Query Evaluation in the Presence of Fine-grained Access Control

by

Huaxin Zhang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2008

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Huaxin Zhang

Abstract

Access controls are mechanisms to enhance security by protecting data from unauthorized accesses. In contrast to traditional access controls that grant access rights at the granularity of the whole tables or views, fine-grained access controls specify access controls at finer granularity, e.g., individual nodes in XML databases and individual tuples in relational databases.

While there is a voluminous literature on specifying and modeling fine-grained access controls, less work has been done to address the performance issues of database systems with fine-grained access controls. This thesis addresses the performance issues of fine-grained access controls and proposes corresponding solutions. In particular, the following issues are addressed: effective storage of massive access controls, efficient query planning for secure query evaluation, and accurate cardinality estimation for access controlled data.

Because fine-grained access controls specify access rights from each user to each piece of data in the system, they are effectively a massive matrix of the size as the product of the number of users and the size of data. Therefore, fine-grained access controls require a very compact encoding to be feasible. The proposed storage system in this thesis achieves an unprecedented level of compactness by leveraging the high correlation of access controls found in real system data. This correlation comes from two sides: the structural similarity of access rights between data, and the similarity of access patterns from different users. This encoding can be embedded into a linearized representation of XML data such that a query evaluation framework is able to compute the answer to the access controlled query with minimal disk I/O to the access controls.

Query optimization is a crucial component for database systems. This thesis proposes an intelligent query plan caching mechanism that has lower amortized cost for query planning in the presence of fine-grained access controls. The rationale behind this query plan caching mechanism is that the queries, customized by different access controls from different users, may share common upper-level join trees in their optimal query plans. Since join plan generation is an expensive step in query optimization, reusing the upper-level join trees will reduce query optimization significantly. The proposed caching mechanism is able to match efficient query plans to access controlled query plans with minimal runtime cost.

In case of a query plan cache miss, the optimizer needs to optimize an access controlled query from scratch. This depends on accurate cardinality estimation on the size of the intermediate query results. This thesis proposes a novel sampling scheme that has better accuracy than traditional cardinality estimation techniques.

Acknowledgements

First I shall say many thanks to my supervisor Professor Ken Salem. During the past several years Ken has gradually changed my way of doing research through his organized style of thinking and writing. His scrutiny and thoroughness has greatly benefit my projects along the course of the thesis. This thesis will not be possible without his continuous encouragement and support.

I shall thank Prof. Ihab Ilyas for his pertinent suggestions. His sparkling ideas has always been enlightening. I am grateful to Professor Frank Tompa for improving this thesis in my aspects. I am also thankful to Professor Nick Koudas and Professor Anwar Hasan for reviewing my thesis.

I shall acknowledge my great appreciation for the financial support from the David R. Cheriton School of Computer Science at University of Waterloo, and a funding from Communications and Information Technology Ontario (CITO) and OpenText Corporation.

I owe greatly to my family during these years. My parents gave me continuous support, and my wife has always been considerate to me. My aunt and uncle gave me the biggest help during my hard time here. I feel indebted to them all.

While staying in Waterloo, I have made great friends. I am grateful to Lei Chen, Khuzaima Daudjee, Ye Qin, Qiang Wang, Ning Zhang and many others that share so many wonderful experiences with me.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Access controls are mechanisms to enhance security by protecting data from unauthorized accesses. They define what actions or operations a user is allowed to perform on a specific piece of data. Traditional access control mechanisms for database systems are implemented by granting users access rights to the entirety of the entities in the databases, such as tables and views. When a user submits a query, all of the entities referred to in his query will be checked against the user's access rights. If one of the entities is inaccessible to the user, the whole query will be rejected. Such mechanisms are also referred to as *coarse-grained access controls*.

As systems scale and become more complex, coarse-grained mechanisms fail to satisfy the requirements imposed on modern database systems. For example, suppose a system has thousands of users, each of whom is trying to see his payroll information. Suppose the payroll information for all users is centralized inside one table, and each user is allowed to see only his own information. Using a traditional coarse-grained access control mechanism, each user needs to be provided with a distinct view containing only his information from the table. This will give rise to thousands of views in the system just to implement one access control requirement.

In contrast to coarse-grained access controls, fine-grained access controls (FGAC) specify access rights at finer granularity. For example, FGAC may specify, for each tuple or cell in a relational table, or for each element or attribute in XML data, whether or not a user has read or write access to that data. If we consider the users, the data, and the action modes (such as read, write) as being three orthogonal dimensions, FGAC specifications can be modeled as a three dimensional matrix with a *true* or *false* value in each cell indicating whether the corresponding piece of data is accessible or inaccessible to the specific user under the specific action mode. When a user submits a query, the system may provide a *partial answer* to the query based on the *accessible portion* of the entities referred to in the query, instead of rejecting the whole query if the user does not have access rights to the entirety of these entities.

FGAC enables us to fulfill sophisticated access control requirements much more efficiently

than traditional coarse-grained access control does. Take the previous example in which users query their payroll information. With FGAC, each tuple in the table is conceptually associated with an extra attribute encoding the accessibility information for that tuple. The database system only applies each user's query on the tuples he has access to, by checking the tuples' associated access control information. Now we are able to treat access control information as part of the data, by slightly modifying the schema. Recall that in the case of coarse-grained access control we need to create thousands of views (schema objects) to fulfil this same requirement. By using FGAC, not only is the schema cleaner, but the access control information is easier to manage, as we shall see later in this thesis.

Moreover, FGAC is capable of enforcing access control requirements that are beyond the expressiveness of coarse-grained access control. For example, coarse-grained access control is based on views and will not change once data value changes. FGAC also allows the determination of more sophisticated access control rules, such as access control propagations and conflict resolution. We will see more of these examples in Chapter 2. These features make FGAC an effective mechanism for enforcing more sophisticated security requirements. Therefore, all major database vendors have started to provide FGAC implementations in their products [78]. For example, Oracle provides Virtual Private Database (VPD) [70] and Oracle Security Label (OLS) [69] as two means of supporting FGAC, DB2 offers Label-Based Access Control (LBAC) [45], and SQL Server delivers a similar label-based mechanism [66].

1.1 Motivation

While there is a voluminous literature on specifying and modeling FGAC, as we shall see in Chapter 2, less work has been done to address the performance issues of database systems with FGAC. A database system that supports FGAC will inevitably have components not found in a traditional database system. These components will in turn affect query evaluation performance and system maintenance.

Figure 1.1 depicts database query processing with and without FGAC. Compared to query evaluation without FGAC (the left side of the figure), a system with FGAC needs to store the FGAC specifications that correspond to the access control matrix mentioned in the previous section. If access controls are implemented using the view-based approach describe in Chapter 2, user queries must be modified so that they are applied against the user's view of the database objects. The optimizer needs both database statistics and the FGAC specifications to estimate the cardinality of intermediate results for effective query planning for the modified user's query. When processing the query plan, the database engine needs to look up the accessibility of each piece of data for the current user, if access controls are enforced through instance-based FGAC (also see more in Chapter 2).

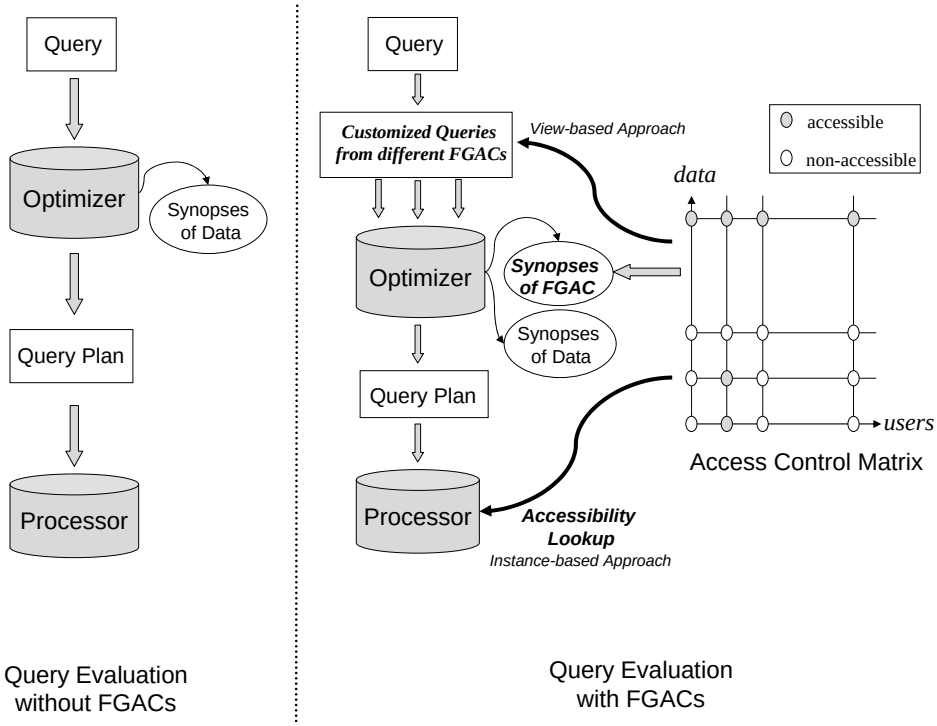


Figure 1.1: Comparison between query evaluation with and without FGAC

From the above, we can summarize the following issues for database systems with FGAC:

1. FGAC specifications will incur extra storage and maintenance costs. As we will see in Chapter 2, uncompressed FGAC matrix takes space proportional to the product of the number of fine-grained data elements and the number of users.

2. When there are no access controls, a given query can be answered by the same query plan regardless of which user submits the query. The database system only needs to optimize this query once, as it can cache and reuse the query plan if the query is subsequently submitted by other users. With coarse-grained access controls, the database system only needs to validate the users' right to access each of the relations in the query before reusing the same cached query plan. This checking can be done with negligible cost. However, with FGAC, the same query submitted by different users will be rewritten into many different queries according to the distinct users' fine-grained access controls. These queries, known as FGAC-customized queries, will have different structures and semantics, and may not share the same optimal query plan. Therefore, the database system needs to compute query plans for all these customized queries, adding significant cost to query optimization.

3. FGAC adds to the complexity of query optimization for each FGAC-customized query. Without FGAC, the optimizer only needs to estimate the cardinality of data selected by the queries. With FGAC, the optimizer needs to estimate the amount of data satisfying both the original query and the access controls. If the access controls and the queries are correlated, these cardinality estimations may have large errors and may therefore lead to inefficient query plans.

4. Query execution will incur extra cost for checking access rights at runtime. Moreover, if the FGAC specifications are not compressed, loading these specifications may incur significant I/O cost. On the other hand, if the FGAC specifications are compressed to alleviate the storage problem, the runtime cost to interpret such compressed specifications can be high.

1.2 Contributions and Scope

This thesis addresses the issues outlined in the previous section, and makes the following contributions:

1. A compact storage scheme called DOL (Document Ordered Labeling) for storing FGAC specifications for XML databases. This storage scheme is scalable in the size of data and

the number of users, and is easy to maintain. The DOL scheme achieves compression by leveraging the high correlation of access controls found in real system data. This correlation comes from two sides: the structural similarity of access rights among database, and the similarity of access patterns of different users. This work addresses the first issue defined in the previous section.

2. A scheme called POM (Partitioned query Optimization for Multiple users) for efficient relational query planning in the presence of FGAC. POM is an intelligent plan caching mechanism that reduces the amortized cost of query optimization of a large number of FGAC-customized queries. The rationale behind this query plan caching is that the queries, customized for different users' access controls, may share common upper-level join trees in their optimal query plans. Since join tree enumeration and creation are expensive operations in query optimization, reusing the upper-level join trees will reduce query optimization significantly. This POM mechanism is able to match efficient query plans to FGAC-customized queries with negligible runtime cost, and hence addresses the second issue outlined in the previous section.

3. In case of a cache miss on a query plan from POM, the optimizer needs to optimize the FGAC-customized query from scratch. This depends on accurate estimation of the cardinality of intermediate query results. This thesis presents PSALM (Partitioned SAmpLing for Multiple-user system), a synopsis structure that captures the distribution of data in the presence of FGAC. More specifically, PSALM is a sampling scheme that provides more accurate estimation of the cardinality of the data satisfying both the query and the access controls than the traditional uniform sampling scheme. PSALM is compact and scalable in the number of users with distinct access rights. This work addresses the third issue mentioned in the previous section.

4. A query processing mechanism that efficiently verifies access rights. This thesis shows a way of embedding the DOL representation of the access controls into a linearized representation of XML data such that a query evaluation framework called NoK [98] is able to compute the answer to the access controlled query with reduced disk I/O to the access controls. This addresses the fourth issue in the previous section.

The scope of this thesis can be illustrated by the chart in Figure 1.2. This thesis focuses on the issues that affect query evaluation performances and database architecture, and addresses such issues for XML and relational databases. For query evaluation, we look at both query optimization and query processing. For database architecture, we look at FGAC storage and maintenance, and auxiliary data structures.

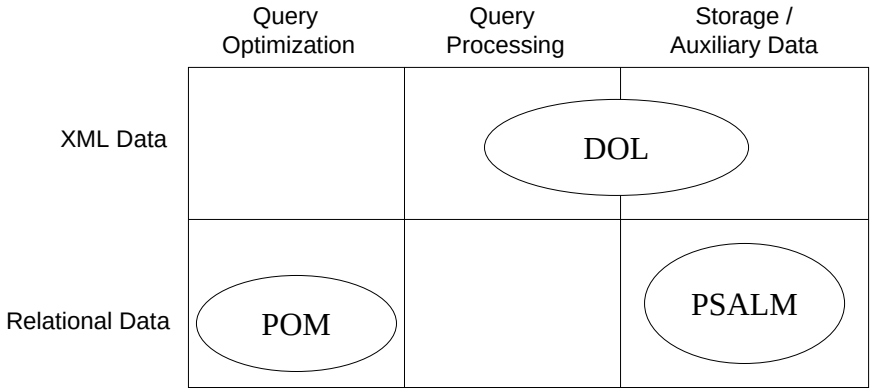


Figure 1.2: Thesis contribution

The work of DOL covers both FGAC storage and query processing for XML data. The POM scheme handles query optimization for relational data, and the PSALM work handles issues in query optimization and auxiliary data structures for FGAC. The approach in PSALM can be applied for both XML and relational data.

This thesis does not address the issues that are not relevant to query evaluation with FGAC, such as the specification of FGAC. There is already a substantial amount of work in the field of FGAC specification, as will be shown in Chapter 2.

1.3 Thesis Organization

This thesis is organized as follows: Chapter 2 presents background and related work on FGAC. Chapter 3 presents the DOL representation of fine-grained access controls. Chapter 4 describes the POM query optimization scheme. Chapter 5 presents the PSALM synopsis scheme. Chapter 3, Chapter 4, and Chapter 5 together provide the solutions for the building blocks in a high-performance database system with FGAC, as was illustrated in Figure 1.1. Chapter 6 concludes the thesis with discussions and proposes future work.

Chapter 2

Background

This thesis addresses many aspects of query processing under the theme of FGAC. Because there is a voluminous literature in the field of query evaluation, this thesis only includes related work on query evaluation that is most relevant to the theme of the thesis. Furthermore, the related work on query evaluation is split among the subsequent chapters for side-by-side comparison with the thesis work. For similar reasons, we do not address any database security technologies [19] that are not directly related to access controls, such as defending against inference attacks [29, 35], privacy preservation through K-anonymity [59, 79, 86], intrusion detection [61], authentication and encryption [17, 68, 87], watermarking [14, 84], and logging and auditing [50]. In this chapter we provide an overview of related work in FGAC modeling and implementation.

2.1 Definition of FGAC

As mentioned in Chapter 1, database systems are increasingly being used to store information for large, sophisticated applications. Data stored in these systems are vital business assets and therefore need to be kept from unauthorized parties. Access controls, in general, provide such protection to ensure that the data are not revealed or modified to certain users.

There are three essential elements in access controls for a database system: the *users*, the *data*, and the *reasons for access*. Here the *users* can be the real users, or processes accessing the databases. The *data* can be base tables, views, or other entities in the databases. The *reasons for access*, also known as *action modes*, includes *read*, *update*, and others.

One way to model access controls is to think of them as a grid in a three-dimensional space, where the X, Y, and Z axes of the space corresponding to the *users*, the *data*, and the *reasons for access*. Each cell in this grid represents a mapping from a specific user, a data item, and a reason for access to a decision of either *accessible* or *inaccessible*. Therefore, this three-dimensional

space is also referred to as the *access control matrix* [42, 55].

For example, the traditional coarse-grained access control mechanism found in relational database systems corresponds to an access control matrix with the users in the system forming the *users* dimension, the tables and the views as the *data* dimension, and the action modes such as *read* and *update* as the *reasons for access* dimension. Each cell in the matrix has a flag indicating whether the user at that cell is able to read or update the corresponding data item.

Compared to the number of users and the number of data items, the number of elements in the *reasons for access* dimension is usually a small constant. Therefore, we can remove this dimension from the access control matrix by slicing the 3D matrix into a fixed number of two-dimensional matrices, each having only the users and data dimensions. In the rest of the thesis, we will focus on this two-dimensional representation for simplicity.

As we have already mentioned in Chapter 1, FGAC specifies access controls at a fine data granularity. Speaking in terms of the access control matrix model, the major difference between FGAC and the traditional coarse-grained access control is that the tables or the views (or the whole XML documents) in the *data* dimension in coarse-grained access control matrix are now replaced with rows, columns, or field (for relational data) or nodes (for XML data) in the FGAC access control matrix.

2.2 Specifying Fine-grained Access Control

Because the size of the access control matrix is proportional to the number of items in the *data* dimension, FGAC will have a much larger matrix representation than that of the coarse-grained access control. Moreover, as the system scales up, more and more users are added to the scene. Therefore, this matrix representation must have a compact implementation to be practical for database systems.

One way to represent the matrix compactly is to record the cells as tuples of $<$*user, object*$>$, where the existence of such tuple denotes that the *user* has access to the *object*. Such tuples form an *Authorization Relation* [82]. If the tuples are clustered by the $user$ column, this relation is called a *Capability List* [19]; if the tuples are clustered by the $object$ column, the relation is called an *Access Control List* [19]. Even though these tuple-based representations take less space than the original access control matrix, their sizes are still proportional to the product of the number of users and the number of objects. With that size, FGAC specification and storage is still cumbersome.

For this reason, sveral access control modeling techniques are proposed to implement FGAC efficiently. Each of these modeling techniques is designed for a specific type of access control needs. The Discretionary Access Control (DAC) [55] [82] model, which is heavily used in operating systems and existing database systems, allows the owner of an object to dictate the

access control based on his discretion. The Mandatory Access Control (MAC) [28][81] model, which is mostly used in military information systems, has a hard-coded security level for each object and each user in the system. A user with security level M can only read objects with security level equal to or less than M, and can only write to objects with security level equal to or higher than M. This way MAC is able to specify information flows in a strict top-down or bottom up fashion. A third model, called Role-Based Access Control (RBAC), has received lots of attention from researchers. The idea behind RBAC is to create roles and assign *privileges* to the roles. Users are assigned to roles, and the roles can be assigned to other roles, forming a *subject hierarchy*. A user derives his access rights from all his ancestors in the subject hierarchy.

It was shown in [71] [80] that RBAC is able to express both MAC and DAC. Therefore, we will describe RBAC and its variants in the rest of this chapter.

2.2.1 Modeling Through Explicit Specifications

The power of RBAC is that a system administrator is able to define a few *explicit access control specifications* corresponding to a few *<user, object>* cells in a two-dimensional access control matrix. These explicit access control specifications can *propagate* to the other cells in the access control matrix. After propagation, every cell in the matrix will be defined as either *accessible* or *inaccessible*. The derived, unambiguous specifications over the entire two-dimensional matrix are also referred to as the *effective access control specifications*.

The access control propagation procedure is based on hierarchy among the users or among the data. Data often have a tree structure. One example of this tree structure is found in Unix file systems. The users also form groups or subject hierarchies, as from the roles mentioned earlier. One strategy to exploit such hierarchy is to specify explicit access rights for the roles or user-groups in the subject hierarchy, or for the items in the data hierarchy. Then these explicit access specifications are propagated by letting the children inherit accessibility from their ancestors, or letting the users inherit access rights from their groups or roles. For example, the file system in $Windows$® operating system specifies that a file by default inherit its access controls from its parent. On the other hand, users in UNIX file systems inherit their access rights from the user groups. Hierarchy in the *reason for access* dimension and access rights propagation along that dimension can be found in [53, 97].

Figure 2.1 shows an example of propagating explicit access control specifications to effective access control specifications. Suppose we have the subject hierarchy as shown on the left side in Figure 2.1, where a subject S_3 derives his access rights from subject S_2 and subject S_1. The three hollow dots in the matrix in the middle of the figure represent the explicit specifications. Each hollow dot means the corresponding user is allowed access to the corresponding object at the explicit specification level. If the propagation rule specifies that the descendant users inherit access rights from their ancestors, these explicit specifications will propagate to produce the

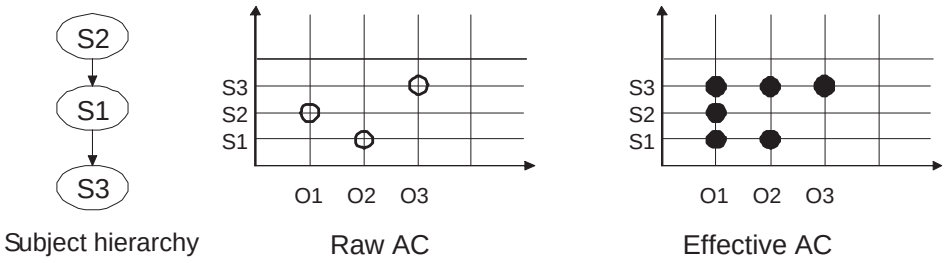


Figure 2.1: Access control propagation

effective specifications on the right side in the figure, where the existence of a solid dot means the user at the cell has access to the corresponding object.

In the rest of the discussion, we use the term *subject* to denote an item in the subject hierarchy, which can be a user, a user-group, or a role. We also use the term *object* to denote an item in the data hierarchy, which can be a tuple, a column, or a field in relational data or a node in XML data.

2.2.2 Negative Authorizations

In the previous discussion, we assume the explicit access specifications are *positive authorizations*, i.e., they represent *privileges* granted to the subjects. However, lack of positive authorization on a subject does not prohibit the subject from getting access rights. For example, the subject can derive such access rights from its ancestor subjects. If we want to make sure that certain users are never allowed access to some sensitive data, we will need a mechanism to explicitly enforce such a requirement. A *negative authorization* allows the administrator to explicitly specify that a $subject$ *cannot* access an $object$, preventing that $subject$ from being implicitly authorized to access the $object$, e.g., through access rights propagation from the $subject$'s ancestors. Negative authorization is also an effective means to specify "exceptions" to access right assignments within a large user group, in which only a few users do not have access to the same objects as the other members of the group.

The effective access control specification must be consistent and complete, which means every cell of the access control matrix must be defined as being either "accessible" or "inaccessible". If a system allows both positive and negative explicit authorizations, both types of authorizations can propagate along the subject or the object hierarchy to the same cell in the access control matrix, leading to conflicts.

Therefore, we need policies to prevent conflicts and to resolve them after the explicit access control specifications propagate to the effective specifications. There are several such policies in

literature [23, 51, 76], and we list some of the policies with respect to subject hierarchy from the work by Jajodia et al. [51]:

Most Specific Override Propagation This policy specifies propagation of explicit access control specifications in the following manner: a subject a propagates its explicit access control specification on an object to its descendant subject c if there is no subject b that is both a descendant of a and an ancestor of c, and that has a conflicting explicit specification on the same object.

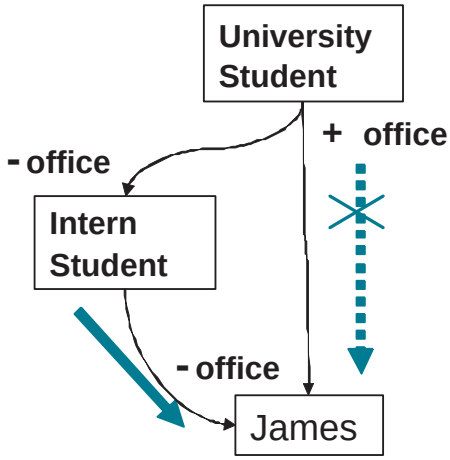


Figure 2.2: Most specific propagation

For example, the subject *University Student* in Figure 2.2 cannot propagate its explicit positive authorization on the object *office* to the subject *James* since there is another subject *Intern Student* between them with explicit negative authorization on the same object *office*. In this example, only the subject *Intern Student* can propagate its access control specification to the subject *James*.

Path Override Propagation This policy specifies that a subject a propagates its explicit access control specification on an object to its descendant subject c if the descendant c does not have a conflicting explicit specification on the same object. Using the example in Figure 2.2, both the subject *University Student* and the subject *Intern Student* can propagate their explicit access specifications to the subject *James*.

Negative Precedence Conflict Resolution Conflicts may exist after applying propagations using either of the above policies. As shown in Figure 2.3, both the explicit positive and the explicit negative authorization from the subject *DB student* and the subject *intern student* can be propa-

gated to the same subject *James*, since there is no ancestor-descendant relationship between the subjects *DB student* and the subject *intern student*. In this scenario, the conflicts can be resolved by adopting the *Negative Precedence* policy, which always allows a negative authorization override a positive authorization.

Other conflict resolution policies are the *Strong-Weak* policy [76] (a strong authorization always overrides a weak authorization), and the *Positive-Precedence* [51] policy (a positive authorization always overrides a negative authorization). The Strong-Weak policy does not resolve the conflicts between two strong authorizations or between two weak authorizations. The Positive-Precedence policy is simply symmetric to the Negative Precedence policy.

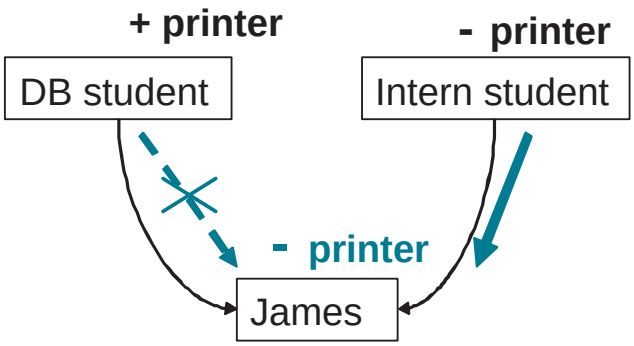


Figure 2.3: Negative precedence resolving

Closed and Open Policies

Under certain scenarios, there can be some <*subject, object*> cells without access control specifications in the matrix after explicit specification propagation and conflict resolution. These cells contain undefined access control information. There are two policies to resolve these undefined cells, namely the *Closed Policy* and the *Open Policy* [51]. The Closed Policy assumes that all the objects in these non-defined cells are by default *inaccessible* to the corresponding subjects, while the Open Policy says all subjects by default have access to the corresponding objects. By adopting either the Open or the Closed Policy together with the conflict resolution policies, we can make sure that every cell in the access control matrix is defined as being either accessible or inaccessible.

2.2.3 Other Specification Languages

The access control matrix is a conceptual authorization model. This model can be specified using approaches such as DAG, MAC, or RBAC, as has been described in the previous section. More-

over, this conceptual matrix model can be enriched and implemented using different languages. The expressiveness of these languages and their runtime complexity varies. We already listed several policies for implementing the access control matrix using a small set of explicit access control specifications. In this section we list some other specification languages that have been described in the literature.

Early attempts to model and implement access controls date back to the 1970s, when the first rule-based access control language was designed for operating systems [42]. That language supports changing of access specifications and updating the domains by adding or deleting users and objects. However, the language is undecidable. Later Woo et al. [94] specified an access control language that is independent of the underlying system. This language is decidable, but its computation is still intractable.

Jajodia et al. [52] modeled access controls using stratified Datalog. Their language supports several propagation policies, and conflict resolution policies, together with the open and closed policies. The system administrator is able to specify a few explicit specifications and let the Datalog program convert it to a fully defined, non-conflicting access control matrix. This language takes polynomial time to compute. Later two independent works [51, 92] both provided algorithms to incrementally maintain the access controls in response to updates to the high level specifications. Bertino et al. [11] addressed access control modeling using a larger fragment of Datalog (i.e., not just stratified Datalog). Further more, they allowed user defined predicates in their language. However, the model cannot be computed within polynomial time.

Wijesekera et al. [90, 91] used algebra to model a class of access control policies that is equivalent to those modeled by Jajodia et al. Their language is able to reason about completeness and consistency of the access control semantics, to determine if two sets of access controls are equivalent, and to decide whether an algebraic framework is ambiguous. Other algebraic approaches can be found in [16, 89].

There are access control languages that model provisional authorization (pre-conditions) or the obligations to the users or the systems that must be fulfilled (post-conditions) [15] [99]. Temporal constraints on access controls are introduced in [9, 54].

For database systems, there are access control languages specifically for relational data [13] and XML data [10, 12]. However, these languages do not have polynomial time guarantees and hence cannot be implemented efficiently.

2.3 Query Evaluation with FGAC

Most existing database systems support coarse-grained access controls by assigning each user with several logical views as the user's accessible data. In the rest of the discussion, we name these logical views as *authorization views* according to their special purposes. The user's query

should only refer to his authorization views, i.e., the query can never syntactically refer to any relation that he does not have access to.

One drawback of the above approach is that the users are forced to use their authorized views to formulate their queries, which means the users have to worry about the access controls while formulating the business logic of their queries.

Instead of prohibiting the users' queries from syntactically referring to inaccessible relations, researchers propose that the users be allowed to formulate queries referring to any relation in the database. The database system will enforce the access controls by rewriting the queries so that they use only the users' accessible data. The rewritten query will be evaluated, and the result will be returned to the user as if the original query had been evaluated. This can be formalized as follows:

Definition 2.3.1. *A user's authorized data consists of a set of authorization views* $\mathcal{V} = \{V_1, V_2, \ldots, V_m\}$. *Each authorization view* V_i *is a two tuple* $< V_i^D, V_i^I >$ *such that* V_i^D *is the view definition and* V_i^I *is the view instance.*

A ***query rewriting for a user's query*** Q ***based on his authorization views*** *is a query* Q' *such that* Q' *only refers to the view definitions* V_i^D *in the authorization views* $\mathcal{V}$*, and* Q' *is only applied on the view instance* V_i^I *in the authorization views* $\mathcal{V}$.

There are two models of answering the query based on the relationship between Q and Q', one is called the *certain answer model*, and the other is called the *equivalent query rewriting model*.

2.3.1 Certain Answer Model

This model is based on the notion of *certain answers*, whose definition depends on two different assumptions, namely the *closed-world assumption* and the *open-world assumption*.

Let $\mathcal{V} = \{V_1, V_2, \ldots, V_m\}$ be a set of authorization views. Let $\{V_1^D, V_2^D, \ldots, V_m^D\}$ be view definitions in these authorization views, and let $\{V_1^I, V_2^I, \ldots, V_m^I\}$ be the view instances.

Definition 2.3.2. *[3] Under the closed world assumption(CWA), the authorization view instance stores all the tuples from the database* R *that satisfy the authorization view definition, i.e.* $V_i^I = V_i^D(R)$ *for every* $i, 1 \leq i \leq m$. *Under the open world assumption (OWA), the authorization view instance is possibly incomplete and might only store some of the tuples from the database* R *that satisfy the authorization view definitions, i.e.,* $V_i^I \subseteq V_i^D(R)$ *for every* $i, 1 \leq i \leq m$.

Definition 2.3.3. *[3]*

Let Q *be a query and* $\mathcal{V} = \{V_1, V_2, \ldots, V_m\}$ *be a set of authorization views with both view definitions and view instances.*

*A tuple α is a **certain answer** to the query Q **under the closed-world assumption** if $\alpha \in Q(I)$ for every database instance R such that $V_i^D(R) = V_i^I$ for every $i, 1 \leq i \leq m$.*

*A tuple α is a **certain answer** to the query Q **under the open-world assumption** if $\alpha \in Q(I)$ for every database instance R such that $V_i^D(R) \supseteq V_i^I$ for every $i, 1 \leq i \leq m$.*

For example [3], consider the following two authorization view definitions on the same base relation R:

$$V_1^D(X) \ :- \ R(X, Y)$$

$$V_2^D(Y) \ :- \ R(X, Y)$$

Also the authorization view instances are the following:

$$V_1^I = \{a\}$$

$$V_2^I = \{b\}$$

Under the open world assumption, we only know that some tuple in relation R has value a as its first component, and some (possibly different) tuple has value b as its second component. Under the closed world assumption, however, we can conclude that all tuples in R have value a as their first component and value b as their second component, i.e. base relation R contains exactly the tuple (a, b).

Definition 2.3.4. *Under the **Certain Answer Model**, the rewritten query Q' computes all the certain answers of the user's original query Q with respect to the set of authorization views granted to the corresponding user.*

It is easy to see that the rewritten query Q' is always contained in the original query Q for any database instance. In fact, there is a relationship between the *maximally contained query* with the *certain answers under the open-world assumption*. Details of this relationship can be found in [41].

According to the definition of the certain answer model, if the view instances given to the users are computed from the entire database instance, the rewritten query based on the authorization views should compute all the certain answers under the closed-world assumption. On the other hand, if the view instances given to the users are partial results computed from the database instance using the view definitions, the rewritten query based on the authorization views should compute all the certain answers under the open-world assumption.

Under the open-world assumption, it was shown in [3] that all the certain answers can be computed in polynomial time for many practical cases. However, if we allow union operators in the view definition or $\neq$ operator in the users' queries, computing all the certain answers is co-NP hard [3]. It was also shown in [3] that computing certain answers under the closed-world assumption is co-NP in the size of the view instances, even if the views and the queries are defined by conjunctive queries without comparison predicates.

The Truman Model as Relaxation

We already mentioned the computational intractability in the previous discussion under the closed world assumption. Even under the open world assumption, the results from [3] are only applicable to queries and views under set-semantics. The computational complexity under multiset-semantics is still unknown.

Several restrictions have been proposed to make the query rewriting based on authorization views feasible. A restricted version of authorization views was proposed in industry, such that each user has only one authorization view on each base relation, and each authorization view has the same schema as the underlying base relation. This can be formally defined using the concept of *filter authorization view* through the notion of *filter function*:

Definition 2.3.5. *A **filter function** with respect to user u on relation R maps each tuple from R to either **true** (accessible to u) or **false** (inaccessible to u).*

Definition 2.3.6. *A **filter authorization view** for user u on relation R is an authorization view $< V^D, V^I >$ such that the view definition V^D is $\sigma_{filter}(R)$, which selects the tuples from instance of R using a filter function for u on R. The view instance V^I is derived by applying the view definition on the complete instance R.*

If we restrict to one filter authorization for each base relation, and do not insist on the certain answer model under either open or closed world assumption, there exists a query rewriting model (called the *Truman Model* [77]) based on such restricted authorization views. The rewriting is straightforward and can be computed in linear time in the number of authorization views. We illustrate this model using the following procedure.

Suppose user u submits a query which refers to relations $R_1, R_2, \ldots, R_k$. We denote this query as $Q_{R_1,R_2,\ldots,R_k}$. We use $R_i^I, 1 \leq i \leq k$ to denote the contents of these relations in the database instance. When there are no access controls, the answers can be computed from the following:

$$Q_{R_1,R_2,\ldots,R_k}\left(R_1^I, R_2^I, \ldots, R_k^I\right)$$

Definition 2.3.7. *With FGAC under the **Truman Model**, the system defines a filter function for each user on each relation. The filter function for user u and relation R_i generates an authorization view, which we denote by $<R_i, V_i^I>$. When a user submits query $Q_{R_1,R_2,...,R_k}$, this query gets customized to a rewritten query (denoted by Q^u) according to the user's access controls so that it is applied on the view instances from the user's filter authorization views:*

$$Q^u = Q_{R_1,R_2,...,R_k}\left(V_1^I, V_2^I, \ldots, V_k^I\right)$$

The Truman model is straightforward to understand and incurs negligible query rewriting cost. Therefore, this model is implemented in Oracle VPD, Oracle OLS, and DB2 LBAC for relational data. For XML data, there is a similar filter based query processing mechanism. Details on this perspective can be found in Chapter 3.

The disadvantage of the Truman model is that it does not follow a rigorous model such as the certain answer model. Thus it may mislead the user if the user is not fully-aware of the query rewritings. Several such scenarios are presented in the work by Rizvi et al. [77].

2.3.2 Equivalent Query Rewriting Model

This model specifies that a user's query be accepted and evaluated only if there is a *valid* query rewriting using his accessible data. If there is no such *valid* rewriting, or the system fails to find such a rewriting, the user's query will be rejected.

The key concept here is the definition of a *valid* query rewriting. The work by Rizvi et al. [77] illustrates two types of valid query rewriting using a user's accessible data. One is called the *unconditionally valid query rewriting*, and the other is called the *conditionally valid query rewriting* [77].

Unconditionally Valid Query Rewriting

This type of query rewriting is defined formally as the following:

Given a set of definitions for authorization views $(V_1^D, V_2^D, \ldots, V_m^D)$ defining the accessible data for user u from the database. A user's query Q^u can be answered by a rewritten query Q' based on these authorization view definitions (denoted as $Q'_{V_1^D,V_2^D,...,V_m^D}$), such that for any database

$$Q^u \equiv Q'_{V_1^D,V_2^D,...,V_m^D}$$

Such a query rewriting $Q'_{V_1^D,V_2^D,\ldots,V_m^D}$ is called an *unconditionally valid query rewriting* for the user's query.

Conditionally Valid Query Rewriting[77]

This type of query rewriting is more complex and can be defined formally as follows:

Given a set of authorization views $(V_1, V_2, \ldots, V_m)$ where each authorization view V_i consists of view definition V_i^D and view instance V_i^I, a user's query Q^u can be answered by a rewritten query if the rewritten query is based on the view definitions from the authorization views (denoted by $Q'_{V_1^D,V_2^D,\ldots,V_m^D}$), and for every database instance R satisfying $V_i^D(R) = V_i^I, 1 \leq i \leq m$, we have

$$Q^u(R) = Q'_{V_1^D,V_2^D,\ldots,V_m^D}(V_1^I, V_2^I, \ldots, V_m^I)$$

Such query rewriting $Q'_{V_1^D,V_2^D,\ldots,V_m^D}$ is called a *conditionally valid query rewriting* for the user's query.

Although these two equivalent query rewriting models will not give misleading results to the users, they do suffer from several drawbacks. First, instance-based FGAC will result in a greater chance of rejecting queries. Moreover, the computational complexity for finding conditionally valid query rewriting is $\prod_2^p$ complete even under set semantics [100]. Unconditionally valid query rewriting is an application of query rewriting using views and is shown to be intractable in many cases [41].

2.4 FGAC Implementations

FGAC is typically implemented in database systems using two approaches, depending on the components of the authorization views. The first approach, namely the *view-based FGAC*, depends on the authorization view definition. The view instance in the corresponding authorization view is derived by applying the view definition on the underlying database instance. Hence view-based FGAC has authorization views in the form of $< V^D, V^D(R) >$, where R denotes the underlying database. The view definitions can be a complete query, or a parameterized query with variables bound to any constant at runtime. The second approach, namely the *instance-based FGAC*, defines each view instance in the authorization views without giving the corresponding view definition. Usually this approach is accomplished by bundling each piece of data with its access control information. The view definition V^D is the same as the original relation R, but the view instance V^I contains only the tuples that have an 'accessible' label to the user. Therefore, instance-based FGAC has authorization views in the form of $< R, V^I >$. Currently, both

view-based and instance-based FGACs are implemented in commercialized database systems [45, 66, 69, 70].

2.4.1 View-based FGAC Case Study

The Oracle Virtual Private Database [70] implements FGAC using view-based FGAC. The authorization views are user-specific, parameterized views defined by *security predicates* and *security policies*. A security predicate is in the form of $X\theta c$, where X is an attribute in a base relation, and c is a constant or a variable in the current user's profile, and $\theta \in \{>, \geq, =, \leq, <\}$. A security policy is a set of *"if-else"* statements that maps each user in the system to exactly one security predicate. Each base relation in the database is associated with one security policy. Therefore, each user has an authorization view defined on each base relation by applying his security predicate (mapped from the security policy) on that base relation. In this way, the authorization views have the same schema as the base relations.

Answering queries using the authorization views can be done efficiently as follows. The database system rewrites the user's queries by checking the security policy of each relation referred to by the query. If there is a security predicate for the user on that relation, the security predicate will be transparently appended to the "where" clause of the user's query to make sure the user only queries the accessible tuples from this relation.

To summarize Oracle Virtual Private Database using the terms described earlier in this section, the *security policies* and the *security predicates* define the *filter authorization views* for each user. The user's query is rewritten and answered under the *Truman model*.

As an example, consider a *CENSUS* table in the database with three attributes: *age, salary, citizenship*. There are three user groups: *administrator, employee*, and *trainee*. Suppose that the *administrators* are allowed to access every tuple in the *CENSUS* table, and the *employees* are allowed to access tuples having salary less than $100K$, and the *trainees* have no access to any of the tuples in this relation. Then the security policy for the *CENSUS* table can be defined using the following statement, which returns one of the three predicates based on the user's identity:

```
IF ($USER = ADMINISTRATOR)
  THEN RETURN  TRUE;
ELSE IF($USER = EMPLOYEE)
  THEN RETURN  salary>100K
ELSE IF($USER = TRAINEE)
  THEN RETURN  FALSE
```

Later, suppose a user's query is

```
select * from CENSUS
```

This query will be transparently rewritten to the following queries, depending on the identity of the user:

```
ADMINISTRATOR:  select * from CENSUS where TRUE;

EMPLOYEE:       select * from CENSUS where salary>100K;

TRAINEE:        select * from CENSUS where FALSE;
```

This view-based implementation is easy to understand and straightforward to implement. However, this implementation has the following drawback: if a user is both an *administrator* and an *employee*, the order of the *if-else* statements in the security policy decides the access control predicate. If the system allows a user to be member of multiple user groups, the security policy should consider the combined effect of several access control constraints. The other drawback of this model is that it does not explicitly support negative authorizations.

2.4.2 Instance-based FGAC Case Study

Examples of instance-based FGAC include the Label-Based Access Control (LBAC) in DB2 [46] and the Oracle Label Security (OLS) in Oracle [69]. We focus on LBAC here.

In LBAC, the accessibility of each tuple to a user is decided by the *security labels* associated with that tuple, the *security label* assigned to that user, and several *security functions* defined on the relation containing that tuple. Each such *security function* takes the tuple's security label (L_t) and the user's security label (L_u), and returns either **true** or **false**. When a user submits a query, the query executor invokes all security functions relevant to the relation referred to by the user's query. The accessibility of the tuple to the user is decided by the disjunction of all these security functions as in Formula 2.1, where a **true** value from the disjunction means the user has access to the tuple and a **false** value means otherwise:

$$F_1(L_u,\ L_t)\ \vee\ F_2(L_u,\ L_t)\ \vee\ \cdots F_n(L_u,\ L_t) \tag{2.1}$$

A security label can be one of the two types: a set of elements, or a set of nodes from a tree. Users' security labels and the tuples' security labels are always chosen to be the same type and consist of elements from the same domain. If the security labels of the user and of the tuple are sets of elements, the security function will return *true* if the user's label is a *superset* of the tuple's label, and *false* otherwise. When the security labels are a set of nodes from a tree, the security

function will return $true$ if *any* node in the tuple's security label is equal to, or is a descendant of some node in the user's security label. We can see that essentially LBAC is implementing Role-based access control with no propagation under closed-world policy.

2.5 Summary

Specifying FGAC manually at the conceptual access control matrix level can be a daunting task. Several approaches have been proposed to ease the task of this specification. Among these approaches, the Role-based Access Control specification model is the most popular one, and it has been enriched by various models and been formalized using several languages.

An implementation of FGAC should be both flexible and efficient. This means not only should it capture the expressiveness of various business requirements, but it should also have storage and runtime efficiency. However, many models proposed from academia do not have tractable runtime, and even the tractable models are not necessarily feasible to be implemented. For example, Jajodia et el [51] proposed a model that is flexible and takes polynomial time and space for processing. However, its size is still quadratic in the number of data items in the database, which makes it impractical in reality.

Compared to existing coarse-grained access control mechanisms, query evaluation with FGAC enforcement is more flexible yet more sophisticated. Several models have been proposed to describe how to evaluate queries in a secure fashion using users' accessible data. Computational models based on the *certain answers* or the *equivalent query rewritings* are intractable in general.

Because of the intractability of both FGAC specification and FGAC query evaluation, commercial database vendors propose restricted FGAC models. For example, the Truman Model imposes the restriction that each user has one authorization view for each table, the authorization view has the same schema as the underlying relation, and the answers are not restricted to the certain answer model or the equivalent query rewriting model.

View-based FGAC and instance-based FGAC are the two ways of specifying data accessibility in database systems. Compared to view-based FGAC, instance-based FGAC has more flexibility in specifying arbitrary access controls. For example, by bundling the access specifications with data, we are able to define different access controls for two tuples of the same value in the same relation. The instance-based FGAC also has the property that access controls do not change as the database is updated. On the other hand, the view-based FGAC mechanism has the advantage of storage efficiency and ease of maintenance.

Aside from having different expressiveness and maintenance costs, view-based FGAC and instance-based FGAC have different query evaluation performance. For example, while the view-based approach can be enforced by just rewriting the original query using the authorization views, the instance-based approach requires that the query processor check the access control

specifications bundled with each piece of data. Loading these access control specifications and checking each of them can be very costly.

2.6 FGAC Settings in the Thesis

This thesis addresses performance issues in database systems with FGAC. Therefore, our work focuses on improving existing FGAC implementations in database systems. For this reason, the work in this thesis is based on the Truman Model, and it considers both view-based and instanced-based FGAC for XML and relational data.

For relational data, we assume a FGAC implementation similar to the Oracle Virtual Private Database (VPD). However, we allow each user to have multiple *security predicates*. Some of the predicates specify positive authorizations, and other security predicates correspond to negative authorizations. These explicit specifications can be converted to effective access controls in an off-line manner as described in Jajodia et al. [51]. Recent work [24] gives an efficient solution for getting effective access controls from explicit specifications at query evaluation time. We argue that this model does not have the same ordering issue as Oracle VPD as outlined in Section 2.4.1, and is more flexible since it allows negative authorizations.

For XML data, we implement FGAC using instance-based FGAC that is much simpler than that of LBAC (Section 2.4.2). We do not assume any FGAC specification model, but rather start from the instance-based, effective access control specifications on an XML document. Thus we assume each node in the XML document is associated with a vector containing the accessibility information for all users under a specific action mode (reason for access). We will show how to store and maintain these data in Chapter 3.

Chapter 3

Compact Access Control Labeling for Efficient Query Processing

3.1 Introduction

As already described in Chapter 2, fine-grained access controls specify users' access rights at the granularity of individual nodes for XML data. Therefore, the size of such access control specifications can be proportional to the product of the number of nodes (elements and attributes) and the number of users. This requires an effective mechanism for specifying these rights and a compact scheme to store the access control specifications. Moreover, access control lookup must be efficient, since a single query evaluation may involve many access right checks.

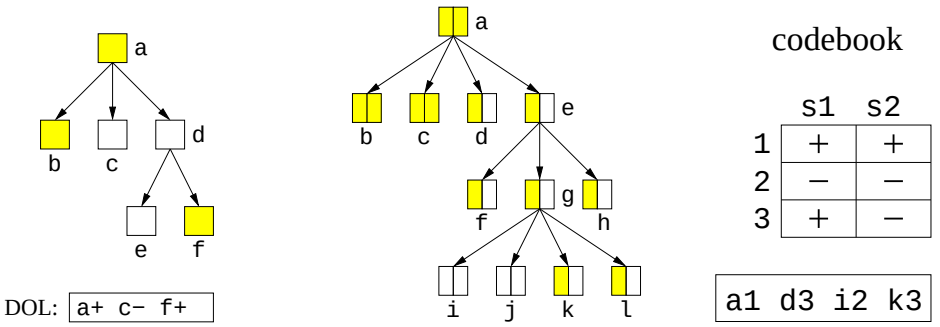
There is a plethora of literature on specifying fine-grained access controls on XML data using high level languages, as already described in Chapter 2. Deriving FGACs from *raw access controls* and *propagation rules* is much more efficient than manually specifying access control for each XML node. Meanwhile, the raw access controls and propagation rules are compact to store. However, inferring accessibility at runtime can be costly. A second approach is to materialize the net effect of these access control rules into an incrementally maintainable accessibility map, in which each XML node is labeled as either accessible or inaccessible for each subject under each action mode [96]. The key to this approach is the compression ratio of the accessibility map and the efficiency of the runtime accessibility check.

This chapter takes the second approach and addresses the problem of storing and checking the net effect of FGACs, and presents a scheme called *Document Ordered Labeling* (DOL). Like the CAM work by Yu et al. [96], DOL exploits the structural locality of the access control data to achieve compression. Unlike CAM, DOL is able to exploit correlations among the access rights of different users to achieve a substantial amount of additional compression in multi-user

environments. Moreover, DOL is a disk-oriented, multi-user scheme, while a CAM is intended to store a single user's access control data in memory.

The DOL access control representation is also highly compatible with next-of-kin (NoK) pattern matching, which is an efficient technique for processing XML twig queries [98]. NoK query processing uses a compact representation of document structure to efficiently evaluate certain types of structural query constraints (e.g. parent/child relationships). This chapter also shows how to implement secure twig query processing by integrating DOL-based access control with NoK query processing, such that access control lookup cost is reduced. We have used both real and synthetic access control data to evaluate the DOL technique. In terms of space efficiency, our results show that a single-user DOL is somewhat less compact than a single-user CAM. However, in a multi-user environment the DOL representation is much more compact than a set of per-user CAMs.

3.2 DOL: Access Control Labeling



(a) one user and his DOL

(b) two users

(c) DOL of two users

Figure 3.1: XML data with fine-grained access control and its DOL

We model an XML document as a tree in which the nodes correspond to the document's elements and the edges represent parent/child relationships among the elements. Sibling nodes in the tree are ordered.

As described in Chapter 2, our fine-grained access control model consists of a set of subjects (denoted by $\mathcal{S}$), a set $\mathcal{M}$ of action modes such as read and write, and a set $\mathcal{D}$ of nodes in the XML tree whose access is to be controlled. Here the *subjects* can be *users* or *user groups* or *roles*, which together form a *subject hierarchy*. The subject hierarchy, which describes group membership or role assignment, is assumed to be maintained separately from the XML data. We

model the net effect of a set of raw access control specifications over an XML database instance as an access control matrix with these three sets of elements forming three dimensions. For XML data, the access control matrix can be represented by associating with each XML node a list of subjects able to access that node under each action mode. In the rest of the chapter we will refer to an XML tree without a FGAC specification as a *data tree*, and to an XML tree with a FGAC specification as a *secured tree*.

We will initially assume that there is only one single action mode (i.e., $|\mathcal{M}| = 1$). We will relax this assumption in Section 3.5. We first present the DOL scheme for a single subject, and then show how to generalize it to multiple subjects. Figure 3.1(a) shows a secured tree for a single subject, and the corresponding DOL representation of the subject's access rights. Shaded nodes are accessible to the subject, unshaded nodes are inaccessible. We define a *transition node* to be a secured tree node whose accessibility is different from its document-order predecessor (i.e., its immediately preceding node in document-order). As a special case, the root node of a secured tree is also considered to be a transition node. The DOL corresponding to a given secured tree is simply a list, in document order, of the tree's transition nodes, together with their accessibilities. In the DOL shown in Figure 3.1(a), accessible and inaccessible transition nodes are labeled with "$+$" and "$-$", respectively.

Document order is, of course, one of many possible orderings one could choose. We have chosen document order for several reasons. First, since XML parsers and other tools process XML data in document order, a document order encoding of access rights can be constructed on-the-fly using a single pass through a labeled XML document. Second, NoK query processing uses a document order encoding of document structure [98], and we want DOL to be compatible with NoK. Finally, and most importantly, it allows structural locality of access controls to be exploited to reduce their size. The terms "vertical" and "horizontal" locality have been used to describe locality among parent/child nodes and among sibling nodes, respectively [96]. Structural locality is encouraged by access control specifications that propagate access rights along the hierarchical structure of the XML data, and it has been observed in real access control data [96]. Nodes that are adjacent in document order often have parent/child or sibling relationships in the document. Although this is not always the case, we expect that much of the structural locality that exists in a document's access controls will translate to locality in document order. Such locality will reduce the number of transition nodes, and hence the size of the DOL. In Section 3.6 we measure the impact of this locality on the size of the DOL using several access control datasets.

3.2.1 DOL for Multiple Subjects

Aside from the structural locality of access controls for a single subject, we conjecture that different subjects in an access control system may exhibit correlated access constraints. (There may also exist correlations among action modes. We will discuss this in Section 3.5.) For

example, users in the same department may have similar access controls. We wish to further compress the access control labeling by taking advantage of such correlations.

Figure 3.1(b) shows a tree labeled with access rights for two subjects. In the figure, each node is divided into two parts, with the left part representing the access rights of one subject and the right part representing the access rights of the other. As was the case in Figure 3.1(a), shading represents accessibility. For example, node e is accessible to the first subject but not to the second.

We can encode these access rights in much the same way as we did for a single user, by recording a list of transition nodes. With each transition node we record its access control list. Thus, when several consecutive nodes have the same access control list, we only record it once. Furthermore, we expect the access control lists for the transition nodes will reoccur frequently throughout the secured tree. We can exploit this using dictionary compression: each distinct access control list that appears in the secured tree is recorded once in a codebook (dictionary). With each transition node in the DOL we record a reference to the appropriate access control list in the codebook, rather than the access control list itself.

Figure 3.1(c) shows the multi-user DOL that corresponds to the secured tree in Figure 3.1(b). Each transition node in the list has a numeric index. This is the *access control code* (index into the codebook) for that transition node. The codebook itself contains three entries, because only three of the four possible distinct access control lists actually appear in the secured tree. Each codebook entry is an access control list, which we present as a bit vector with one bit for each access control subject.

The overall storage cost of DOL includes the distinct access control lists (the codebook), the list of transition nodes and their associated codes. The number of distinct access control lists and transition nodes depends on the correlations among subjects' access controls. If the access controls are not closely correlated, the transition nodes will be dense and the number of codebook entries will be large. Suppose the data is of size $|D|$, and we have $|S|$ single subject DOLs, each having T transition nodes (in reality, each DOL would have a different number of transition nodes, but we simplify this here). In the worst case, when subjects access controls are independent, the number of distinct access control codes in the combined multi-subject DOL would grow exponentially with the number of subjects until it reaches an upper bound of $min(|\mathcal{D}|, 2^{|\mathcal{S}|})$. Meanwhile, the number of non-transition nodes would be:

$$|\mathcal{D}| \times (1 - \frac{T}{|\mathcal{D}|})^{|\mathcal{S}|}$$

That is, as S goes up, the number of non-transition nodes would shrink exponentially until each XML node becomes a transition node. In practice, however, we expect the situation to be much better than this worst case. The real access control systems that we have studied do not exhibit worst case behavior. We will see in Section 3.6 that there *do* exist strong correlations of

access controls among subjects in these systems that make the overall size of DOL grow slowly as the number of subjects increases.

3.3 Physical Representation of the DOL

In this section we describe our physical representation of the DOL, which is intended to be incorporated into an existing query processor framework called NoK [98] to reduce disk I/O for access control lookup. For this reason, we begin with a brief overview of NoK query processing.

3.3.1 NoK Query Modeling and Physical Storage

A NoK query processor accepts twig queries described by pattern trees and evaluates them against an XML document by pattern matching. Each successful pattern match generates a set of bindings between pattern tree nodes and data tree nodes. The query result consists of all possible bindings. For example, the pattern tree in Figure 3.2(a) will generate one match from the data tree in Figure 3.2(c).

The NoK query processor first partitions the pattern tree into *NoK* subtrees, each having only parent-child or following-sibling relationships (the so-called "next-of-kin" relationships) among its nodes. Then the processor finds matches for these NoK subtrees from the data tree. Finally it matches the ancestor-descendant relationships using structural joins. For example, the pattern tree in Figure 3.2 would be split into two NoK subtrees, each matches to a fragment in the data tree. The two fragments are then connected by the ancestor-descendant relationship between nodes a and h.

The NoK query processor uses a physical representation of the data tree that allows it to match NoK subqueries very efficiently. The structure of the data tree is stored separately from the node values in a compact representation. The structure is encoded by listing the nodes in document order, with embedded markup to indicate where subtrees begin and end. For example, the structure of the data tree of Figure 3.2 would be encoded using the following string: $(a(b)(c)(d)(e(f)(g)(h(i)(j)(k)(l))))$, where the nesting of parentheses captures the nesting of subtrees. This document-order string is decomposed into blocks for storage on disk. Each block has a header with meta data for that page (e.g., the number of nesting parentheses for the first node in the page).

It can be seen that nodes connected by "next-of-kin" relationships tend to be clustered in this physical representation and thus such nodes are more likely to be located in the same physical block. By keeping the block meta data in memory and exploiting this clustered representation of document structure, a NoK query processor can match a NoK pattern using a few I/O operations [98].

3.3.2 Integrating Access Control Data

To integrate access control with NoK query processing, we physically cluster the access control data with the NoK structural data. Specifically, our scheme for physical representation of the access control data consists of the following three components (Figure 3.3):

- The DOL codebook is maintained in memory for fast accessibility lookup. If the codebook grows beyond the capacity of memory, each accessibility lookup may result in an extra physical page read for loading the codebook entry. However, our results in Section 3.6 show that, in practice, the codebook will be quite small.
- The DOL transition nodes are embedded into the NoK structural data. Figure 3.3 shows the embedding for the secured tree of Figures 3.1(b) (the page header does not show NoK meta-data for simplicity). For the purposes of the illustration, we have assumed that these data are spread across three disk blocks. In the physical encoding, we treat the first node in each block as if it were a transition node, regardless of whether it is actually a transition node. The access control code for this initial transition node is stored in the block header, which is described next. These initial transition nodes ensure that we can determine the access rights of any node using only the codes in that node's block.
- For each disk block, there is a small access control header which contains two items. The first is the access control code for the first data node in the block. The second is a "change" bit which is set if there is at least one transition node (other than the initial node) in the block, and cleared otherwise. The aggregate size of these headers is small. For example, for 1TB of XMark data [2], the NoK query processor will first convert it to a succinct physical layout as in [98], which only occupies about 10^7 blocks, assuming a 4KB block size. Measurements from our data sets (Section 3.6) suggest that 2 bytes is more than sufficient for an access control code. If we conservatively assume 4 bytes per access control block header, the total size of the headers is only 40 MB. Therefore, we can keep all of the block headers in memory (or part of the headers that include enough information for the current pages to be read), and the NoK query processor can further optimize I/O operations, as we shall describe shortly.

3.3.3 Access Lookup

To check the accessibility of a node d for subject s, the query processor locates the transition node that precedes node d (if d is not itself a transition node). Since the first node in every block is a transition node, the transition node will be found in d's block. That transition node's access control code is then used to identify an entry in the in-memory access control codebook. The s-th bit in that codebook entry indicates the accessibility of the node for subject s. As we will

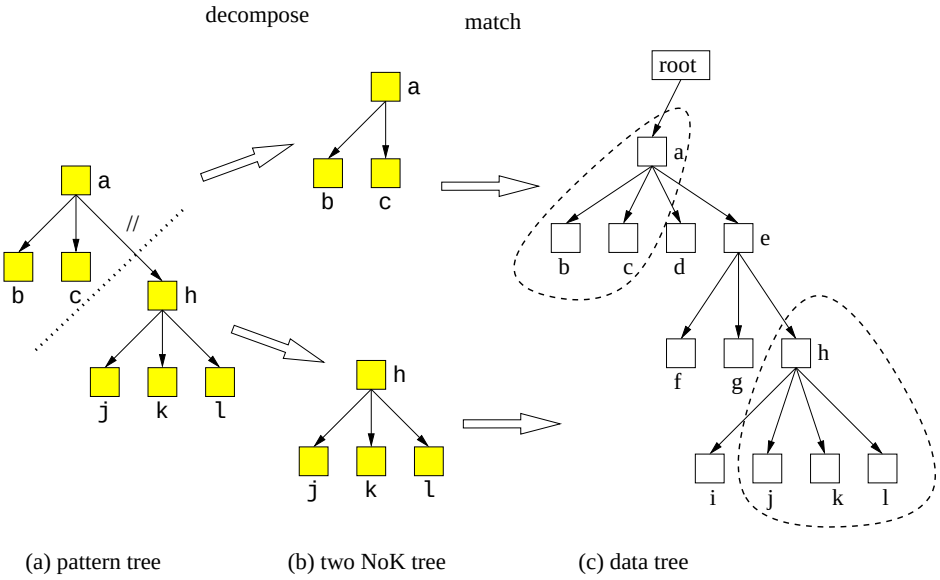


Figure 3.2: A pattern tree with two NoK trees matched to data tree

codebook

	s1	s2
1	+	+
2	−	−
3	+	−

header body

	header	body
page 1	acc=1 c=1	(a(b)(c)(d3)
page 2	acc=3 c=0	(e(f)(g)(h
page 3	acc=2 c=1	(i)(j)(k3)(l))))

Figure 3.3: DOL at physical level

see in Section 3.4, the NoK query processor checks nodes' accessibility while it matches NoK query patterns. Provided that d's disk block has been loaded for query evaluation by the NoK evaluator, the access control check for d requires no additional I/O.

In some cases, the query processor can make use of the in-memory DOL page header to avoid unnecessary page reading: if the starting transition node in the header indicates inaccessible to the user, and the "change" bit in the header is not set (meaning there are no other transition nodes in this page), all nodes in that page are inaccessible to the user. Thus the query processor can entirely avoid loading that page.

3.3.4 DOL Updates

We consider two types of access control update operations: the updates to the accessibility of nodes (*e.g.*,adding read permission for a given subject to a node), and updates to the subjects.

We first consider how to change the accessibility of a single node, x. Suppose we are to make x "accessible" for a certain subject. We first locate the nearest preceding transition node (or the node itself if it is a transition node). We will denote this node by $\hat{x}$. If $\hat{x}$'s access control code indicates "accessible" for the subject, we stop. Otherwise, we mark x as a new transition node and set its access control list to be the same as that of $\hat{x}$, except that x is accessible to the specified subject. We need to find this access control list in the codebook, and assign its access control code to node x. If the required access control list is not already in the codebook, then we need to add it. Finally, if the node immediately after x is not already a transition node, we mark it as a transition node with an access control code equal to that of $\hat{x}$. All of these operations occur in memory after loading the page containing x. Note that $\hat{x}$ is guaranteed to be on the same page as x, because the first node in every page is always a transition node. For the same reason, if the node after x is on a different page than x then it must also be a transition node, which will not require modification. Thus the I/O cost for updating x's access control list is a page read followed by a page write flushing the updates to disk.

To change the accessibility of all of the nodes in a subtree rooted at node x, we can use the above procedure to change the accessibility of each node in the subtree, in document order. The I/O cost of such a bulk update will be much smaller than one read and one write per subtree node. This is because the physical representation clusters these nodes. If each page can hold B nodes, the total I/O cost for updating the accessibility of a subtree with N nodes will be N/B pages reads (and writes). It is worth noting that all updates to DOL have the *update locality* property, i.e., an update to a subtree only affects the nodes within the pair of transition nodes that surround the subtree. This property guarantees that updates are confined within a contiguous region of the affected data.

In addition to the updates to the XML data, we need to consider updates to $\mathcal{S}$, the set of access control subjects. With DOL, it is relatively simple to add a new subject who has no (initial)

access rights, or whose access rights initially match those of some existing subject. This can be accomplished by simply adding an additional column to each entry in the in-memory codebook. No changes to the embedded transition nodes and the references are required. Deletion of a subject can also by accomplished within the codebook. This may leave unnecessary codes embedded in the structural data, since the deletion of a subject may decrease the number of transition nodes. However, any such redundancy can be corrected lazily.

3.4 Secure Query Evaluation

We use the semantics by Cho et al. [25] for secure query evaluation. The semantics is based on the Truman model. Recall that the (unsecured) evaluation of a twig query Q returns all of the possible sets of bindings of query pattern nodes to data nodes. Secure evaluation of Q for subject s eliminates from this result any sets of bindings that include data nodes that are inaccessible to s. In other words, this semantics require us replace all inaccessible nodes in the secured tree with "dummy nodes" that do not match any nodes in the query. For example, the pattern tree shown in Figure 3.2 will return a single set of bindings if nodes a, b, c, h, j, k and l in the data tree are all accessible to the subject s. It will return no bindings if any of those nodes are inaccessible to s. Note that the accessibility of nodes d, e, f, g and i has no impact on the secure evaluation of the particular query shown in Figure 3.2.

3.4.1 DOL for Secure NoK Pattern Matching

In Section 3.3.1 we described that a NoK query processor works by first decomposing a pattern tree into NoK subtrees, and then attempting to match each NoK subtree to the data. One node in the NoK pattern tree is set as the *returning node*, which means the nodes in the data tree that match this node should be returned for this pattern matching.

The secure NoK pattern matching algorithm is shown in Algorithm 1. The input parameter *proot* is the current node from the NoK pattern tree, and *droot* is the current document node that is being matched to *proot*. The third parameter R is set to $\emptyset$ initially and will contain a list of data tree nodes (in document order) that match the returning node. To match NoK subtrees, the query processor uses a recursive navigational approach, starting with an initial match from the data for the root of the NoK pattern tree. It then proceeds by recursively matching children of *proot* to children of *droot*. This top-down recursive pattern matching requires $O(|P| \cdot |D|)$ time (instead of exponential time) to find *all* matches, where P is the size of the pattern tree and the D is the size of the document [98]. The subroutines FIRST-CHILD and FOLLOWING-SIBLING use the block-oriented physical encoding of the document structure to return the first child of the *droot* in document-order, or the next sibling of the current node, respectively. The subroutine

Algorithm 1 ε-NoK Pattern Matching

NoK-Pattern-Matching$(proot, droot, R)$

Pre-condition: $droot$ is accessible; $proot$ and $droot$ match on tag and node value

```
1:  if proot is the returning node
2:     then LIST-APPEND(R, droot);
3:  P ← all children of proot;
4:  d ← FIRST-CHILD(droot);
5:  repeat
6:          if ACCESS(d) = TRUE
7:            then for each unmarked p ∈ P that matches d with
                        both node tag and node value
8:                   do
9:                      b ← NoK-Pattern-Matching(p, d, R);
10:                     if b = TRUE
11:                       then mark p in P;
12:         d ← FOLLOWING-SIBLING(d);
13:    until d = NIL or all nodes in P are marked
14: if ∃ node in P that is not marked
15:    then R ← ∅;
16:         return FALSE;
17: return TRUE;
```

ACCESS (line 6) checks the accessibility of the child of the current document subtree root to be matched. Since a node's accessibility is checked immediately after it is loaded (by FIRST-CHILD or FOLLOWING-SIBLING), and since its access control code will be found on the same page as the node itself, no additional I/O will be required for node accessibility checks. Note that the pre-condition of Algorithm 1 is the $droot$ node be accessible. This means before we use Algorithm 1 to recursively match NoK pattern trees, the root of the NoK data tree should be checked to make sure it is accessible. According to the query evaluation semantics given early in Section 3.4, we can also skip the recursion on the child if the child is not accessible.

After NoK subtree matches are located, they can be structurally joined based on ancestor-descendant relationships. We have the following theorem:

Theorem 1. *Algorithm ε-NoK, together with any non-secured structural join algorithm, securely evaluates XML twig queries.*

Proof. The NoK pattern matching algorithm returns all and only those matched pattern trees[98]. Further, NoK pattern matching together with any (non-secured) structural join algorithm will evaluate any XML twig query [98]. Algorithm ε-NoK is based on the NoK pattern matching algorithm and adds an accessibility check for each potential matching node from the data tree. This ensures that each matching NoK pattern trees reported by ε-NoK consists only of accessible nodes. Unlike the unsecure NoK pattern matching algorithm, ε-NoK terminates its recursion as soon as it encounters an inaccessible node within a potential pattern match in the data tree. Since all data nodes in a pattern match must be accessible, this early-out mechanism will not cause ε-NoK to fail to report any accessible matches. Furthermore, it will *not* cause ε-NoK to miss any accessible, matching NoK tree that is beneath the inaccessible node, since the (secure) NoK matching algorithm will be invoked for each document node that has the same node-tag as the root of the NoK pattern [98]. If there is an accessible matching NoK tree underneath the inaccessible node, it will be checked and returned by another ε-NoK invocation. Therefore ε-NoK retrieves all and only the accessible NoK patterns according to the secure query semantics. Since ε-NoK pattern matching returns only accessible matches, and since the accessibility of nodes outside of these matches is not relevant to the secure query evaluation semantics, the structural-join algorithm does not require any further accessibility checks. Therefore, the ε-NoK algorithm and any non-secured structural join algorithm will securely evaluate XML twig queries. □

3.4.2 Alternative Access Control Semantics

There is another semantics for secure query evaluation, which is defined by Gabillon and Bruno [36]. This semantics is also based on the Truman model, but is more restrictive than the semantics by Cho et al. This semantics specifies that a subtree rooted at an inaccessible node can not provide

answers, even if it contains accessible nodes. For example, the pattern tree in Figure 3.2 will not find any matches from the data tree if node e is not accessible, even if all of the remaining nodes are accessible.

Secure NoK pattern matching, as implemented in Algorithm 1, is not sufficient for enforcing this more restrictive semantics. In addition to ensuring that each NoK query pattern matches only accessible nodes, we must also ensure that when these matches are structurally joined to produce the final answer, the structural joins do not depend on inaccessible nodes. Continuing with the example from Figure 3.2, this means that we must ensure that the matches for the two NoK subqueries (rooted at nodes a and h), do not structurally join through inaccessible nodes, e.g., node e.

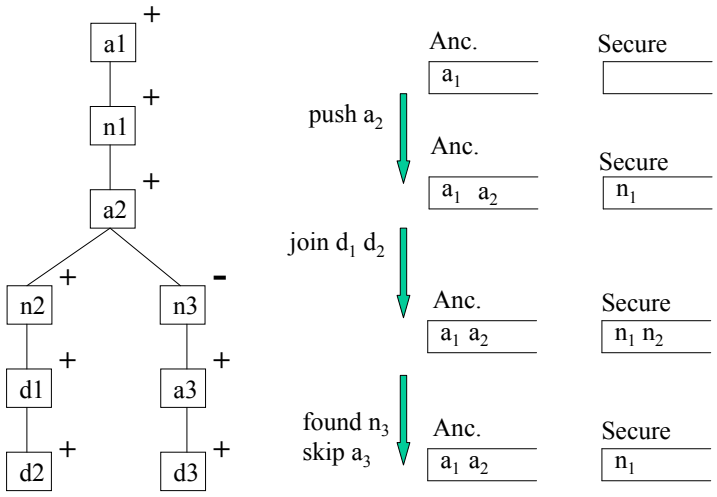


Figure 3.4: Anc stack and $Secure$ stack during structural join

We have developed a secure structural join algorithm which, together with the secure NoK pattern matching algorithm from Figure 3.2, can be used to implement the more restrictive semantics of Gabillon and Bruno if this is desired. The secure structural join algorithm must check the access control lists of all nodes on the path between the NoK subtrees that are being joined. In the NoK physical representation, these nodes are not necessarily clustered on the same physical pages as the NoK subtrees. As a result, secure query evaluation under this more restrictive semantics may be much more expensive than secure evaluation under our original semantics.

We based our secure structural join algorithm on the most efficient structural join paradigm, the *Stack_Tree_Desc* (STD) join algorithm [7]. The algorithm uses a stack (the Anc Stack) for storing ancestor nodes, with each node in the stack being a descendant of the node below it. Once a descendant node is found to be a descendant of the top of stack, it is joined with the nodes in the Anc stack all at once. Figure 3.4 illustrates the structural join of ancestor nodes $\{a_1, a_2, a_3\}$

with descendant nodes $\{d_1, d_2, d_3\}$. Since a_2 is a descendant of a_1, the Anc stack has a_1 below a_2. When d_1 and d_2 are found to be descendants of a_2 (the stack top), they are joined with a_1 as well, avoiding extra Anc-Desc relationship checking with a_1.

For secure structural join, we need to check the nodes on the path between the Anc-Des pairs. There are two accessible nodes n_1, n_2 and one inaccessible node n_3 in the figure. Both d_1 and d_2 have n_1 on the path from a_1, therefore we could check the accessibility of n_1 once for both Anc-Des pairs. On the other hand, there is an inaccessible node (n_3) on the path from a_2 to d_3. That means we could also abort the join between a_1 and d_3, since a_1 is ancestor of a_2 and can not reach d_3 via a path of all accessible nodes. We could also ignore a_3 since it has an inaccessible ancestor (n_3) and will not join with any descendants.

The above optimizations can be accomplished by using an additional stack (the Secure stack) containing the nodes that are already checked for accessibility. The complete secure structural join algorithm (ε-STD, Algorithm 2) consists of a *Stack_Tree_Desc* variant plus a security checking sub-routine (**Check**).

Before we push an ancestor node into the Anc stack, we check the nodes on the path from stack top to the current ancestor node (via sub-routine **Check**). We check the nodes in sequence, and if a node is accessible, we push it onto the Secure stack (Line 5 in **Check**), otherwise we skip the current ancestor node and all its descendants (Line 6 in **Check**). We do the same when a descendant node is joined with the Anc stack. Before we push nodes into the Secure stack, we pop out the top nodes that are not ancestors of the node to be pushed in (Line 2 in **Check**). Figure 3.4 illustrates the changing content of the Secure stack. The last change occurs when a_3 is to be pushed into the Anc stack and n_3 is to be checked for accessibility. This requires us to pop n_2 from the Secure stack since it is not ancestor of n_3. Then we find that n_3 is inaccessible and skip a_3 and d_3.

We have the following theorem for ε-STD algorithm's I/O cost:

Theorem 2. *If there are N nodes between the ancestors and descendants for structural join, and these N nodes are located on M pages, where $M \leq N$, the ε-STD algorithm loads each of the M pages at most once (in document order) for all of the AD joins.*

Proof. The algorithm only loads the nodes in one pass, thus the same page will not be loaded twice. □

We also have the following theorem for the correctness of ε-STD algorithm for securely computing structural joins:

Theorem 3. *Algorithm ε-NoK and Algorithm ε-STD together compute the alternative semantics defined in [36].*

Algorithm 2 ε-STD Structural Join

ε-STD$(\mathbf{AList}, \mathbf{DList})$

```
      Pre: AList, DList are sorted on document order
 1:   Anc ← ∅;
 2:   Let a, d be the first node in AList and DList
 3:   while a and d are not null
 4:         do while top(Anc) is not ancestor of a or d
 5:                do pop(Anc);
 6:            if a is ancestor of d
 7:              then if CHECK(a, AList) = TRUE
 8:                      then push(Anc, a)
 9:                   a = Alist.next;
10:              else  if a is before d in document order
11:                      then a = Alist.next
12:              else  if CHECK(d, DList) = TRUE
13:                      then for each a1 in Anc
14:                                do output (a1, d)
15:                           d = DList.next
```

CHECK$(n, \mathbf{List})$

```
 1:  while top(Secure) is not an ancestor of n
 2:        do pop(Secure)
 3:  for each p on the path from top(Secure) to n
 4:        do if (p is accessible)
 5:              then push(Secure, p)
 6:              else  remove n's descendants from List
 7:                    return FALSE
 8:  return TRUE
```

Proof. The *Stack_Tree_Desc* join algorithm computes all and the only matching Ancestor-Descendants [7]. The ε-STD algorithm adds a **Check** sub-routine before it stores each ancestor/descendant for matching. The **Check** routine relies on the **Secure** stack, which, at all times, contains only accessible nodes. The **Check** sub-routine returns $true$ if and only if all of the ancestors of the input node n are accessible. Therefore, both the ancestors (in **AList**) and the descendants (in **DList**) must have only accessible ancestors in order to be matched by the ε-STD algorithm. Hence, the ε-STD algorithm reports all and only those matching Ancestor-Descendants from the *Stack_Tree_Desc* algorithm that have no intervening inaccessible nodes, as required by the alternative secure query evaluation semantics. Further, from the proof of Theorem 1 we know ε-NoK retrieves all and only the accessible NoK patterns. If one node in an accessible NoK pattern tree has accessible ancestors only, every other node in the that NoK tree will also have accessible ancestors only. Therefore, the ε-STD algorithm reports Ancestor-Descendant NoK tree pairs that contains accessible nodes only, and every node in the output has accessible ancestors only. Hence the theorem holds. □

3.5 Multiple Action Modes

Access control correlation may also exist between different action modes. The correlation may come from the action mode hierarchy [53], through which, for example, a *write* access right might imply a *read* access right on a given object. However, we envision that even when there is no such action hierarchy between different action modes, there may still be correlation between different action modes. In fact, our codebook based approach could be similarly applied on multi-action modes regardless of the existence of an action mode hierarchy.

Suppose a system has multiple action modes ($\mathcal{M} > 1$). We have two choices in building codebook entries. First, we can view the $\mathcal{S} \times \mathcal{D} \times \mathcal{M}$ 3-dimensional access cube as a stack of $\mathcal{D}$ independent slabs; each slab is a $\mathcal{S} \times \mathcal{M}$ 2-dimensional matrix, holding the complete access control information for a specific object. Following this perspective, we can store one transition node for each object that has a different access control slab than its predecessor, and each transition node points to one "slab" of size $\mathcal{S} \times \mathcal{M}$ in the codebook. Alternatively, we can view the 3-dimensional cube as $\mathcal{D} \times \mathcal{M}$ 1-dimensional vectors; each vector has $\mathcal{S}$ bits, recording the complete access control information for a specific object and action mode. In this case, the entries of codebook are still vectors of size $\mathcal{S}$; but for each transition object (one that does not agree with its predecessor on the accessibility for any user under any action mode), we need to maintain $\mathcal{M}$ pointers, one for each action mode [1]. Suppose T denotes the number of transition nodes (the number of transition nodes are the same for either approaches), R denotes the size of

[1] We could also keep only the changing pointers for each transition node, but in that case, DOL lookup becomes expensive since we might need to look ahead for several transition nodes

one pointer from a transition node, V_l denotes the number of distinct access control slabs, and V_c denotes the number of distinct access control vectors. The space cost of the vector approach and the slab approach can be calculated by formula 3.1 and 3.2 respectively.

$$SIZE_{slab} = \underbrace{T \times R}_{\text{on disk}} + \underbrace{V_l \times (|\mathcal{M}| \times |\mathcal{S}|)}_{\text{in memory}} \tag{3.1}$$

$$SIZE_{vector} = \underbrace{T \times |\mathcal{M}| \times R}_{\text{on disk}} + \underbrace{V_c \times (|\mathcal{S}|)}_{\text{in memory}} \tag{3.2}$$

Both approaches have advantages and disadvantages. Since T grows with $\mathcal{D}$ and is usually much greater than $\mathcal{S}$ and $\mathcal{M}$, the overall space cost is likely to be dominated by the cost of the on-disk part. Comparing with the vector approach, the slab approach has lower overall space cost, as it has a smaller on-disk part, but it requires more memory, depending on the value of V_l and V_c. We analyze these values in Section 3.6.3.

3.6 Performance Evaluation

We evaluated the DOL technique using both synthetic and real access control data. We generated single-user, single-access-mode synthetic access controls on XML data from XMark benchmarks [2] by randomly choosing some nodes from the document as *seeds*, and then labeling these seeds as accessible or inaccessible. We simulate horizontal structural locality by selecting nodes from among the seeds' direct siblings and set them with the same accessibility as the seeds, provided that the siblings are not themselves seeds. Then, we simulate vertical structural locality by propagating the accessibilities of the labeled nodes to their descendants using the Most-Specific-Override policy [51], i.e., a node inherits its accessibility from its closest labeled ancestor. We always choose the document root as a seed to ensure that all nodes are labeled. Access control generation is controlled by two parameters. The *propagation ratio* determines percentage of nodes that are seeds while the *accessibility ratio* determines the percentage of seeds that are accessible.

In addition, we used two sets of real multi-user access control data. The first data set describes the fine-grained access control information from a production instance of OpenText LiveLink [2], which provides web-based collaboration and knowledge management services in a corporate intranet. The LiveLink system has a total of 8639 subjects (among which there are 1568 leaf users), around $370,000$ data items in a tree-structure with an average depth of 7.9 and a maximum depth of 19, and ten action modes. The second data set consists of the access control data from

[2]LiveLink is a trademark of OpenText Corporation.

a multiuser Unix file system at the University of Waterloo. This system has 182 users and 65 user groups, more than 1.3 million files/directories, and three action modes (read, write, execute). Although neither of these systems stores real XML data, both provide tree-structured data models and instance-level access controls. In the absence of real access controlled XML data, we treat these systems as surrogates for real multi-user access controlled XML databases for the purposes of our experiments. In order to map our unordered tree data to ordered XML nodes, we pick one arbitrary ordering among sibling nodes in our test data: files under the same directory are ordered by their names, and sibling LiveLink data items are ordered by their appearance in the LiveLink system catalog. These two ordering choices are orthogonal to access controls, but the structural locality in the access controlled data still holds, as we shall see from the experiments.

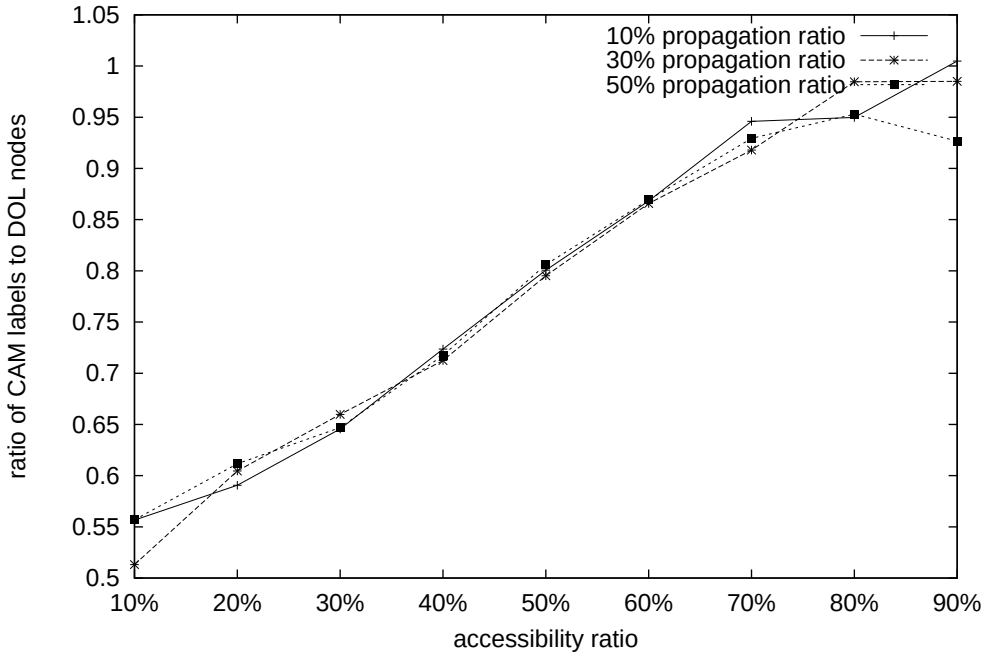
3.6.1 Space Efficiency in Single-User Environments

We first evaluate DOL for a single subject. Since CAM [96] is the state of the art compact labeling for single subject access controls, we compare DOL with CAM. We first use an XMark document of about $17,000$ nodes with synthetic access controls produced by different accessibility and propagation ratios. Our metric is the ratio of the number of CAM nodes to the number of DOL transition nodes (the codebook size is negligible for one subject). Thus, values less than 1.0 favor CAM and those greater than 1.0 favor DOL.

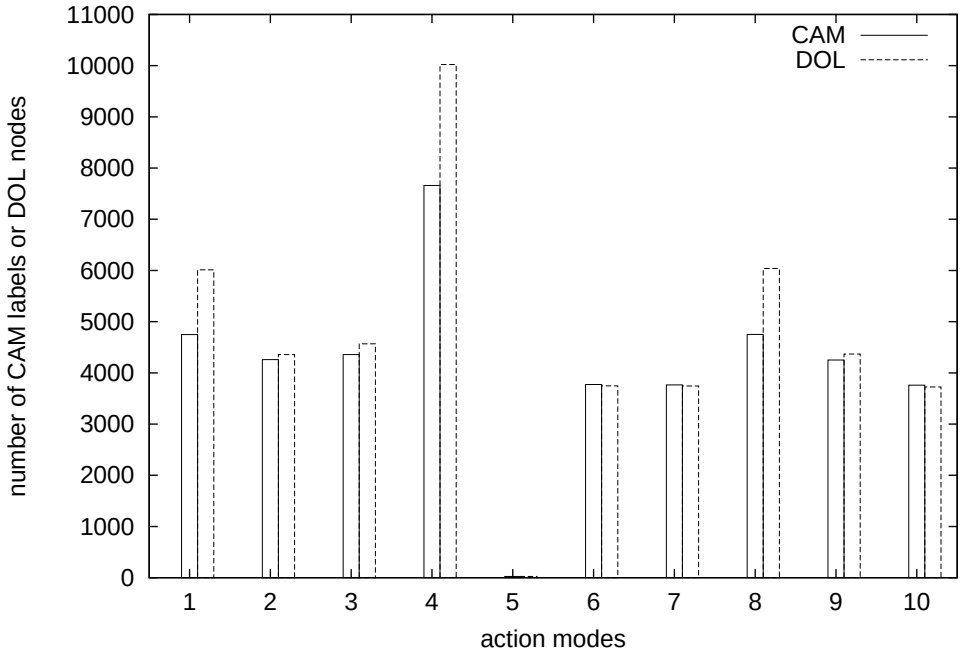
Figure 3.5(a) shows the comparisons as the accessibility ratio varies from 10% to 90%. We tried three propagation ratios with these different accessibility and the results are similar. When the accessibility ratio is low (few nodes are accessible), the number of CAM nodes is around 53% of the number of DOL transition nodes. As accessibility goes up, this difference becomes smaller.

We also compared single user CAM and DOL using the LiveLink data. The LiveLink system supports ten different action modes. For each of the ten action modes we sample a number of users and built CAM and DOL for each single user. The ratio of the number of DOL transition nodes to the number of CAM labels for an average user is shown in Figure 3.5(b), with the ten action modes shown on the horizontal axis. In the worst cases, DOL had 20-25% more nodes than CAM labels. In other cases, CAM has about the same number of labels as DOL transition nodes.

Note that our performance metric implicitly favors CAM because it assumes that CAM nodes and DOL transition nodes are the same size. In practice, however, the DOL nodes are likely to be much smaller. This is because CAM stores the access rights separately from the data. As a result, each CAM node must include a reference to a document node and pointers to the node's children in the CAM, in addition to the access control information itself. In contrast, DOL, which piggybacks access control information into the document encoding, stores only an access control code per transition node. Thus, although CAM may have fewer nodes than DOL, the total space



(a) synthetic data



(b) LiveLink system

Figure 3.5: CAM labels and DOL transition nodes for single subject

required for CAM may be greater.

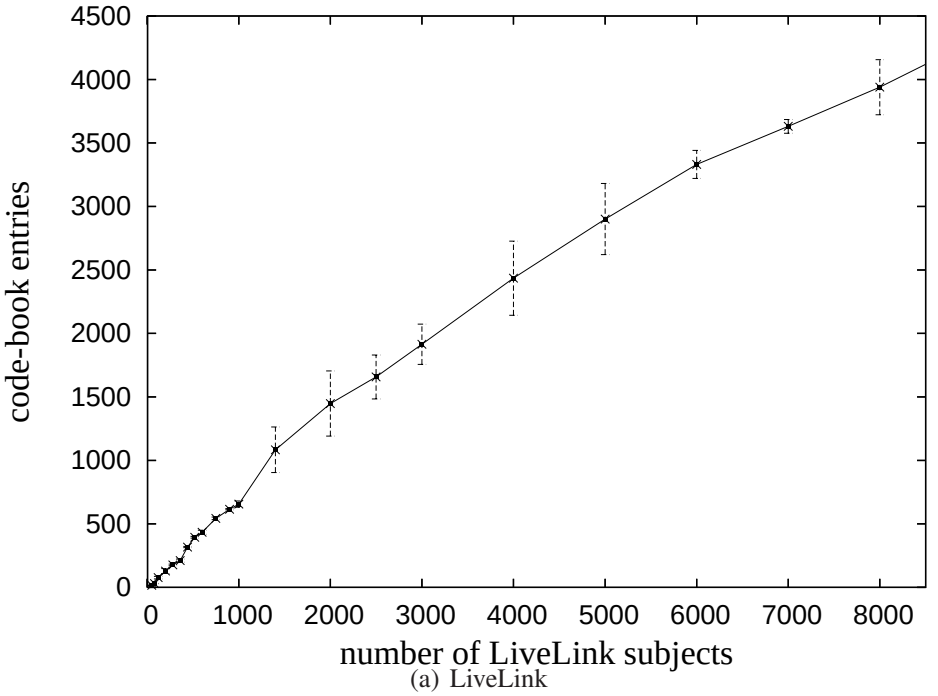
3.6.2 Space Efficiency in Multi-User Environments

We used our two real data sets to evaluate the space efficiency of DOL in multi-user environments. We only present the results under "SEE" action mode of LiveLink and the "READ" action mode of the Unix file system, since other action modes of both data show similar trends. To get a sense of how the codebook size might vary as a function of the number of subjects, we selected a number of subjects randomly and computed DOL codebooks for the selected subjects only. In Figures 3.6(a) and 3.6(b) we have plotted the number of codebook entries as a function of the cardinalities of these selected group of subsets. The X-axis shows the number of randomly chosen subjects in the group. Each group is a subset of the next larger group. We tried three independent runs. The lines in these figures show the mean value of number of codebook entries over the three runs. The min and the max values at each point over the three runs are shown as the error boxes. If subjects' access controls were uncorrelated, we would expect to see exponential growth in the number of codebook entries as the number of subjects increased. However, our results show that the growth is much slower in practice. With all 8639 subjects, the LiveLink system required around 4000 codebook entries. At approximately 1100 bytes per codebook entry (one bit per subject), the complete LiveLink codebook would occupy only about 4.4MB of memory. The Unix system required about 855 codebook entries for 247 subjects, with an overall size of only 25KB.

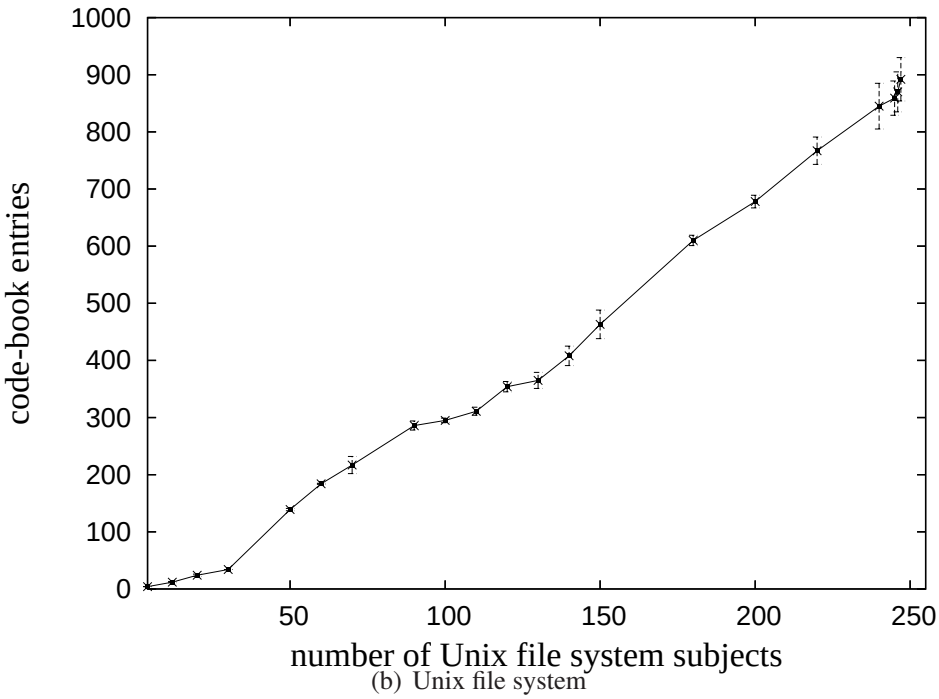
The other major storage concern is the number of DOL transition nodes. Figures 3.7(a), 3.7(b) show the numbers of transition nodes required for the LiveLink and Unix systems as the number of subjects increases (using the same methodology that was used for Figures 3.6(a) and 3.6(b))[3]. Figure 3.7(a) shows a sub-linear growth of transition nodes. For over 8000 subjects, the number of transition nodes is only about 4 times larger than the number for a single subject. For Unix file system, we see a similar situation in Figure 3.7(b), in which the number of transition nodes of 247 subjects is only twice as many as the number for 50 subjects. Recall there are about $370,000$ objects in the LiveLink system, and the number of files (directories) in the Unix system is about 1.3 million. Thus, the density of transition nodes is less than 10% for both systems (for all the subjects). We tried different subsets of subjects and the result is similar. These results indicate that the access rights for different subjects are highly correlated in real world.

To compare the overall storage cost between DOL and CAM, we first look at a single subject in LiveLink (under action mode 1): DOL needs about 6000 transition nodes while CAM needs 4500 labels. However, for all 8639 subjects in the same system under the same action mode, DOL needs 18800 transition nodes while CAM needs 8639×4500 labels, a difference of three orders

[3]We tried different subset of users in each run, and the number of subjects varies within 10% between each runs

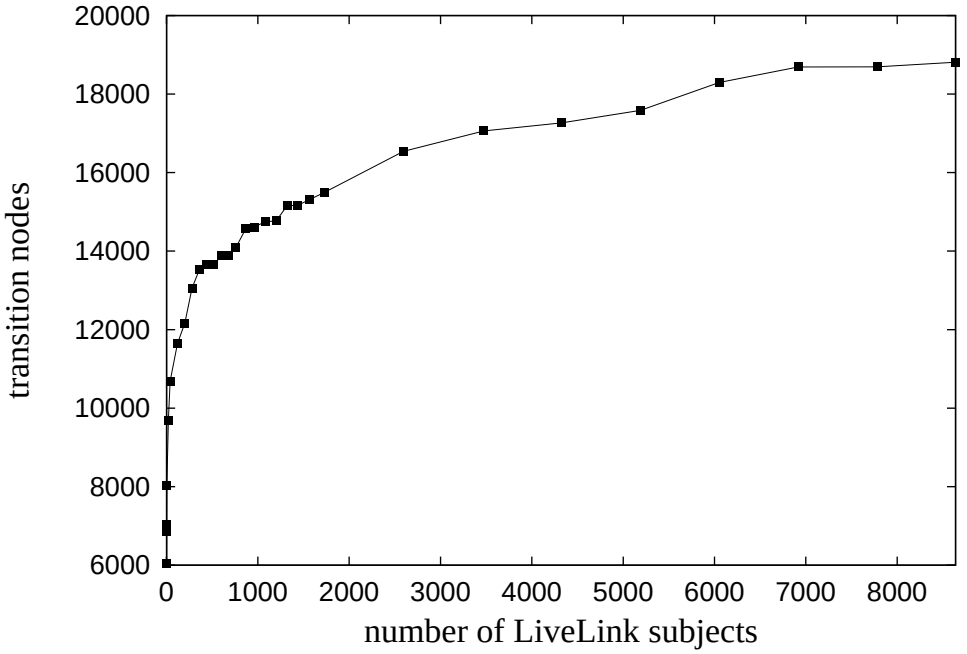


(a) LiveLink

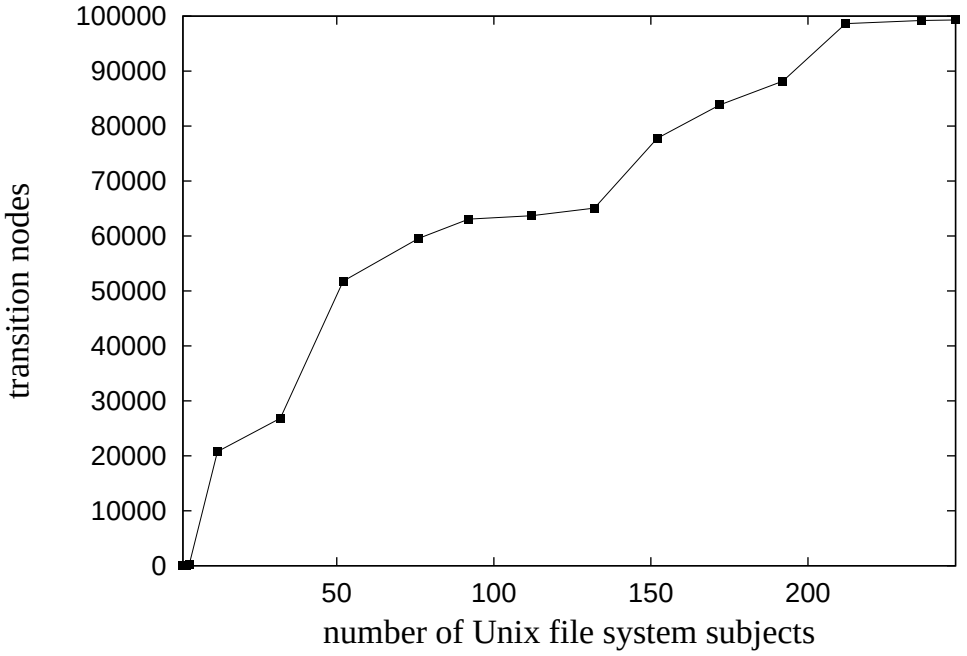


(b) Unix file system

Figure 3.6: Codebook entries as a function of the number of subjects



(a) LiveLink



(b) Unix file system

Figure 3.7: Transition nodes as a function of the number of subjects

Action Mode	***read***	***write***	***execute***	***all***	***slab***
CodeBook#	855	883	857	995	1608

Table 3.1: Codebook entries for vector/slab-based approaches (Unix system)

of magnitude. Assuming each DOL transition node requires a 2 byte access control code (for the 4000 codebook entries), and each CAM label takes 2 bits for its accessibility encoding, and (unrealistically) only 1 byte for node pointers, the DOL's total space requirement will be a 4MB codebook plus 40KB of embedded transition nodes, while CAM's will be 46.6MB. The Unix file system's situation is similar. Clearly, correlation among the subjects contributes substantially to compression effectiveness.

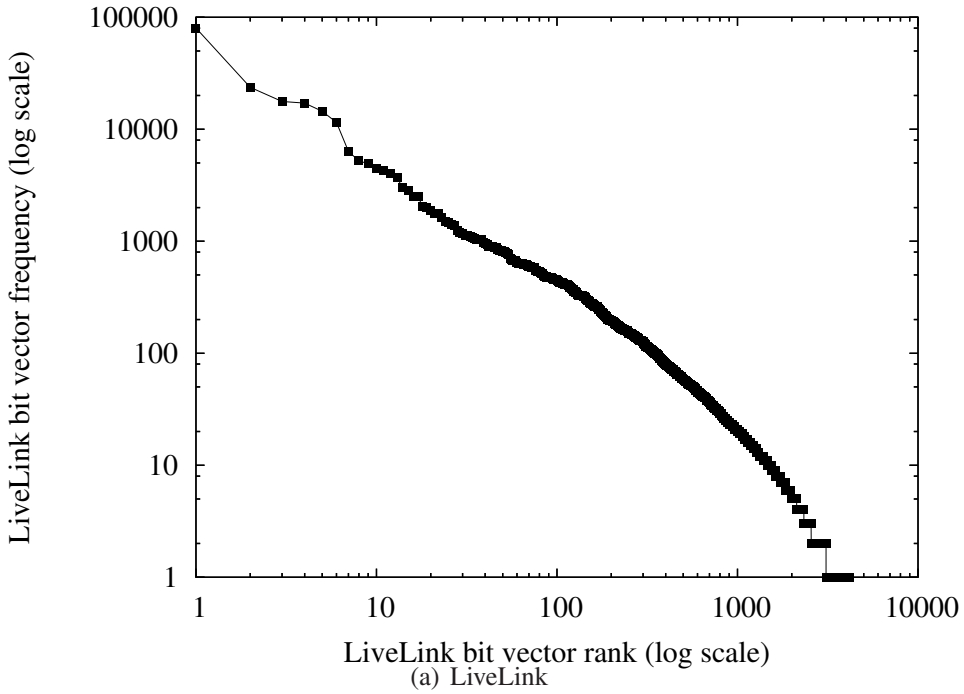
A final observation, from both systems, is that the frequency of the access control lists in the codebook is highly biased. Figure 3.8 shows the rank/frequency plots of the access control list in the Unix and LiveLink Systems. Here we choose "Read" action mode for Unix and action mode 1 for LiveLink since they are representative of the remaining action modes. In the Unix system, about ten access control lists account for more than 80% of all the files' access controls. A single access control list (granting access to everyone) accounts for more than half of the access controls on all the files. Similarly, in the LiveLink system, ten access control lists account for about half of the all objects' access controls.

This frequency skew, together with the locality of access rights, helps to explain the space efficiency of the DOL approach. It also suggests that, in small-memory environments, good performance can be achieved without keeping the entire codebook in memory. If only the most frequently used access control lists are kept in memory, it should be possible to answer most access control lookups efficiently. In our experiments with LiveLink data, if we load in memory 1000 most frequently entries out of all the 4198 entries of the codebook, we can answer more than 95% of the accessibility queries without loading any codebook entries from the disks.

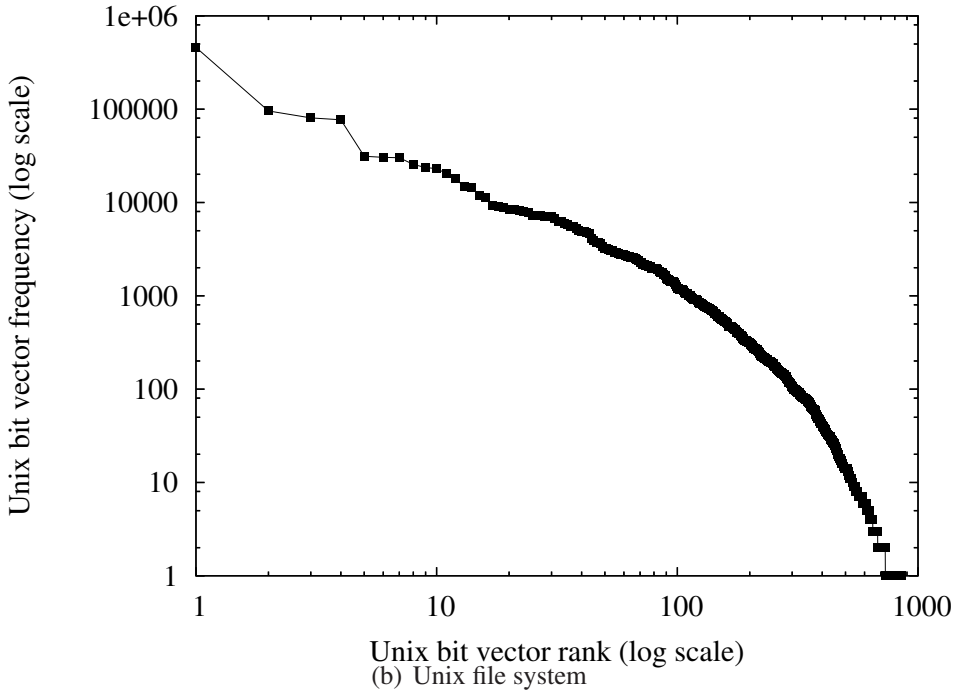
3.6.3 Multiple Action Modes: Vectors Versus Slabs

Recall that the total space cost of the codebook scheme includes two parts: the on-disk part occupied by the access control codes and the in-memory part occupied by the access control codebook. When there are multiple action modes, the DOL can be implemented either as a codebook with access control vectors for all action modes (where each transition node will have multiple pointers to different codebook entries for each action mode) or as an array of access control slabs (where each transition node points to one slab entry for all action modes), as described in Section 3.5.

We first tested the vector approach on the Unix file system. Table 3.1 illustrates the number of distinct access control vectors (i.e., the number of codebook entries) for read, write, and execute



(a) LiveLink



(b) Unix file system

Figure 3.8: Rank frequency of access control lists for Unix system and LiveLink system

modes, together with the number of distinct codebook entries across all three action modes (the ***all*** column). We can see the total number of distinct access control vectors of all three action modes is much fewer than the sum of the number of distinct vectors under the individual action modes. This tells us that the access control vectors of different action modes are similar.

Next, we tested the slab approach by concatenating the access control vectors into slabs. If the access controls under different action modes are not correlated, we would have seen all combinations of concatenated access control vectors in the slabs, making the number of distinct slabs grow exponentially in the number of action modes. However, as we observed from our real data, the number of distinct slabs is only about four times the number of distinct access control vectors (see column ***slab*** in Table 3.1). Since the Unix system has 247 subjects, each access control vector would require 247 bits, leading to a total codebook size of approximately 31 KB under the vector-based approach. Since there are three access modes, each slab would require 741 (247×3) bits, giving a total codebook size of 149 KB under the slab-based approach. Therefore, the codebook of the slab approach is only around five times the size of the codebook under vector approach.

Action Mode	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*all*	*slab*
CodeBook#	4235	3926	5591	4273	4144	4362	4684	22	4325	4198	9863	10976

Table 3.2: Codebook entries for vector/slab-based approaches (LiveLink system)

We noticed an even stronger correlation between action modes in the LiveLink data (Table 3.2), where the number of distinct access control vectors for all action modes is less than 25% of the total of all distinct access control vectors under each action mode, and the number of distinct slabs is almost the same as the number of distinct access control vectors for all action modes. Since the LiveLink system has around 8600 subjects, each access control vector would require 1075 bits, thus the vector-based codebook has size of 10.6MB (1075×9863). The slab-based codebook has each entry as ten vectors concatenated, thus its size is approximately 118MB ($10976 \times 1075 \times 10$). Therefore, the slab-based codebook is only about ten times the size of the combined vector-based codebook.

Aside from codebook entries, we can be sure that the density of transition nodes for the slab approach is the same as the vector approach, since either approach would need a transition node whenever the accessibility changes for one user under one action mode. However, under the vector approach, each transition node needs $\mathcal{M}$ pointers to the entries of the codebook for the $\mathcal{M}$ action modes, whereas the slab approach only requires one. Considering that the number of transition nodes grows with the number of objects in the system (recall from Section 3.6.2 that transition nodes are around 10% of the total objects for all users for both real data sets), the size of transition nodes would dominate the size of the codebook by several orders of magnitude. Thus we could save considerable space by using the slab based approach at the cost of a (moderately)

larger codebook in main memory.

3.6.4 Query Evaluation

We compared the performance of secure NoK query evaluation to the performance of non-secured NoK query evaluation to measure the overhead of fine-grained access control. We implemented both ε-NoK, ε-STD, and the non-secure versions of the NoK and STD algorithms using Java 1.5. All of the experiments were conducted using a dedicated PC with a Pentium III 997MHz CPU, 512MB RAM, and 40GB hard disk running Windows XP.

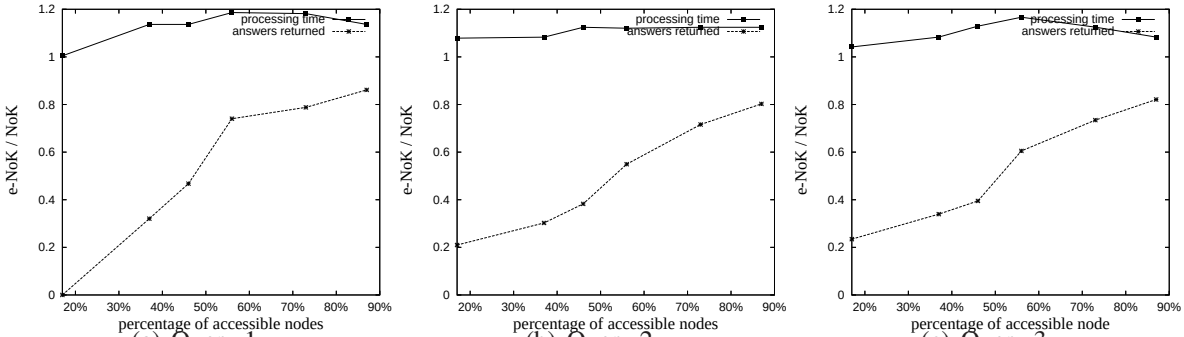
Our test database is a 50MB XMark instance (832911 element nodes) with synthetic access controls. The data are stored on disk with each page at 4K bytes. The benchmark queries are shown in Table 3.3. The top three queries represent three classes of NoK pattern trees: those with branches at the end (Q1), in the middle (Q2), or a single path (Q3). The bottom three queries are for ancestor-descendant structural joins and represent those having descendants located close to the ancestors(Q4), far from the ancestors (Q6), or at a medium distance (Q5). To ensure that file system and other level of caching does not affect our experiments, we force the system to read a large file whose size exceeds main memory between consecutive queries.

Q1	/site/regions/africa/item[location][name][quantity]
Q2	/site/categories/category[name]/description/text/bold
Q3	/site/categories/category/description/text/bold
Q4	//parlist//parlist
Q5	//listitem//keyword
Q6	//item//emph

Table 3.3: Sample queries

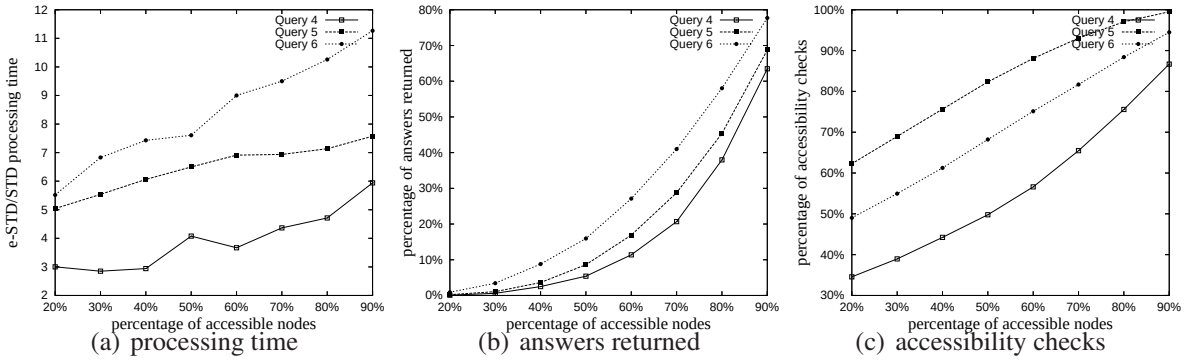
Figures 3.9(a), 3.9(b) and 3.9(c) show the performance of the ε-NoK algorithm. We measure the total wall clock time required (CPU time plus I/O plus other latency) for both algorithms, by taking the difference between the time stamp of submitting the query and the time stamp of getting the complete results back. The result set is not large so the time to output the results is negligible. The two lines in each figure depict the ratio of processing time and answers returned between the ε-NoK and non-secure NoK algorithms. The processing time of the ε-NoK algorithm is at most 20% greater than the processing time of the non-secure NoK algorithm, and does *not* depend on the accessibility ratio. This is still true when the majority of the document is accessible (thus most answers of the original NoK algorithm are returned, and nodes in these answers are all checked). The reason is that accessibility checking does not require extra I/O for the ε-NoK algorithm.

Figures 3.10(a),3.10(b) and 3.10(c) show the performance of the secure ε-STD algorithm. Figure 3.10(a) shows the ratio of processing time between the ε-STD and non-secure STD algorithms for Queries 4, 5, and 6. As we can see, ε-STD is much more expensive. However, when the number of answers decreases as node accessibility decreases (in Figure 3.10(b)), the corresponding processing time also goes down. Also, we note that processing time decreases faster for the queries with longer ancestor-to-descendant paths. This shows that ε-STD is more sensitive to node accessibility and query topology, and its performance is better when majority of the document is inaccessible. The reason is that the ε-STD algorithm can optimize by skipping the accessibility checks for some nodes on the ancestor-to-descendant path. This is verified in Figure 3.10(c), where the y-axis shows the percentage of nodes (between each ancestor-descendant pairs) checked with accessibility by ε-STD Algorithm. As we can see, the number of accessibility checks goes down (at different speed for three queries) as node accessibility goes down.



(a) Query 1 (b) Query 2 (c) Query 3

Figure 3.9: Performance between ε-NoK and NoK as a function of node accessibility



(a) processing time (b) answers returned (c) accessibility checks

Figure 3.10: Performance between ε-STD and STD as a function of node accessibility

3.7 Comparison to Related Work

Gabillon and Bruno [36] define view-based semantics for secure query evaluation. They define a secured view for each user group, and queries are applied against the secure views. Since their approach is based on view-based access controls (see Chapter 2), the FGAC specifications do not take much space. However, their approach prunes all subtrees with an inaccessible root, regardless whether there are accessible nodes in that subtree or not. Cho et al. [25] define a more relaxed pattern-matching based semantics. This semantics allows answers to come from a subtree whose root is inaccessible. They use schema information to rewrite queries to optimize query evaluation time. Stoica et al. [85] use secured views and secure data schema (both generated from original data schema and access control policies) for answering XML queries. Similarly, Fan et al. [34] use secured views generated from DTD schemas and access control rules for secure query evaluation. The access control rules in these two approaches are based on the data schema, which must exist. Moreover, all these models are also built on view-based access controls, and have the limitation as described in Chapter 2. On the other hand, instance-based access control is necessary when there is no schema, or when the desired access controls cannot be described by schema level specifications.

Lee et al. [57] propose a secure XML evaluation framework in which access controls are specified on the XML model but enforced in an underlying relational database. However, the XML data instance needs to be first shredded into the relational model. There is also work [30] on efficient dissemination of sensitive XML data based on pattern matching. The DOL approach can be similarly used for dissemination of XML data to multiple users. The difference is that DOL is based on instance-based fine-grained access controls.

Yu et al. [96] developed the CAM representation for fine-grained access controls on XML data. A separate CAM structure is required for each user under each action mode, so CAM does not exploit the commonality among multiple users to achieve better compression ratio. Recently, Jiang and Fu proposed Integrated Compressed Accessibility Maps (ICAM) as an improvement over the original CAM approach [53]. Like DOL, this approach exploits correlations between different action modes and compresses the CAMs for different action modes into one integrated structure. However, the approach also requires that the action modes follow a strict *operational hierarchy*, i.e., a subject with write access rights on an object always has read access on that same object. This is not always desireable. e.g., a user may be able to write to a file while not being able to read it, as in the Mandatory Access Control model [82]. The DOL, in contrast, neither requires nor exploits an operational hierarchy.

Chapter 4

Intelligent Query Plan Caching

4.1 Introduction

Under the Truman model described in Section 2.3.1, each user's query will be applied on his accessible data only. If access controls are view-based, the queries will be rewritten to be applied on the views containing the user's accessible data. We already described in Section 2.4.1 that Oracle VPD implements view-based access control under the Truman model by appending access controls as predicates to the $where$ clauses in user queries. In this way, one query can be customized into different user-specific versions corresponding to different users' access controls. We use the term *customized query* to denote such a query. In this chapter we consider the problem of query optimization for such customized queries.

Because access controls can be based on personal data or user profiles, the number of *distinct* customized queries from the diversified access controls can be as large as the number of users. These different customized queries pose new challenges to the query optimizer. Suppose a broker (u_1) submits a query Q, searching for any stock whose price chart shows a double formation (or M-Top pattern) in the past week. The system first customizes this query Q into Q^{u_1} according to the broker's access rights for stock information, and then computes the optimal query plan P_1 for Q^{u_1}. Since the uncustomized query is already complex [27], optimization of the customized query may take some time. Suppose another broker (u_2) submits the same query Q, which gets customized to Q^{u_2}. Q^{u_2} is largely the same as Q^{u_1} except that Q^{u_2} manifests different access control customization than Q^{u_1} if u_2 has different access rights. If the system is to optimize Q^{u_2} from scratch, the cost of optimization has to be paid again.

Suppose there are $\mathcal{Q}$ queries and $\mathcal{U}$ users, and each user submits his customized version of $\mathcal{Q}$, it will take $\mathcal{Q} \times \mathcal{U}$ time to optimize all possible customized queries from scratch at runtime. We call this approach the "Näive Dynamic" approach or *ND* as shown in Figure 4.1. The ND

approach is expensive and unsuitable for the online nature of these queries.

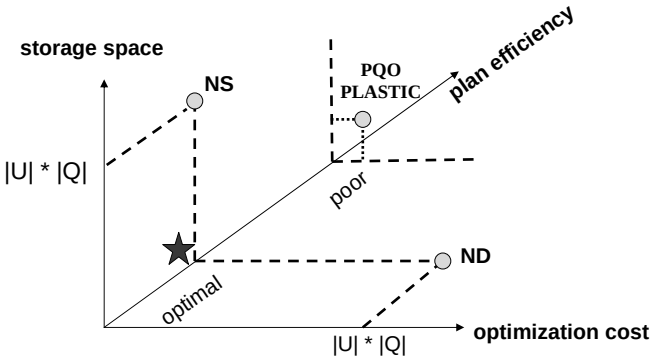


Figure 4.1: Query optimization for multiple users

Our objective is to reduce the optimization cost of the ND approach, while continuing to produce high-quality query plans. There are several existing techniques that are potentially applicable to this problem:

1. Cache query plans so that the same query plan can be used for exactly the same query if it is submitted again. However, this requires exact matching on queries, and does not help if all these customized queries different from each other in both syntax and semantics.

2. Compute a query plan for each customized query in advance. When user u submits query Q, the system locates the pre-computed plan for Q^u and executes it. In this way, the run-time cost of query optimization is near zero. However, this approach requires that we know all customized queries at query compilation time, and the users' access controls remain fixed thereafter, which may not always be the case. Moreover, this approach requires space proportional to $|\mathcal{Q}| \times |\mathcal{U}|$ to store all query plans, which is prohibitive when $|\mathcal{U}|$ is large. We call this approach the Näive Static approach or *NS* in Figure 4.1.

3. Prepare query plans offline in a manner similar to parametric query optimization (PQO) [43, 44, 49]. However, we will show in Section 5.8 that this PQO approach fails to compute efficient query plans even if it is enriched by extra means.

4. Reuse query plans in a way similar to *plan selection with query plan clustering* (PLASTIC) [39]. This is not a suitable approach, as we will show in Section 5.8, since it depends on query structure.

In the context of user-customized queries, each of these approaches has a significant weakness. We hope to overcome these limitations with the approach presented here.

4.1.1 Approach Overview

This chapter presents an intelligent plan caching mechanism, called Partitioned Optimization for Multiple-user (POM), as a technique for optimizing a large number of customized queries. Our observation is that although the customized queries are different from each other in syntax and semantics, they stem from the same uncustomized query. Therefore their query plans *may* only differ at the parts that implement customizations. In this case, partial plans can be reused, avoiding re-optimization of customized query versions. For example, if the customizations are limited to the access paths (e.g., leaves of the query plans), the whole upper-level join trees of the query plans can be reused.

We study the characterizations of customized queries that share the same upper-level partial query plan. Based on the study, we are able to cache and reuse the partial plan from previously optimized queries for new queries. With a careful design of the partial plan reuse algorithm, we propose POM as an intelligent plan caching mechanism that

1. reduces query optimization cost compared to the ND approach, if the partial plan corresponds to a time-consuming component of a query plan,
2. achieves a *much higher* cache hit ratio than a traditional plan caching mechanism, and guarantees superior performance of the query plan built from the cached partial plan, and
3. has an adjustable cache size which does not have to be as large as the NS approach.

The proposed POM plan caching mechanism is illustrated by the star in Figure 4.1.

4.2 Example of Query Plan Reuse

We will use the following running example for the rest of the chapter.

Example 1. *There are three customized queries as follows, with their customizations shown in bold font.*

Q_1 select * from R, S where R.a=S.b and **R.a=5 and S.b=5**

Q_2 select * from R, S where R.a=S.b and **R.c<5**

Q_3 select * from R, S where R.a=S.b and **R.a<5**

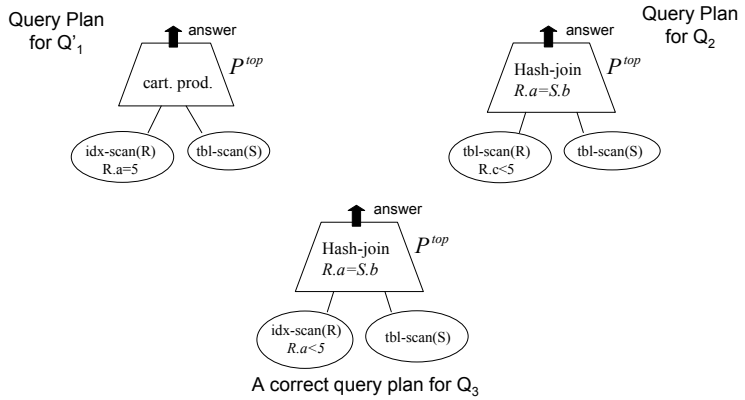


Figure 4.2: Example of reusing partial query plan

Assume there is an unclustered index on attribute $R.a$.

Suppose Q_1 is submitted first and the optimizer rewrites Q_1 into Q_1' by removing the redundant join predicate:

Q_1': select * from R, S where R.a=5 and S.b=5

The optimizer computes Q_1's optimal query plan based on this rewriting as a Cartesian product over an index-scan on $R.a$ and a table-scan on S, as shown on the top left of Figure 4.2.

Then Q_2 is submitted and the optimizer computes Q_2's optimal query plan as a hash-join over a table-scan on R with select condition (R.c<5) and a table-scan on S, as shown on the top right of the figure.

Now suppose Q_3 is submitted. If we modify Q_2's query plan by replacing its left child with an index-scan on $R.a$ together with the select condition ($R.a < 5$), the modified query plan, which is shown on the bottom in Figure 4.2, correctly *computes Q_3 . However, we are not sure if this query plan is the* optimal *query plan for Q_3.*

On the other hand, if we try to replace the left subplan of Q_1's query plan with an index-scan on $R.a$ having a select condition ($R.a < 5$), the modified query plan may not compute the correct answer for Q_3. The reason is that Q_1 experiences different query rewrites than Q_3, and its query plan does not have the join predicate that Q_3 depends on.

The above example shows that under some scenarios we are able to modify the query plan of a customized query to create a query plan that computes a different customized query. However, we may not do so if the customized queries have gone through different query rewrites. Example of such query rewrites includes removing the $distinct$ operator, adding or removing join predicates, or replacing sub-expressions with materialized views. Moreover, even if we are able to modify one customized query's query plan to compute the correct answer for another customized query, we may not guarantee the *optimality* of the modified query plan for the new

query. Therefore, we aim to provide answers to the following questions:

1. Under what situation can we substitute parts of an existing query plan to get a correct query plan for a new customized query?

2. Will the modified query plan be an optimal query plan for the new customized query?

Note that the queries in the example are deliberately made simple for presentation purposes. We envision our proposed solution will be beneficial for complex queries that are time-consuming for query planning.

4.3 Foundation of Query Plan Substitution

Before introducing our plan caching and plan modification mechanism, we need some insights into query plan composition. We first observe that a query including joins over several tables can be expressed as a top-level query over several subqueries, where each subquery is either a single table or a join over several tables, i.e.

$$Q \equiv Q_{top}(Q_1, Q_2, \ldots, Q_m)$$

Correspondingly, the optimal query plan for this query consists of a top-level plan and several subplans, where each subplan is an access path on a relation or a sub-join query plan, i.e.

$$P \equiv P_{top}(P_1, P_2, \ldots, P_n)$$

Suppose we have another query Q' whose query plan is P'. Moreover, Q' and P' can be expressed as $Q'_{top}(Q'_1, Q'_2, \ldots, Q'_{m'})$ and $P'_{top}(P'_1, P'_2, \ldots, P'_{n'})$ respectively. We have the following proposition:

Proposition 4.3.1. *If the following conditions hold between Q and Q' and between P and P':*

1. *There is a one to one mapping between subqueries in Q and subqueries in Q' such that each pair of subqueries has the same schema.*

2. *There is a one-to-one mapping between the subplans of P and the subqueries in Q such that each subplan P_i implements the corresponding subquery Q_i; and the same applies to P' and Q'.*

3. *The top-level sub-queries Q_{top} and Q'_{top} are identical.*

4. *Each subplan P'_i is* physically compatible *(to be explained later) with P_i with respect to the top-level query plan P_{top} of P.*

then we are able to create a new query plan $P'' = P_{top}(P'_1, P'_2, \ldots, P'_{n'})$ such that P'' computes Q', as illustrated in Figure 4.3.

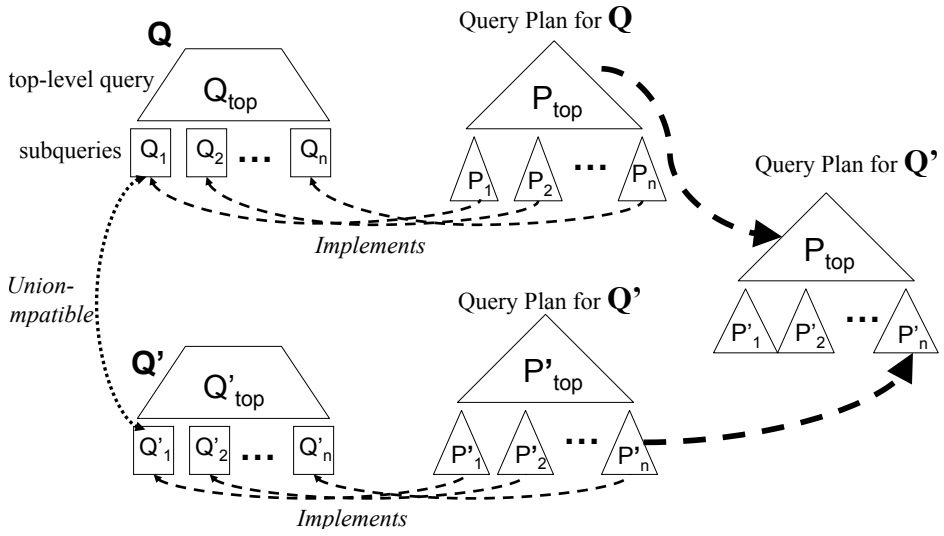


Figure 4.3: Subplan replacement

Figure 4.3 illustrates this proposition. As mentioned in the proposition, when connecting the subplans of P' to the top-level part P_{top} of P, we need to consider whether the subplans are *physically compatible* with the top-level query plan at the connecting points. Each database system has its own specific requirements on what types of plan operators are *physically compatible*. We will illustrate this in Section 4.3.1.

Definition 4.3.1. *A subplan P'_i is a **compatible substitute** for subplan P_i with respect to query plan P if*

1. *P'_i and P_i have the same schema.*
2. *P'_i is physically compatible with P_{top} of P at the point at which P_i connects to P_{top}.*

Proposition 4.3.1 suggests a plan caching mechanism, which can be applied for computing query plans for a large number of customized queries. Suppose we have Q and Q' as two customized queries, and we have already computed the optimal query plan for Q and cached its

top-level query plan P_{top}. Now we are able to compute a query plan P'' for Q' by reusing P_{top} and connecting it with the subplans of Q' that are *compatible substitutes* to the subplans in P with respect to P_{top}. If the subplans of Q' can be computed much faster than computing the whole query plan of Q', we reduce the query optimization cost for Q', by using P'' as the query plan for Q' rather than computing optimal query plan for Q' from scratch. However, in order to apply this proposition, we need to be able to do the following:

1. Decompose Q and Q' into top-level queries (Q_{top} and Q'_{top}) and subqueries ($\{Q_i\}$ and $\{Q'_i\}$).
2. Check whether there exists one-to-one correspondence between the subqueries $\{Q_i\}$ and $\{Q'_i\}$ such that each pair of subqueries have the same schema, and check whether Q_{top} and Q'_{top} are identical.
3. Decompose P to establish a one-to-one correspondence between subplans of P with subqueries of Q, such that each subplan P_i implements exactly the corresponding subquery Q_i.
4. Compute subplans $\{P'_i\}$ for subqueries $\{Q'_i\}$.
5. Check whether each subplan P'_i is a compatible substitute for another subplan P_j with respect to P_{top}.

With the above prerequisites, we are able to show the POM plan reuse framework as in Algorithm 3.

The query plan produced by procedure $Reuse\_Plan$ in Algorithm 3 is called a *stitched plan*. There are several issues with this plan caching and reusing algorithm. First, the decompositions required in line 1 and line 3 of the procedure $Cache\_Plan$ and in line 2 of the procedure $Reuse\_Plan$ cannot be arbitrary decompositions. We wish to decompose queries and query plans such that the conditions at line 3 and line 5 of the procedure $Reuse\_Plan$ can be satisfied. Second, we can not guarantee the optimality of the replaced query plan for Q'. To resolve these issues, we need to look at the details of query planning for an actual query optimizer.

In the next section we will use the optimizer of PostgreSQL, an open-source database system, to show how we are able to accomplish the five prerequisites for Algorithm 3. Also, we will show how to extend Algorithm 3 to decompose queries and query plans to accommodate plan stitching, and to ensure the correctness and the optimality for the stitched query plan. We will also extend Algorithm 3 for a more general optimizer model in Section 4.3.2.

Algorithm 3 POM Plan Reuse Framework

CACHE_PLAN(Q) // Q is a customized query

1: Decompose Q into Q_{top} and subqueries $Q_1, Q_2, \ldots, Q_m$
2: Compute optimal query plan P for Q using existing optimizer;
3: Decompose P into P_{top} and subplans $P_1, P_2, \ldots, P_n$
4: Cache $Q_{top}, Q_1, Q_2, \ldots, Q_m$ and $P_{top}, P_1, P_2, \ldots, P_n$

REUSE_PLAN(Q') // Q' is another customized query

1: Check whether Q' and Q are from the same base query
(i.e., whether they only differ in selection predications)
2: Decompose Q' into Q'_{top} and $Q'_1, Q'_2, \ldots, Q'_{m'}$
3: **if** Q_{top} and Q'_{top} are not identical
4: **then** Return $null$;
5: **if** $\{Q'_1, Q'_2, \ldots, Q'_{m'}\}$ and $\{Q_1, Q_2, \ldots, Q_m\}$
have one-to-one mapping such that each subquery pair has the same schema
6: **then** Build subplans $\{P'_i\}$ for $\{Q'_1, Q'_2, \ldots, Q'_{m'}\}$
7: **if** for each cached subplan P_i there is a compatible substitute
from $\{P'_i\}$ with respect to P_{top}
8: **then** Replace each subplan in P with its compatible substitute;
9: **return** the replaced query plan;
10: **return** $null$;

4.3.1 Case Study: PostgreSQL

We first briefly describe query processing in PostgreSQL. A query is first parsed and represented in a block-based structure, with subqueries as blocks linked from the main query block. Each block represents a join over several relations, and the block includes all join predicates and select predicates in its join. Each block may be wrapped with top-level operators such as $aggregation$, $distinct$, and $order\text{-}by$. The optimizer then rewrites the query by flattening the block structure, normalizing and pushing the select operators to each relation. The left side of Figure 4.4 illustrates the internal representation of rewritten query of Q_1 from Example 1. We can see the redundant join predicate is removed after query rewriting.

The optimizer then generates a query plan based on the rewritten query in a bottom-up fashion. Access paths on the leaf relations are built first. These implement *exactly* the same logic of the single-table subqueries in the rewritten query. For example, in Figure 4.4, the index-scan on $R.a$ implements the left subquery ($\sigma_{R.a=5}(R)$) while the table-scan on S implements the right subquery ($\sigma_{S.b=5}(S)$).

After the access paths are built, the optimizer enumerates join trees on top of these access paths using dynamic programming, forming optimal joins over all pairs of relations, then forming optimal joins over every triple relations, and so on until the complete join plan of all the relations in the query is built. Finally the join plan is wrapped by the top-level operator of the query block to form a complete query plan, which is the optimal query plan.

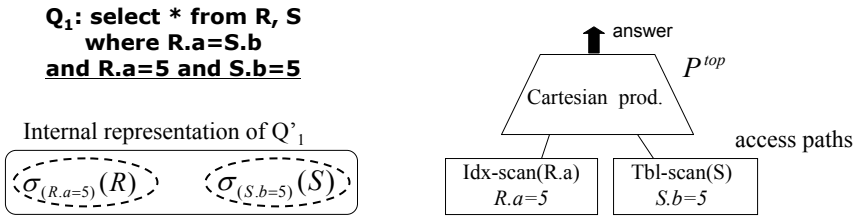


Figure 4.4: Subplans correspond to subqueries in PostgreSQL

Physical Compatibility Requirements

Recall that Proposition 4.3.1 requires each subplan P_i' be a *compatible substitute* to some subplan P_i with respect to the top-level query plan P_{top} of P. To be a compatible substitute for P_i, P_i' must be physically compatible with P_{top} at the point at which P_i attaches to P_{top}. To see why we need this, consider the example illustrated in Figure 4.5. The left side shows the query plan for Q_2 in Example 1, which is a hash-join over two table-scans. The right side shows the query plan for Q_3, which is a merge-join on an index-scan on $R.a$ and an table-scan on S sorted by $S.b$. Suppose we have cached the query plan of Q_3, and wish to replace its access paths with the access path of Q_2 to create a query plan for Q_2. The conditions for subplan-subquery

correspondence are established already. However, we cannot replace the index-scan on $R.a$ in the query plan of Q_3 with the table-scan on R from the query plan of Q_2, since the table-scan on R does not generate the *interesting order* required by the merge-join operator.

In PostgreSQL, there are two physical properties that we need to consider when stitching subplans to top-level partial plan.

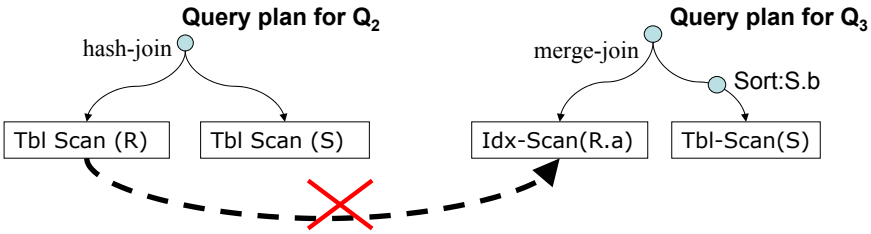


Figure 4.5: Order requirement

a The *interesting order* property of a subplan which sends tuples to its parent operator in a specific order.

b The *index-probe-able* property of a subplan corresponds to the right child of an indexed-nested-loop join operator. This property requires that the subplan get a specific tuple from an index given the tuple's identity.

Other database systems may have different sets of physical compatibility requirements. The complete list of physical compatibility requirements between query plan operators depends on the specific optimizer, and we will discuss the more general cases in the next section.

Physical Property Requirement for Subplans

Once we identify the physical properties for the physical compatibility requirements, we can check whether a subplan is physically compatible with a top-level partial plan. To do this, we need to cache the set of requirements on physical properties that the new subplans need to satisfy to be physically compatible to the already cached top-level partial plan. We call these requirements *physical property requirements* for the cached partial plan. Hence we need to store the upper-level query plan, and the physical property requirements on its subplans, in the POM cache.

Applying Algorithm 3 for PostgreSQL

To apply Algorithm 3 for PostgreSQL, we need to satisfy the five prerequisites as outlined in Section 4.3.

1. *How to decompose Q and Q' into top-level queries (Q_{top} and Q'_{top}) and subqueries ($\{Q_i\}$ and $\{Q'_i\}$)*:

   We decompose the queries such that each subquery involves a single relation.

2. *How to check whether there exists one-to-one correspondence between the subqueries $\{Q_i\}$ and $\{Q'_i\}$ such that each pair of subqueries have the same schema, and check whether Q_{top} and Q'_{top} are identical:*

   By decomposing two queries Q and Q' as in the previous step, we are able to check whether the two queries have subqueries working on the same relation. If Q_{top} and Q'_{top} are identical, and the two single-table subqueries are from the same relation, then these two subqueries have the same schema. This is because these two subqueries must contribute the same number of fields to their upper-level partial query, and their projection operators must be the same. We can then check whether there exists a one-to-one mapping between the subqueries of Q and Q' having the same schema. We extract Q_{top} and Q'_{top} from the internal representation of the rewritten queries (denoted as Q_{rw} and Q'_{rw} respectively), by trimming all selection and projection operators on these relations from the query block. The trimmed representation corresponds to the top-level partial queries of the rewritten query. To check whether the top-level queries Q_{top} and Q'_{top} are identical, we simply compare Q_{top} and Q'_{top}.

3. *How to decompose P to establish a one-to-one correspondence between subplans of P with subqueries of Q, such that each subplan P_i implements exactly the corresponding subquery Q_i:*

   Since each access path exactly implements each single relation subquery, the one-to-one correspondence between the access paths (subplans) and the subqueries can be established.

4. *How to compute subplans $\{P'_i\}$ for subqueries $\{Q'_i\}$:*

   PostgreSQL creates query plans in a bottom-up fashion. Thus we can invoke the optimizer to build the subplans (access paths) for the subqueries.

5. *How to check whether each subplan P'_i is a compatible substitute for another subplan P_j with respect to P_j's upper level partial plan P_{top}:*

   P'_i is a compatible substitute for P_j with respect to P_{top} when the following two conditions hold:

   (a) P'_i's returned tuples are sorted in an order that is consistent with or dominates the sort order required by P_{top} from P_j.

(b) if P_j is the right child of an indexed-nested-loop join operator, then P_i' is the right child of an indexed-nested-loop based on the same index (or a composite index that dominates the same index).

With the above prerequisites, we are able to apply Algorithm 3 for PostgreSQL. According to Proposition 4.3.1, Algorithm 3 will generate correct query plan for Q'.

POM allows us to compute query plans by computing access path (subplans) for each subquery without computing the whole query plan. Therefore, POM saves us from computing the whole upper-level join plan for the queries. Since upper-level join plan generation is the most expensive part of query optimization [48], the savings from POM can be significant. We will illustrate this by experiment in Section 4.7.3.

4.3.2 More General Cases

The above approach in PostgreSQL can be extended for other optimizers. For example, DB2 internally represents queries using the Query Graph Model (QGM) [74]. Like PostgreSQL, DB2 builds *access plans* that correspond to the subqueries of leaf relations in QGM. Thus the one-to-one correspondence from the subplans to the subqueries can be established. We can "trim" a QGM by removing all of its base table boxes and the edges linked to these boxes. We also remove all of the loop links, which correspond to all select predicates, in the remaining boxes. The trimmed QGM represents a top-level query and can be serialized into string for checking top-level query identicalness. Moreover, this optimizer allows us to compute the subplans of each leaf subquery without computing the whole query plan.

Different database systems have different query plan operators and may have different physically compatibility requirements for its query plan operators. For example, a Bloomjoin operator [63] requires one of its children to hash its tuples into a bit-vector while the other child reduces its input using the bit-vector and the same hash functions. A Rankjoin operator [47] requires its children to have certain properties in terms of ordering. Given a specific database system, we need to carefully examine its query plan operators to add other property requirements to the picture.

4.4 POM Caching for Access Control Customized Queries

In this section we show how to build the intelligent plan caching mechanism for access control customized queries. This caching mechanism is based on the foundation of plan reuse presented in Section 4.3. For the rest of the discussion we continue to use PostgreSQL to demonstrate our plan caching mechanism. We will explain how this architecture can be modified for other optimizers in Section 4.8.

4.4.1 Access Control Customization Formalism

We envision a query issued by many users $\mathcal{U}$ where each user invokes a customization of the query. We first formalize the representation of access control customization through the follow definitions:

Definition 4.4.1. *A **local predicate** on relation R is a predicate that involves only attributes of R and constants.*

Definition 4.4.2. *A **local predicate formula** on relation R is one or several local predicates connected by AND, OR and NOT.*

Suppose the query is defined on relations $R_1, R_2, \ldots, R_k$. We use $Q(R_1, R_2, \ldots, R_k)$ to denoted this query. For each user $u \in \mathcal{U}$ and each relation R_i, there is a user-customization view of R_i for u, which we denote by V_i^u. Each customization view V_i^u is defined over R_i in the form of $\sigma_{C_i^u} R_i$, where C_i^u is a *local predicate formula* on R_i. When user u submits a query $Q(R_1, R_2, \ldots, R_k)$, the database system customizes Q for u into a user-specific version (denoted by Q^u), by replacing references to each R_i in Q with references to V_i^u:

$$Q^u \equiv Q(V_1^u, V_2^u, \ldots, V_k^u)$$

In this chapter we model users' access control customization as local predicate formulae on relations. This is a natural extension to the Oracle Virtual Private Database model, which customizes user's query by appending at most one local predicate to each relation referred to in the query [70]. In Section 4.8 we will show that our approach can be extended for more general access control customizations (e.g., customization through join predicates).

4.4.2 Architecture

Before illustrating the POM architecture, we need to define some terminology:

Definition 4.4.3. *A **leaf query** is a subquery whose operators only refer to one relation. The query with its leaf queries removed is called a* partial query.

Definition 4.4.4. *A **partial plan** ($\hat{P}$) is a query plan with all its access paths removed. Each point of removal is labeled with the name of the relation accessed by the removed access path.*

Definition 4.4.5. ***Plan stitching** is the process of connecting a set of access paths to the leaves of a partial plan, such that each access path matches to the relation labeled at the leaf of the partial plan. A stitched query plan has every leaf connected to an access path.*

Since access control customizations are in the form of local predicate formulae, it is anticipated that the queries' query plans compute different answers just because of their different access paths. In another word, their upper level partial plans implements the same logic. This suggests we apply Proposition 4.3.1 to reuse top-level partial query plans for customized queries and save query optimization effort. More specifically, we cache partial plans for customized queries, and stitch them with the access paths computed for new customized queries to create query plans for the new customized queries. This idea leads to the architecture as illustrated in Figure 4.6. It has two components: the plan-cache component and the plan matching-and-stitching component.

Plan-Cache: For each query Q^u that is optimized from scratch, POM stores the top-level query Q_{top} from Q^u, the partial plan $\hat{P}$ for Q^u, and some meta-data, in its cache. The meta-data includes the schema information of the single-table subqueries, and the *physical property requirements* from the partial plan on the removed access paths, and some other data whose purposes will be described in Section 4.5.

Plan Matching and Stitching Component: After a new customized query Q'^u is submitted, POM first decomposes this query into upper-level query and single-table subqueries. Then POM finds all cache entries whose top-level query is identical to the top-level query of Q'^u, and whose subqueries have the same schema as this new query's subqueries. These cache entries corresponds to other customized queries from the same base query. Then POM computes access paths for Q'^u, and for each of these cache entries, POM checks whether Q'^u has a set of access paths that can be stitched to the partial plan $\hat{P}$ of that cache entry, by looking at the *physical property requirements* of the partial plan. If there are such access paths, POM stitches them to that partial plan to create a query plan that correctly computes Q'^u. After doing this, POM also checks whether the stitched query plan has the potential to be an optimal query plan. We will describe this step in detail in Section 4.5. There can be multiple stitched query plans from multiple cached partial plans that match the new customized query. In the last step, POM will pick the stitched query plan with the lowest cost.

Using this architecture, POM significantly reduces query optimization time compared to optimizing each plan from scratch. This is because POM avoids enumerating and evaluating all possible join trees for each new customized query once there is a partial plan matched, and join tree enumeration and generation is considered the dominant expense of query optimization[48]. Moreover, each cached partial plan can be matched for a *group* of new queries, if the new queries have the same partial rewritten query and access paths that are physically compatible. Thus the cache hit ratio can be much higher than syntax level query plan caching. Finally, the cache size can be adjusted based on resource availability and does not have to be as large as the *NS* approach described in Section 4.1.

We will show in Section 4.5 that a carefully designed partial plan matching algorithm will

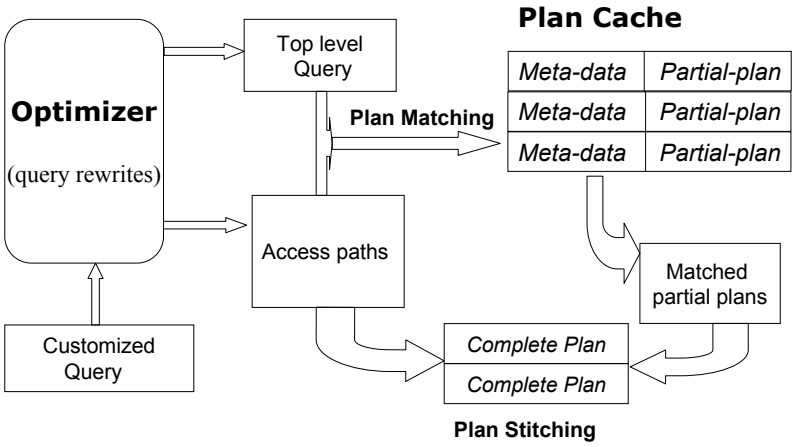


Figure 4.6: POM architecture

guarantee that the generated query plans have performance close to the optimal query plans for the new queries. Therefore, POM fulfills all the design goals set in Section 4.1.1.

Like other plan caching mechanisms, we require that the data be relatively static in order for POM to work. If the system undergoes significant changes (adding or removing indexes, changing data distribution, etc), we need to discard the cached partial plans and restart from scratch.

For ease of presentation, in the rest of the discussion we use Q to denote a query that is already optimized, and use Q' to denote a new query for which we wish to reuse the query plan for Q. We use P to denote the optimal query plan for Q, and we use P' to denote the stitched query plan for Q'.

4.5 Plan Matching for Optimality

Section 4.3 presents the foundation of query plan reuse by plan stitching. Section 4.4 applies this theory to access controlled customized queries. In this section, we focus on the performance of stitched query plans generated by PostgreSQL. We assume the stitched query plans are correct, i.e., the top-level query of Q is identical to that of Q', and there exists a set of access paths of Q' that are compatible substitutes for the original access paths with respect to $\hat{P}$. These two conditions are prerequisites for the correctness of the stitched plan, as described in Section 4.3.

Our approach consists of two steps. First, we try to establish an upper-bound on the cost of the stitched query plan using $\hat{P}$. Then we show that under certain conditions this upper-bound is also the lower-bound. Therefore, the stitched query plan is an optimal query plan for Q'.

In the remainder of Section 4.5, we consider any query optimizer that has the following properties:

1. The cost of each subplan depends only on the costs and sizes of its child subplans, and the operator at the root of the subplan.
2. Query plans are computed in a cost-based, bottom-up fashion, by creating the access paths first, and then the join trees. There are no randomized heuristics involved.
3. Access paths implement the single-table subqueries, while the top-level query plan (with access paths removed) corresponds to the upper-level query (with single-table subqueries removed).

For example, the PostgreSQL optimizer has such properties.

4.5.1 An Upper Bound on Stitched Query Plan Cost

Before introducing the upper bound on stitched query plan's cost, we need the following definition:

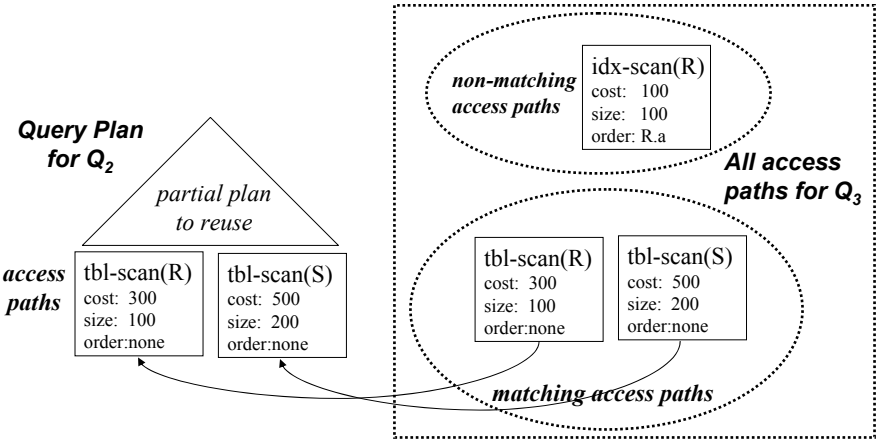
Definition 4.5.1. *If a subplans P_1 and P_2 have the same schema, the same physical properties, and the same cost, and P_1 and P_2 return the same number of tuples, then P_1 and P_2 are* matching subplans*. If both P_1 and P_2 are access paths, then P_1 and P_2 are* matching access paths *(Figure 4.7).*

For any optimizer that has properties listed at the beginning of Section 4.5, we have the following performance guarantee on stitched query plans:

Theorem 4. *If the access paths in P of query Q are replaced with their matching access paths for another query Q', then the new query plan ($\hat{P}$ stitched with these matching access paths) not only correctly computes Q', but also has the same cost as P.*

Proof. Since we assumed the partial plan $\hat{P}$ of P implements the top level of Q', the top-level queries of rewritten Q and Q' must be identical. Therefore, the access paths of Q' that are physically compatible to $\hat{P}$ of P can be stitched with $\hat{P}$ to create correct query plan for Q', as already described in Section 4.2. Since the optimizer computes the cost of query plan based only on the sizes and costs of subplans, and the generated query plan has exactly the same sizes and costs with P for each element, the theorem holds. □

Theorem 4 shows when we are able to generate a query plan with the same cost as P. When such a plan can be generated, its cost can serve as an upper bound on the cost of an optimal query plan for Q'. In the next section we will show this upper bound is also the lower bound of the optimal query plan under certain scenario, hence the optimality of the stitched query plan.



Q2: select * from R,S where R.a=S.b and R.c<5

Q3: select * from R,S where R.a=S.b and R.a<5

Assuming ***(R.a<5)*** and ***(R.c<5)*** select the same number of tuples

Figure 4.7: Matching access paths to partial plans

4.5.2 Ensuring Optimality of Stitched Query Plans

Now we show how to ensure the optimality of the stitched query plan P'. There are two possible scenarios under which the stitched query plan P' can be a sub-optimal plan for Q':

1. The optimal query plan P'' has a different set of access paths than P'.
2. The optimal query plan P'' has the same set of access paths but a different partial plan than P'.

We first show that the second scenario will not happen in PostgreSQL. In fact, for any optimizer that has the first property listed at the beginning of Section 4.5, if P' and the optimal query plan P'' for Q' have the same set of access paths, then their partial plans must be identical.

In fact, if every access path in P'' is a matching access path of the corresponding access path in P', then P' and P'' must have identical partial plans, by the same reasoning.

Therefore, the only scenario in which the stitched query plan P' may not be optimal is a scenario in which at least one access path failed to match when the stitched plan P' was computed. If this does not happen, we will be able to guarantee the optimality of the stitched query plan P'.

Before continuing, we first introduce the notion of a subplan P_i being a *universally* compatible substitute for another subplan P_j: P_i is compatible substitute for P_j with respect to *any* upper-level query plan. For example, in PostgreSQL, if an access path P_1 has a stronger or

equivalent order than access path P_2, and P_1 and P_2 have the same schema, then P_1 is a universally compatible substitute for P_2. Please note that a pair of matching subplans are universally compatible substitutes for each other.

Now, we focus on the case in which the optimal query plan P''' has a different set of access paths than P'. We have the following Lemma:

Lemma 1. *If, for every non-matching access path P_i'' of Q', there exists an access path P_i''' for Q such that*

1. *P_i''' is a universally compatible substitute for P_i'',*
2. *P_i''' has a lower cost than P_i'',*

then P_i'' will not be in the optimal query plan for Q'.

Proof. Assume the non-matching access path P_i'' appears in the optimal query plan P'' for Q'. We prove that this cannot happen. We can replace the non-matching access path P_i'' in P'' with access path P_i''', since this P_i''' is a universally compatible substitute for P_i''. We then replace each of the remaining matching or non-matching access paths in P'' with their compatible substitute access paths for Q. We are able to do this since the assumption of this lemma is that each non-matching access path has a universally compatible substitute, and each matching access path of Q' has a matched access path for Q that can replace it.

The resulting query plan (denoted by P''') computes Q. This is because:

1. the top-level queries of Q and Q' are identical;
2. P_i''' is a universally compatible substitute of P_i'', so P_i''' is physically compatible with the upper-level query plan of P'';
3. all other replacement access paths are compatible substitute for the access paths of P'' with respect to the upper-level query plan of P''.
4. the upper-level query plan P'_{top} implements the upper query Q'_{top}, which is equivalent to Q_{top}. The access paths of P_i''' and P_j'' implement the single-table subqueries of Q.

Now since P_i''' has a lower cost than P_i'', P''' is cheaper than P'', which is cheaper than P'. We already know that the stitched plan P' has the same cost as P from Theorem 4. Therefore, P''' should have a cost lower than P. This contradicts our assumption that P is the optimal query plan for Q.

□

Figure 4.8 illustrates Lemma 1, where we have Q optimized and its query plan P cached. We have a new query Q', whose only non-matching access path P_1'' has a compatible substitute P_1''' for Q. P_1''' also has a lower cost than P_1''. Assume Q' has an optimal query plan P'' with P_1'' as its access path, as shown on the lower right side of the figure. We can replace P_1'' with P_1''', and each of the remaining access paths (P_2'' and P_3'') with their compatible substitutes (P_2 and P_3). We know that we can do this, because if P_2'' (or P_3'') is a non-matching access path, then from the assumption of the lemma there must be an access path of Q that is a universally compatible substitute for P_2'' (or P_3'') with a lower cost; if P_2'' (P_3'') is a matching access path, then there must be a matched access path for Q that is already a universally compatible substitute. Now we have a query plan P''' as shown on the lower left side of Figure 4.8. This plan computes Q, even if P_{top} is not the same as P'_{top}.

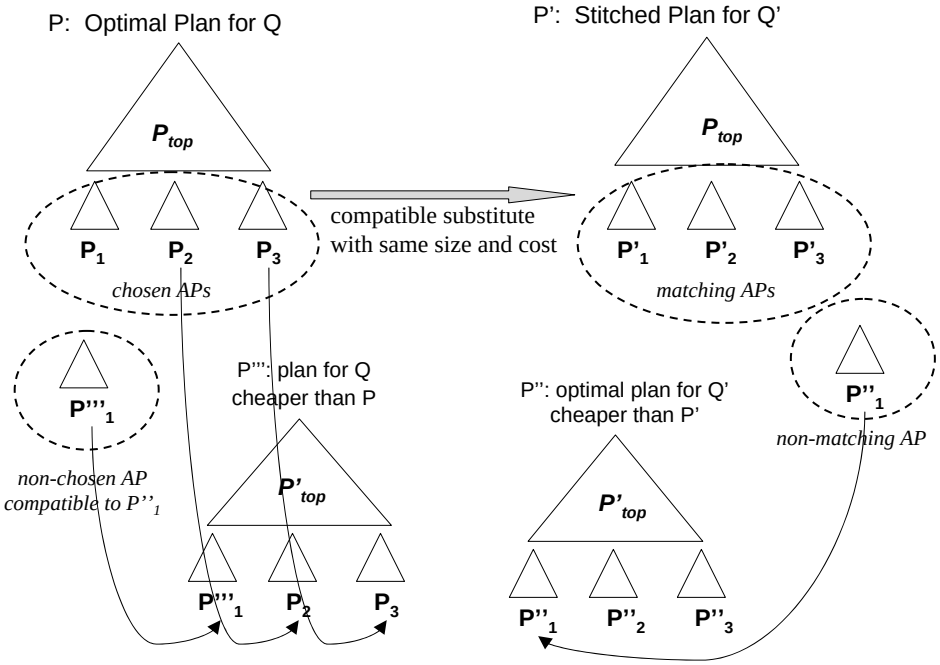


Figure 4.8: Optimality guarantee

Theorem 5. *If each non-matching access path of Q' has an access path for Q that is a compatible substitute with a lower cost, then the stitched query plan from Theorem 4 is an optimal plan for Q'.*

Proof. According to Lemma 1, none of these non-matching access paths should appear in the optimal query plan for Q'. Since there are only two possible scenarios under which the stitched

plan is not an optimal query plan for Q', and both scenarios are not possible, then the stitched plan is an optimal plan for Q'. □

On the other hand, if one of the non-matching access paths for Q' does not have an equivalent compatible substitute from any of the access paths of Q, or if none of its equivalent compatible substitutes for Q has a lower cost, then we cannot guarantee the optimality of the stitched query plan from the matching access paths. For example, suppose we have already cached Q_2's query plan, and try to reuse it for Q_3, as in Figure 4.9. Q_3 has all of its access paths computed as in Figure 4.7, and its index-scan on $R.a$ is not matched to the access paths in the optimal query plan for Q_2, since its *cost* and *order* properties do not match with the corresponding access path on table R. We find that there is one compatible substitute access path for Q_2, which is the index-scan access path on $R.a$. However, this index-scan access path has a tremendous cost at 20000, which is much higher than the cost of the non-matching index-scan access path for Q_3. Therefore, we cannot guarantee the optimality of the stitched query plan from the previous section. In fact, the optimal query plan for Q_3 is a merge-join based plan (the lower right of Figure 4.9).

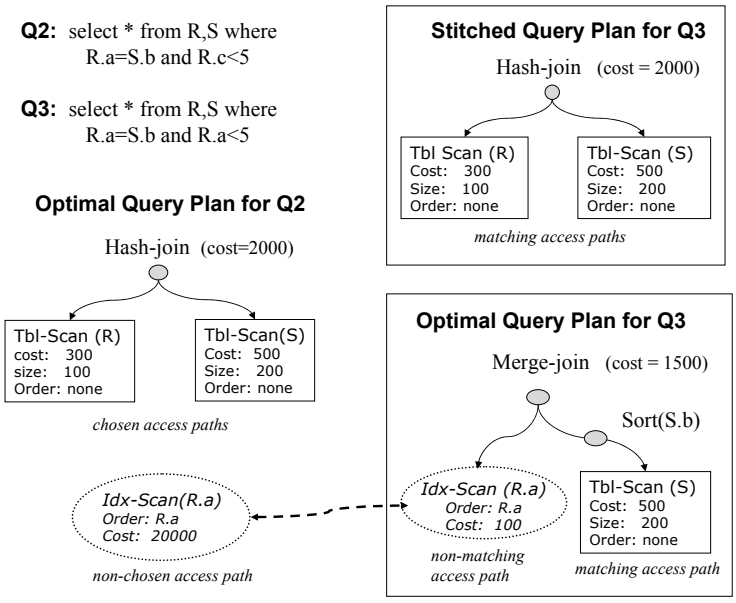


Figure 4.9: Non-matching access paths change optimality

4.5.3 Implementing POM Plan Caching

Theorem 4 tells us how to compute a stitched query plan with the same cost as a previous query plan. Theorem 5 allows us to check whether this stitched query plan is indeed optimal. To apply these theorems in POM, we need to record the sizes, costs, and ordering properties of both the access paths for Q that appear in its optimal query plan P (i.e., the chosen access paths), and the access paths for Q that do not appear in P (i.e., the non-chosen access paths). Using the example in Figure 4.9 again, when the optimizer computes query plan for Q_2, we store the *cost*, *size*, *order* and index information of all three possible access paths for Q_2: the table-scan on R, the table-scan on S, and the index-scan on $R.a$. All this information, together with the top-level of the rewritten query of Q_2, are the meta-data associated with the partial plan $\hat{P}$ of P.

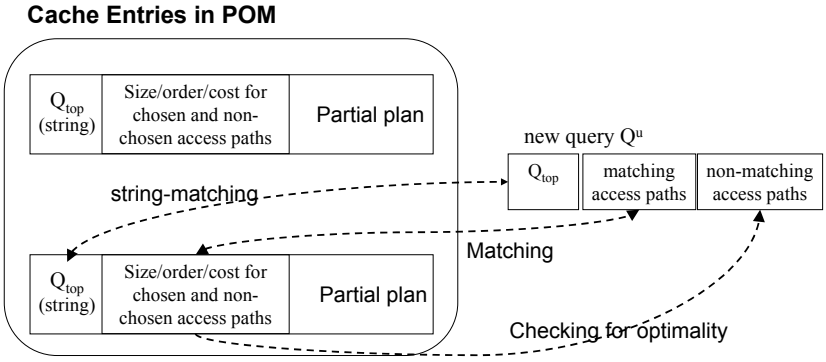


Figure 4.10: POM cache entry matches to a query

When a new customized query Q_3 is issued and we wish to reuse Q_2's partial plan, we do so using the following steps:

1. We first use Q_2's top-level query and the information of its chosen access paths to check whether Q_2's partial plan can be used to correctly compute the new query in a manner described in Theorem 4.3.1.

2. We then invoke the optimizer to compute all possible access paths for Q_3.

3. We look at the information of Q_2's chosen access paths, and try to find a set of matching access paths of Q_3 for the chosen access paths of Q_2. If such access paths exist, we connect these access paths to Q_2's partial plan and get the stitched plan. Otherwise, we will have to compute the complete query plan for Q_3 and cache the partial plan and meta-data as was done for Q_2.

4. We then check whether the stitched plan is optimal. We do this by looking at the non-matching access paths of Q_3 and the information of all of Q_2's access path, as described in Theorem 5. If each of the non-matching access path has a universally compatible substitute with a lower cost, we know that the stitched plan is the optimal plan. In this case, we say the non-matching access path is *dominated* by that POM cache entry. If one of the non-matching access path does not have a universally compatible substitute access path with a lower cost, then this POM cache entry does not dominate this non-matching access path, and we will have to compute the query plan for Q_3 from scratch and cache the partial plan and meta-data as was done for Q_2.

The detailed procedure for reusing previously cached plans for customized queries under PostgreSQL is illustrated in Algorithm 4. Note that the one-to-one mapping between subqueries of Q and subqueries of Q is implied by compatible substitutes between access paths. The purpose of $\mathcal{PLAN}_{match}$ and the procedure $Update\_Cache$ will be discussed in the next section.

4.5.4 Cache Maintenance

Whenever a customized query Q^u fails to find any matching partial plan from the cache, POM needs to optimize the new query from scratch, and update the cache with a new entry consisting of the top-level of the rewritten query of Q^u, the partial query plan, together with the information on the chosen and non-chosen access paths.

For example, suppose POM first computes query plan for Q_2 from Example 1 and caches its partial plan. Now suppose another customized query Q_4 is submitted as follows:

Q_4: *select * from R, S where R.a=S.b and* ***(R.a>1000)***

POM first computes the access paths of Q_4, which include an index-scan access path on $R.a$. This access path has cost as 200, which is much lower than the corresponding access path of Q_2. We also assume the predicate (R.a>1000) selects the same number of tuples as the predicate (R.c<5) in Q_2. We can see that POM does not match the partial plan of Q_2 for Q_4 since there does not exist a non-chosen access path of Q_2 that is a universally compatible substitute with lower cost to the index-scan access path of Q_4. Therefore, we cannot apply Theorem 5 to ensure the optimality of the stitched plan. We will have to computes the query plan for Q_4 from scratch. Suppose the optimal query plan for Q_4 is also a hash-join over two table scans. POM caches the partial query plan of this query plan. Now the POM cache consists of two entries, as illustrated in Figure 4.11 (the upper one comes from Q_2 and the lower one from Q_4).

A close look at these two entries shows that they have an identical top-level query Q_{top}, identical information on the chosen access paths, and an identical partial plan. The only difference is that the first entry (corresponds to Q_2) has a higher cost for the non-chosen access path. In this case, we are certain that this entry is redundant. The reason is that any query that is matched by

Algorithm 4 Detailed POM Procedure

PLAN_REUSE(Q^u)

1: $\mathcal{PLAN}_{match} = \emptyset$ // plans match on chosen access paths
2: $\mathcal{PLAN}_{stitch} = \emptyset$ //stitched query plans
3: P_{opt} = NULL; // to store the optimal plan for Q^u

4: Compute Q_{top} for Q^u;
5: Build access paths $\mathcal{AP}$ for Q^u;

6: Find POM cache entries $\mathcal{E}$ with same Q_{top} and same base relations;
7: **for** each entry $E \in \mathcal{E}$
8: **do if** ($\mathcal{AP}$ has matching access paths for E)
9: **then** Add E to $\mathcal{PLAN}_{match}$;
10: Record the matching access paths;
11: **if** (every non-matching access path of Q^u is dominated by E)
12: **then** P = $\hat{P}$ stitched with matching access paths;
13: Add P to $\mathcal{PLAN}_{stitch}$;
14: **if** ($\mathcal{PLAN}_{stitch} \neq \emptyset$)
15: **then** P_{opt}= the cheapest plan in $\mathcal{PLAN}_{stitch}$;
16: **else** Compute query plan for Q^u, assign to P_{opt};
17: $Update\_Cache(P_{opt}, Q_{top}, \mathcal{AP}, \mathcal{PLAN}_{match})$;

UPDATE_CACHE($P_{opt}, Q_{top}, \mathcal{AP}, \mathcal{PLAN}_{match}$)

1: **if** ($\mathcal{PLAN}_{match} \neq \emptyset$)
2: **then for** each $E \in \mathcal{PLAN}_{match}$
3: **do if** ($\hat{P}$ in E is same as $\hat{P}$ of P_{opt})
4: **then** Update costs of non-matching access paths in E
5: **else**
6: Create new POM entry E_{new};
7: Add Q_{top} to E_{new};
8: Add $\hat{P}$ of P_{opt} to E_{new};
9: Add properties of access paths of P_{opt} to E_{new};
10: Add properties of all rest access paths in $\mathcal{AP}$ to E_{new};
11: Add entry E_{new} to POM cache;

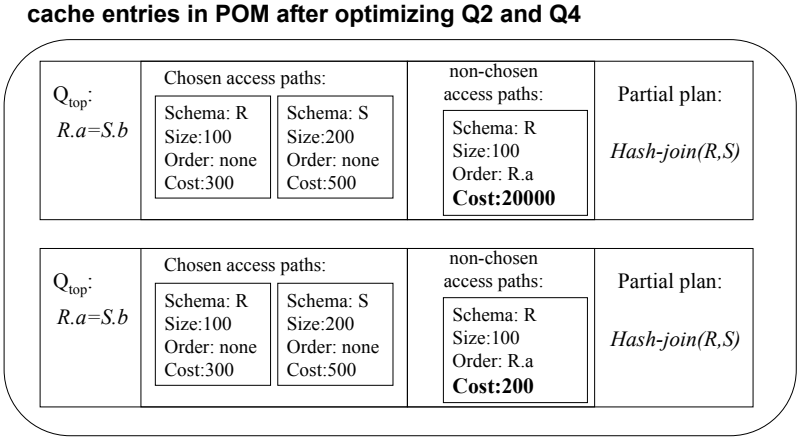


Figure 4.11: Two entries cached for Q_2 and Q_4

the first entry can be matched by the lower entry with a lower cost for the stitched plan, but not vice versa. Therefore we can remove the first entry.

More generally, if we find two or more entries with the same top-level query, the same partial plan and the same information on chosen access paths, we are able to merge them by taking the *minimum* costs for all their non-chosen access paths. This merge step is critical for the effectiveness of POM. First we reduce the number of redundant entries to make the cache more effective. Second, the new entry has the minimum costs on its non-chosen access paths among all of the merged entries. The lower the costs of non-chosen access paths, the less likely the "optimality check" step in Theorem 5 fails, and the more cache hits from POM.

The detailed POM cache update procedures is formalized as procedure $Update\_Cache$ in Algorithm 4. Here $\mathcal{PLAN}_{match}$ from procedure $Plan\_Reuse$ represents all the POM cache entries that might be merged with the entry of the new customized query.

4.5.5 Practical Concerns

Checking each entry in the POM cache for a customized query can be time-consuming. Therefore, we sort and cluster these entries according to the *size* property of the matching access paths. In this way we are able to quickly discard a large number of entries that do not match the incoming queries.

Although Theorem 4 and Theorem 5 guarantee the optimality of the stitched plan if all condition in the theorems hold, it is very rare that access paths from two queries have the same *size* and *cost* properties. Therefore, most of the time POM does not have a cache hit. A remedy for this is to sacrifice optimality for higher cache hit ratio: instead of requiring the access paths of a new

query agree exactly on the two properties for the chosen access paths, we relax the requirement so that the two properties are matched for access paths if they are within a relatively close range. The side-effect of this trade-off is that the stitched query plan might be sub-optimal. However, our experiment shows that the stitched query plans are still mostly optimal at a moderately large approximation range for these properties. Another issue is that once we bring in approximation, there can be multiple partial plans that can be used for creating a query plan for the new query. In this case, we simply pick the stitched query plan with the lowest cost.

Aside from approximation of the *cost* and *size* properties, POM may also produce sub-optimal query plans due to our assumptions on the optimizer model. We assume that the optimizer estimates join sizes and join costs based on the join predicates, the inputs' sizes and the inputs' costs. Other database optimizers may have factors that are not included in our model. However, those *ad hoc* factors can be treated as "properties" in a similar fashion and added to our model to improve POM accuracy.

4.6 Comparison to Related Work

There is a history of work on efficient query processing for large numbers of users with different user preferences [8, 31, 95]. However, these works do not address cost-effective query planning. The approaches in [20] and [83] address multiple query planning with common sub-expressions, but their emphasis is on creating a single query plan that makes the best use of intermediate results that correspond to common subexpressions from similar queries.

The closest related work is parametric query optimization (PQO) [49]. The idea is that there may exist certain query parameters that can not be determined at query optimization time, yet these parameters affect the choice of optimal query plans. Examples of these parameters include buffer pool size and data distribution at query time. The optimizer thus treats these parameters as variables, and creates many query plans, each being an optimal query plan when these variables fall in a value range. At runtime the system chooses the optimal query plan from these pre-compiled query plans according to the actual value of these parameters. For example, a query Q involves joining two tables. The system does not know how much memory will be assigned to this query at runtime. If the memory size is big enough to fit a hashtable of a smaller table, a hash-join will be more efficient, while a merge-join will be faster under lower memory size. Therefore, the optimizer may create two query plans, one based on hash-join for larger memory size, one based merge-join for smaller memory size. At runtime, the actual memory allocation between concurrent queries is known and the database system is able to choose from the two query plans according to the actual memory allocated. In this way, PQO provides an optimal query plan with negligible query planning cost at runtime.

PQO itself does not provide a viable solution to our problem since it is for planning queries

with varying parameter values, not for queries with unknown structures and values from customizations. In other words, PQO assumes a set of *complete* query plans to choose from at runtime, each of which computes exactly the answer for the query submitted. This can not be used to solve our problem since we do not know what the customizations are when computing query plans. However, we can enrich PQO by modeling the selectivity of the user-customizations as a runtime parameter, and compute several query plans each being optimal at a range of selectivities. At runtime, we first compute the selectivity of the actual user-customizations, and choose the query plan that has matching selectivities. The chosen query plan can be augmented with the user-customization at its plan leaves to correctly answer the customized query. However, this enriched PQO approach does not generate good query plans, as we shall see from the experiments. Moreover, the number of query plans to compute offline is exponential in the number of parameters, although many configuration of the parameters will not be used at all by the users.

Another closely related work is called *query PLan SelecTIon based on Clustering* (PLASTIC)[39]. The rationale of this approach is to cluster "similar queries"' and assign to each cluster one *query plan template*. Any query in this cluster is able to get its optimal query plan by replacing the table name, attribute name, or constants in the template with the table name, attribute name, or constants from the query. Queries in one cluster must have similar syntax and schema, similar data distribution, and similar auxiliary data structures (e.g., indexes and materialized views) for the tables in the query. Later when a new query is issued, a *classifier* is used to determine which cluster this new query belongs to. In this way PLASTIC is able to reduce query optimization cost at runtime.

However, we argue that the PLASTIC approach may wrongly match a query plan for a query, or fail to detect a good matching query plan due to the over-simplified matching algorithm of its classifier. Consider again the following three customized queries, where the customizations are in bold characters.

Q_1 *select * from R, S where R.a=S.b and* ***R.a=5 and S.b=5***

Q_2 *select * from R, S where R.a=S.b and* ***R.a=5***

Q_3 *select * from R, S where R.a=S.b and* ***R.a=4 and S.b=6***

PLASTIC will assign Q_1 and Q_3 to the same cluster, leaving Q_2 to a different cluster. Now suppose a smart optimizer detects the join predicate $R.a = S.b$ in Q_1 to be redundant, and generates optimal query plan for Q_1 as a cartesian product over the inputs from two tables R and S. Since Q_3 is matched to the same cluster, PLASTIC will use this query plan as template for Q_3, thus generating a wrong query plan. On the other hand, Q_2 might have the same internal representation as Q_1 after query rewriting. In this case, Q_2 should use the same query plan

template as Q_1. Even if we require the classification be based on rewritten queries, PLASTIC may still miss matches. Consider another query Q_4 as follows:

Q_4: *select * from R, S where R.a=S.b and* ***R.a=5 and R.a2=9***

If the conjunctive predicate **R.a=5 and R.a2=9** selects the same number of tuples as the predicate **R.a=5**, Q_4 and Q_2 may share the same upper level join plan. However, PLASTIC will miss the match due to its syntactic matching algorithm. PLASTIC will not be able to detect such matching between customized queries unless we modify it to use the "property-based" matching in POM.

4.7 Experimental Evaluation

The POM prototype is implemented on top of PostgreSQL 8.0.3. We perform our experiments on a Pentium III 800HZ with 256Mb memory, running Linux Redhat 9.0. We create eight tables as in TPC-H, and duplicate each table twice to make a total of 24 tables in our database. These tables are populated using TPC-H tool with 150M data. More than twenty indexes are created on the primary key or foreign-key columns in these tables, such that each of the big tables ($lineitem, orders, partsupp, part, supplier, customer$) has one or two indexes.

Each uncustomized query is formulated over a subset of these 24 tables, ranging from 12 to 16 tables. The tables in each query are connected by join predicates randomly from a pool of predefined primary-foreign key join predicates. The queries are either star queries, or consist of small cliques of (three to five) tables fully connected by join predicates. There are no cartesian product allowed in these queries.

To simulate user customizations, for each uncustomized query we choose four big tables (from $lineitem$, $orders$, $partsupp$, $part$) as customizable tables on which to apply user customization. We simulate user customizations by creating a set of local predicates on these tables. Each of these local predicates is applied on an index-sargable attribute, or on a non-index-sargable attribute, or on a join attribute. We generate enough local predicates and pick a set that has a varying selectivity from $\frac{1}{6}$ to $\frac{2}{3}$. Thirty-three percent of the user customizations are single local predicates and 67% are compound local predicates. Among the compound local predicates 75% are conjunctions and 25% are disjunctions.

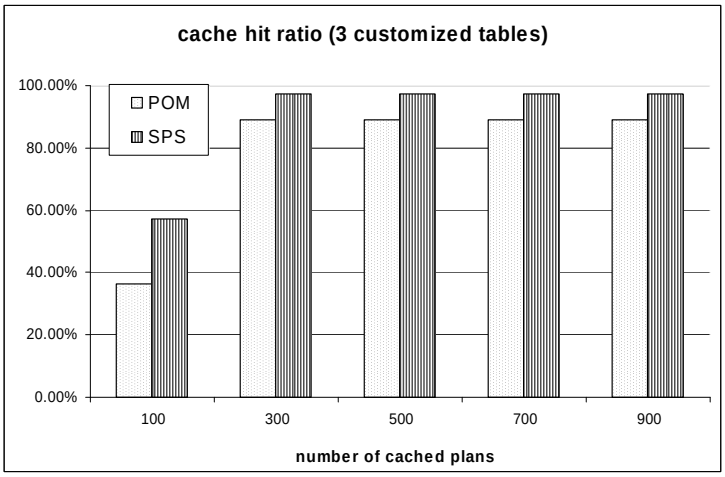
We set up the experiment by filling up the POM cache with 100 customized queries as *training queries*. We then feed POM with 500 customized queries as *test queries*. Both the training and test customized queries are derived from the same uncustomized query. We compare POM with a caching mechanism that resembles parametric query optimization, i.e., selectivity-based plan stitching (denoted as SPS). Given the training queries, SPS estimates the selectivity of user-customizations in each of these queries, and computes the optimal query plans based on the selectivities for each customized query. When doing this, SPS assumes the access paths on the

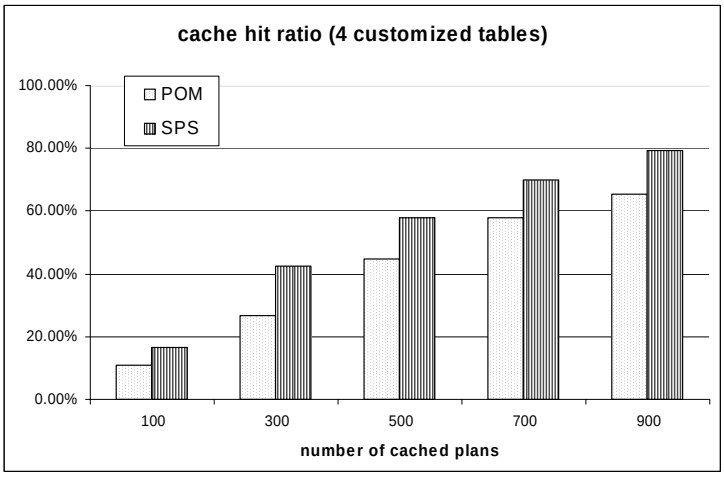
relations do not generate any interesting orders and have costs that correspond to table-scans. Then SPS caches the partial plans of these queries, together with the user-customizations' selectivities. When a new customized query is submitted, SPS computes the access paths for the query and matches the access path with the lowest cost to the partial plan that matches on Q_{top} and selectivities. Then the access paths are stitched to the matching partial plan as the query plan for the new query. This SPS approach is more space efficient than the enriched PQO approach described in Section 5.8 since it only caches the partial plans for customized queries that it has encountered. On the other hand, compared to POM, this SPS approach ignores *order* and *cost* properties, and does not have any optimality guarantee.

4.7.1 Cache Hit Ratio

The entries in the POM cache are determined by the customized queries, the cache size, and the cache replacement policy. We choose to use the Least-Recently-Used (LRU) policy for cache replacement. The cache hit ratio depends on the degree of approximation when matching on the *size* and the *cost* properties of access paths. If we insist on matching *sizes* and *costs* exactly, we can guarantee the optimality of the stitched query plan according to previous discussion. However, the cache hit ratio will be extremely low, and the saving on query optimization time from POM will be trivial. To avoid having extremely low cache hit ratio, we can relax the matching algorithm such that the *sizes* are matched if the size of the new access path is within a *bounding box* [64] of the *size* of the original access path. Matching on the *cost* property can be similarly defined. If we adopt such matching algorithm, the cache hit ratio will increase, and savings from query optimization will increases. However, such a relaxed matching algorithm will no longer guarantee the optimality of the stitched query plan. The difference of the stitched query plan and the optimal query plan will depend on the size of the bounding box, and the join operator on top of the access paths, as described in the work by Markl et al. [64]. In our experiments we define that two access paths are matched on their *size* property if their costs (C_1 and C_2) differ by at most 10%, i.e., $\frac{C_1 - C_2}{\max(C_1, C_2)} \leq 10\%$. Matching on cardinalities is defined using the same 10% bounding box. We choose this 10% empirically as it shows good tradeoff between cache hit ratio and quality of stitched plan. In the future we will experiment with different bounding boxes for each type of join operator to maximize cache hit ratio without hurting query plan quality too much.

We first compare the cache hit ratio between POM and SPS under varying cache sizes and query sizes. Given the same cache size, SPS needs to store the partial queries, the partial plans and the *size* property of the access paths, while POM needs to store the *order* and *cost* properties of the access paths in addition to what SPS stores. The number of non-chosen access paths depends on the number of indexes on the relations. In our experiment setting, small tables





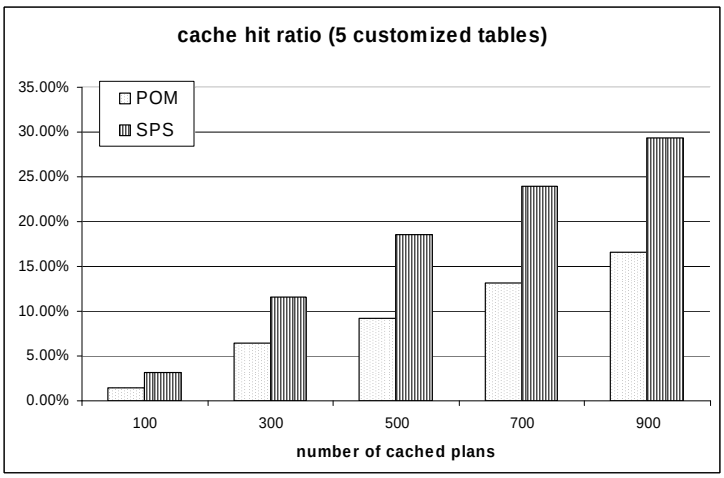


Figure 4.12: Cache hits for POM and SPS under different cache size and query size

like $nation$ or $region$ do not have indexes while large tables like $lineitem$ or $orders$ have two indexes each. In this setting, POM needs less than 8% more space than SPS to store the same number of query plans. Therefore, giving the same size of cache, SPS is able to store roughly 8% more partial plans than POM.

Even with the same amount of cached query plans, the cache hit ratio of SPS is higher than that of POM. This is because POM not only requires that query plans be matched on *sizes*, but also that they be matched on the $cost$ and the $order$ properties. Further, POM requires a potential match to pass the optimality check on non-matching access paths.

Figure 4.12 shows the number of cache hits from POM and SPS on 500 *test* customized queries under the same *number* of cached partial plans. Both POM and SPS are warmed up using 2500 *training* customized queries. We can see that the cache hits ratio of POM is mostly lower than SPS, and the larger the number of customizable tables, the larger the gap between SPS and POM. On the other hand, the gap of cache hit ratio between POM and SPS is smaller when the number of customizable tables is small. The reason is that SPS's *size* property space is fully covered and its cache becomes saturated, while POM continues to grow its coverage on a larger property space given more cache space.

Note the above comparison assumes the same *number* of cached partial plans for POM and SPS. It is anticipated that for the same fixed cache size the cache hit ratio would be roughly 8% more in favor of SPS, due to the space overhead of POM entries.

4.7.2 Query Plan Quality

Although POM loses to SPS in terms of cache hit ratio as expected, the quality of the query plans generated by POM is superior to that of SPS. We compare the costs of the stitched query plans from these two approaches. Figure 4.13 compares the performance of POM plans and SPS plans. The X-axis shows the cost of a SPS stitched plan while the Y-axis shows the cost of the POM stitched plan for the same query. A dot on the lower-right means that the POM stitched plan is superior. We can see that mostly the POM stitched plans are superior to the SPS query plans, except certain rare cases. A close look reveals that POM has fewer matching partial plans than SPS since SPS only matches based on selectivities. Therefore, SPS is able to compare more stitched plans and return the best one, while POM has only one matched partial plan.

Figure 4.14 shows the percentage of stitched plans from POM and SPS that differ from the optimal query plan as selected by the original optimizer. The percentage of deviant POM plans are consistently lower than that of SPS. Moreover, we see from Figure 4.15 that among the plans that differ from the optimal query plan generated using the un-modified optimizer, the plans from

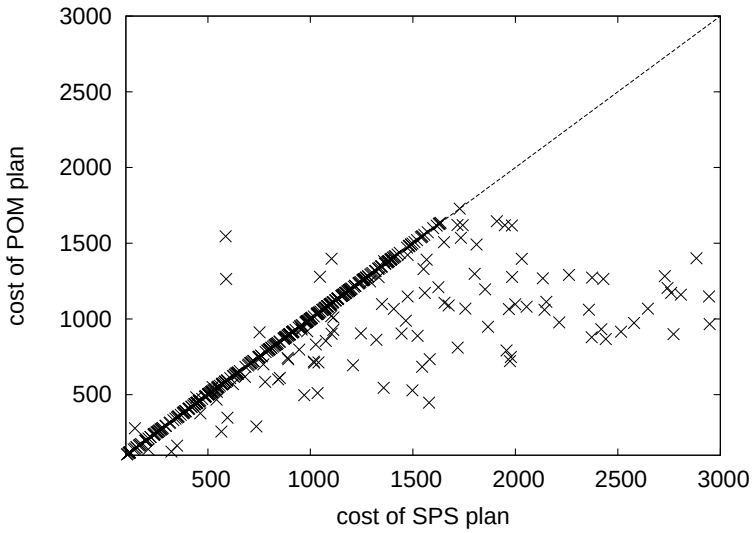


Figure 4.13: Cost of POM plan vs. cost of SPS plan

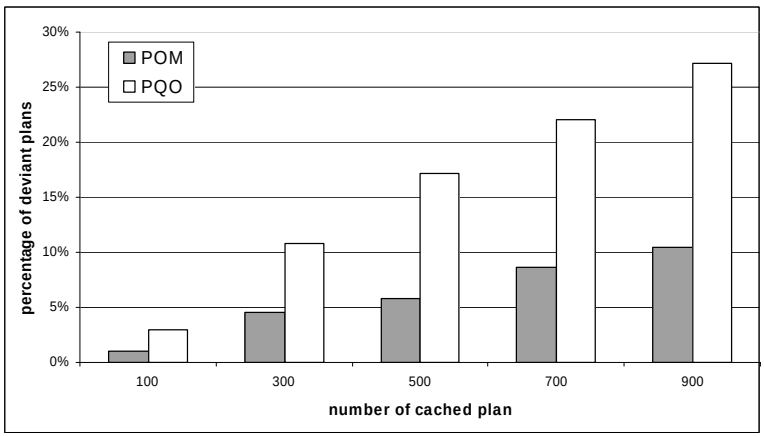


Figure 4.14: Percentage of stitched plans that differ from the optimal query plan from POM and SPS

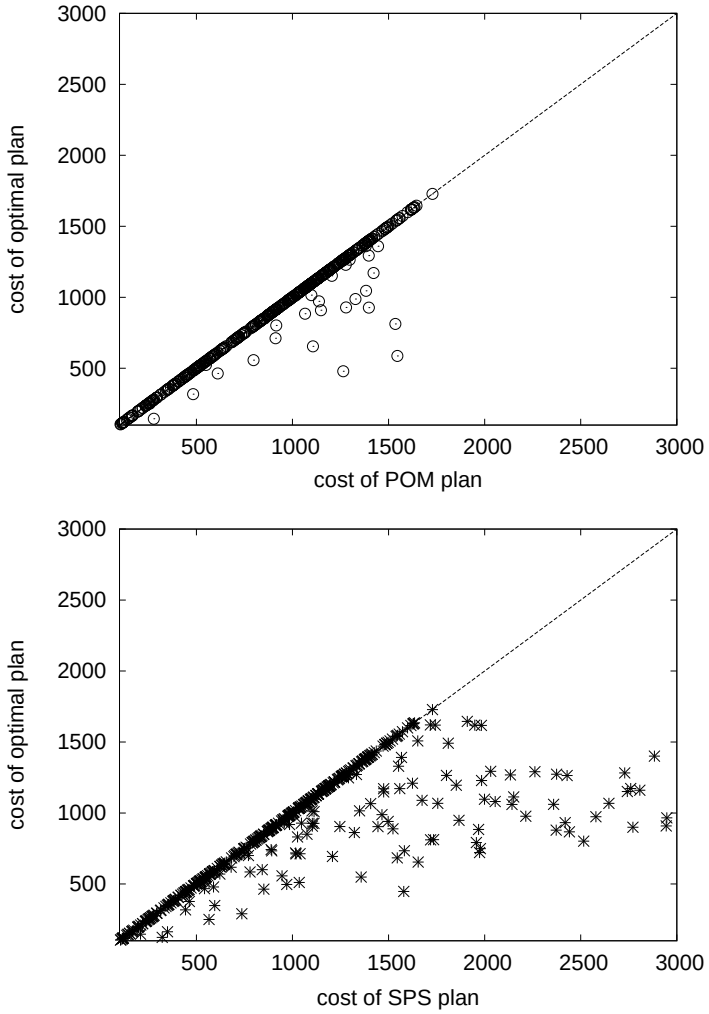


Figure 4.15: Cost of POM/SPS plans vs. cost of the optimal plan

POM are highly in accordance with the optimal plan. However, the SPS plans on the average deviate significantly from the optimal plans.

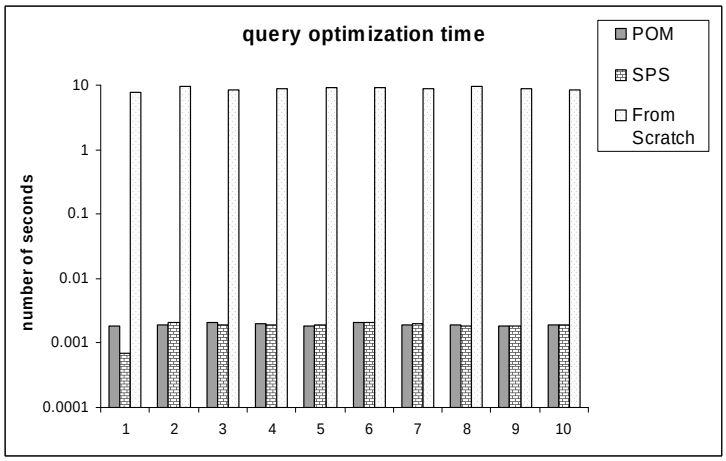


Figure 4.16: Query planning overhead between POM, SPS, and ND

4.7.3 Query Planning Overhead

We showed that POM generates better query plans on a cache hit than SPS, and that POM query plans are very close to the optimal query plans that are computed from scratch. We now compare the query planning overhead between POM, SPS, and the "Näive Dynamic" (ND) approach presented at the beginning of this chapter. We pick 10 customized queries that have matches from both POM and SPS, and record the query planning time for all three approaches. Figure 4.16 shows that for all these queries, POM and SPS have negligible query planning overhead. However, the ND approach takes significant amount of time for query planning from scratch.

4.7.4 Concluding Remarks on Experiments

Figure 4.17 shows the query planning cost and the cost of generated query plans for 500 queries between POM, SPS, and ND approaches. Both POM and SPS have 900 plan cache entries, and their caches are warmed up using 2500 training queries. The queries are all customized queries for the same base query, where the customizations are single local predicates randomly applied on five pre-defined tables. The X-axis shows the number of seconds to create all query plans for the 500 queries, and the Y-axis shows the average cost (in terms of PostgreSQL optimizer plan cost unit) of the 500 query plans under each approach. Compared to SPS, POM has a lower

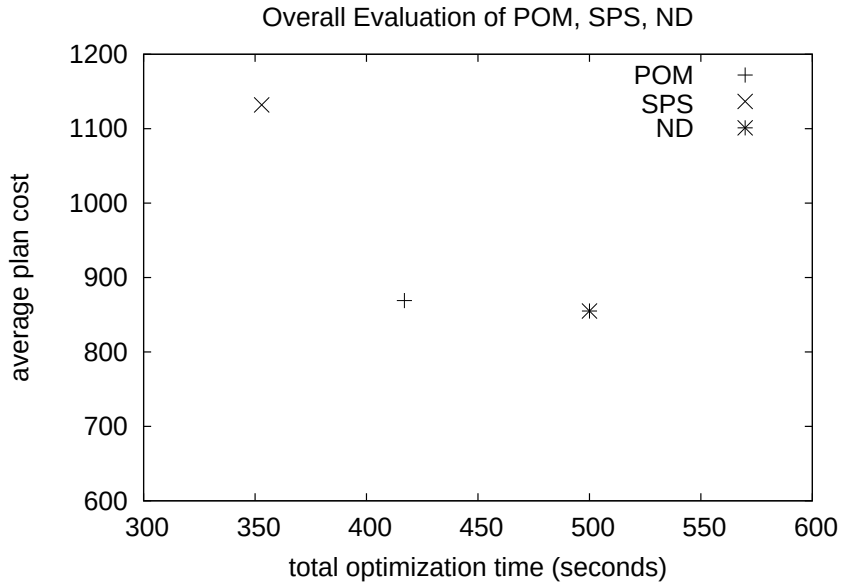


Figure 4.17: Query planning cost vs. plan quality between POM, SPS, and ND

cache hit ratio and a higher query planning cost, but POM query plans have much lower costs than SPS query plans.

It should be noted that Figure 4.17 is for illustrating the performance characteristics of POM, which is:

1. Total optimization cost of POM is always lower than ND, but higher than SPS.

2. Average plan cost from POM is close to that from ND, but lower than SPS.

For both SPS and POM, their total optimization cost depends on the queries. Complex queries have higher optimization cost. Thus the more complex the queries are, the more saving from a cache hit. However, more complex queries need more space to cache information about all of their access paths and their partial query plans. More complex queries may have more different upper level partial plans. Therefore, given the fixed cache size, both POM and SPS 's cache hit ratio will become low for more complex queries. It is anticipated that under a fixed cache size, both POM and SPS has its own "optimal" workload such that the reduction in its total optimization time is maximized.

On the other hand, and average query plan cost of POM and SPS depends on both the queries and the bounding box of our matching algorithm as described in Section 4.7.1. Larger bounding box may benefit the cache hit ratio, but may also increase average query plan costs. However,

some queries may be insensitive to larger bounding boxes while some queries may be significantly affected. To avoid having very bad query plans, we need to carefully choose the size of the bounding boxes according to the upper-level partial plans. Details of sizing bounding box can be found in Markl et al.'s work [64].

To compare SPS and POM, we need to balance the trade off between total query optimization time and average plan costs. The total query processing cost consists of query optimization cost and query execution cost, and can be formulated as follows, where H denotes cache hit ratio, O denotes optimization cost and E denotes query execution cost:

$$(1 - H) \cdot O + E$$

We ignore query matching and stitching time from POM and SPS since it remains a constant that is negligible relative to other costs. POM and SPS have different cache hit ratio (denoted by H_{pom} and H_{pqo} respectively), and their stitched plans have different costs (denoted by E_{pom} and E_{pqo}). The total query processing cost using POM is lower than SPS when the following holds:

$$\cdot O \leq \frac{E_{pqo} - E_{pom}}{H_{pqo} - H_{pom}}$$

Similarly, when comparing to the Näive Dynamic approach described in Section 4.1, POM becomes favorable when the following holds:

$$O \geq \frac{E_{pom} - E_{opt}}{H_{pom}}$$

The cache hit ratio can be guessed from running a subset of the workload. Query optimization cost and execution cost can be estimated using optimization time estimator [48] and execution time estimator [22, 62] respectively. Hence we have a way of choosing POM or alternatives based on quantitative analysis.

4.8 Discussion and Possible Extensions

Throughout this chapter, we assume that the optimizer separates logical rewriting from query plan generating. For an optimizer that intermingles these two tasks (e.g., SQL Server), we need to check at each query transformation step whether the top-level query Q_{top} remains the same. Example transformation rules that might change Q_{top} include adding or dropping join-predicates, removing unnecessary $distinct$ operator, flattening a nested subquery, using materialized view for answering query, any many more. If there is one such transformation, the partial plan computed cannot be used for other customized query. Applying this check at each step ensures there is no incorrect plan stitching.

We also assume all users' access-control customizations are local predicate formulae. The difference between users' customizations differentiates the costs of the access paths, which further differentiates the top-level join trees of their optimal query plans. That is why we partition optimization between the *leaf queries* and the partial queries, and cache the partial plans for the varying plan leaves generated from users' customizations. However, if POM builds all access paths *plus* all pair-wise join on these access paths at runtime, and find matching partial plans based on information from *both* the access paths and the pair-wise joins, it is able to handle user-customizations as join-predicates. The reason is that query plans are built based on lower-level subplans, and if we have information on the *properties* of all subplans on single relations and two relations, we will be able to know what top-level query plan will be. Another way to understand this is by considering all pair-wise joins on access paths as materialized views. If two queries have the same pair-wise matching access paths, and same set of materialized views that might be useful for answering this query, their top-level join trees must be the same. In this way, POM is able to handle all join predicates involving two relations. However, the number of pair-wise joins grows exponentially to the size of query, and the cache hits based matching both the access paths and the pair-wise joins drop exponentially. Therefore, the expense on storage and runtime computation may not be worth the marginal runtime optimization saving.

Chapter 5

Accurate Sampling for Cardinality Estimation

5.1 Introduction

Recall that under the *Truman Model* [77] described in Section 2.3.1, a user's query is answered as if the database includes only those tuples that are accessible to the user. Oracle implements the Truman model through its *Virtual Private Database*, which extracts an *access control predicate* from the user's profile and appends the predicate to the *where* clause of the user's query [70]. Similarly, DB2 Label-Based Access Control [45] implements the Truman model by hiding every tuple having an inaccessibility label from the user. In both cases, the fine-grained access controls become *access control filters* applied to the tuples from the base relations.

More formally, suppose that a query includes a predicate P_Q on one of its input relations, T. P_Q may be a simple or complex predicate. Suppose further that each user's access rights for the target relation are defined by a user-specific access control predicate. We use P_{ACi} to denote the access control predicate for the i^{th} user on the target relation. Tuples are accessible to user i if and only if they satisfy P_{ACi}. The form of P_{ACi} is not important. It may be an actual SQL predicate, as in Oracle VPD [70], or it may be implemented as an externally-defined function, as is the case in some fine-grained label-based access control systems [45, 66, 69]. For the purposes of our work, it is only necessary to be able to evaluate the predicate given a tuple from the target relation. If the query is being executed on behalf of the i^{th} user, the effect of the fine-grained access controls is to replace P_Q in the query with $P_Q \wedge P_{ACi}$.

To create efficient query plans under this setting, a query optimizer needs accurate cardinality estimates that account for the effect of the access control filters. That is, instead of estimating the number of tuples that satisfy a particular query predicate, the optimizer must estimate the number

of tuples that satisfy *both* the query predicate and the access control filters, i.e., the number of tuples satisfying $P_Q \wedge P_{ACi}$. This chapter addresses the problem of accurately estimating the number of such satisfying tuples.

There are many approaches for estimating the number of satisfying tuples for $P_Q \wedge P_{ACi}$ through the conjunctive selectivity of P_Q and P_{ACi}. Among them the most widely used technique uses multi-dimensional histograms. We can treat P_{ACi} just the same as P_Q and build multi-dimensional histogram on the attributes in P_Q and P_{ACi}. However, this approach of treating P_{ACi} and P_Q alike inherits all the problems of cardinality estimation with multiple attributes.

1. First, the users' access controls can be based on a large number of *different* attributes. It is hard to anticipate all possible correlations between the query attributes and different users' access control attributes. If we are going to build multidimensional histograms for all possible groups of correlated attributes from P_Q and P_{ACi}, the cost of building such histograms can be prohibitively expensive.

2. Second, if we do not exhaustively build all the histograms, certain $P_Q \wedge P_{ACi}$ will not be accommodated with relevant statistics, and we will have to rely on the *attribute value independence* (AVI) assumption on the query attributes and the access control attributes. It is well-known that cardinality estimation techniques based on the AVI assumption would generate results that are wildly off the mark.

In fact, treating P_{ACi} as a normal SQL predicate does not recognize nor leverage the special feature of the access control predicates: they are known *a priori*. The access control predicates are also relatively static: a user's access rights change only in response to changes in the system's access control policies or the users' position in the company, and we expect that such changes will occur much less frequently than adhoc queries do. Moreover, if P_{ACi} is expressed as complex functions on access control specifications bundled with each tuple, as in the case of DB2 Label-based Access Controls [45], Oracle Label Security [69], or SQL-Server security mechanism [66], we simply cannot build multi-dimensional histograms to estimate the cardinality of $P_Q \wedge P_{ACi}$.

In this chapter, we propose an approach that leverages the fact that P_{ACi} is known *a priori* and is relatively static. Our approach is essentially a lazy way to achieve the same accuracy as the prohibitively expensive multi-dimensional histograms, by projecting sample data on the views defined through the access control predicates P_{ACi}. Since the samples are already projected from P_{ACi}, this approach does not suffer from the high-dimensional correlation brought from the access control attributes. We will show that this approach not only provides more accurate estimation than multi-dimensional histograms under the same sample space budget, but also has a system-wide accuracy guarantee, a feature that can not be achieved using multi-dimensional histograms. The samples also support accurate cardinality estimation with disjunctions on the ac-

Table 5.1: Symbols used in this chapter

U	number of users in the system
P_Q	query selection predicate
P_{ACi}	access control predicate for the i^{th} user
N	cardinality of the target relation
N_i	number of tuples matching P_{ACi}
$N_{i,j}$	number of tuples matching both P_{ACi} and P_{ACj}
C_i	number of tuples matching $P_Q \wedge P_{ACi}$
$\tilde{C}_i$	estimate of C_i
n_i	number of tuples in i^{th} user's private sample
c_i	number of tuples in i^{th} user's private sample that satisfy $P_Q \wedge P_{ACi}$
ϵ_i	estimation error for i^{th} user
ϵ	mean estimation error over all users
ϵ	maximum estimation error over all users
$\hat{\epsilon}_i$	estimation error bound for i^{th} user
$\hat{\epsilon}_{mean}$	mean estimation error bound over all users
Δ	confidence level for estimation error bounds

cess control predicates, which enable cardinality estimation in the presence of role-based access controls (see more in Section 5.7).

The remainder of the chapter is organized as follows. Section 5.2 introduces preliminaries for selectivity estimation in the presence of fine-grained access controls, including terminology and notation. Section 5.3 presents basic-PSALM and compares it to simple random sampling. Section 5.4 formalizes a general, but computationally intractable approach to improve basic-PSALM. Section 5.5 and Section 5.6 describe two practical steps to improve the accuracy of basic-PSALM by using a hybrid scheme and a refinement procedure that exploits access rights correlations between users. The ordering of these two steps matters, as described in Section 5.6. Section 5.7 describes how to use PSALM to estimate cardinality in the presence of role-based access controls. Section 5.8 compares these techniques with related work.

5.2 Definition and Notation

As discussed in Section 5.1, our problem is to estimate the selectivity of query predicates in the presence of access control predicates. We will focus on the problem of estimating the cardinality

of the result of applying the query predicate, P_Q, to a single access-controlled relation. When there are multiple relations, our estimation techniques can be applied independently to each relation.

We use P_{ACi} to denote the access control predicate for the i^{th} user on the target relation. We use N to denote the cardinality of the target relation, and N_i to denote the number of tuples from the target relation that are accessible to the i^{th} user. That is, N_i is the number of target relation tuples for which P_{ACi} is true. Finally, we use C_i to denote the number of tuples from the target relation that satisfy $P_Q \wedge P_{ACi}$. C_i is the cardinality we wish to estimate, for any given user, target relation, and query predicate P_Q. Table 5.1 summarizes the notation that we use in this chapter.

Given a fixed space budget for cardinality estimation, our goal is to design estimation techniques with small estimation error. We use $\tilde{C}_i$ to denote the cardinality estimate produced by one of the estimation techniques to be presented in this chapter. Following earlier work in this area [6, 21], we define ϵ_i, the estimation error for the i^{th} user, to be

$$\epsilon_i = \frac{|\tilde{C}_i - C_i|}{C_i}$$

This metric characterizes the estimation error relative to the actual cardinality of the query result. For example, $\epsilon_i = 0.3$ indicates that the estimate is $\pm 30\%$ of the actual cardinality. Unlike absolute error metrics such as $|\tilde{C}_i - C_i|$ or $|\tilde{C}_i - C_i|/N$, our relative error metric reflects the fact that the same cardinality estimation error may be more significant to the query optimizer when the true cardinality is small than when the true cardinality is large. For example, if $|\tilde{C}_i - C_i| = 10000$ and $\tilde{C}_i = 100000$, the estimation error may have little effect on the optimizer. However, if $|\tilde{C}_i - C_i| = 10000$ and $\tilde{C}_i = 100$, the optimizer may significantly underestimate the cost of a candidate query plan. One disadvantage of our metric is that estimation error explodes as $C_i \rightarrow 0$. To avoid the blowup, we simply avoid scenarios in which C_i is extremely small when comparing the accuracy of estimations. Please note that our cardinality estimation techniques do not require calculating estimation error. Estimation error is used only for comparing the estimation techniques.

5.3 Single-Sampling and Basic-PSALM

We begin by presenting two simple sampling-based estimation techniques. The first uses a single uniform random sample to generate a cardinality estimate for any user. That is, estimates for all users are based on the same sample from the target relation. The second technique is *basic Partitioned SAmpling for MuLtiple users*, or *basic-PSALM*. It partitions the available space and

draws a separate, smaller sample for each user. Each user's cardinality estimations are based on that user's private sample.

We determine the bounds on the estimation error resulting from each technique, and characterize the situations in which basic-PSALM provides more accurate estimates than the single-sample approach. In the subsequent sections, we will present techniques that improve the basic-PSALM technique such that it always provide more accurate estimates than the single-sample approach.

5.3.1 Single-Sampling

We can estimate the cardinality of $P_Q \wedge P_{ACi}$, for any user and any query predicate, using a single random sample of n tuples from the target relation. Such a sample can be built with a single pass over the target relation, using a technique such as reservoir sampling [88]. If desired, such a sample can be incrementally maintained in the face of tuple insertions and deletions in the target relation [38]. Alternatively, the target relation can be periodically resampled as necessary to account for changes.

To obtain a cardinality estimate for P_Q for the i^{th} user, we evaluate $P_Q \wedge P_{ACi}$ for each sample tuple, and count the number of tuples for which the predicate is true. An unbiased cardinality estimate can then be obtained by

$$\tilde{C}_i = c_i \frac{N}{n}$$

where c_i is the number of sample tuples matching $P_Q \wedge P_{ACi}$.

Because $\tilde{C}_i$ is based on a random sample of tuples rather than the entire relation, the estimate may not be accurate. Since each sample tuple satisfies $P_Q \wedge P_{ACi}$ with probability $\frac{C_i}{N}$, c_i can be modeled as a binomial random variable with parameters n and $\frac{C_i}{N}$. Using the Chernoff inequality, we can bound the estimation error as follows:

$$\begin{aligned} & Prob[\tfrac{|\tilde{C}_i - C_i|}{C_i} \leq \epsilon] = Prob[\tfrac{|\frac{c_i N}{n} - C_i|}{C_i}| \leq \epsilon] \\ = \;& Prob[(1-\epsilon) n \tfrac{C_i}{N} \leq c_i \leq (1+\epsilon) n \tfrac{C_i}{N}] \\ \geq \;& 1 - 2e^{(-nC_i\epsilon^2)/(4N)} \end{aligned}$$

Thus, with probability $(1 - \Delta)$, the cardinality estimation error ϵ_i for the i^{th} user under the single random sample method is bounded by

$$\epsilon_i \leq \hat{\epsilon}_i = \sqrt{\frac{4N}{nC_i} \log \frac{2}{\Delta}} \tag{5.1}$$

We refer to Δ as the confidence level of the error bound $\hat{\epsilon_i}$. Note that the estimation error bound is inversely related to C_i which is the number of tuples that satisfy $P_Q \wedge P_{ACi}$. Thus, as either the query predicate P_Q or the user's access controls P_{ACi} become more selective, the estimation error bound increases.

5.3.2 Basic-PSALM

Another way to estimate the cardinality of $P_Q \wedge P_{ACi}$ is to create and maintain a separate sample for each user. Let n_i denote the size of the sample for the i^{th} user, and let U represent the number of users. We can choose the n_i such that $\sum_{i=1}^{U} n_i = n$ to ensure that basic-PSALM technique uses the same amount of space as the single-sample approach.

Under the basic-PSALM approach, the sample for the i^{th} user is a simple uniform random sample of the tuples *that are accessible to the i^{th} user*. As was the case for the single-sample approach, we can draw all U such random samples using a single pass over the target relation by maintaining Useparate reservoir samples in parallel during the scan. Each tuple encountered in the scan is considered separately and independently for inclusion in the sample for each user. For the i^{th} user, the tuple is first tested against P_{ACi}. If the tuple satisfies the predicate, then it is considered for inclusion in the reservoir sample for user i as usual. Otherwise, it is not included in the i^{th} sample.

To estimation cardinality for P_Q for the i^{th} user, we evaluate P_Q for each tuple in the i^{th} user's sample, and count the number of tuples for which the predicate is true. All other users' samples are ignored. An unbiased cardinality estimate can then be obtained by

$$\tilde{C}_i = c_i \frac{N_i}{n_i}$$

where c_i is the number of sample tuples matching P_Q in the i^{th} user's sample. Notice that this estimator makes use of N_i, the total number of target relation tuples that are accessible to the i^{th} user. This value can be determined exactly for every user during the same scan that is used to draw the tuple samples from the target relation.

Using an analysis similar to the one in Section 5.3.1, we can bound, with confidence level Δ, the estimation error that results from the basic-PSALM technique:

$$\epsilon_i \leq \hat{\epsilon_i} = \sqrt{\frac{4N_i}{n_i C_i} \log \frac{2}{\Delta}} \tag{5.2}$$

Recall that under the single sample approach, the estimation error bound is inversely related to the selectivity of the joint predicate $P_Q \wedge P_{ACi}$. For the basic-PSALM technique, the estimation error is inversely related to C_i/N_i, which is the *conditional selectivity* of the query predicate P_Q,

given that a tuple is accessible to the i^{th} user. Unlike the single sample approach, basic-PSALM's estimation error is independent of the selectivity of the users' access control predicates. This makes sense, because the basic-PSALM technique ensures that it has a sample of size n_i of tuples accessible to the i^{th} user, regardless of how many such tuples exist.

One question remains for the basic-PSALM technique: how should we determine the number of tuples, n_i, to include in the sample for the i^{th} user? To answer this question, we define $\hat{\epsilon}_{mean}$ as the mean estimation error bound for all users in the system:

$$\hat{\epsilon}_{mean} = \frac{\sum_{i=1}^{U} \hat{\epsilon}_i}{U}$$

For example, from Formula 5.1 we have the mean estimation error bound for a single uniform sampling as the following:

$$\frac{1}{U} \sum_{i=1}^{U} \sqrt{\frac{4N}{n \cdot C_i} \log \frac{2}{\Delta}} \tag{5.3}$$

Similarly, the mean estimation error bound for basic-PSALM is

$$\frac{1}{U} \sum_{i=1}^{U} \sqrt{\frac{4N_i}{n_i \cdot C_i} \log \frac{2}{\Delta}} \tag{5.4}$$

Ideally, we would like to distribute the available space among the users' samples in basic-PSALM in such a way as to minimize the mean estimation error bound. In general, the best such distribution will be workload dependent, determined by the conditional selectivities of the query predicates with respect to each user's accessible tuples. However, for the special case in which the query predicates P_Q in the workload are independent of the users' access controls, there is a simple workload-independent way to determine optimal per-user sample sizes such that the mean estimation error bound is minimized:

Theorem 6. *If the query predicate P_Q is independent of the access control predicates P_{ACi} of all users, then the expected $\hat{\epsilon}_{mean}$ is minimized by choosing $n_i = n/U$ for every user.*

Proof. Basic-PSALM has a mean estimation error bound of

$$\frac{\sum_{i=1}^{U} \hat{\epsilon}_i}{U} = \frac{\sum_{i=1}^{U} \sqrt{\frac{4N_i}{n_i C_i} \log \frac{2}{\Delta}}}{U}$$

Because we assume P_Q is independent of the access control predicates, (N_i/C_i) is in expectation the same for all users, and we can rewrite the mean estimation error as

$$\alpha \sum_{i=1}^{U} \sqrt{\frac{1}{n_i}}$$

where α is a constant term independent of i. Our goal is to minimize this quantity by adjusting $\{n_i, i \in (1, \ldots, U)\}$ under the constraint that $\sum_{i=1}^{U} n_i = n$. We use the method of Lagrange multipliers. Adding the constraint to our formula, we have:

$$\Phi(X, \lambda) = \alpha \sum_{i=1}^{U} \sqrt{\frac{1}{n_i}} + \lambda \left(\sum_{i=1}^{U} n_i - n \right)$$

The critical value of Φ occurs when the gradients on $\{n_i, i \in (1, \ldots, U)\}$ and λ are all zero:

$$\frac{\partial \Phi}{\partial n_i} = \frac{\partial \frac{\alpha}{\sqrt{n_i}}}{\partial n_i} + \lambda = -\alpha n_i^{-\frac{3}{2}} + \lambda = 0, i \in \{1, \ldots, U\}$$

$$\frac{\partial \Phi}{\partial \lambda} = \sum_{i=1}^{U} n_i - n = 0$$

Solving the above, we have $n_i = \frac{n}{U}, i \in (1, \ldots, U)$. □

Note that we are assuming independence of the query predicate and the access controls *only* for the purpose of choosing the sizes of the per-user samples. The actual estimation of cardinalities using those samples does not rely on the independence of query predicates and access controls.

Comparing Formula 5.1 and Formula 5.2, we can see that the i^{th} user will have a tighter error bound in basic-PSALM under the condition:

$$\frac{n_i}{N_i} > \frac{n}{N}$$

This condition states that basic-PSALM will provide more accurate estimates than the single-sample approach for the i^{th} user if the i^{th} user's accessible tuples are represented more in the samples. In Assuming that the basic-PSALM sample sizes are chosen according to Theorem 6 ($n_i = n/U$), then basic-PSALM will have a tighter estimation error bound for the i^{th} user if

$$N_i < \frac{N}{U} \tag{5.5}$$

Thus, users with very limited access rights, i.e., highly selective access control predicates, will benefit from the basic-PSALM technique. However, users with broad access rights may

experience larger estimation errors. Over all of the users in the system, we have the following theorem concerning the mean estimation error bound:

Theorem 7. *If the query predicate P_Q is independent of the access control predicates P_{ACi} of all users, basic-PSALM using the sample quota assignment from Theorem 6 has a lower mean estimation error bound $\hat{\epsilon}_{mean}$ than the single sample approach if*

$$\sum_{i=1}^{U} \sqrt{\frac{N}{N_i}} > U^{\frac{3}{2}}$$

Proof. When (N_i/C_i) is in expectation the same for all users, the mean estimation error bound of basic-PSALM reaches its minimum $(\alpha U \sqrt{\frac{U}{n}})$ when we assign equal sample quota to each user, and the mean estimation error bound of uniform sampling is $(\alpha \sum_{i=1}^{U} \sqrt{\frac{N}{nN_i}})$. Here α is a constant. Comparing these two values, we get the theorem. □

5.4 Generalized Approach to Improve Basic-PSALM

The condition in Theorem 7 is very strict. In most cases it will not hold for a realistic multi-user system, which means that basic-PSALM fails to out-perform single-sampling in terms of mean estimation error bound.

To improve on basic-PSALM, we first look at the expression of the basic-PSALM's mean estimation error bound (Formula 5.2). We notice that there are three sets of variables that determine the value of this expression. The first set of variables are the sizes of users' accessible data (N_i). The second set of variables are the sizes of data satisfying the joint predicate (C_i). The third set of variables are the sizes of samples that are uniformly taken from the users' accessible data (n_i). The first and the second sets of variables are fixed once P_{ACi} and P_Q are determined. Therefore, we aim at increasing n_i for all of the users. Our general approach is to group users and let them share tuples from their private samples. If a sample tuple is shared by more than one user in a group, we can remove the duplicates from the private samples of the users in this group. This allows us to sample more tuples from the users' accessible data in the same amount of space. If the users have more sample tuples from their accessible data, the users will have lower cardinality estimation error bounds.

More formally, given U samples (one per user) such that the total sample size is n, we partition the users into k groups $\mathcal{G} = \{G_1, G_2, \ldots G_k\}$. We assign a sample size n_i to each group $G_i, 1 \leq i \leq k$, and the sample tuples for each group are extracted uniformly at random from the *union* of the sets of accessible tuples for each user in the group. If we use $Gr(i)$ to denote user u_i's group,

and $N_{Gr(i)}$ to denote the size of the union of the accessible tuples from all users in $Gr(i)$, and $n_{Gr(i)}$ to denote the size of the private sample for $Gr(i)$, then $\left(n_{Gr(i)}(\frac{N_i}{N_{Gr(i)}})\right)$ is the expected number of user u_i's accessible tuples in his group's private sample.

We estimate the cardinality for $P_Q \wedge P_{ACi}$ by applying $P_Q \wedge P_{ACi}$ on user u_i's groups' private sample. If c_i is the number of matching tuples from the private sample, we estimate the cardinality of tuples satisfying $P_Q \wedge P_{ACi}$ as:

$$\tilde{C}_i = c_i \frac{N_i}{\left(n_{Gr(i)}(\frac{N_i}{N_{Gr(i)}})\right)}$$

Our goal is to find a grouping of the users (e.g., the groups and the group mapping function), and a way to decide each group's sample size n_i so that $\sum_{1 \le i \le k} n_i = n$, and the mean estimation error bound from all users is minimized. By referring to Formula 5.2, and assuming (N_i/C_i) is in expectation the same for all users, our goal is equivalent to minimizing the following:

$$\sum_{1 \le i \le U} \sqrt{\frac{4 \log \frac{2}{\delta}}{n_{\mathcal{F}(i)}(\frac{N_i}{N_{G_{\mathcal{F}(i)}}})}} \tag{5.6}$$

5.4.1 Computing the Optimal Sample Size Assignment

We have the following theorem on how to analytically compute the optimal sample size assignment:

Theorem 8. *Under a user grouping $\mathcal{G}$, the optimal sample assignment $n_j, 1 \le j \le |\mathcal{G}|$ will be the following:*

$$n_j = \frac{n \cdot H_j^{\frac{1}{3}}}{\sum_{j=1}^{|\mathcal{G}|} H_j^{\frac{1}{3}}}$$

Here H_j is $\sum_{(i:u_i \in Gr(j))} \frac{N_{Gr(j)}}{N_i}$, which is fixed constant once the user-grouping is chosen. Under this optimal sample size assignment, the mean estimation error bound will be:

$$\sqrt{4 \log \frac{2}{\delta}} \cdot n^{\frac{1}{2}} \left(\sum_{j=1}^{|\mathcal{G}|} H_j^{\frac{1}{3}} \right)^{\frac{3}{2}}$$

Proof. Formula 5.6 can be rewritten as follows:

$$\sum_{1\leq j\leq|\mathcal{G}|}\sqrt{\frac{4\log\frac{2}{\delta}\cdot N_{Gr(j)}}{n_j}\cdot\sum_{(i:u_i\in Gr(j))}\frac{1}{N_i}} \tag{5.7}$$

If we have chosen a user grouping (and thus a fixed $\mathcal{G}$), we can rewrite the above as

$$\sum_{j=1}^{|\mathcal{G}|}\left(\alpha\sqrt{\frac{H_j}{n_j}}\right) \tag{5.8}$$

Here α equals $\sqrt{4\log\frac{2}{\delta}}$, which is a constant, H_j is $\sum_{(i:u_i\in Gr(j))}\frac{N_{Gr(j)}}{N_i}$, which is fixed once the user-grouping is chosen.

The Lagrange formula for the minimum average estimation error bound becomes

$$\sum_{j=1}^{|\mathcal{G}|}\left(\sqrt{\frac{H_j}{n_j}}\right)+\lambda\left(\sum_{j=1}^{|\mathcal{G}|}n_j-n\right) \tag{5.9}$$

Then we have

$$\lambda=\frac{1}{2}H_j^{-\frac{1}{2}}{n_j}^{-\frac{3}{2}} \tag{5.10}$$

From the above and $\sum_{j=1}^{|\mathcal{G}|}n_j-n=0$ we have:

$$\lambda=\frac{\left(\sum_{j=1}^{|\mathcal{G}|}H_j^{\frac{1}{3}}\right)^{\frac{3}{2}}}{2n^{\frac{3}{2}}} \tag{5.11}$$

Then, we have the following:

$$n_j=\left(\frac{\left(\sum_{j=1}^{|\mathcal{G}|}H_j^{\frac{1}{3}}\right)^{\frac{3}{2}}}{n^{\frac{3}{2}}\cdot H_j^{\frac{1}{2}}}\right)^{-\frac{2}{3}}=\frac{n\cdot H_j^{\frac{1}{3}}}{\sum_{j=1}^{|\mathcal{G}|}H_j^{\frac{1}{3}}} \tag{5.12}$$

We put this back to Equation 5.8, then we have the mean estimation error bound as:

$$\alpha \cdot n^{\frac{1}{2}} \left(\sum_{j=1}^{|\mathcal{G}|} H_j^{\frac{1}{3}} \right)^{\frac{3}{2}} \tag{5.13}$$

□

The problem now reduces to finding a grouping over the users such that the following is minimized:

$$\sum_{j=1}^{|\mathcal{G}|} H_j^{\frac{1}{3}} \tag{5.14}$$

5.4.2 Finding an Optimal User-grouping

The grouping of the users that minimizes the value of Formula 5.14 lies in the search space of all possible groups over the users. The number of possible groupings of N users is the same as the number of partitions of N different item, which is known as the Bell Number [93]. If we use B_n to denote the Bell Number of n items, we have the following recursive representation for B_n [93]:

$$B_{n+1} = \sum_{k=0}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) B_k$$

which has a non-recursive representation as Dobinski's formula [93]:

$$B_n = \frac{1}{e} \sum_{k=0}^{\infty} \frac{k^n}{k!} \tag{5.15}$$

It is obvious from Formula 5.15 that the number of partitions B_n is exponential in the number of items n, which means a brute force enumeration of all possible user-groupings is infeasible. Even if we apply techniques such as dynamic programming to solve the problem, the runtime remains exponential in general. The existence of a polynomial time solution that computes the optimal partition is still unknown. However, in Section 5.5 and Section 5.6, we introduce two polynomial-time algorithms that rely on heuristics to aggressively prune the search space. These algorithms are applied in an ordered fashion, and we will describe the reason for this ordering in Section 5.6.

5.5 Exploiting Access Privilege Skew

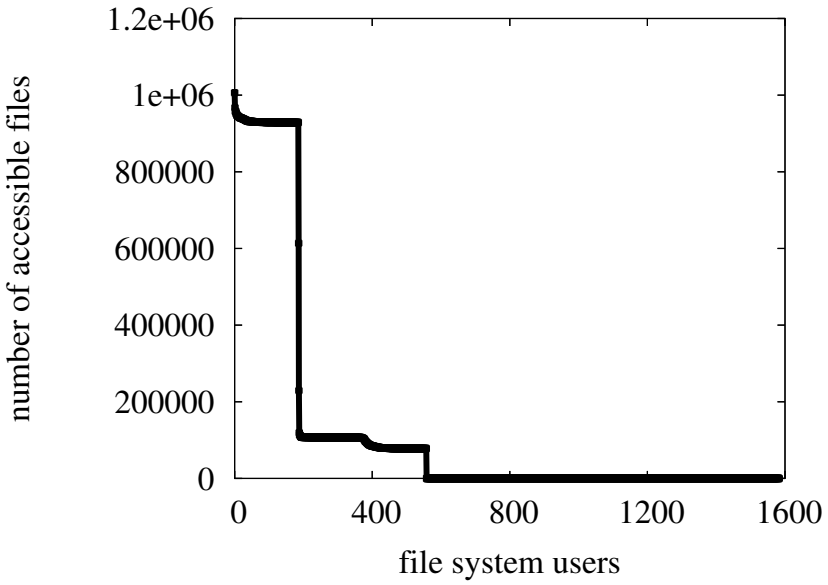


Figure 5.1: Amount of accessible data for users in University of Waterloo file system

Formula 5.5 shows that basic-PSALM works better than single-sampling for those users having few accessible tuples. However, basic-PSALM does not provide a lower mean estimation error bound than single-sampling if the condition in Theorem 7 is not satisfied. In this section, we develop a hybrid technique that combines advantages of both approaches, and we show that this hybrid technique always provides a mean estimation error bound that is at least as tight as those provided by single-sampling and basic-PSALM. Moreover, this hybrid estimation technique runs in time linear in the number of users.

Figure 5.1 shows the amount of data accessible to various users in a UNIX file system at University of Waterloo, and Figure 5.2 shows the amount of data accessible to various users in a Opentext Livelink content management system. In both systems the amount of data accessible to different users is highly skewed: some *high-privileged users* have access to most of the data while many *low-privileged users* have very limited access. This skew suggests the use of a hybrid approach, which we call *hybrid-PSALM*, where the users are divided into two groups; one group consisting of high privilege users and the other consisting of low privilege users. We will denote the high privilege and low privilege groups by $\mathcal{U}_H$ and $\mathcal{U}_L$, respectively. Later, we will show how to decide which group each user should belong to.

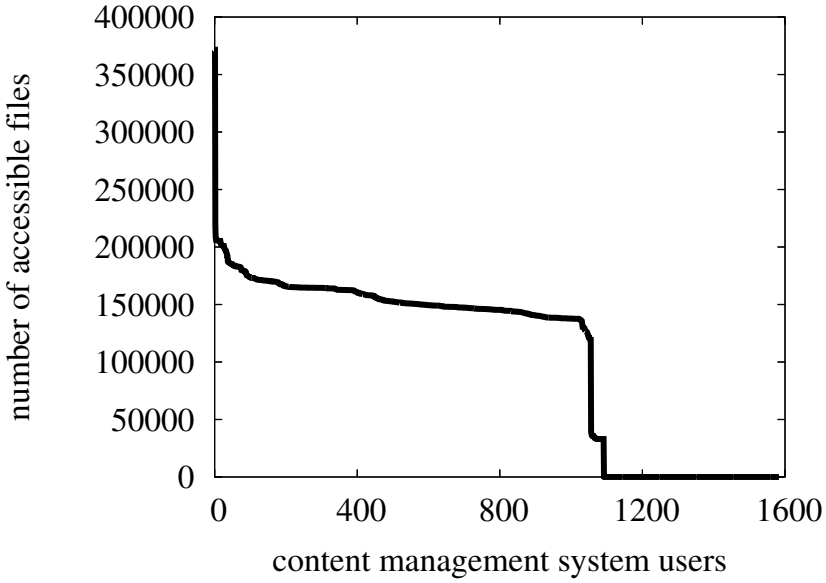


Figure 5.2: Amount of accessible data for different users in Opentext Livelink system

Hybrid-PSALM maintains a single shared sample for all users in $\mathcal{U}_H$, and a separate private sample for each individual user in $\mathcal{U}_L$. We will use n_0 to represent the size of the shared sample for the users in $\mathcal{U}_H$. If the i^{th} user is in $\mathcal{U}_L$, we will use n_i to represent the size of that user's private sample.

To obtain a cardinality estimate for P_Q for the i^{th} user, hybrid-PSALM proceeds as follows:

- If the i^{th} user is in $\mathcal{U}_H$, we evaluate $P_Q \wedge P_{ACi}$ against the tuples in the shared sample. If c_i is the number of matching tuples from the shared sample, then the cardinality estimate for P_Q is

$$\tilde{C}_i = c_i \frac{N}{n_0}$$

  As was the case for the single-sample approach, we can show that the estimation error for this estimate is bounded, with confidence level Δ, by

$$\epsilon_i \leq \sqrt{\frac{4N}{n_0 C_i} \log \frac{2}{\Delta}} \tag{5.16}$$

- If the i^{th} user is in $\mathcal{U}_L$, evaluate P_Q against the tuples in that user's private sample. If c_i is the number of matching tuples from the private sample, the cardinality estimate is

$$\tilde{C}_i = c_i \frac{N_i}{n_i} \tag{5.17}$$

and the estimation error for this estimates is bounded, with confidence level Δ, by

$$\epsilon_i \leq \sqrt{\frac{4N_i}{n_i C_i} \log \frac{2}{\Delta}} \tag{5.18}$$

To implement this hybrid approach, we must determine which users belong in $\mathcal{U}_L$ and which belong in $\mathcal{U}_H$. In addition, we must determine the sizes n_i of the private samples for users in $\mathcal{U}_L$ and the size n_0 of the shared sample for those in $\mathcal{U}_H$. As was the case for basic-PSALM, our goal is to minimize the mean estimation error bound $\hat{\epsilon}_{mean}$ over all of the users.

5.5.1 Minimizing the Mean Estimation Error Bound

The following theorem tells us how to choose sample sizes given an arbitrary partitioning of the users into two groups, $\mathcal{U}_H$ and $\mathcal{U}_L$.

Theorem 9. *Suppose that the users are partitioned arbitrarily into two groups, $\mathcal{U}_L$ and $\mathcal{U}_H$, such that each user in $\mathcal{U}_L$ is given a private sample for cardinality estimation while all users in $\mathcal{U}_H$ share a single common sample, and that the sum of the sample sizes is n. If query predicates P_Q are independent of the access control predicates of all users, then the mean estimation error bound $\hat{\epsilon}_{mean}$ for hybrid-PSALM is minimized when the sample sizes are chosen as*

$$n_0 = n \frac{Y^{\frac{2}{3}}}{Y^{\frac{2}{3}} + |\mathcal{U}_L|} \quad \textit{for } \mathcal{U}_H$$

$$n_i = n_L = n \frac{1}{Y^{\frac{2}{3}} + |\mathcal{U}_L|} \quad \textit{for each user } u_i \textit{ in } \mathcal{U}_L$$

where $Y = \sum_{u_i \in \mathcal{U}_H} \sqrt{\frac{N}{N_i}}$.

Proof. Hybrid-PSALM has mean estimation error bound of

$$\frac{1}{U} \left(\sum_{u_i \in \mathcal{U}_L} \sqrt{\frac{4N_i}{n_i C_i} \log \frac{2}{\Delta}} + \sum_{u_i \in \mathcal{U}_H} \sqrt{\frac{4N}{n_0 C_i} \log \frac{2}{\Delta}} \right) \tag{5.19}$$

Because the query predicate P_Q is independent of the access control predicates P_{AC}, this can be rewritten as

$$\beta \left(\sum_{u_i \in \mathcal{U}_L} \sqrt{\frac{1}{n_i}} + \sum_{u_i \in \mathcal{U}_H} \sqrt{\frac{N}{n_0 N_i}} \right) \tag{5.20}$$

where β is a constant independent of i. Our goal is to find n_0 and n_i to minimize this expression under the constraint that $n_0 + \sum_{u_i \in \mathcal{U}_L} n_i = n$.

We use Lagrange multipliers to find the minimum, by adding the constraint $n = n_0 + \sum_{u_i \in \mathcal{U}_L} n_i$ as multiplier:

$$\begin{aligned} \Phi(X, \lambda) &= \beta \left(\sum_{u_i \in \mathcal{U}_L} \sqrt{\frac{1}{n_i}} + \sum_{u_i \in \mathcal{U}_H} \sqrt{\frac{N}{n_0 N_i}} \right) + \\ & \quad \lambda \left(\sum_{u_i \in \mathcal{U}_L} n_i + n_0 - n \right) \end{aligned}$$

The critical value of Φ occurs when the gradients on n_0, n_i, λ are all zero:

$$\begin{aligned} \frac{\partial \Phi}{\partial n_i} &= \beta \frac{\partial \frac{1}{\sqrt{n_i}}}{\partial n_i} + \lambda = -\frac{\beta}{2} n_i^{-\frac{3}{2}} + \lambda = 0, u_i \in \mathcal{U}_L \\ \frac{\partial \Phi}{\partial n_0} &= -\frac{\beta}{2} \sum_{u_i \in \mathcal{U}_L} \sqrt{\frac{N}{N_i}} n_0^{-\frac{3}{2}} + \lambda = 0 \\ \frac{\partial \Phi}{\partial \lambda} &= \sum_{u_i \in \mathcal{U}_L} n_i + n_0 - n = 0 \end{aligned}$$

We use Y to represent $\left(\sum_{u_i \in \mathcal{U}_H} \sqrt{\frac{N}{N_i}} \right)$. The mean estimation bound reaches its minimum when the following holds:

$$\begin{aligned} \lambda &= \frac{\beta}{2} [\frac{Y^{\frac{2}{3}} + |\mathcal{U}_L|}{n}]^{\frac{3}{2}} \\ n_0 &= \frac{Y^{\frac{2}{3}}}{Y^{\frac{2}{3}} + |\mathcal{U}_L|} n \\ n_i &= \frac{1}{Y^{\frac{2}{3}} + |\mathcal{U}_L|} n, \quad u_i \in \mathcal{U}_L \end{aligned}$$

□

With Theorem 9 in place, we next consider how to partition the users into $\mathcal{U}_H$ and $\mathcal{U}_L$ to minimize the mean estimation error bound. To do this, we use Algorithm 5, which checks all possible partitions having the property that no user in $\mathcal{U}_L$ has access to more tuples than any user in $\mathcal{U}_H$. There are $(U+1)$ such partitions to be checked.

Theorem 10. *The partition returned by Algorithm 5 minimizes the mean estimation error bound $\hat{\epsilon}_{mean}$ over all possible two-way partitions of the users.*

Proof. Using Theorem 9 and Formula 5.20, we can write the mean total estimation error of a given user grouping as:

$$\begin{aligned}
& \beta\left(\sum_{u_i\in\mathcal{U}_L}\sqrt{\frac{Y^{\frac{2}{3}}+|\mathcal{U}_L|}{n}}+\sum_{u_i\in\mathcal{U}_H}\sqrt{\frac{N}{N_i}}\sqrt{\frac{Y^{\frac{2}{3}}+|\mathcal{U}_L|}{nY^{\frac{2}{3}}}}\right)\\
= \;& \beta\sqrt{\frac{Y^{\frac{2}{3}}+|\mathcal{U}_L|}{n}}\left(|\mathcal{U}_L|+\frac{\sum_{u_i\in\mathcal{U}_H}\sqrt{\frac{N}{N_i}}}{\sqrt{Y^{\frac{2}{3}}}}\right)\\
= \;& \beta\sqrt{\frac{Y^{\frac{2}{3}}+|\mathcal{U}_L|}{n}}\left(|\mathcal{U}_L|+\frac{Y}{Y^{\frac{1}{3}}}\right)\\
= \;& \beta\frac{\left(|\mathcal{U}_L|+Y^{\frac{2}{3}}\right)^{\frac{3}{2}}}{\sqrt{n}}
\end{aligned}$$

Because both n and β are constants and the function $F(x)=x^{\frac{3}{2}}$ is monotone, minimizing the above expression is equivalent to minimizing

$$|\mathcal{U}_L|+\left(\sum_{u_i\in\mathcal{U}_H}\sqrt{\frac{N}{N_i}}\right)^{\frac{2}{3}} \tag{5.21}$$

Suppose the user-grouping returned by Algorithm 5 is:

$$\mathcal{U}_L=\{u_1,u_2,\ldots,u_j\},\ \mathcal{U}_H=\{u_{j+1},u_{j+2},\ldots,u_U\}$$

We use E to denote the value of Formula 5.21 for this user-grouping. Now suppose there is another user-grouping as follows:

$$\mathcal{U}'_L=\{u_{p_1},u_{p_2},\ldots,u_{p_l}\},\ \mathcal{U}'_H=\{u_{p_{l+1}},u_{p_{l+2}},\ldots,u_{p_U}\}$$

We use E' to denote the value of Formula 5.21 from this user-grouping. Assume $E'<E$. We will show that this assumption leads to a contradiction.

From Formula 5.21, we have:

$$E = j + \left(\sum_{i \in (j+1,\ldots,U)} \sqrt{\frac{N}{N_i}} \right)^{\frac{2}{3}}$$

$$E' = l + \left(\sum_{i \in (p_{l+1},\ldots,p_U)} \sqrt{\frac{N}{N_i}} \right)^{\frac{2}{3}}$$

Now suppose that we sort all of the users in descending order of the number of tuples to which they have access, and we consider a configuration with $\mathcal{U}_H$ consisting of the first l users, i.e., the users that can access the most tuples. Let E'' represent the value of Formula 5.21 under this configuration:

$$E'' = l + \left(\sum_{i \in (l+1,\ldots,U)} \sqrt{\frac{N}{N_i}} \right)^{\frac{2}{3}}$$

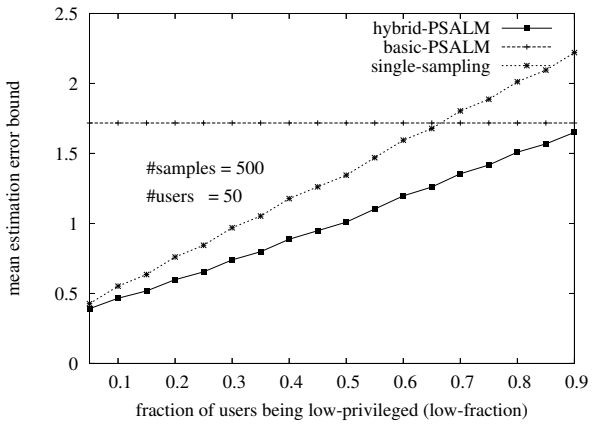
Because $\mathcal{U}_H$ consists of the l users with access to the most tuples, we have $E'' \leq E'$. We also have $E \leq E''$ because both of those configurations are considered by Algorithm 5, which returns the configuration with the lowest error bound among those that it considers. Therefore $E \leq E'$, a contradiction. □

By partitioning the users into $\mathcal{U}_H$ and $\mathcal{U}_L$ according to Algorithm 5 and choosing sample sizes according to Theorem 9, we can minimize the mean estimation error bound for hybrid-PSALM. Because both the single-sample estimation technique and the basic-PSALM technique are special cases of hybrid-PSALM, we have the following corollary:

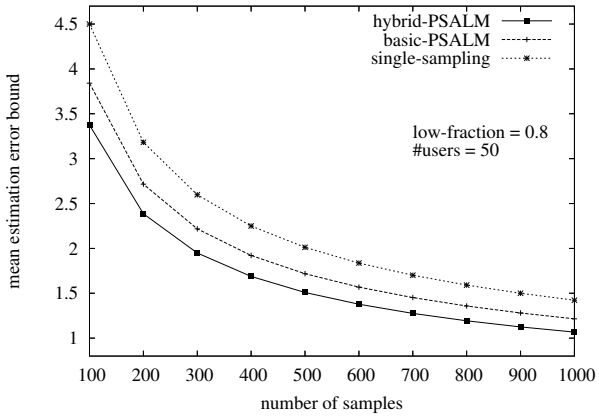
Corollary 1. *Hybrid-PSALM under the user-partitioning from Algorithm 5 and the sample size assignment from Theorem 9 has a mean estimation error bound no greater than the mean estimation error bound of single-sample and no greater than the mean estimation error bound of basic-PSALM.*

5.5.2 Analysis of Mean Estimation Error Bounds

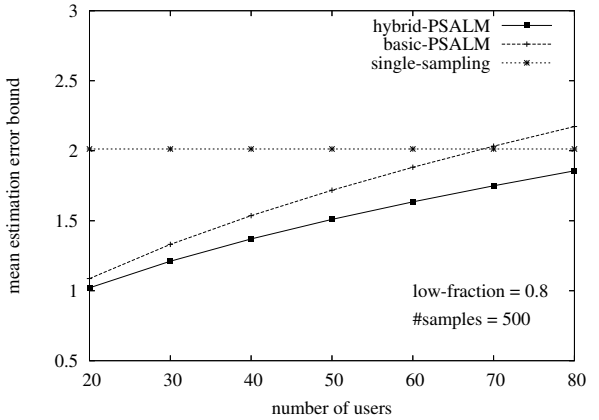
Figure 5.3 illustrates the mean 95%-confidence estimation error bounds for the single-sample, basic-PSALM and hybrid-PSALM approaches under a variety of conditions. These curves are determined by Equations 5.1 and 5.2 for the single-sample and basic-PSALM approaches, and by Formula 5.19 for hybrid-PSALM. To obtain these curves, we varied the total number of users (U)



(a)



(b)



(c)

Figure 5.3: Mean Estimation Error Bounds of hybrid-PSALM, basic-PSALM, and single-sampling

Algorithm 5

input: the number of accessible tuples (N_i) for each user
output: optimal user grouping $\{\mathcal{U}_L, \mathcal{U}_H\}$

1: Sort the users by their number of accessible tuples
   in ascending order into list $\{u_i \; i \in (1, \ldots, U)\}$;
2: $min = \infty$, $G_{opt} = \emptyset$;
3: **for** $p = 0$ to U
4: **do** $\mathcal{U}_L = \{u_j, j \in (1, \ldots, p-1)\}$;
   $\mathcal{U}_H = \{u_j, j \in (p, \ldots, U)\}$;
5: $val = p + \left(\sum_{u_j \in \mathcal{U}_H} \sqrt{\frac{N}{N_j}} \right)^{\frac{2}{3}}$
6: **if** $(val < min)$
7: **then** $min = val$;
   $G_{opt} = \{\mathcal{U}_L, \mathcal{U}_H\}$;
8: **return** G_{opt};

and the total sample size (n) as shown in the Figure. We chose the number of tuples accessible to each user (N_i) by choosing some percentage of users as high-privileged users ($\mathcal{U}_H$), and the remaining users as low-privileged users ($\mathcal{U}_L$). The percentage of high-privileged users is denoted as *high-fraction* in the experiments. Each high-privileged user has 50% of the data accessible to him while each low-privileged user has 1% of the data accessible to him. Finally, we fixed the selectivity of the query predicate P_Q at 50%, and assumed independence between P_Q and the users' access control predicates. Varying the selectivity of P_Q rescales the reported mean estimation error bounds, but does not affect the shape of the curves.

Figure 5.3(a) shows the behavior of the sampling techniques as the percentage of low-privileged users varies. The basic-PSALM technique is insensitive to this parameter, while the performance of the single-sample approach gets better as we tend towards more high-privilege users and fewer low-privilege users. This is because the single-sample technique is more effective for high-privilege users than for low-privilege users. The hybrid-PSALM approach outperforms both of the simpler alternatives. Figure 5.3(b) shows that all of the techniques are able to exploit an increasing sample space budget to improve accuracy. Finally, Figure 5.3(c) shows estimation error as a function of the number of users, with the sample budget fixed at 500 tuples. The single-sample technique is relatively insensitive to the number of users, while both PSALM techniques do best when the number of users is small. The basic-PSALM and single-sample techniques have

a crossover point, as shown in Theorem 7. However, the hybrid-PSALM technique performs at least as well as either simpler technique regardless of the number of users.

5.5.3 Simulation Results

We used simulation analysis to study the actual mean estimation error (ϵ) of hybrid-PSALM and multi-dimensional histograms. Our experiments use synthetic data created using an approach similar to that of Bruno, Chaudhuri and Gravano in their evaluation of STHoles multidimentional histograms [18]. We first create 500,000 tuples from a data domain of $[0, 1000)^d$, where d is the dimension, i.e., number of attributes in these tuples. We experiment with different d ranging from one to eight. These tuples are distributed among 100 clusters in the data domain. This is done by randomly creating 100 evenly distributed tuples as the center (mean) of each cluster, and then for each cluster, randomly creating tuples from its mean with perturbation such that the tuples in each cluster form a multidimensional Gaussian distribution with standard deviation 25. The number of tuples contained in each cluster follows a zipfian distribution with parameter 1.0.

Our workload consists of pairs of access control predicates and query predicates. These predicates are generated as multi-dimensional range queries using an approach similar to that of Pagel et al. [72]. These multi-dimensional range predicates are applied on randomly chosen attributes of the data, where each range is a window around a randomly chosen spot in the data domain. In our case, the width of the window is chosen to be $(100 * d)$. The reason we choose larger window for larger data dimension is that as the dimension goes up, the number of conjuncts in a predicate increases, and we wish the predicate's selectivity remains non-zero. We create predicates using three types of range queries: type A having their mean follow the same distribution as the data, type B having their query centers following uniform distribution in the data domain, and type C having their query centers following a Gaussian distribution independent of the data distribution. Similar predicates have been used by Bruno, Chaudhuri and Gravano [18]. We can use different types of range queries for P_Q and P_{AC}. We only show the result of using type A for P_{AC} since other combinations show similar trends.

The multidimensional histogram is based on an equi-width grid-based implementation similar to the work by Aboulnaga and Chaudhuri [4]. We create histograms with different numbers of dimensions, under the same space budget as hybrid-PSALM. For example, if we use a total sample space budget of 560 bytes, we can have a sample-based approach with a total of 70 sample tuples, with each tuple having two attributes of type *float* (4 bytes); or an $11 * 11$ 2D histogram, which takes 121 float values for the frequency values and 20 float values for the grid rulers; or a $5 * 5 * 5$ 3D histogram. In our experiment we choose the total space budget as being 1000 tuples each having d attributes.

For each experiment, we choose a value for d and the number of dimensions in the query and access control predicates. We use $D4Q2$ to denote test on data with four attributes, and P_{AC}

and P_Q each is a conjunction of range queries on two randomly chosen attributes, the histogram is a 4D histogram under the same space budget of 1000 tuples from the test data. Similarly, $D8Q4$ means test on data with eight attributes, with P_{AC} and P_Q each as a conjunction of range queries on four randomly chosen attributes. We generated random pairs of query and access control predicates. For each pair, we determined the actual cardinality of $P_Q \wedge P_{ACi}$ as well as the estimated cardinality produced by hybrid-PSALM and the multi-dimensional histogram. We then calculated the mean estimation error (ϵ) for both technique.

Figure 5.4 shows a comparison of the mean estimation error of the two techniques using data and queries with different number of attributes, using type A for $P_Q \wedge P_{ACi}$ We can see that when the data dimension is low, the multi-dimension histogram (or the single dimension histogram as in $D1Q1$) has mean estimation error close to or lower than hybrid-PSALM. However, as soon as dimension goes to four, hybrid-PSALM has much lower mean estimation error than multi-dimensional histogram. In all of these experiments, hybrid-PSALM chooses different user partitions ranging from 0% users in $\mathcal{U}_L$ to 100% users in $\mathcal{U}_L$, with an average of 90% users in $\mathcal{U}_L$.

5.6 Exploiting Access Privilege Correlation

Data in Chapter 3 shows strong correlations in access privileges among different users. In this section, we explore an enhancement to the hybrid-PSALM technique that exploits such correlations.

Suppose that we use the hybrid-PSALM algorithm to partition users into $\mathcal{U}_L$ and $\mathcal{U}_H$ and to assign sample sizes n_L for users in $\mathcal{U}_L$ and n_0 for users in $\mathcal{U}_H$, as described in Section 5.5. The enhancement to be introduced in this section will improve the accuracy of cardinality estimates by exploiting access privilege correlations among the users in $\mathcal{U}_L$. The approach is to identify groups of low-privilege users that have correlated access privileges, using a technique that will be described shortly. Suppose that $G \subseteq \mathcal{U}_L$ is such a group. Instead of creating a separate, private sample for each user in G, we create a single, combined sample of size $|G|n_L$ that is shared by all users in G. Here n_L is the private sample size for the users in $\mathcal{U}_L$, and is a consistent value for all such users in G according to Theorem 9. This new shared sample is a random sample drawn from among all tuples that satisfy P_{ACG}, defined as

$$P_{ACG} = \bigvee_{i \in G} P_{ACi}$$

That is, we sample from among those tuples that are accessible to at least one user in the group. To compute an unbiased cardinality estimate for predicate P_Q for user $u_i \in G$, we use

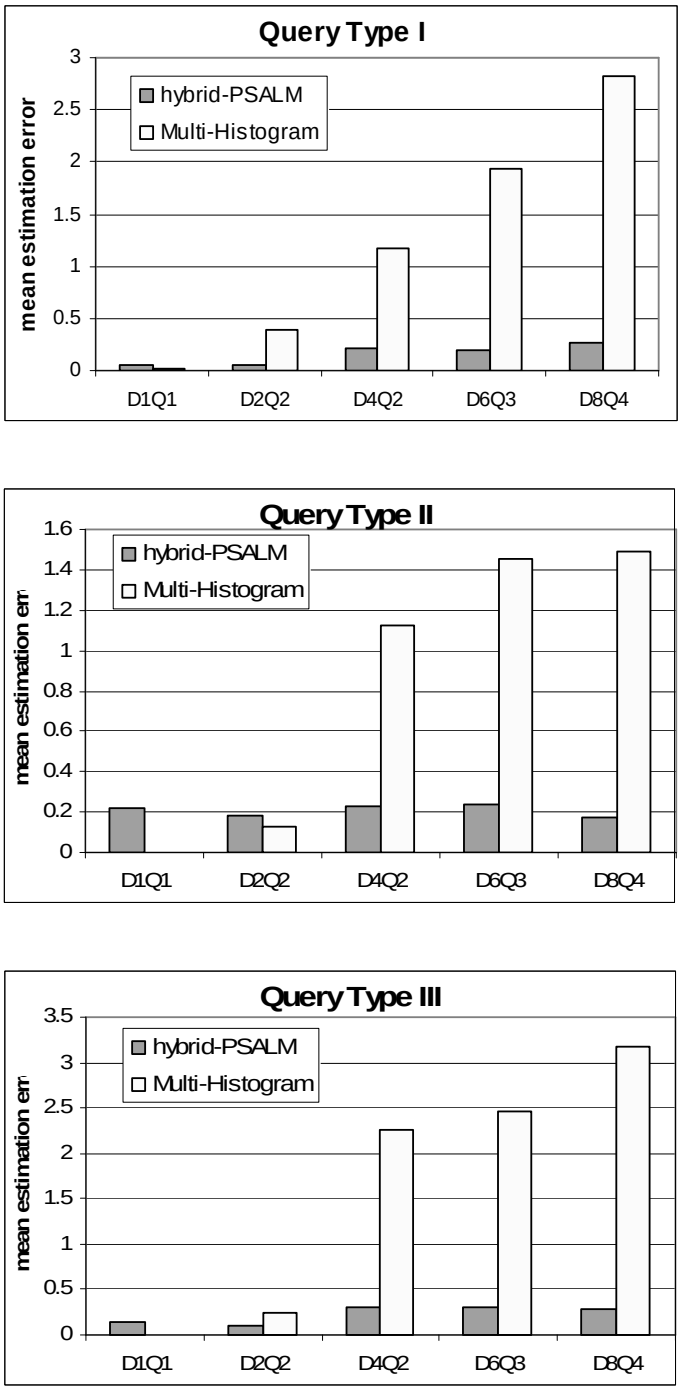


Figure 5.4: Mean estimation error between hybrid-PSALM, and multidimensional histogram

$$\tilde{C}_i = c_i \frac{N_G}{|G| n_L} \tag{5.22}$$

where c_i is the number of sample tuples matching $P_Q \wedge P_{ACi}$ and N_G is the number of tuples in the target relation that satisfy P_{ACG}. This results in a Δ-confidence estimation error bound of

$$\epsilon_i \leq \sqrt{\frac{4N_G}{|G| n_L C_i} \log \frac{2}{\Delta}} \tag{5.23}$$

for all users in G.

Grouping users in this way does not change the cardinality estimates of users outside the group, nor does it affect the accuracy of those estimates. Thus, any change in the total estimation error bound from all users comes from the change of the estimation error bound of the users in G. By combining Equations 5.2 and 5.23, we can see that user grouping will result in lower mean estimation error bound among the users in the group if

$$\sum_{i \in G} \sqrt{\frac{4N_G}{|G| n_L C_i} \log \frac{2}{\Delta}} < \sum_{i \in G} \sqrt{\frac{4N_i}{n_i C_i} \log \frac{2}{\Delta}}$$

Simplifying, we find that this inequality holds provided that

$$\sum_{i \in G} \sqrt{\frac{N_G}{|G| C_i}} < \sum_{i \in G} \sqrt{\frac{N_i}{C_i}}$$

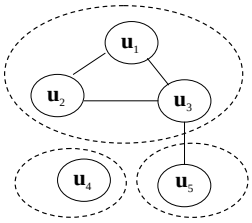
When there is little overlap among the accessible tuples of the grouped users, then N_G approaches $\sum_{i \in G} N_i$ and there is no benefit to grouping users to reduce the total estimation error bound or the maximum estimation error bound. However, if the grouped users have most of their accessible tuples in common, then N_G is much smaller and grouping will result in improved estimates for all users in the group because of the larger size of the group's sample. Thus, to form groups of users we should identify users that have correlated access rights and group them together. In the following, we describe how we group the users together.

5.6.1 Grouping Correlated Users

To determine which users to group together, we first define a pairwise similarity function, *SIM*, over the users in $\mathcal{U}_L$. The similarity between the i^{th} and j^{th} users is defined as follows:

$$SIM(i,j) = \min\left(\frac{N_{i,j}}{N_i}, \frac{N_{i,j}}{N_j}\right) \tag{5.24}$$

	u_1	u_2	u_3	u_4	u_5
t_1	1	1	1	0	0
t_2	1	1	1	0	0
t_3	0	0	1	0	1
t_4	0	0	1	1	1



Min cluster partition with AC similarity thresh-hold 0.45

Figure 5.5: An example of user grouping

Here $N_{i,j}$ is defined as the number of tuples from the target relation that are accessible to both user i and user j. This similarity function has property similar to other set similarity functions like the Jaccard Coefficient [1],i.e., it has a value between 0 and 1, and a value close to 1 indicates a strong similarity while a value of 0 indicates no similarity.

Using this function *SIM*, we can define a *similarity graph* as an undirected graph with one node for each user in $\mathcal{U}_L$. There is an edge between user i and user j if and only if $SIM(i, j) \geq \theta$, where θ is a tunable parameter of the grouping algorithm. We will describe how to pick θ in Section 5.6.2.

To place the users into groups, we attempt to find a *minimum clique partition* [26] of the user similarity graph. This identifies a set of non-overlapping cliques that, together, cover the entire user similarity graph. Each such clique becomes one of the user groups for which we will create a sample. By using cliques as sampling groups, we ensure that all users in a group have pairwise similar access rights. By minimizing the number of cliques we minimize the number of separate samples that are required, thus allowing us to use larger samples while remaining within the space budget.

The example in Figure 5.5 illustrates the process of identifying user groups. The example illustrates a scenario in which there are five users ($u_1, u_2, \ldots, u_5$) and eight tuples ($t_1, t_2, \ldots, t_4$). The large matrix indicates which tuples are accessible to each user, with a 1 indicating accessibility. Using a similarity threshold θ of 0.5, we obtain the user similarity graph with pairwise connections among u_1, u_2 and u_3, as shown in Figure 5.5. A minimum clique partition for this graph is illustrated using dashed lines.

The problem of finding a minimum clique partition is NP-complete [26]. Thus, we use a fast greedy algorithm to partition the graph. The algorithm produces a set of non-overlapping cliques that cover the user similarity graph, but the set is not guaranteed to have minimum cardinality. The greedy algorithm first finds a maximal clique in the graph by starting from a random node, finding the clique around the node, and removing all nodes in the clique and all edges induced by these nodes. This process continues until there are no more nodes. We repeat this greedy algorithm several times from different initial nodes and record the smallest clique partition that

is found. Recall that hybrid-PSALM already partitions the users into $\mathcal{U}_H$ and $\mathcal{U}_L$, so the users to apply this greedy algorithm do not include the high privileged users. This is one reason why we want to apply the hybrid-PSALM technique first before grouping users with similar access controls.

5.6.2 Choosing θ for Grouping Users

Although grouping correlated users reduces the mean estimation error bound, some users may incur significant loss in estimation accuracy. For example, consider two users u_i and u_j from $\mathcal{U}_L$. Suppose each of them would have a private sample size of k tuples without correlation-based grouping. If u_i and u_j are grouped based on correlation, they have a total sample size of $2k$ tuples. If we randomly sample $2k$ tuples from the union of their accessible data, user u_i will have the following expected number of tuples from his accessible data included in the sample:

$$2k \cdot \frac{N_i}{N_i + N_j - N_{i,j}}$$

If N_j is much larger than N_i, we can see from the above formula that the expected number of sample tuples for u_i will be less than his original sample quota k. This means u_i will more likely to have a higher estimation error bound after grouping with u_j. Fortunately, hybrid-PSALM already partitions users into $\mathcal{U}_H$ and $\mathcal{U}_L$ so that the number of accessible tuples among users in $\mathcal{U}_L$ will be not as skewed as it is among users in general. This is another reason that we need to apply the hybrid-PSALM technique before grouping users with correlated access controls.

We have the following lemma for grouping user u_i and u_j together:

Lemma 5.6.1. *Both user u_i and user u_j will have lower estimation error bound after grouping if the following holds, where $N_{i,j}$ denotes the tuples in the database that are accessible to both user u_i and user u_j.*

$$N_{i,j} > |N_i - N_j|$$

Proof. The number of tuples accessible to both user u_i and u_j is $(N_i + N_j - N_{i,j})$. The number of tuples in each of these two user's private samples is n_L, and their merged private sample size will be $2 \cdot n_L$. From Formula 5.1, both user u_i and u_j will have a lower cardinality estimation bound if the following two formulae hold:

$$\sqrt{\frac{4(N_i + N_j - N_{i,j})}{2n_L \cdot C_i} \log \frac{2}{\Delta}} < \sqrt{\frac{4N_i}{n_L \cdot C_i} \log \frac{2}{\Delta}}$$

$$\sqrt{\frac{4(N_i + N_j - N_{i,j})}{2n_L \cdot C_j} \log \frac{2}{\Delta}} < \sqrt{\frac{4N_j}{n_L \cdot C_i} \log \frac{2}{\Delta}}$$

The above is equivalent to

$$N_j - N_i < N_{i,j}$$

$$N_i - N_j < N_{i,j}$$

From the above we have the proof. □

We can extend the above lemma to the case of grouping more than two users. We have the following theorem based on the access control similarity threshhold θ computed from Formula 5.24.

Theorem 11. *Suppose we have users $\{u_1, u_2, \ldots, u_{|G|}\}$ such that they have access controls with pairwise similarity (as defined in Formula 5.24) above a constant θ. Then everyone of these users will have a lower cardinality estimation error bound after merging all their sample quotas if the following hold for every user $u_i, 1 \leq i \leq |G|$:*

$$\theta > \frac{\sum_{j=1}^{|G|} N_j - |G| \cdot N_i}{\sum_{j=1}^{|G|} N_j - N_i} \tag{5.25}$$

Proof. Let's look at an arbitrary user, say u_i. If another user u_j has access controls that are $SIM(i,j)$ similar to u_i, then the size of the union of their accessible tuples will be bounded by

$$\begin{aligned} & N_i + (1 - SIM(i,j)) \cdot N_j \\ = \; & SIM(i,j) \cdot N_i + (1 - SIM(i,j))(N_i + N_j) \end{aligned}$$

Therefore, if user u_i is to be merged with users $\{u_1, u_2, \ldots, u_{i-1}, u_{i+1}, \ldots, u_{|G|}\}$, and suppose the similarity function value between user u_i and all these users are greater than a constant θ, then the union of user u_i and these users' accessible tuples will be bounded by

$$\theta \cdot N_i + (1 - \theta) \cdot \sum_{j=1}^{|G|} N_j$$

In order to ensure that user u_i has a lower estimation error bound after merging his sample quota with all other users', the following must hold:

$$\sqrt{\frac{4(\theta \cdot N_i + (1-\theta) \cdot \sum_{j=1}^{|G|} N_j)}{|G| n_L \cdot C_i} \log \frac{2}{\Delta}} < \sqrt{\frac{4N_i}{n_L \cdot C_i} \log \frac{2}{\Delta}}$$

Solving the above, we have the proof.

□

To make sure that no user will have a higher estimation error bound after grouping than before grouping, we need to verify that Formula 5.25 satisfies for every user in each clique when performing the greedy clique-partitioning algorithm as described in Section 5.6.1. At the end of every iteration, we check if this formula is violated for some users in a clique, and remove those users from that clique. We keep iterating until this formula is satisfied for every user in each clique.

5.6.3 Cost of Exploiting Correlation

Once user groups have been defined and the corresponding samples have been drawn, the cost of estimating query cardinalities is essentially the same whether low-privilege users are grouped or not. In either case, a predicate is evaluated against each tuple in the appropriate sample and an estimate is computed using either Formula 5.17 or Formula 5.22. Similarly, all of the necessary samples can be drawn in a single pass over the target relation, regardless of whether grouping is used.

Since grouping depends only the access rights of the various users, and not on the query predicates, user groups will not need to change unless access privileges are redefined. We assume that this will occur rarely with respect to query evaluation.

There is an additional cost associated with determining how users should be grouped together. To group users according to Formula 5.24, it is necessary to obtain $\{N_{i,j}\}$ for all pairs of users in $\mathcal{U}_L$ to compute $SIM(i,j)$. This can be piggy-backed during the same scan on the target relation to get $\{N_i\}$ before running Algorithm 5, if we maintain a $U * U$ integer array for all pairs of users. After getting $\{N_{i,j}\}$ and computing the correlated user groups, we need one more scan on the target relation to get N_G for each correlated user group G. During that scan we can do a reservoir sampling for each correlated user group. Thus, exploiting access privilege correlations increases the cost of one scan on each target relation to two scans.

5.6.4 Effectiveness of User-Grouping

As a test of the procedure for forming user groups, we applied the procedure to the access control data taken from the LiveLink multi-user content management system as described in Section 3.6.

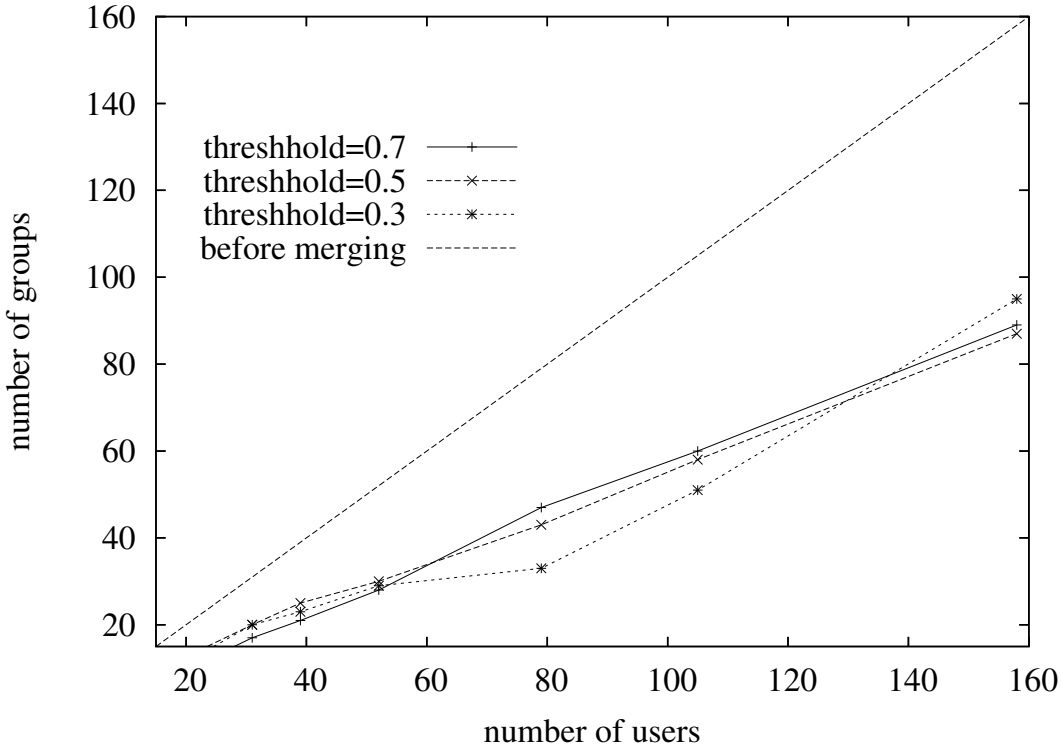


Figure 5.6: Effect of grouping on the number of samples required for low-privilege users

The system had approximately 370000 access-controlled objects and a total of 1584 users. We experimented with different similarity thresholds θ ranging from 0.3 to 0.7. We randomly selected sets of users from among all 1584 users, and we applied our user grouping algorithm to each selected set of users.

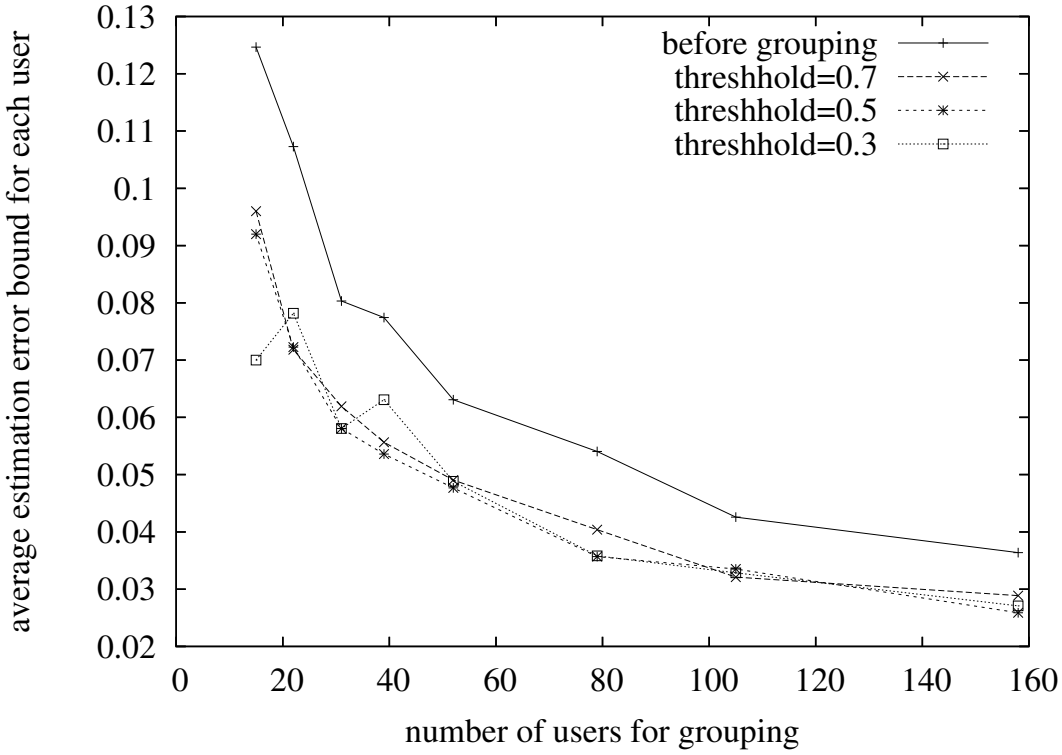


Figure 5.7: Average estimation error bound after grouping users with similar access controls

Figure 5.6 shows the number of user-groups (or private samples) required after merging, with different threshholds θ. We found that the user grouping algorithm effectively reduces the number of samples. However, the effect of θ is not that significant. The reason is that while a lower θ value introduces larger cliques and results in fewer groups, many users in the cliques may fail Formula 5.25, and have to be discarded.

Figure 5.7 shows the improved mean estimation error bound after grouping users in $\mathcal{U}_L$ under different threshholds θ. In this figure, we fix the private sample size for each user, and the total sample size grows linearly to the number of users for grouping. The more users for grouping, the higher chances that users with similar access controls can merge their sample quota and share a larger private group sample. The more users in each group, the larger their group's private sample

size after merging, and the lower these users' new estimation error bounds. While user-grouping lowers the average estimation error bound, the effect of θ is again not significant.

5.7 Cardinality Estimation Under Role-based Access Control

For the previous discussion we have assumed that the fine-grained access controls are explicitly specified for each user in the system. However, as described in Section 2.2.1, fine-grained access controls are usually specified through *raw access controls* define on several *roles*, and the raw access controls propagate to each of the users. In this section, we show how to use PSALM to estimate cardinality under such role-based access control mechanisms.

Suppose a user is authorized with k roles directly or indirectly from the role hierarchy, and each role R_i has access control predicate P_{ACi} on a target relation I. The user's access control predicate on the target relation is thus $(\bigvee_{i\in\{1,\ldots,k\}} P_{ACi})$. Given query selection predicate P_Q, we are trying to estimate the cardinality of the set of tuples matching $P_Q \wedge (\bigvee_{i\in\{1,\ldots,k\}} AC_i)$ from the target relation.

A straightforward approach to compute the estimated cardinality is to compute the *effective access controls* for each user from all his roles, and apply PSALM sampling scheme for all the users. However, if there are many users with distinct access controls, the private samples may become extremely small. On the other hand, the users have different effective access controls simply because they are granted *different combinations* of roles. If we build PSALM samples based on roles, we will have fewer, larger private samples. Therefore, our goal is to accurately estimate the number of accessible tuples satisfying a given query predicate by applying the PSALM sampling scheme to the roles.

We first apply the distributive law to rewrite the access control predicate:

$$P_Q \wedge (\bigvee_{i\in\{1,\ldots,k\}} P_{ACi}) \equiv \bigvee_{i\in\{1,\ldots,k\}} (P_Q \wedge P_{ACi}) \tag{5.26}$$

We are able to apply PSALM sampling on the roles to estimate the cardinality of each $P_Q \wedge P_{ACi}(I)$ accurately. However, to the best of our knowledge, there is no effective approach to directly estimate the cardinality of disjunctive predicates on a relation. One approach used in PostgreSQL is to estimate cardinality of disjuncts by rewriting them into a conjunctive form. For example, the cardinality of the disjunctive predicate $(P_1 \vee P_2 \vee P_3)$ on relation I can be estimated as follows:

$$
\begin{aligned}
& |(P_1 \vee P_2 \vee P_3)(I)| \\
= \; & |P_1(I)| + |P_2(I)| + |P_3(I)| - \\
& \quad |(P_1 \wedge P_2)(I)| - |(P_1 \wedge P_3)(I)| - |(P_2 \wedge P_3)(I)| + \\
& \quad\quad |(P_1 \wedge P_2 \wedge P_3)(I)|
\end{aligned}
$$

where the cardinality of each conjunct is estimated separately. However, one significant disadvantage of this approach is that the number of terms in the formula is exponential in the size of the disjunctions. In the next section, we present an alternative approach that solves this problem.

5.7.1 Coverage Algorithm for Estimating Size of Union of Sets

We use the Coverage Algorithm [67] to estimate cardinality of disjunctive predicates. This approach takes time polynomial in the number of conjunctions in a disjunctive formulae, has more accurate estimates than the approach used in PostgreSQL, and can be built on top of PSALM. The Coverage Algorithm is able to accurately estimate the size of union $\bigcup_{i \in (1,\ldots,k)} C_i$, provided that [67]:

1. we can accurately estimate the cardinality of each set C_i, and
2. we can sample uniformly at random from each C_i, and
3. we can determine in polynomial time whether a given tuple belongs to C_i.

The Coverage Algorithm takes as input the sets $\mathcal{C} = \{C_i, i \in \{1, \ldots, k\}\}$ and proceeds as in Algorithm 6. The algorithm starts with a counter W with value zero, and a total sample size n. It then samples from each of the sets C_i, where the number of samples taken from C_i is proportional to $|C_i|$. For each sample from C_i, we determine whether it belongs to any C_j such that $j \in \{1, \ldots, i-1\}$. If it does *not* belong to any of these sets, we increment the counter W by one.

After checking all the samples from $C_1, C_2 \ldots, C_k$, we take the counter W from this routine and compute estimated cardinality $|C| = |\bigcup_{i \in (1,\ldots,k)} C_i|$ as follows:

$$|C| = \sum_{i \in \{1,\ldots,k\}} |C_i| \frac{W}{N} \tag{5.27}$$

Theorem 12. *[67] Algorithm 6 and Formula 5.27 yield an ϵ-approximation to $|C|$ with $(1 - \Delta)$ probability, provided sample size $n \geq \frac{4k}{\epsilon^2} \ln \frac{2}{\Delta}$.*

Algorithm 6 Coverage Algorithm

[67]

COVERAGE$(\mathcal{C}, n)$

1: set counter $W = 0$;
2: sort sets in $\mathcal{C}$ by their sizes in descending order
3: **for** each $C_i \in \mathcal{C}$
4: **do** sample $\frac{n|C_i|}{\sum_{C \in \mathcal{C}} |C|}$ tuples uniformly from C_i
5: **for** each tuple t sampled
6: **do if** $t \notin C_j, j \in \{1, \ldots, i-1\}$
7: $W++$;
8: **return** W;

5.7.2 Applying the Coverage Algorithm

We now show how to apply the Coverage Algorithm for estimating the cardinality of disjunctive predicates in the form shown in Formula 5.26. Given a target relation T, we need to be able to do the following to apply the Coverage Algorithm:

1. estimate the size of each $P_Q \wedge P_{ACi}(T)$, and
2. sample uniformly from each $P_Q \wedge P_{ACi}(T)$, and
3. decide whether a given tuple satisfies $P_Q \wedge P_{ACi}$ in polynomial time.

The first requirement is already met by $PSALM$. The third requirement can be met if the predicate does not involve exponential computation. For access control predicates, this requirement is easily satisfied. For the second requirement, we can satisfy it by sampling from role R_i's already materialized PSALM private sample tuples. We can accomplish this by reservoir-sampling on the PSALM private samples of role R_i that satisfy P_Q, i.e., we sample from role R_i's PSALM private sample on the fly, discard all tuples that do not satisfy P_Q, and feed the qualifying sampled tuples into the reservoir sample. We have the following Lemma:

Lemma 5.7.1. *Reservoir-sampling on the (already materialized) PSALM private sample tuples of role R_i that satisfy P_Q generates a uniform sampling of $P_Q \wedge P_{ACi}$.*

Proof. We use D_i to denote the reservoir from role R_i. We first have that each tuple in $P_Q(D_i)$ is also in $P_Q \wedge P_{ACi}$. We also know that a tuple in $P_Q \wedge P_{ACi}$ has probability $\frac{|P_Q \wedge P_{ACi}|}{N_i}$ to appear in D_i, and this probability is independent of other tuples in $P_Q \wedge P_{ACi}$. Hence the proof. □

5.7.3 Evaluating the Coverage Algorithm

We evaluated the Coverage Algorithm on top of the PSALM technique using a data set that describes American household expenditures [1]. This data set consists of 127931 tuples, with each tuple representing a family's expenses on insurance, property tax, electricity, gas, water, and fuel (altogether six attributes). There is some correlation among the different attributes.

We simulate users' access controls by randomly creating four roles. The access control of each role is a single predicate P_{AC} on one of the six attributes of the data. We then randomly assign these four roles to users. Each user may have one to four roles, and the user's access control predicate is a disjunct over the single attribute predicates from his roles. Therefore, there are altogether 15 distinct access controls, which represent all our users. All of the disjunctions that we considered in this experiment had an actual selectivity of approximately 0.25, meaning that each user has access to approximately one quarter of the tuples. The reason for this selectivity will be clear later.

Our workload for this experiment consists of one-dimensional range queries with the same selectivity (0.25) as the access controls. We generated a series of such queries and, for each pair of query and user's access control predicate, calculated both the actual result cardinality and the estimated cardinality. We compared two approaches for cardinality estimation. The first is the PSALM combined with the Coverage Algorithm, as described previously. This approach uses PSALM to generate cardinality estimates for each of the four roles, and then combines those estimates using the coverage algorithm to obtain cardinality estimates for user queries.

The second approach is to ignore the disjunctive structure of the access controls. Instead we treat each user's disjunctive predicate as an atomic predicate, and we apply PSALM directly on these 15 users to estimate the cardinality of that predicate.

Figure 5.8 shows the estimation error of the two estimation techniques, assuming that both techniques are given a sample space budget of 60 tuples. Each point in the figure represents a query from a user. The x-coordinate denotes the estimation error using PSALM directly on all users and the y-coordinate denotes the estimation error using the Coverage Algorithm on top of PSALM on the roles. In most cases, the estimates produced by the Coverage Algorithm are significantly more accurate than those produced by PSALM directly on users.

5.8 Comparison to Related Work

There exists a broad literature on cardinality estimation techniques for conjunctive predicates without the attribute value independence assumption. Among them, multi-dimensional histograms [4, 18, 58, 75], wavelet-based techniques [65] and sampling [40, 60, 56] are the three

[1] available from http://www.ipums.org

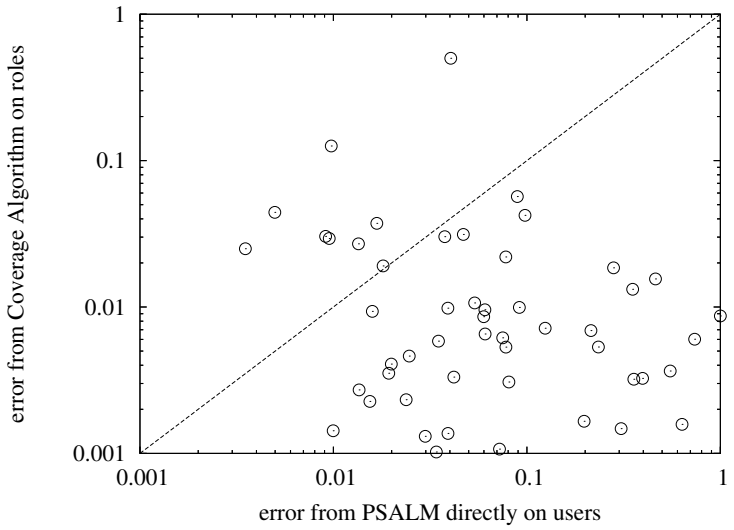


Figure 5.8: Accuracy of two techniques for estimating the cardinality of disjunctive predicates

most common techniques. We focus on sampling-based techniques in this chapter, because it not only accommodates SQL predicates but also predicates based on user-defined functions, as in the case of label-based access controls [45, 66, 69].

There are two main approaches to sampling data for cardinality estimation. One is to sample for each query at runtime, either proactively before executing the query [32], or reactively after executing the query [5]. The other approach is to sample off-line without prior knowledge of the queries. Although PSALM takes advantage of prior knowledge access controls, it is closer to the off-line sampling category since it does not make any assumptions about the users' queries.

Kolmogorov's statistics show that a moderately-sized sample gives accurate selectivity estimation for queries, and that the required sample size does not depend on the size of the underlying dataset [33]. However, that evaluation is based on the relative error of selectivity, rather than the relative error of cardinality. It is well-known that uniform random sampling does not provide accurate cardinality estimation when the data distribution is highly skewed or the query results are very small. Therefore, instead of specifying a fixed sample size for all queries, adaptive sampling [60] (also known as sequential sampling [40]) iteratively samples from data until the accuracy of the estimation satisfies a stopping rule. This approach falls into the runtime sampling category.

Another approach for handling skewed data or highly selective queries is to sample without uniformity. This includes biased sampling [73] or stratified sampling [21], whose idea is to partition data into non-overlapping clusters of different density, and then assign different weights

to the sample tuples from clusters of different density. The density distributions are either estimated through a pilot sampling phase, or from prior knowledge of the queries together with some statistics on the data.

The work by Acharya, Gibbons and Poosala [6] is similar to our approach. Their sampling mechanism is intended for efficient approximate answering of aggregation queries, and their approach is to partition data according to the prior knowledge of grouping attributes, and judiciously assign sample quota among all the partitions to minimize estimation variance. However, their approach, like those of other biased or stratified sampling techniques, assumes that the data are partitioned into non-overlapping sets for sampling. In our setting, the set of tuples accessible to different users may overlap.

Some recent work proposes the use of statistics on views [37] as well as *sample views* [56], which consist of a sample of tuples drawn from a particular view. The set of tuples accessible to a given user from a given target relation can be thought of as a user-specific view defined by the user's access control predicate. From this perspective, the samples drawn by the PSALM technique are sample views, and could potentially be exploited by the mechanism described by Larson et al. [56]. However, their work does not consider the problem of determining *which views* to define. The view definitions assumed to be given, and the emphasis is on how to maintain and exploit the sample views. In contrast, PSALM is about choosing which views to define, given some information about static predicates (the access control predicates) that appear repeatedly in many queries.

Chapter 6

Conclusion

6.1 Summary and Contributions

This thesis addresses three problems of query evaluation in the presence of fine-grained access controls: physical storage of access control specifications, query optimization for access controlled queries, and cardinality estimation for access controlled queries. These three issues mark the major distinctions between normal query evaluation and access controlled query evaluation. This thesis presents novel solutions for each of these problems.

Fine-grained access controls specify accessibility at a fine data granularity. The access control specifications can be gigantic and therefore call for a compact representation. On the other hand, each piece of data needs to be checked for accessibility during query processing, and that requires an efficient access control lookup mechanism. This thesis presents a physical storage scheme that meets these two requirements at the same time:

1. We first noticed from real-world applications that the access controls show strong structural locality among database for each user. Based on that observation, we designed a compact encoding for each user's access controls. This encoding can be embedded inside the original data.

   We also observed that there is strong correlation between different users' access controls. We encoded the access controls of all users on one piece of data as a bit vector. We found that the number of distinct bit vectors across all pieces of data is relatively small. This observation suggests that we use a code-book to store all distinct access control bit vectors among all users, and replace the access control encodings of all users with pointers to the entries in the code-book. This further compresses the access control data.

2. Since the code-book is small, we can fit it in memory. This makes accessibility checks

efficient during query processing. Other access control data that are embedded in the original data fit nicely with an existing query evaluation framework. This framework, called NoK, is able to piggy-back access control data while doing query evaluation. Therefore, the disk-IO required to retrieve access control data is reduced.

Fine-grained access controls require that each user's query be evaluated against his accessible data. In a multi-user system, the same original query will have many different access controlled versions based on the users' access controls. If the database system computes query plans for all of these access controlled queries from scratch, the query optimization cost can be significant. This thesis presents an intelligent query caching mechanism that reduces the amortized query optimization cost for access controlled queries. This caching mechanism is based on the observation that the user-specific access controlled queries may share a significant common structure in their optimal query plans. The optimizer could cache query plans computed from earlier user-specific queries and reuse the upper-level portion of the query plans for later queries. By reusing the upper-level portion of the query plans, which is the most time-consuming portion to compute, the system reduces query optimization cost.

There are two concerns when reusing the cached query plans for the new access controlled queries:

1. First, the system needs to know whether a cached query plan is suitable to make a correct query plan for a new query. Our query plan caching mechanism addresses this problem by not only caching the query plans, but also caching the query rewrites of the corresponding queries. When a new query comes, the optimizer rewrites the query and compares the rewritten query with the cached query rewrites. Only when two query rewrites agree for their upper level part (i.e., all excluding the leave operators) the optimizer will reuse the corresponding cached query plan. This step guarantees the correctness of the new query plan.

2. Second, the new query plan built from a cached query plan must be an efficient query plan for the new query. The thesis presents a plan matching algorithm that is based on the characteristics of the access paths of the new query and the characteristics of the access paths of the cached query plans. The thesis demonstrates that if the two sets of characteristics match, the query plan built on the cached query plan has performance close or equals to the optimal query plan.

Cardinality estimation is for estimating the size of intermediate results during query processing. The accuracy of the estimate is crucial for query optimization since an inaccurate estimate may result in a query plan that runs much slower than the optimal query plan. To accurately estimate the cardinality of the intermediate query results, the system maintains certain synopses that

capture the statistic distribution of the data. These synopses are usually in the form of histograms or samples.

Under fine-grained access controls, the optimizer needs to estimate the joint selectivity of the original query predicates and the access control predicates of each user. This thesis shows that the cardinality estimation from one single uniform sample will not be accurate for those users having small amounts of accessible data. On the other hand, cardinality estimation from many individual samples with one sample for each user will not be more accurate than a single uniform sample under the same total sample size. To minimize the estimation error for all users, the thesis proposes a novel sampling mechanism that is guaranteed to provide estimates at least as accurate as those provided by simple technologies, such as a single uniform sample. Moreover, we show this sampling scheme is better than multi-dimensional histograms when the query predicates and access control predicates refer to a large number of attributes.

6.2 Future Work

The three contributions presented in this thesis are beneficial in the presence of fine-grained access controls: the compact representation of access control specifications alleviates the hefty storage requirement; the POM query plan caching mechanism reduces the amortized query optimization cost from all access control customized queries, and the PSALM sampling technique results in accurate cardinality estimation for high-quality query planning.

However, a substantial amount of work is still required to build a database system that fully supports fine-grained access controls without significant performance degradation. We list some of the improvements on the three components presented in this thesis as follows:

1. The DOL access control labeling mechanism compresses access control specifications by taking advantage of access control correlations among XML data and among different users. Although the same correlation among the users may exist in relational databases, the correlation of access controls among the data may be different among tuples that are non-tree structured. Further, the DOL scheme nicely fits with the NoK query processor, and lowers the cost of access control lookup. It remains unknown if this scheme can be integrated into existing relational query processors to reduce runtime access control lookup overhead. One big challenge in embedding such a piggy-back mechanism in a relational system is that we need not only change every access path operator, but also the query optimizer to take this factor into consideration.

2. The POM query caching mechanism works for optimizers that separate query rewriting from physical plan generating. It is challenging to customize POM for optimizers of different architecture, i.e., an optimizer that mixes logical query rewrites with physical plan

generation. Moreover, if a database system has its unique physical properties or logical properties that affect its query planning (e.g., ranking operators), POM needs to be customized for those new properties. It is a daunting task to customize POM for every possible property coming from novel database operators.

3. The PSALM sampling scheme requires the samples be built in advance. Adaptive sampling is not currently present. In the future, PSALM can continuously update its samples while executing the access-control-customized queries, in a similar fashion as self-tuning statistics [5]. Moreover, PSALM can be used to assist design sample views [56] under a total sample space budget.

Bibliography

[1] Jaccard Similarity Coefficient. Available at http://en.wikipedia.org/wiki/Jaccard_index.

[2] The XML benchmark project. Available at http://www.xml-benchmark.org.

[3] Serge Abiteboul and Oliver M. Duschka. Complexity of Answering Queries Using Materialized Views. In *Proc. 17th ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 254–263. ACM Press, 1998.

[4] Ashraf Aboulnaga and Surajit Chaudhuri. Self-tuning Histograms: Building Histograms Without Looking At Data. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 181–192, 1999.

[5] Ashraf Aboulnaga, Peter J. Haas, Sam Lightstone, Guy M. Lohman, Volker Markl, Ivan Popivanov, and Vijashankar Raman. Automated Statistics Collection in DB2 Stinger. In *Proc. 30th Int. Conf. on Very Large Data Bases*, pages 1146–1157, 2004.

[6] Swarup Acharya, Phillip B. Gibbons, and Viswanath Poosala. Congressional Samples for Approximate Answering of Group-by Queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 487–498, 2000.

[7] Shurug Al-Khalifa, H. V. Jagadish, Nick Koudas, Jignesh M. Patel, Divesh Srivastava, and Yuqing Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. 18th Int. Conf. on Data Engineering*, page 141, 2002.

[8] Mehmet Altinel and Michael J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proc. 26th Int. Conf. on Very Large Data Bases*, pages 53–64, 2000.

[9] Elisa Bertino, Piero Andrea Bonatti, and Elena Ferrari. TRBAC: A Temporal Role-Based Access Control Model. *ACM Trans. Inf. Syst. Secur.*, 4(3):191–233, 2001.

[10] Elisa Bertino, Silvana Castano, Elena Ferrari, and Marco Mesiti. Controlled access and dissemination of XML documents. In *Workshop on Web Information and Data Management*, pages 22–27, 1999.

[11] Elisa Bertino, Barbara Catania, Elena Ferrari, and Paolo Perlasca. A Logical Framework for Reasoning about Access Control Models. *ACM Transactions on Information and System Security*, 6(1):71–127, 2003.

[12] Elisa Bertino and Elena Ferrari. Secure and Selective Dissemination of XML Documents. *ACM Transactions on Information and System Security*, 5(3):290–331, August 2002.

[13] Elisa Bertino, Sushil Jajodia, and Pierangela Samarati. A Flexible Authorization Mechanism for Relational Data Management Systems. *ACM Transactions on Information and System Security*, 17(2):101–140, 1999.

[14] Elisa Bertino, Beng Chin Ooi, Yanjiang Yang, and Robert H. Deng. Privacy and ownership preserving of outsourced medical data. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, pages 521–532, Washington, DC, USA, 2005. IEEE Computer Society.

[15] Claudio Bettini, Sushil Jajodia, X. Sean Wang, and Duminda Wijesekera. Provisions and Obligations in Policy Management and Security Applications. In *Proc. 28th Int. Conf. on Very Large Data Bases*, pages 502–513, 2002.

[16] Piero Bonatti, Sabrina De Capitani di Vimercati, and Pierangela Samarati. An algebra for composing access control policies. *ACM Transactions on Information and System Security*, 5(1):1–35, 2002.

[17] Kellogg S. Booth. Authentication of signatures using public key encryption. *Commun. ACM*, 24(11):772–774, 1981.

[18] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. STHoles: A Multidimensional Workload-Aware Histogram. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 211–222, 2001.

[19] Silvana Castano, Maria Grazia Fugini, Giancarlo Martella, and Pierangela Samarati. *Database Security*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[20] Sharma Chakravarthy. Divide and Conquer: A Basis for Augmenting a Conventional Query Optimizer with Multiple Query Proceesing Capabilities. In *Proc. 7th Int. Conf. on Data Engineering*, pages 482–490, 1991.

[21] Surajit Chaudhuri, Gautam Das, and Vivek Narasayya. A Robust, Optimization-Based Approach for Approximate Answering of Aggregate Queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 295–306, 2001.

[22] Surajit Chaudhuri, Raghav Kaushik, and Ravishankar Ramamurthy. When Can We Trust Progress Estimators for SQL Queries? In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 575–586, 2005.

[23] Amir H. Chinaei. *Access Control Administration with Adjustable Decentralization*. PhD thesis, University of Waterloo, 2007.

[24] Amir H. Chinaei and Huaxin Zhang. Hybrid Authorizations and Conflict Resolution. In *Secure Data Management*, pages 131–145, 2006.

[25] SungRan Cho, Sihem Amer-Yahia, Laks V. S. Lakshmanan, and Divesh Srivastava. Optimizing the Secure Evaluation of Twig Queries. In *Proc. 28th Int. Conf. on Very Large Data Bases*, pages 490–501, August 2002.

[26] P. Crescenzi and V. Kann. *A Compendium of NP Optimization Problems*, page 13. Springer Verlag, 1999.

[27] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. A General Algebra and Implementation for Monitoring Event Streams. Technical report, Cornell University, 2005.

[28] Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243.

[29] Dorothy E. Denning, Peter J. Denning, and Mayer D. Schwartz. The Tracker: A Threat to Statistical Database Security. *ACM Trans. Database Sys.*, 4(1):76–96, 1979.

[30] Yanlei Diao, Peter M. Fischer, Michael J. Franklin, and Raymond To. YFilter: Efficient and Scalable Filtering of XML Documents. In *Proc. 18th Int. Conf. on Data Engineering*, pages 341–353, 2002.

[31] Yanlei Diao, Shariq Rizvi, and Michael J. Franklin. Towards an Internet-Scale XML Dissemination Service. In *Proc. 30th Int. Conf. on Very Large Data Bases*, pages 612–623, 2004.

[32] Amr El-Helw, Ihab F. Ilyas, Wing Lau, Volker Markl, and Calisto Zuzarte. Collecting and Maintaining Just-in-Time Statistics. In *Proc. 23st Int. Conf. on Data Engineering*, pages 516–525, 2007.

[33] Benjamin Epstein. Introduction to Statistical Analysis. *SIAM Review*, 1(1):75–77, 1959.

[34] Wenfei Fan, Chee-Yong Chan, and Minos Garofalakis. Secure XML Querying with Security Views. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 587–598, 2004.

[35] Csilla Farkas and Sushil Jajodia. The Inference Problem: A Survey. *SIGKDD Explor. Newsl.*, 4(2):6–11, 2002.

[36] Alban Gabillon and Emmanuel Bruno. Regulating Access to XML Documents. In *Proc. 15th Int. Conf. on Database and Application Security*, pages 299–314. Kluwer Academic Publishers, 2002.

[37] César A. Galindo-Legaria, Milind Joshi, Florian Waas, and Ming-Chuan Wu. Statistics on Views. In *Proc. 29th Int. Conf. on Very Large Data Bases*, pages 952–962, 2003.

[38] Rainer Gemulla, Wolfgang Lehner, and Peter J. Haas. A Dip in the Reservoir: Maintaining Sample Synopses of Evolving Datasets. In *Proc. 32th Int. Conf. on Very Large Data Bases*, pages 595–606, 2006.

[39] Antara Ghosh, Jignashu Parikh, Vibhuti S. Sengar, and Jayant R. Haritsa. Plan Selection based on Query Clustering. In *Proc. 28th Int. Conf. on Very Large Data Bases*, pages 179–190, 2002.

[40] Peter J. Haas and Arun N. Swami. Sequential Sampling Procedures for Query Size Estimation. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 341–350, 1992.

[41] Alon Y. Halevy. Answering Queries Using Views: A Survey. *The VLDB Journal*, 10(4):270–294, 2001.

[42] M.A. Harrison, M.L. Ruzzo, and J.D. Ullman. Protection in Operating Systems. *Communications of the ACM*, pages 461–471, 1976.

[43] Arvind Hulgeri and S. Sudarshan. Parametric Query Optimization for Linear and Piecewise Linear Cost Functions. In *Proc. 28th Int. Conf. on Very Large Data Bases*, pages 167–178, 2002.

[44] Arvind Hulgeri and S. Sudarshan. AniPQO: Almost Non-intrusive Parametric Query Optimization for Nonlinear Cost Functions. In *Proc. 29th Int. Conf. on Very Large Data Bases*, pages 766–777, 2003.

[45] IBM. *Label-based Access Control (LBAC) Overview*. Available at http://publib.boulder.ibm.com/infocenter/db2luw/v9/topic/com.ibm.db2.udb.admin.doc/doc/c0021114.htm.

[46] IBM. *DB2 Administration Guide: Implementation*, 2006. Available at ftp://ftp.software.ibm.com/ps/products/db2/info/vr8/pdf/letter/db2d2e80.pdf.

[47] Ihab F. Ilyas, Walid G. Aref, Ahmed K. Elmagarmid, Hicham G. Elmongui, Rahul Shah, and Jeffrey Scott Vitter. Adaptive rank-aware query optimization in relational databases. *ACM Trans. Database Sys.*, 31(4):1257–1304, 2006.

[48] Ihab F. Ilyas, Jun Rao, Guy Lohman, Dengfeng Gao, and Eileen Lin. Estimating Compilation Time of A Query Optimizer. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 373–384, 2003.

[49] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. Parametric Query Optimization. *The VLDB Journal*, 6(2):132–151, 1997.

[50] Sushil Jajodia. *Database Security and Privacy*. CRC Press, 2004.

[51] Sushil Jajodia, Pierangela Samarati, Maria Luisa Sapino, and V. S. Subrahmanian. Flexible Support for Multiple Access Control Policies. *ACM Trans. Database Sys.*, 26(2):214–260, 2001.

[52] Sushil Jajodia, Pierangela Samarati, V.S. Subrahmanian, and Elisa Bertino. A Unified Framework for Enforcing Multiple Access Control Policies. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 474–485, May 1997.

[53] Mingfei Jiang and Ada Wai-Chee Fu. Integration and Efficient Lookup of Compressed XML Accessibility Maps. *IEEE Trans. Knowledge and Data Eng.*, 17(7):939–953, 2005.

[54] James B.D. Joshi, Elisa Bertino, Usman Latif, and Arif Ghafoor. A Generalized Temporal Role-Based Access Control Model. *IEEE Transactions on Knowledge and Data Engineering*, 17(1):4–23, January 2005.

[55] Butler Lampson. Protection. In *ACM Operating Systems Review*, pages 8–24, 1974.

[56] Per-Ake Larson, Wolfgang Lehner, Jingren Zhou, and Peter Zabback. Cardinality Estimation Using Sample Views With Quality Assurance. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 175–186, 2007.

[57] Dongwon Lee, Wang-Chien Lee, and Peng Liu. Supporting XML Security Models Using Relational Databases: A Vision. In *Proc. 1st Int. XML Database Symposium*, pages 267–281, 2003.

[58] Ju-Hong Lee, Deok-Hwan Kim, and Chin-Wan Chung. Multi-dimensional Selectivity Estimation Using Compressed Histogram Information. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 205–214, New York, NY, USA, 1999. ACM Press.

[59] Kristen LeFevre, David J. DeWitt, and Raghu Ramakrishnan. Incognito: Efficient Full-Domain K-Anonymity. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 49–60, 2005.

[60] Richard J. Lipton and Jeffrey F. Naughton. Query Size Estimation by Adaptive Sampling. In *Proc. 9th ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 40–46, 1990.

[61] Teresa F. Lunt. A Survey of Intrusion Detection Techniques. *Comput. Secur.*, 12(4):405–418, 1993.

[62] Gang Luo, Jeffrey F. Naughton, Curt Ellmann, and Michael Watzke. Toward a Progress Indicator for Database Queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 791–802, 2004.

[63] Lothar F. Mackert and Guy M. Lohman. R* Optimizer Validation and Performance Evaluation for Local Queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 84–95, 1986.

[64] Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. Robust query processing through progressive optimization. pages 659–670, 2004.

[65] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. Wavelet-based Histograms for Selectivity Estimation. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 448–459, 1998.

[66] Microsoft. *Implementing Row- and Cell-Level Security in Classified Databases Using SQL Server 2005*, 2005. Available at http://download.microsoft.com/download/4/7/a/47a548b9-249e-484c-abd7-29f31282b04d/RowCellLvlSecSQL.doc.

[67] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge University Press, New York, NY, USA, 1995.

[68] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978.

[69] Oracle. *Data Classification with Oracle Label Security*. Available at http://www.oracle.com/technology/deploy/security/database-security/pdf/twp_security_db_ols_10gr2.pdf.

[70] Oracle. *The Virtual Private Database in Oracle9ir2*. Available at http://otn.oracle.com/deploy/security/oracle9ir2/pdf/vpd9ir2twp.pdf.

[71] Sylvia Osborn, Ravi Sandhu, and Qamar Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Transactions on Information and System Security*, 3(2):85–106, 2000.

[72] Bernd-Uwe Pagel, Hans-Werner Six, Heinrich Toben, and Peter Widmayer. Towards an Analysis of Range Query Performance in Spatial Data Structures. In *Proc. 12th ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems*, pages 214–221, New York, NY, USA, 1993. ACM Press.

[73] Christopher Palmer and Christos Faloutsos. Density Biased Sampling: An Improved Method for Data Mining and Clustering. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 82–92, 2000.

[74] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible Rule Based Query Rewrite Optimization in Starburst. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 39–48, 1992.

[75] Viswanath Poosala and Yannis E. Ioannidis. Selectivity Estimation Without the Attribute Value Independence Assumption. In *Proc. 23th Int. Conf. on Very Large Data Bases*, pages 486–495, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[76] Fausto Rabitti, Elisa Bertino, Won Kim, and Darrell Woelk. A Model of Authorization for Next-Generation Database Systems. *ACM Trans. Database Sys.*, 16(1):88–131, 1991.

[77] Shariq Rizvi, Alberto Mendelzon, S. Sudarshan, and Prasan Roy. Extending Query Rewriting Techniques for Fine-Grained Access Control. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 551–562, 2004.

[78] Walid Rjaibi. An Introduction to Multilevel Secure Relational Database Management Systems. In *Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 232–241. IBM Press, 2004.

[79] Pierangela Samarati and Latanya Sweeney. Protecting Privacy When Disclosing Information: K-Anonymity and Its Enforcement Through Generalization and Suppression. Technical Report SRI-CSL-98-04, SRI Computer Science Laboratory, 1998.

[80] Ravi Sandhu and Qamar Munawer. How to do Discretionary Access Control Using Roles. *Proceedings of the Third ACM Workshop on Role-Based Access Control*, pages 47–54, 1998.

[81] Ravi S. Sandhu. Lattice-based Access Control Models. *IEEE Computer*, (11):9–19, 1993.

[82] Ravi S. Sandhu and Pierrangela Samarati. Access Control: Principles and Practice. *IEEE Communications Magazine*, 32(9):40–48, 1994.

[83] Timos K. Sellis. Multiple-Query Optimization. *ACM Trans. Database Sys.*, 13(1):23–52, 1988.

[84] William Stallings. *Cryptography and Network Security (2nd ed.): Principles and Practice*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.

[85] Andrei G. Stoica and Csilla Farkas. Secure XML Views. In *Proc. 15th Int. Conf. on Database and Application Security*, pages 133–146, 2002.

[86] Latanya Sweeney. Achieving K-Anonymity Privacy Protection Using Generalization and Suppression. *International Journal on Uncertainty, Fuzziness, and Knowledge-based Systems*, 10(5):571–588, 2002.

[87] Woei-Jiunn Tsaur, Shi-Jinn Horng, and Chia-Ho Chen. An Authentication-Combined Access Control Scheme Using A Geometric Approach In Distributed Systems. In *Proc. of the 1997 ACM symposium on Applied computing*, pages 361–365, New York, NY, USA, 1997. ACM Press.

[88] Jeffrey S. Vitter. Random Sampling with a Reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.

[89] Duminda Wijesekera and Sushil Jajodia. Policy Algebras for Access Control: the Propositional Case. In *Proc. 8th ACM Conference on Computer and Communications Security*, pages 38–47. ACM Press, 2001.

[90] Duminda Wijesekera and Sushil Jajodia. Policy Algebras for Access Control – the Predicate Case. In *Proc. 9th ACM Conference on Computer and Communications Security*, pages 171–180. ACM Press, 2002.

[91] Duminda Wijesekera and Sushil Jajodia. A Propositional Policy Algebra for Access Control. *ACM Transactions on Information and System Security*, 6(2):286–325, 2003.

[92] Duminda Wijesekera, Sushil Jajodia, Francesco Parisi-Presicce, and Asa Hagstrom. Removing Permissions in the Flexible Authorization Framework. *ACM Trans. Database Sys.*, 28(3):209–229, 2003.

[93] Wolfram Mathworld. *Bell Numbers*. Available at http://mathworld.wolfram.com/BellNumber.html.

[94] Thomas Y. C. Woo and Simon S. Lam. Authorizations in Distributed Systems: A New Approach. *Journal of Computer Security*, 2(2-3):107–136, 1993.

[95] Tak W. Yan and Hector Garcia-Molina. The SIFT Information Dissemination System. *ACM Trans. Database Sys.*, 24(4):529–565, 1999.

[96] Ting Yu, Divesh Srivastava, Laks V. S. Lakshmanan, and H. V. Jagadish. A Compressed Accessibility Map for XML. *ACM Trans. Database Sys.*, 29(2):363–402, 2004.

[97] Huaxin Zhang, Ning Zhang, Kenneth Salem, and Donghui Zhuo. Compact Access Control Labeling for Efficient Secure XML Query Evaluation. *Journal of Data and Knowledge Engineering*, 60(2):326–344, February 2007.

[98] Ning Zhang, Varun Kacholia, and M. Tamer Özsu. A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML. In *Proc. 20th Int. Conf. on Data Engineering*, pages 54–65, 2004.

[99] Xinwen Zhang, Jaehong Park, Francesco Parisi-Presicce, and Ravi Sandhu. A Logical Specification for Usage Control. In *Proc. 9th ACM Symp. on Access Control Models and Technologies*, pages 1–10, New York, NY, USA, 2004. ACM Press.

[100] Zheng Zhang and Alberto O. Mendelzon. Authorization Views and Conditional Query Containment. In *Proc. 10th Int. Conf. on Database Theory*, pages 259–273, 2005.