eid.

To My Wife Manar, and My Daughters Sarah and Ayah.

Table of Contents

List of Figures

Listings

Chapter 1

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 [26], 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, such as social security numbers, credit cards information, or medical history 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 EUs Safe Harbour Law [43]. These laws usually require strict technical security measures for guaranteeing privacy, and failure to comply with them is normally strongly sanctioned, with severe penalties being imposed.

In this context, organizations make promises to keep the data they collect and store secret. But keeping one's promises is usually easier said than done. The data may be attacked in various ways: from outside through penetrating the security defense system or from inside by violating the privacy commitments under which the data have been collected. The outsider attacks can be prevented by using a variety of defensive mechanisms (e.g., firewalls, or encryption methods). However, the most challenging threats often come from insiders. That is the case because the insider knows what the company's most valuable data assets are, as well as where to look for and how to access them. In a survey of 43 organizations that had experienced a data breach, the Ponemon Institute found that over 88 percent of all cases involved incidents resulting from insider attacks [95]. Thus, organizations need to maintain a high level of security to protect the data from malicious users while, at the same time, ensuring the availability of data to legitimate users. How one prevents privacy breaches cause by inappropriate disclosures while simultaneously ensuring access to valid users is in fact the main topic of this research.

One of the main problems that organizations face in this context is the reconciliation of two contradictory goals: *privacy* and *utility* for analytical purposes. The tension between these goals is clear: perfect privacy (but no utility) can be achieved by simply refusing to answer any queries about the data; perfect utility (but no privacy) can be achieved by answering all queries about the data exactly. Clearly, neither of these extremes is likely to be acceptable in production environments. Thus,

the organization must carefully employ a rich but flexible security model in order to balance the competing goals of privacy and utility.

In fact, any such security model must provide two main countermeasures: Access control and Inference control.

- Access Control is the process that restricts unauthorized users from accessing protected data. It can be thought of as occurring in two phases: Authentication and Authorization. Authentication is an identity verification mechanism that attempts to determine whether a user has valid credentials to access the system or not. Usually, the client query provides user credentials, which are then received and checked against a list of valid accounts in order to make the authentication decision. If the provided user credentials correspond to a valid account, then the authentication decision is successful and the query is passed on for authorization checking. Otherwise, the query will be rejected. In turn, Authorization is the process of determining if the user has permission to access a certain data resource. The outcome of the authorization process is an authorization decision that permits or denies the user access, based on the user's credentials or privileges.

  Access control techniques in traditional data management systems, such as relational databases, are quite mature. However, these techniques cannot be *directly* applied to OLAP domains. Specifically, OLAP's conceptual data model is considerably more abstract than the relational model used by conventional relational database management systems. The difference in the data models makes it more difficult to express and enforce the security requirements one would commonly see in OLAP settings. In addition, existing access control

mechanisms are largely unaware of more subtle inference attacks, a problem of considerable importance in the OLAP domain.

- Inference Control attempts to prevent attackers from inferring or guessing sensitive data. The malicious inference is a threat that access control methods are insufficient to thwart on their own. For example, it is possible for a user possessing some degree of external knowledge to combine the results of multiple valid queries so as to obtain data that is itself meant to be protected. As a simple example, a user might infer the salary of a second employee by virtue of the fact that she knows the combined salary of both employees. Access control alone cannot capture such an inference, as the total salary would represent seemingly innocent aggregation to the access control mechanism.

  Inference control has been extensively studied in statistical databases and census data since the 1970's [2]. However, most of the proposed techniques demand complicated computations over the entire data set and/or archiving all previous queries. Such requirements often lead to prohibitive performance overhead and storage demands, which is particularly problematic for interactive OLAP systems in which query results may be expected in a matter of seconds or minutes.

1.1 Research Overview

The primary purpose of the research described in this thesis is to understand and provide solutions to the privacy threats in OLAP environments. More specifically, the current research is essentially divided into three stages. In the first stage, we explore the development of a robust OLAP authentication and authorization framework that supports the basic security countermeasures for an OLAP data model known as

a cube. In particular, we seek to provide reliable security mechanisms that support the hierarchical nature of the cube. To do so, we propose a set of concise but robust transformation rules that are used to rewrite queries containing unauthorized data access and thereby ensure that the user only receives the data that he/she is authorized to see. 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 (i.e., alternative aggregation levels). In addition, we have designed and integrated data structures and algorithms utilized by the functions that manipulate the elements of the conceptual data model. To underscore the practical viability of the proposed framework, the experimental evaluation highlights the processing overhead relative to the execution costs of the underlying query. The results demonstrate that the ratio of checking time to execution time is quite small.

In the second stage, we focus on an inference control framework. While many restriction based models adopt a detect-and-remove approach, which typically requires complex computations over the data and is thus too expensive to be applied in OLAP systems, our framework prevents inferences through restrictions. In other words, instead of *detecting* inferences, we *prevent* them by eliminating the source of the inference. This is done by pre-computing the accessible data and then restricting access to only this data. Restrictions can be computed off-line before queires are actually received. Hence, computational costs can be significantly minimized. Moreover, the effectiveness of this approach does not depend on specific types of aggregation functions, external knowledge, or sensitivity criteria.

To demonstrate the validity of our research, we have coupled our framework with two different database management systems, a popular open source column-store DBMS (Monetdb), and an object-relational row-based DBMS (Postgre SQL). We utilize the Star Schema Benchmark (SSB), a variation of the original TPC-H benchmark augmented for OLAP settings, and the APB-1 OLAP Benchmark release II, a widely-known data warehouse benchmark, to conduct a series of experiments. Results show that our methods are efficient in terms of both run-time performance and storage requirements.

In the third and final stage of the research, we focus on the interface between the security components and the database management system itself. Specifically, in order to utilize the proposed authorization and inference prevention mechanisms, it is necessary to extend the languages and/or APIs that permit system administrators and security specialists to define and maintain security policies. To this end, we propose several enhancements to current DBMS software stacks. First, we describe an abstract security design model based upon an Object Oriented paradigm. Fundamental policy classes are defined, along with state and behavior characteristics relevant to the OLAP domain. Next, we discuss language extensions and programmatic APIs that would expose the underlying model to end users and programmers in an intuitive manner. Finally, we present the details of the server-side policy database and policy logic manager, along with a discussion of the issues relevant to the creation of complex Role hierarchies. To underscore the viability of the design model, we provide a small but representative case study using policy examples from the thesis itself.

1.2 Thesis Structure

The thesis is organized as follows. Chapter 2 provides basic background material needed to understand data warehousing and Online Analytical Processing (OLAP), the core characteristics of the conceptual data model, schema mapping issues, and the primary features of column store databases. The succeeding chapters present the core contributions of the thesis. Chapter 3 describes an OLAP-aware framework for query authentication and authorization via query re-writing. Chapter 4 describes extensions required in order to protect OLAP data cubes from malicious inferences, and discusses the implementation and architecture. A set of algorithms is also presented for each of the primary operations. Chapter 5 then discusses the OO design model, including the use of declarative language enhancements and a client side API to support graphical design tools. Finally, in Chapter 6, we offer final conclusions and briefly describe possible future work.

Chapter 2

Background Material

2.1 Introduction

Organizations collect data for social and commercial purposes and typically store that data in the enterprise's Data Warehouse (DW). In short, a Data Warehouse is a repository for multiple heterogeneous data sources, organized under a unified schema in order to facilitate management decision making [57]. Data warehouse technology includes data cleaning, data integration, and Online Analytical Processing (OLAP). The latter is associated with analysis techniques such as summarization, consolidation, and aggregation, as well as the ability to view information from different perspectives.

In warehousing environments, data is often represented using a data model known as a *Cube*, a multi-dimensional representation of the core measures and relationships within the associated organization. We note that large portions of the data cube are either confidential or sensitive and must, by definition, be restricted to authorized users. However, the data may still be attacked by malicious users in order to access or infer sensitive knowledge from the database. Such occurrences, which significantly impact the trustworthiness and reliability of the OLAP model, have motivated the

development of recent approaches that seek to devise meaningful privacy preserving OLAP techniques [109, 93, 44, 4, 38, 37].

In this chapter, we are going to introduce the primary concepts that we need to be familiar with before discussing the details of our work. Section 2.2 gives an overview of a typical Data Warehouse and its architecture. Section 2.3 introduces OLAP systems and the canonical data cube. Section 2.4 then looks at the architecture of a conventional relational OLAP server. The mapping of logical to conceptual data objects is then presented in Section 2.5. A data structure for the management of hierarchical OLAP attributes is described in Section 2.6. Section 2.7 discusses the primary features row and column store databases, while Section 2.8 concludes the chapter with brief summary.

2.2 The Data Warehouse

The Data Warehouse (DW) is a repository of organizational data and is designed to facilitate reporting and analysis within the organization [54]. In a warehouse, the data is often a combination of multiple, and usually varied, sources that are then combined into one comprehensive and easily manipulated database. As the data enters the warehouse, it is cleaned and transformed and stored with a fully integrated structure and format. The transformation process may involve conversion, summarization, filtering and condensing of data. Multiple technologies are utilized in the analysis process, including the generation of "canned queries", data mining, and OLAP [26].

2.2.1 Architectural Overview

Many researchers and practitioners support the notion that a data warehouse architecture can be formally understood as a series of layered materialized views, with data from one layer derived from data of a lower level. Data sources, also called *operational databases*, form the lowest such layer. The central component of the architecture is the global (or primary) data warehouse (i.e., the warehouse proper). Ultimately, it maintains a historical record of the data that results from the transformation, integration, and aggregation of detailed data found in the data sources. The topmost layer is the local, or client, warehouses which contain highly aggregated data, directly derived from the global warehouse. There are various forms of local warehouses, including the data marts often associated with OLAP databases. These OLAP-centric systems may utilize either relational database systems or proprietary multidimensional data structures.

All the data warehouse components, processes, and data are — or at least should be — tracked and administered from some form of metadata repository. This repository serves as an aid both to the administrator and the designer of the warehouse. Figure 2.1 gives a rough overview of the typical data warehouse architecture. A brief listing of its physical components would include the following:

- **Sources**. The data sources (or operational DBs) are often relational databases but may include information from other sources (text, XML, bar code readers, etc.). They are designed for "every day", data entry purposes and are not well suited to analytical queries.

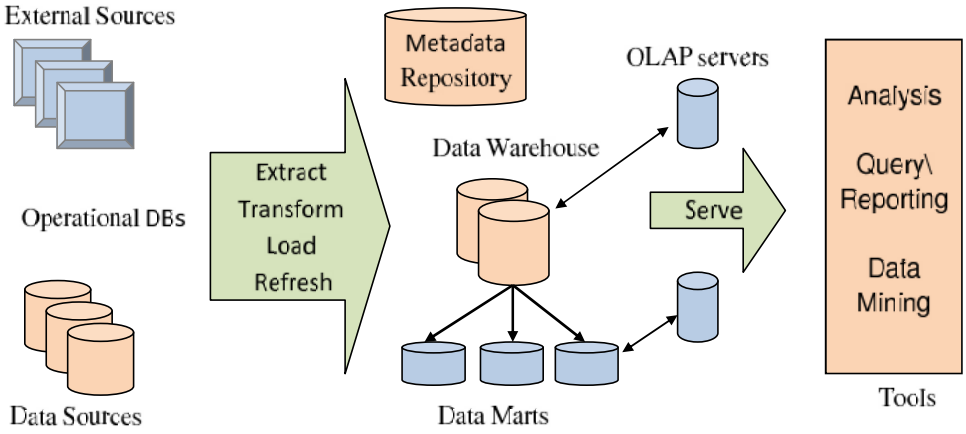


Figure 2.1: Data Warehouse architecture

- **Data Loading**. The Extract, Transform, and Load (ETL) tools are the backbone of the data warehouse input process. They are used to copy data from operational data sources to the data warehouse and eventually on to the data marts.

- **Core Warehouse**. The central warehouse database is the cornerstone of the architecture. It can be extremely large in practice, primarily as a consequence of storing historical data at a very granular level of detail. For example, every "sale" that has ever occurred in an organization may be recorded and subsequently related to various dimensions or entities of interest. This allows the data to be sliced and diced, summed and grouped in an almost limitless number of ways.

- **Data Marts**. The Data Marts are subsets of the data warehouse. Each Data Mart can contain different combinations of tables, columns and rows from the Enterprise Data Warehouse. They enable faster response to queries because the volume of the managed data is much smaller than in the data warehouse, and

the queries can be distributed between different machines. There are a number of design types for the data mart; the star schema and snowflake schema are the most common models [25].

- **Client Tools**. The principal purpose of data warehousing is to provide information to users in support of the decision making process. These users interact with the data warehouse using front-end tools, which typically provide a graphical, interactive interface that obscures much of the overwhelming detail and structure of the underlying warehouse.

2.2.2 The Star Schema

The *Star Schema* is perhaps the simplest data warehouse design, and consists of two basic types of tables: *Facts* and *Dimensions*. The fact table typically contains a large number of rows, sometimes in the hundreds of millions or even billions of records. These records provide us with the facts or *measures* that become the objects of analysis. Each of these numeric measures depends in turn on a set of dimensions, which provide the context for the measure. For example, Figure 2.2 shows a part of the star schema for the data warehouse benchmark known as APB-1 release II [8]. This star schema consists of one fact table ("Sales") that is joined to four dimension tables: Product, Customer, Time, and Channel. When a unit or item is sold a new measure is entered in the fact table (i.e., "Units_Sold") and linked with dimensions via the relevant *foreign keys*: product code, customer ID, and time ID. Taken together, the dimensions are assumed to uniquely determine the measure. Thus, the multidimensional data model views a measure as a value in the multidimensional space of dimensions.

Each dimension is in turn described by a set of *attributes*. For example, the Product dimension consists of six attributes: division, line, family, group, class, and code. Moreover, the attributes of a dimension may be related via a *hierarchy* of relationships that aid in summarization. In the above example, the product information would often contain a hierarchy that separates products into divisions such as food, drink, and non-consumable items, with each of these divisions further subdivided a number of times until reaching the lowest level (i.e., "Product Code").

Dimension are typically de-normalized (i.e, not all table redundancy is removed), with dimension hierarchies represented within a single physical dimension table. At query time, each de-normalized dimension table is joined to the fact table as necessary. In this scenario, de-normalizing the dimension tables significantly decreases the number of costly joins that would otherwise be required with a normalized schema. Since the dimension tables are comparatively small when compared to the enormous fact tables, and since updates are typically performed via system controlled ETL processes, the redundancy produced by the de-normalization process is of little interest in most OLAP contexts. It is worth noting that when standard normalization is employed, a *Snowflake Schema* is created. In this less common model, dimensional hierarchies are explicitly represented with multiple physical tables.

In addition to the fact and dimension tables, data warehouses store selected summary tables containing pre-aggregated data. In the simplest case, this summary data corresponds to the aggregation of the fact table on one or more selected dimensions.. Pre-aggregated data can be represented in the database in at least two ways. Let us consider, for example, how a summary table(s) from the star schema of Figure 2.2 could represent total sales "by product by year". We could potentially represent

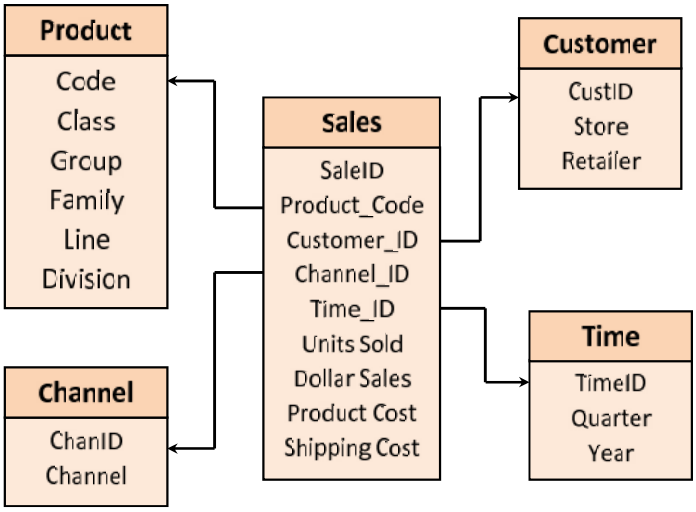


Figure 2.2: Logical schema of the data warehouse benchmark APB-1 release II

the summary data by encoding the aggregated tuples in the same fact table without adding any new tables. This may be accomplished by adding new level/summary fields in the fact table and using nulls for non-summary rows. However, this approach tends to create a bloated fact table (most rows do not contain summary data) and an awkward schema design. Alternatively, we can create a summary table that essentially functions as a very compact supplemental fact table.

2.3 OLAP systems

The term Online Analytical Processing or OLAP was coined by E.F. Codd in 1992 [19]. Codd defined twelve features that should be present in any OLAP application, of which the following five are probably the most important:

1. **Multidimensional conceptual view.** The data model representation in OLAP should be multidimensional in nature, thus allowing analysts to execute intuitive operations such as "slice and dice", rotate and axis pivot.

2. **Transparency.** The user should not have to understand the physical resources that are used for storage and data processing and how the data is actually organized.

3. **Accessibility.** OLAP should present the user with a single logical schema of the data.

4. **Flexible reporting.** Reporting facilities should present information in almost any way the user wants to view it.

5. **Unlimited dimensions and aggregation levels.** A serious tool should support more than just a few concurrent dimensions (Codd actually indicated that 15–20 would be ideal).

2.3.1 The Multi Dimensional Model

To facilitate complex analysis and visualization, OLAP data is organized according to the multidimensional data model. Specifically, we represent data within a d-dimensional space such as the one depicted in Figure 2.3. In this context, the multidimensional model can be described as a data abstraction that allows one to view aggregated data from a number of perspectives (dimensions). In fact, for a d-dimensional space, A_1, A_2, ... , A_d there are exactly 2^d distinct dimension combinations (cuboids) that represent the underlying Star Schema. We often refer to this collection of views as the Power Set [58], with each cuboid representing a unique view of the data at a given level of granularity.

In practice, not all cuboids need actually be present in order to carry out OLAP analysis. This is the case since any cuboid can be computed by aggregating the cell

values in the Fact table (or *base* cuboid) across one or more dimensions. Nevertheless, for cuboids that have been pre-computed and stored to disk, real time query performance can be significantly enhanced. If the data cube does in fact contain all 2^d possible views, it is described as a "full cube", while the term "partial cube" is used if only a subset of views has been constructed. In reference to Figure 2.3, we can see that, in addition to the base cuboid, we have materialized five three-dimensional cuboids (i.e., "By Time, Customer, and Product"), five two-dimensional cuboids (i.e., "By Customer and Product"), four one-dimensional cuboids (i.e., "By Product", "By Customer", "By Time", and "By Channel", respectively), and one zero-dimensional cuboid (i.e., the "Grand Total").

Ultimately, the data cube provides a more intuitive representation of the data warehouse Star Schema. It requires of the user no assumptions about how the data is physically stored. Advanced OLAP servers may in fact take the data from the tables of the original Star Schema and further process it. The data may be stored in a series of new tables or even a multidimensional array. We refer to the first type of system as Relational OLAP (ROLAP), while the second is known as Multi-dimensional OLAP (MOLAP). With MOLAP, an implicit indexing along the axes of the multi-dimensional array is provided, but performance sometimes deteriorates as the space and the associated cube array becomes more sparse (high dimensionality/high cardinality). Conversely, in ROLAP systems, cuboids are stored as distinct tables and tend to scale well since only those records that actually exist are materialized and stored. However, an explicit multidimensional indexing is required in order to be used effectively.

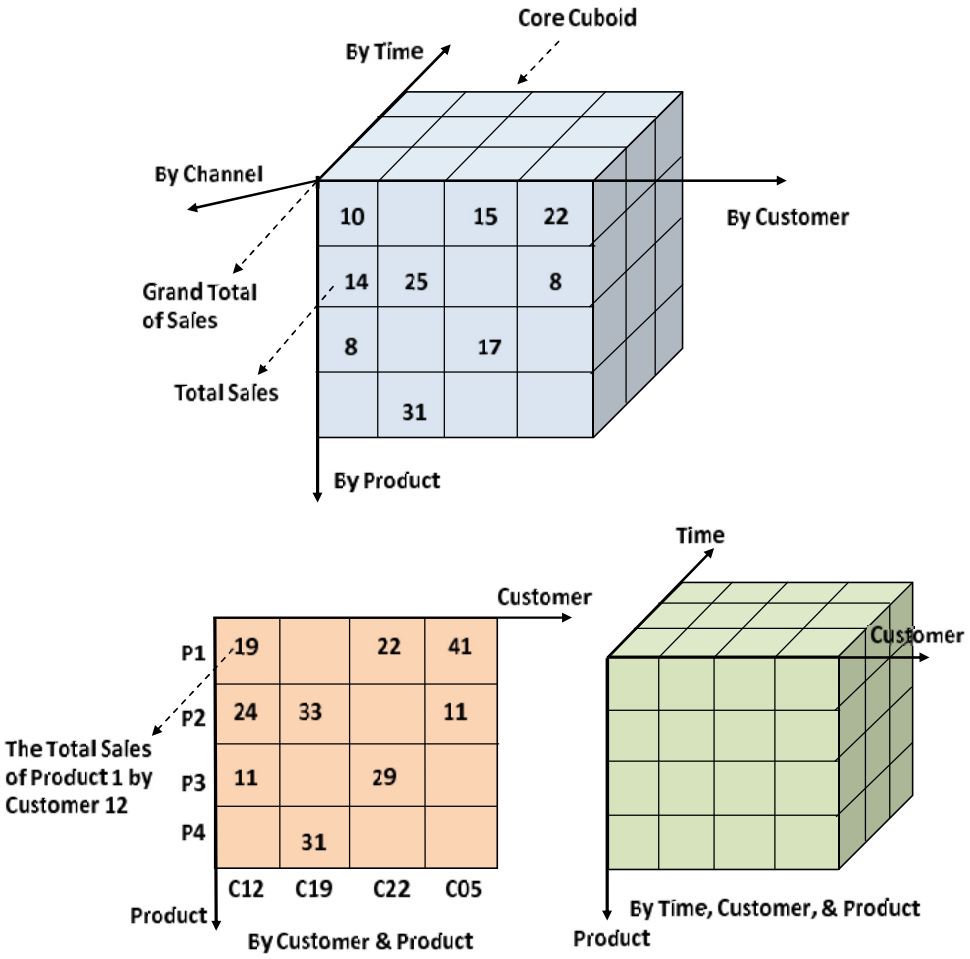
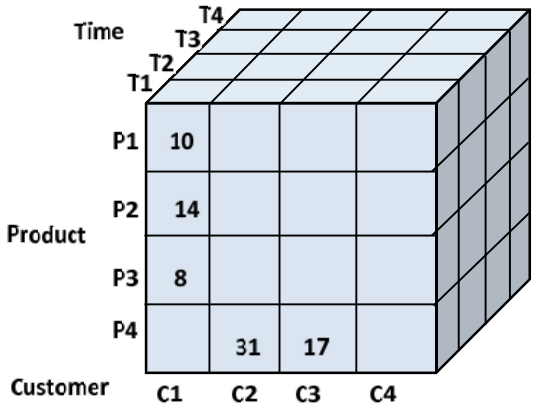


Figure 2.3: A three dimensional OLAP space showing all 2^d cuboids



(a)

Product	Customer	Time	Sales
P1	C1	T1	10
P2	C1	T1	14
P3	C1	T1	8
P4	C2	T1	31
P4	C3	T1	17

(b)

Figure 2.4: Two views of the base cuboid of a three dimensional cube (a) MOLAP model and (b) ROLAP model

2.3.2 Dimension Hierarchies

A dimension hierarchy describes a hierarchical relationship among two or more dimension members. For example, Figure 2.5 presents an example of a dimension hierarchy for a customer dimension and shows how data is ordered by CustomerID, Store, and Retailer. A specific customer belongs to a particular store which, in turn, belongs to a particular retailer. A dimension hierarchy can use ordered levels to organize and aggregate data, with each level containing aggregate values for the levels below it. It is possible to group all values of the dimension into a single "total" value (i.e., the top or coarsest level), while the lowest contains the most detailed values for that dimension. For instance, let us consider the dimension Customer of Figure 2.5. CustomerID is the most detailed or granular level, with Retailer being the topmost member for this dimension. Note that the number 87 that appears beside the Montreal Retailer node represents the aggregated total sales for its lower levels (i.e., customers 12 and 19).

Physically, we can represent the previous hierarchy by additional columns in the

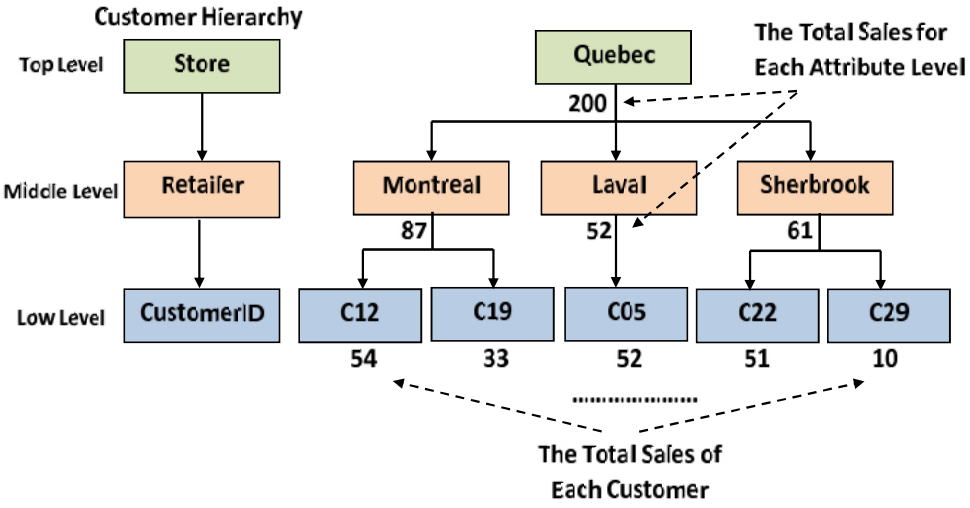


Figure 2.5: The Customer dimension hierarchy example

Seq	CustomerID	Retailer	Store
1	C21	Montreal	Quebec
2	C19	Montreal	Quebec
3	C05	Laval	Quebec
4	C22	Sherbrook	Quebec
5	C29	Sherbrook	Quebec

Figure 2.6: A hierarchy dimension table

associated dimension table, as in Figure 2.6. The concept of the dimension hierarchy will be further discussed in Chapter 3.

2.4 The Sidera server prototype

The work described in this thesis is part of a larger project whose focus is to design, implement and optimize an OLAP-specific DBMS server (Sidera)[39]. Sidera offers us the possibility of defining security measures over the multidimensional data model. It allows us to implement our theoretical ideas within a true OLAP server engine. This gives us the opportunity to explore performance and correctness issues that would not be possible without such direct access to a DBMS. In this section, we present a

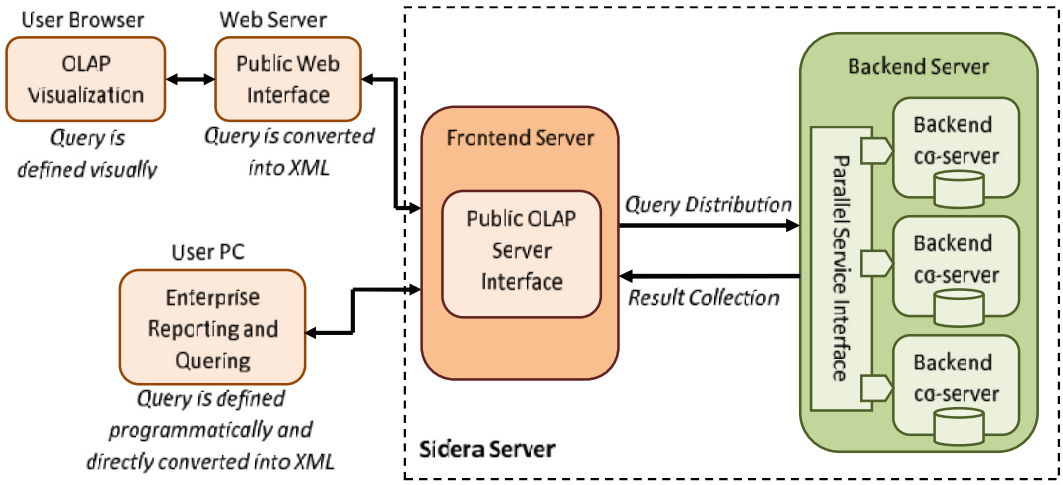


Figure 2.7: The core architecture of the parallel Sidera OLAP server

simple architectural overview of the Sidera system, with a brief look at the Sidera frontend and backend servers.

2.4.1 DBMS Architecture

Eavis et al., in their paper entitled "Sidera: a cluster-base server for Online Analytical Processing", presented a comprehensive architectural model for a fully parallelized OLAP server [39]. The system consists of a pair of server executables (i.e. frontend and backend) that boot simultaneously and subsequently share a collection of communication channels. Figure 2.7 illustrates the fundamental design of the prototype.

1. The Sidera Frontend

The server interface (i.e., the frontend node) serves as an access point for end user queries. Figure 2.8 is an illustration of the Sidera frontend, a multi-threaded head node that handles logins, authentication, and transfer of queries to the backend nodes. Its core function is to receive user requests and to pass them along to the backend nodes for resolution. It does not participate in query resolution directly, other than

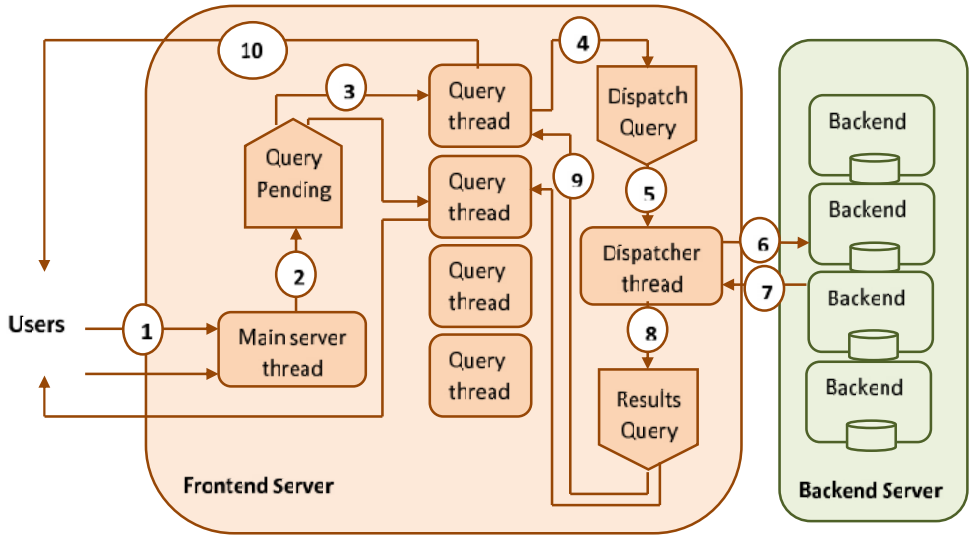


Figure 2.8: The Sidera Frontend

to collect the final result from the backend instances and return it to the user. The numbered sequence in the Figure indicates the processing cycle for a typical query. We note that the Authentication and Authorization components discussed in this thesis will be integrated into the frontend.

2. The Sidera Backend

The backend server consists of multiple nodes, which operate within the fully coordinated architecture. Each node houses a Parallel Service Interface (PSI) component that allows it to hook into the global communication fabric and thereby participate in parallel sorting, merging, and aggregation operations. The backend network is responsible for storage, indexing, query planning, I/O, buffering, and metadata management. As such, it is responsible carrying out virtually all of the query resolution tasks. Figure 2.9 depicts the processing loop on the backend server instances.

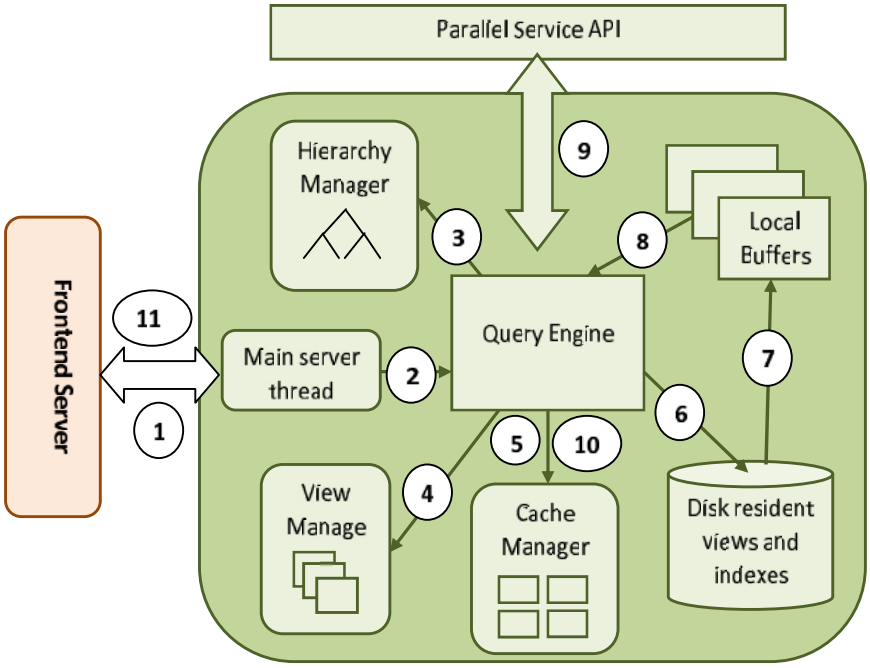


Figure 2.9: The Sidera Backend

2.4.2 Query Interface

Sidera uses a native OLAP query language API called NOX to provide responsive and intuitive query facilities [40]. NOX queries are object-oriented and support inheritance, re-factoring and compile-time checking. Underlying this functionality is a domain specific algebra and language grammar that are used to transparently convert queries written in the native development language into algebraic operations understood by the server. This algebra forms the basis of the intermediate query format that will be used by the security framework discussed throughout the rest of the thesis. Below, we provide a listing of the core operations defined by the NOX algebra [110].

- **SELECTION Operator ($\sigma_p$cube)**: provides the identification of one or more cells from within the full d-dimensional search space. This is one of the two

core OLAP operations and is commonly referred to as "slicing" and "dicing". A logic predicate p defines cells of interest within the d-dimensional space. The predicate has a syntactical form that allows mathematical expressions to be compared to each other and different conditional expressions can be combined with logical operators such as AND and OR.

- **PROJECTION Operator ($\pi_{measure_1,...,measure_n}$cube)**: provides the identification of presentation attributes, including both measure attributes and feature attributes. This is the second core OLAP operation and it is mainly concerned with preparing results for presentation in formats such as diagrams, objects or simply text. In other words, it selects of a subset of display attributes (measures or features).

- **Set Operations on Data Cubes**: OLAP set operations includes UNION ($cube_1 \bigcup cube_2$), which performs the union of two cubes over an n-dimensional space sharing common dimensional axes, INTERSECTION ($cube_1 \bigcap cube_2$) that performs the intersection of two cubes over an n-dimensional space sharing common dimensional axes, and DIFFERENCE ($cube_1$ - $cube_2$) that performs the difference of two cubes over an n-dimensional space sharing common dimensional axes.

- **CHANGE LEVEL Operator ($\gamma^{level_i \to level_j}_{f(measure_1),...,f(measure_n)}$)**: modifies of the granularity of aggregation for the current result set. This process is typically referred to as "drill down" and "roll up". The roll-up operation groups cells in a Cube based on a given aggregation hierarchy, while the drill-down goes down through

an aggregation hierarchy in order to show more detailed data. The gamma representation of the change level operation implies that as the level of the data is changing, the measure values are also changing according to functions that aggregate or decompose data values (along the levels of the hierarchy).

- **PIVOT(ϕ_{base})**: performs the rotation of the cube axes to provide an alternate perspective of the cube. In this operation, no recalculation of cell values is required.

It is important to mention that after defining the queries programmatically, they are directly converted into XML format (embedded within the re-written source code). As a concrete example, Listing 2.1 illustrates a simple SQL query that summarizes the total sales of Quebec's stores in 2011. The corresponding XML encoding of the query is provided in Appendix C. With a little effort one can see how the total sales in 2011 for Quebec stores is captured by the sequence of nested XML statements.

```
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 2.1: A simple SQL OLAP query

To validate the received XML query, the system relies on a Document Type Declaration (DTD) grammar that is used to describe the structure of the XML query. The grammar itself is given in Appendix C. (We note that the development of the grammar itself was not part of the current thesis). Ultimately, its purpose is to represent the functionality of the analytics queries one would expect to see in a Business

Intelligence context.

2.5 Conceptual to Logical Data Model Mapping

To build a security module that can apply access restrictions to an underlying relational database, the module must know how hierarchies are structured or defined in the database in order to map the high level security constraints to the relational schema. There are in fact a number of ways that such a framework can be implemented, including both text-based and graphical methods. For instance, in the open source Mondrian OLAP query processing system, there is a full XML grammar that is used to define this mapping [81]. In short, an XML schema is utilized to define a multi-dimensional database. The schema contains a conceptual model, consisting of cubes, hierarchies, and members, and defines a mapping of this model onto the logical (i.e., relational) model. The logical model (typically a star schema) ultimately represents the low level tabular data physically stored by the DBMS backend. In our work, we assume the existence of such a mapping component.

A sample Mondrian schema for the star schema depicted in Figure 2.2 is given in Listing 2.2, where the schema contains a single cube, called "Sales". The Sales cube has four dimensions, "Product", "Customer", "Time", "Channel" and four measures, "Units Sold", "Dollar Sales", "Product Cost", and "Shipping Cost". Each dimension has a hierarchy. For instance, the Customer has three levels — Retailer, Store, and CustID — each associated with a specific column in the Customer table. Using this schema, it is therefore possible to map conceptual elements such as hierarchies to the tables and columns storing the relevant data values. In turn, query (re)writing modules can use this information to dynamically restructure user queries in order to

remain consistent with administrator-defined data access policies.

```
<Schema>
 <Cube name=''Sales">
  <Table name=''Sales"/>
   <Dimension name=''Product" foreignKey=''Code">
    <Hierarchy hasAll=''false" primaryKey=''Code">
     <Table name=''Product"/>
      <Level name=''division" column=''Division"
     uniqueMembers=''true"/>
     <Level name=''family" column=''Family"
         uniqueMembers=''false"/>
      <Level name=''class" column=''Class" uniqueMembers=''false"/>
     <Level name=''code" column=''Code" type=''Numeric"
         uniqueMembers=''true"/>
    </Hierarchy>
   </Dimension>

   <Dimension name=''Customer" foreignKey=''CustID">
    <Hierarchy hasAll=''true" primaryKey=''CustID">
     <Table name=''Customer"/>
      <Level name=''retailer" column=''Retailer"
     uniqueMembers=''true"/>
     <Level name=''store" column=''Store" uniqueMembers=''false"/>
      <Level name=''custID" column=''CustimeID" type=''Numeric"
     uniqueMembers=''true"/>
   </Hierarchy>
  </Dimension>

  <Dimension name=''Time" foreignKey=''TimeID">
   <Hierarchy hasAll=''true" primaryKey=''TimeID">
    <Table name=''Time"/>
     <Level name=''year" column=''Year" type=''Numeric"
         uniqueMembers=''true"/>
     <Level name=''quarter" column=''Quarter"
         uniqueMembers=''false"/>
      <Level name=''timeID" column=''TimeID" type=''Numeric"
     uniqueMembers=''true"/>
   </Hierarchy>
  </Dimension>

  <Dimension name=''Channel" foreignKey=''ChanID">
   <Hierarchy hasAll=''true" primaryKey=''ChanID">
```

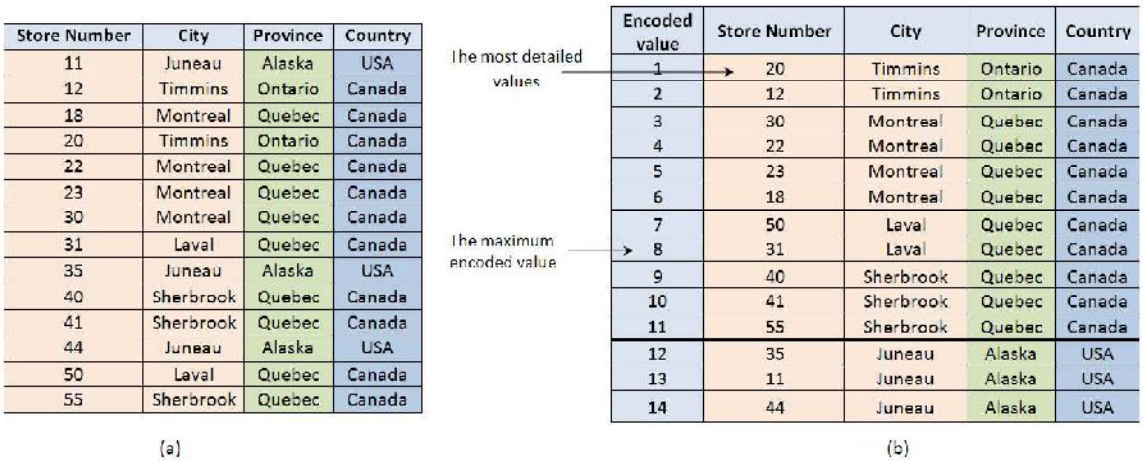
```
    <Table name="Channel"/>
      <Level name="channel" column="Channel" type="Numeric"
          uniqueMembers="true"/>
      <Level name="chanID" column="ChanID" type="Numeric"
          uniqueMembers="true"/>
    </Hierarchy>
  </Dimension>

  <Measure name="Units Sold" column="Units Sold"
      aggregator="sum" formatString="#,###"/>
  <Measure name="Dollar Sales" column="Dollar Sales"
      aggregator="sum" formatString="#,###.##"/>
  <Measure name="Product Cost" column="Product Cost"
      aggregator="sum" formatString="#,###.00"/>
  <Measure name="Shipping Cost" column="Shipping Cost"
      aggregator="sum" formatString="#,###.00"/>
 </Cube>
</Schema>
```

Listing 2.2: An XML definition schema

2.6 The mapGraph Hierarchy Manager

Internally, Sidera houses meta data for dimension hierarchies using the mapGraph hierarchy manager. The mapGraph [41] is a suite of algorithms and data structures for the manipulation of attribute hierarchies in "real time". MapGraph builds upon the notion of hierarchy *linearity* [83]. A hierarchy is consider linear if there is a contiguous range of values R_j of dimension attribute A_j that may be aggregated into a contiguous range R_i. To establish the linearity of each dimension hierarchy, a sorting technique is used with data subsequently being stored at the finest level of granularity. For instance, Figure 2.10(a) shows the Store dimension table with the hierarchy Store_Number, City, Province, and Country. The finest level of the Store dimension is Store_Number; therefore, transactional data would be stored at this level. Figure 2.10(b) illustrates the sorted table. Finally, a compact, in-memory



Store Number	City	Province	Country
11	Juneau	Alaska	USA
12	Timmins	Ontario	Canada
18	Montreal	Quebec	Canada
20	Timmins	Ontario	Canada
22	Montreal	Quebec	Canada
23	Montreal	Quebec	Canada
30	Montreal	Quebec	Canada
31	Laval	Quebec	Canada
35	Juneau	Alaska	USA
40	Sherbrook	Quebec	Canada
41	Sherbrook	Quebec	Canada
44	Juneau	Alaska	USA
50	Laval	Quebec	Canada
55	Sherbrook	Quebec	Canada

(a)

Encoded value	Store Number	City	Province	Country
1	20	Timmins	Ontario	Canada
2	12	Timmins	Ontario	Canada
3	30	Montreal	Quebec	Canada
4	22	Montreal	Quebec	Canada
5	23	Montreal	Quebec	Canada
6	18	Montreal	Quebec	Canada
7	50	Laval	Quebec	Canada
8	31	Laval	Quebec	Canada
9	40	Sherbrook	Quebec	Canada
10	41	Sherbrook	Quebec	Canada
11	55	Sherbrook	Quebec	Canada
12	35	Juneau	Alaska	USA
13	11	Juneau	Alaska	USA
14	44	Juneau	Alaska	USA

(b)

Figure 2.10: (a)A Store dimension table (b)The corresponding sorted table

look-up data structure (i.e., the hMap structure) is then used to support efficient real time transformations between arbitrary levels of the dimension hierarchy. Figure 2.11 shows the corresponding hMap.

Each record in the hMap consists of two values: a native attribute representation (i.e., values of attributes Types of Store dimension) and an integer value that represents the corresponding maximum encoded value in the primary attribute. For example, the city of Timmins has two stores with encoded numbers 1 and 2 while Montreal has four stores, encoded with the numbers 3 through 6. Using this structure, one can perform a mapping from the most detailed encoded level value to the corresponding sub-attribute value (i.e., attribute level values), and vice versa. For instance, a Store encoded as 13 is located in Juneau and, as a consequence, it is located in the state of Alaska in the USA country (i.e., Alaska has a maximum encoded 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

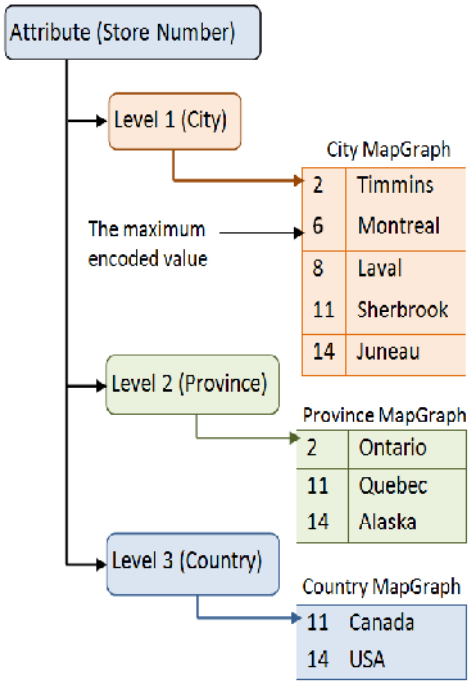


Figure 2.11: The hMap of the Store dimension.

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.

2.7 Column-oriented Database Systems

In recent years we have seen the introduction of a number of column-oriented database management systems, including MonetDB [94] and C-Store [108]. These systems have been shown to perform more than an order of magnitude faster than traditional row-oriented database systems (row-stores) for large, read-intensive data repositories

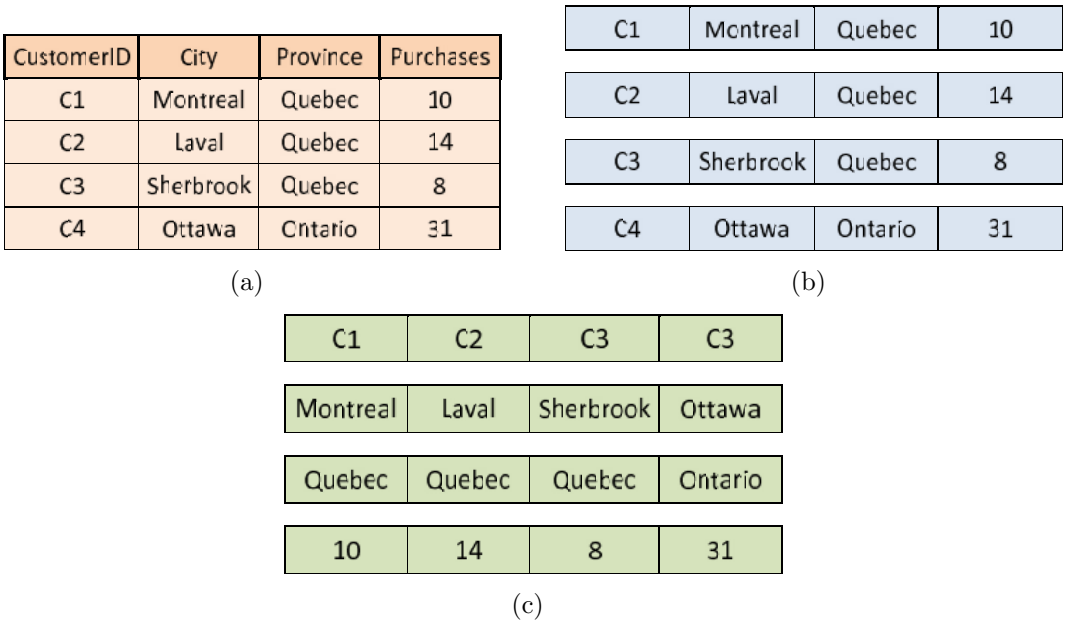


Figure 2.12: (a)Simple two dimensions table (b)Row-oriented database serialization (c)Column-oriented database serialization

such as those found in data warehouses, decision support, and business intelligence applications that support analytical workloads. The reason behind this performance advantage is straightforward: column-stores are much more I/O efficient for read-only queries since they only have to read from disk those attributes actually referenced (directly or indirectly) by a query [1]. Figure 2.12 shows a simple table, and illustrates how a row-oriented and column oriented database would serialize data during a query IO operation. Specifically, Figure 2.12(a) presents a two-dimensional table, upon which a typical query might be executed. In turn, Figure 2.12(b) shows how the row-oriented database would serialize values, while Figure 2.12(c) illustrates how a column-oriented database would serialize the same values.

Loosely speaking, the DBMS must coax the two-dimensional table into a one-dimensional series of bytes in order for the operating system to write to RAM, the

hard drive, or both. In our example, the table includes an employee identifier (EmpId), name fields (Lastname and Firstname) and a salary (Salary). The row-oriented database serializes all of the values in a row together, then the values in the next row, and so on. Conversely, the column-oriented database serializes all of the values of a column together, then the values of the next column, and so on. Consequently, the serialization of the column database is more efficient when an aggregate needs to be computed over many rows and when only a subset of the available columns is required. It can also be more efficient for write operations if, for example, a sequence of column values (perhaps all of them) are updated at once. These scenarios may seem unlikely in the general case but one must keep in mind that warehouses are often loaded or updated via large bulk operations.

We note that, while the original Sidera design includes its own storage engine (R-trees, bitmaps, compression algorithms, etc), it has been extended in the current project so as to include the column-store MonetDB as an alternative storage backend. MonetDB was one of the earliest column-store solutions and includes the following primary features:

- A column-store database kernel. MonetDB is built on the canonical representation of database relations as columns. They can be large entities, up to hundreds of megabytes in size, and are swapped into memory by the operating system and compressed on disk as required.

- Multi-core support. MonetDB is designed for multi-core parallel execution on desktops to reduce response time for complex query processing.

- A versatile algebraic database kernel. MonetDB is designed to accommodate

different query languages through its proprietary algebraic-language, called the MonetDB Assembly Language (MAL).

- High-performance features. MonetDB excels in applications where the database can be largely held in main-memory or where a few columns of a broad relational table are sufficient to handle individual requests.

2.8 Conclusion

In this chapter, we have examined some of the core concepts of Online Analytical Processing and the Multi-Dimensional data model and explained how OLAP provides meaningful analysis by summarizing the data from this model. We also discussed the data warehouse architecture as a three tiered model, including a review of the canonical Star Schema. We then presented the architectural model for the Sidera platform, a robust parallel OLAP server designed for cluster applications. The server consists of a publicly accessible frontend and a collection of identical backend servers. The mapping from conceptual to logical data model was then discussed, along with an example to illustrate the basic process. Finally, we looked at the mapGraph hierarchy manager and the column-store database architecture that have been integrated into Sidera during this research project.

Chapter 3

An OLAP-aware framework for query authentication and authorization

3.1 Introduction

Many organizations collect personal data during their social, medical or financial activities in order to analyze and extract useful knowledge from it. Individuals have strong concerns about this data. In fact nine out of ten respondents to a survey by the Economist Intelligence Unit said they were worried that their financial data would be compromised and then used to steal money from them, while eight out of ten were worried their personal data would be used to target marketing campaigns at them [42]. In order to reduce privacy concerns, various pieces of legislations have been introduced, including the health Insurance Portability and Accountability Act (HIPAA) enacted by the US. Congress and the GLBA (Gramm-Leach-Bliley Act, also known as the Financial Modernization Act). As such, a failure to protect individual's privacy cannot only cause immense damage to an organization, but also lead to lawsuits and regulatory fines resulting from violation of the law by which the data

was collected.

Adopting these protocols helps organizations convince individuals that their personal data is protected from intentionally being misused for profit. However, privacy breaches may occur in various ways *after* personal data have been collected. The data may be stolen by attackers that infiltrate the system through the exploitation of existing vulnerabilities. Such outsider attacks can be addressed by defensive mechanisms such as firewalls, encryption and the like. A more challenging threat comes from insiders. The insider knows what the most valuable data assets are, where to look, and how to access them. For example, a financial services company may want to analysis its customer's records, which usually contain private personal data. Without sufficient security mechanisms safeguarding the data, the organization's analysts may obtain and later misuse the financial data, which leads to privacy breaches of individuals and consequently causes damages to the organization's interests.

To address these privacy concerns, organizations rely upon two control abstractions: policies and mechanisms. Security policies are high-level requirements that determine which user, under what circumstances, may access specific data. This can be accomplished by defining a series of conditions (restrictions and/or exceptions), usually by the system administrator, for controlling and monitoring user access [50]. A security policy is determined primarily by the sensitivity of the data. If the data is sensitive, a security policy should be developed to maintain tight control over accessing that data. For instance, within a hospital the pathological history of patients may be considered as sensitive data. The policy could establish that only doctors and nurse practitioners may access the pathological history of patients; any other user should be restricted from accessing this data.

Security mechanisms are responsible for imposing security policies. Briefly, the security mechanism translates a user's access request, often in terms of a structure that a system provides, and evaluates it against security policies. If the mechanism reveals any violation of policy conditions, the access request is considered insecure and, as a consequence, it will be denied. In fact, this is done in two phases: Authentication and Authorization. Authentication provides a way of identifying a user, typically by providing a valid user name and valid password (user's credentials) before access is granted. The user's credentials are verified against a list of valid accounts provided by the database administrator. If the credentials match, the user is granted access. If they don't, authentication fails and the access is denied.

Following authentication, the authorization process determines what the user can actually access. Simply put, authorization is the process of enforcing policies: authorizing a user to access only a subset of the database based on his/her permissions. Together, authentication and authorization work in a synergistic relationship to provide an access control mechanism. The main purpose of the access control mechanism is to prevent unauthorized access that could lead to a breach of security.

Many access control mechanisms have been proposed for traditional data managements systems (i.e., relational database systems) [62, 123, 96]. Some of them provide only a way to restrict data access at coarse granularity, where "granularity" refers to the size of the individual data items that can be authorized. For instance, one might allow access to a table at the granularity of rows or columns. Others are more restrictive and can support access control to an individual cell. The authors in [96] for example, proposed a model for fine-grained authorization based on adding predicates to authorization grants to support authorization at the level of row, column or cell.

However, directly applying the access control techniques of relational databases to the OLAP domain meets with two difficulties. First, the data model of OLAP is different from the standard relational model used by relational databases. The difference in the data models makes it difficult to express and enforce the security requirements of OLAP. Second, access control in relational databases provides little protection against malicious inferences of sensitive information to which OLAP is more vulnerable (this problem will be discussed in Chapter 4).

We note that the distinct data model changes the logic or focus of the access control mechanism. Specifically, OLAP is associated with a more abstract conceptual model and includes dimensions of the multidimensional cube, hierarchies within each dimension, and the aggregated cells. One can, of course, work at the level of tables, columns, etc., but this is quite difficult if abstract ideas like hierarchies or dimensions are spread throughout or across tables. Moreover, this sort of "traditional" approach is very error prone since one must be extremely careful to manually restrict or include all possible levels of aggregation.

For example, assume the simple star schema for a commercial enterprise DW shown in Figure 3.1. The schema consists of one fact table "Sales" and three dimension tables: Store, Time and Product. The Sales table contains sales records that become the objects of analysis, while the dimension tables contain descriptive attributes for those records. Each dimension is associated with a distinct aggregation hierarchy. Stores, for instance, are organized in Country, Province, City and Store_number. Now, suppose we have the following policy: the sales totals of the individual provinces should not be accessed by the marketing employees.

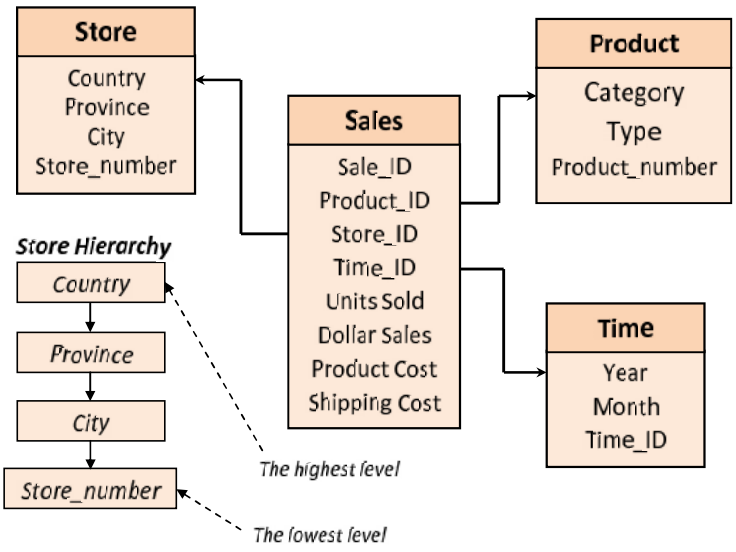


Figure 3.1: A simple star schema

According to the policy, the marketing employees should be restricted from accessing the provincial sales. However, even with this restriction, the employees can in fact compute (or indirectly access) it by aggregating the sales of the lower levels of the associated dimension hierarchy (City or Store_number in our example). This is also applicable to specific values where it can be computed from the lower level values.

Ideally, the system administrator should not be responsible for identifying and manually ensuring that all implied levels/values are included in the policy. Instead, we present in this chapter an OLAP-aware framework for query authentication and authorization that is based on a *query rewriting* technique. The framework 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. In case of query modification, the user is informed by a warning message that telling him/her that the query is modified during security concerns. However, in this case, the user should not know what he/she is restricted from, since this may be used as external information in some case to infer the protected data [64].

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 primary objective is to allow DB people to work at a higher level of abstraction — one that matches their intuitive understanding of an OLAP database. Ideally, we would produce a system that transparently propagates aggregation logic across hierarchy levels. In a sense, this is conceptually similar to the purpose of SQL itself — we tell the DBMS what we want to secure rather than telling it exactly how to provide the protection.

In addition, we present the data structures and algorithms utilized by the mechanisms 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 include an experimental section that highlights the processing overhead relative to the execution costs of the underlying query.

The remainder of this chapter is organized as follows. In Section 3.2, we present an overview of related work. Basic concepts related to our research are introduced in Section 3.3. Our objectives and methodology are discussed in Section 3.4. The OLAP

query rewriting model and its associated transformation rules, including query representation and hierarchy processing, are then presented in detail in Section 3.5. The implementation and performance issues are presented in Section 3.6. Experimental results are discussed in Section 3.7, with final conclusions offered in Section 3.8.

3.2 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 restrict users and/or processes to performing only those operations (read, write, or execute) on the objects, tables or columns for which they are authorized. For each such operation, the access controls either allow or disallow that operation to be performed.

During the early stages of database security research, the primary focus was on *Discretionary Access Controls* (DACs) [51]. 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) [11]. 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 (i.e., access to only their own data address), nor fine-grained least privilege.

An alternative approach was introduced in the 1990's [101]. 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 modeling.

A number of security models that restrict data warehouse access have also been proposed in the literature [46, 61, 98, 13]. Some of them 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 [46]. In fact, a number of researchers have looked at similar techniques for setting access constraints at an early stage in the OLAP design process [12, 60]. Others have developed security requirements for the entire Data Warehouse life cycle [65, 82, 127]. 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 authentication and authorization algorithms per se, but rather on design methodologies that would most effectively use existing technologies, such as, Model Driven Architecture (MDA) and the standard Software Process Engineering Metamodel Specification (SPEM) from the Object Management Group (OMG).

Other researchers have attempted to augment the core Database Management System (DBMS) with authorizations views [63, 98, 99, 130]. 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 [31]. Beyond that, however, rewriting has also been utilized to provide fine grained access control in Relational databases [69, 96]. To answer a query Q in [69], masked versions of related tables are generated by replacing all the cells that are not allowed to be seen with *NULL*. After that, Q is evaluated as a normal query on the masked versions of the tables. This approach does not leak information not allowed to be seen, but it returns incorrect results when a query contains any negation, as expressed using the keywords MINUS, NOT EXISTS or NOT IN [123].

In the Truman model [96], on the other hand, the database administrator defines a *parametrized* authorization view for each relation in the database. Note that parametrized 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 parametrized 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.

The query rewriting technology has also been adopted in some commercial databases for fine-grained access control. For example, Oracle Virtual Private Database (VPD) [118] limits access to data by appending a predicate clause to the user's query. 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. However, it is difficult to write predicates involving the case of self-joins. For instance, Oracle does not allow defining policies on a table that would access the table itself as this would create an infinite loop [90]. As demonstrated in [102], in complex cases the user may be authorized to view certain data, but a correct query is transformed into an invalid one by attaching the predicate. Moreover, writing policy functions corresponding to business policies is a lot of work since we have to write a function for each protected data item.

Ultimately, we note 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 would make such constraints exceedingly tedious to define.

3.3 Basic Concepts

Before discussing the details of our authorization framework, we first introduce a number of basic concepts that are relevant to our research.

3.3.1 Data Cube

As introduced in Chapter 2, a data cube is composed of a series of d dimensions — sometimes called *feature* attributes — and one or more *measures* [58]. 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 3.2

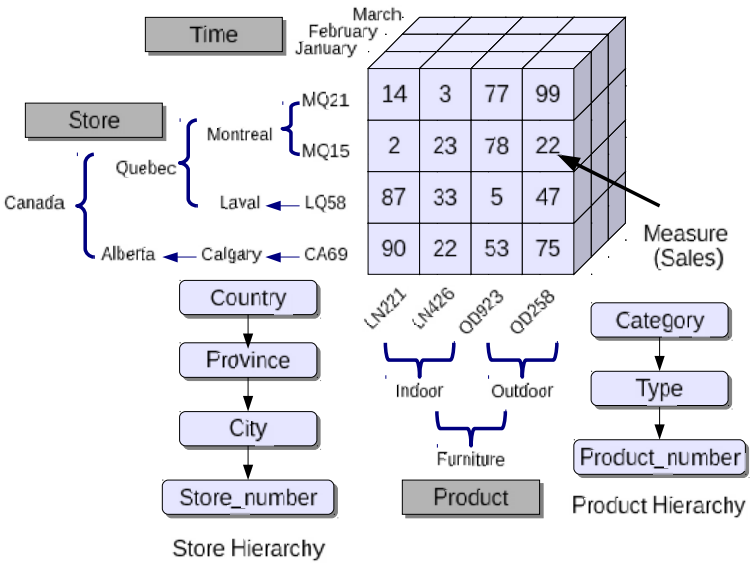


Figure 3.2: A Simple three dimensional data cube

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).

3.3.2 Dimension Hierarchy

In the discussion that follows, we shall closely follow the *dependency lattice* defined in [53]. The attributes of each dimension are partially ordered by the dependency relation $\preceq$ into a dependency lattice. For instance, referring to Figure 3.2, we see that Stores are organized in `Country` $\rightarrow$ `Province` $\rightarrow$ `City` groupings. These attributes are partially ordered into `Store Number` $\preceq$ `City` $\preceq$ `Province` $\preceq$ `Country` within the Store dimension. The `Store Number` is the lowest or *base* level in the Store dimension. In practice, data is physically stored at the base level so as to support run-time aggregation to coarser hierarchy levels. 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 $(0 \leq h \leq j)$.*

We note that there are in fact many variations on the form of OLAP hierarchies [78] (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 chapter.

3.3.3 Authorization Objects

Authorization objects define those elements that are not allowed to be accessed by users. In relational settings, objects can be tables, records of a table, or fields of a record. An analogous representation can be defined on data cubes along one or more dimensions. Objects can also be associated with any vertical portion of the data cube. Specifically, one can restrict one or more dimensions or dependency lattices, thus giving additional opportunities for finer authorization. Further details are given in Section 3.5.3.

3.3.4 Access Control Policies

Access Control Policies restrict — per user— any portion of the data cube (defined as Authorization Objects) by specifying for each user a set of restrictions on that object. Policies can be defined not only on dimensions but also on hierarchies, or on any level within each hierarchy. This can divide a data cube vertically or horizontally depending on how the policy is actually defined. The partitioning is considered to

be vertical if the policy is created on a dimension, and horizontal if it is created on a specific hierarchy level value. Of course, in some cases, the authorization policy can be more complex and enforced at a fine-grained level (cell value). For ease of exposition, and without loss of generality, in the rest of this chapter we restrict authorization to only one cube.

It is also possible that an *exception* to an authorization policy is required. One can set a policy to restrict access to a specific data object, and also define exceptions to specify a subset of the object that may be accessed. Suppose we have the following policy for instance: all users *except* local administrators should not know the sales of stores. We note that the exception should be a subset of the restriction. In our example, only administrators are permitted to know the sales of stores.

3.4 Objectives and Methodology

3.4.1 Objectives

In this chapter, we describe an OLAP-aware framework for query authentication and authorization. The framework is based on a query rewriting technique that enforces distinct security policies that, in turn, may be associated with user populations of arbitrary size. In brief, our framework relies on a set of structures and rules to dynamically transform user queries that contain unauthorized data access into equivalent queries that can be executed against the original data.

The primary objectives of our approach are:

- To enforce privacy policies without any modification to the existing data cubes and/or without creating any additional materialized views. Doing so would allow for relatively simple integration into existing systems.

- To define an OLAP-centric authentication and authorization database for storing, managing, and mapping privacy policies and users permissions to the conceptual properties and elements of the data cube model itself.

- To apply query restrictions not only on direct access to data cube elements, but also on certain forms of indirect access, while ensuring its availability to legitimate users.

- In cases where the results of rewritten queries may differ from those of original query, the user should be notified.

3.4.2 The Methodology

The methodology employed in pursuing these objectives requires a clear understanding of the tradeoffs inherent in this domain. Ultimately, the key technical challenge is to balance utility and privacy. Perfect privacy (but no utility) can be achieved by refusing to answer any query that breach as the policy; perfect utility (but no privacy) can be achieved by answering all queries about the data exactly. In the middle, a sufficient security model should balance the competing goals of privacy and utility. It should return as much information as possible, while satisfying authorization policies.

As such, our general approach builds upon a series of *Authorization Rules* to decide whether user queries should be rejected, executed directly, or transparently and dynamically transformed. In the latter case, we identify a set of minimal changes that would allow queries to proceed against a subset of the requested data. The rules are classified into four classes depending on the restricted data level and the existence of exceptions, as follows: (i) The first class is considered when a user requests access to protected coarse level data with no exception(s); (ii) The second class is considered

for the complementary case (i.e., for protected coarse level data with exception(s)); (iii) The third class is applied when a request access protected fine-grained values (i.e., cell value) with exception(s); (iv) Finally, the last class is applied to protected fine-grained values without exception(s).

To demonstrate that the modified query is secure (i.e., returns only the permitted data), we then have to show that the modified query is equivalent to the initial query, minus the possible restrictions. Formally, for data objects D involved in a query Qc whose the modified version is Qm, $Qm(D) \equiv Qc(D - O)$ should be true for the authorization object O (*see Definition 4*).

Because of the maturity of the database domain in general, it is important to ground our approach with a robust experimental evaluation. To this end, we will provide a working prototype. Figure 3.3 describes the prototype components and the correlations between them. In our prototype, the system administrator is responsible for defining security policies and users accounts (identified by users credentials), which are stored in a centralized authentication and authorization database (i.e., the Authentication database). Once the query is received, the query is parsed and converted into a simplified data structure (i.e., the Query Object).

3.5 Authentication and Authorization Modules

In this section, we will describe our general framework, giving a detailed description of its two primary components (**Authentication** and **Authorization**) and how they are associated with other system components. This work has been undertaken as part of a larger experimental OLAP-specific DBMS server called Sidera. Sidera is a comprehensive prototype of a fully parallelized OLAP server. However, it currently

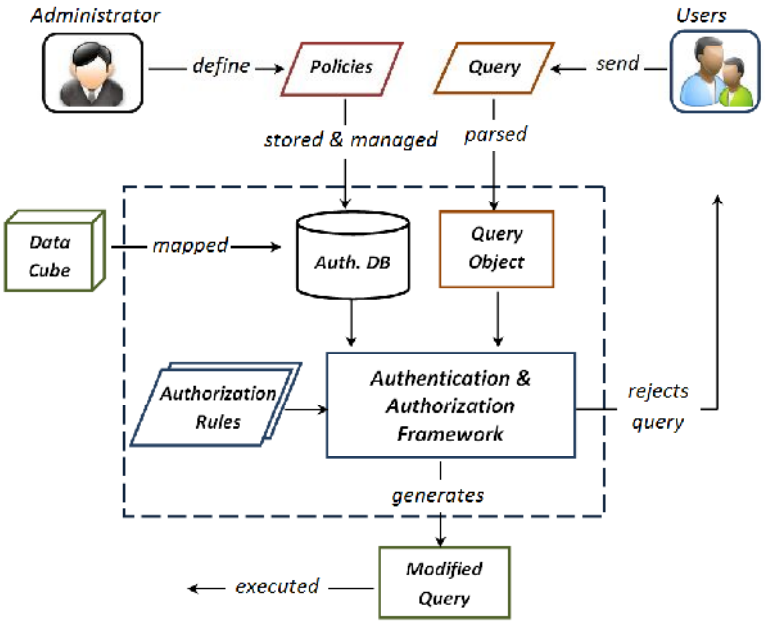


Figure 3.3: The prototype components

does not provide any form of access control. The work described in the remainder of this thesis provides such a framework.

3.5.1 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 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 a user. For example, the `users` table stores basic user credentials (typically a user name and password pair), while the `permissions` table records the fact that a given user(s) may or may not access certain controlled objects. Figure 3.4 illustrates a slightly simplified version of the Authentication DB schema. In the current prototype,

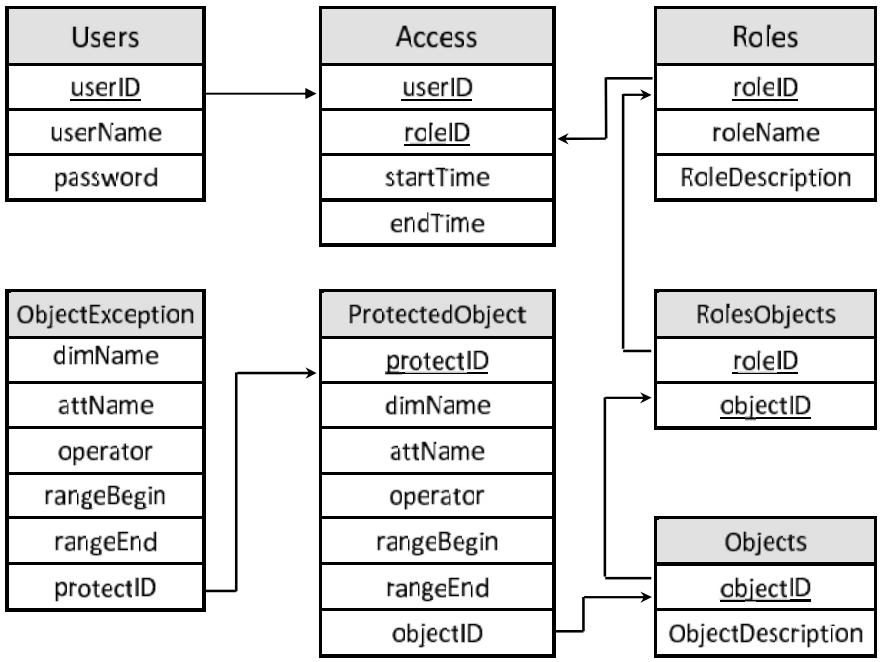


Figure 3.4: The Authentication DB

storage and access to the Authentication DB is provided by the SQLite toolkit [107]. 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.

As mentioned in Chapter 2, the user's query is represented in XML format in our system. 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 — the grammar itself is depicted in Appendix C. 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

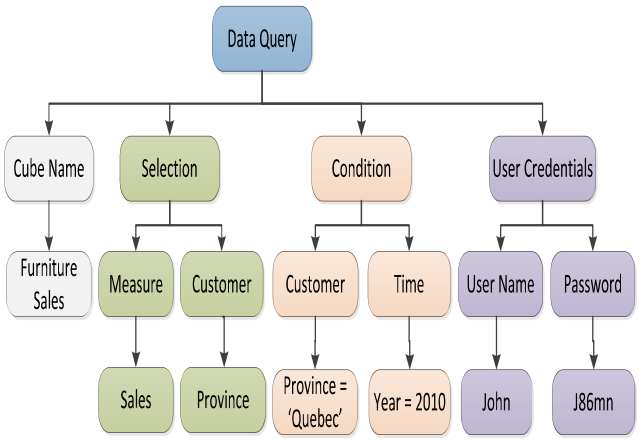


Figure 3.5: A small Parse Tree fragment

parsing error message is returned to the user.

As an example, suppose a user John sends the query given in Appendix C that summarizes the total sales of Quebec's stores in 2011. The corresponding node tree is shown in Figure 3.5. We can easily see that the content of this parse tree is equivalent to the original query. 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

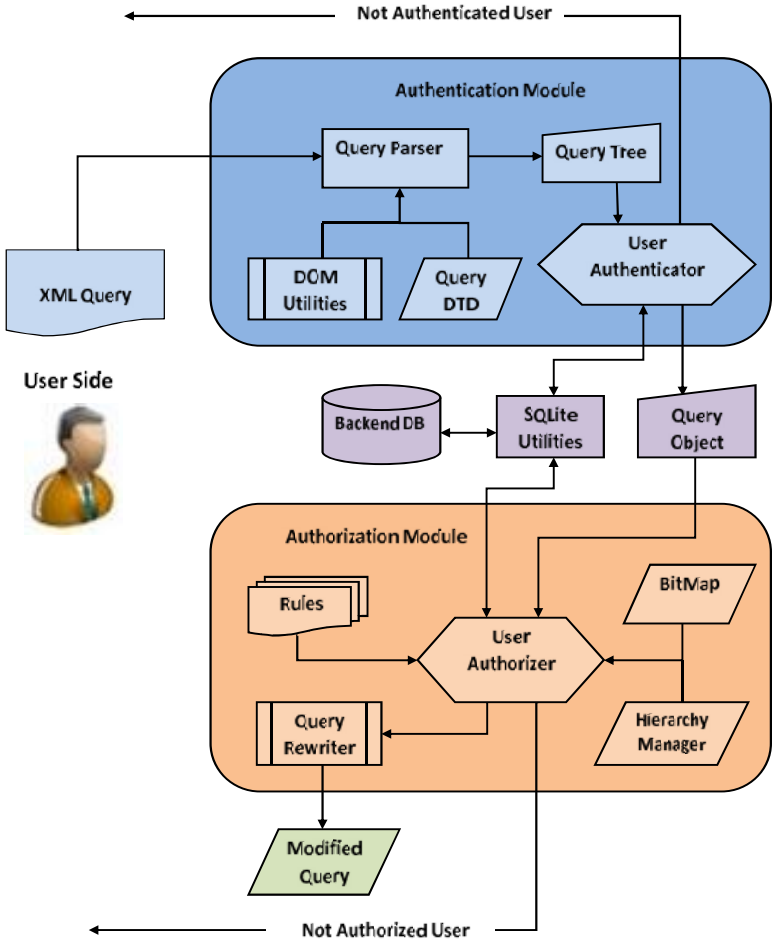


Figure 3.6: Authentication and Authorization Framework

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 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. The upper part of Figure 3.6 depicts the processing logic of the Authentication module.

3.5.2 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 Authentication 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 3.5.4. The lower portion of Figure 3.6 graphically illustrates the Authorization module and indicates its interaction with the Authentication component.

3.5.3 Specifying Authorization Objects

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. In the discussion that follows, we assume the Store-Product-Time data cube of Figure 3.2 where the Store dimension hierarchies include Store_Number $\preceq$ City $\preceq$ Province $\preceq$ Country. 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 data cube. 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). 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

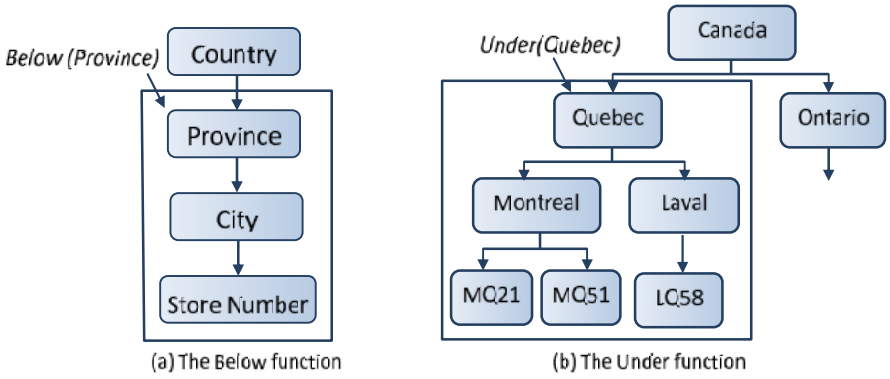


Figure 3.7: The Below and Under functions

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 3.7(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) = $\{L_i \bigcup L_j$: 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

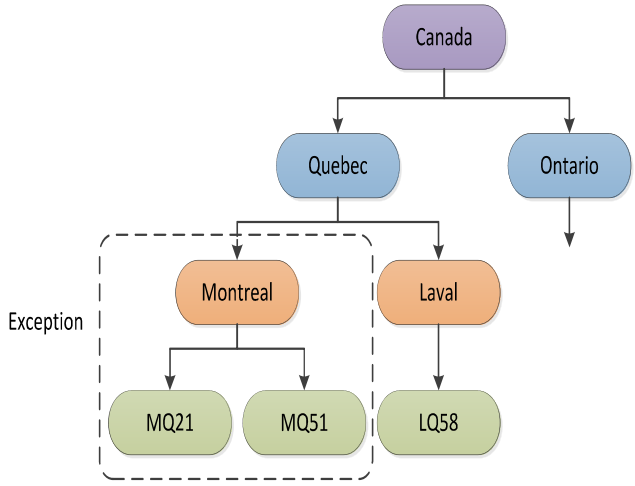


Figure 3.8: An authorization exception

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 3.7 (b) provides an example using *Under(Quebec)*. Here, all the values surrounded by the dashed line should be protected.

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 3.8 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.

```
Selection:
      Product.Name, Store.province, Sum(sales)
Condition:
      Store.Province = 'Quebec' AND
      Time.Year = 2011
From:
      Sales
```

Listing 3.1: A Query in Simple Form

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 = $\{\{v\}$: $\{v\}$ = values of Under(L_i) - Ev$\}$, where L_i is the prohibited level, and Ev are the exception values.*

3.5.4 Authorization Rules

We now turn to the query authorization process itself. As noted, 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, suppose the simple query in Listing 3.1 that 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 3.2. 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 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.

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.

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

Listing 3.2: Rule 1 example

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 3.2, which summarizes the total sales of Canada's stores in 2011 for furniture 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.

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

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

Listing 3.3: Rule 4 example

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 3.1, 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 3.3. 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 3.4. 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 3.5.*

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

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

Listing 3.4: 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 3.5: Rule 5 example

in the query CONDITION element. Here, the query can either be allowed to execute directly or further modified. It is executed 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.

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.

Algorithm 1 The procedure of Policy Class 2

Input: The prohibited level L_i and the exception E.
Output: Decision to directly execute or modify.

```
1: Let Ev = E value
2: for Each level L_j ∈ Below(L_i) do
3:    if L_j exists in the query CONDITION element then
4:       Let Lv = L_j value
5:       if Lv == Ev OR Lv ∈ Under(Ev) then
6:          Allow the query to execute without modification
7:       else if Ev ∈ Under(Lv) then
8:          Replace E by L_j, and Ev by Lv, then inform the user, and allow the query
            to execute
9:       end if
10:   else if L_j exists only in the query SELECTION element then
11:      Add E as new condition to the user query, inform the user, and allow the
         query to execute
12:   end if
13: end for
```

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 the level of the prohibited value L_i exists in the SELECTION element only? In this case, the query should be modified by adding a new predicate obtained from the authorization policy to the query CONDITION element as formalized in Rule 7 below.

Rule 7. *If the prohibited level L_i or any of its more granular levels (Below(L_i)) exists in the* SELECTION *element only, then the query should be modified by adding a new predicate to the query* CONDITION *element.*

Now, let's look at the following example:

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

Listing 3.6: Simple OLAP Query 3

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

Listing 3.7: Rule 7 example

Example 6. *Suppose that Alice is restricted from accessing Quebec's sales. If Alice sends the query depicted in Listing 3.6, the query should be modified as shown in Listing 3.7.*

The associated query summarizes the sales of cities in 2011 for outdoor products. As noted, the SELECTION element contains an implicitly prohibited child level (City), so instead of rejecting the query we modify it by adding (Store.Province != 'Quebec') as a new predicate to the condition. Note that the new predicate applies the authorization policy. In this example, it excludes all the sales cities of the province of Quebec which are not allowed for Alice.

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.

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

Listing 3.8: Rule 9 example

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 P 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 3.8, 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 value 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 P exists in the query* CONDITION *element AND Ev is under P, the query is modified by replacing the prohibited level L_i by the exception level E AND the prohibited level value P 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 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 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. The query is allowed to execute directly if the prohibited level value P is equal to the exception value *Ev* OR P belongs to the values *Under(Ev)*. Conversely, the query is modified by replacing the condition that contains the prohibited value by a new one containing the exception if the exception value *Ev* belongs to the set of values under *P*.

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

3.6 Implementation and Performance Issues

To implement the Below and Under functions, a number of additional algorithms and data structures are needed in order to efficiently 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.

Algorithm 2 The procedure of Policy Class 4

Input: The prohibited value P of level L_i and the exception E.
Output: Decision to directly execute or modify.

```
1: Let Ev = E value
2: for Each level L_j ∈ Below(L_i) do
3:    if L_j exists in the query CONDITION element then
4:       Let Lv = L_j value
5:       if (Lv == P) AND (P ∈ Under(Ev)) then
6:          Add E as a new condition instead of the condition that contains L_j, inform
            the user, and allow the query to execute
7:       else if (Lv ∈ Under(P)) AND (Lv == Ev OR Ev ∈ Under(Lv)) then
8:          Allow the query to execute without modification
9:       end if
10:   else if L_j exists only in the query SELECTION element then
11:      Add E as new condition to the user query, inform the user, and allow the
         query to execute
12:   end if
13: end for
```

3.6.1 The Implementation of the Below Function

We begin by giving a brief description of the primary data structures (i.e., mapGraph) utilized during function execution. The mapGraph [41] was discussed in Chapter 2; however, we briefly review the relevant concepts here. The mapGraph builds upon the notion of dimension hierarchy linearity [83]. We encode a hierarchical dimension table (for example called *dim*) by building a mapping table that is sorted by A_k, A_{k1}, ..., A_1, where A_1 is the base attribute in the hierarchical dimension. For each hierarchical attribute level Att in dim, we change the schema of dim by adding a new column called AttID. The values of AttID are created as consecutive integer IDs. Specically, we associate the consecutive distinct values for each column Att with consecutive integer IDs. Finally, a look-up data structure is built based on these integers in order to support efficient real time transformations between arbitrary levels

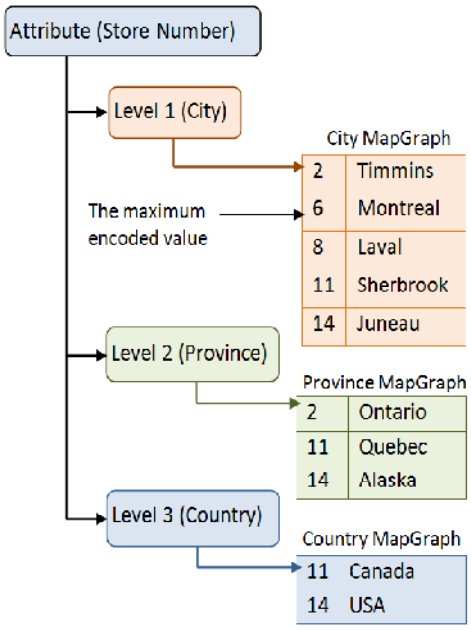


Figure 3.9: The mapGraph of the Store dimension

of the dimension hierarchy. Figure 3.9 provides an illustration of this structure for the Store dimension with the hierarchy Store_Number, City, Province, and Country.

Referring to the Store mapGraph, each record consists of an integer value that represents the corresponding maximum encoded value in the primary attribute (i.e., Store_Number) and a native attribute representation (i.e., values of attributes Types of Store dimension). For instance, the maximum encoded value for the City of Montreal is 6, which means that Montreal has four stores, encoded with the numbers 3 through 6. Using this structure, one can perform a mapping from the most detailed encoded level value to the corresponding sub-attribute value (i.e., attribute level values), and vice versa. In the Store dimension, Store_Number = 7 is located in City = 3 (Laval) and, as a consequence, it is located in Province = 2 (Quebec) in the Country = 1 (Canada).

Performance and storage requirements for the mapGraph are quite impressive. Worst case query time is bounded as O(log n), where n is the cardinality of the destination level of the dimension hierarchy. Moreover, the collective size of a d-attribute mapGraph depends on the cardinality of non-base levels exclusively, which are very small compared to the base level. In practice, this would likely be no more than a few dozen kilobytes for large data cube problems.

Now suppose we have the query in Listing 3.9 and the following policy: the marketing users should not know the sales totals of the individual provinces. In order to make the authorization decision (execute, modify, or reject the query), we have to specify the authorization objects. To this end, the returned attributes and the query conditions in the Selection and Where clauses should be inspected against the policy. We note that the query returns the sales of individual cities, which can be used to compute the restricted values. However, we need an automated way to determine if a restricted level value can be computed from another level. The Below function is responsible for that. When it is invoked, it takes the prohibited level (i.e., Store.Province in our example) as an argument and uses the mapGraph to retrieve a list consisting of the prohibited level and all the lower levels of the associated dimension hierarchy (i.e., Province, City, and Store_Number). Then, according to the proposed Rules, one can determine if the query is safe to execute or not. In this example, according to Rule 1 (i.e., the query should be rejected if a restricted level or any of its children exists in the query), the query should be rejected because one of the prohibited levels (Store.City) exists in the returned list.

```
Select Product.Name, Store.City, Sum(sales)
From Product, Store, Sales
Where Product.product_ID = Sales.product_ID
  AND Store.store_ID = Sales.store_ID AND
   Store.Province = 'Quebec' AND
   (Product.Name = 'LN*' AND
    Product.price >= 24000)
Group by Product.Name, Store.city
Order by Product.Name, Store.city;
```

Listing 3.9: Simple SQL OLAP Query

3.6.2 The Implementation of the Under Function

The Under function is invoked when the policy is less restrictive, as is the case in the following situation: an employee Alice should not know the sales total for the country of Canada. Alice is restricted from accessing specific value (the sales of Canada). Suppose that Alice sends the query in Listing 3.9, assuming this less restrictive policy. We note that the user query has two conditions, with the second condition related to the policy (i.e., Store.Province = 'Quebec'). The condition filters the sales for the province of Quebec. However Alice should not know the sales total for the country Canada, so we have to determine if the province of Quebec is in Canada or not (i.e., if Quebec is *Under* Canada). By using the Under function, one can retrieve the encoded values of Canada and Quebec from the mapGraph structure, then find if there is an intersection between them. If so, we say that Quebec is Under Canada. In our example, Canada has Stores with encoded numbers 1 through 11, and Quebec has Stores of encoded numbers 3 through 11. Clearly, there is an intersection between them. According to Rule 6, the query should be rejected because Alice attempts to access a restricted data.

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, we need another structure that deals with non-hierarchical attributes. The FastBit [128] bitmap index structure is used to create a very efficient compressed bitmap index. Fastbit uses the Word-Aligned Hybrid compression mechanism to compress the bitmap indexes [128]. This compression scheme produces a FastBit compressed index that is up to 10 times faster than the compressed bitmap index (run-length encoding) implementation from popular commercial database management (DBMS) product (i.e., Microsoft and Oracle). FastBit compressed bitmap indexes for each non-hierarchical attribute also provide very efficient searching and retrieval operations compared with other techniques such as the B+ tree or B* tree.

Given FastBits performance and its open source license, we chose to integrate the FastBit library with our framework. This integration 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 3.10(a), while Figure 3.10(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.

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 3.9. Since the Product Price and the Product Name are non-hierarchical attributes, we

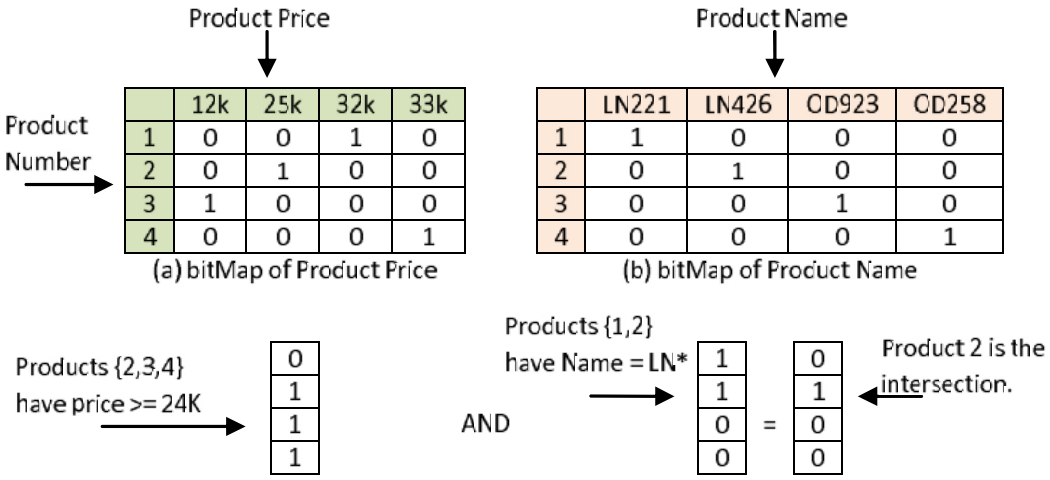


Figure 3.10: The bitMap of Product Price and Product Name

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 3.10 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 are 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 3 summarizes the logic of the checking process.

3.7 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

Algorithm 3 The processing logic of Below and Under Functions

Input: The policy condition S, and the Query Q.
Output: Returns True if Q is valid, False otherwise.

```
1: Initialize the mapGraph (hM) and the bitMap (fB) if they have not been initialized
2: Let QA be the query attributes
3: if S has a hierarchy attribute then
4:    Let SR be the range of S using hM
5: else
6:    Let SR be the range of S using fB
7: end if
8: for each attribute a_i in QA do
9:    if a_i is hierarchy attribute then
10:      Get the range of a_i QR using hM
11:      if QR ∩ SR ≠ ∅ then
12:         Return False
13:      end if
14:   else if a_i is non-hierarchy attribute then
15:      Get the range of a_i QR using fB
16:      if QR ∩ SR ≠ ∅ then
17:         Return False
18:      end if
19:   end if
20: end for
21: Return True
```

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 authentication 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 (Sidera). Sidera provides its own query optimizer and storage facilities. However, for testing purposes, 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 authentication model has the potential for integration with standard database servers.

For the above reasons, we have plugged-in a standards-compliant DBMS server (i.e., MonetDB) as the backend database. MonetDB is a popular open source database management system [94]. It 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. In the current context, MonetDB is essentially responsible for execution of the final query, while the Sidera DBMS components described in the chapter carry out all authentication and authorization operations.

3.7.1 The Test Environment

We utilize the Star Schema Benchmark (SSB) [92], 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 analytic 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 A. 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 chapter, it was necessary to develop SQL conversion middleware, a significant research effort of its own. The details of the middleware architecture are described in the next chapter.

For the following tests, we have used the SSB generator (with default settings) to produce a Fact table of 6 million records, with each dimension housing up to 200,000 records. These tables are created and loaded into the backend MonetDB. 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 $<$ 1998 is restricted except d_year = 1995 or 1996).
- The last constraint prohibits access to a cuboid as a whole.

These policies are enforced by our framework and applied against the SSB queries. In particular, policies are defined as a set of conditions that are stored in the Authentication database. Once a query is arrived, it is parsed and decomposed into its algebraic components for authentication and authorization. The user credentials are obtained from the query, and used to authenticate the user. If authentication is successful, the applicable policies are retrieved from the repository based on the user credentials. Finally, the query elements (i.e., the returned attributes in the Select clause and the conditions in the Where clause) are validated against the policies. If there is no violation, the query is sent to the MonetDB management system for execution against the SSB data as originally written, otherwise, the query is modified according to the authorization rules and the modified version is then executed.

For example, suppose that our framework received the query *Qc* depicted in Listing 3.10 (i.e., Query number 3 of SSB) with the existing policies. As noted, policy number 3 is violated by the last condition listed in the Where clause of *Qc*. Therefore, according to the authorization rules, *Qc* should be modified before executing by replacing the condition that breaches the policy with a new one *Cnew* derived from the policy exception. Specifically, the condition (i.e., d_year BETWEEN 1992 AND 1997) is replaced with the *Cnew* (i.e., d_year = 1995 or d_year = 1996) to become safe query. Listing 3.11 depicts the modified version *Qm*.

3.7.2 The Test Results

In terms of the results, we have isolated each of the four query classes and show authentication processing versus the subsequent query execution time in Figure 3.11, Figure 3.12, Figure 3.13, and Figure 3.14. Note that for queries which violated the policy and were not candidates for re-writing, the query execution time is still listed

```
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 BETWEEN 1992 AND 1997
GROUP BY c_city, s_city, d_year
ORDER BY d_year asc, revenue desc;
```

Listing 3.10: Query3 of SSB

```
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 = 1995 OR d_year = 1996
GROUP BY c_city, s_city, d_year
ORDER BY d_year asc, revenue desc;
```

Listing 3.11: A Modified Version of Query3of SSB

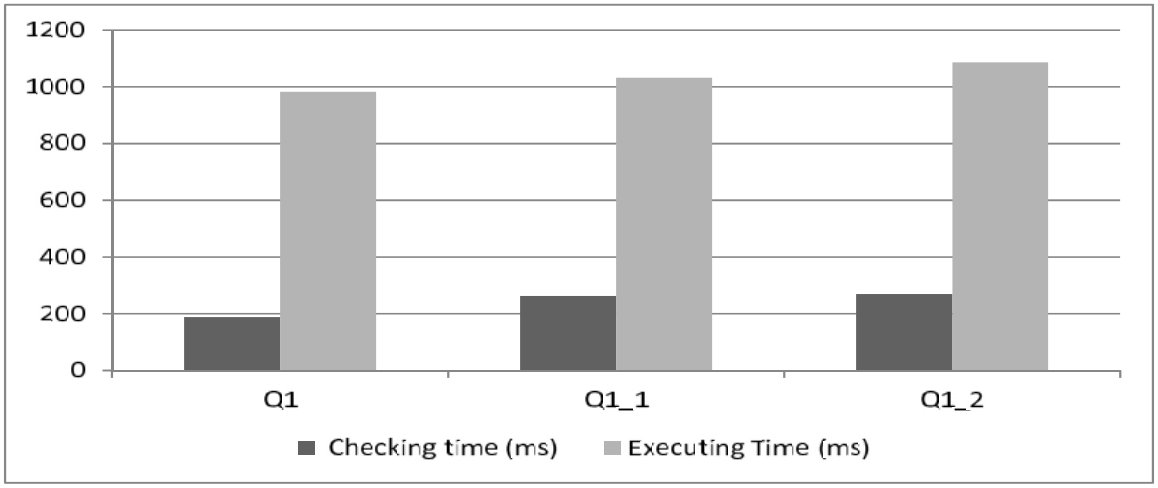


Figure 3.11: Performance for SSB schema, Query 1 category

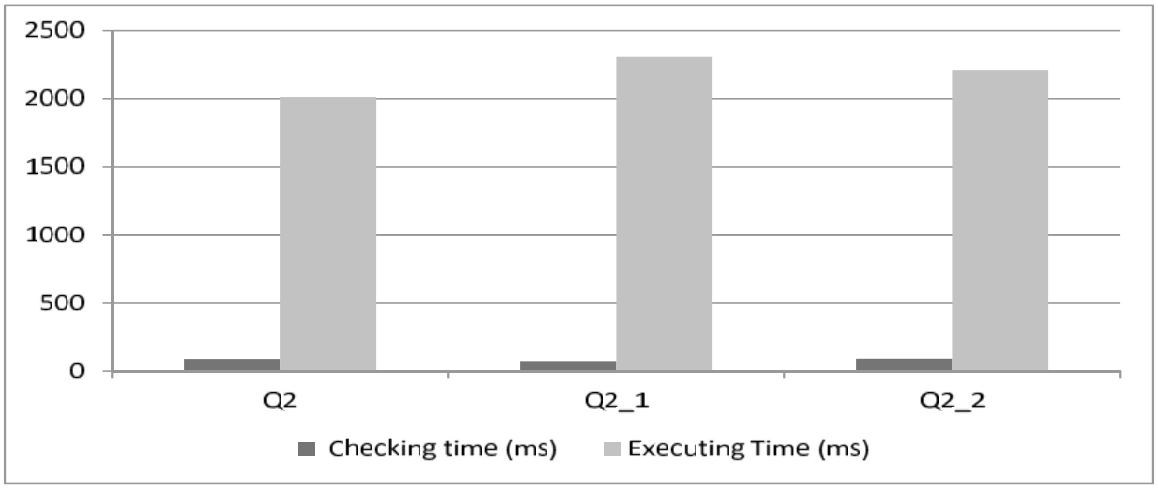


Figure 3.12: Performance for SSB schema, Query 2 category

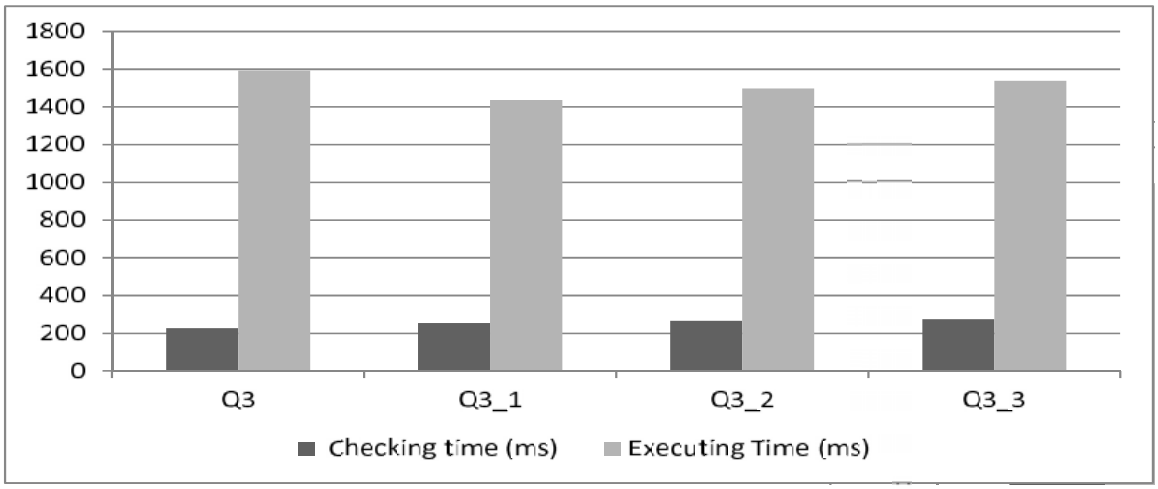


Figure 3.13: Performance for SSB schema, Query 3 category

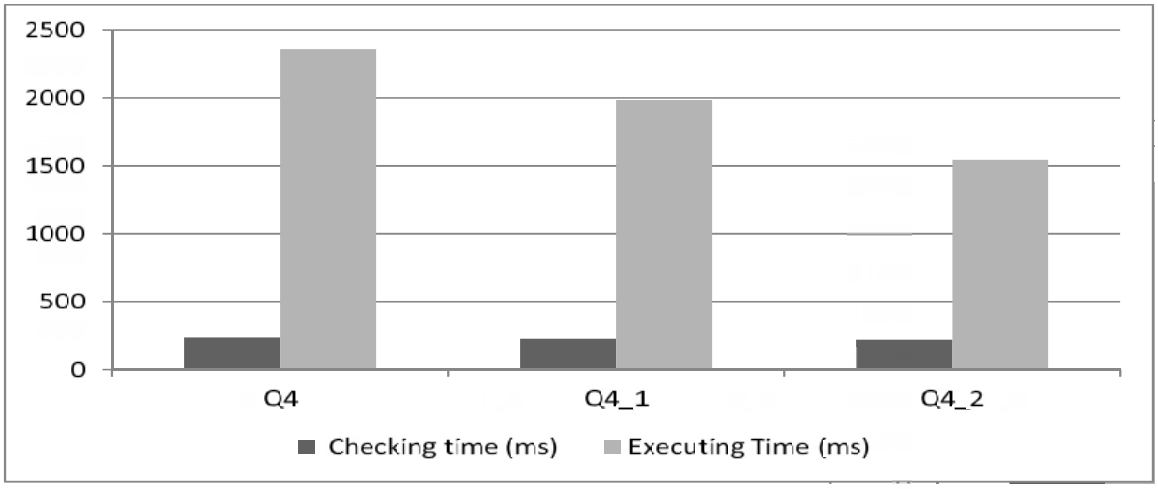


Figure 3.14: Performance for SSB schema, Query 4 category

so as to give the reader a better sense of the relative balance between checking and execution. A few additional points are worth noting. First, the ratio of checking time to execution time varies considerable, 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, execution times range from less than a second for Query Class 1 to about 2.4 seconds for Query Class 2. As the database gets larger, of course, these times will continue to grow. Second, the checking costs are quite modest, in the range of 80-300 milliseconds. More importantly, the size of the underlying database has little effect upon the checking costs, as only the cube meta data is inspected. 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. The execution times for traditional row store database servers can be one to two orders of magnitude larger. In such environments, the ratio of checking to execution costs would be far more extreme. Figurer 3.15, Figurer 3.16,

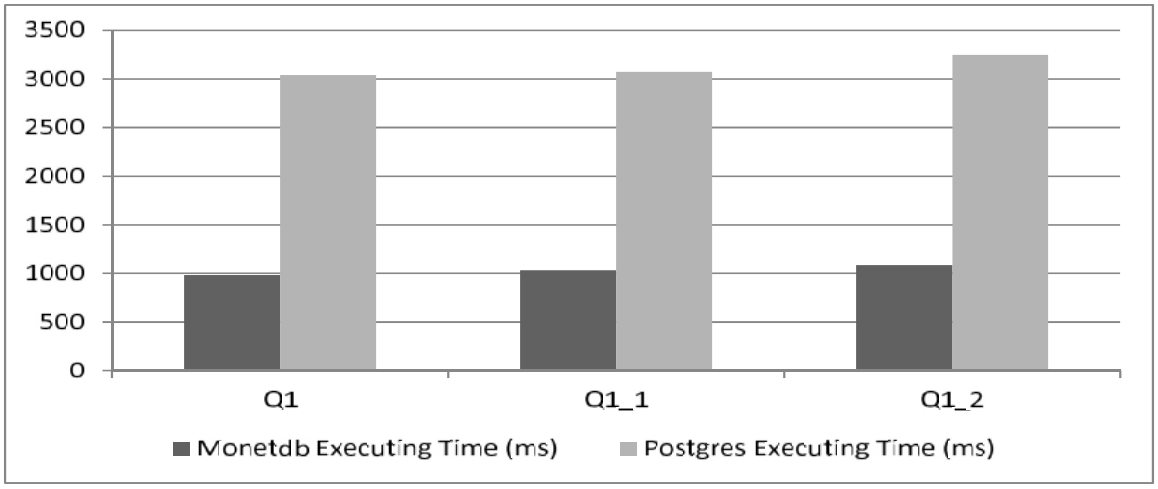


Figure 3.15: Performance for SSB schema on Monetdb and PostgreSQL, Query 1 category

Figurer 3.17, and Figurer 3.18 show the execution time for SSB queries using Monetdb (column-oriented) and Postgre SQL (row-store) database systems. One must note the difference between them, which results from the amount of data read from disk. Specifically, in column-store database systems each attribute is stored in a separate column, such that successive values of that attribute are stored consecutively on disk. This is in contrast to most common database systems (i.e., Oracle, Microsoft SQL Server, and Postgre SQL) database systems that store relations in rows (row-stores) where values of different attributes from the same tuple are stored consecutively. Further experiments and discussions will be presented in Chapter 4.

3.8 Conclusions

In this chapter, we have discussed a query re-writing model to provide access control in multi-dimensional OLAP environments. We began by high lighting a conceptual model that focused on the data cube and its constituent dimension hierarchies. From

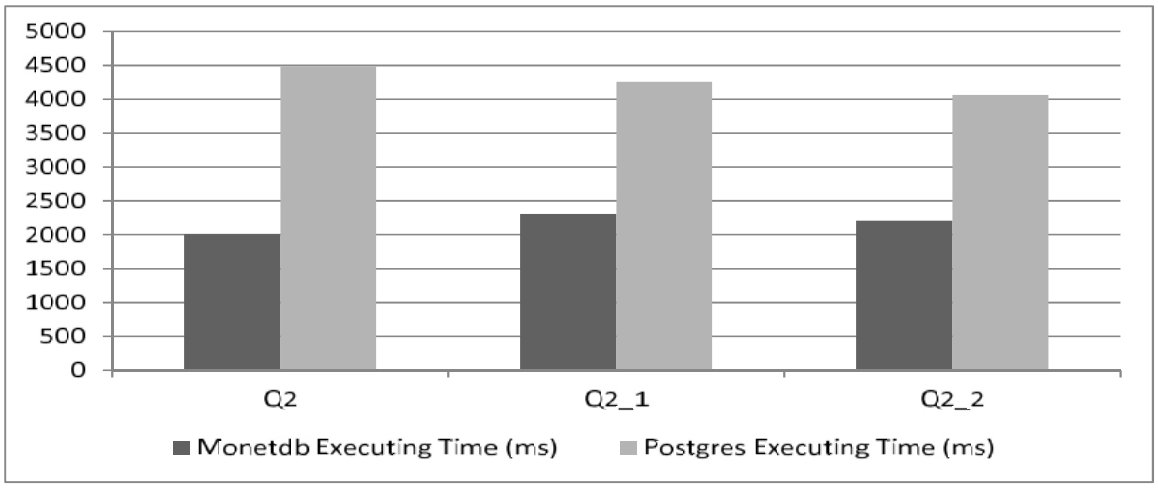


Figure 3.16: Performance for SSB schema on Monetdb and PostgreSQL, Query 2 category

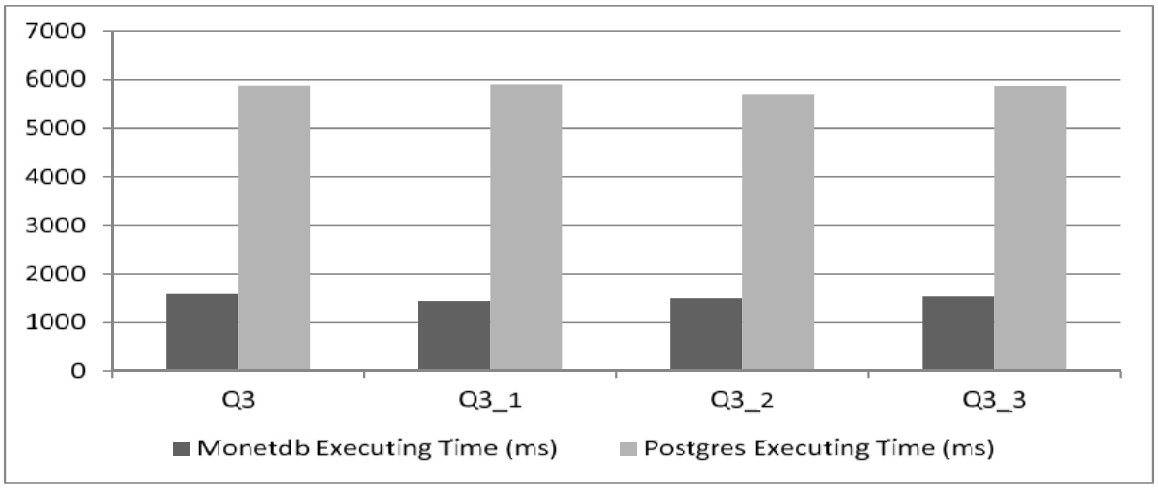


Figure 3.17: Performance for SSB schema on Monetdb and PostgreSQL, Query 3 category

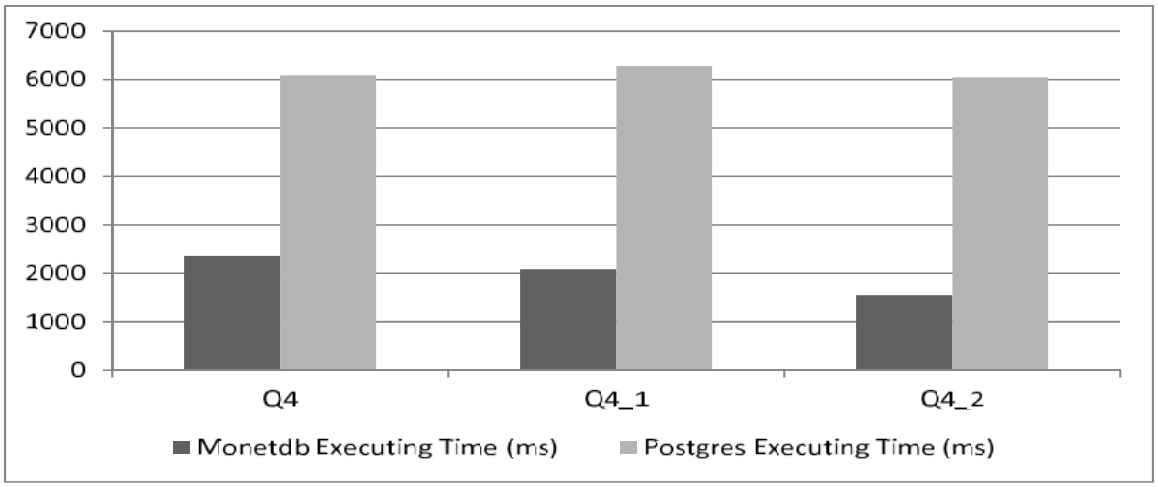


Figure 3.18: Performance for SSB schema on Monetdb and PostgreSQL, Query 4 category

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.

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 time. 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 chapter 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. In the next chapter we will present mechanisms for inference detection that will piggy back on top of the core authentication and authorization framework to provide an even greater level of security for OLAP data.

Chapter 4

Protecting OLAP Cubes from Inference Attacks

4.1 Introduction

In the previous chapter, we discussed a query re-writing model to provide access control in multi-dimensional OLAP environments. In short, we introduced a series of rules to decide whether user queries should be rejected, executed directly, or transparently and dynamically transformed. The model can be used to ensure that user queries pertaining to sensitive data will not be answered. However, users can still uncover sensitive information by combining non-sensitive data with their own knowledge (e.g., external knowledge). In practice, a user can issue a sequence of non-sensitive queries whose answers, when taken together, will allow the user to infer something that was meant to be protected. We refer to such privacy breaches as inference problems, and the goal of inference control is to protect sensitive data from being inferred.

Two general environments have been targeted by inference control mechanisms: the *interactive* and *non-interactive* settings. With the non-interactive setting, the original database is first *sanitized* so as to preserve privacy and then the modified

Employee	Year	Sales
Tom	2011	100
Jim	2011	120
David	2011	90
Tom	2012	120
Jim	2012	125

(a)Sales Relation.

	2011	2012	SUM(Sales)
Tom	100	120	220
Jim	120	125	245
David	90		90
SUM(Sales)	310	245	

(b)Two-dimensional Data Cube.

Figure 4.1: An example of an Inference Problem

version is released. In the interactive setting, on the other hand, a security mechanism is placed between the users and the database. Queries posed by the users, and/or the responses to these queries, may be modified in order to protect the privacy of the respondents.

In this work, we consider the interactive setting (OLAP systems), where the user poses aggregate queries in order to analyze data and then navigates through the details interactively. These queries usually involve aggregating a large amount of data, with the result expected in a few seconds. Such a fast response is often achieved through comprehensive pre-processing, often focusing on the materialization of multi-dimensional aggregates (represented as a data cube) from which the answer to queries can be more easily derived. Although these aggregates give the user the ability to analyze the data in real time, they can also be used to infer sensitive data.

For example, Figure 4.1(a) shows a base table that is used to construct the data

cube in Figure 4.1(b). Here, the attributes Employee and Year are dimensions and the attribute Sales is the measure. Suppose the company invites an analyst Bob to analyze the data, but worries that Bob may misuse the sensitive information about each individual. Consequently, they restrict access to the measure values. Aggregations of those values however are allowed to be retrieved. Moreover, Bob can identify the empty cells. An inference problem may occur if Bob sums the total sales of years 2011 and 2012, then subtracts from the result (e.g.,555) the total sales of Tom and Jim (220 and 245 respectively). The individual sales of David are then successfully inferred as 90. In fact, Bob can infer all the restricted values in the base table in the following way. Bob finds the maximum sales in 2011 and gets 120 as the result. He can then infer that the sales of Jim must be 100 or 120, though he cannot match the values to exact cells. Bob then finds the minimum sales of Jim and gets 120. Now, he can infer that the sales of Jim and Tom in 2011 are 120 and 100 respectively. By subtracting these values from their total sales, Bob also infers their sales in 2012 as 125 and 120 respectively.

The inference problem has been studied extensively in statistical databases [2]. In general, the proposed methods restrict the queries or perturbate the original data or answers of queries in order to provide sufficient protection of sensitive information. However, most of the methods considered in statistical databases have two limitations. First, they adopt a detect-then-remove strategy, which requires complex computations over the data and hence are too expensive to be applied in large OLAP systems. Second, some of the proposed methods are only suitable for limited situations. For example, they provide privacy for only one type of aggregation (e.g., SUM) or provide privacy for a fixed sensitivity criteria. Because of these limitations, most modern

commercial OLAP systems lack effective security countermeasures or, at best, only utilize a simple — and often inadequate — security model.

In this chapter, we present a robust model for controlling malicious inferences. The model is motivated by Cross-Out the work first presented by Wang et al. [119], in which the authors suggested a possible approach for inference prevention. Instead of detecting inferences directly, complex inferences are prevented by restricting queries. The approach does not depend on specific types of aggregation functions, external knowledge, or sensitivity criteria. We note, however, that this approach has not yet been implemented in real world settings, and its viability in practice has remained an open question. In fact, straightforward or direct implementations are likely to be very slow. Therefore, in this part of our work, we present a series of algorithms and data structures that can be used to efficiently implement the general model. In particular, we address performance concerns associated with the extensive disk accesses that are required in order to make inference assessments. To further ground the research, we have integrated the inference control model on the top of a DBMS designed specifically for OLAP (Sidera), as well as a pair of popular relational database management systems, and have evaluated its efficiency against common benchmarks.

The remainder of this chapter is organized as follows. In Section 4.2, we present an overview of related work. The objects and methodology are presented in Section 4.3. Section 4.4 provides a brief overview of the basic terminology and structures relevant to the inference problem in general. Section 4.5 and 4.6 discuss the primary contribution of the chapter. First, data cube inferences are illustrated through a series of representation examples, then methods and algorithms are presented in Section 4.6 to prevent such inferences. Section 4.7 describes the architecture of our prototype

framework along with its primary components. The experimental results are then presented in Section 4.8, with final conclusions offered in Section 4.9.

4.2 Related Work

There is a rich literature on the inference problem, principally from the statistics community [35, 113, 30, 28, 34, 33, 112]. The proposed methods can be classified into two broad categories: perturbation-based and restriction-based methods. The perturbation-based inference control methods prevent malicious inference by adding noise to data to provide approximate answers. These methods can be organized into:

- *Input Perturbation*, where the original data are randomly modified, and answers to queries are computed using the modified data.
- *Output Perturbation*, where correct answers to queries are computed exactly from the real data, but noisy versions of these are reported.
- *Rounding*, where the original data values are replaced with rounded ones.

The methods based on data perturbation have many limitations. Firstly, if the data are modified, re-identication by means of matching algorithms is harder and uncertain. Secondly, when a user re-identifies data, he/she cannot be confident that the compiled data is consistent with the original data. Finally, because OLAP tasks may require low-level details, potential errors in modified values may be significant, preventing OLAP users from gaining trustworthy insights.

The restriction-based methods do not alter data; rather, they produce partial suppression or reduction of detail in the original data. There are many techniques in this category:

- *Table Restriction* [28], which restricts the complete tables that are likely to have a positive disclosure.

- *Cell Suppression* [35], which suppresses the sensitive cells that can cause disclosure of individual data so that possible inferences can subsequently be detected and removed. While such a detection method is effective for two-dimensional cases, it is intractable for three or more dimensional tables, even for small sizes [22].

- *Microaggregation* [33, 124], which replaces clusters of sensitive values with their averages. However, this method defines partitions without considering the rich hierarchies inherent in data cubes, and hence the results may contain many clusters of values that are meaningless to OLAP users.

- *Query Restriction* [79, 80], which rejects unsafe queries that can lead to a compromise, depending on the number of values involved in the query answer. However, this technique does not consider aggregations in data cubes, where the distinct values can be inferred from aggregations.

The privacy problem has also been considered in the data mining area. A number of techniques such as Randomization and k-anonymity [44, 37, 3, 93] have been suggested in recent years. In Randomization, the sensitive data are modified by adding random distortion values in order to create data distributions that can be used for data mining purposes [3]. In most cases, the underlying data cannot be recovered and thus, may be of marginal use. It is also important to note that OLAP users heavily depend on ad-hoc queries that aggregate small, overlapping sets of values. The precise answers to such queries are not obtainable from distribution models alone, even

if the models can be successfully reconstructed.

In k-Anonymity [109, 93], the underlying data is not destroyed; instead, the granularity of the data representation is reduced in such a way that a given record cannot be distinguished from at least (k-1) other records. So any attempt to link an individual in the physical world to (the sensitive values in) such records will end up with at least k indistinguishable choices. In [17], it has been shown that optimal k-anonymization is NP-hard. Furthermore, the technique is not effective with increasing dimensionality, since the data can typically be combined with either public or background information to reveal the identity of the underlying record owners. Similarly, there are a number of other partition-based privacy models such as L-diversity [77] and t-closeness [70] that model the adversary differently and have different assumptions about background knowledge.

Query auditing techniques have also been utilized [68, 64]. Here, the query is denied if the response could reveal sensitive information and answered exactly otherwise. This technique returns precise answers to queries; however, it often needs either complicated computations over the entire data set or the bookkeeping of every single answered query. This results in prohibitive on-line performance overhead and storage requirements.

It also has been pointed out that denying a query can actually leak extra information to snoopers in some cases [64]. A mechanism called Simulatable Auditing has been proposed in response to this issue. Simulatable Auditing implies that the decision of answering or denying a query can be deduced by both auditor and attacker. In other words, the auditor makes the decision based only on the past answered queries and the newly posted query, without accessing stored data values. Thus, if there

exists a result for the newly posted query which is consistent with the past queries and causes privacy disclosure for one variable, the query is denied. Simulatable Auditing has two drawbacks. First, utility is greatly reduced because it denies many queries which are not actually harmful. Second, it adds computational overhead since for each consistent result, the auditor needs to examine if there is privacy disclosure occurring, in addition to the work of finding the consistent result.

Recently, differential privacy has been proposed as a rigorous privacy model that makes no assumption about an adversary's background knowledge [36]. A differentially-private mechanism ensures that the probability of any output data is equally likely from all nearly identical input data sets and thus guarantees that all outputs are insensitive to any individual's data. Most of the differential privacy methods are based on an interactive framework [48], where a user can pose aggregate queries through a private mechanism, and a response is returned after adding noise to it. To ensure privacy, a database owner can answer only a limited number of queries before she has to increase the noise level to a point that the answer is no longer useful. Thus, the database can only support a fixed number of queries for a given privacy budget. This is a big problem when there are a large number of users because each user can only ask a small number of queries. In a non-interactive framework, the database owner first anonymizes the raw data and then releases the anonymized version for public use [86]. However, this approach is not suitable for high-dimensional data with a large domain because when the added noise is relatively large compared to the count, the utility of the data is significantly diminished.

Finally, the inference problem has also been studied in OLAP systems. There are two types of methods that have been proposed: inference control [72, 120, 122] and

input/output perturbation [4, 38, 100], which follows the same principle of perturbation techniques as seen in statistical databases. Specifically, it either perturbs the original data stored in the data warehouse server with random noise, or adds random noise to the query answers in order to preserve privacy. Existing studies support COUNT [4] and SUM [100] queries for input perturbation, and general aggregation functions for output perturbation [38]. While this approach can be very useful for privacy protection, precise query answers (without perturbation) are often preferred in OLAP systems when important decisions need to be made based on the data [87]. Thus, we focus on the inference control approach in this work.

With the inference control approach, after receiving a query from a user, the inference control mechanism determines whether answering the query may lead to an inference problem, and then either rejects the query or answers it precisely. Most existing solutions for inference control in OLAP are suitable for only one type of aggregation, SUM-only [72, 120, 121, 122], COUNT-only [4], or MIN/MAX-only [87]. It is important to note, however, that OLAP relies on complex queries involving the aggregation of data with mixed aggregate functions, such as SUM, MIN, MAX, etc., so such restrictions are not likely to be accepted in practice.

4.3 Objectives and Methodology

4.3.1 Objectives

In this chapter we discuss the issue of inference attacks on protected data and present a general model for controlling malicious intrusions within the OLAP domain. Building upon ideas from the recent literature, we will describe mechanisms to balance the accessibility with appropriate access constrains. Specific objectives include:

- Improve the performance of the inference checking process to the extent that the checking phase has minimal impact on real time query requirements.
- To clearly define the primary data structures required to support this approach.
- To present efficient algorithms that utilize these data structures in a manner that allows the performance objectives to be reached.
- To provide complexity analysis, where necessary, to demonstrate that execution costs — using parameter settings relevant to the OLAP domain — can be appropriately bounded.
- To ground the underlying research with a concrete prototype that demonstrates practical viability.

4.3.2 Methodology

Although many methods have been proposed in order to preserve privacy in the database context, the majority are suitable only for special, restricted cases. A small number have been shown — in theory at least — to be capable of providing inference protection that could be extended to the OLAP domain, but the computational overhead is clearly prohibitive. In this chapter, we build on recent research that has strong theoretical support but whose real world viability has not yet been demonstrated. In fact, it is the design, implementation and evaluation of the algorithmic framework that represents both the primary objective and achievement of this phase of the research. We emphasize that the maturity of the database field, coupled with scale of current industrial applications, all but requires that abstract ideas be grounded in such a way.

We therefore propose a series of algorithms and data structures that support efficient real time inference control. To demonstrate practical applicability, we will integrate the associated inference control module into common DBMS products — both row-store and column-store — to demonstrate performance on accepted benchmark test suites. We will show that the experimental analysis supports the claim that inference checking can, in fact, be carried out without a meaningful impact upon final query execution times.

4.4 Basic Concepts

Before going into the details of the proposed methods, we first briefly discuss basic concepts that are relevant to our research.

4.4.1 OLAP System Model

We have discussed the basic concepts of the OLAP model in Chapter 2. We now review these concepts through a motivating example. Figure 4.2 shows the underlying *Star Schema* database that is used to create a Data Cube. The central table (Sales) is the Fact table that contains *measure* values used for analysis, and the surrounding tables are the Dimension tables (Customer, Product, and Time) that contain descriptive attributes. Each dimension attribute is partially ordered into a dependency lattice by the dependency relation $\preceq$ to produce a hierarchy. For instance, the Customer dimension has the following lattice: custkey $\preceq$ c_nation $\preceq$ c_region $\preceq$ all. The product of the previous lattices gives distinct dimension combinations (aggregations or cuboids) that collectively represent the data cube.

Figure 4.3 represents a simple *2*-dimensional data cube lattice based upon the

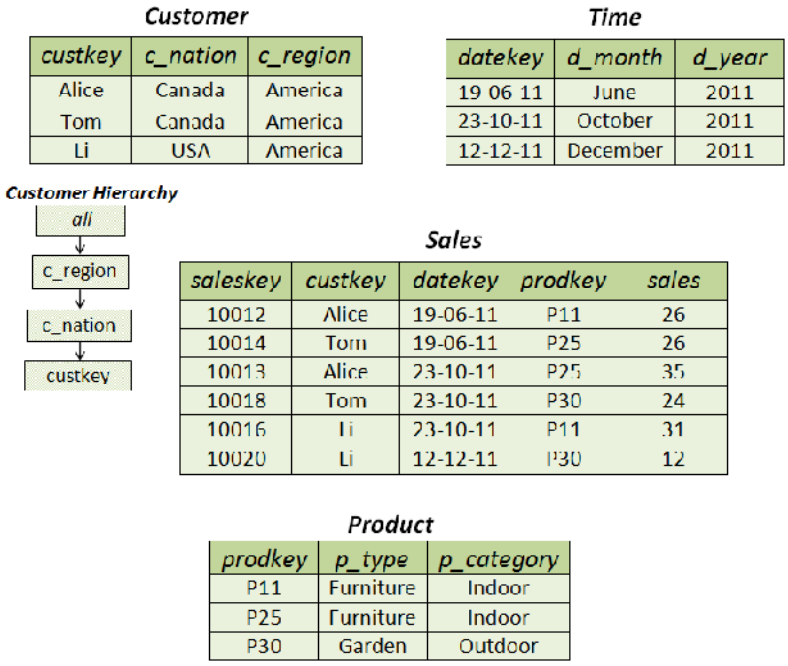


Figure 4.2: A simple database instance

Customer and Date dimensions — a more complex lattice will be presented later in the chapter. The base cuboid contains the individual records that are used to populate/compute the cuboids. All cuboids are related by an *Ancestor* or *Descendant* relationship if the dependency relation exists amongst their attributes. We say that a cuboid c_i with attributes $(a_1, a_2, \ldots, a_n)$ is an Ancestor to a cuboid c_j with attributes $(b_1, b_2, \ldots, b_n)$ if the c_i attributes are partially ordered with respect to c_j, such that $b_1 \preceq a_1, b_2 \preceq a_2, \ldots, b_n \preceq a_n$. For example, the cuboid <c_region,datekey> is an ancestor to the cuboid <c_nation,datekey>.

The dependency relation also exists amongst cuboids cells. For example, the cuboid <c_region, datekey> depends on the cuboid <c_nation, datekey>; hence, the cell <America, 23-10-11> of the former also depends on the cells <Canada, 23-10-11> and <USA, 23-10-11> of the latter and can be computed using them. Generally,

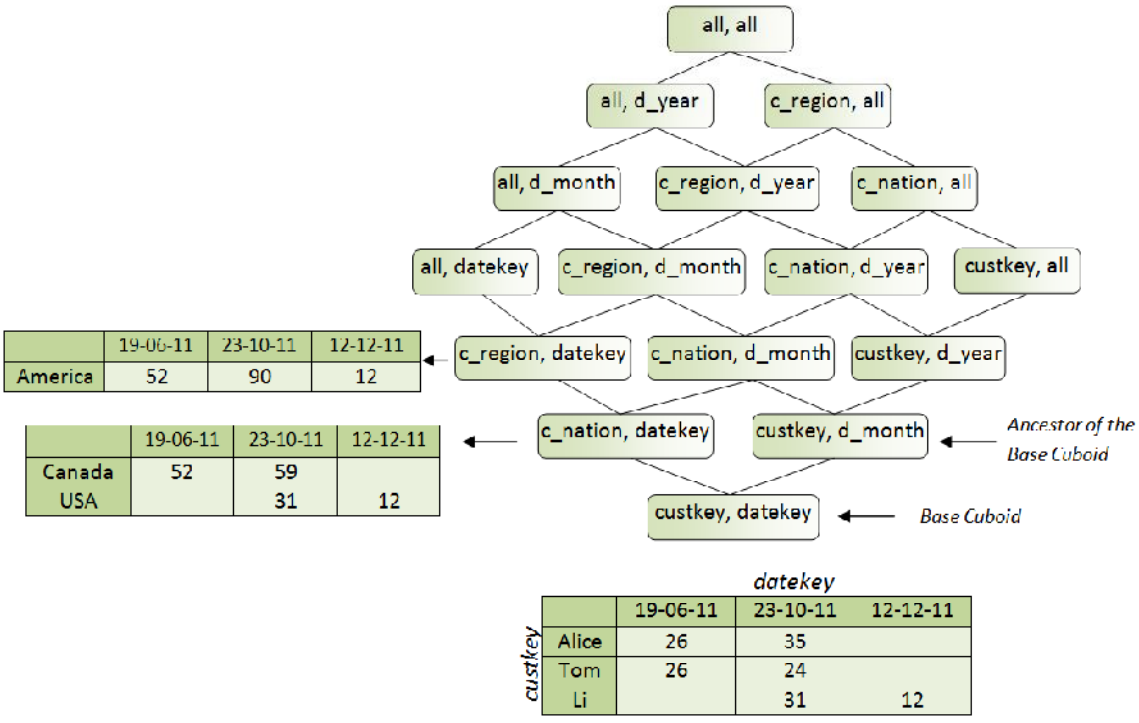


Figure 4.3: A simple 2-dimensional data cube

any cuboid can be computed using its descendant cuboids or, of course, using the base cuboid, as it depends on both.

4.4.2 External Knowledge

A user may have previous knowledge about data that is obtained from external sources outside the OLAP system. When the user combines this knowledge with his/her legitimate query answers, sensitive data can be inferred. We refer to such knowledge as the *external knowledge* of the user. There are various types of external knowledge.

Example 8. Due to privacy concerns, suppose that the administrator of the data cube depicted in Figure 4.3 does not want a user Bob to have access to the base cuboid, but will allow access to its ancestor <c_nation, datekey>. If Bob knows about empty cells — as external knowledge — then the Sales of Li in (12-12-11) can be inferred from just one aggregate query over the ancestor cuboid <c_nation, datekey> on datekey (12-12-11). Moreover, if Bob knows that the Sales of Alice and Tom in (19-06-11) are equal, then these values can also be inferred as 26, which is half the sales of 52 retrieved from the cuboid <c_nation, datekey> on datekey (19-06-11).

It is difficult to present a universal model that captures all external knowledge types. Developing such a model is still an open problem drawing considerable attention from data privacy studies [84, 16]. In this work, we assume that if the user knows the value as pre-knowledge, then the value is accessible (not sensitive) to the user. This assumption can also be applied to cuboids.

4.5 Protecting OLAP Cubes from Inference Attacks

In the previous chapter, we discussed an OLAP-aware framework to provide access control in multi-dimensional OLAP environments. The framework prevents any sensitive value from being computed from below (from its descendants). However, in many cases sensitive data can be inferred from above (from its ancestors). Such inferences can easily infiltrate the first line of defense established by the access control framework. This section addresses inference control in data cubes. We begin by presenting a number of examples that illustrate various inference scenarios. Then, methods that target inference attacks are discussed.

4.5.1 Inferences in OLAP Data Cubes

An inference problem occurs when the value of a sensitive data item is disclosed to users by combining non-sensitive data with their own knowledge (external knowledge). In practice, a user can issue a sequence of non-sensitive queries whose answers, when taken together, will allow the user to infer something that was meant to be protected.

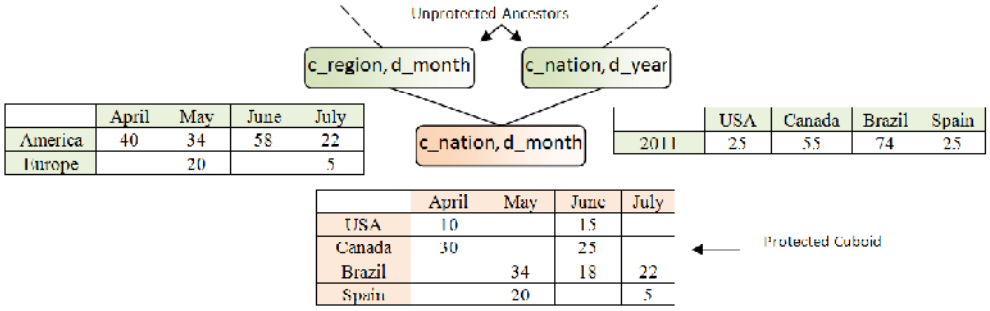
Many methods have been proposed to address the inference problem. However, most adopt a detect-then-remove approach that requires complex computations over the data and hence are too expensive to be applied in large OLAP systems. For instance, the auditing method proposed in [18] has been shown to be NP-hard. Others are only suitable for limited situations. For example, only one type of aggregation is allowed, SUM-only [72, 122], COUNT-only [4], or MIN/MAX-only [87]. Unfortunately, a practical OLAP system does not restrict its users to SUM-only or COUNT-only. Complex OLAP queries rely on different aggregate functions, such as

SUM, COUNT, MIN, MAX, etc. Moreover, inference checking time should be modest relative to query execution time since the checking is done at run time.

We note that some of the inference cases discussed earlier can be addressed in relatively straightforward ways. For instance, in Example 8, Bob infers the cell value of <Li, 12-12-11> because the query is issued over an extremely small set size (only one value). That is, the inference is made when a query with a set size = 1 is answered. A simple solution is to block such queries, so the inference can be identified. This technique has been considered in statistical databases [29] and also investigated and formalized in the OLAP domain [129]. A second inference occurs when Bob infers cell values of <Alice, 19-06-11> or <Tom, 19-06-11>. This can also be prevented by assuming one of these cells to be an empty cell. As a result, the same solution can be applied since the set size of the query is then equal to 1. However, there still exists another important kind of inference that cannot be easily solved. The following example illustrates this case.

Example 9. Figure 4.4 shows three cuboids (<c_nation, d_month>, <c_region, d_month>, and <c_nation, d_year>) taken from the *2*-dimensional data cube lattice depicted in Figure 4.3. Suppose that Bob is restricted from accessing cuboid <c_nation, d_month>, but can access its ancestors cuboids (<c_region, d_month> and <c_nation, d_year>). An inference is possible using SUM-only queries in the following way.

- Bob sums the two cells <America, April> and <America, June> in the ancestor cuboid <c_region, d_month> and gets 98 as the result.
- Then he subtracts from the result the two values of <USA, 2011>, <Canada, 2011> (25 and 55 respectively) in the other cuboid <c_nation, d_year>.



	April	May	June	July
America	40	34	58	22
Europe		20		5

	USA	Canada	Brazil	Spain
2011	25	55	74	25

	April	May	June	July
USA	10		15	
Canada	30		25	
Brazil		34	18	22
Spain		20		5

Figure 4.4: Another example of inference problem on data cubes

- The protected cell <Brazil, May> is then correctly inferred as 18.

In this case, there is no longer a straightforward solution for inference control. Moreover, we note that a second inference is also possible using MAX, MIN, and SUM, as indicated in the following query plan:

- Bob applies the MAX to <2011, Spain> and <Europe, July> and gets 20 and 5 as the result, respectively.
- He can infer that one of the three cells <Spain, April>, <Spain, May>, and <Spain, June> must be 20, but he does not exactly know which is 20.
- Bob then sums the sales of Spain and gets 25 as the result. He concludes that one of these three cells must be 20 and the other two cells are zeroes.
- Finally, Bob applies the MIN to Europe's cells and determines all the sales of Spain. Consequently, Bob can infer all the protected cells by applying additional queries, and hence the whole data cube can be discovered.

From the above examples, we note that the inference is made possible by accessing multiple ancestors of the protected cuboid. We can formalize this type of inference problem as follows.

Definition 5. *Given a data cube containing n cuboids (c_1, c_2, ..., c_n), inference may occur when values of a protected cuboid c_i can be computed from other unprotected cuboids c_j, such that $c_i \preceq c_j$ holds, and $i, j \in n$.*

4.5.2 Preventing Malicious Inference

As noted, protected data can be inferred by accessing its ancestors as shown in Example 9. A simple and direct solution to control this inference is to decline all access to the ancestor's values. However, such an approach is far too restrictive, as it hides too much information from users. While it meets the security requirements, it also renders OLAP systems useless. An ideal solution should maximize the availability of queries while at the same time providing satisfactory security.

Wang et al. [119] proposed such a solution, where malicious inferences are prevented through *restrictions*. Specifically, a set of cuboids free of inference problems is first computed with respect to the protected cuboids, then the user is restricted to access only this *safe set*. That is, the resulting set acts as a virtual tier between the user queries and the data cube, where all cuboids belonging to this set are considered as unprotected cuboids while the remaining are protected. We refer to the safe set as the *Answerable* set. Example 10 illustrates the process.

Example 10. Suppose the administrator of the data cube depicted in Figure 4.5 restricts an employee Alice from accessing the cuboid c <c_region, datekey>. As noted, restricting access to only this specific cuboid c does not, however, prevent Alice from revealing it, as inferences are possible from unprotected cuboids to the protected one. As shown in Example 9, by combining the two ancestors of c (e.g., <all, datekey> and <c_region, d_month>) Alice can infer c (e.g., <c_region, datekey>). Therefore, additional cuboids must be restricted in order to prevent such inferences.

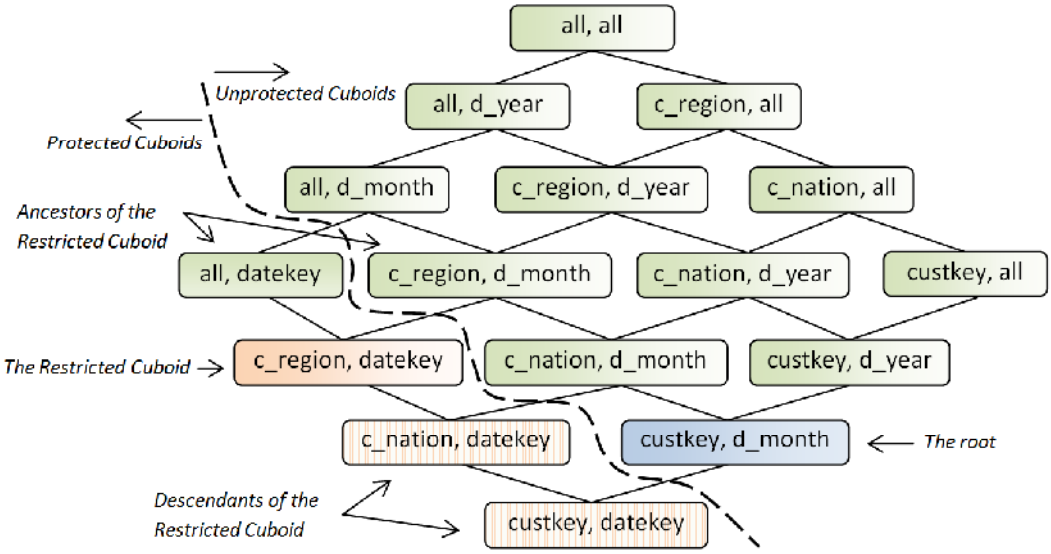


Figure 4.5: An example of preventing Inferences Problem

Specifically, we must identify those that *contribute* to the inference.

To specify these additional cuboids, not all cuboids in the data cube need be considered. Specifically, given the redundancy inherent in a data cube, we can simplify the process by ignoring any cuboid that either carries redundant information or is irrelevant to the inference. In other words, a cuboid can be ignored if it is either not an ancestor to c or an ancestor of other cuboids. The first simplification is due to the assumption that only ancestors cause inferences, while the second is due to the fact that derived cuboids are redundant. Thus, we have the minimal elements in the set of ancestors, namely the basis of the data cube with respect to c, as formalized in Definition 6.

Definition 6. *[119]*
Given a data cube L of n cuboids (c_1, c_2, ..., c_n) and a protected cuboid c, we define a function Basis(L,c) = { c_a : $c_a \in L$, $c \preceq c_a$, where ($c_b \in L$ $c_a \preceq c_b$) implies $c_a = c_b$ }.

For instance, from Example 10, we have: *Basis(L,<c_region, datekey>)* = { <all,

datekey>, <c_region, d_month> }. That is, we only need to consider the two ancestors <all, datekey> and <c_region, d_month>.

Because the basis result includes more than one ancestor, the following conclusion can be made. Inferences are possible only if the *Basis(L,c)* includes more than one cuboid. That is, although L may include many cuboids, those that are in the basis are the only ones that contribute to the inference. For instance, inferences are possible in Example 10 because the basis includes two cuboids. Thus, restricting access to only one of these cuboids will eliminate such inferences (e.g., restricting the access to <all, datekey> or <c_region, d_month> would prevent the inference of <c_region, datekey>).

A question then arises: is the protected cuboid now safe from disclosure? Unfortunately, the answer is NO. Because inference is transitive, the unprotected finer descendants can first be inferred using their ancestors, at which point they can be used to derive the protected one c. Therefore, we should not only prevent inferences to c but also to those that can be used to derive it or, more precisely, to the finer descendant cuboids that can be aggregated to compute c. This would include cuboids <c_nation, datekey> or <custkey, datekey> in our example.

Thus, for each protected cuboid's descendants, the process of restricting ancestors should be repeated. Conversely, to prevent such a situation, a subset of the cuboids that are free of inferences (i.e., an answerable set) can be constructed by growing from an unprotected cuboid or root r. More precisely, we first choose an unprotected cuboid r satisfying $c \preceq r$, and then include all the ancestors of r to form a set. The result must satisfy the requirement that its basis with respect to c includes only one cuboid. Hence, the result is free of inferences to c. Referring to Figure 4.5, the cuboid

<custkey, d_month> is chosen as a root. All its ancestors are then included to form the answerable set (i.e., all cuboids above the dashed line). From any cuboid — other than those in the answerable set — exactly one line extends into the answerable set. For instance, two lines go from the protected cuboid <c_region, datekey>, one to the cuboid <c_region, d_month>, and another for cuboid <all, datekey>, which is not in the answerable set. Similarly, from <c_nation, datekey>, a line extends to <c_nation, d_month> but the other line does not go to the answerable set. This formation indicates the absence of inferences.

Beside the security aspect, we note that the answerable set should also be *maximal*. Maximality is achieved by choosing a root that can be used to build the answerable set and that includes as many cuboids as possible. We refer to such a root as a *minimal* root. For instance, referring again to Figure 4.5, the cuboid <custkey, d_month> is the minimal root, given that it has the maximum number of cuboids above it.

Finally, we note that the correctness of this general approach — in term of its ability to prevent complex inferences — has been formally demonstrated [119]. We reiterate, however, that it has not yet been implemented in a real-world setting and its practical viability is unknown. In fact, a direct implementation is likely to be unusable in practice as the required data set scanning would severely impact query execution time. In other words, each cuboid in the data cube should be inspected in order to determine if it houses protected data. In addition, to find a minimal root, all the unprotected cuboids should be checked to determine the one that best achieves the maximality aspect.

Given its potential, however, we have chosen to build upon this model, and in the next section we will describe algorithms and data structures that can be used to

implement it efficiently. In particular, we address the issue of finding the minimal root to construct the answerable set and the performance concerns associated with the extensive disk accesses that are required in order to make inference assessments.

4.6 Inference Control Module

To ground the research, we propose a framework for the practical implementation of the previous model. The framework consists of several components that communicate with one another in order to prevent inferences in large-scale OLAP systems. The primary component of this architecture is the inference control module. In this section, we discuss this module, while the other components are discussed in the next section.

We note at the outset that the purpose of the inference control module is to assess every query in order to prevent any possible inference attacks. To achieve this, the module employs a series of algorithms in order to find a minimal root and construct an answerable set. The response to the query is then returned to the user only if the module concludes that the response is derived from the answerable set. In the remainder of this section, we present the details of the control module architecture. The structures used by the module are discussed in Section 4.6.1, while procedures for finding a minimal root and assessing the query are discussed in Section 4.6.2 and Section 4.6.3, respectively.

4.6.1 The Hierarchy Manager and the Cuboid Hash Manager Structures

We first present a pair of data structures central to the efficient implementation of the model. We begin with a hierarchical structure that manipulates dimension

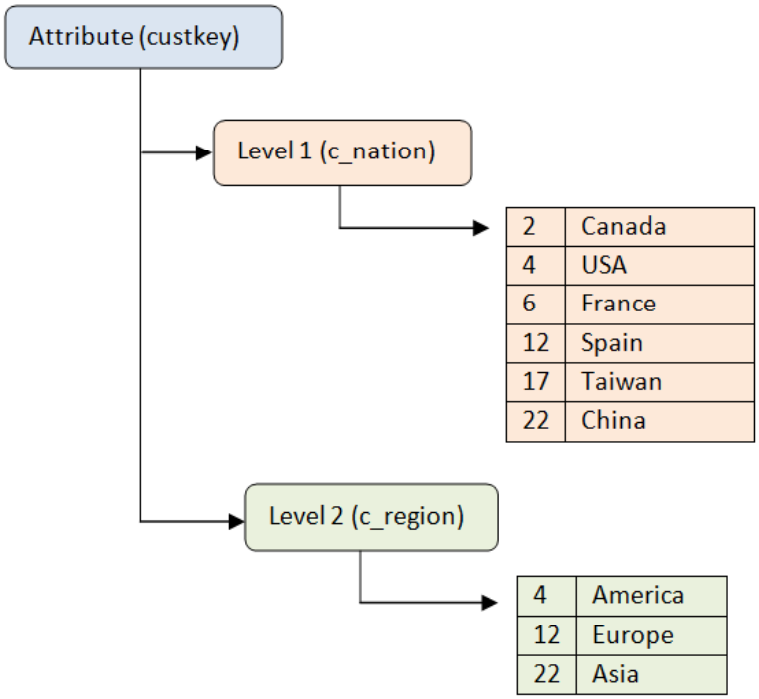


Figure 4.6: The Hierarchy Manager structure for the Customer dimension

hierarchies, and then discuss a look-up structure that facilitates cuboid searching during the root finding process. We refer to the two structures as the Hierarchy Manager and the Cuboid Hash Manager respectively.

The Hierarchy Manager

The Hierarchy Manager has been discussed in Chapter 2. However, we briefly review the relevant concepts here. The Hierarchy Manager is used to manipulate complex cubes at arbitrary granularity levels. It includes meta data that describes the dimension hierarchies in terms of the number of levels, their ordering, and the values corresponding to each level. Figure 4.6 provides an illustration of this structure for the Customer dimension hierarchy of Figure 4.5.

As noted, the Customer dimension has three hierarchy levels ordered as custkey,

which is the base level, followed by c_nation and finally c_region, which is the top level. Each level in turn has different values. Using this meta data, the cuboid can be converted to its numeric representation as a pre-step in the creation of minimal roots. Moreover, it is also used in conjunction with the Cuboid Hash Manager to create a candidate's minimal roots list, as discussed in the next subsection.

The Cuboid Hash Manager

The Cuboid Hash Manager, on the other hand, indexes cuboids by their weights in conjunction with the Hierarchy Manager. The cuboid weight represents the number of cuboids *above* it (e.g., the number of ancestors). The Cuboid Hash Manager is used to create a candidate's minimal roots list if the constructed Answerable Set is not valid (to be discussed shortly). To actually build the Cuboids Hash Manager, the cuboid names are first transformed into numeric format by giving a distinct identifier to each attribute in the cuboid. The given number represents the level number of the attribute in its corresponding dimension hierarchy.

Recall, for example, the Customer dimension from Figure 4.6. This dimension has four hierarchy levels: custkey $\preceq$ c_nation $\preceq$ c_region $\preceq$ all. Each hierarchy level is given a distinct number from the base level to the top level. For instance, the custkey attribute is given the number 1, the c_nation attribute is given the number 2, and so on for the other dimension attributes. So the transformation of the cuboid <c_nation, d_year> is <2,3>, where the number of the levels in Customer and Date are 4. Secondly, the weight of each cuboid is found and stored in the Hash Manager. For instance, referring to the data cube depicted in Figure 4.5, the cuboid <2,3> is of weight 5 (i.e., there are five cuboids above it). Figure 4.7 provides an illustration of the Hash Manager structure for this data cube.

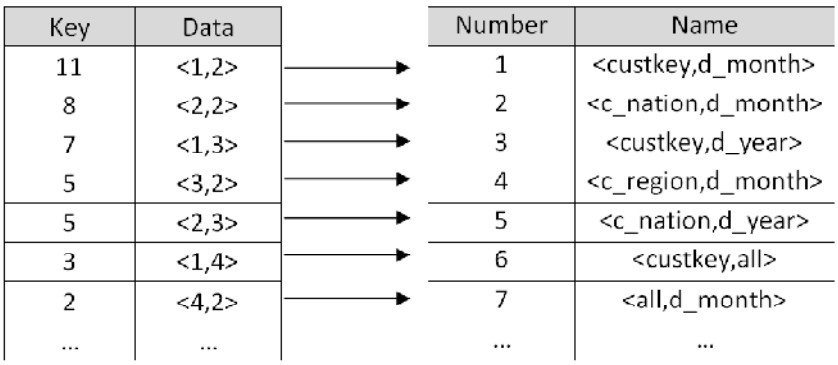


Figure 4.7: The cuboid Hash Manager structure

As noted, the records in the Hash Manager are sorted according to their weights in order to efficiently retrieve these values at run time. Each record in the left structure has two values: the key, which stores the cuboid weight, and the data that stores the cuboid's numeric representation. In turn, entries in the right column contain the cuboid number and its native name. For instance, the cuboid <custkey, d_month> has the numeric representation <1,2> and it is of weight 11.

4.6.2 Finding a Minimal Root and Constructing an Answerable Set

Finding a root is the first step of constructing an Answerable Set. Recall that the purpose of constructing this set is to prevent the protected cuboid c from being inferred. However, due to the fact that derived cuboids are often redundant within the data cube, we know that c can actually be derived from its more granular levels (descendants). This gives users an additional opportunity to compute c at an alternate level of detail and, as a consequence, to bypass restrictions. Therefore, any cuboid that allows one to derive c must be specified as a protected cuboid.

At the same time, the administrator should not be burdened with the responsibility of specifying all these cuboids. For this reason, we define the function **Descendants(.)** that finds all cuboids from which the protected cuboid can be derived. The function definition is formally stated in Definition 7. We refer to this set of cuboids as the Protected Cuboids Set P. Definition 8 then describes the set P.

Definition 7. *For any protected cuboid* c *within a data cube* L*, we define Descendants(L,c) =* $\{ c_a : c_a \in L$, *such that* $c_a \preceq c$ *holds*$\}$.

Definition 8. *For a data cube L and a protected cuboid c,* $P = \{ c \bigcup c_p$*:* $c_p \in$ *Descendants(c)* $\}$.

That is, P is the union of c and all cuboids that can be used to derive it. However, restricting access to cuboids in P does not completely secure c from being inferred. Because inferences are transitive, a user can first infer a cuboid finer descendants, and then continue to derive c. For instance, the base cuboid can be inferred from its unprotected ancestors, and then can be used to compute any protected cuboid within the data cube. Thus, cuboids in P must be restricted not only from direct access, but also from inferences. This should be considered when finding a root and constructing an answerable set.

In addition to that scenario, the answerable set should be maximal with respect to the number of cuboids it includes. This is achieved by choosing a minimal root that has as many cuboids as possible above it. However, it is not practical in most real-world data cubes to check every unprotected cuboid to determine that it satisfies this notion of maximality. In data cubes, the number of cuboids increases exponentially with the number of dimensions/levels. Specifically, we may represent the total number of such cuboids as follows: $\mathrm{T} = \prod_{i=1}^{n} (L_i + 1)$, where n is the number of dimensions in the cube, and L_i is the number of levels associated with dimension i.

Algorithm 4 Constructing a Minimal Root

Input: A protected cuboid c.
Output: A minimal root r.

```
1: Initialize the Hierarchy Manager hM
2: Convert c to its numeric representation cn
3: Find the deepest/lowest attribute c_a among cn
4: for Each attribute a in cn do
5:    if c_a is not equal a then
6:       find the base level a_1 of a
7:       set r_i = a_1
8:    else
9:       find the next level of the deepest attribute c_{a+1} of c_a
10:      set r_i = c_{a+1}
11:   end if
12: end for
13: Return r
```

We therefore describe a new algorithm for the efficient identification of the minimal root. The algorithm — listed in Algorithm 4 — assumes a protected cuboid c as its input and produces a minimal root r as output. The process starts by initializing the Hierarchy Manager hM with meta data that describes the dimension hierarchies. It then converts the restricted cuboid c — in conjunction with the hM — into its numeric representation. This representation supports very fast creation of the root at run time. For instance, for a k-attribute cuboid $< a_1, a_2, \ldots, a_k >$ a numeric representation can be created as $< x_1, x_2, \ldots, x_k >$, where x_i is the number of attributes a_i in the corresponding dimension.

The deepest/lowest attribute amongst the c attributes is found in step 3. The key components of the algorithm then begin at step 4, where the minimal root is constructed by replacing the lowest attribute by the next attribute level in the corresponding dimension, and the remaining attributes by the base attribute level of their corresponding dimensions. The following example illustrates the process.

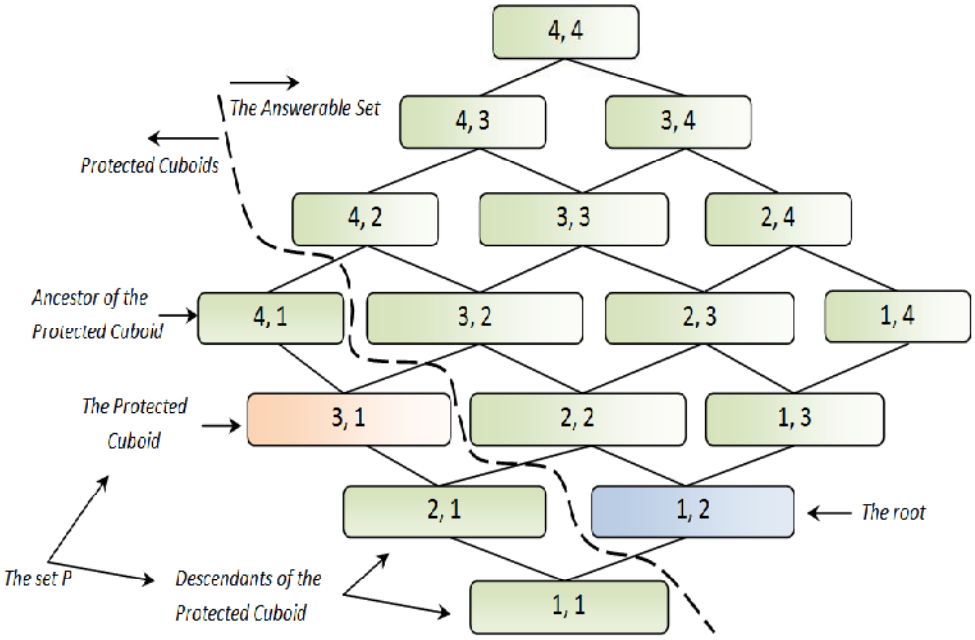


Figure 4.8: An example of constructing a Minimal Root

Example 11. Figure 4.8 shows the dependency lattice of the 2-dimensional data cube depicted in Figure 4.5. Each integer denotes an attribute of the corresponding dimension. For instance, <c_region, datekey> is represented as <3,1>, where the c_region attribute is the 3rd level in the dimension, and datekey is the 1st level in the dimension. Let c = <c_region,datekey>, which is converted into <3,1>. The lowest attribute is defined as datekey (1) and will be replaced by the next attribute level d_month (2). The remaining attributes are then replaced by their base levels (in our example, c_region(3) is replaced by custkey(1)). This process results in a minimal root r <1,2>.

After specifying r, all cuboids above it are included to form the Answerable. The function **Ancestors(.)** is then used to specify these cuboids as formalized in Definition 9.

Definition 9. *A function Ancestors(.) is defined as Ancestors(r) = { c_r: such that $r \preceq c_r$ holds}.*

This set is safe to be queried, while the remaining cuboids are considered as protected and cannot be accessed. In our example, all cuboids above the dashed line form the answerable set. Note that the protected cuboids — not included by the answerable set — either belong to P (see Definition 8), or are ancestors of cuboids in P that may cause inferences if they remain accessible. For instance, referring to Figure 4.8, the cuboids in P include c <3,1>, along with its descendants, and the cuboid <4,1>, which is an ancestor of c.

The Answerable Set is considered to be valid if it contains at most one ancestor for any protected cuboid. That is, any protected cuboid must have only one cuboid above r to prevent inferences. The function **Above()** in Definition 10 formalizes this concept.

Definition 10. *Let A be the answerable set, while P is the set of all protected cuboids of size n. We then have Above({(*c_i*): 1 ≤ i ≤ n}) = { Ancestors(*c_i*) ⋂ A }.*
We say that A is valid if Above({(c): ∃ c ∈ P}) = 1.

For instance, referring to Figure 4.8, no inference is possible from the Answerable Set to any protected cuboid, since Above(.) only includes one cuboid for any protected cuboid.

If the Answerable Set is not valid, a new root should be specified. To do so, the Cuboid Hash Manager is first used to build a *candidate root list* from the cuboids of the Answerable Set. This list consists of candidate cuboids having a weight (e.g., the number of cuboids above it) equal to or smaller than the weight of the rejected root, ordered by their weights. The entries of the list are then checked to find a valid root. In the worst case, we have to check all the entries of the list. If no valid root is

found or the list length is equal to zero, then there are no more candidate roots and the query should be rejected. Algorithm 5 describes this process.

Algorithm 5 Finding the Minimal Roots

Input: A set C of protected cuboids.
Output: The Minimal Roots R.

```
1: Initialize the Hierarchy Manager (hM)
2: Initialize the Cuboids Hash Manager (cM)
3: Minimize the protected cuboid (PC)
4: for Each protected cuboid c_i in PC do
5:    Construct a Candidate Root r_i for c_i
6:    Create an Answerable Set A_i using r_i
7:    if A_i is valid then
8:       add r_i to R
9:    else
10:      Construct a Candidate Root List (CL) using cM
11:      if length of CL equals to 0 then
12:         print "No Root"
13:      else
14:         for Each c_r in CL do
15:            Create an Answerable Set A_r using c_r
16:            if A_r is valid then
17:               add c_r to R
18:               break
19:            end if
20:         end for
21:      end if
22:   end if
23: end for
24: Return R
```

Essentially, Algorithm 5 starts by initializing the Hierarchy Manager and the Cuboids Hash Manager in Steps 1 and 2 respectively. The Hierarchy Manager is used to construct a minimal root, while the Cuboid Hash Manager is used to create a

candidate roots list if the initial constructed minimal root is not valid. The algorithm assumes a set of protected cuboids C as its input and produces a set of minimal root(s) R as output. Multiple roots are possible in practice because sometimes more than one cuboid could be protected. If so, C is examined and minimized to produce the set PC in Step 3. Specifically, a cuboid c_i can be eliminated from C if $c_i \preceq c_j$ exists, where c_j is another protected cuboid in C. For instance, suppose C contains two overlapping protected cuboids <s_nation, d_month> and <s_region, d_year>. The former can be eliminated because it is a descendant of the latter. In other words, inferences to the former will be automatically prevented when finding the latter's root. The minimization process is formalized as the function *Minimize()* in Definition 5.

Definition 11. *For any protected cuboid C, we define a function Minimize(.) as Minimize(C) = { C - c_i, where $c_i \in C$ and there is another protected cuboid $c_j \in C$ satisfying $c_i \preceq c_j$ }.*

At this point, we are finally in a position whereby every thing is set to construct a root r for each cuboid in PC. This is done in Step 5, followed by creating an Answerable Set A corresponding to r in Step 6. The set A is then validated in Step 6 and, if the validation is successful, r is added to the returned roots set R. Otherwise, a Candidate Root List CL is constructed using the Cuboid Hash Manager cM and the entries of this list are then checked to find a valid root. If no valid root is found or the list length is equal to zero, the query is rejected. Otherwise, the valid root is added to R.

Time Complexity

In this section, we discuss the computational and storage requirements of our algorithms. We note that our algorithms can be done off-line. In other words, Roots

and answerable sets can be computed regardless of the queries the user might eventually ask. Therefore, before receiving any query, the algorithms are executed and the results (e.g., roots and answerable sets) are specified. This has great advantage in reducing the on-line complexity, because the most computationally expensive tasks can be shifted to off-line processing.

However, if necessary, the algorithms can also be executed efficiently at run time (based on actual incoming queries). To see this, note that the key function of constructing a minimal root r (as per Algorithm 4) depends only on the number of attributes in the protected cuboid c, which is relatively small. Specifically, finding the base level for each attribute in c is done in O(1), and finding the next level for the deepest attribute in c takes O(log m) using the Hierarchy Manager, where m is the cardinality for the destination level. Therefore, the complexity of Algorithm 4 is O(n + log m), where n is the attribute count of c, which is relatively small for typical data cubes.

On the other hand, Algorithm 5 finds a set of minimal roots for a set of protected cuboids. In terms of minimal construction, cost can be bounded as O(|C|* |S|), where C is the number of protected cuboids and S is the number of the remaining unprotected cuboids. We note, however, that OLAP servers rarely materialize all possible cuboids and, as such, the cost of computation in practice is quite acceptable (as will be shown in the Experimental Results section). Moreover, it is important to understand that data set size has no impact whatsoever on the cost of minimal root finding, since the cost is related to the metadata only. Consequently, the performance does not deteriorate when production scale warehouses are employed.

4.6.3 Checking User Queries

The inference control model discussed in this chapter can be integrated with existing access control mechanisms. In our system, Sidera has been equipped with the access control module discussed in Chapter 3. The module checks every incoming query and denies/modifies any request for accessing protected data. The inference control mechanism is then enforced by rejecting any query that can not be derived from the answerable set. Such an approach brings no additional on-line performance overhead to the OLAP system since a query needs to be checked by the access control mechanisms anyway. This partially explains why the checking times listed in the experimental section are similar to those given in Chapter 3.

Before looking at the experimental results themselves, we walk through a complete example to demonstrate how costs are accumulated in practice. The example considers part of the 4-dimensional data cube depicted in Figure 4.9. We begin by introducing a set of constraints that are specified by the data cube administrator, and then we illustrate the steps that form the checking process itself.

Example

Figure 4.9 depicts part of a lattice for a 4-dimensional data cube based upon Customer, Product, Supplier, and Date dimensions. The Customer dimension consists of the following attributes: custkey, c_nation, c_region. The Product dimension consists of prodkey, p_type, and p_category attributes. The Supplier dimension is made up of suppkey, s_nation, and s_region attributes. Finally the Date dimension has datekey, d_month, and d_year attributes. All cuboids are related by the dependency relation.

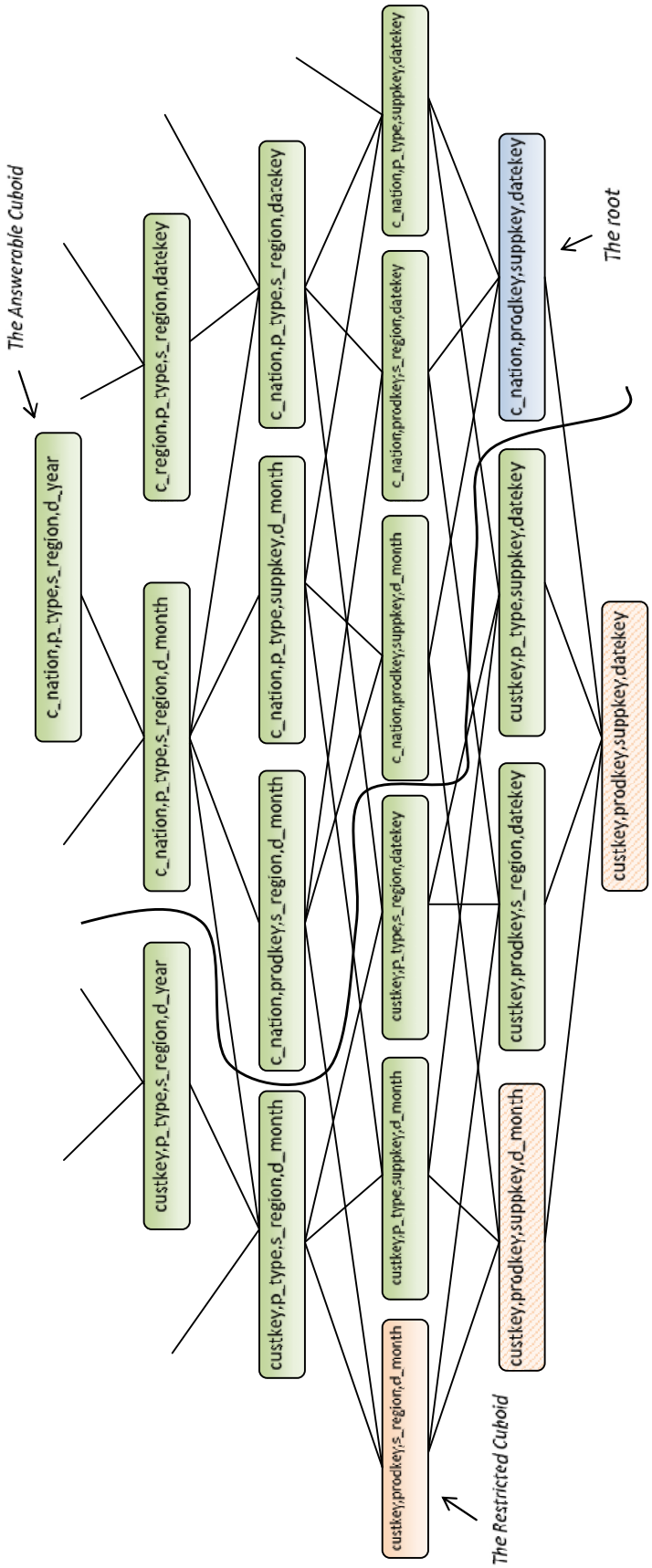


Figure 4.9: Preventing Inferences in 4-dimensional data cube

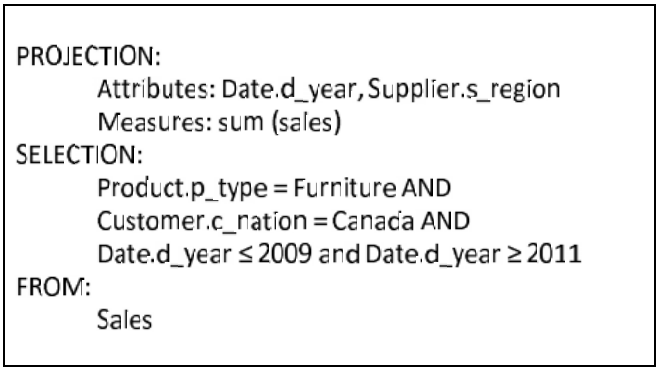


Figure 4.10: An example of checking the user query

Due to privacy concerns, the administrator of the data cube may not want a user to have access to all the data stored in it. In this case, we regulate the access of user Jim as follows. First, Jim should not learn the value of any customer's sales, although he may access the aggregated values of nations or regions. Second, any sales before 2011 should not be accessed by Jim. Now, suppose that Jim issued the query depicted in Figure 4.10 (represented in the IR format) that sums up the total sales for Furniture by customers who live in Canada for years between 2009 and 2011.

As illustrated in Figure 4.9, Jim should not access the three cuboids <custkey, prodkey, suppkey, datekey>, <custkey, prodkey, suppkey, d_month>, and <custkey, prodkey, s_region, d_month>. Moreover, he should not access cells before 2011. Suppose the first constraint is specified by the administrator with a cuboid <custkey, prodkey, s_region, d_month>, indicating that no aggregation finer than those in this cuboid should be accessed by Jim. Clearly, the two cuboids <custkey, prodkey, suppkey, d_month> and <custkey, prodkey, suppkey, datekey> are implied by this constraint in the sense that from either of them the specified cuboid <custkey, prodkey, s_region, d_month> can be computed.

As mentioned earlier, restricting access to these cuboids does not completely secure them. Jim can infer <custkey, prodkey, suppkey, datekey> or <custkey, prodkey, suppkey, d_month>, then continue to derive <custkey, prodkey, s_region, d_month>. Thus, an Answerable Set needs to be computed by growing from a root and including all its ancestors. This can be accomplished by converting the cuboid <custkey, prodkey, s_region, d_month> to its numeric representation, specifying the lowest attribute level, and finding a root as <c_nation, prodkey, suppkey, datekey>. Then all cuboids above the root are from the answerable set (cuboids above the dashed line in the figure).

To determine if the query can disclose protected data, the cuboid that will be used to answer the query (i.e., the Answerable Cuboid) is specified and examined to make sure that it is above the root (not a protected cuboid). For this query, the Answerable Cuboid <c_nation, p_type, s_region, d_year> *is* above the root. Thus, the query is processed to check the next constraint.

The second constraint in the example restricts Jim from accessing sales before 2011. This constraint is different from the first one. The first divides the organization dimension into two parts: employee and {department, branch, all}, based on the dimension hierarchy. The second partitions the time dimension horizontally, based on the attribute values. This, in fact, can be enforced directly by the access control model (see Chapter 3), which identifies the protected cell and anything below it.

Referring back to the query, there are three conditions in the Selection element (p_type = Furniture, c_nation = Canada, and d_year between 2009 and 2011). The last condition filters data for years between 2009 and 2011. However, there is a constraint on year (sales of year 2010 is restricted). According to the rules proposed in Chapter

3, if there is an intersection between a query condition and a constraint and there is no exception, the query should be rejected. Clearly, there is an intersection between them and, as a result, the query will be rejected. Algorithm 6 illustrates the checking process.

Algorithm 6 Checking Query Selection Elements

Input: A set S of constraints and the Query Q.
Output: Returns True if Q is valid, False otherwise.

```
1: Let QC be the query conditions
2: for Each Condition c_i in QC do
3:    for Each Constraint s_j in S do
4:       if c_i ∩ s_j ≠ ∅ then
5:          Return False;
6:       end if
7:    end for
8: end for
9: Return True
```

The time for checking a query is quite small. In the first phase, the time is equal to the time to build the Answerable Cuboid + the time to determine if this cuboid is above any valid root. To build the cuboid, we rely on the memory resident IR that consists of the query's operational elements. Query attributes are simply retrieved directly from the IR structure, which can be done quickly. The latter time (the time to determine if the Answerable Cuboid is above any valid root) is also small, and is equivalent to O(|R|), where R is the number of valid roots. In practice, the number of valid roots is quite small (i.e., at most it equals the number of protected cuboids).

The complexity of the second phase (checking the query conditions) represented in Algorithm 6, is O(|Q|* |S|), where Q is the number of query conditions and S is

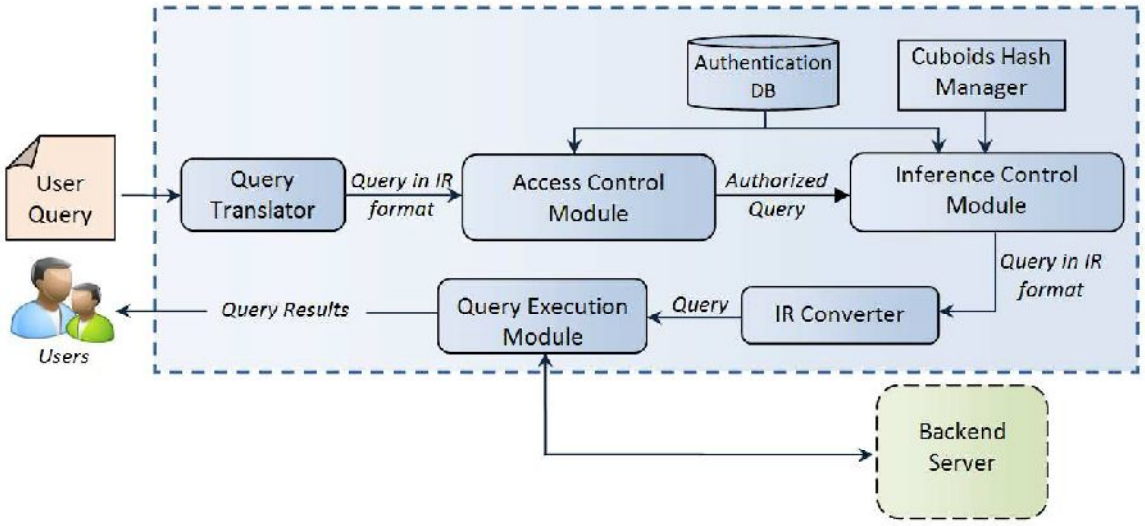


Figure 4.11: The architecture of the security framework

the number of constraints. Both Q and S are small in practice, typically in the range of 4 to 5 based on queries listed in the widely used OLAP benchmarks [8, 92].

4.7 Our Prototype Framework

In this section, we discuss the remaining components of our framework. Figure 4.11 illustrates the basic design. The framework has been incorporated into a DBMS prototype specifically designed for OLAP storage and analysis (Sidera). While this server provides its own query optimizer and storage facilities, it does not provide any form of data access security. This offers us the possibility to define security measures on top of the multidimensional data model, and gives us the opportunity to explore performance and correctness issues that would not be possible without such direct access. Our objective therefore is to enhance the existing DBMS with not only access control, but inference control as well.

Referring to Figure 4.11, the user initiates the process by submitting a query using a standard client query language. In this chapter, we will assume for convenience that this language is SQL, though the general principles would also apply to other languages such as MDX or even Sidera's own native XML language. When the query arrives, the Query Translator compiles the query into an Intermediate Representation (IR) using an OLAP-specific algebra developed for the original Sidera DBMS [110]. The OLAP algebra supports fundamental cube operations and allows for their compact representation. More importantly, the algebraic format easily captures the modifications to the query made during the security checking stages. Further details are given in Section 4.7.1.

In our prototype, we assume that the OLAP system is protected by a basic access control mechanism. In this case, the methods discussed in Chapter 3 are integrated into the Access Control Module so as to restrict access to authorized data only. Upon receiving a query in the IR format, the Access Control Module first decides whether the query is legitimate with respect to the queried data. This can be done by checking the Authentication database, which contains access control rules and policies. Then, if necessary, it modifies/rewrites the query in order to prevent unauthorized data access.

Once the query has been examined and possibly modified, the query should then be assessed for possible inference attacks. The Inference Control Module assumes this responsibility and utilizes a specific set of algorithms and data structures for this purpose as illustrated in the previous section. Assuming that the query is acceptable at this stage (either in its native or modified form) it will if necessary be converted back to the appropriate query format (e.g., SQL, MDX) by the IR Converter module

before execution. Finally, the Query Execution module receives the query in its final form, sends it to the backend query engine for execution and receives the results that will be returned to the end user. The functionality of the IR Converter and Query Execution module is quite straightforward and hence is not discussed further.

4.7.1 Query Translating

The initial SQL query is received by the Query Translator, and then converted to the corresponding IR form using the OLAP-specific algebra. The algebra operators are listed in Section 2.4.2. Further details can be found in [110].

We note that the specification of the algebra is strongly influenced by prior language research in the OLAP domain [97, 47]. That being said, it's utilization within the context of inference control, or even security in general, is to the best of our knowledge unique.

The Query Translator itself can be broken down into three conceptual components:

- **Parser**: The parser decomposes the original SQL query into its core elements, and then maps them to the corresponding Schema Objects.

- **Producer**: The producer creates the IR using the OLAP algebra/grammar as described by the Schema middleware. The generated IR will be used to determine if unauthorized accesses or malicious inferences exist. If the validation process is successful, the IR will be transformed back to the SQL format for execution.

- **Schema Objects**: The Schema Objects serve as the bridge between the Parser and Producer. In other words, they represent parsed IR elements such as Dimensions and Attributes or operations like Selection, Projection, etc.

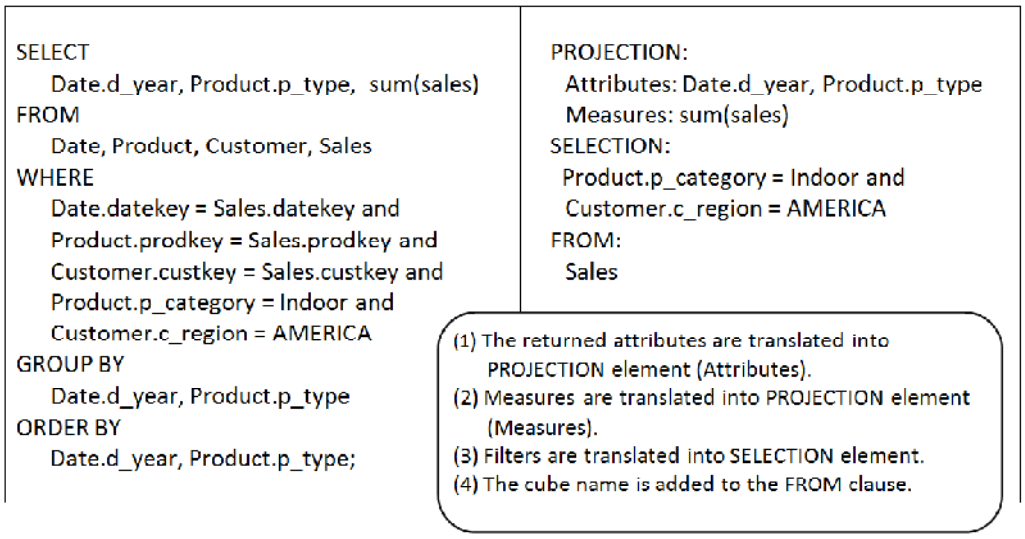


Figure 4.12: A simple query in SQL format and an equivalent query in our OLAP algebra

For example, suppose we have the simple query depicted in Figure 4.12, which generates a list of product types and years with sales totals for category (Indoor) that are sold in America. A more compact, but equivalent query represented in the IR, is illustrated in the right side of the figure.

Note that some of the original query elements are ignored in the translation process and thus will be ignored in the security checking process. These elements are related to GROUP BY clauses, ORDER BY clauses, and JOIN operations defined within the WHERE clause that already exist either in the SELECT or WHERE clauses. We can see, for example, in Figure 4.12 that the the GROUP BY and ORDER BY attributes (e.g., Date.d_year, Product.p_type) are already specified in the SELECT clause, which is itself checked during the checking process. Therefore, rechecking such elements would represent a pointless duplication of resources.

4.8 Experimental Results

In the previous sections, we described components of our framework and their constituent processes in detail. In this section, our focus turns to the practical efficiency of the framework. Specifically, it is important to verify that our approach does not impact query performance in a meaningful way. As noted in the previous chapter, our framework has been incorporated into a DBMS prototype specifically designed for OLAP storage and analysis. While Sidera provides its own query optimizer and storage facilities, it is important to understand that there is no tight coupling between the security subsystem and the backend database. As such, it is possible to plug-in any standards-compliant DBMS server. For test purposes, in fact, this can be an advantage as it provides more intuitive test results and underscores the potential for integration with standard database servers. As such we have also coupled our security model with two common database management systems, a column store DBMS (MonetDB), and a row-based DBMS (Postgre SQL).

In terms of the test environment, all of the experiments were run on a 12-core AMD Opteron server with a frequency of 2100 MHz, L1/L2 cache sizes 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.

The first series of experiments was carried out using the Star Schema Benchmark [92], a data warehousing benchmark derived from TPC-H. The SSB schema consists of a central fact table (i.e., Lineorder) and four dimension tables (i.e., Customer, Part, Supplier, and Date) as illustrated in Figure 4.13. The SSB generator

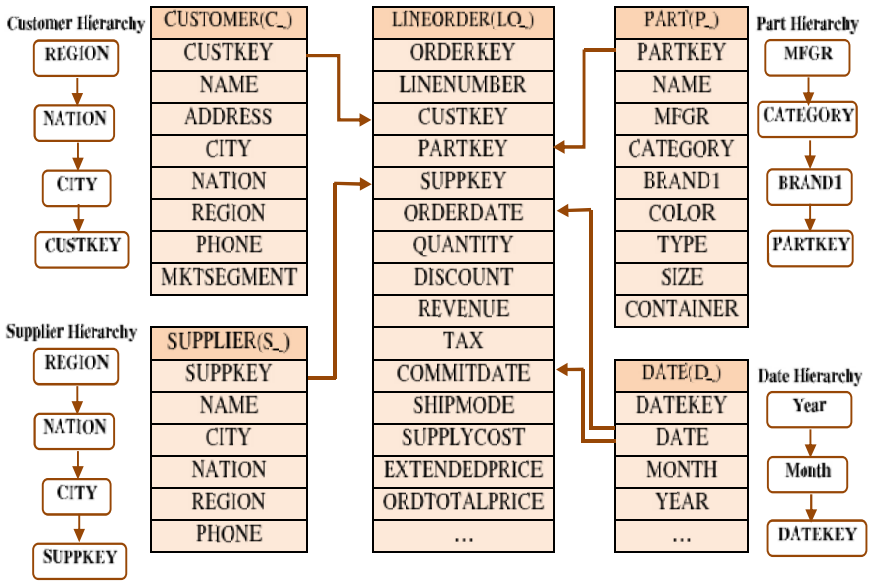


Figure 4.13: The SSB Schema

(with default settings) was used to produce a fact table of 180 million records, in which each dimension houses between 60,000 and one million records. Note that the SSB data set utilized in this chapter is larger than the one used in Chapter 3. We do that to emphasize scalability issues. In other words, we want to show that the checking time is not affected in seriously way by the underlying data set.

The second benchmark employed is the APB-1 benchmark, release I [8]. The APB database generator (with a channel of 40) was used to load two fact tables: Actvars, with approximately 86 million records, and Planvars with approximately 62 million records. The fact tables are joined to four dimension tables: Product, Customer, Time, and Channel, each housing up to 36000 records.

Finally, it is important to mention that these experiments were carried out using five simple but typical authorization constraints. Specifically, we generated one

constraint for each dimension. For example, one of them was on the dimension itself, along with an exception (e.g., the Part is restricted except for the p_brand1 attribute). A second constraint was placed on an attribute value (i.e., Date.d_year $>$ 1997). Two additional constraints were placed on an attribute value with an exception value (e.g., Customer.c_region = AMERICA is restricted except c_nation = CANADA, and Supplier.s_region = ASIA is restricted except s_nation = TAIWAN). The last constraint prohibits access to a cuboid as a whole. Every tested query violates one or more constraints.

4.8.1 Evaluation using the Star Schema Benchmark

In the following two subsections, we examine both the translation time and the performance of the SSB queries. (Note that testing is conducted using the MonetDB backend — we will discuss PostgreSQL performance at the end of the Section). To begin, the SSB provides 13 analytic queries that can be divided into four categories. Each category provides increasingly sophisticated restrictions on the associated dimensions. For example, the first category consists of queries with conditions on only one dimension; the second category has queries with conditions on two dimensions, and so on. However, SSB does not focus on self-joins, sub queries, or compound query forms; therefore, we provided a complementary set of six queries that cover the query forms that are lacking in SSB. These queries are listed in Appendix B, while the SSB queries themselves can be found in Appendix A.

Translation Time

Recall that our framework builds upon an OLAP algebra designed specifically for multi-dimensional models. However, all SSB queries (and complementary queries)

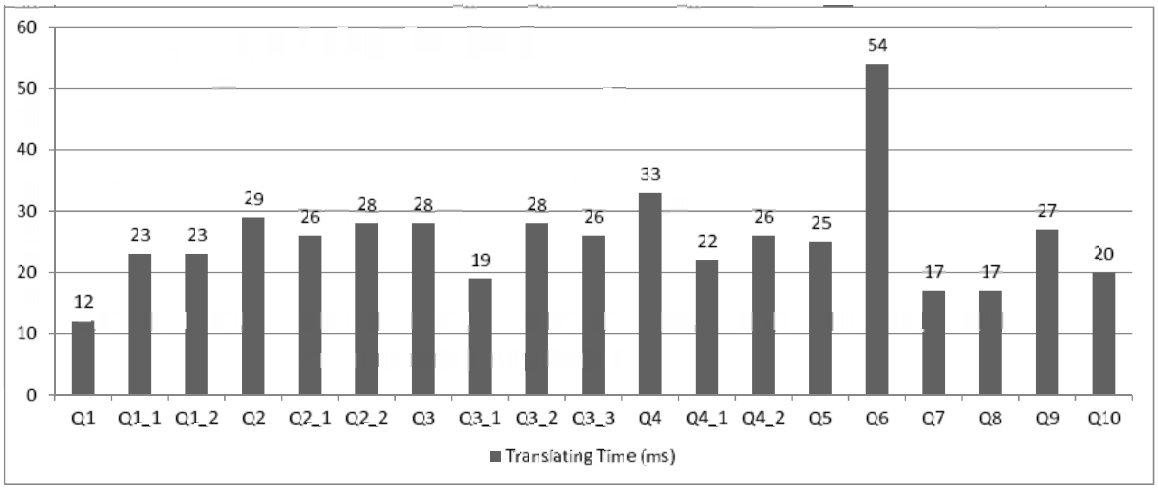


Figure 4.14: Translation time for SSB queries

are written in SQL format and thus need to be translated into our algebra before the checking process can proceed. SQL transformation is performed as per the methods defined in [15]. Figure 4.14 shows the query translation time for the 13 SSB queries (i.e., Query number 1 through 4_2), and of the complementary set queries (i.e., Query number 5 through 10). As should be clear, the translation costs for both query groups are quite small, in the range of 12-54 milliseconds. That being said, at least one of the complementary queries is slightly more expensive to process due to the complexity of the transformation.

Checking Versus Execution

In this section, we will look at the checking times of our framework compared to the final execution times of the SSB queries. We have again divided the queries into two query classes (standard and complementary) and show the performance ratios in Figure 4.15 and Figure 4.16. Note that the query execution time is still listed for queries that violate the policy and were not candidates for re-writing so as to give

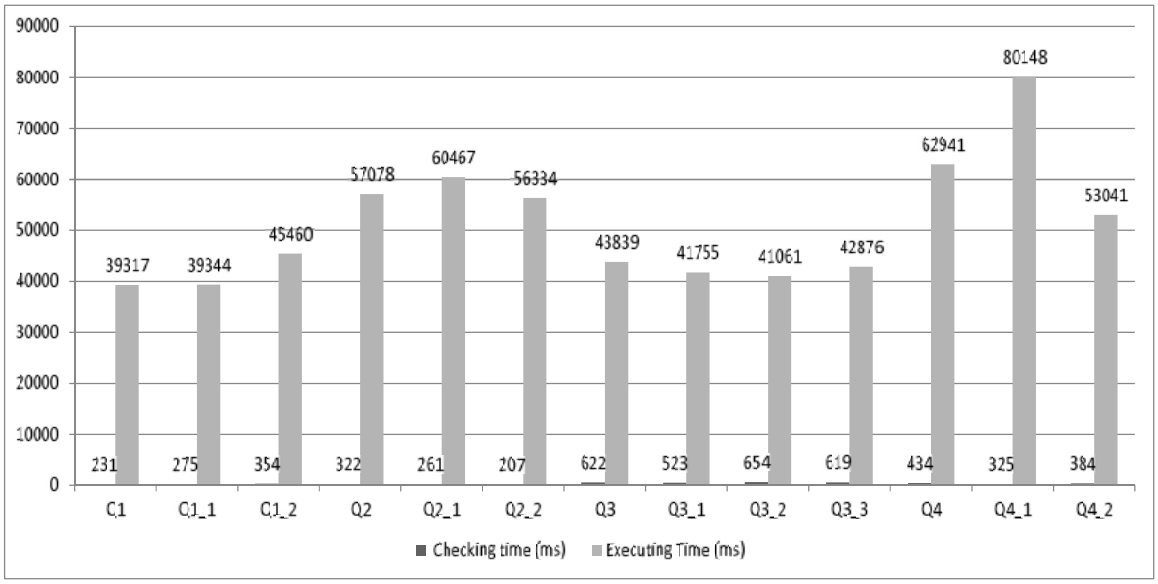


Figure 4.15: Performance for SSB queries

the reader a better sense of the relative balance between checking and execution.

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, execution times ranged from about 39 seconds for Query Class 1 to more than 80 seconds for Query Class 4. As the database gets larger, of course, these times will continue to grow. However, the checking costs are quite modest, in the range of 200-600 milliseconds. Compared to the execution time, these times should be more than acceptable, particularly given that checking costs do not grow with data set size.

Checking using various restrictions number

In this subsection, we present a set of experiments to determine if the checking time would be influenced with the number of violated restrictions. In other words, we want to show that our framework does not effect seriously by the number of defined

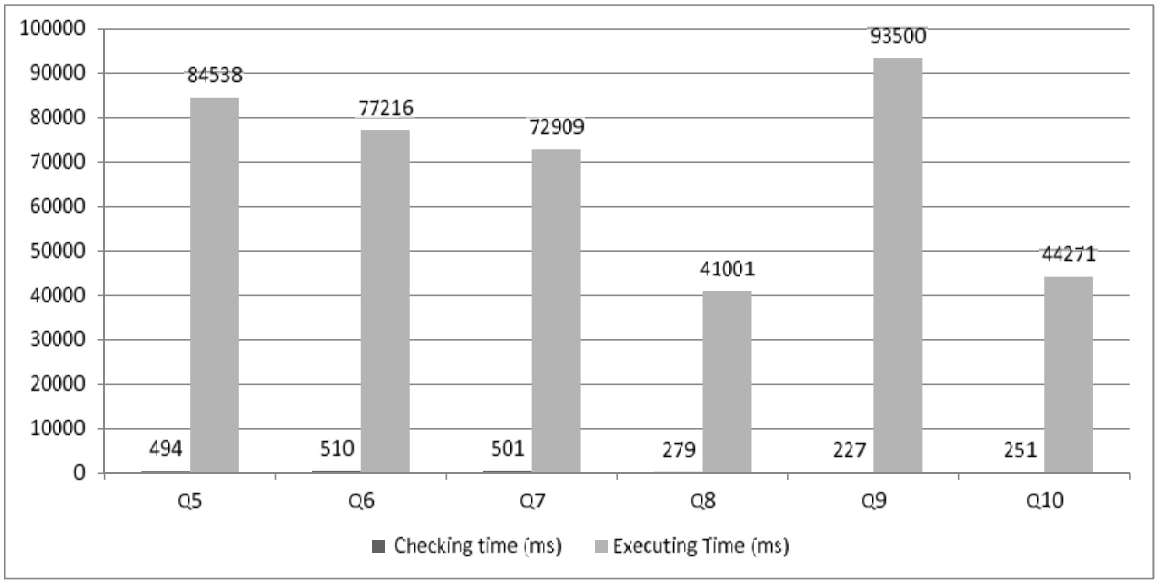


Figure 4.16: Performance for complementary queries

restrictions. For this purpose, we selected queries number 4 through 4_2 and defined restrictions on one dimension, then two dimensions, then three dimensions, and then four dimensions. Figure 4.17 illustrates these times. One must note that the number of restrictions does not affect the checking time significantly as the checking process is dependent on the meta data not on the under laying data set. For instance, in most case the different between the checking time when there are four restrictions to one restriction is less than 50 ms.

4.8.2 Evaluation using the APB-1 Benchmark

In this subsection, we discuss the experimental results using the APB Benchmark. Again, we focus on the translation and execution times. In this case, we consider 10 analytic queries, with each providing sophisticated restrictions on the associated dimensions (i.e., some queries have only one condition, others have up to four conditions). The queries themselves are executed against two fact tables. For example,

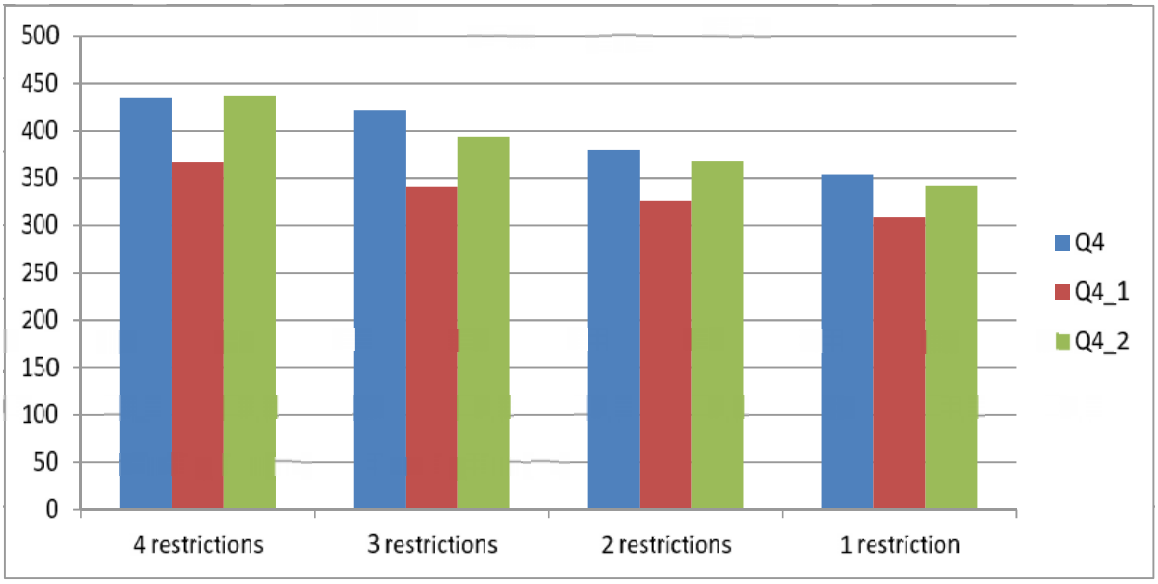


Figure 4.17: Checking times for various restrictions

Figure 4.19 illustrates queries posed against the Actvars table, while Figure 4.20 illustrates queries posed against the Planvars table. The full query set is listed in Appendix D.

Translation Time for APB Benchmark Queries

Again, the APB queries are written in SQL format and need to be translated into our algebra. Figure 4.18 shows the translation time. As was the case with SSB, the translation costs are quite small, in the range of 38-75 milliseconds.

Performance of APB Queries

In terms of the checking results, we have isolated the APB queries into two query classes based on the target fact table. For example, Figure 4.19 shows the ratio of processing cost to query execution time against the Actvar fact table, while Figure 4.20 shows the results for the Planvars fact table. Again, the query execution time is still

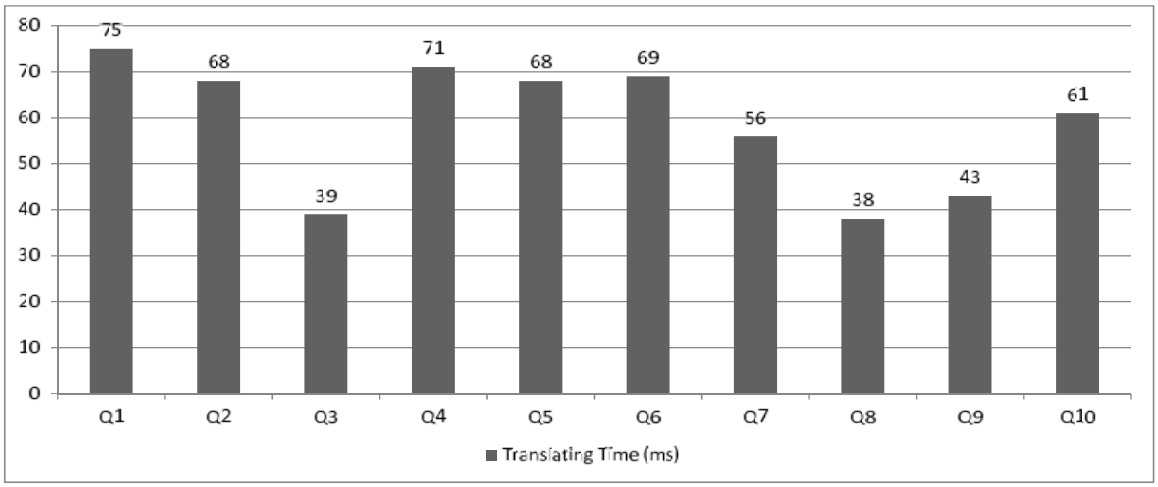


Figure 4.18: Translating time of APB queries

listed for queries that violate the policy and were not candidates for re-writing, in order to give a better sense of the relative balance between checking and execution.

It should be clear that the checking time is again quite small, in the range of 39-600 milliseconds. This implies that little or no I/O is required during the checking phases once the DBMS process has started. This is the case since the Hierarchy Manager can be pre-loaded with the appropriate meta data. For the queries, however, both benchmarks' databases are extremely large and we cannot expect them to be preloaded into memory without enormous hardware resources. In other words, there are strong limits on the amount of optimization that can be performed during the execution phase. We also note that, in terms of this specific test case, the conditions of Query 1 and Query 8 do not violate the policy constraints and thus the checking times for each of them are less than for other queries.

As a final point, we re-iterate that column stores are well suited to the analytics environment. Specifically, column stores tend to be much more I/O efficient for read-only queries since they only have to read from disk (or from memory) those attributes

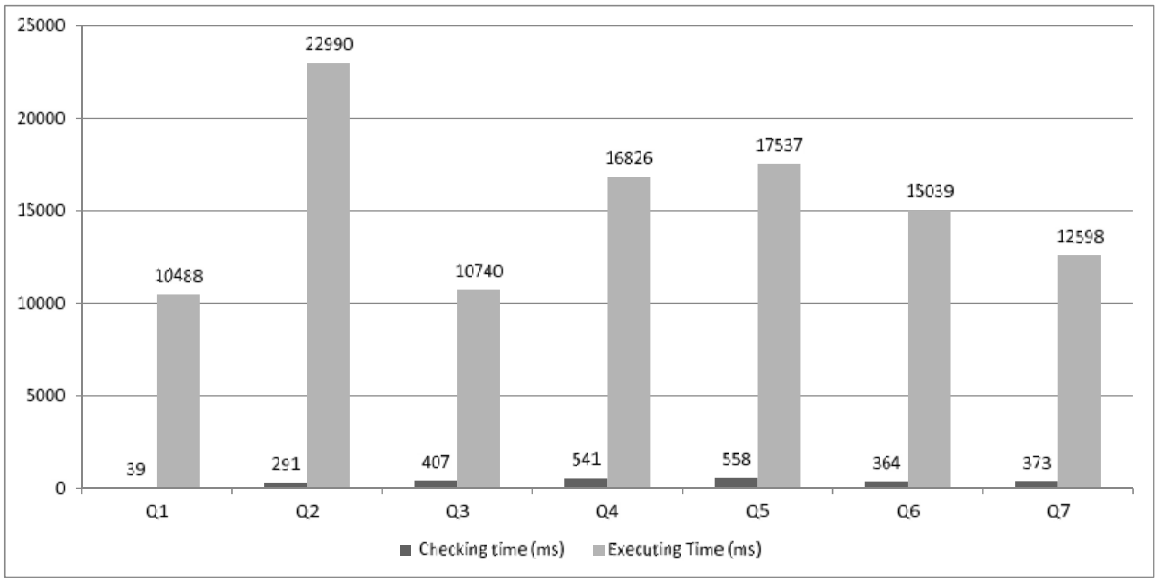


Figure 4.19: Performance for APB queries against Actvars table

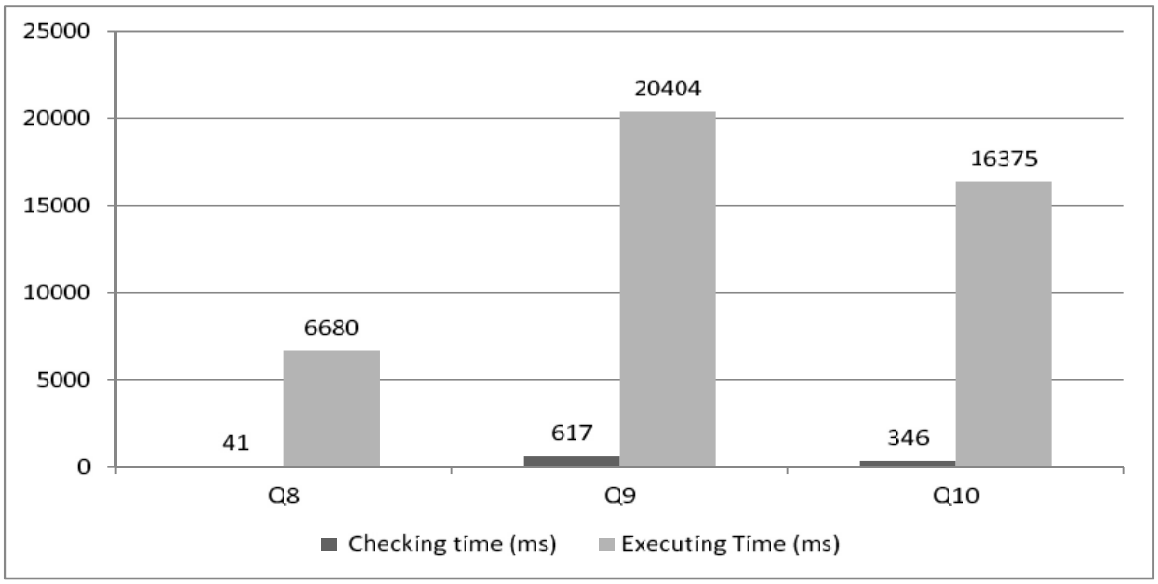


Figure 4.20: Performance for APB queries against Planvars table

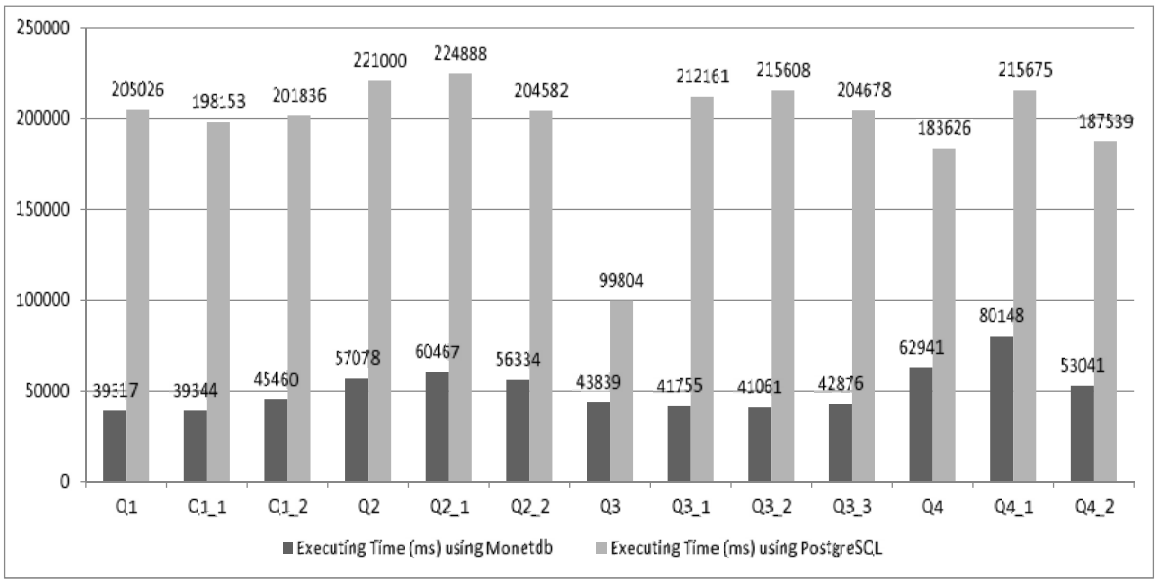


Figure 4.21: Execution time for SSB queries on MonetDB and PostgreSQL

actually accessed by a query. Figure 4.21 and Figure 4.22 illustrate the execution time for both SSB and APB queries using MonetDB and Postgre SQL. Here we can see that the execution times for traditional row store database servers are noticeably larger than those for column store databases. In such environments, the ratio of checking to execution costs would be far more extreme.

4.9 Conclusions

In this chapter, we addressed the issue of inference attacks on protected data. Building on the authorization mechanism discussed in the previous chapter, we presented a general model for controlling malicious inferences in the OLAP domain. Rather than employing a detect-then-remove approach, we exploit recent findings in the literature suggesting that a more robust system can be produced by only allowing queries that are determined up front to be free of inferences. Because no viable implementation of

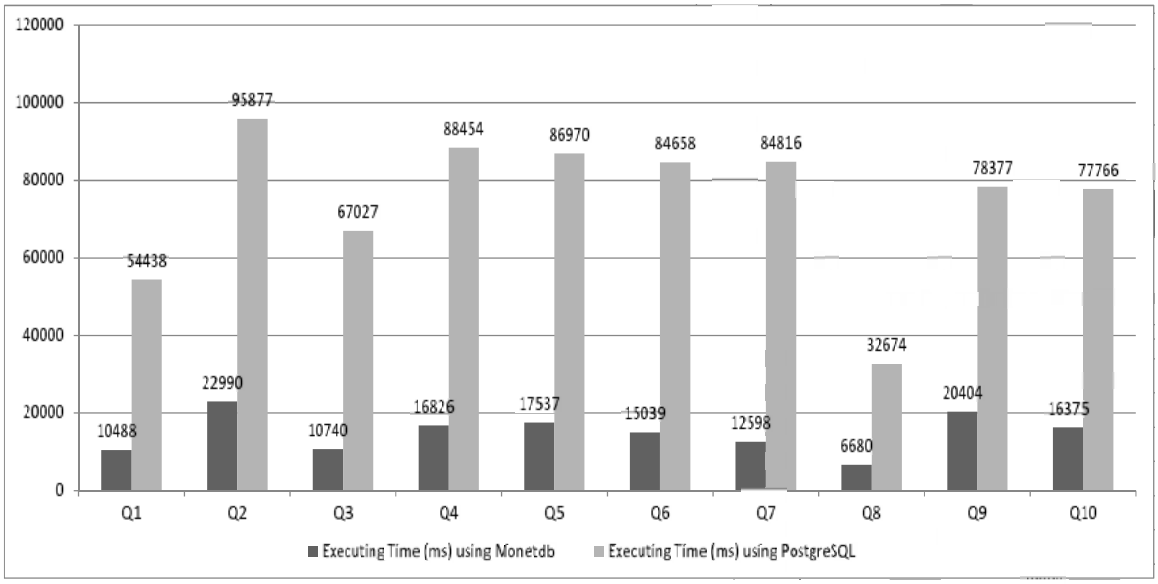


Figure 4.22: Execution time for APB queries on MonetDB and PostgreSQL

the Answerable Set approach has been presented, however, the focus of the current research has been the design of such a framework. To this end, we proposed a series of algorithms and data structures that support efficient real time inference control.

To demonstrate the viability of our approach, we coupled our framework with MonetDB, a popular column-sore database system. MonetDB does not, of course, provide any form of OLAP-centric security, whether access control or inference control. The success of this integration demonstrates that the general principles behind our framework are broadly applicable to any standards compliant DBMS product, regardless of any native understanding of the data cube model. Our analysis of complexity, coupled with experimental analysis, underscores that fact that inference checking can, in fact, be carried out without a meaningful impact upon query execution times (both in column store and row store environments). We believe this achievement represents an important contribution to the literature in the area.

Chapter 5

OSSM: An Object Oriented Security Specification Language for OLAP Systems

5.1 Introduction

One of the most important features of any OLAP system is the protection of data against unauthorized disclosure (secrecy), while at the same time ensuring accessibility by authorized users whenever needed (availability). Considerable effort has been devoted to addressing various aspects of secrecy and availability. Two main objectives are considered in this context. The first is the identification and specification of suitable security policies that establish for each subject which object he/she can access within the system and under which circumstances. The second is the development of a suitable access control mechanism implementing the stated policies so as to ensure that a user accesses only what is authorized and no more.

As a result, a number of researchers have focused on developing more expressive security policies and more powerful access control systems. In fact, many publications

have emphasized supporting access control mechanisms for traditional data managements systems (i.e., relational database systems) [62, 123, 96] and OLAP systems [7, 61, 12, 65]. Security policies, on the other hand, have also been considered in the literature, with three levels of policy specification having been identified [23]: (i) *High-level abstract policies*, which can be business goals, service level agreements, or trust relationships. These policies are not enforceable and their realization involves refining them into one of the other two policy levels. (ii) *Specification-level policies* or business-level policies, which are specified by the system administrator and related to specific objects. (iii) *Low-level policies* or configurations such as security mechanism configurations or device configurations that are related to hardware. In this phase of the research, we focus on the second kind of policies (i.e., Specification policies), and discuss language elements and concepts used to express these policies. We then describe a policy specification model that is based on the Object Oriented paradigm.

Many languages have been designed or extended for expressing specification-level policies (hereafter simply referred to as "policies"). Broadly, these languages are based on XML concepts [89, 125, 115] or logic programming [9, 27, 88]. For instance, the eXtensible Access Control Markup Language (XACML) is one of the most relevant XML-based languages [56, 89]. It is an industry-driven declarative policy language that has been adapted to provide native database support. The purpose of XACML is the expression of policies against objects that are themselves identified in XML. Such XML-based languages are particularly suitable to convey requirements related to authorization and privacy for web-based systems [103]. However, it can be more difficult to adapt them to other environments (i.e., OLAP domains) [126]. Moreover, policies expressed directly in XML are verbose and hard to read and write [10].

Conversely, languages such as ALOPA [88], SecPal [9], and Binder [32] rely on concepts and techniques from formal logic, specifically from logic programming. Logic languages are particularly attractive as policy specification languages. One obvious advantage lies in their clean and unambiguous semantics, suitable for implementation validation, as well as formal policy verification. However, some researchers and practitioners object that logic-based specifications may be complicated or even intimidating to some users [117]. Specifically, security administrators and end users are usually not experts in formal logic and need simple and user-friendly approaches that allow them to easily understand system behavior in order to maintain control over security specifications, and to easily learn the syntax of policy languages.

Moreover, most of these policy languages are generic in nature and would map poorly, if at all, to the OLAP domain. In particular, access privileges in OLAP databases can be intuitively associated with conceptual entities such as dimensions and hierarchical aggregation levels. Mapping these conceptual entities to the physical elements of the DBMS storage layer (i.e., tables, rows, columns) can, however, be a significant technical challenge. Furthermore, the existence of aggregation hierarchies provides opportunities for malicious users to subvert the intended protection layers. For example, one might work around a restriction on the summation of provincial sales totals by "rolling up" municipal sales results instead. Failure to protect all such possibilities would introduce potential vulnerabilities into the system.

In this chapter, we present an OLAP Security Specification Model (OSSM) that formulates comprehensive OLAP-specific policies in an intuitive manner. The formal semantics for the model are based on the concepts of the Object Oriented paradigm. OSSM enables admins or security users of OLAP systems to make use of the concepts

of classes and objects in building their policies. A primary goal of this approach is to model policies in a way that as closely as possible reflects the way humans tend to think about them. Simply put, they can think of policy components (Subjects, Objects and Roles) as abstract objects residing in memory. Each abstract object can be re-used and combined in order to create a complete policy. That is, our approach defines abstract objects without any reference to the physical data; this provides a higher level of abstraction, and a clearer and more flexible design.

To support the evaluation of the OSSM approach, we provide a policy implementation and evaluation framework developed specifically for this environment. The framework hides the low-level policy details and enables administrators to intuitively define policy specifications. Four main components comprise this framework:

- An abstract object oriented API that allows security specialists to design policies using an intuitive compositional paradigm.
- Concrete language integration options permitting both traditional declarative specification and more powerful programmatic applications.
- A policy repository component that stores the policies generated by the user interface.
- A policy manager component, which is a bidirectional component that retrieves policies stored in the policy repository upon requests received from the outside access control module.

Ultimately, to underscore the practical viability of the proposed approach, an OLAP-centric case study is also provided.

The remainder of this chapter is organized as follows. In Section 5.2, we present an overview of related work. Section 5.3 describes the conceptual OLAP data model that grounds the language's design, as well as the basic terminology relevant to policy specification. Our objectives and methodology are discussed in Section 5.4. The OSSM and its associated classes and methods are presented in detail in Sections 5.5, 5.6, and 5.7. A discussion of the integration with the policy engine is described in Section 5.8. Managing Overlapping Roles is then disccused in Section 5.8.3. A case study is presented in Section 5.10. Final conclusions are offered in Section 5.11.

5.2 Related Work

The need for security languages to support the specification of access control policies has long been identified in the literature. During the early stages, the primary focus was on network policy specification. Multiple approaches have been proposed that range from formal policy languages, to rule-based policy notation using an if-then-else format, to the representation of policies as entries in a table consisting of multiple attributes [49, 59, 24]. The most notable work in this area is the Internet Engineering Task Force (IETF) policy model, in which the authors introduce a policy language based on the notion of a path [59]. A key objective of this language is to ensure that all attributes associated with the policy — including the service type of the traffic, conditions used to trigger the policy, and the actions executed when the policy is triggered — are all bound to a predefined path.

Policy specification languages within distributed environments have also been investigated. A variety of security languages have been proposed [75, 76, 10, 91, 115]. Ponder2, for example, is a language for specifying security and management policies

for distributed systems. It utilizes XML as the specification mechanism and specifies policies in a subject-action-target (SAT) format [115]. SecPAL credentials, on the other hand, are expressed using predicates defined by logical clauses, in the style of constraint logic programming [10], while the RDBAC framework utilizes Transaction Datalog to provide reflective access control in which privileges are expressed as database queries [91]. Of course, these approaches have great value when used on large-scale networks and distributed systems. However, they do not scale well within OLAP domains since they are often fragmented, dependent on infrastructure, and do not take into account the multidimensional conceptual data model.

The Unified Modeling Language (UML) is also used to formulate security policies. For example, in [5, 6], Alam et al proposed a security language called SECTET-PL, and show how to express trust policies by using predicate expressions whose grammar is expressed in UML. In [52], the authors presented a Trust Management Framework that supports policy life cycle management using UML diagrams. However, none of these works are designed specifically for the OLAP domain. Instead, they target the policy model design itself.

Recently, policy specification in web-based applications has been proposed [111, 14, 20, 55]. SELinks, for instance, targets web apps and provides a uniform programming model (in the style of LINQ and Ruby on Rails), with language syntax for accessing objects residing either in the database or at the server [20]. Still other frameworks investigate the association of security policies with client side code, with protection provided by the interception and analysis of database queries [45]. XACML [89], initially defined by the Organization for the Advancement of Structured Information Standards (OASIS), is a declarative access control policy language that expresses

policies in an XML specification for information control over the Internet. For the most part, however, all such policy languages and/or extensions are generic in nature and would map poorly, if at all, to the OLAP domain.

Modifications to SQL have occasionally been discussed as well, including extensions to the SELECT and REVOKE statements in order to define a purpose driven authorization model [116]. SQL to XML transformations have also been studied. It is possible, for example, to employ control expression languages to map policies from relational environments to those represented in XML [71]. In this context, the relational database is first published as an XML structure (i.e., XML tree), then the policies defined over the relational database are formulated over XML trees. However, this model has not been implemented and it's efficiency is unknown even for small databases, also it is not applicable for multidimensional modeling where millions of records may be materialized. In terms of commercial systems, implementations generally provide very basic privilege mechanisms that are directly associated with relational tables; the concepts of the multidimensional model are rarely considered.

Finally, we note that while language extensions to specifically support OLAP have not been addressed, a number of researchers have investigated more general design issues for the data warehousing context, including both early requirement targets such as agents, decisional goals, and quality goals [66], as well as late stage conceptual-to-logical model mappings for authorization and auditing purposes [106]. A survey of objectives, features, and limitations of warehouse security modeling was provided in [105]. Still, we re-iterate that design methodologies typically assume the existence of conventional security languages and policy engines upon which the design model would eventually be implemented.

5.3 Preliminaries

Before discussing the semantics and syntax of the OSSM, we provide an introduction to the conceptual data model upon which the language is based and give a brief overview of the basic terminology and structures relevant to policy specification in general.

5.3.1 Subjects, Objects and Roles

As mentioned in Chapter 2, we consider multidimensional environments that consist of one or more data cubes. Each cube is composed of a set of dimensions and one or more measures of interest. A dimension can contain a hierarchy that enables the data associated with dimension values to be aggregated at various levels. Figure 5.1 provides an illustration of a very simple three dimensional cube on Store, Time and Product. Store, for instance, is organized in Country $\rightarrow$ Province $\rightarrow$ City $\rightarrow$ Store_number levels.

Security considerations in the data cube context range from simple authentication to the complex data authorization that provides protection for sensitive data. Such security considerations can be achieved by using *policies*. Policies determine which subjects or users have access to a specific data object. For example, the policy could establish that a user is restricted from accessing a specific level of aggregation within a dimension but not the more coarse levels.

Supporting policies are typically based upon a combination of three basic components. The first component, *Subjects*, represents users to which authorizations are granted. A Subject can be a single user or a group of users within the system. *Objects* on the other hand, refer to the data to be protected. An object can be any partition

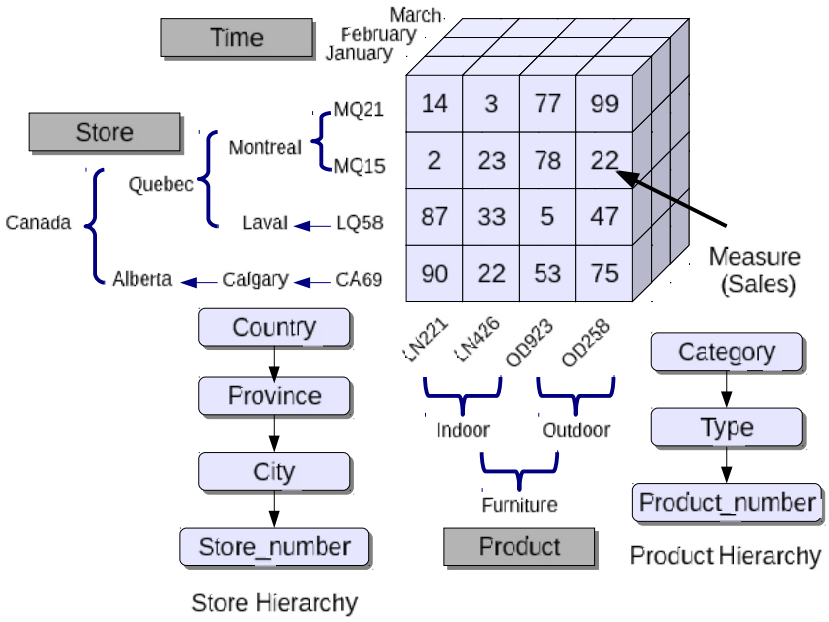


Figure 5.1: A simple three dimensional data cube

of a data cube defined along one or more dimensions in order to give an additional opportunity for finer authorization. Finally, *Roles* are named collections of privileges and represent organizational agents intended to perform certain job functions within an organization.

For example, let us assume that we have an organization in which roles are created based on the job functions of users or subjects. Constraints are subsequently assigned to specific roles based on the requirements of these jobs. Subjects in turn are then assigned appropriate roles based on their qualification. A Subject can be authorized to play several roles, and a role may be assigned to multiple subjects. Figure 5.2 shows the basic relationship between subjects, roles and constraints. As illustrated, a subject may be mapped to one or more roles, and each role may have different constraints with respect to the access of a specific data item.

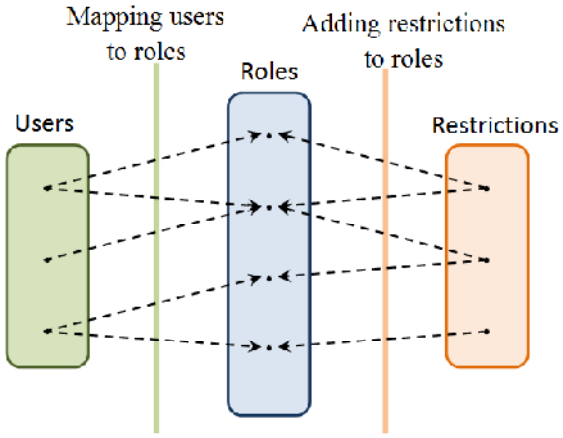


Figure 5.2: Subjects, Roles, and Constraints relationship

5.3.2 Role Hierarchy

Because roles within an organization typically have overlapping permissions, roles themselves may be organized into a hierarchy which, in turn, defines a partial ordering, denoted as $\preceq$. Such hierarchies support a more expressive representation of their semantics. More formally, we can say that given a role domain R, let r_i, $r_j \in R$ be individual roles. If r_i precedes r_j in the hierarchy ordering ($r_i \preceq r_j$), we say that r_i is partially ordered relative to r_j and, furthermore, that r_i is a child of r_j, and r_j is a parent of r_i. This implies that r_i inherits all constraints that are assigned to r_j, and that all users who are mapped to r_i are affected by the r_j constraints. This is formally expressed in Definition 12.

Definition 12. *A role r_i in a role hierarchy R inherits all constraints of roles $L = (r_j, \ldots, r_z)$, where $r_i \preceq r_j$ and $r_j \preceq r_x \preceq r_z$ for a role $r_x \in R$. We say that r_i inherits all constraints of roles reachable from r_i to the Root role of R.*

An example of a role hierarchy is illustrated in Figure 5.3, where any role inherits all constraints that are assigned to its parents up to the Root role. For instance,

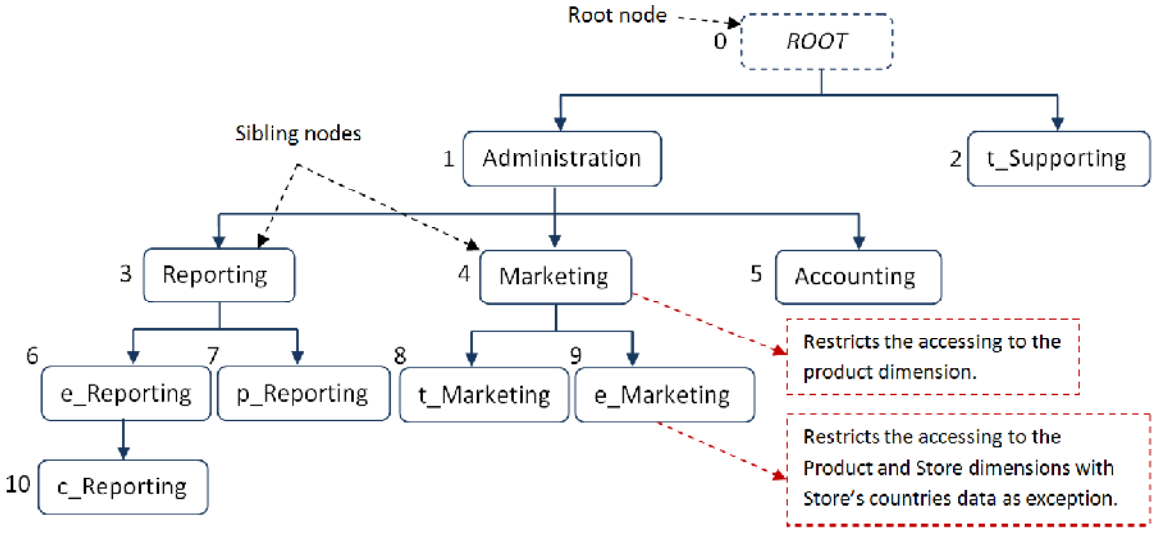


Figure 5.3: Role hierarchy

suppose that a Store dimension with four attributes (e.g., Country, Province, City, and Store_Number) should not be accessed by users of the Marketing role. Consequently, any user who is assigned to the Marketing Role or any of its children is restricted from accessing the Store dimension and, by extension, is also restricted from accessing all the attributes of the specified dimension.

5.3.3 Object Oriented Concepts

The Object Oriented paradigm is a design philosophy based on the concepts of classes and attributes that are often used for more accurately modeling real world objects [67]. Such objects share two fundamental characteristics: *states* and *behaviors*. States describe object characteristics, while behaviors define what operations an object can perform.

Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. An object stores its state in fields/members and exposes

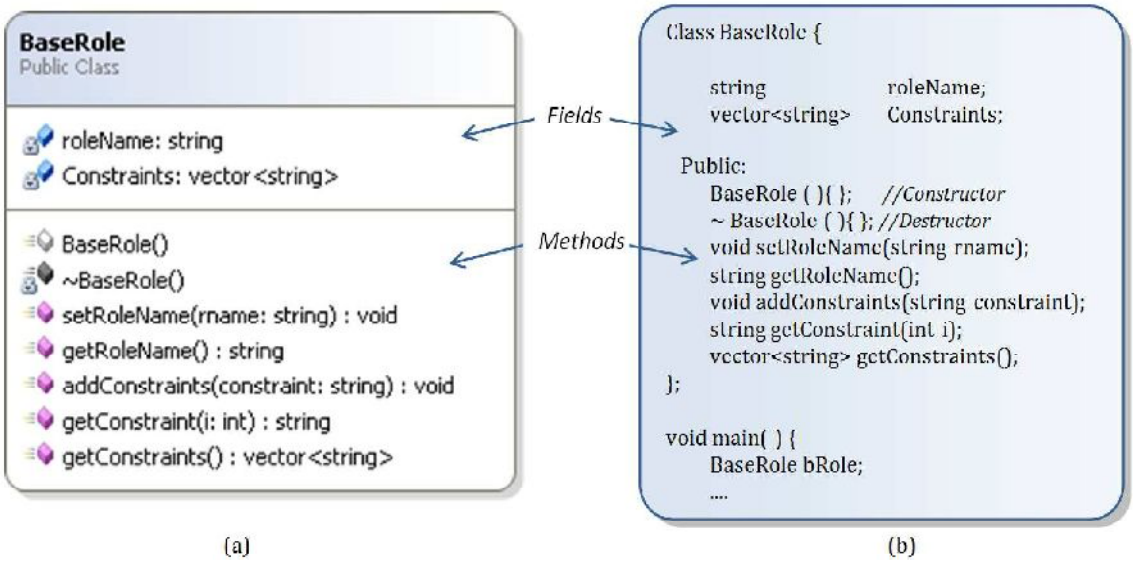


Figure 5.4: (a)A Role Class in UML notations (b)A Role Class in the C++ language

its behavior through methods/functions. Both fields and methods are encapsulated in a class, which provides the structure for objects. For example, with reference to the role hierarchy depicted in Figure 5.3, all roles have common characteristics, including a role name and a set of constraints that are manipulated by a set of actions (e.g., create_role, set_name, add_constraints, etc.). Figure 5.4(a) and 5.4(b) show the Role class declaration in the UML class notation, and in the C++ language, respectively.

The BaseRole class describes the details of a role object. It is composed of three components: the name of the class, a list of fields, and a list of methods. The main() function depicted in Figure 5.4(b) simply illustrates how objects are created. Specifically, an object is an *instance* of a class. We say that the bRole object has been created out of the BaseRole class.

The Object Oriented paradigm offers numerous opportunities to improve and simplify the design of applications. Besides the encapsulation concept, which implies

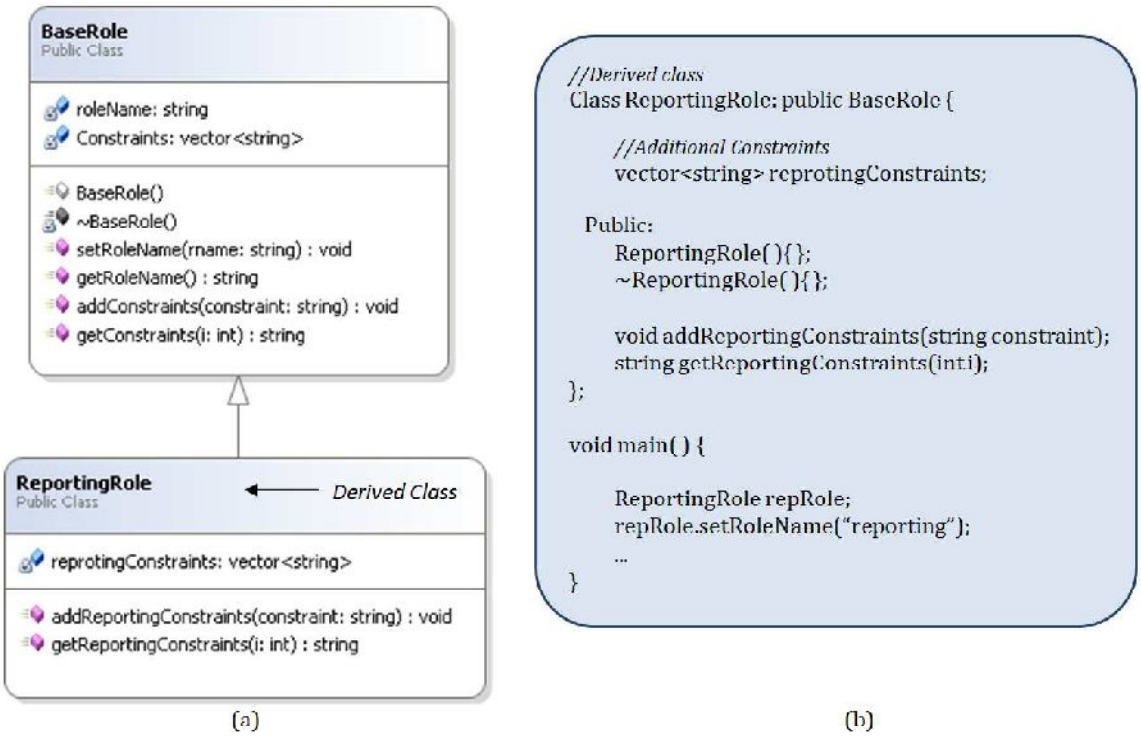


Figure 5.5: (a) Inheritance in UML notations (b) Inheritance in the C++ language

that access to fields in an object can only be performed via a specified method, *inheritance* and *polymorphism* are also provided [85]. As we will see, both can be effectively exploited in policy specification settings. Simply put, inheritance allows one to define a class in terms of another class. The existing class is called the *base* class, and the new class is referred to as the *derived* class. For example, Figure 5.5 illustrates a simple inheritance relationship, where the relationship between the base class (e.g., BaseRole) and the derived class (e.g., ReportingRole) are represented in (a) UML notation and (b) the syntax for creating the derived class in the C++ language. Here, rights may be inherited across the defined roles.

Polymorphism on the other hand, allows one to exploit method overloading [104]. Method overloading is the ability to define several methods for a given class, each

with the same name but different parameters. In the policy context, this gives an administrator the ability to instantiate a policy — or any of its components — multiple times, but with new parameters.

5.4 Objectives and Methodology

5.4.1 Objectives

Most of the proposed policy languages and APIs presented in the literature are generic in nature and would map poorly, if at all, to the OLAP domain. Therefore, the following objectives have been identified with respect to the design and development of a reliable policy model suitable for the OLAP domain.

- Define a high-level design model based on the well-established typing and inheritance features of object-oriented languages. Such a system should allow policies to be mapped directly to the conceptual data cube without any reference to the underlying relational data or physical configurations.
- Model policies without requiring modifications to the existing access control mechanism.
- Verify policy semantics to ensure that statements are mutually consistent.
- In cases where the policies are valid, provide a repository to store policy requirements.
- Provide a simple mapping of the OO design model to the "flat" declarative syntax of the SQL language. Doing so would not only allow security administrators to work in a familiar setting, but would enable simple updating and reporting with standard SQL client software.

- Provide a native language(s) API that directly exposes the features of the object oriented design model. These client side libraries would be particularly well suited to the development of graphical design tools.

5.4.2 The Methodology

Policy specification requires a formal basis (i.e., a suitable programming language or API with clear syntax) that allows administrators or security personnel to specify and manage appropriate security policies. For this purpose, we propose a syntactically clear policy specification model (i.e., OSSM) that borrows from the feature set of the object-oriented paradigm. OSSM relies on the concepts of classes and objects to create instances of various policy constructs such as Subject, Role, and protected objects. These instances/objects are then combined together to create policies.

The object orientation provides us with the means to cope with expressivity, extensibility, flexibility, and reusability. Expressivity is provided via a direct mapping to the conceptual elements of the OLAP domain, rather than to the low-level storage components of the physical layer. Extensibility is achieved by allowing policies to be added or modified to meet the requirements associated with existing or future users. Flexibility is provided since unique policy instances can be created to cater to special security requirements. Finally, reusability is achieved by allowing designers to instantiate policies — or any of their components — multiple times with new parameters.

In keeping with our focus on grounding our conceptual work, OSSM is then folded into a prototype implementation that natively understands the OLAP model. The prototype is depicted in Figure 5.6 and includes three main components: the User Interface Tool, the Policy Repository, and the Policy Manager. The User Interface

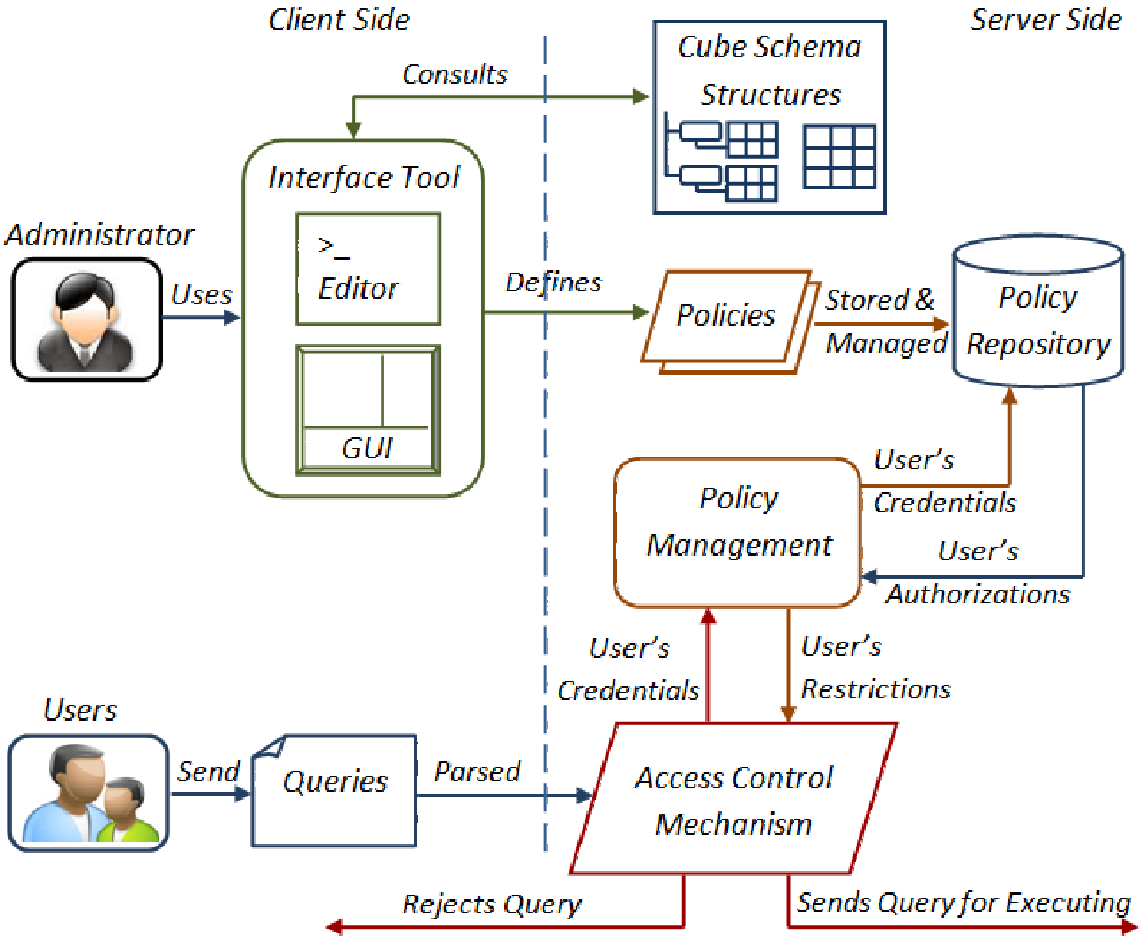


Figure 5.6: The Policy Engine components

Tool permits the administrator to define policies either programmatically or visually in a declarative manner (we note that in this research project we do actually design or provide such graphical software). The policies are then validated syntactically and semantically, and stored in the Policy Repository. In turn, the Policy Manager retrieves policies from the policy repository and sends them to the access control mechanism for enforcement.

The core principles of our approach are two fold. Firstly, we describe the object oriented representation of security elements. This creates an intuitive picture for

designers in the same way that the data cube provides an intuitive picture for OLAP users (even if the data is actually stored in flat tables). Secondly, we avoid replacing the existing access control mechanisms, and at the same time hide the low-level policy details. This assists human administrators in modeling policies in a way that as closely as possible reflects the way humans tend to think about them, taking into account the nature of the underlying multidimensional OLAP model.

Finally, in order to illustrate the simplicity of our approach and to demonstrate how such specifications might be defined using OSSM, we provide a small but representative policy case study.

5.5 The OLAP Security Specification Model (OSSM)

As noted in Section 5.2, researchers have proposed a number of general authorization policy languages, as well as methods to support more fine grained, and hence more flexible, security models. That being said, such approaches are by design "lowest common denominator" solutions. In other words, none assume any specific underlying conceptual model. While this assures a wide range of application options in theory, it also implies that domain-specific implementations would require extensive development, with each such project being both expensive and incompatible with similar systems. Given the multi-billion dollar scale of the OLAP/analytics industry, we believe there is ample motivation for the design of a security specification and execution environment that supports the development of intuitive policy schemas mapping directly to the conceptual model understood by both end users and administrators.

In this section, we describe a security model (OSSM) that is based on the Object Oriented paradigm. OSSM is used for specifying and managing security policies to

cover the wide range of security requirements implied by the OLAP domain. We consider the concepts or the basic components of the security policies (e.g., Subjects, Objects and Roles) as objects, and define a set of classes to represent them. Each class focuses on a specific concept and defines all functionality related to that concept. This includes all aspects of object management, such as an object's properties, operations and necessary data structures. That is, classes encapsulate both the properties of objects and the operations that can be carried out on those objects. Therefore, the defined objects can be re-used and combined with each other in order to represent more sophisticated policies.

The core OSSM classes are the Subject Class (SC), Object Class (OC), Role Class (RC), and Policy Class (PC) and, collectively, they represent the basic elements of our object oriented security model. Specifically, SC defines the objects that provide properties of the system's users, OC defines the objects to be protected, RC defines objects based on the job functions of users and, finally, the PC is created from these SC and RC classes, based on the requirements of the security policy.

Operations are also defined within classes in order to modify object characteristics. The most basic object actions are those that allow objects to be created and destroyed. Other common actions include updating objects, assigning or revoking an existing subject to/from one or more existing roles, or obtaining information about these objects. These actions and others are defined as methods in the corresponding classes. A detailed description of these methods and their corresponding classes, along with a series of examples, are presented in the next sections.

5.6 OSSM Classes

Before providing the fundamental structure of the OSSM classes, there are several points worth noting. First, the *open world* policy is adopted, whereby Restrictions specify denials for an access. Specifically, access is denied if there exists a negative authorization/restriction for it, and allowed otherwise. 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.

Second, Restrictions and Exceptions are defined relative to the OLAP conceptual model. In other words, specifications refer explicitly to cube elements such as dimension and hierarchies. In addition, Restrictions are implicitly read-only, as updates are expected to be performed by administrator-defined ETL (Extract-Transform-Load) processes. Consequently, the language does not expose read-write access options.

Third, Restrictions and Exceptions are grouped together or encapsulated into Roles, which may be organized hierarchically in a tree-based schema as shown in Figure 5.3. Child Roles inherit the properties of the parent. However, additional restrictions in the Child produce more limited access privileges. Note that single inheritance (e.g., in the Java style) is utilized.

Finally, Users may be associated with multiple Roles. In such cases, cube access rights are represented as the union of individual Role specifications. In particular, a user is never denied access to existing privileges by virtue of their concurrent membership in a more limited Role. More details are given in Section 5.8. In the remainder of this section, we list and describe the OSSM classes.

5.6.1 The Subject Class (SC)

The Subject Class (SC) provides a template, or blueprint, to define the properties and the operations common to all of the system's users. These properties and operations are represented within a class as fields and methods, and are defined as a tuple (A, M), where A is a set of fields that characterizes the class objects — as formally stated in Definition 13 — and M is a set of methods allowed on these objects.

Definition 13. *Let A be a set of attributes shared by a set of subjects. A is represented by class fields and defined as the union of the subject's identifier SI and the subject's attributes SA, where SA is a n-tuple of fields (a_1, a_2, ..., a_n) that characterizes the subject, and SI is an identifier that explicitly defines every subject of that particular class for it's entire life. We say that $A = SI \cup SA$.*

Assume now the following simple example, which will be used throughout this section to illustrate the concepts and the notions of the classes.

Example 12. Suppose an analyst Bob is invited to study the data cube depicted in Figure 5.1, given that the following two restrictions should be satisfied. First, due to privacy concerns, the sales totals of all cities in the province of Quebec should not be accessed by Bob except for the sales of the city of Montreal. Second, assume that any sale done before 2005 should not be used for the analysis.

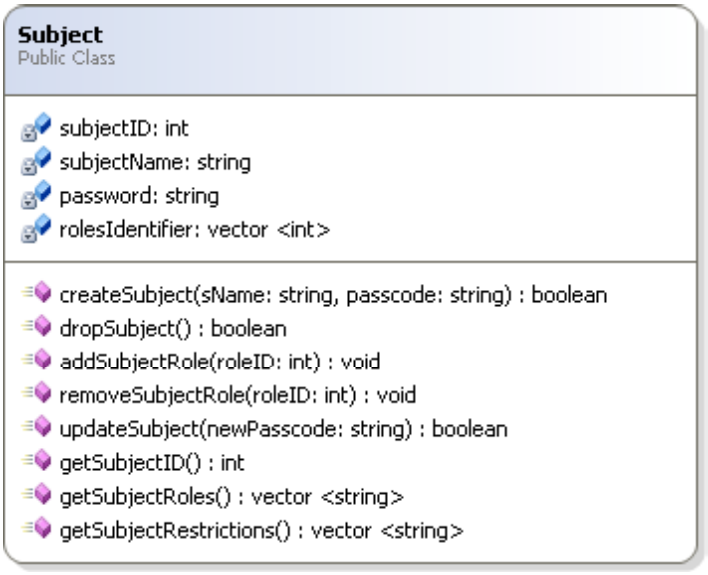
By applying the definition of SC to Example 12, we will have a Subject (e.g., *Bob*) with the following elements:

- A = A subject identifier $\cup$ a set of attributes that describes the subject (e.g., subject name, password, and role identifiers).
- M = A set of methods (e.g., create subject, update subject, drop subject, assign subject-role, get subject info).

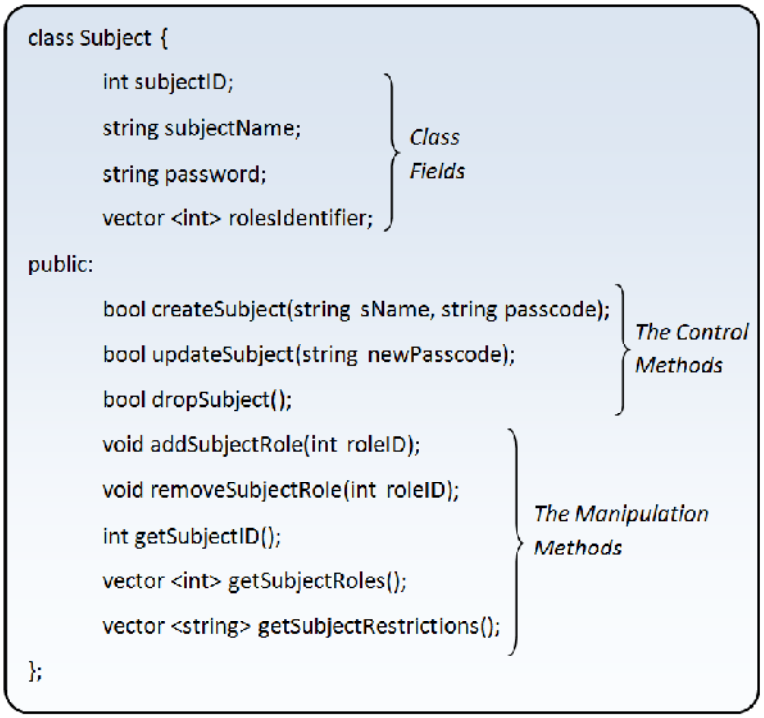
Figure 5.7(a) then depicts the SC in UML class diagram notation. The class contains four fields. The first stores the subject identifier, which is of type int and is automatically generated when the object is created. The next two fields are of type string and are used for storing the user credentials (e.g., subjectName and password) provided by the administrator. The last field is used to hold a list of role identifiers that the user is associated with. We note that these fields are private members, which are accessible only from within the methods of the same class. However, the class methods are defined with public access — they are accessible from anywhere the class object is visible. The SC methods can be classified into two broad categories: Control Methods and Manipulation Methods. Together, they are used to create and assign privileges to users in order to give them permissions to access certain data. Figure 5.7(b), for example, illustrates how one would provide the SC declaration in the C++ language. For now, only method declarations are presented; syntax and definitions are given in Section 5.7. However, brief descriptions for these methods are given below.

- The Control Methods are used to create, update, and destroy subjects, and include the createSubject() method that creates a SC object, the updateSubject() method that modifies the attributes of an existing object, and the dropSubject() method that removes a specified object if it is not being used anymore.

- The Manipulation Methods, on the other hand, provide the most direct method for granting and revoking user privileges and authority. While some of these methods are used to maintain user-role memberships, others are used to obtain user information. For instance, the methods addSubjectRole() and removeSubjectRole() are used to associate an existing user with one or more roles, and to



(a)



(b)

Figure 5.7: (a)The UML Class Diagram of SC (b)The SC declaration in C++

remove a user from the relevant role membership, respectively. The methods getSubjectRoles() and getSubjectRestrictions(), on the other hand, are used to obtain all roles for a specific user, and all restrictions on a specific user, respectively. Further details and examples are given in Section 5.7.

5.6.2 The Object Class (OC)

The Object class (OC) provides a template to describe the protected data. The data contained in data cubes is presented at different levels of granularity (here "granularity" refers to the size of individual data elements that can be authorized to users) such as the dimensions of the multidimensional cube, hierarchies within each dimension, and the aggregated cells. Administrators can, of course, specify the protected data at the level of tables, columns, etc. but this is quite difficult if abstract ideas like hierarchies or dimensions are spread throughout or across tables. Moreover, due to the redundancy inherent in a data cube, even if such data is protected, it can be computed from its more granular levels. Therefore, any level that allows protected data to be derived must also be protected.

Ideally, the system administrator should not be responsible for identifying or specifying all possible data items. Instead, we define the protected data as abstract objects that are directly associated with the conceptual properties and elements of the OLAP data model itself. These abstracted objects are defined using Object Classes without any reference to their locations or their granularity. A primary advantage of this approach is to allow administrators to work at a higher level of abstraction — one that matches their intuitive understanding of an OLAP database — and to transparently propagate authorizations associated with the data object across all levels that may be used to reveal such data.

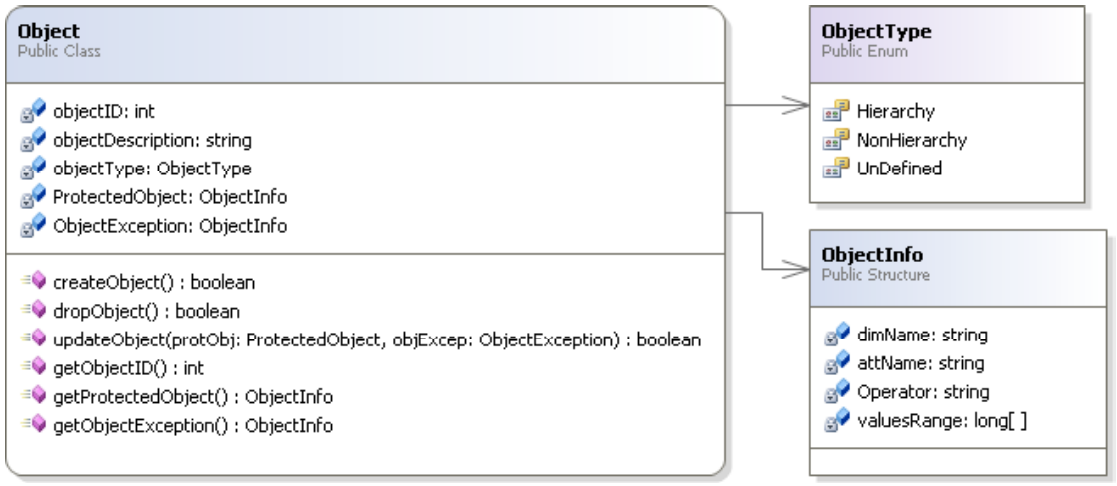
Given the above, we now turn to the declaration of OC itself. OC consists of a tuple (A, C, E, M), where A is a set of attributes describing the data object to be protected, C is a set of conditions that specifies the protected values, E is a set of conditions that specifies the exception, if one exists, and M is the set of methods allowed on the class objects. Definition 14 and Definition 15 formalize C and E, respectively.

Definition 14. *Let C be a set of conditions that specifies the protected values. C is defined as an n-tuple (c_1, ..., c_n) that may be connected by logical operators (AND, OR) to define complex predicates protecting a cube element. We say that C forms a general representation of any criteria/condition that may restrict any element in a data cube by applying both arithmetic and logical operators.*

Definition 15. *Let E be a set of conditions (e_1, ..., e_m) that specifies the exception values. E defines a subset of the protected values as an exception, such that $E \subset C$.*

Example 13. Recall the restrictions of Example 12. The previous definitions will be applied to the first restriction only, where the sales of Quebec cities are restricted except the city of Montreal. However, the definitions can be easily applied to the second one. As such, a protected object *obj* can be defined as the following:

- A = An object identifier $\cup$ a set of attributes to describe the object (e.g., object description, object type).
- C = The restriction (e.g., Stores.Province = "Quebec").
- E = The exception (e.g., Stores.City = "Montreal").
- M = A set of methods (e.g., create object, drop object, update object, get object info).



(a)

```
struct ObjectInfo
{
        string dimName;
        string attName;
        string Operator;
        int valuesRange[];
};

class Object {

        int objectID;

        string objectDescription;

        ObjectType objectType;

        ObjectInfo ProtectedObject;

        ObjectInfo ObjectException;

public:

        bool createObject();

        bool dropObject();

        bool updateObject(ObjectInfo protObj, ObjectInfo objExcep);

        int getObjectID();

        ObjectInfo getProtectedObject();

        ObjectInfo getObjectException();
};
```

(b)

Figure 5.8: (a)The UML Class Diagram for OC (b)The OC declaration in C++

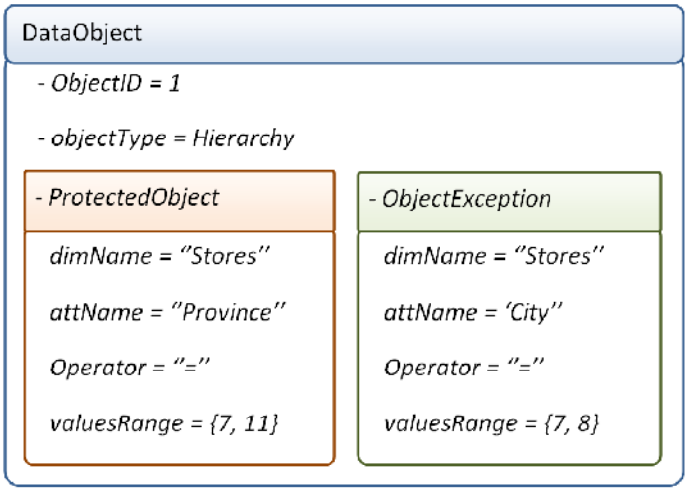


Figure 5.9: An instance of Object Class

For instance, Figure 5.8(a) and Figure 5.8(b) provide the declaration of OC in UML class notation, and in the C++ language to illustrate a practical implementation. As noted, besides the attributes A that identify the protected data object, OC contains two structures that explicitly define the protected data object (i.e., dimension, an attribute within an attribute, or an attribute value). Specifically, while the first structure (ProtectedObject) refers to the protected data object, the second structure (ObjectException), defines the exception for this object, if it exists.

Figure 5.9 then depicts an instance of the Object Class, as per the given example. The dimName and the attName attributes in the ProtectedObject structure store the dimension name and the attribute name of the protected objects (e.g., Stores and Province). The operator attribute stores the conditional operator. In this case, it is the equality operator =, but other possibilities include <, >, <>, etc. Finally, the valuesRange attribute stores the range of the protected values. On the other hand, the attributes of the ObjectException structure store the same information but for the exception.

Ultimately, the OC objects are controlled and managed by a set of methods M. While the methods createObject(), dropObject(), and updateObject() are used to create, drop, and update an object, the methods getProtectedObject() and getObjectException() are used to obtain the object's details. The syntax and definitions of these methods, along with further details, are given in Section 5.7.

5.6.3 The Role Class (RC)

The Role Class (RC) is used to create objects that represent the system's roles. A role regulates the activities of its members through a set of restrictions. In our approach, instead of specifying such restrictions for each user, they are specified on data objects representing shared roles. In other words, restrictions are associated with the data objects that are grouped together or encapsulated into Role objects, as formally stated in Definition 16.

Definition 16. *Let OC_1, OC_2, ..., OC_n be n objects of Object Class (OC). We define a Role class from these n objects as a tuple (O,A,M), where O is the set of protected objects created by OC, A is a set of fields that characterizes the Role objects, and M is the set of methods allowed on the Role objects.*

That is, the RC is always based on OC objects and therefore at least one object of OC should exist prior to the initialization of any RC object.

Example 14. Suppose a new role (e.g., *Analysis*) is defined with the protected object (e.g., *obj*) created in Example 13. The analysis role would be defined as:

- $O = \{obj\}$.
- $A =$ A role identifier $\cup$ a set of attributes to describes the role (e.g., role name, role description).

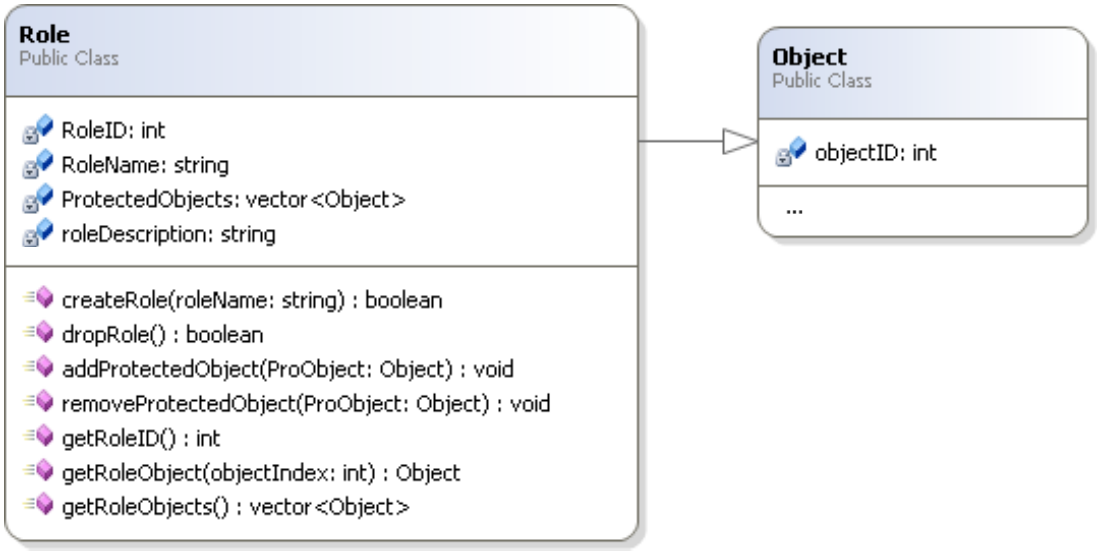
- M = A set of methods (e.g., create role, drop role, update protected objects, get role info).

Figure 5.10(a) and Figure 5.8(b) depict RC in both UML class notation and the C++ language. As noted, besides the protectedObjects attribute that contains the associated protected objects, RC includes attributes that describe the Role Objects (e.g., RoleID, RoleName, etc.) and a set of methods that controls the RC objects (e.g., createRole(), dropRole(), getRoleObjects(), and getRoleID()) and that manipulates the protected objects associated with these objects (e.g., addProtectedObject() and removeProtectedObject()). Further descriptions of these methods are given in Section 5.7.

Ultimately, Roles may be structured within a role-hierarchy, with inheritance of restrictions such that a role may inherit all restrictions that are assigned to its parents up to the top of the hierarchy. Consequently, all users who are mapped to this role are affected by the role's restrictions *plus* all the inherited restrictions. Figure 5.11(a) shows a Role Class Hierarchy in UML Class notation, while Figure 5.11(b) illustrates it's implementation in the C++ language. In this figure, objects created by the DerivedRole class inherit all the protected objects from the base Role.

Example 15. Suppose the administrator in Example 12 wishes to build the simple roles hierarchy that is depicted in Figure 5.12,

To build such a hierarchy, each child role should be created as a derived role with respect to its parent. This can be done by using the Role class to create the base roles, and the DerivedRole class to create the derived roles. For instance, the e_Reporting role should be created using the DerivedRole class. Consequently, any user who is assigned to this role will be restricted from accessing the Product dimension data —

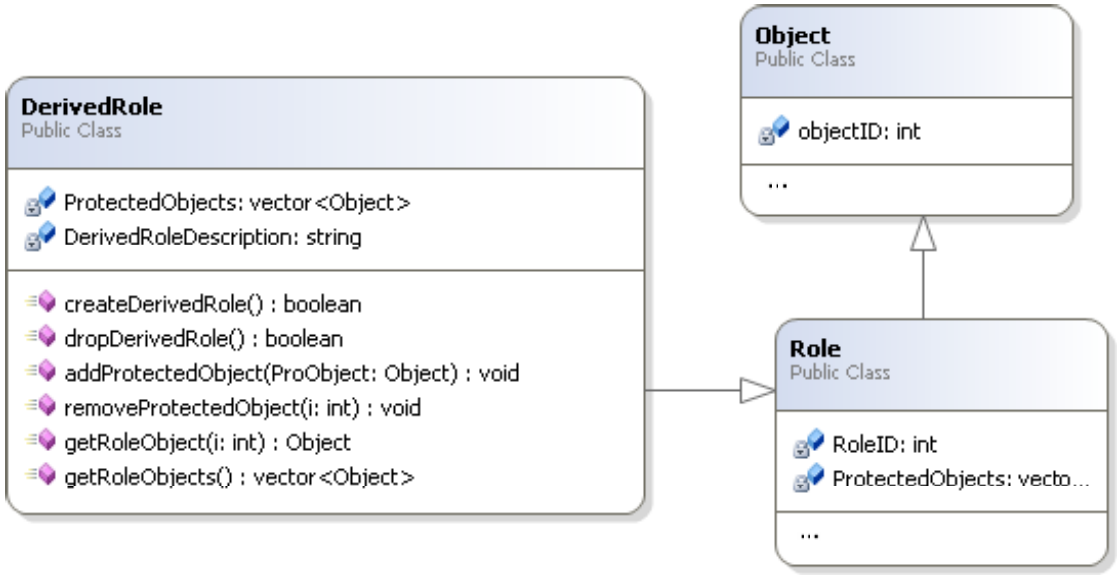


(a)

```
class Role {
        int RoleID;
        string RoleName;
        string roleDescription;
        vector <Object> ProtectedObjects;   <-- Predefined protected objects
public:
        bool createRole();
        bool createRole(Object);
        bool dropRole();
        void addProtectedObject(Object ProObject);
        void removeProtectedObject(Object ProObject);
        int getRoleID();
        Object getRoleObject(int objectIndex);
        vector<Object> getRoleObjects();
};
```

(b)

Figure 5.10: (a)The UML Class Diagram for RC (b)The RC declaration in C++



(a)

```
class DerivedRole : public Role {
        vector<Object> ProtectedObjects;
        string DerivedRoleDescription;
public:
        bool createDerivedRole();
        bool dropDerivedRole();
        void addProtectedObject(Object ProObject);
        void removeProtectedObject(int i);
        Object getRoleObject(int objectIndex);
        vector<Object> getRoleObjects();
};
```

(b)

Figure 5.11: (a)The Role hierarchy in UML Class Diagram (b)The Role hierarchy in C++

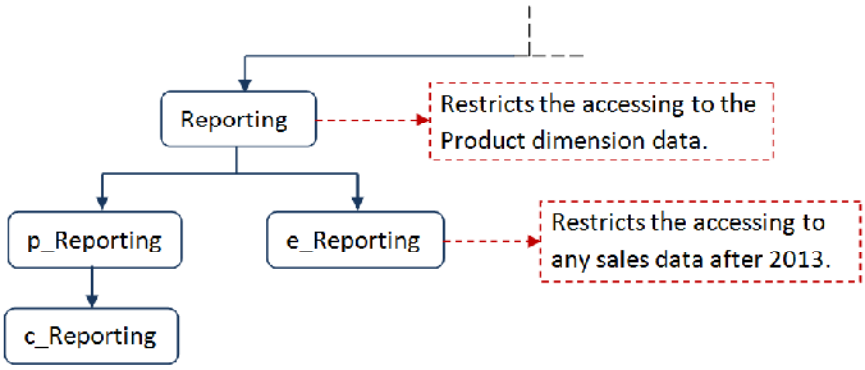


Figure 5.12: A simple roles hierarchy

by the role restrictions — and any sales data after 2013 — by the inherited restrictions.

5.6.4 The Policy Class (PC)

Once we have defined the different classes that we will have in our system, we proceed to the definition of the policy class (PC). The PC allows administrators to specify the requirements of the organization's security policies relevant to the various cube elements. These requirements outline the association between Subjects, protected data, and Roles — the security policy components. So far, we have defined the different classes that create objects to represent these basic components. We now turn to the PC itself. The definition of PC is formally stated in Definition 17, followed by an example to show its construction.

Definition 17. *Let SC be a Subject Class and RC be a Role Class. We define a Policy Class (PC) as a tuple (SO,RO,A,M), where*

- *SO and RO are the Subject and Role objects that have been created by SC and RC.*
- *A is a set of fields that identifies the policy object and defines the association between its components.*
- *M is the set of methods allowed on the Policy objects.*

Example 16. By applying the class definition of PC to the policy of Example 12, the following policy (e.g., *policy1*) can be constructed.

- SO = {*Bob*} — the subject that has been created by the SC.
- RO = {*Analysis*} — the protected object that has been created by the RC.
- A = A policy identifier ∪ a set of attributes to describe the policy and the association between its components (e.g., role name, role description, Subjects Roles Assignment).
- M = A set of methods (e.g., create role, drop role, update protected objects, get role info).

With respect to the definition of PC, it can be considered as a composite class where Subject and Role objects are the components. In other words, it encapsulates the data that defines these components and the association between them. For instance, referring to Example 16, *Bob* and *Analysis* objects represent these components, and the association between them is represented by the Subjects Roles Assignment attribute. In addition, the class includes a set of methods that control these components. Figure 5.13(a) and Figure 5.13(b) depict the PC representation in UML class notation and in the C++ language, respectively.

As noted in the figure, the subjects and roles fields store the SC and RC objects, while the association between them is represented by the field SubjectsRolesAssignment and can be modified by using the methods assignSubjectRole() and withdrawSubjectRole() that allows one to define complex policies and reduces the complexity of associating users with roles. Further details about these methods are given in the next section.

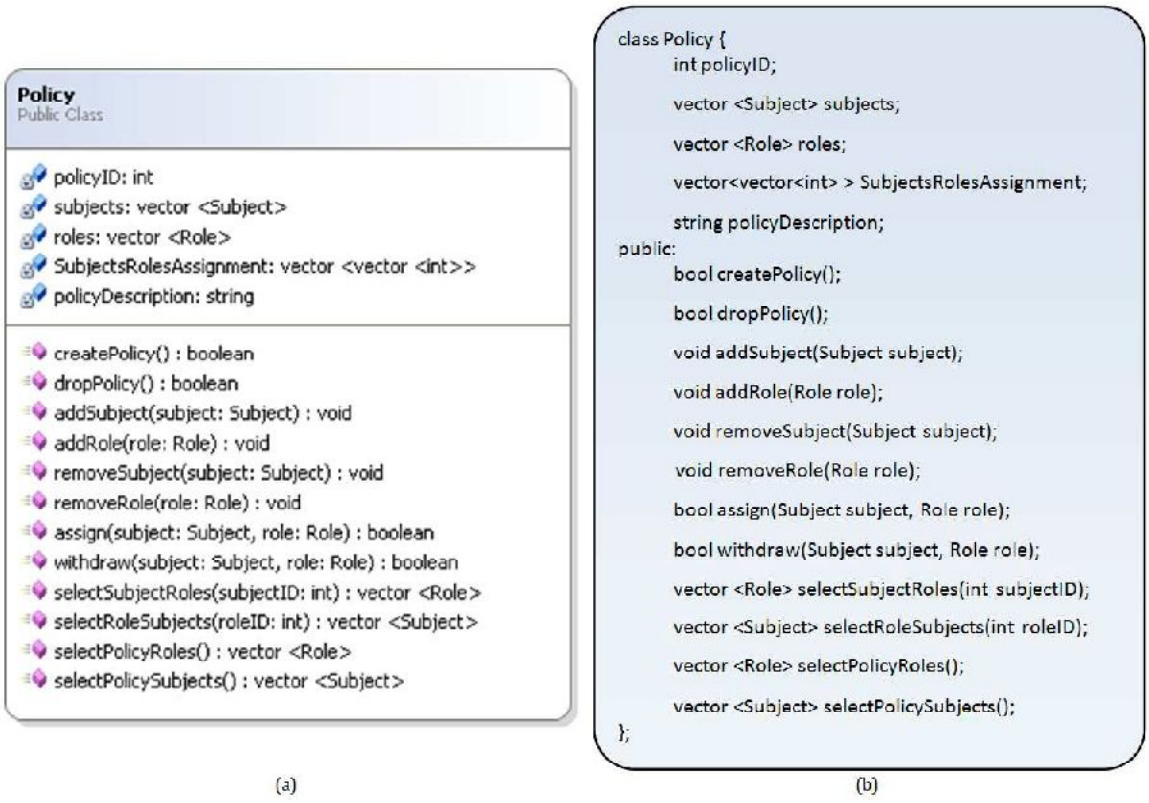


Figure 5.13: (a)The PC in UML Class Diagram (b)The PC declaration in C++

5.7 Integration Options

In the preceding sections, we have described an abstract data model for the specification and maintenance of authorization objects within multi-dimensional environments. Such a model provides end users — in this case administrators and schema designers — with the ability to focus on access constraints without concern for physical storage representation or even syntactical idiosyncrasies. That being said, policies must eventually be specified in a concrete manner and entered into a policy repository in some way. In the current context, we suggest that this would be done in two distinct, but cooperative forms. On the one hand, the conceptual representation can be exposed directly in an Object Oriented programmatic API. On the other, a more conventional SQL-style interface can be provided for direct interaction with the DBMS in a manner consistent with conventional approaches. Because the OO approach ultimately "piggy backs" on top of the SQL model, we begin the discussion of the integration options with this more conventional implementation.

5.7.1 Declarative Language Extensions

Existing access policies are typically specified via SQL statements, most notably the GRANT/REVOKE commands of the Data Control (sub) Language (DCL). Here, basic privileges (e.g., INSERT, SELECT, UPDATE, EXECUTE) can be associated with the elements of the logical schema (e.g., tables, views, procedures). For convenience, privileges can in turn be organized into simple Roles. For example, the **testing** Role can be created by the database designer, with the ability to create new tables then assigned to this role.

The more sophisticated security constraints discussed throughout this thesis can

in fact be integrated into this specification environment. Moreover, the OO characteristics of the conceptual model can be "flattened" to suit the simple declarative structure of the DCL without sacrificing expressibility. In the remainder of this section, we describe the extensions — organized in two categories: Control Commands and Manipulation Commands — to the current SQL/DCL language required to support a rich authentication and authorization model for the OLAP domain.

The Control Commands

The control commands are responsible for defining policy objects (e.g., Subjects, Protected Objects, and Roles). We provide an overview of each command below.

1. The CREATE Command creates Subjects, Roles, and Protected Objects depending on the given parameters, as listed below.

- The **Create Subject** command creates a new user account identified by a password. The new user's privilege domain is initially empty; thus, it should be assigned to one or more roles.

  The Syntax: *Create* ***Subject*** *subject_name* ***With*** *password*

  The parameters *subject_name* and *password* respectively determine the name of the new user account and its password.

- The **Create Role** command defines a new role that can be a child of another existing role. After creating the role, one can add restrictions and define membership within that role.

  The Syntax: *Create* ***Role*** *role_name* ***Child****? parent_name?*

The keyword *Role* specifies that a new role may be created, while the *Child* keyword indicates that the new role is a child of another role. If the parent role is missing, the new role is assumed to be the *root role*. Parameters *role_name* and *parent_name* determine the name of the new role, and the parent role name respectively.

- The **Create Restriction** command creates a restriction on accessing a specified data cube element. This includes a dimension, an attribute within a dimension, or an attribute value(s) with/without exception.

  The Syntax: *Create* ***Restriction*** *restriction_name* ***On*** *cube_element*

  The keyword *Restriction* specifies a new limitation to be created. The *On* keyword determines the restricted element. Parameters *restriction_name* and *cube_element* determine the name of the new restriction and the name of the restricted element respectively. However, certain restrictions may have exception(s). In this case, the syntax is extended as following:

  The Syntax: *Create* ***Restriction*** *restriction_name* ***On*** *cube_element* ***Except*** *exception;*

  The keyword *Except* indicates that an exception exists, while the parameter *exception* determines the limitation exception.

2. The DROP Command removes a specified object. Using this command one can drop a pre-initialized object as shown below.

- The **Drop Subject** command removes a user account permanently from the policy repository. Consequently, its roles memberships will also be removed

automatically.

The Syntax: *Drop* ***Subject*** *subject_name*

- The **Drop Role** command removes a specific role. It is not necessary to remove role memberships. Instead, the command automatically revokes any user membership for the specified role.

  The Syntax: *Drop* ***Role*** *role_name*

- The **Drop Restriction** command removes a defined restriction and consequently removes it from all associated roles.

  The Syntax: *Drop* ***Restriction*** *restriction_name*

2. The UPDATE Command updates the properties of an existing protected object/restriction or a subject. There are two variants of this command. The first variant updates the restriction itself or its exception if it is exists, while the second one updates the subject password.

The Syntax: *Update restriction_name* ***Set Restriction*** *newRestriction*

The keywords *Set Restriction* specify that an existing restriction would be updated. The name of the restriction is determined by the *restriction_name* parameter, while the *newRestriction* parameter determines the new restriction. However, in order to update an exception, the syntax is modified by as follows:

The Syntax: *Update restriction_name* ***Set Exception*** *newException;*

The keywords *Set Exception* specify that an existing exception should be updated. The name of the restriction is determined in the *restriction_name* parameter, while the *newException* parameter determines the new exception.

The Manipulation Commands

The manipulation commands are used to manage policy objects. While some commands are used to maintain role memberships, others are used to retrieve the security object's information. Generally, these commands contain clauses referring to the name of the security object that is being processed. Some examples:

1. The Assign Command assigns an existing subject to one or more existing roles according to his/her duties.

The Syntax: ***Assign*** *subject_name* ***To*** *role_name*

The *subject_name* parameter determines the name of the user account, while the *role_name* parameter determines the role to be assigned. The following example adds the user account *Sue* to the role *Marketing*.

Example: *Assign Sue To Marketing*

2. The Revoke Command is used when a subject's duties are changed; for instance, when a subject's restrictions are no longer needed. The subject should then be revoked from the role that restricts the subject access. The syntax of this command is shown next, followed by an example that illustrates how *Sue* can be revoked from the role *Marketing*.

The Syntax: ***REVOKE*** *subject_name* ***From*** *role_name*

Example: *REVOKE Sue From Marketing*

3. The Add Command adds a restriction to a specific role. As a result, all subjects assigned to that role will be affected by this restriction.

The Syntax: ***Add*** *restriction_name* ***To*** *role_name;*

The *restriction_name* and *role_name* parameters determine the name of the restriction to be added to the role.

4. The Remove Command represents the complementary case of the previous command (e.g., the Add command). A restriction and/or its exception can be removed from a specific role to make it less restrictive.

The Syntax: *Remove* ***Restriction*** *restriction_name* ***From*** *role_name*

The parameter *restriction_name* determines the name of the restriction to be removed from a role that, in turn, is determined by the parameter *role_name*. In case of deletion of the exception only, the following command is used. However, this deletion will affect all roles that hold the specified restriction.

Syntax: *Remove* ***Exception From*** *restriction_name*

5. The Select Command retrieves an object's details or its membership information. The same command is used with all objects but uses different parameters. Some examples are listed next.

- **Select** Subjects Of Role *role_name*: Gets all subjects assigned to a specific role.
- **Select** Roles Of Subject *subject_name*: Gets all roles of a specific subject.
- **Select** Restrictions On Subject *subject_name*: Gets all restrictions on a specific subject.
- **Select** Restrictions Of Role *role_name*: Gets all restrictions of a specific role.

5.7.2 Programmatic API

While SQL has been the de facto standard for general database interaction for the past several decades, programmatic APIs have also been developed. Perhaps most significantly, ODBC and JDBC have become the standard means by which to deliver query statements to the DBMS and, subsequently, receive results. More recently, Object Relational Mapping (ORM) frameworks such as Hibernate [73] have been developed to minimize the impact of the impedance mismatch caused by Object-to-Table mapping logic.

In such cases, of course, it is important to note that the queries encapsulated by the APIs methods are typically data queries. In other words, such queries are interactively retrieving and updating dynamic operational data. Strictly speaking, it is possible for an OO implementation of the security classes previously discussed to be used directly against the DBMS. However, in most situations it would be cumbersome to specify and maintain policy specifications programmatically (i.e., writing application code to view and maintain policy objects). In fact, a more likely scenario would be the use of graphical tools to allow intuitive modeling and maintenance of complex enterprise policies.

It is in this context that an OO API could be quite useful. Specifically, the design of modeling applications would be significantly simplified by a direct proxy interface. In other words, the policy classes discussed in the preceding sections could be exposed as wrappers to the backend repository. Developers could then manipulate policy objects within the graphical interface in order to provide security specialists with an intuitive, point-and-click mechanism for setting and modifying authorization constraints.

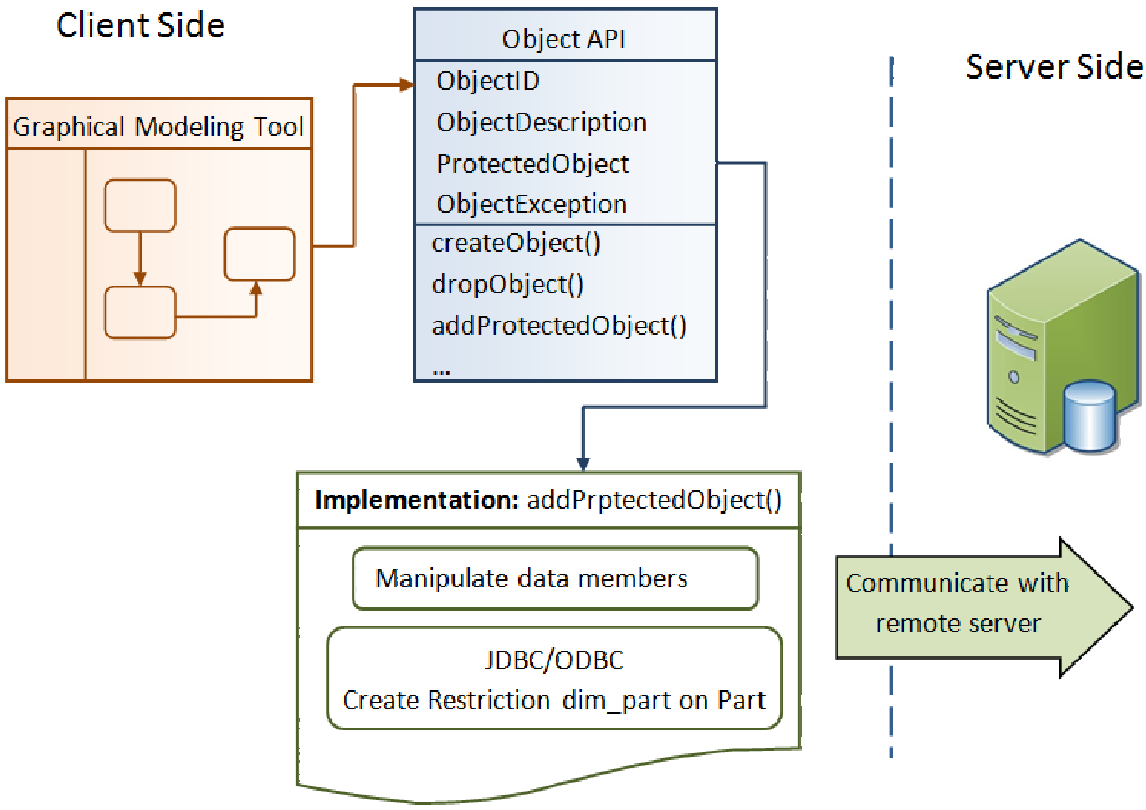


Figure 5.14: The logic of our model

Figure 5.14 provides a simple illustration of the model. Here a client side Object (i.e., OC) houses data related to a given Policy Object specification. The implementation exposes an OO API to the programmer who can then provide graphical representation. It terms of the class implementation itself, it would function as a proxy in the sense that it would relay data (e.g., object instantiation and updates) to the DBMS repository. To do so, it would utilize the low level JDBC/ODBC API to transmit the extended SQL statements described in Section 5.7.1. All of this logic would, however, be transparent to the user as he/she would simply work with the graphical representation of the policy objects.

In the remainder of this section, we provide a brief overview of the primary methods (classified into two broad categories: Control and Manipulation methods) that one would utilize as part of the OO API. A more complete listing of the methods can be found in Appendix E.

The Control Methods

1. The CREATE Methods are responsible for creating Subjects, Protected objects, Roles, and Policies. They act as constructors that prepare the new objects for use either as targets or as subjects in the policy definition process. Four variants of this method are listed below.

- The **CreateSubject()** Method is used to create new subject accounts. The new account privilege is initially empty; thus, it should be assigned to one or more roles to define its privileges.

- The **CreateObject()** Method creates a protected data object specified by a set of restrictions. These restrictions can be considered as conjunctions of basic

conditions, which are deduced as per the given policy.

- The **CreateRole()** Method can be used to create different roles based on the fundamental processing tasks within the organization. Each role encapsulates one or more protected data objects to limit the accessing of its members.

- The **CreatePolicy()** Method creates different policy instances. Each policy defines the association between a Subject and a Role in order to specify the subject access rights. However, the policy complexity can be substantially increased by combining additional Subjects and Roles. These components can either be created within the method itself or passed as formal parameters to provide re-usability. Multiple policies can then be created and individually tailored in order to achieve the required security objectives.

2. The UPDATE Methods update the properties of an existing Protected Object or Subject. On one hand, the attributes of a protected object can be updated using the **UpdateObject()** method. In this case, variants include (i) modification to the protected data itself (ii) modification of the exception value(s), if it exists, or addition of the exception otherwise. On the other hand, the **UpdateSubject()** method updates the Subject's properties, such as the Subject's password.

3. The DROP Method permanently removes an existing object. It acts as the destructor in object oriented languages in deallocating memory and doing other cleanup for a class object and its class members when the object is not used anymore. It is important to mention that it is not necessary to remove the object membership(s) beforehand. Instead, the drop method should automatically remove any membership(s)

for the given object. For instance, there is no need to explicitly remove Subjects assigned to a Role prior to dropping it.

The Manipulation Methods

1. The ASSIGN Method provides the most direct method for granting user privileges and authority. It is used to associate an existing Subject with one or more existing Roles. Consequently, the Subject will be affected by the new restrictions through his/her Role participation.

2. The WITHDRAW Method takes the responsibility of removing a Subject from a relevant Role membership due to the evolution of the Subject's duties. This removal will not affect his membership in any other Role.

3. The ADD Methods add predefined Subjects and Roles to a Policy. Specifically, the **AddSubject()** method adds a predefined Subject, while the **AddRole()** method adds a Role to limit the subject.

4. The REMOVE Methods form the complementary case of the previous methods (e.g., the Add methods). An existing Subject or Role can be removed from a Policy using the **RemoveSubject()** or **RemoveRole()** methods, respectively.

5. The SELECT Methods are used to query information about the policy components. Four different methods are utilized. The first two methods are used to retrieve memberships information. For instance, the **SelectRoleSubjects()** retrieves the membership for a specific Role, while the **SelectSubjectRoles()** retrieves the

Roles that a specific Subject is involved in. The other two methods are **SelectPolicyRoles()** and **SelectPolicySubjects()**. They retrieve the information about the Roles and the Subjects that participate in a specific Policy, respectively.

To illustrate how such methods can be utilized, recall the concrete policy of Example 12 that restricts the access of the analyst Bob. Further, we assume that the role Analysis of Example 14 will be defined with a membership that includes Bob. According to this example, the subject Bob and the role Analysis should be defined first through the methods CreateSubject() and the CreateRole(), then Bob will be assigned to the role Analysis through the Assign() method. Figure 5.15 and Figure 5.16 show the syntax diagrams for the CreateSubject() and CreateRole() methods, respectively. The syntax diagram of the Assign() method as well as the syntax diagrams of the rest of the methods are provided in Appendix E.

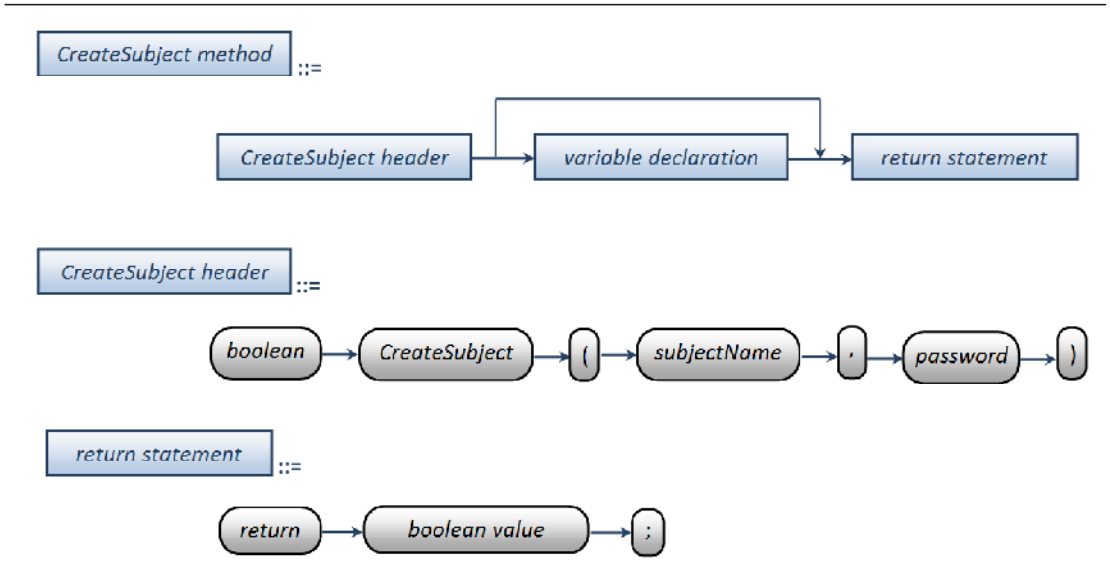


Figure 5.15: The Syntax Diagram for the CreateSubject method

Referring to Figure 5.15, the first syntax diagram starts with the method header, followed by some optional variable declarations, and ends with the **return** statement. The method header shows the return type, which is a boolean value, followed by the method name and two parameters (e.g., *subjectName* and *password*). The parameters subjectName and password determine the name of the new subject account and its password. These parameters are used to identify each subject in the system uniquely. Finally, the return statement returns true if the subject is created successfully, and false otherwise.

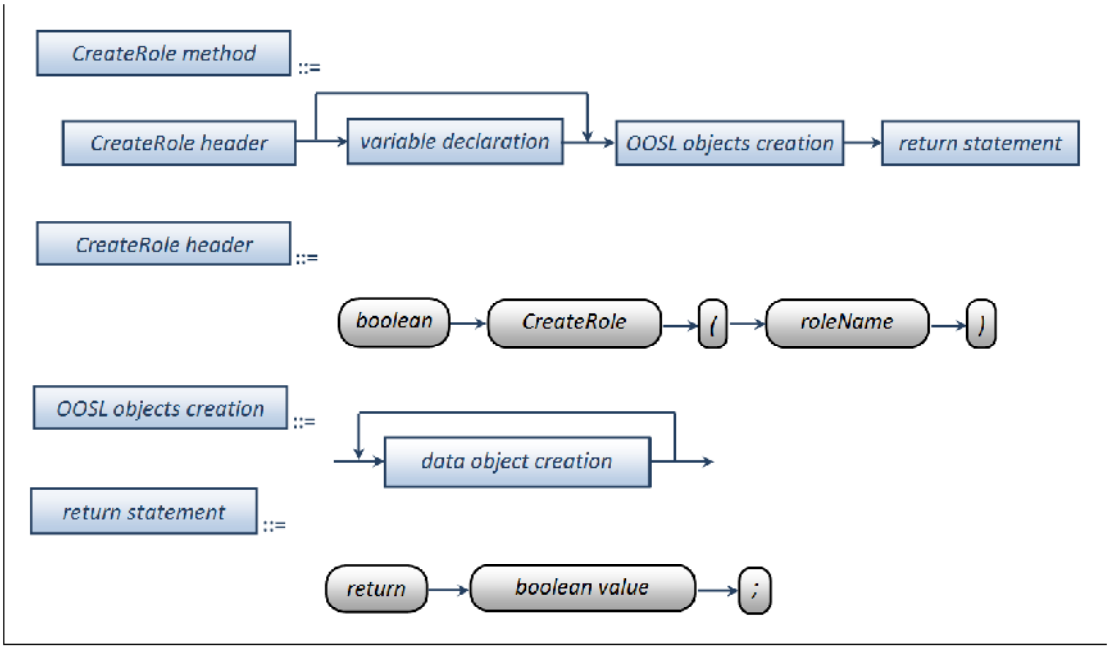


Figure 5.16: The Syntax Diagram for the CreateRole method

On the other hand, Figure 5.16 depicts the syntax diagram for the CreateRole() method. As shown in the figure, the create method header is followed by optional variable declarations and object creation, and ends with the return statement. The

objects here are of type Object and define different restrictions on user access. As noted, at least one object must be defined to execute this method successfully. In the case of successful creation, the method returns true, and false otherwise.

5.8 The OSSM Policy Engine Structure

While the proposed security model provides the logic and syntax for defining security policies, it is important to note that without an appropriate server-side engine to support the client-side language interfaces, the process of defining, storing, and managing security policies is not possible. In this section, we discuss a policy management engine and provide a detailed description of its structure and functions.

Referring back to Figure 5.6, we note that the engine consists of three main components; namely, the user interface component that is used for defining security policies, the policy repository component that stores the policies generated by the user interface, and the policy manager component that manages the policies stored in the policy repository. In the following subsections, we will discuss these components and describe their functionality.

5.8.1 The User Interface

The user interface is the means by which an administrator actually defines policies. It is an important part of the overall framework since it hides low-level policy details and permits the administrator to define the language classes using either a declarative syntax or an OOP interface. Details of the options for language interfaces have been discussed in the previous section.

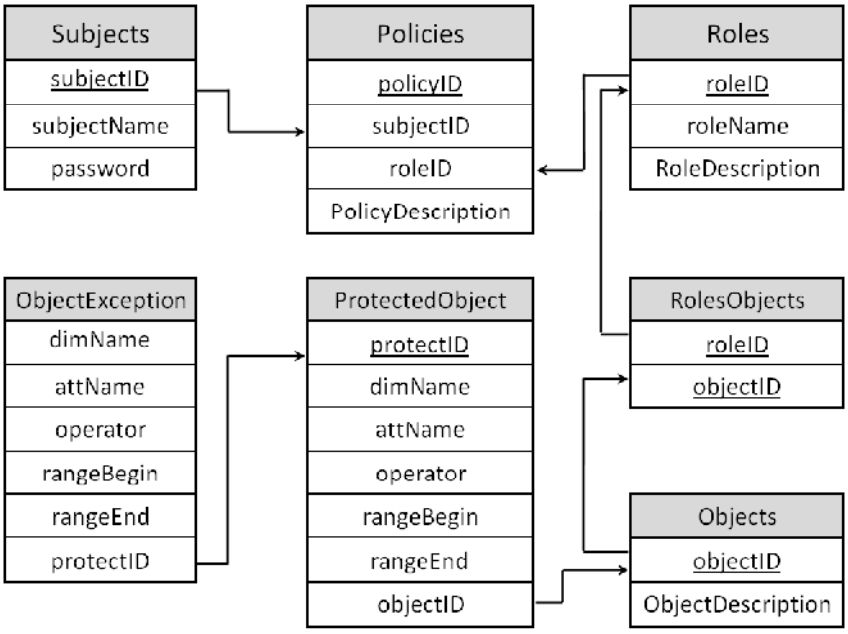


Figure 5.17: The Policy Repository

5.8.2 The Policy Repository

The policy repository is used to store the policies generated by the language interface. It acts as a bridge between the interface software and the policy management component. In short, the interface is used to create policies, which are then stored in the repository, and which the policy management component later consumes.

The Policy Repository itself consists of a set of tables (Subjects, Objects, Roles, and Policies) that collectively represent the meta data required to define security measures. For example, the Subjects, Objects, and Roles tables store the main components required to create policies, which are subsequently stored in the Policies table. Figure 5.17 illustrates a slightly simplified version of the policy repository. Note that this meta-schema is the same as the one depicted in Figure 3.4 but with some additional components.

In the current prototype, storage and access to the Policy Repository is provided by the SQLite toolkit [107]. 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.

5.8.3 The Policy Management component

The policy manager is a bidirectional component: the access control module requests policies that are associated with a specific user when it receives his/her query; the policy manager, in turn, contacts the policy repository and retrieves the applicable policies, which are then sent to the access control module to enforce them. The access control module enforces policies by permitting/rejecting requests to access a particular data item. However, the policy manager may need to resolve the conflicting authorizations before forwarding them to the access control mechanism. In other words, in a typical scenario, a user may play multiple roles, according to his/her duties. Roles may therefore have overlapping access rights due to these overlapping responsibilities. In such situations one role (i.e., senior role) may have privileges to access all pieces of data authorized for one or more other roles (i.e., junior roles) but the inverse is not true. For instance, an administration role may have full access to the data cube, while an analysis role might restrict access to one or more dimension(s). To address such issues, our policy management component relies on an open source library called tree.hh [114] to organize the roles in a hierarchical structure (i.e., a tree structure). A discussion of how inheritance is resolved in the context of overlapping roles is provided in the next section.

5.9 Managing Overlapping Roles

It would be helpful to be able to assign a user to more than one role according to his/her duties. Each role would have different permissions to access specific data elements. In most cases, roles do not overlap/conflict. However, in a rapidly changing enterprise environment, a user would likely have more than one role, and some of these roles may very well overlap. Most existing OLAP servers apply a "Restriction takes Precedence" principle. The problem with this approach is that it may lead to unintended restrictions on accessible data. In other words, if a user is a member of several roles with different permissions, the user will be restricted according to the permissions of the least powerful role, which results in restricting the user from access to data even if he/she is permitted to access them by using other role(s).

For example, suppose the administrator of the data cube depicted in Figure 5.1 is included in the roles shown in Figure 5.18 and assigns user Alice to the Administration role, which has full access to the whole cube. Over time, and because of special situations, Alice is assigned also to the Marketing role, which restricts her from accessing Product dimension data. In this example, Alice's roles conflict; Alice is restricted from accessing Product data because of the Marketing role, but at the same time, she is allowed to access the same data because of the Administration role.

To address this issue, Alice should be given the highest permissions amongst her roles by finding her highest non-conflicting roles. To do this, roles should be organized in a hierarchal structure (i.e., a tree structure) in order to pick-up the highest role/node. To represent roles in this form, the open source tree.hh library is employed, an STL-like container class designed to represent n-ary trees [114]. Tree.hh

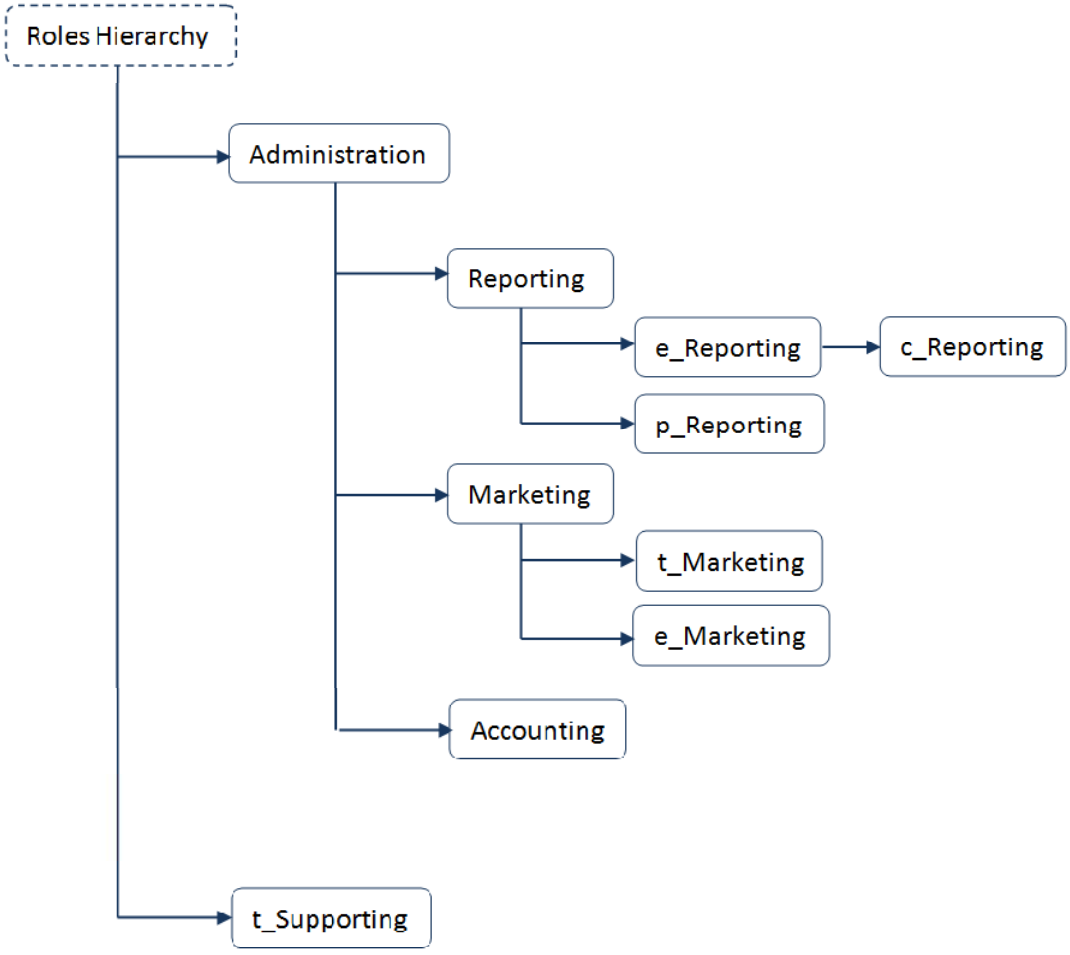


Figure 5.18: An example of Roles Hierarchy

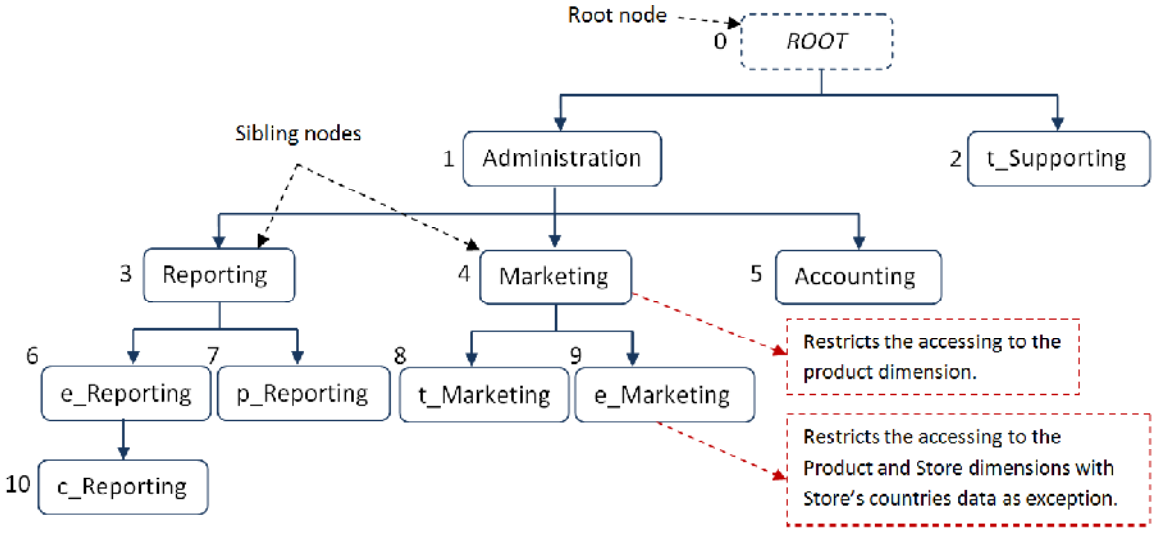


Figure 5.19: Roles Hierarchy represented in a tree structure

provides various types of iterators such as breadth first, depth first, and sibling iterators. In general, access methods are compatible with the C++ STL libraries. In our case, we assume a depth first search strategy, where the algorithm traverses the tree starting from the root role(s) and explores as far as possible along each branch before backtracking. If the role is identified as an assigned role, the role's restrictions are extracted and the traversal then backtracks to another branch. In the worst case, we have to visit each node exactly once, since we do not cross the same edge more than once. As such, the time complexity is O(n) where n is the number of roles/nodes, which is generally quite small.

Figure 5.19 illustrates the Role Tree associated with the roles hierarchy depicted in Figure 5.18. Numbers near each node represent the roleID. In the roles tree, every node is connected to an arbitrary number of child nodes/roles. At the top of the tree, there may also exist a set of roles which are characterized by the fact that they do not have any parents. Nodes at the same level are called "siblings" and are not

Users Roles Table			
UserID	**User_name**	**RoleID**	**Role_name**
1	Sue	4	Marketing
1	Sue	9	e_Marketing
1	Sue	6	e_Reporting
1	Sue	2	t_Supporting
...	...	...	...

Highest Roles Table		
UserID	**RoleID**	**Role_name**
1	4	Marketing
1	6	e_Reporting
1	2	t_Supporting
...	...	.....

Figure 5.20: User's roles in the Policy Repository

overlapping. So, if a user is assigned to sibling roles, the user's permissions will be the union of all his role's restrictions. However, nodes at different levels may indeed overlap. Each node may inherit its parent's restrictions if any exist.

To improve the search performance, the user's role(s), along with their restrictions, are stored in the Policy Repository. In addition, the highest or most privileged roles amongst the user roles are also stored there. So, instead of re-executing the search process each time the user sends a query, his highest roles are retrieved from the security database and cached in memory for future queries. Figure 5.20 illustrates an example of user roles, along with the highest roles that are stored in the repository.

Of course, the user's roles can be changed or affected by API methods (e.g., Assign, Withdraw, and Drop). For example, assume that the user Sue is assigned to the following roles: Marketing, e_Marketing, e_Reporting, and t_Supporting. Sue's restrictions will be defined by the union of these roles. Note, e_Marketing is a child of the Marketing role and because the user utilizes the highest role, e_Marketing is not listed in the highest roles table. As a consequence, its restrictions will be ignored.

Now, suppose the user Sue becomes an Administrator user. The roles Marketing,

Users Roles Table

UserID	User_name	RoleID	Role_name
1	Sue	4	Marketing
1	Sue	9	e_Marketing
1	Sue	6	e_Reporting
1	Sue	2	t_Supporting
1	Sue	1	Administration
...	...	...	...

Highest Roles Table

UserID	RoleID	Role_name
1	1	Administration

(a) After executing: Assign Sue to Administration

Users Roles Table

UserID	User_name	RoleID	Role_name
1	Sue	4	Marketing
1	Sue	9	e_Marketing
1	Sue	6	e_Reporting
1	Sue	2	t_Supporting
...	...	...	...

Highest Roles Table

UserID	RoleID	Role_name
1	4	Marketing
1	6	e_Reporting
1	2	t_Supporting
...	...	.....

(b) After executing: Withdraw Sue from Administration

Users Roles Table

UserID	User_name	RoleID	Role_name
1	Sue	9	e_Marketing
1	Sue	6	e_Reporting
1	Sue	2	t_Supporting
...	...	...	...

Highest Roles Table

UserID	RoleID	Role_name
1	9	e_Marketing
1	6	e_Reporting
1	2	t_Supporting
...	...	.....

(c) After executing: Drop Role Marketing

Figure 5.21: How the tables are affected by Assign, Withdraw, and Drop operations

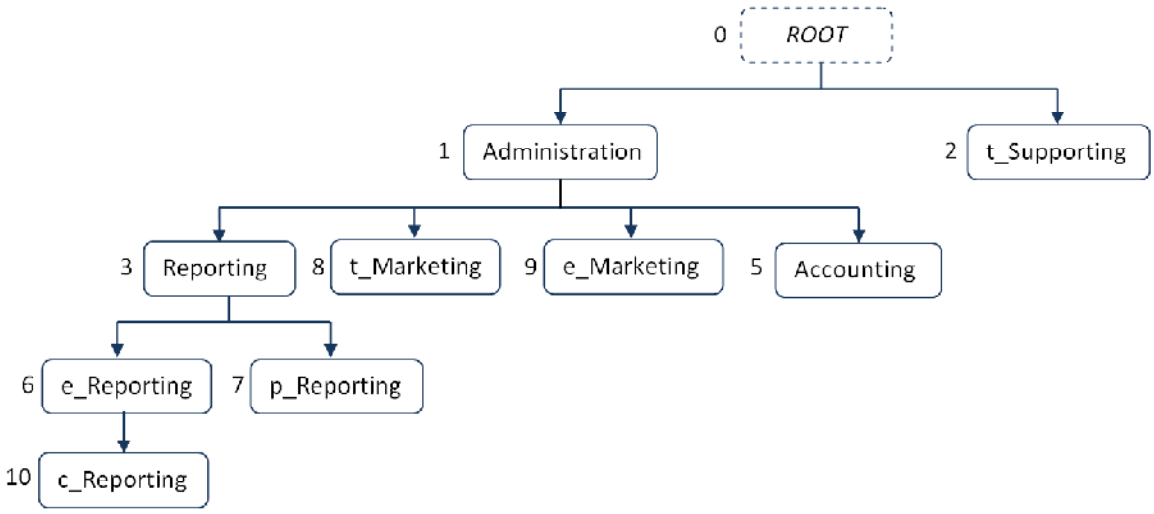


Figure 5.22: The updated roles tree

t_Marketing, and e_Reporting will no longer be listed in the highest roles table, and their restrictions will be ignored in the security checking process because they are children of the the Administration role. Now, suppose that the policy is once again altered and the user Sue is withdrawn from the same Administration role. The user's highest roles should then be reset to Marketing, e_Reporting, and t_Supporting. Figure 5.21 shows an example using the Assign, Withdraw and Drop methods, and their effect on the security tables.

Finally, we note that the Roles Tree itself may also be affected by the Drop Role command. For instance, when a role is dropped, the tree is re-structured by moving all children of the deleted role so that they become siblings of that role. For example, suppose the Marketing role is dropped. Here, the t_Marketing and e_Marketing roles should be connected directly to the Administration role. For this purpose, we also provide an algorithm to rebuild the tree. It starts from the parent of the deleted node, extracts the sub-trees of its child nodes, then attaches each one to the parent

of the deleted role. The full tree can be re-structured if necessary. Figure 5.22 shows the roles tree after dropping the Marketing role.

5.10 Case Study

We now discuss an example to illustrate both the power and intuitive nature of our approach. In the following, the policy and the database employed are borrowed from the Experimental Section of Chapter 4 (Section 4.8). Specifically, our main goal is to show how such a policy can be specified using the OSSM approach. For this purpose, we utilize the database defined by the Star Schema Benchmark (SSB) [92], which consists of one Fact table (e.g., Lineorder) and four Dimension tables (e.g., Customer, Part, Supplier, and Date) as depicted in Figure 5.23. Each dimension defines its own hierarchy. Supplier, for example, includes the suppkey $\preceq$ s_city $\preceq$ s_nation $\preceq$ s_region hierarchy. We note that while the SSB schema itself is defined at the logical level (i.e., a series of relations/tables), it is a straightforward process to map the elements of the OLAP conceptual model (i.e., the data cube) to the logical level and to the elements of the physical DBMS (i.e., columns, indexes, data types). We will therefore discuss policy specification at the cube level, without further concern for the details of physical storage.

Concerning the policy, the SSB schema exposes numerous cube elements that can be secured to various degrees. For instance, one can create restrictions on the whole data cube, a dimension, an attribute within a dimension, or a specific value within the data cube. We begin by assuming that we have a Role hierarchy equivalent to the simple one depicted in Figure 5.3, with three users — Sue, Bob and Yan — having access to the database. Sue is included in the t_Marketing and t_Accounting

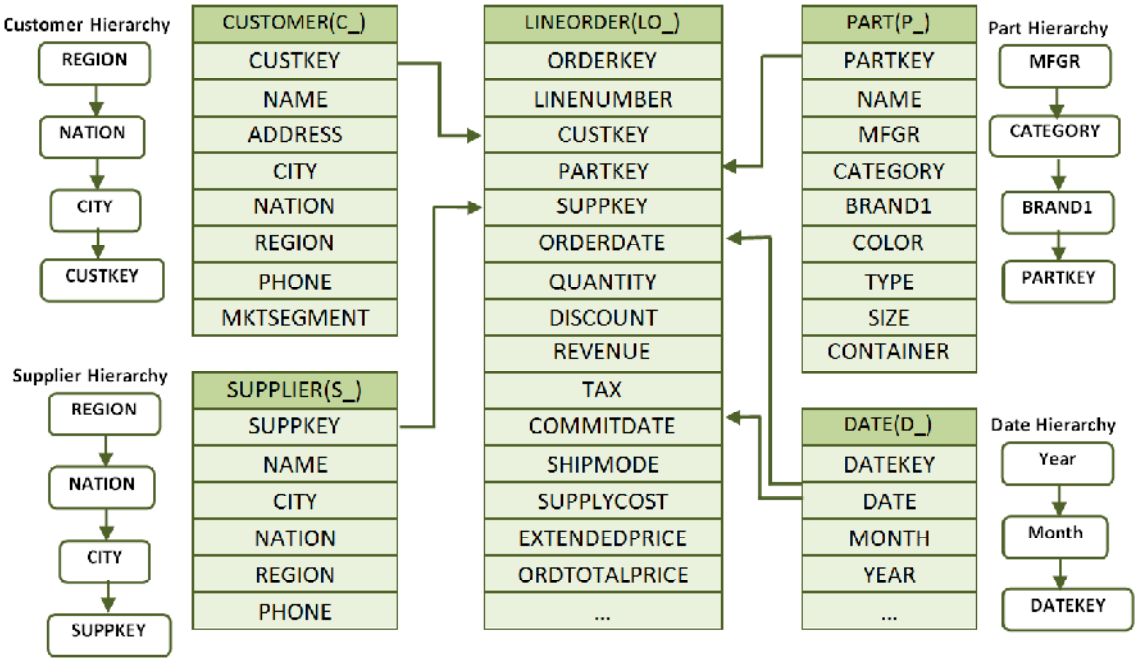


Figure 5.23: The SSB Schema.

Roles. Bob is associated with the Accounting Role only, while Yan is mapped to the Administration Role. The following restrictions/constraints are then added.

- Users within the Marketing Role are restricted from accessing the data of the Part dimension (note that this restricts all Part attributes and attribute values).
- The child Role e_Marketing should prohibit access to North American customer data except that of Canadian customers.
- An additional constraint is added to the Accounting Role, whereby its users are restricted from accessing the data of Asian suppliers, except those in Taiwan.
- The child Role t_Marketing should prevent access to Sales data after the year 1997.

A few additional points are worth noting. First, the policy constraints are specified on the conceptual entities of the data cube, such as dimensions and hierarchical aggregation levels. Second, the constraints themselves restrict some/all data in each entity of the data cube, and therefore every query of the Benchmark will be affected by at least one constraint of this policy. For instance, the first three constraints restrict access to the dimensions Part, Customer, and Supplier, respectively, and the last constraint restricts access to the sales data in the Fact table. Third, the constraints cover the different granularity of the data cube. For example, while the first one restricts the whole dimension (e.g., Part), the second and the third restrict specific values within the data cube with/without exceptions. As a final point, the roles are organized into a hierarchy that enables inheritance. This means that the constraints of the parent roles are inherited by their children, and therefore, the child members are affected by these inherited constraints.

Throughout the previous sections, we discussed the classes and methods that are used to create and manipulate policy objects. In the next subsections, we show how these objects are gathered together to define policies. We also demonstrate how our methods would be invoked in order to maintain policy objects.

5.10.1 Defining the Policy Objects

Listing 5.10.1 illustrates the process of defining policy objects, in which the create methods are used to define instances of various policy constructs such as Subjects, Roles, and Protected Objects. We note that this particular example utilizes the programmatic API (via a C++ library) rather than the SQL-style declarative interface, though either could of course be used by the administrator. The process starts by creating a new subject (e.g., Sue) with the user name and the password Sue and

Su465, respectively. While only one subject is created in the example, other subjects would be created in the same way.

The protected objects are then defined using the createObject() method that specifies the protected dimension, attribute and values. For instance, the first restriction is defined on the Part dimension and named as partDimension — the other restrictions can be defined in like manner. These protected objects are then grouped together and added to a specific role in order to restrict the access of its members. For instance, the partDimension will be added to the role Marketing to restrict its users from accessing the Part dimension data.

```
void main()
{
 //Create the subjects (e.g., Sue)
 Subject Sue;
  try {
       John.createSubject("Sue", "Su465");
  }
  catch(const invalid_argument& e) {
       cerr << "Error: " << e.what( ) << endl;
  }

 //Create the restrictions (e.g., on Part dimension)
  Object partDimension;
 try {
       partDimension.createObject();
  }
  catch(const invalid_argument& e) {
       cerr << "Error: " << e.what( ) << endl;
  }

 //Create the roles (e.g., t_Marketing)
 Role t_marketingRole;
 try {
       t_marketingRole.createRole();
  }
  catch(const invalid_argument& e) {
       cerr << "Error: " << e.what( ) << endl;
```

```
  }

 //The manipulation methods
 //...
}
```

5.10.2 Manipulating the Policy Objects

After creating the different policy objects, the administrator can maintain these objects by using the manipulation methods. Listing 5.10.2 depicts these methods. As shown in statement 1 and 2, the subjects and roles are added to the policy prior to the assignment step, where a subject can be a member of one or more roles. We note that the protected objects can be created during role creation or separately. In this example, the protected objects are created — as per the methods in the previous section — then added to the roles. For instance, the partDimension object — created in Listing 5.10.1 — is added to the Marketing role in statement 3.

After that, the subjects should be assigned to roles in order to to carry out their duties. For instance, in statement 4, Sue is assigned to the role t_Marketing. Note that via her association with t_Marketing, Sue has restricted access to Sales data after the year 1997. However, another restriction is implicitly included (e.g., partDimension) through Role inheritance. Specifically, this restriction is inherited from the Marketing Role (via t_Marketing).

Throughout the lifetime of the system, the administrator can obtain information about Roles, Subjects or Policies. For instance, the policy roles are retrieved in statement 5, then a specific role (e.g., the t_Marketing role) is queried to obtain its memberships (e.g., statement 6). Finally, if the policy changed due to security reasons, a subject can be revoked from a certain role as shown in statement 7. Of

course, this does not affect his/her other roles. However, if a subject is removed from a policy as in statement 8, his/her memberships in all roles in the policy are removed completely.

```
void main()
{
 //The creation methods
 //...

 Policy policy1;
 if(policy1.createPolicy()){

  //1.Add the subject Sue to the policy
  policy1.addSubject(Sue);

  //2.Add the Marketing role to the policy
  policy1.addRole(marketingRole);

  //3.Add the protected object to the role
  marketingRole.addProtectedObject(partDimension);

  //4.Assign Sue to the role t_Marketing
  policy1.assign(Sue,t_marketingRole);

  //5.Obtain the policy roles
  vector <Role> PolicyRoles;
  PolicyRoles = selectPolicyRoles();

  //6.Obtain the role memberships
  vector <Subject> RoleSubjects;
  RoleSubjects = selectRoleSubjects(t_marketingRole.getRoleID());

  //7.Revoke Sue from the role Marketing
  policy1.withdraw(Sue,t_marketingRole);

  //8.Remove the subject Sue from the policy
  policy1.removeSubject(Sue);
}
```

5.11 Conclusions

In this chapter, we have presented a security policy design model called OSSM. OSSM is based on the object oriented paradigm, which is appropriate for managing complex objects and relationships. The basic policy type supported by the model is the Authorization Policy. Such policies specify the relationship between subjects, objects, and conditions. All policies in OSSM are defined over sets of objects created by corresponding classes in an object-oriented manner. Roles, Subjects, and other elements can in turn be defined and reused within policy specifications in order to enhance reusability. Furthermore, our model supports an evolving role hierarchy with possibly overlapping role specifications.

To facilitate the use and proper execution of the OSSM commands, we also provide a small policy engine prototype. The engine hides policy details and associates constraints with the conceptual elements of the data cube model. To illustrate the simplicity of the approach, we have also provided a case study and indicated how policies might be specified in such a scenario. Given the size of the OLAP market, and the importance of properly protecting analytics/warehouse data, we believe that this kind of domain-specific approach represents a significant improvement relative to current alternatives.

Chapter 6

Conclusions

6.1 Summary

In this thesis we have presented a comprehensive solution for the protection of sensitive data within OLAP domains. Ultimately, the algorithms and data structures we have discussed allow users to identify and exploit useful data trends and patterns without concern that the integrity of the underlying data will be compromised. The contributions of the current work are three-fold. First, we have developed a framework to restrict unauthorized users from accessing data that has been explicitly protected. Second, we presented a solution for controlling malicious inferences caused by unprotected coarser aggregations. Third, we enhanced the functionality of our security framework with a flexible Object Oriented design model, as well as providing support for both declarative policy specification and programmatic interaction.

In terms of authorization and authentication, 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 here, 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 in order 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.

In the second part of the thesis, we presented a framework for controlling malicious inferences. We began by describing the framework components and how they interacted with the OLAP query. Specifically, after receiving the query, the DBMS middleware parses the query, identifies the relevant OLAP objects, and re-writes the original query to include a native DBMS representation of the query. In turn, the control component rejects any query that might lead to malicious inference. We then discussed data structures and algorithms utilized by the model in order to provide acceptable performance characteristics during the query checking process. The experimental results demonstrate that the model is efficient and readily implementable within OLAP environments.

Finally, we discussed the issue of design abstractions and associated language support. To that end, we described an object oriented design model that would allow security professional to focus on the specification of policies without concern for physical schema design. Declarative language extensions, as well a direct mapping to the OO design model, were presented in the context of client side programming options. On the server side, we discussed the structure and maintenance of the repository database, including the methods used to address overlapping security roles. A case study utilizing examples from the thesis was also analyzed.

6.2 Future Work

The research described in this thesis represents a foundation for the development of a complete OLAP security system. That being said, this is a large topic and there are numerous opportunities for additional research. Below, we identify a number of possible projects or research themes that would significantly extend the functionality of the current research:

1. **Improve the query language conversion middleware**.

   At present, our query language conversion middleware (i.e., SQL and MDX translators) supports a wide range of common query patterns. However, they do not cover support the full syntax of either language. For example, the SQL conversion module cannot at present translate all forms of nested queries, while the MDX middleware does not support all MDX functions (there are more than 100 functions in the language).

2. **Full Implementation of the IR Converter**.

   The IR Converter is not yet fully implemented. At present, the backend server supports either direct execution of the OLAP algebra (i.e., using Sidera's native query processor) or execution via an SQL compliant DBMS (the MonetDB). In the latter case, valid queries are converted from our internal OLAP algebra to SQL form prior to execution. However, it would be important to extend the IR converter to be able to convert valid queries not only to SQL but to MDX format as well.

3. **Provide security countermeasures for materialized views**.

At present it is conceivable that materialized views could be considered as a mechanism for bypassing security for sensitive data that has been otherwise protected. However, since the query re-writing is performed during the security checking phase, all security restrictions can be associated with common data cube elements (i.e., dimensions, attributes, etc). We can protect the sensitive data when using pre-generated views by executing the query only if a user has access to elements that are used to define the corresponding view.

4. **Consider other kinds of attacks**.

   To maximize data utility, our security framework considers those situations in which inference is associated with the computation of an *exact* sensitive value. However, other methods can be used to infer sensitive data. For example, a classifier can be built based on query results, and then utilized to infer sensitive data[21]. Considering this type of attack could significantly extend the scope of our security framework.

5. **Provide security countermeasures for the collaborative business intelligence**.

   Collaborative environments introduce new requirements for access control, which cannot be met by using existing models developed for non-collaborative domains [74]. The reason is that there are more resources to be accessed and more complex security policies and rules from different partners to be obeyed in such a system than these in an individual information system. This increases the security concerns since, simultaneously cooperating attackers may be existed. Considering this type of attack could motivate the future work research.

In conclusion, we believe that the mechanisms presented in this thesis represent a significant contribution to the literature in that they define a comprehensive data security model specifically tailored to the OLAP domain. Not only have we discussed general solutions to core problems, but we have also tried to emphasize the application and implementation issues relevant to real world settings. While there is, as we have noted above, additional work that can be pursued in the future, we believe that the current work clearly demonstrates that enhanced OLAP security is both useful *and* possible in practice.

Bibliography

[1] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 967–980, New York, NY, USA, 2008. ACM.

[2] Nabil R. Adam and John C. Worthmann. Security-control methods for statistical databases: a comparative study. *ACM Comput. Surv.*, 21(4):515–556, December 1989.

[3] Charu C. Aggarwal and Philip S. Yu. On static and dynamic methods for condensation-based privacy-preserving data mining. *ACM Trans. Database Syst.*, 33(1):2:1–2:39, March 2008.

[4] Rakesh Agrawal, Ramakrishnan Srikant, and Dilys Thomas. Privacy preserving olap. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 251–262, New York, NY, USA, 2005. ACM.

[5] Masoom Alam, Ruth Breu, and Michael Hafner. Model-driven security engineering for trust management in sectet. *Journal of Software*, 2(1):4759, 2007.

[6] Muhammad Alam, Michael Hafner, Ruth Breu, and Stefan Unterthiner. A framework for modeling restricted delegation in service oriented architecture. *Trust and Privacy in Digital Business*, pages 142–151, 2006.

[7] Ahmad Altamimi and Todd Eavis. Securing access to data in business intelligence domains. *International Journal On Advances in Security*, 5(3 and 4):94 – 111, 2012.

[8] OLAP Council: APB-1 OLAP Benchmark, Release ii, 1998. `http://www.olapcouncil.org/research/bmarkly.htm`.

[9] Moritz Y. Becker, Cédric Fournet, and Andrew D. Gordon. Secpal: Design and semantics of a decentralized authorization language. *J. Comput. Secur.*, 18(4):619–665, December 2010.

[10] Moritz Y. Becker, Cédric Fournet, and Andrew D. Gordon. Secpal: Design and semantics of a decentralized authorization language. *J. Comput. Secur.*, 18(4):619–665, December 2010.

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

[12] Carlos Blanco, Ignacio Garcia-Rodriguez de Guzman, David Rosado, Eduardo Fernandez-Medina, and Juan Trujillo. Applying QVT in order to implement secure data warehouses in SQL Server Analysis Services. *Journal of Research and Practice in Information Technology*, 41:135–154, 2009.

[13] Carlos Blanco, Eduardo Fernandez-Medina, Juan Trujillo, and Mario Piattini. How to implement multidimensional security into olap tools. *Int. J. Bus. Intell. Data Min.*, 3(3):255–276, December 2008.

[14] PieroAndrea Bonatti, JuriLuca Coi, Daniel Olmedilla, and Luigi Sauro. Rule-based policy representations and reasoning. In Franois Bry and Jan Mauszyski, editors, *Semantic Techniques for the Web*, volume 5500 of *Lecture Notes in Computer Science*, pages 201–232. Springer Berlin Heidelberg, 2009.

[15] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '98, pages 34–43, New York, NY, USA, 1998. ACM.

[16] Bee-Chung Chen, Kristen LeFevre, and Raghu Ramakrishnan. Privacy skyline: privacy with multidimensional adversarial knowledge. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 770–781. VLDB Endowment, 2007.

[17] Keke Chen and Ling Liu. Privacy preserving data classification with rotation perturbation. In *Data Mining, Fifth IEEE International Conference on*, pages 4 pp.–, 2005.

[18] F.Y. Chin and G. Ozsoyoglu. Auditing and inference control in statistical databases. *Software Engineering, IEEE Transactions on*, SE-8(6):574–582, 1982.

[19] E. F. Codd, S. B. Codd, and C. T. Salley. Providing OLAP (On-Line Analytical Processing) to User-Analysts: An IT Mandate. E. F. Codd and Associates, 1993.

[20] Brian J. Corcoran, Nikhil Swamy, and Michael Hicks. Cross-tier, label-based security enforcement for web applications. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, SIGMOD '09, pages 269–282, New York, NY, USA, 2009. ACM.

[21] Graham Cormode. Personal privacy vs population privacy: learning to attack anonymization. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '11, pages 1253–1261, New York, NY, USA, 2011. ACM.

[22] Lawrence H. Cox. On properties of multi-dimensional statistical tables. *Journal of Statistical Planning and Inference*, 117(2):251 – 273, 2003.

[23] Nicodemos Damianou. *A Policy Framework for Management of Distributed Systems*. PhD thesis, Imperial College London, March 2002.

[24] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. In *Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, POLICY '01, pages 18–38, London, UK, UK, 2001. Springer-Verlag.

[25] The Datamarts, June 2012. `http://www.data-warehouses.net/architecture/datamarts.html`.

[26] Thomas H. Davenport and Jeanne G. Harris. *Competing on Analytics: The New Science of Winning*. Harvard Business School Press, Boston, MA, USA, 1st edition, 2007.

[27] Matteo Dell'Amico, Gabriel Serme, Muhammad Sabir Idrees, Anderson Santana de Olivera, and Yves Roudier. Hipolds: a security policy language for distributed systems. In *Proceedings of the 6th IFIP WG 11.2 international conference on Information Security Theory and Practice: security, privacy and trust in computing systems and ambient intelligent ecosystems*, WISTP'12, pages 97–112, Berlin, Heidelberg, 2012. Springer-Verlag.

[28] D.E. Denning and J. Schlorer. Inference controls for statistical databases. *Computer*, 16(7):69–82, 1983.

[29] D.E. Denning and J. Schlorer. Inference controls for statistical databases. *Computer*, 16(7):69 –82, july 1983.

[30] Dorothy E. Denning. Secure statistical databases with random sample queries. *ACM Trans. Database Syst.*, 5(3):291–315, September 1980.

[31] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1:1–140, 2007.

[32] John DeTreville. Binder, a logic-based security language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, SP '02, pages 105–, Washington, DC, USA, 2002. IEEE Computer Society.

[33] J. Domingo-Ferrer and J. M. Mateo-Sanz. Practical data-oriented microaggregation for statistical disclosure control. *IEEE Trans. on Knowl. and Data Eng.*, 14(1):189–201, January 2002.

[34] Josep Domingo-Ferrer. Advances in inference control in statistical databases: An overview. In Josep Domingo-Ferrer, editor, *Inference Control in Statistical Databases*, volume 2316 of *Lecture Notes in Computer Science*, pages 1–7. Springer Berlin Heidelberg, 2002.

[35] Josep Domingo-Ferrer, Anna Oganian, and Vicenç Torra. Information-theoretic disclosure risk measures in statistical disclosure control of tabular data. In *Proceedings of the 14th International Conference on Scientific and Statistical Database Management*, SSDBM '02, pages 227–231, Washington, DC, USA, 2002. IEEE Computer Society.

[36] Cynthia Dwork. Differential privacy: a survey of results. In *Proceedings of the 5th international conference on Theory and applications of models of computation*, TAMC'08, pages 1–19, Berlin, Heidelberg, 2008. Springer-Verlag.

[37] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. Our data, ourselves: privacy via distributed noise generation. In *Proceedings of the 24th annual international conference on The Theory and Applications of Cryptographic Techniques*, EUROCRYPT'06, pages 486–503, Berlin, Heidelberg, 2006. Springer-Verlag.

[38] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Proceedings of the Third conference on Theory of Cryptography*, TCC'06, pages 265–284, Berlin, Heidelberg, 2006. Springer-Verlag.

[39] Todd Eavis, George Dimitrov, Ivan Dimitrov, David Cueva, Alex Lopez, and Ahmad Taleb. Sidera: a cluster-based server for online analytical processing. In *Proceedings of the 2007 OTM confederated international conference on On the move to meaningful internet systems: CoopIS, DOA, ODBASE, GADA, and IS - Volume Part II*, OTM'07, pages 1453–1472, Berlin, Heidelberg, 2007. Springer-Verlag.

[40] Todd Eavis, Hiba Tabbara, and Ahmad Taleb. The NOX framework: native language queries for business intelligence applications. In *DaWak*, pages 172–189, 2010.

[41] Todd Eavis and Ahmad Taleb. Mapgraph: efficient methods for complex olap hierarchies. In Mrio J. Silva, Alberto H. F. Laender, Ricardo A. Baeza-Yates, Deborah L. McGuinness, Bjrn Olstad, ystein Haug Olsen, and Andr O. Falco, editors, *CIKM*, pages 465–474. ACM, 2007.

[42] Economist Intelligence Unit. `http://www.eiu.com/default.aspx`.

[43] Fernandez-Medina Eduardo, Trujillo Juan, Villarroel Rodolfo, and Piattini Mario. Access control and audit model for the multidimensional modeling of data warehouses. *Decision Support Systems*, 42(3):1270 – 1289, 2006.

[44] Alexandre Evfimievski, Johannes Gehrke, and Ramakrishnan Srikant. Limiting privacy breaches in privacy preserving data mining. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '03, pages 211–222, New York, NY, USA, 2003. ACM.

[45] Adrienne Porter Felt, Matthew Finifter, Joel Weinberger, and David Wagner. Diesel: applying privilege separation to database access. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 416–422, New York, NY, USA, 2011. ACM.

[46] Eduardo Fernández-Medina, Juan Trujillo, Rodolfo Villarroel, and Mario Piattini. Developing secure data warehouses with a uml extension. *Inf. Syst.*, 32(6):826–856, September 2007.

[47] Enrico Franconi and Anand Kamble. The GMD data model and algebra for multidimensional information. In Anne Persson and Janis Stirna, editors, *Advanced Information Systems Engineering*, volume 3084 of *Lecture Notes in Computer Science*, pages 446–462. Springer Berlin Heidelberg, 2004.

[48] Arik Friedman and Assaf Schuster. Data mining with differential privacy. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '10, pages 493–502, New York, NY, USA, 2010. ACM.

[49] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, February 1992.

[50] Dieter Gollmann. *Computer Security, 3rd ed.* Wiley Publishing, 2011.

[51] Patricia P. Griffiths and Bradford W. Wade. An authorization mechanism for a relational database system. *ACM Trans. Database Syst.*, 1(3):242–255, September 1976.

[52] Skogsrud Halvard, R. Motahari-Nezhad Hamid, Benatallah Boualem, and Casati Fabio. Modeling trust negotiation for web services. *Computer*, 42(2):54–61, 2009.

[53] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, SIGMOD '96, pages 205–216, New York, NY, USA, 1996. ACM.

[54] William Inmon. *Building the Data Warehouse*. John Wiley and Sons, 2005.

[55] Ian Jacobi, Lalana Kagal, and Ankesh Khandelwal. Rule-based trust assessment on the semantic web. In Nick Bassiliades, Guido Governatori, and Adrian Paschke, editors, *Rule-Based Reasoning, Programming, and Applications*, volume 6826 of *Lecture Notes in Computer Science*, pages 227–241. Springer Berlin Heidelberg, 2011.

[56] Sonia Jahid, Carl A. Gunter, Imranul Hoque, and Hamed Okhravi. Myabdac: compiling xacml policies for attribute-based database access control. In *Proceedings of the first ACM conference on Data and application security and privacy*, CODASPY '11, pages 97–108, New York, NY, USA, 2011. ACM.

[57] Han Jiawei, Kamber Micheline, and Pei Jian. *Data Mining: Concepts and Techniques, 3rd Edition*. The Morgan Kaufmann Series in Data Management Systems, Jim Gray, Series Editor Morgan Kaufmann Publishers, 2011.

[58] Gray Jim, Bosworth Adam, Layman Andrew, Reichart Don, and Pirahesh Hamid. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1:29–53, 1997.

[59] Strassner John and Schleimer Stephen. Policy framework definition language. In *Internet draft draft-ietf-policy-framework-pfdl-00.txt*, Nov 1998.

[60] Trujillo Juan, Soler Emilio, Fernndez-Medina Eduardo, and Piattini Mario. An engineering process for developing secure data warehouses. *Information and Software Technology*, 51(6):1033 – 1051, 2009.

[61] Trujillo Juan, Soler Emilio, Fernndez-Medina Eduardo, and Piattini Mario. A UML 2.0 profile to define security requirements for data warehouses. *Computer Standards and Interfaces*, 31(5):969 – 983, 2009.

[62] Govind Kabra, Ravishankar Ramamurthy, and S. Sudarshan. Redundancy and information leakage in fine-grained access control. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 133–144, New York, NY, USA, 2006. ACM.

[63] N. Katic, G. Quirchmayr, J. Schiefer, M. Stolba, and A. M. Tjoa. A prototype model for data warehouse security based on metadata. In *Proceedings of the 9th International Workshop on Database and Expert Systems Applications*, DEXA '98, pages 300–, Washington, DC, USA, 1998. IEEE Computer Society.

[64] Krishnaram Kenthapadi, Nina Mishra, and Kobbi Nissim. Simulatable auditing. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '05, pages 118–127, New York, NY, USA, 2005. ACM.

[65] Krishna Khajaria and Manoj Kumar. Modeling of security requirements for decision information systems. *SIGSOFT Softw. Eng. Notes*, 36(5):1–4, September 2011.

[66] Krishna Khajaria and Manoj Kumar. Modeling of security requirements for decision information systems. *SIGSOFT Softw. Eng. Notes*, 36(5):1–4, September 2011.

[67] Eugene Kindler and Ivan Kriv. Object-oriented simulation of systems with sophisticated control. *Int. J. General Systems*, 40(3):313–343, 2011.

[68] Jon Kleinberg, Christos Papadimitriou, and Prabhakar Raghavan. Auditing boolean attributes. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '00, pages 86–91, New York, NY, USA, 2000. ACM.

[69] Lefevre Kristen, Agrawal Rakesh, Ercegovac Vuk, Ramakrishnan Raghu, Xu Yirong, and Dewitt David. Limiting disclosure in hippocratic databases. In *In Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, pages 108–119, 2004.

[70] Kristen LeFevre, David J. DeWitt, and Raghu Ramakrishnan. Workload-aware anonymization techniques for large-scale datasets. *ACM Trans. Database Syst.*, 33(3):17:1–17:47, September 2008.

[71] Gregory Leighton. Preserving sql access control policies over published xml data. In *Proceedings of the 2009 EDBT/ICDT Workshops*, EDBT/ICDT '09, pages 185–192, New York, NY, USA, 2009. ACM.

[72] Yingjiu Li, Haibing Lu, and Robert H. Deng. Practical inference control for data cubes (extended abstract). In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 115–120, Washington, DC, USA, 2006. IEEE Computer Society.

[73] Jeff Linwood and Dave Minter. *Beginning Hibernate*. Apress, 2nd edition, 2010.

[74] Qiang Liu, Xinhui Zhang, Xindu Chen, and Lei Wang. The resource access authorization route problem in a collaborative manufacturing system. *Journal of Intelligent Manufacturing*, pages 1–13, 2012.

[75] Suping Liu and Yongsheng Ding. An adaptive network policy management framework based on classical conditioning. In *Intelligent Control and Automation, 2008. WCICA 2008. 7th World Congress on*, pages 3336–3340, 2008.

[76] Suping Liu and Yongsheng Ding. A classical conditioning model for policy-based management. In *Networks Security, Wireless Communications and Trusted Computing, 2009. NSWCTC '09. International Conference on*, volume 1, pages 249–252, 2009.

[77] Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, and Muthuramakrishnan Venkitasubramaniam. L-diversity: Privacy beyond k-anonymity. *ACM Trans. Knowl. Discov. Data*, 1(1), March 2007.

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

[79] F.M. Malvestuto and M. Moscarini. Censoring statistical tables to protect sensitive information: easy and hard problems. In *Scientific and Statistical Database Systems, 1996. Proceedings., Eighth International Conference on*, pages 12–21, 1996.

[80] F.M. Malvestuto and M. Moscarini. Computational issues connected with the protection of sensitive statistics by auditing sum-queries. In *Scientific and Statistical Database Management, 1998. Proceedings. Tenth International Conference on*, pages 134–144, 1998.

[81] Pentaho Analysis Services (Mondrian), June 2012. `http://mondrian.pentaho.com`.

[82] Kumar Manoj, Gosain Anjana, and Singh Yogesh. Stakeholders driven requirements engineering approach for data warehouse development. *JIPS*, 6(3):385–402, 2010.

[83] V. Markl and Rudolf Bayer. Processing relational olap queries with ub-trees and multidimensional hierarchical clustering. In *IN PROCEEDINGS OF DMDW 2000*, pages 5–6, 2000.

[84] D.J. Martin, D. Kifer, A. Machanavajjhala, J. Gehrke, and Joseph Y. Halpern. Worst-case background knowledge for privacy-preserving data publishing. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 126–135, 2007.

[85] Scott Michael Lee. *Programming language pragmatics*. Morgan Kaufmann, 3rd edition, 2009.

[86] Noman Mohammed, Rui Chen, Benjamin C.M. Fung, and Philip S. Yu. Differentially private data release for data mining. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 493–501, New York, NY, USA, 2011. ACM.

[87] Shubha U. Nabar, Bhaskara Marthi, Krishnaram Kenthapadi, Nina Mishra, and Rajeev Motwani. Towards robustness in query auditing. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 151–162. VLDB Endowment, 2006.

[88] Aarthi Nagarajan, Vijay Varadharajan, and Michael Hitchens. Alopa: Authorization logic for property attestation in trusted platforms. In *Proceedings of the 6th International Conference on Autonomic and Trusted Computing*, ATC '09, pages 134–148, Berlin, Heidelberg, 2009. Springer-Verlag.

[89] OASIS. eXtensible Access Control Markup Language (XACML) Version 3.0, Jan 2013. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml.

[90] E. OLSON. Reflective database access control. *In PhD. Thesis. University of Illinois at Urbana-Champaign*, 2009.

[91] Lars E. Olson, Carl A. Gunter, and P. Madhusudan. A formal framework for reflective database access control policies. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 289–298, New York, NY, USA, 2008. ACM.

[92] Patrick ONeil, Elizabeth ONeil, Xuedong Chen, and Stephen Revilak. The star schema benchmark and augmented fact table indexing. In Raghunath Nambiar and Meikel Poess, editors, *Performance Evaluation and Benchmarking*, volume 5895 of *Lecture Notes in Computer Science*, pages 237–252. Springer Berlin Heidelberg, 2009.

[93] Hyoungmin Park and Kyuseok Shim. Approximate algorithms for k-anonymity. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 67–78, New York, NY, USA, 2007. ACM.

[94] Boncz Peter, Zukowski Marcin, and Nes Niels. Monetdb/x100: Hyper-pipelining query execution, 2005.

[95] http://www.ponemon.org/.

[96] Shariq Rizvi, Alberto Mendelzon, S. Sudarshan, and Prasan Roy. Extending query rewriting techniques for fine-grained access control. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 551–562, New York, NY, USA, 2004. ACM.

[97] Oscar Romero and Alberto Abelló. On the need of a reference algebra for olap. In *Proceedings of the 9th International Conference on Data Warehousing and Knowledge Discovery*, DaWaK'07, pages 99–110, Berlin, Heidelberg, 2007. Springer-Verlag.

[98] A. Rosenthal, E. Sciore, and Arnon Rosenthal. View security as the basis for data warehouse security. In *CAiSE Workshop on Design and Management of Data Warehouses*, pages 5–6, 2000.

[99] Arnon Rosenthal and Edward Sciore. Administering permissions for distributed data: Factoring and automated inference. In *Proceedings of the Fifteenth Annual Working Conference on Database and Application Security*, Das'01, pages 91–104, Norwell, MA, USA, 2002. Kluwer Academic Publishers.

[100] Y. Sung Sam, Liu Yao, Xiong Hui, and A. Ng Peter. Privacy preservation for data cubes. *Knowledge and Information Systems*, 9(1):38–61, January 2006.

[101] Ravi Sandhu, David Ferraiolo, and Richard 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*, RBAC '00, pages 47–63, New York, NY, USA, 2000. ACM.

[102] Dwivedi Santosh, Menezes Bernard, and Singh Ashish. Database access control for e-business a case study. In *Proceeedings of the 11th International Conference on Management of Data COMAD*, 2005.

[103] Q.Z. Sheng, Jian Yu, Z. Maamar, Wei Jiang, and Xitong Li. Compatibility checking of heterogeneous web service policies using vdm++. In *Services - I, 2009 World Conference on*, pages 821–828, 2009.

[104] Kathy Sierra and Bates Bert. *Head First Java*. O'Reilly Media, Inc., 2nd edition, 2005.

[105] Indu Singh and Manoj Kumar. Evaluation of approaches for designing secure data warehouse. In *Proceedings of the International Conference on Advances in Computing, Communications and Informatics*, ICACCI '12, pages 69–73, New York, NY, USA, 2012. ACM.

[106] E. Soler, J. Trujillo, E. Fernandez-Medina, and M. Piattini. Application of qvt for the development of secure data warehouses: A case study. In *Availability, Reliability and Security, 2007. ARES 2007. The Second International Conference on*, pages 829–836, 2007.

[107] SQL database engine, June 2012. `http://www.sqlite.org`.

[108] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: a column-oriented dbms. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 553–564. VLDB Endowment, 2005.

[109] Latanya Sweeney. k-anonymity: a model for protecting privacy. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 10(5):557–570, October 2002.

[110] Ahmad Taleb, Todd Eavis, and Hiba Tabbara. The nox olap query model: from algebra to execution. In *Proceedings of the 13th international conference on Data warehousing and knowledge discovery*, DaWaK'11, pages 167–183, Berlin, Heidelberg, 2011. Springer-Verlag.

[111] Gianluca Tonti, JeffreyM. Bradshaw, Renia Jeffers, Rebecca Montanari, Niranjan Suri, and Andrzej Uszok. Semantic web languages for policy representation and reasoning: A comparison of kaos, rei, and ponder. In Dieter Fensel, Katia Sycara, and John Mylopoulos, editors, *The Semantic Web - ISWC 2003*, volume 2870 of *Lecture Notes in Computer Science*, pages 419–437. Springer Berlin Heidelberg, 2003.

[112] Vicen Torra. Microaggregation for categorical variables: A median based approach. In Josep Domingo-Ferrer and Vicen Torra, editors, *Privacy in Statistical Databases*, volume 3050 of *Lecture Notes in Computer Science*, pages 162–174. Springer Berlin Heidelberg, 2004.

[113] J. F. Traub, Y. Yemini, and H. Woźniakowski. The statistical security of a statistical database. *ACM Trans. Database Syst.*, 9(4):672–679, December 1984.

[114] tree.hh: an STL-like C++ tree class, June 2012. `http://www.tree.phi-sci.com`.

[115] Kevin P. Twidle, Naranker Dulay, Emil Lupu, and Morris Sloman. Ponder2: A policy system for autonomous pervasive environments. In Radu Calinescu, Fidel Liberal, Mauricio Marn, Lourdes Pealver Herrero, Carlos Turro, and Manuela Popescu, editors, *ICAS*, pages 330–335. IEEE Computer Society, 2009.

[116] Wynand JC van Staden and Martin S Olivier. Sql's revoke with a view on privacy. In *Proceedings of the 2007 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, SAICSIT '07, pages 181–188, New York, NY, USA, 2007. ACM.

[117] Sabrina De Capitani Di Vimercati, Sara Foresti, Pierangela Samarati, and Sushil Jajodia. Access control policies and languages. *Int. J. Comput. Sci. Eng.*, 3(2):94–102, November 2007.

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

[119] Lingyu Wang, S. Jajodia, and D. Wijesekera. Securing olap data cubes against privacy breaches. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, pages 161 – 175, may 2004.

[120] Lingyu Wang, Yingjiu Li, Duminda Wijesekera, and Sushil Jajodia. Precisely answering multi-dimensional range queries without privacy breaches. In Einar Snekkenes and Dieter Gollmann, editors, *Computer Security ESORICS 2003*, volume 2808 of *Lecture Notes in Computer Science*, pages 100–115. Springer Berlin Heidelberg, 2003.

[121] Lingyu Wang, Duminda Wijesekera, and Sushil Jajodia. Cardinality-based inference control in sum-only data cubes. In *Proceedings of the 7th European Symposium on Research in Computer Security*, ESORICS '02, pages 55–71, London, UK, UK, 2002. Springer-Verlag.

[122] Lingyu Wang, Duminda Wijesekera, and Sushil Jajodia. Cardinality-based inference control in data cubes. *J. Comput. Secur.*, 12(5):655–692, September 2004.

[123] Qihua Wang, Ting Yu, Ninghui Li, Jorge Lobo, Elisa Bertino, Keith Irwin, and Ji-Won Byun. On the correctness criteria of fine-grained access control in relational databases. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 555–566. VLDB Endowment, 2007.

[124] Leon Willenborg and Ton De Waal. *Statistical disclosure control in practice*. Number 111 in Lecture Notes in Statistics. Springer, 1996.

[125] WS-Policy.Web services policy framework (WS-Policy) Version 1.5, Nov 2010. `http://www.ibm.com/developerworks/library/j-jws18/`.

[126] Mariemma Yagüe. Survey on xml-based policy languages for open environments. *Journal of Information Assurance and Security*, 1(1):11–20, 2006.

[127] Singh Yogesh, Gosain Anjana, and Kumar Manoj. From early requirements to late requirements modeling for a data warehouse. *Networked Computing and*

Advanced Information Management, International Conference on, 0:798–804, 2009.

[128] Morteza Zaker, Somnuk Phon-amnuaisuk, and Su cheng Haw. An adequate design for large data warehouse systems: Bitmap index versus b-tree index, 2008.

[129] Nan Zhang, Wei Zhao, and Jianer Chen. Cardinality-based inference control in olap systems: An information theoretic approach. In *Proceedings of the 7th ACM International Workshop on Data Warehousing and OLAP*, DOLAP '04, pages 59–64, New York, NY, USA, 2004. ACM.

[130] Zhang Zheng and O. Mendelzon Alberto. Authorization views and conditional query containment. In *In Database Theory - ICDT 2005, 10th International Conference*, pages 259–273, 2005.

Appendix A

The Star Schema Benchmark Queries

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

- Query1:

  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;

- Query1_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;

- Query1_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;

- Query2:

  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;

- Query2_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;

- Query2_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;

- Query3:

  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 between 1992 and 1997
GROUP BY c_city, s_city, d_year
ORDER BY d_year asc, revenue desc;

- Query3_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 $\geq$ 1992 and d_year $\leq$ 1997
  GROUP BY c_nation, s_nation, d_year
  ORDER BY d_year asc, revenue desc;

- Query3_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 $\geq$ 1992 and d_year $\leq$ 1997
  GROUP BY c_city, s_city, d_year
  ORDER BY d_year asc, revenue desc;

- Query3_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;

- Query4:

  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;

- Query4_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;

- Query4_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;

Appendix B

The Complementary Query Set

Below, we provide the complementary set of 6 queries.

- Query5:

  SELECT distinct d_year
  FROM supplier, date, lineorder
  WHERE lo_suppkey = s_suppkey and lo_orderdate = d_datekey and s_suppkey < 120

  UNION

  SELECT distinct d_year
  FROM customer, date , lineorder
  WHERE lo_custkey = c_custkey and lo_orderdate = d_datekey and c_custkey < 300 ;

- Query6:

  SELECT d_year, s_nation, sum(lo_revenue) as profit
  FROM date, customer, supplier, lineorder
  WHERE lo_custkey = c_custkey and lo_suppkey = s_suppkey and lo_orderdate = d_datekey and c_region = AMERICA and s_region = AMERICA and (d_year

= 1997 or d_year = 1998)
GROUP BY d_year, s_nation

UNION

SELECT d_year, c_nation, sum(lo_revenue) as profit
FROM date, customer, lineorder
WHERE lo_custkey = c_custkey and lo_orderdate = d_datekey and c_region =
AMERICA and (d_year = 1997 or d_year = 1998)
GROUP BY d_year, c_nation;

- Query7:

SELECT distinct c_nation
FROM customer
WHERE c_region <> ASIA and c_region in
(SELECT distinct s_region
FROM customer
WHERE s_suppkey = 10905);

- Query8:

SELECT c_nation, sum(lo_quantity)
FROM part, customer, lineorder
WHERE p_partkey = lo_partkey and c_custkey = lo_custkey and
c_region = AMERICA and p_brand1 <> MFGR#111 and p_partkey =
(SELECT sum(lo_partkey)
FROM lineorder
WHERE lo_custkey = 100) CROUP BY c_nation;

- Query9:

SELECT distinct p_name
FROM part, supplier, lineorder

WHERE p_partkey = lo_partkey and s_suppkey = lo_suppkey and s_region = AMERICA and

p_partkey not in

(SELECT distinct lo_partkey

FROM lineorder

WHERE lo_orderdate > 19980801);

- Query10:

SELECT distinct s_nation

FROM supplier, lineorder

WHERE lo_suppkey = s_suppkey and lo_custkey in

(SELECT c_custkey

FROM customer

WHERE c_city = 'UNITEDST9')

UNION

SELECT distinct c_nation

FROM customer, lineorder

WHERE lo_custkey = c_custkey and lo_orderdate > 19980801;

Appendix C

A Query in XML format and its DTD grammar

The XML encoding of the query depicted in Listing 2.1.

```
<?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>
     </PROJECTION_DIMENSION>
    </ATTRIBUTE_LIST>
   </PROJECTION>
  </OPERATION>

  <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>
         </EXP_VALUE>
        </SIMPLE_EXP>
                  </BASIC_EXP>
      </RELATIONAL_EXP>
     </EXPRESSION>
    </DIMENSION>
  </DIMENSION_LIST>

  <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>
                  </EXP_VALUE>
                </SIMPLE_EXP>
              </BASIC_EXP>
            </RELATIONAL_EXP>
          </EXPRESSION>
        </DIMENSION>
      </DIMENSION_LIST>
    </COMPOUND_DIMENSION>
  </DIMENSION_LIST>
 </SELECTION>
 </OPERATION>

 </OPERATION_LIST>
</DATA_QUERY>


 <USER_CREDENTIALS>
  <USER_NAME>John</USER_NAME>
  <PASSWORD>J86mn</PASSWORD>
 </USER_CREDENTIALS>

</QUERY>
```

The DTD grammar that is used to describe the structure of the previous XML query.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT QUERY ((DATA_QUERY | META_QUERY), USER)>

<!--User-->
```

```
<!ELEMENT USER (USER_NAME, PASSWORD)>
<!ELEMENT USER_NAME (#PCDATA)>
<!ELEMENT PASSWORD (#PCDATA)>

<!-- Data queries -->
<!ELEMENT DATA_QUERY
   (CUBE_NAME,OPERATION_LIST,FUNCTION_LIST?)>
<!ELEMENT CUBE_NAME (#PCDATA)>
<!ELEMENT OPERATION_LIST (OPERATION+)>
<!ELEMENT OPERATION (
 SELECTION |
 PROJECTION |
 CHANGE_LEVEL |
 CHANGE_BASE |
 PIVOT |
 DRILL_ACROSS|
 UNION |
 INTERSECTION |
 DIFFERENCE)>

<!-- Selection -->
<!ELEMENT SELECTION (DIMENSION_LIST)>
<!ELEMENT DIMENSION_LIST ( DIMENSION | COMPOUND_DIMENSION)>
<!ELEMENT COMPOUND_DIMENSION (DIMENSION_LIST, LOGICAL_OP,
   DIMENSION_LIST)>
<!ELEMENT DIMENSION (DIMENSION_NAME, EXPRESSION)>
<!ELEMENT DIMENSION_NAME (#PCDATA)>


<!-- Projection -->
<!ELEMENT PROJECTION (MEASURE_LIST, ATTRIBUTE_LIST?)>
<!ELEMENT MEASURE_LIST (MEASURE+)>
<!ELEMENT ATTRIBUTE_LIST (PROJECTION_DIMENSION+)>
<!ELEMENT PROJECTION_DIMENSION (DIMENSION_NAME, ATTRIBUTE)>
<!ELEMENT MEASURE (#PCDATA)>


<!-- Rollup/Drill down -->
<!ELEMENT CHANGE_LEVEL (CHANGE_LEVEL_LIST+)>
<!ELEMENT CHANGE_LEVEL_LIST (DIMENSION_NAME, TARGET_LEVEL)>
```

```
<!ATTLIST CHANGE_LEVEL_LIST
 direction (UP | DOWN) #REQUIRED>
<!ELEMENT TARGET_LEVEL (#PCDATA)>

<!-- Changing the base -->
<!ELEMENT CHANGE_BASE (CHANGE_BASE_LIST+)>
<!ELEMENT CHANGE_BASE_LIST (PROJECTION_DIMENSION)>
<!ATTLIST CHANGE_BASE_LIST
 modification (ADD | REMOVE) #REQUIRED>

<!-- Pivot -->
<!ELEMENT PIVOT (PIVOT_LIST)>
<!ELEMENT PIVOT_LIST (PIVOT_PAIR+)>
<!ELEMENT PIVOT_PAIR (OLD_DIMENSION, NEW_DIMENSION)>
<!ELEMENT OLD_DIMENSION (#PCDATA)>
<!ELEMENT NEW_DIMENSION (#PCDATA)>

<!-- Drill across -->
<!ELEMENT DRILL_ACROSS (DATA_QUERY, COMPARE_FACT?)>
<!ELEMENT COMPARE_FACT (#PCDATA)>
<!-- RATIO -->

<!-- <!ATTLIST DRILL_ACROSS
 output (BOTH | REPLACE) #REQUIRED> -->

<!-- Union -->
<!ELEMENT UNION (DATA_QUERY)>

<!-- Intersection -->
<!ELEMENT INTERSECTION (DATA_QUERY)>

<!-- Difference -->
<!ELEMENT DIFFERENCE (DATA_QUERY)>


<!-- Dimension Expressions -->
<!ELEMENT EXPRESSION (RELATIONAL_EXP | COMPOUND_EXP)>
<!ELEMENT COMPOUND_EXP (EXPRESSION, LOGICAL_OP, EXPRESSION)>
<!ELEMENT RELATIONAL_EXP (BASIC_EXP | OLAP_EXP)>
<!ELEMENT BASIC_EXP (SIMPLE_EXP, COND_OP, SIMPLE_EXP)>
```

```
<!ELEMENT OLAP_EXP (SIMPLE_EXP, OLAP_OP, OLAP_LIST)>
<!ELEMENT SIMPLE_EXP (EXP_VALUE | ARITHMETIC_EXP)>
<!ELEMENT ARITHMETIC_EXP (SIMPLE_EXP, ARITHMETIC_OP,
    SIMPLE_EXP)>


<!ELEMENT EXP_VALUE (
 CONSTANT |
 ATTRIBUTE |
 FUNCTION_LIST)>

<!ELEMENT CONSTANT (#PCDATA)>
<!ELEMENT ATTRIBUTE (#PCDATA)>

<!-- Dimension Operators -->
<!ELEMENT LOGICAL_OP (#PCDATA)>
<!-- AND | OR -->

<!ELEMENT COND_OP (
 RELATIONAL_OP |
 EQUALITY_OP )>

<!ELEMENT RELATIONAL_OP (#PCDATA)>
<!-- GT | GTE | LT | LTE) -->

<!ELEMENT EQUALITY_OP (#PCDATA)>
<!-- EQUALS | NOT_EQUAL -->

<!ELEMENT OLAP_OP (#PCDATA)>
<!-- IN_RANGE | IN_LIST)> -->

<!ELEMENT OLAP_LIST (VALUE+)>
<!ELEMENT VALUE (#PCDATA)>

<!ELEMENT ARITHMETIC_OP (#PCDATA)>
<!-- ADD | SUBTRACT | MULTIPLY | DIVIDE) -->


<!-- Generic Functions -->
<!ELEMENT FUNCTION_LIST (FUNCTION+)>
```

```
<!ELEMENT FUNCTION (PARENT, FUNCTION_NAME, ARGUMENT_LIST?)>
<!ELEMENT PARENT (#PCDATA)>
<!ELEMENT FUNCTION_NAME (#PCDATA)>
<!ELEMENT ARGUMENT_LIST (ARGUMENT+)>
<!ELEMENT ARGUMENT (#PCDATA)>


<!-- Meta data queries: this will be extended later -->
<!ELEMENT META_QUERY (CUBE_NAME)>
<!ATTLIST META_QUERY
 scale (FULL | PARTIAL) #REQUIRED>
```

Appendix D

The APB-1 OLAP Benchmark Release II Queries

Below, we provide a listing of the 10 queries found in the APB-1 OLAP Benchmark Release II.

- Query1:

  SELECT division_level, retailer_level, base_level, year_level, sum (unitssold) as units, sum (dollarsales) as Dollars, sum (dollarsales)/sum (unitssold) as AvgSellingPrice
  FROM actvars, prodlevel, timelevel, custlevel, chanlevel
  WHERE product_level = code_level and time_level = month_level and customer_level = store_level and channel_level = base_level and channel_level = 'Y63A674WLAGL'
  GROUP BY division_level, retailer_level, base_level, year_level
  ORDER BY division_level, retailer_level, base_level, year_level;

- Query2:

  SELECT division_level, retailer_level, year_level, sum (unitssold) as units, sum (dollarsales) as dollarsales, sum (dollarcost) as dollarcost, sum (dollarsales)-sum (dollarcost) as margin, (sum (dollarsales)-sum (dollarcost))/sum (dollarsales) as

marginpct
FROM actvars, prodlevel, timelevel, custlevel
WHERE product_level = code_level and time_level = month_level and customer_level = store_level and division_level <> 'PNLY9LT0CW24' and retailer_level > 'RODM4G1OLU1P' and year_level = 1995
GROUP BY division_level, retailer_level, year_level
ORDER BY division_level, retailer_level, year_level;

- Query3:

  SELECT class_level, retailer_level, month_level, sum (unitssold) as Units, sum (dollarsales) as DollarSales, sum (DollarCost) as DollarCost
  FROM actvars, prodlevel, timelevel, custlevel
  WHERE product_level = code_level and time_level = month_level and customer_level = store_level and class_level = 'B48F3WC38AAM' and retailer_level = 'RODM4G1OLU1P'
  GROUP BY class_level, retailer_level, month_level
  ORDER BY class_level, retailer_level, month_level;

- Query4:

  SELECT class_level, retailer_level, quarter_level, base_level, sum (dollarsales) as DollarSales
  FROM actvars, prodlevel, custlevel, timelevel, chanlevel
  WHERE product_level = code_level and time_level = month_level and customer_level = store_level and channel_level = base_level and class_level < 'B48F3WC38AAM' and retailer_level = 'RODM4G1OLU1P' and quarter_level $\geq$ '1996Q1'
  GROUP BY class_level, retailer_level, quarter_level, base_level
  ORDER BY class_level, retailer_level, quarter_level, base_level;

- Query5:

SELECT class_level, retailer_level, quarter_level, sum (dollarsales) as DollarSales
FROM actvars, prodlevel, custlevel, timelevel, chanlevel
WHERE product_level = code_level and time_level = month_level and customer_level = store_level and channel_level = base_level and class_level < 'B48F3WC38AAM' and retailer_level > 'RODM4G1OLU1P' and quarter_level ≥ '1996Q1'
GROUP BY class_level, retailer_level, quarter_level
ORDER BY class_level, retailer_level, quarter_level;

- Query6:

SELECT class_level, retailer_level, quarter_level, sum (dollarsales)
FROM actvars, prodlevel, custlevel, timelevel, chanlevel
WHERE product_level = code_level and time_level = month_level and customer_level = store_level and channel_level = chanlevel.base_level and class_level < 'CW1CAWGGW0ZK' and retailer_level <> 'FV37I72FDIG5' and quarter_level ≥ '1995Q1'
GROUP BY class_level, retailer_level, quarter_level
ORDER BY class_level, retailer_level, quarter_level;

- Query7:

SELECT class_level, retailer_level, base_level, quarter_level, 'Actual',
sum (unitssold) as Units
FROM actvars, prodlevel, timelevel, custlevel, chanlevel
WHERE product_level = code_level and time_level = month_level and customer_level = store_level and channel_level = base_level and class_level < 'B48F3WC38AAM' and retailer_level > 'RODM4G1OLU1P' and base_level = 'JRSPFIZ0Q93N' and quarter_level ≥ '1996Q1'
GROUP BY class_level, retailer_level, base_level, quarter_level
ORDER BY class_level, retailer_level, base_level, quarter_level;

- Query8:

  SELECT retailer_level, month_level, sum (unitssold) as UnitsSold, sum (dollarsales) as DollarSales, sum (dollarsales)/sum (unitssold) as AvgSellingPrice, sum (DollarCost) as DollarCost, sum (dollarsales)- sum (DollarCost) as Margin
  FROM planvars, timelevel, custlevel
  WHERE time_level = month_level and customer_level = store_level and retailer_level = 'RODM4G1OLU1P'
  GROUP BY retailer_level, month_level
  ORDER BY retailer_level, month_level;

- Query9:

  SELECT code_level, quarter_level, sum (unitssold) as UnitsSold, sum (dollarsales) as DollarSales, sum (dollarsales)/sum (unitssold) as AvgSellingPrice, sum (DollarCost) as DollarCost, sum (dollarsales)- sum (DollarCost) as Margin-Dollars
  FROM planvars, timelevel, prodlevel
  WHERE time_level = month_level and product_level = code_level and quarter_level $\geq$ '1996Q1' and code_level > 'UC0AYD4861N6'
  GROUP BY code_level, quarter_level
  ORDER BY code_level, quarter_level;

- Query10:

  SELECT class_level, retailer_level, year_level, sum (unitssold) as UnitsSold
  FROM planvars, timelevel, prodlevel, custlevel
  WHERE time_level = month_level and product_level = code_level and customer_level = store_level and class_level < 'B48F3WC38AAM' and retailer_level = 'RODM4G1OLU1P' and year_level = 1996
  GROUP BY class_level, retailer_level, year_level;

Appendix E

The Methods Syntaxes

Below, we provide a listing of the method syntax illustrated in Chapter 5.

- The CreateSubject() Method: Figure E.1 shows the syntax diagrams for this method. The first syntax diagram starts with the method header followed by some optional variable declarations and ends with the return statement. The method header shows the return type, which is a boolean value, followed by the method name and two parameters (e.g., *subjectName* and *password*). The parameters subjectName and password, respectively, determine the name of the new subject account and its password. These parameters are used to identify each subject in the system uniquely. Finally, the last syntax diagram returns true if the subject is created successfully, and false otherwise.

- The CreateObject() Method: The syntax of the method is shown in Figure E.2, which is similar to the syntax of the CreateSubject() method. However, the variable declarations in this method are mandatory. As shown in the third syntax diagram, the protected data and its exception are defined via ProtectedObject and ObjectException variables, respectively. Together, they represent the deduced conditions required to explicitly state the protected element.

- The CreateRole() Method: Figure E.3 depicts the syntax diagram for the

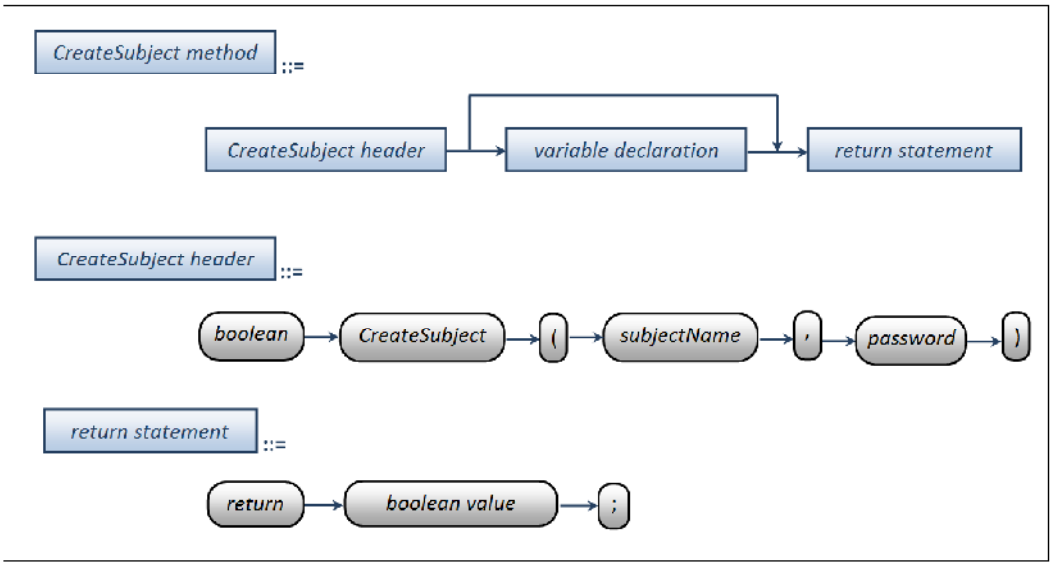


Figure E.1: The Syntax Diagram for the CreateSubject method

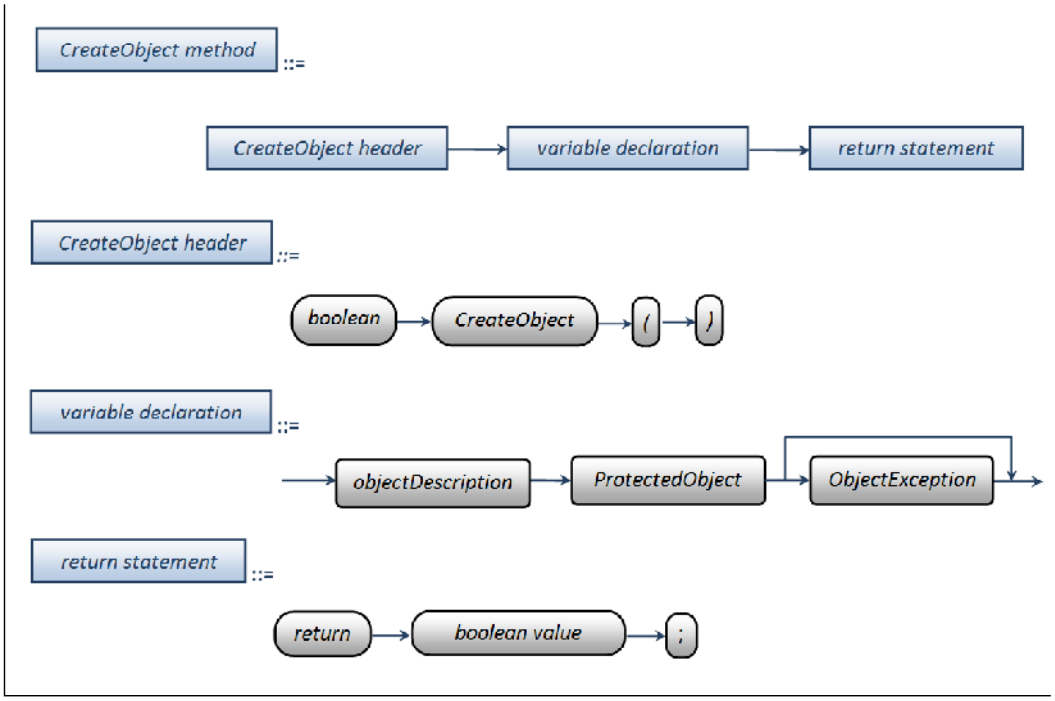


Figure E.2: The Syntax Diagram for the CreateObject method

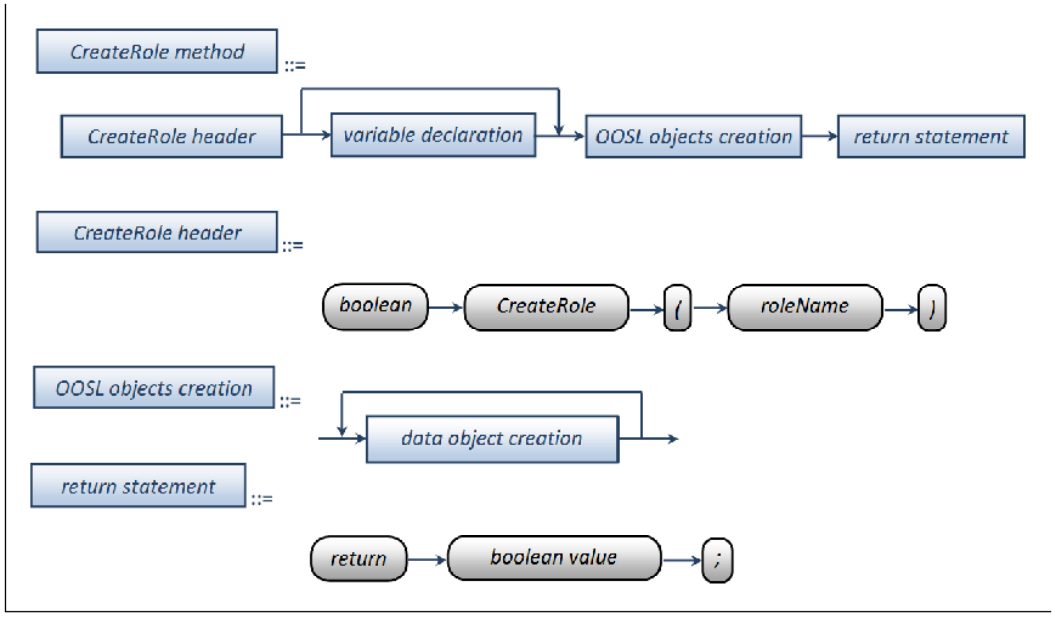


Figure E.3: The Syntax Diagram for the CreateRole method

method. As shown in the figure, the create method header is followed by optional variable declarations and objects creation, and ends with the return statement. The objects are declared, instantiated, and possibly initialized. However, these objects can be created separately/outside the CreateRole method and passed as parameters so they can be reused by other roles.

- The CreatePolicy() Method: Figure E.4 shows the syntax for the method.

  The syntax starts with the header followed by some optional variable declarations and objects creation and ends with the return statement. The objects are declared, instantiated, and initialized. In the CreatePolicy() method, those objects are of types Subject and Role. One or more Subjects and Roles must be defined prior to executing this method. In the simplest case, the created policy specifies the association between one Subject and one Role in order to define the access rights that a subject is authorized to apply. However, complex policies can be specified by defining more than one Subject and Role within the

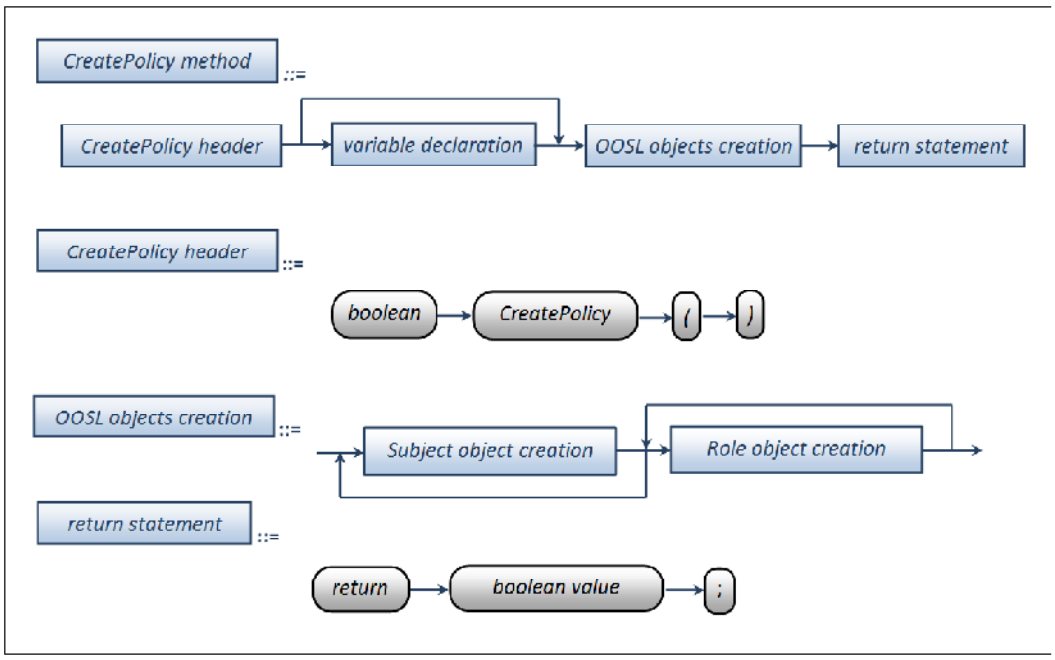


Figure E.4: The Syntax Diagram for the CreatePolicy method

policy. Each Subject then can be assigned to one or more Roles, and each Role in turn, can be associated with more than one Subject.

- The UpdateObject() Method: The syntax diagram of the method is depicted in Figure E.5, which starts with the header that specifies the method's name and the values to be replaced. Theses values are represented by two parameters, the *protObj* that holds the new protected data, and the *objExcep* that holds the new exception value(s). In the case of a successful update the method returns true, otherwise false.

- The UpdateSubject() Method: Figure E.6 depicts the syntax diagram of this method, where it follows the same logic as that of the UpdateObject() method. As depicted in the figure, the syntax starts with the header that shows the return type, followed by some optional variable declarations, and ends with the return statement. The method's header specifies the method's name and one

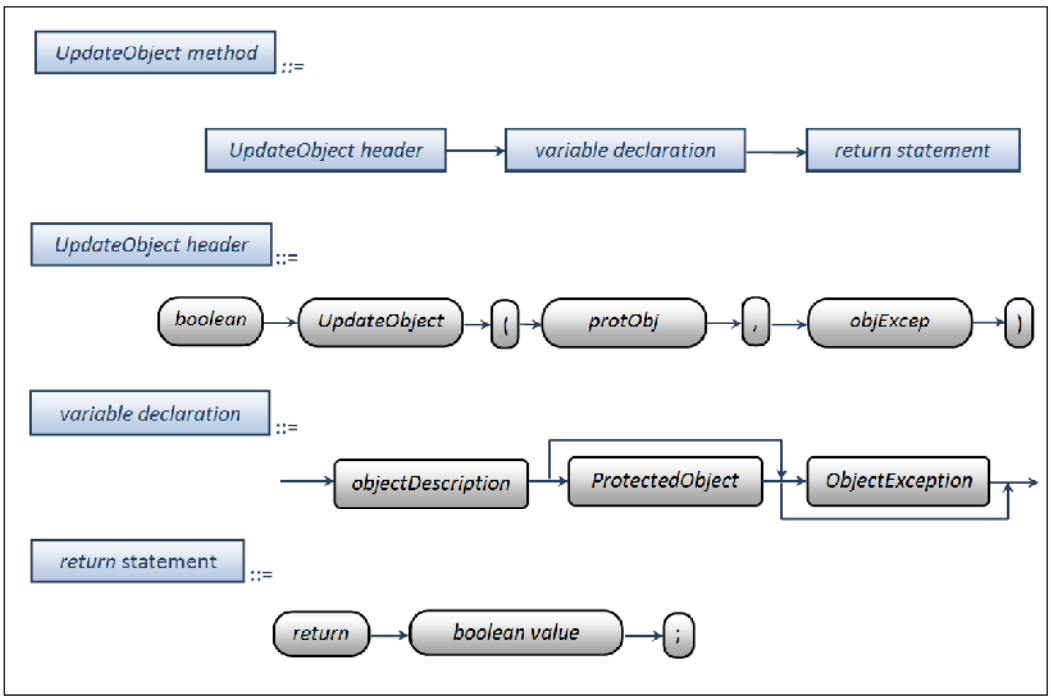


Figure E.5: The Syntax Diagram for the UpdateObject method

parameter (e.g., *newPasscode*), where the newPasscode is the new password value to be replaced. In case of successfully updating, the method returns true.

- The DROP() Method: Figure E.7 depicts the syntax diagram for the Drop method. The first syntax diagram starts with the method's header followed by the return statement. The method's header shows the return type, which is a boolean value, and indicates that it takes no arguments. The last syntax diagram returns the boolean value true if the object is drooped successfully, and false otherwise.

- The ASSIGN() Method: The syntax diagram of the method is illustrated in Figure E.8. As noted, the method header has two parameters: *subject* and *role* objects (e.g., the subject that will be assigned to the role). The main statement in the assign method is the assignment statement that associates the specified

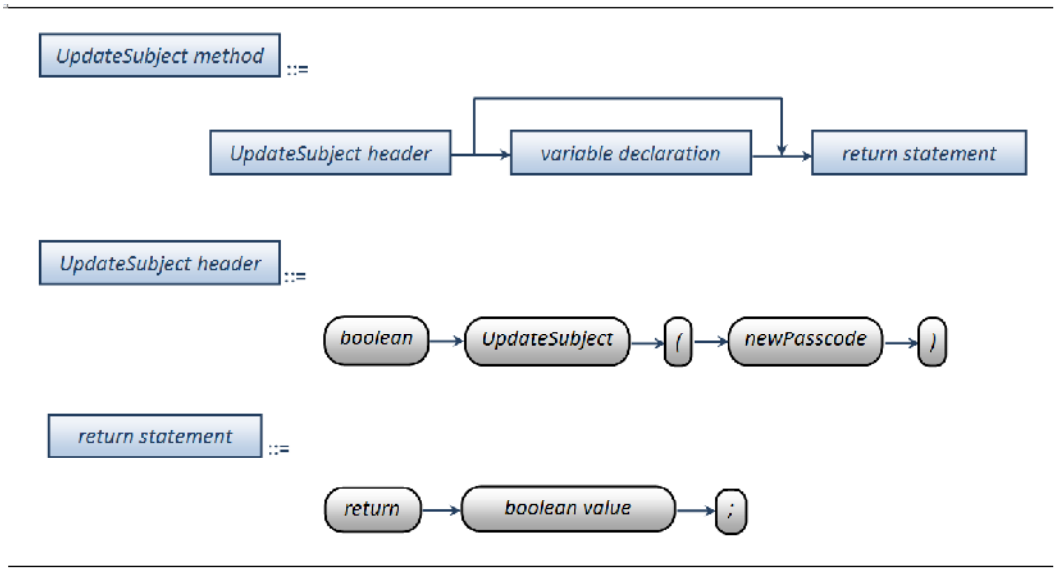


Figure E.6: The Syntax Diagram for the UpdateSubject method

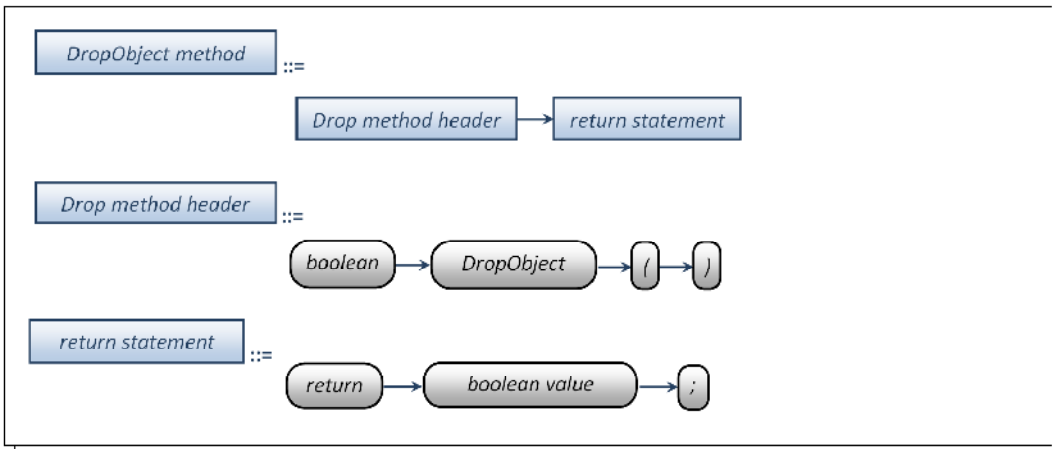


Figure E.7: The Syntax Diagram for the Drop object method

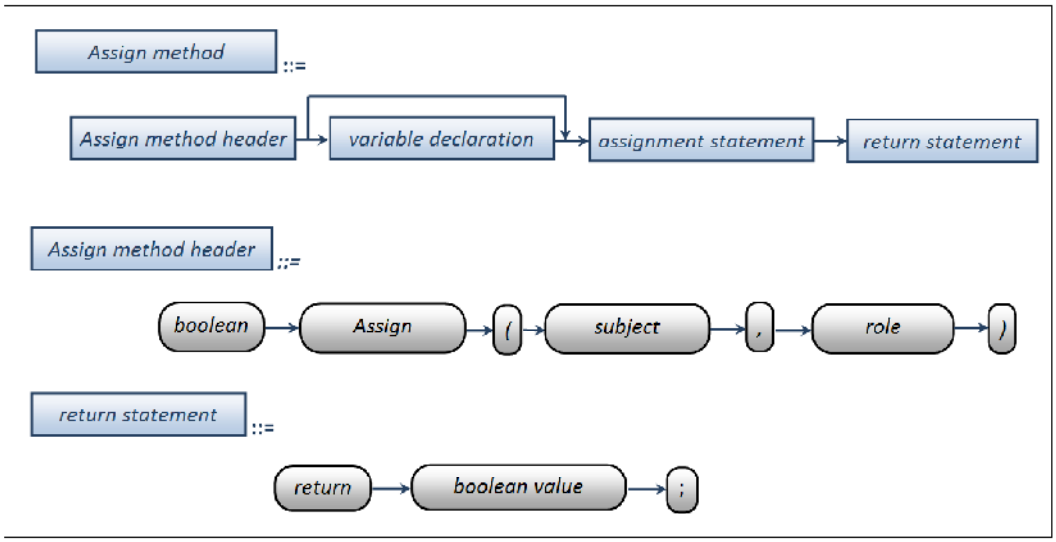


Figure E.8: The Syntax Diagram for the Assign method

Subject with the specified Role. Of course, a Subject can be assigned to more than one role by re-executing this method.

- The WITHDRAW() Method: Figure E.9 illustrates the syntax of the method. As shown in the figure, the method's header has two parameters (e.g., *subject* and *role*) that identifies the involved objects, followed by the withdrawal statement, which is the main statement in this method and which removes the Subject membership from the specified Role. The method returns true if the operation is successfully executed, and false otherwise.

- The ADD Methods: Figure E.10(a) and Figure E.10(b) depict the syntax diagrams for the **AddSubject()** and **AddRole()** methods, respectively. Their syntax starts with the header followed by some optional variable declarations and ends with the return statement. The method's header has one parameter (e.g., *subject* or *role*). As the names indicates, the parameter subject represents the Subject that will be added to the policy, and the role parameter represents the Role that needs to be added to the policy. Finally, the return statement

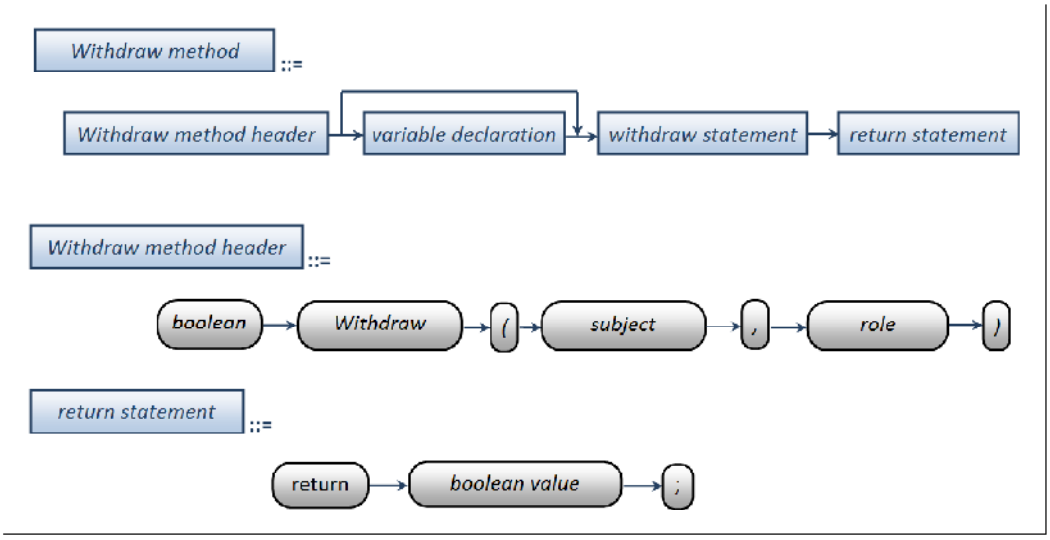


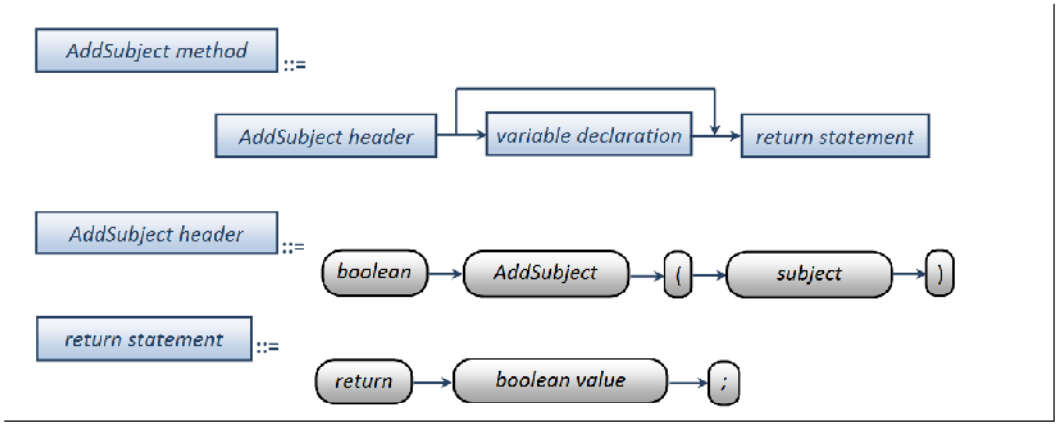
Figure E.9: The Syntax Diagram for the Withdraw method

returns true whenever the operation is executed successfully.

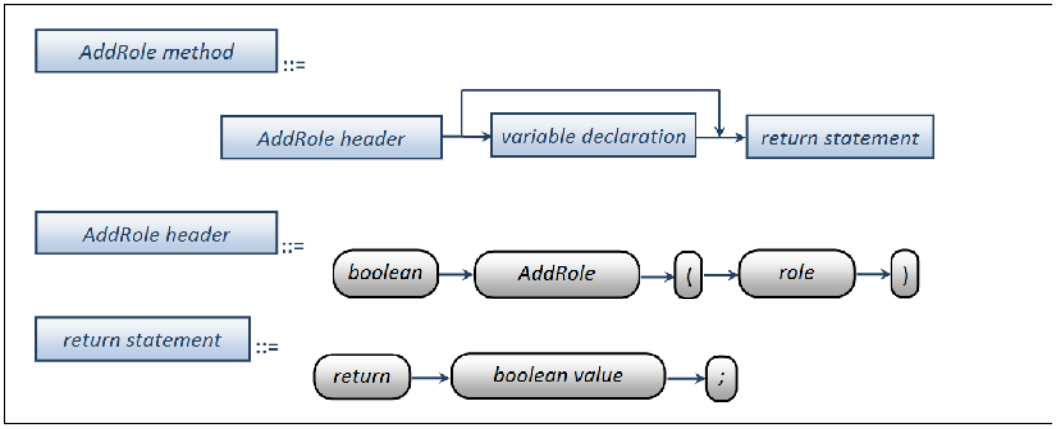
- The REMOVE Methods: Figure E.11(a) and Figure E.11(b) depict the syntax diagrams for the RemoveSubject() and RemoveRole() methods, respectively. They start with the method header that has one parameter (e.g., *subject* or *role*) identifying which object should be removed, followed by an optional variable declaration and ends with the return statement. In case of successful removal, the method returns true, otherwise false.

- The SELECT Methods: Figure E.12(a) and Figure E.12(b) show the syntax of the SelectRoleSubjects() and SelectSubjectRoles() methods, respectively. As shown in the figures, the methods receive one parameter and return a list of objects as a response. For instance, the first method receives a role identifier and returns all subjects assigned to the specified role.

  On the other hand, Figure E.13(a) and Figure E.13(b) show the syntax of the SelectPolicyRoles() and SelectPolicySubjects() methods, respectively. These methods do not receive parameters; however, they return Subjects and Roles

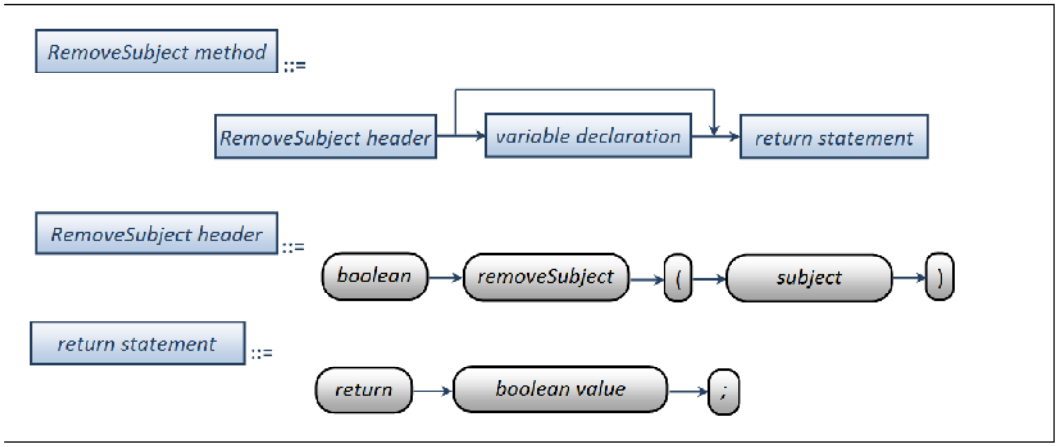


(a)

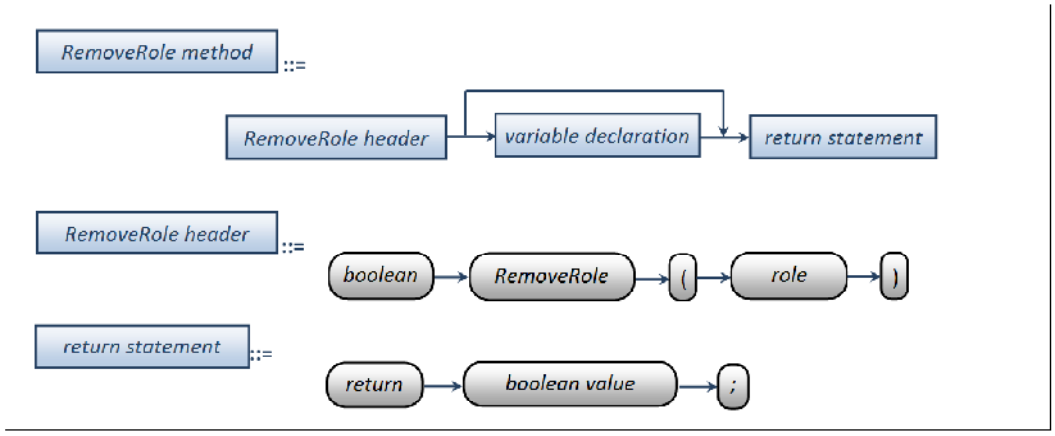


(b)

Figure E.10: (a)The Syntax Diagram for the AddSubject method (b)The Syntax Diagram for the AddRole method



(a)



(b)

Figure E.11: (a)The Syntax Diagram for the RemoveSubject method (b)The Syntax Diagram for the RemoveRole method

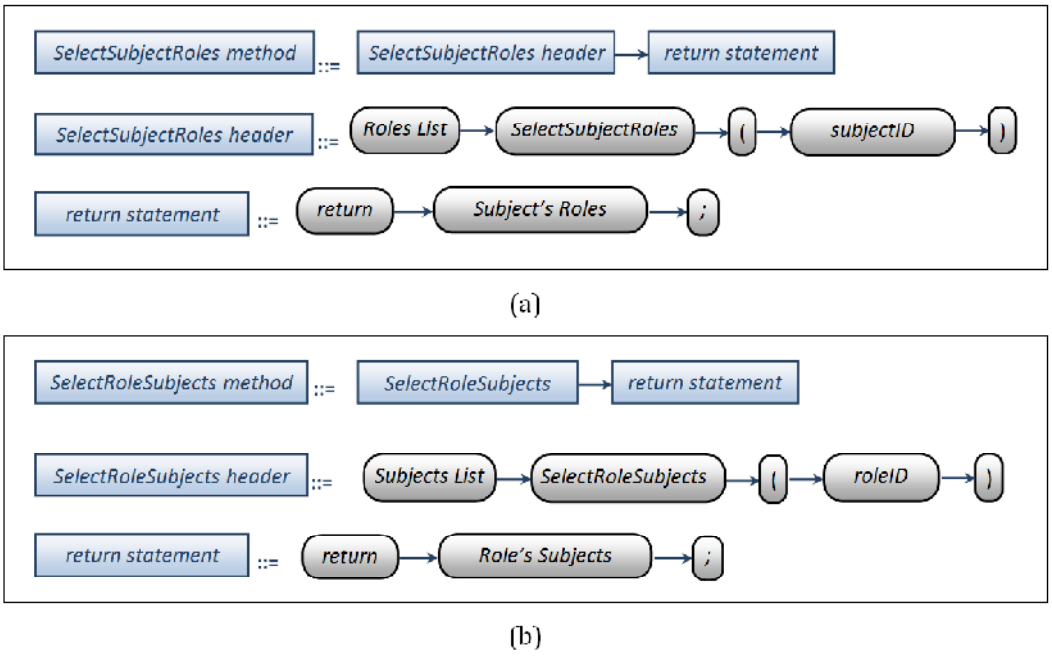


Figure E.12: The Syntax Diagrams for the SelectRoleSubjects and SelectSubjectRoles methods

that are controlled by the Policy.

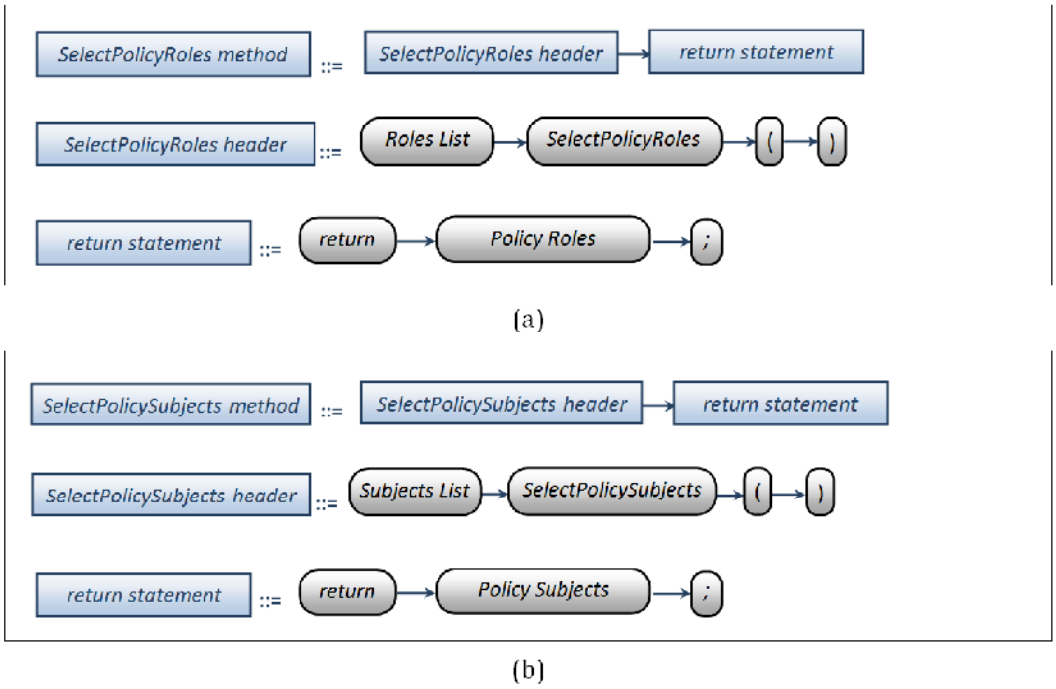


Figure E.13: The Syntax Diagrams for the SelectPolicyRoles and SelectPolicySubjects methods