On Query Optimization Issues in Fine-Grained Access Control

B. Tech Project Report

Submitted in partial fulfillment of the requirements for the degree of Bachelor of Technology

by

B. Aditya Prakash
Roll No: 03005030

under the guidance of

Prof. S. Sudarshan

Department of Computer Science and Engineering
Indian Institute of Technology - Bombay
Mumbai
B.Tech Project Approval Sheet

I hereby state that the contents of this work is mine. Any substantial borrowed material (cut-pasted or otherwise) including figures, tables, and sketches have been duly acknowledged.

Date: B. Aditya Prakash
B.Tech, 03005030

I hereby give approval for the B.Tech Project Report titled “On Query Optimization Issues in Fine-Grained Access Control” by B. Aditya Prakash (03005030) to be submitted.

Date: Prof. S. Sudarshan
(Guide)
Acknowledgments

I would like to thank my advisor, Prof. S. Sudarshan, for his patient guidance, valuable insights and steady encouragement throughout the course of the work; working with him has been a very enriching experience. A special thanks to Ravi G. for his help right from selecting this problem for the project to helping resolve nagging implementation issues. I would also like to thank my friends and labmates for giving company and playing TT with me during those long night-outs.

B. Aditya Prakash
May 03, 2007
Abstract

Access control refers to the rules and mechanisms which govern the access of data in a system. Relational databases today have woken up to the real need of providing fine-grained access control to users. Much work has been done in this area in the past. We first survey the different models proposed for implementing such authorization in a RDBMS. We then focus on query optimization issues arising in view-replacement strategies. We develop formally a better method of combining redundancy removal and safe-plan generation in presence of unsafe user-defined functions. We have implemented the techniques on a Volcano based query optimizer developed in IIT Bombay. We then present performance results which reinforce theory and empirically demonstrate significant gains over the previous method. Finally, we also look at some avenues for future work in this field.
1 Introduction 1

2 Models for Access Control 3
   2.1 Oracle VPD ................................. 3
   2.2 Non-Truman Model ......................... 5

3 Redundancy and Leakage in Truman Models 7
   3.1 Authorization Model ........................ 7
   3.2 Redundancy Removal ........................ 7
      3.2.1 Detecting Redundancy ................. 8
      3.2.2 Rules ................................. 8
   3.3 Information Leakage ....................... 9
      3.3.1 Safety of Plans ....................... 10
      3.3.2 Authorization Inference .............. 10
      3.3.3 Generating Safe Plans ............... 11
   3.4 Redundancy Removal and Safety ............. 12
      3.4.1 Redundancy Removal During Transformation 12
      3.4.2 Conditioned Authorization .......... 12

4 Conditioned Authorization 13
   4.1 Notations .................................. 13
   4.2 Rules ..................................... 14
      4.2.1 $E = E_1 \bowtie_\theta E_2$ ............... 14
      4.2.2 $E = \prod_L(E_1)$ ..................... 17
      4.2.3 $E = \sigma_\theta(E_1)$ .................. 17

5 Implementation 18
   5.1 Overview of the Optimizer .................. 18
   5.2 Extensions to the Optimizer ............... 19
      5.2.1 Introducing Semi-Joins ................ 20
      5.2.2 Handling User-Defined Functions (UDFs) 20
      5.2.3 Handling View Replacement ............ 21
6 Experiments
6.1 Experimental Setup ........................................ 27
6.2 Varying Data Sizes ................................. 28
6.3 Varying Query Sizes ............................... 29
6.4 Comparison with No Redundancy Removal .......... 30

7 Conclusions and Future Work .......................... 34
Chapter 1

Introduction

Access control deals with controlling access of data to users based on a variety of parameters. Needless to say, it is a very important part of data management systems which are increasingly pervasive in a variety of secure installations. The granularity of access control refers to the quantum of the individual data items whose access can be controlled. The current SQL standard specifies only coarse-grained control - only at the level of entire databases, whole relations or columns. This means that a particular user can either access all the tuples of a relation or none. Clearly, this granularity is grossly inadequate in the present scenarios where a large number of applications demand finer levels of authorization. As an example, even for simple databases, like a university’s database to store student records requires the restriction of access of students to records other than his own.

Fine-grained authorization, hence aims to provide access control to specific columns of a subset of rows of a table. Till some years ago, the burden of actually implementing the access-control policies was placed with the application programmer. This approach creates a multitude of problems like redundant code, increased code-size etc. Therefore, fine-grained access control should be the responsibility of the database system itself. A naïve way of doing this would be maintaining a simple list of users with each tuple or cell of a database. But clearly, with extremely large databases having millions of tuples, this is infeasible. We may also maintain a view for each user and each relation which he is authorized to access. But again, this requires maintenance of millions separate views for a single table. In sum, these techniques can not be scaled and only demonstrate the need for an authorization transparent and efficient model of access control.

The rest of this report is organized as follows. We first discuss the various models of authorization enforcement proposed till now (Chapter 2). Then we have a look at redundancy and information leakage in the class of query modification models (Chapter 3). Kabra et al. [KRS06] introduce the concept of conditioned authorization in this context. We formalize and propose a concrete theory for the same (Chapter 4). After this, we look at integrating these techniques into a Volcano based query optimizer (Chapter 5). We then proceed to provide an experimental evaluation using both the methods on various queries (primarily based on the TPC-H schema) (Chapter 6). The last chapter then deals with conclusions and ideas for further work.

We assume a working knowledge of database systems, in particular the optimizer. Also, throughout the report, as far as possible, we try to use the following relation for giving examples - \texttt{emp (eid, dept, bldg, salary, hodName, sat)}, where,
1. **eid** refers to the employee id

2. **dept** refers to the department the employee is working in

3. **bldg** refers to the building in the employee is working in (assuming that more than one department may be housed in the same building)

4. **hodName** refers to the name of the current department HoD

5. **sat** refers to a numerical value which indicates the level of satisfaction of the employee
Chapter 2

Models for Access Control

In this chapter we present an overview of the various models that have been proposed for access control. Using the parlance introduced by Rizvi et al. [RMSR04], they can be broadly classified as Truman or Non-Truman models. Truman models are those which involve query modification of some sort. Oracle’s Virtual Private Database (VPD) is a specific example of such a strategy (but it does not adopt the general approach). Non-Truman models are checking models which first do a validity testing for a query and then run it unmodified if it passes the test.

2.1 Oracle VPD

The VPD feature of Oracle’s 9iR2 RDBMS [Pap02] provides fine-grained access control by modifying the user query at the server itself. In a nutshell, we can dynamically attach, at runtime, a predicate (WHERE clause) to any and all queries issued against a database table or view. The actual authorization policy is encoded into functions defined for each relation. When a user logs in, a specific Application Context is created which sets user-specific information such as who has logged on, where has he logged on from etc. The context provides this information to the procedures written for each relation which uses these as parameters to instantiate a specific predicate which is then attached to the user query. The server then executes this predicated query. For example, consider the following query issued by user USERID:

Query1: SELECT * FROM emp;

Suppose we have a policy that employees can see their own records only. Then the re-written query will be:

VPDQuery1: SELECT * FROM emp
WHERE emp.eid = USERID;

Note that VPDQuery1 is now authorized.

Limitations

Oracle VPD currently provides no mechanism to project out certain attributes i.e., no column level security is present. It is also difficult to write predicates involving cases of
cross-ref, joins of tables etc. Although the user’s query is relatively simple, the the onus of ensuring secure access rests squarely with the author of the security policy - and actually writing policy functions corresponding to business policies is a lot of work.

While the Truman security model, in essence, also re-writes the query as the Oracle VPD, the re-writing is done in a more fundamental and principled manner. The basic idea behind Truman models is to provide each user with a personal and restricted view of the complete database [RMSR04]. Hence the modifications make sure that the data accessible by the query is exactly what the user has been authorized to view.

To this end, parametrized authorization views are defined for each relation in the database. The parameters again may be USERID, DATE etc. (these can be acquired from the context). Note that these kinds of view definitions can be provided by the DBA. For a particular relation, the view completely defines all the data that the user can access from this relation. When the user fires a query, the system simply substitutes each reference of a relation by its corresponding parameterized view (of course with the parameters instantiated).

As an example consider Query1 again. For implementing the same policy as discussed in Section 2.1, suppose the DBA creates a parametrized view ($user-id$ is the parameter here) as:

```sql
View1: CREATE AUTHORIZATION VIEW restrictedEmp AS
  SELECT * FROM emp WHERE emp.eid = $user-id;
```

The re-written query in this case will be:

```sql
TMQuery1: SELECT * FROM restrictedEmp;
```

The view definition itself is expanded later while processing TMQuery1. This strategy hence resolves most of the problems with the other methods. Note that the authorization view might be very complicated and general. In fact, we can also define access pattern views [RMSR04] which just govern access patterns. Thus, we can enforce many kinds of policies and at any granularity. This generality provides these kinds of models with much power. Also, as with all query modification techniques, the user query can address the base tables and/or authorization views - the system will take care of the authorization checks.

**Limitations**

All Truman models suffer from subtler flaws which stem from the fact that they inherently modify the query [RMSR04]. In fact, the result of re-written query may be different from that of the original query. This causes inconsistencies in what the user expects and what the system gives as the final answer. Consider the following example where a employee wants to know about the average salary of employees in the organization. He issues the following query:

```sql
Query2: SELECT avg(salary) from emp;
```

But according to View1, it gets modified to:

```sql
TMQuery2: SELECT avg(salary) from restrictedEmp;
```
Clearly, the result of $\text{TMQ}uery2$ will be the particular user’s own salary - leading him to believe that his salary is same as the average salary. This situation is decidedly undesirable; ideally, this query should have been rejected.

Another issue with Truman models is that the re-written query may be very different from the original query - it may have completely different execution plans, runtimes etc. The modified query may contain redundant tests and joins, expensive views and consequently may be very expensive to execute. We discuss these problems in more detail in Chapter 3.

### 2.2 Non-Truman Model

Motivated by the above limitations, the Non-Truman model has been proposed as an alternative [RMSR04]. Under this model the submitted query is run unchanged if it passes a inference test. The inference test intuitively checks whether the given query can be answered using only the information contained in the authorization views available to the user. This kind of validity testing is an attractive notion as it guarantees correctness - in contrast to the view replacement models.

In this context, there are two notions of validity - Unconditional and Conditional validity.

**Unconditional Validity** A query $Q$ is unconditionally valid if there is a query $Q_1$ which can be written using only the instantiated authorization views and which is equivalent to $Q$ i.e. $Q_1$ produces the same result as $Q$ on all database states. For example, consider the following query:

```
Query3: SELECT * FROM emp WHERE emp.eid = 'CSE5030';
```

Assuming that the user who issues this query actually has CSE5030 as his $\text{user-id}$, clearly Query3 is equivalent to $\text{TMQ}uery1$ and therefore unconditionally valid.

**Conditional Validity** The concept of conditional validity is trickier. Note that in unconditional validity, equivalence of two queries needs identical results for both the queries for all database states. But, intuitively, this is too strong a condition - the information contained in the authorization views might rule out some states - and equivalence on these states is not needed. Thus, we might be able to infer that the given query $Q$ would give the same result as another query $Q_1$ that uses only authorization views, in the current database state. Bases on this information, the query $Q$ should be declared as valid as the user could have got the same information if he had issued $Q_1$ on the current database state. As an example, suppose we define a new view $\text{deptHead}$ which allows a particular employee to see information of HoDs of only those departments in which he has worked before. Now consider the following query:

```
Query4: SELECT hodName FROM emp WHERE emp.depName = 'CSE';
```

Clearly, it is not possible to write any authorization equivalent query for Query4 for all database states. But, if the user has worked in CSE before, then the following query is equivalent to Query4:
NTMQuery4: SELECT hodName FROM deptHead WHERE emp.depName = 'CSE';

Therefore the query **Query4** is conditionally valid depending on the database state. But, as shown in [RMSR04], the weak condition of having the same results only on the current database state is not safe and leaks information. They thus have further characterized the states on which equivalence is needed for conditional validity to be the set of PA-equivalent states of the current state (a PA-equivalent state is basically a state in which the instantiated authorization views for a given set of parameters return the same result as the current state).

Thus, in sum, in a Non-Truman model, the query is accepted if it is either unconditionally or conditionally valid. The exact process of inference whether a given query is valid is or not is carried out by application of inference rules. We request the reader to refer [RMSR04] for details on various inference procedures.

**Limitations**

Although the Non-Truman model address the problems associated with the Truman model, it is not used in practice. First of all, the problem of validity testing is undecidable in the unconditional validity case and the status is unknown for conditional validity. Thus, the systems implementing the model would be unpredictable - one system may accept a query while other would reject it based on their respective sets of inference rules. In addition, inference procedures are expensive to implement. Hence while the bottleneck in Truman model is the expensive re-written query, in Non-Truman model it is the expensive checking procedure. Moreover, the inference rules are far from complete and need to be very powerful for efficiency. As a result, Truman models are used in practice in preference to the Non-Truman model.

Having discussed the various models for fine-grained access control, we now take a closer look at some of the query optimization issues in Truman models. The next chapter deals with the general problem of redundancy and information leakage in view replacement strategies and the proposed solutions to them.
Chapter 3

Redundancy and Leakage in Truman Models

We briefly touched upon these issues in the previous chapter in context of limitations of Truman models. These are two significant problems which have to be addressed by any implementation of the query-modification models. Kabra et al. [KRS06] discuss at length about these two problems and have even proposed solutions for them.

3.1 Authorization Model

First of all, we will fix the authorization view model which we would be following for the rest of this report. As mentioned before, Truman models are authorization transparent - hence they allow for all kinds of views. The methods detailed in [KRS06] are applicable to a specific class of authorization views. These are views which return a subset of the tuples of the relation involved. Note that the view, in addition, may project out some columns. Thus, we allow for any view of the kind:

\[
\text{CREATE AUTHORIZATION VIEW authView AS SELECT L FROM R WHERE P}
\]

where \( L \) is a list of attribute names (it may be \( * \)) and \( P \) may include subqueries. For simplicity we assume that \( L \) is indeed \( * \) - it is easy to extend the solutions if \( L \) is a subset of the attributes. It should be easy to see that authorized views of the above form with \( L \) being \( * \) can be represented as a semi-join expression \( R \bowtie A \), where \( A \) is an expression containing the subqueries in \( P \). Moreover, we assume that the selection conditions in \( P \) are folded into \( \theta \). Also, we would be concentrating on SELECT views only - views for UPDATES etc. can be handled analogously [KRS06].

To summarize, given a join-query such as \([R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n]\), we re-write the query to \([\langle R_1 \bowtie_{\theta_1} A_1 \rangle \bowtie (R_2 \bowtie_{\theta_2} A_2) \bowtie \ldots \bowtie (R_n \bowtie_{\theta_n} A_n)]\).

3.2 Redundancy Removal

In view replacement techniques, the base relations in the given query are replaced by authorized views. The original query usually includes predicates/joins that ensure that
the query accesses only authorized information. Thus, the additional checking introduced by view replacement are redundant and only make the re-written query expensive and hence they should be detected and removed. As a simple example, consider Query3 (Chapter 2) which is submitted by the user. The re-written query will be:

\[
\text{TMQuery3: } \text{SELECT } * \text{ FROM (SELECT } * \text{ FROM emp WHERE emp.eid = 'CSE5030'}) \text{ WHERE emp.eid = 'CSE5030';}
\]

Clearly, there unnecessary checks that are taking place in the re-written query. This kind of redundancy can be very wasteful and only increases running as well as optimization time.

3.2.1 Detecting Redundancy

The next natural question is to how to exactly detect redundancy in queries. Unfortunately, the problem can be reduced to a query minimization problem which is NP-Complete for SPJ (Select-Project-Join or Conjunctive) queries. In fact, for arbitrary expressions, it is undecidable. Only for the special case of distinct relations, it is in \( \mathbf{P} \) for SPJ queries.

Our main aim is to detect redundant semi-joins i.e. we want to find out those semi-joins which are unnecessary. Intuitively, if \( E_2 \) subsumes \( E_1 \), the semi-join \( E_1 \bowtie \theta E_2 \) is redundant and we can replace the semi-join by \( E_1 \).

In this context, Kabra et al. [KRS06] have exploited already existing code for view matching. In view matching, we try to replace an expression \( R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n \) with something of the form \( \sigma_\theta(V) \), where \( V \) is a materialized view. In this, we need to check whether the view \( V \) actually subsumes \( R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n \). So we can re-use these subsumption checks for our case too. One important thing to note is that optimizers adopt the heuristic of using the SPJ normal form of expressions before attempting subsumption checks. This is simply because for most expressions such a form is indeed possible and it is more efficient than matching query trees.

Although it is hard to characterize necessary conditions for subsumption exactly, we can come up with \textit{sufficient} conditions for it to hold. In essence, we adopt the same tests as used in view matching with one important change. The view subsumption requires that \( V \) should have the exact same set of relations as the original expression. Clearly, we can relax this for \( E_1 \bowtie \theta E_2 \) reduction - \( E_2 \) should just have a \textit{subset} of relations of \( E_1 \). Refer [KRS06] for details on the exact subsumption conditions used.

3.2.2 Rules

Given that we know how to test for subsumption, we can introduce transformational rules to be coded in the optimizer so that redundancy removal can be performed. Please refer to the paper for the list of transformational rules. The rules only look for expressions of the \( E_1 \bowtie \theta E_2 \) - so to detect all redundancies we have to apply redundancy removal during the transformation phase of the optimizer, when all the other query transformational rules are also being applied. But in this case, the redundancy test can be fired an exponential number of times. But if we don’t do this, we risk the non-detection of many redundancies.
Hence to simplify the removal and make it more efficient we adopt a heuristic before applying the redundancy rules. After query re-writing, we pull all semi-joins through joins and selections. For example, \(((R_1 \bowtie_{\theta_1} A_1) \bowtie (R_2 \bowtie_{\theta_2} A_2) \bowtie \ldots \bowtie (R_n \bowtie_{\theta_n} A_n))\) will become \(((R_1 \bowtie R_2 \ldots R_n) \bowtie_{\theta_1} A_1 \bowtie_{\theta_2} A_2 \ldots)\) after such an operation. We now apply the removal rules which result in a greater amount of redundancy removal than if we had applied on the immediately re-written query.

### 3.3 Information Leakage

The second issue which plagues all Truman model implementations is the question of effectiveness of the implementation in a general model of attack. Even if the fine-grained authorization implementation ensures that the query result contains only authorized information, there still are other channels which can leak information. They include:

- Exceptions and error messages
- User-Defined Functions (UDFs)

As an example, consider the following query:

**Query6:** \[ \text{SELECT * FROM emp WHERE udf(salary)} \]

Now suppose the view `restrictedEmp` is redefined such that it is now a semi-join between `emp` and some other query `Q1`. The query tree of the re-written query is shown in Figure (3.1a). Now, two result-equivalent plans are shown in Figure (3.1b) and Figure (3.1c). Note that while the former is a safe plan while the latter is not. This is because in the latter plan, the UDF is allowed access to the `emp` table directly. Now the behaviour of this UDF is not under the control of the system. Thus, it could potentially save the whole `emp` table to a file, or print out the values. Thus a channel of information leak is possible with the latter plan.

Exceptions and error messages may also inadvertently leak information. Consider the query:
Query7: SELECT * FROM emp
WHERE emp.eid = ‘CSE5030’ AND 1/(emp.salary-50000) = 0.1;

Clearly, if the employee is not allowed to see his salary and his salary is indeed 50000,
then Query7 may possibly divulge (through a divide-by-zero error) that CSE5030’s salary
is 50000.

Thus it is necessary to manually classify functions and operators as safe or unsafe
(USFs). Once we have done this, we can turn our attention to characterizing exactly when
a given plan is safe w.r.t. to USFs.

3.3.1 Safety of Plans

A naïve way to classify plans as safe would be to mark all those plans as safe where all the
USFs in the plan get their parameters from authorized relations. A relation is authorized if
its access control checks have been enforced. On the other hand, an expression is authorized
if it is equivalent to an expression defined using only authorized views. But, as shown
in [KRS06], this is too weak a condition and hence can leak information. The correct
conditions for safety of a node in a query plan would be:

1. if there are no USFs in the node, and all inputs of the node are safe.
2. the node has a USF, it is not an apply operator (introduced in [GLJ01]), and all its
   inputs are safe and authorized.

There are other conditions for the apply operator too which we will not go into here.
In sum, the whole plan is safe if its root is safe. Note that the property of being authorized
is different from plan safety. Safety is the property of a particular plan evaluating an
expression which ensures that the query plan does not leak information using channels
other than the query result itself. On the other hand, authorization is the property of the
expression itself.

To conclude, intuitively, in a safe plan, every USF is invoked only on the results of
authorized expressions.

3.3.2 Authorization Inference

The next step therefore is to efficiently check whether a given expression is authorized.
Note that this step is exactly same as inferring Unconditional Validity of a given query
in Non-Truman models (Chapter 2). For this purpose, many inference rules have already
been developed by Rizvi et al. [RMSR04]. But, there we had to check the authorization
of only a single query; here we have to check the authorization of potentially a very large
number of expressions. So we have to adopt a heuristic here. Kabra et al [KRS06] have
used the validity propagation approach for inferring authorization. This rule by itself is
very simple but is quite powerful. If all the query plan space is searched, then this rules
guarantees that all authorized expressions will be marked authorized. The rule is:
Validity Propagation Rule  If all children group nodes of an operation node are marked authorized, then the group node which is the parent of that operation node is also marked as authorized.

The correctness of the rule is quite obvious. The intuition behind this is that authorized views which replace base relations in the the query plan are all authorized and this authorization can be propagated up the tree during the transformation phase of the optimizer. As an example, consider Figure (3.2). The initial query tree’s (Figure(3.2a)) DAG is shown in Figure(3.2b). The group nodes which are directly representing the substituted authorized views are marked as authorized by default (black nodes are the authorized nodes). Figure(3.2c) shows an expanded DAG with repeated application of the propagation rule.

3.3.3 Generating Safe Plans

After having seen which plans are safe, we can now describe approaches to generate safe plans. A simple solution would be to pull-up all USFs to the top. This approach trivially guarantees safe plans. But, the resultant query plan may not be optimal at all. Hence, we will try to determine if there is a straightforward way instead of heuristics. Two strategies have been proposed by Kabra et al. [KRS06]. They are:

1. Modify all optimizer rules such that USFs are pushed only on top of authorized-marked expressions.

2. We allow for unsafe transformations, but enforce safety when picking the opti-
mal plan. This enforcement can be done by using the optimizer feature of en-forcers [Roy00].

3.4 Redundancy Removal and Safety

We have seen solutions for both redundancy removal and safety of plans in isolation. But, these solutions interact in a subtle way. Recall that in redundancy removal, we simplify the query in a simplification stage before the optimizer’s query transformation phase. Thus, potentially we will lose out on a number of authorization views which would have been crucial in validity propagation. Due to this, many nodes which are actually authorized, would not be marked so and hence we will not be considering many safe plans. Two ways have been proposed to handle this issue by [KRS06]. Please refer to the reference for more details on these approaches.

3.4.1 Redundancy Removal During Transformation

One approach would be to apply redundancy removal during the transformation phase only. Thus at the start of the transformation phase all the authorization views \((A_1, A_2, \ldots, A_n)\) are intact and they can be marked as authorized. But as pointed out previously (Section 3.2.2), this results in multiple application of the redundancy checking rules. Moreover as the redundant parts are detected late, the search space to explore is also larger. We will refer to this approach as Normal Authorization in the rest of the report.

3.4.2 Conditioned Authorization

Another approach to integrate safe plan generation is to perform redundancy removal at simplification time itself, but use an extended notion of authorization, which is called conditioned authorization. Kabra et al. [KRS06] briefly motivate this approach. We formalize and propose a theory for the same in the next chapter.
Chapter 4

Conditioned Authorization

The concept of Conditioned Authorization was introduced in the previous chapter as an alternative approach for integrating redundancy removal and generation of safe plans. With conditioned authorization, instead of marking an expression as authorized, we mark it as authorized conditioned on a semi-join with some other expression. Thus, this property does not change even if the other expression (which enforces authorization) is removed during the query simplification stage or is moved elsewhere during transformations. Authorization conditioned on an empty expression is same as unconditioned authorization - in other words, the expression can be marked as authorized. Note that this allows us to do redundancy removal before transformation during simplification and take advantage of the reduced search space. Therefore, this is a better solution than the one suggested in Section 3.4.1.

Group nodes derive their authorization as follows. If any child is unconditionally authorized, so is the group. Otherwise, the authorization condition for the group is the disjunction of the authorization conditions of its children. We can push a USF only on a group node that has been marked authorized or equivalently when it is unconditionally authorized.

Note that now we have to modify the propagation rules to accommodate conditioned authorization. We consider that an expression $E$ is conditionally authorized (c.a.) on a list of the form $[(A_1, \theta_1), \ldots, (A_n, \theta_n)]$ where $A_i$’s are expressions with which $E$ is successively semi-joined and $\theta_i$’s are the respective semi-join conditions. We now consider various rules for simplification and propagation of conditioned authorization up the query tree. We also give examples for some tricky cases.

4.1 Notations

Here are some notations which we will use throughout this chapter.

1. $A \cap B$ : Set of attributes common to both $A$ and $B$.

2. $A \cup B$ : Set of attributes in $A$ or $B$.

3. $A \supseteq_L B$ : Relation $A$ subsumes relation $B$ on the attributes in $L$. That is, $\Pi_L(A) \supseteq \Pi_L(B)$.
4. $\mathcal{A}(i)$ : The set of attributes used in the item $i$ (where $i$ can be a predicate, relation etc).

5. $\pi_L(\theta)$ : The projection of predicate $\theta$ on attributes in the set $L$. That is, if $\theta = \theta_1 \land \theta_2 \ldots \land \theta_n$, then $\pi_L(\theta) = \theta_{i_1} \land \theta_{i_2} \ldots \land \theta_{i_m}$ such that $\forall j, \mathcal{A}(\theta_{i_j}) \subseteq L$.

6. $\theta_1 \supseteq \theta_2$ : The conditions in $\theta_1$ imply the conditions in $\theta_2$.

4.2 Rules

We consider certain forms of expressions and give rules for c.a. propagation and simplification. The expressions which are considered are of the form $E = E_1 \bowtie_{\theta} E_2$, $\prod_L R$ and $\sigma_\theta R$. Another important point is that we assume throughout that the formulae are well-defined. For example, in $\sigma_\theta R$ we assume that $\mathcal{A}(\theta) \subseteq R$, in $R \bowtie_{\theta} S - \mathcal{A}(\theta) \subseteq R \cup S$ and so on. Also, we only define rules for the case where each of $E_1$ and $E_2$ are c.a. on $[A_1, \theta_1]$ and $[A_2, \theta_2]$. It is easy to see how to extend this for the general case. We adopt the following algorithm for it:

1. Consider each $(A_i, \theta_i)$ in the list separately and if possible, pull it up.

2. Now simplify the resulting authorization list by applying rules successively on each element of the list until it can not be further simplified.

A basic simplification is that if the authorization is redundant (i.e. if it subsumes the relation in the sense we know), then we can simply drop it. Other cases for simplification follow.

4.2.1 $E = E_1 \bowtie_{\theta} E_2$

Consider expression $E = E_1 \bowtie_{\theta} E_2$. If $E_1$ is c.a. on $[(A_1, \theta_1)]$ and $E_2$ is c.a. on $[(A_2, \theta_2)]$ then $E$ is c.a. on the list $[(A_1, \theta_1), (A_2, \theta_2)]$. In some cases however we can simplify this list.

Firstly, without assuming any structure on any $A$’s, we define the conditions on which they can be dropped and not carried up. Let $L_1 = \mathcal{A}(\theta) \cap (A_1 \cap E_2)$ and $L_2 = \mathcal{A}(\theta_1) \cap (A_1 \cap E_2)$ and $L = L_1 \cup L_2$. Then the conditions on which $(A_1, \theta_1)$ can be dropped are:

- $A_1 \supseteq_I E_2$
- $\mathcal{A}(\theta) \subseteq (A_1 \cap E_2) \cup E_1$
- $\mathcal{A}(\theta_1) \subseteq (A_1 \cap E_2) \cup E_1$, and
- $\theta_1 \supseteq \theta$

Analogously, we can write out the conditions for $(A_2, \theta_2)$ to be dropped.

As an example illustrating the above rule, consider that we have a table medical_record with the schema $(\text{empid, disease})$. Further suppose we have an authorization view as follows ($\text{user-blidg}$ returns the building housing the department; there can be more than one department in the same building):
View3:  
CREATE AUTHORIZATION VIEW authMedical AS
SELEn* FROM medical_record
WHERE empid IN (SELECT emp.eid FROM emp
WHERE emp.bdlg = $user-bldg) AS K;

Now consider the following query ($user-dept returns the department of the user):

Query8:  
SELECT *
FROM medical_record M, (SELECT * FROM emp
WHERE emp.dept = $user-dept) AS E
WHERE M.empid = E.eid;

Note that authMedical implies that medical_record is authorized conditioned on (K, empid = K.eid). Also, the user query is a $\theta$-Join of medical_record and E with $\theta$ being empid = E.eid. Note that this situation trivially satisfies the last condition as $\theta = \theta_1$. The other conditions except the first one are also clearly satisfied. To see that even the first condition is satisfied, remember that there may be more than one department in the building and hence by definition, K will contain all the tuples in E and some more. Hence even the first condition is satisfied and we can simplify and drop the authorization condition. Query8 is unconditionally authorized - intuitively, it can not release more information than what would have been allowed by the authorization view itself.

We now successively assume some particular structure on the A’s and check if we can do something better in cases when the above simplification is inapplicable. We give the rules only for the simplification of (A1, $\theta_1$). As a join is commutative, the result can be symmetrically written for (A2, $\theta_2$).

A1 = R $\bowtie_{\theta_3}$ S

Let $L_1 = A(\theta) \cap (R \cap E2)$, $L_2 = A(\theta_1) \cap (R \cap E2)$ and $L_3 = A(\theta_3) \cap (R \cap E2)$ and $L = L_1 \cup L_2 \cup L_3$. Then if,

- $R \supseteq_L E2$
- $A(\theta) \subseteq (R \cap E2) \cup E1$
- $A(\theta_3) \subseteq (R \cap E2) \cup S$
- $A(\theta_1) \subseteq (R \cap E2) \cup S \cup E1$, and
- $\pi_{A(E1) \cup A(R)}(\theta_1) \supseteq \theta$

the expression E is c.a. on $(S, (\theta_1 - \pi_{A(E1) \cup A(R)}(\theta_1)) \land \theta_3)$. In the same way, we can also write down conditions for simplification if $S \supseteq E2$.

As an example, suppose we have another relation prevHead (depName, disease, sat), where disease is the disease that the previous HoD of depName was affected by and sat refers to a numerical value indicating the level of satisfaction of the HoD. Consider that we the following view instead of authMedical:
CREATE AUTHORIZATION VIEW authMedical_1 AS
SELECT * FROM medical_record
WHERE EXISTS (SELECT * FROM emp, prevHead
    WHERE emp.dept = prevHead.depName AND
    emp.eid = empid AND
    prevHead.disease = disease AND
    emp.dept = $user-dept);

Semantically, it allows a user to see the medical records of only those employees who work in his own department and who have/had a disease which even their previous HoD had. Note that $R$ here consists of only those tuples from $emp$ where the department is same as the $\$user-dept$. In short, $R$ is same as $E$. Further, $R$ is joined to $prevHead$ (hence $S$ is $prevHead$ in this case). Now consider Query8 again. Note the following:

1. $\theta \equiv M.empid = E.eid$
2. $\theta1 \equiv emp.eid = empid AND prevHead.disease = disease$
3. $\theta3 \equiv emp.dept = prevHead.depName$

Now, it is easy to verify that all the conditions hold true and hence the simplification can take place.

$A1 = R \bowtie_{\theta3} S$

Similarly, let $L1 = A(\theta) \cap (R \cap E2)$, $L2 = A(\theta1) \cap (R \cap E2)$ and $L = L1 \cup L2$. Then if,

- $R \supseteq L E2$
- $A(\theta) \subseteq (R \cap E2) \cup E1$
- $A(\theta3) \subseteq (R \cap E2) \cup S$
- $A(\theta1) \subseteq (R \cap E2) \cup E1$, and
- $\theta1 \supseteq \theta$

the expression $E$ is c.a. on $(S, \theta1 \land \theta3)$.

The previous example with minor changes again works here. Consider the same query (Query8), but with a new view:

CREATE AUTHORIZATION VIEW authMedical_2 AS
SELECT * FROM medical_record
WHERE EXISTS (SELECT * FROM emp
    WHERE emp.eid = empid AND
    emp.dept = $user-dept AND
    EXISTS (SELECT * FROM emp, prevHead
        WHERE
        emp.dept = prevHead.depName AND
        emp.sat = prevHead.sat));
This view allows employees to see the medical records of those employees (in the same department) who have the same amount of satisfaction as their previous HoD. It is quite trivial to see that this example also fits the above rule and suitable simplification can be carried out.

\[ A_1 = \sigma_{\theta_3} R \]

It is easy to see that \( \theta_3 \) can be folded into the corresponding semi-join condition \( \theta_1 \). Actually, \((\sigma_{\theta_3} R, \theta_1) = (R, \theta_1 \land \theta_3)\). Using this fact with the other rules, we can simplify appropriately.

4.2.2 \( E = \prod_L (E_1) \)

We now consider the simplification of expressions of the type given above. Let \( E_1 \) be c.a. on \((A_1, \theta_1)\). Clearly, we can pull up the authorization only if \( A(\theta_1) \subseteq L \cup A_1 \). Otherwise, \( E \) is not authorized (or we pull up false).

Now, using this and the other rules, it is easy to see how to simplify when \( E = E_1 \Join_\theta E_2 \). As \( R \Join_\theta S = \prod_{A(\theta)}(R \Join_{\theta} S) \), we just have to simplify using other rules the inner join expression for authorization. The final pull up will be possible only if the above rule is satisfied.

4.2.3 \( E = \sigma_\theta (E_1) \)

Clearly, in this case the authorization condition can always be pulled up. That is, if \( E_1 \) is c.a. on \((A_1, \theta_1)\) then even \( E \) is c.a. on the same pair.

Having seen all these rules enabling propagation of conditioned authorization, the next step is integrating the techniques for combining safe-plan generation with redundancy removal into a Volcano style query optimizer. The next Chapter deals with this implementation issue.
Chapter 5

Implementation

In this chapter, we will deal with integrating the various techniques for redundancy removal
and safe-plan generation in a Volcano based query optimizer [Roy00]. We first provide a
general overview of the optimizer and then we proceed to describe our additions.

5.1 Overview of the Optimizer

Volcano is a well-known top-down, cost-based optimizer generator designed for providing
efficient and extensible tools for query processing. The model achieves the separation of
the physical algorithms from the logical structure of the query. Thus, the framework is
independent of the data model or the execution model and so is extensible to new data
models, implementations etc. The original model was explained by Graefe et al. [GM93].
Roy [Roy00] gave a modified Volcano style optimizer. We would be concentrating on this
modified framework. The reader is requested to refer to the sources for more details.

Volcano is a top-down cost-based optimizer. By top-down we mean that it generates
plans top-down - i.e. it computes the best plans for only those expressions on \( n \) relations
which are included in some expression on greater than \( n \) relations being expanded. And
by cost-based we mean that it chooses the best plan it can find in terms of the statistically
estimated cost of that plan.

Volcano uses two types of algebras - logical and physical. Logical algebra corresponds
to the query model algebra, such as relational algebra. So an expression in a logical algebra
consists of logical operators such as \textit{join} and \textit{select}. Physical algebra consists of the set
of algorithms and implementations. So, for example, merge-join and nested-loops-join
algorithms are operators in this algebra. Thus, the task of the optimizer is to map a
logical algebra expression (a query) into an equivalent optimal physical algebra expression
(a query evaluation plan consisting of algorithms).

This task can be divided into three distinct steps:

\textbf{Logical Plan Space Generation} In this step we use logical transformation rules and
re-write a given query into semantically equivalent logical expressions.

\textbf{Physical Plan Space Generation} Now we use the logical space generated above and
algorithms and generate a physical plan space consisting of various execution plans
for the logical expressions.
Searching the physical plan space Given a set of plans which implement the given logical expression, we now have to choose the optimal plan based on cost estimates. This step does that using a dynamic programming algorithm coupled with pruning and memoization.

We follow the “breadth-first” ordering of these steps as described in [Roy00], in which each step is performed completely before the next step is started. It is easy to see that for implementing the relevant techniques, we need to modify only the logical plan space generation step. Hence, only this step is discussed further.

Logical Plan Space Generation In this step we repeatedly keep applying transformational rules to the query till we have completely generated the logical plan space. The given query is converted into its LQDAG form. We have seen this DAG previously in Figure (3.1b). To recall, it is an AND-OR graph with equivalence nodes being the OR nodes and logical operator nodes corresponding to the AND nodes. Now this LQDAG is expanded by applying all possible transformations on every operation node in the DAG. If an expression not already present in the LQDAG is found, it is added to the DAG. As we need to check for the presence of a logical expression in the DAG, a hash-table is used where the expressions are hashed appropriately. Note that equivalence nodes are also unified along the way whenever an expression generates an equivalent expression which is present in another equivalence node. Also note that the expansion is done in one pass, without re-visiting any node, by applying the transformations in a bottom-up topological manner. All inputs of a node are expanded first before it itself is expanded. The completeness of this procedure can also be proved [Roy00].

For example, consider the query $(A \bowtie B \bowtie C)$. Figure 5.1 shows expansion of the query (assuming only associativity rules).

5.2 Extensions to the Optimizer

Although the basic structure of the optimizer was not changed, there were significant additions made to the optimizer. A detailed definition of all the new classes is given in the Appendix.
5.2.1 Introducing Semi-Joins

First of all, we had to introduce another binary operator called Semi-Join which was not present in the original optimizer. A new class SemiJoin_t was defined which inherited from the LogicalOp_t → Filter_t hierarchy. Apart from implementing the standard virtual functions associated with any operator (such as ComputeLogicalProp etc.), several new functions were created specifically for this operator. Note that this operator is present only in the LQDAG, i.e. just before passing the final expanded LQDAG to the physical DAG expansion module, we decompose all semi-joins in the LQDAG using the following rule (assuming no duplicates): $A \bowtie_{\theta} B \equiv \prod_{a}(A \bowtie_{\theta} B)$.

5.2.2 Handling User-Defined Functions (UDFs)

The original optimizer did not contain support for handling UDFs. So, first of all we extended the parser grammar to accept UDF expressions. A UDF, containing a name and parameter list, reduces to the primary expression non-terminal (expr_primary), hence allowing it to be present in any valid expression. The non-terminals param_list and param were added as follows:

```plaintext
param_list  : param_list ',' param
    {
      $1->put($3);
      $$ = $1;
    }
    | param
    {
      $$ = new ExprList;
      $$->put($1);
    }
    ;

dparam  : expr
    {
      $$ = $1;
    }
    ;
```

Having extended the grammar, we also had to extend the backend code to the parser by creating a new class corresponding to a UDF - UDFExpr. It inherits from the usual Expr class. Also, relevant changes were propagated to the interface code by implementing the postorder traversal of the UDF expression for conversion to the DAG structure. This postorder traversal creates an object of UDFExpr_t class which inherits from the standard DAG expression class (Expr_t). Associated with each UDF object (in the DAG), there is a function IsUnsafe which determines whether the UDF is safe or not. In addition, we had to extend the IsEquivalent and IsValidSchema functions to take UDFs into account.
5.2.3 Handling View Replacement

We have to rewrite the input query using the respective authorized views for each relation. Hence, for this purpose, we take a simple ASCII text file as input which simply lists the authorized view for each relation. If there is no view defined in the file, it is assumed no authorization is needed on that relation. The file is processed just before parsing of the user query and the views are stored in memory as a list of *RelView* objects. The syntax of the file is as follows:

- number of views in the file
- name of relation
- number of semi-joins in its authorized view
- the first authorization semi-join expression
- the first authorization semi-join predicate (embedded in a dummy query)
- the second authorization semi-join expression
  ...
  ...

A sample file snippet would be:

```
4
emp
1
select dep.empid from dep where dep.depid < 26
select xyz.xy from xyz, emp, dep where emp.empid = dep.empid
...
```

Just to reiterate, the file specifies that *emp* is semi-join authorized on `select dep.empid from dep where dep.depid < 26` with semi-join predicate as `emp.empid = dep.empid` and so on.

Now given the respective views, we rewrite the given query in the postorder traversal of the parsed structure itself by replacing a relation by a corresponding semi-join with the authorization. Another point is that the *memo* object used for this phase is different than that of rest of the modules. This is being done to eliminate needless checking and recursive re-writings. Finally, the query DAG with view-authorization checks in place is generated which is carried forward to other modules.

5.2.4 Subsumption Checks

Note that we check for subsumptions only for the cases where the SPJ (Select-Project-Join) representation of the two input expressions of a semi-join exists. This is because the SPJ representation allows for efficient checking of subsumption. Hence, associated with each equivalence node there is an SPJ representation called a *ViewRep* (*vR*). It stores the list of relation nodes which are being joined in the expression, the projection list and the select predicate of the final representation. Of course, code was written for computing the *vR* for
any equivalence node which required code for computing the \( vR \) for any logical operator
node following the standard rules. For e.g. for a select node the rule would be:

\[
vR(\sigma_\theta A) = \{ \text{SelectPredicate}(vR(A)) \wedge \theta, \text{ProjectList}(vR(A)), \text{RelationsList}(vR(A)) \}
\]

and so on for other operators. After that we also defined a static SubsumptionCheck
function of Equivalence_t class which checks for subsumption between equivalence nodes. The conditions being checked for subsumption of \( A \) by \( B \) are:

- \( \text{RelationsList}(vR(A)) \supseteq \text{RelationsList}(vR(B)) \)
- Using notation introduced in the previous chapter:

\[
\pi_{\text{RelationsList}(vR(B))}(\text{SelectPredicate}(vR(A))) \Rightarrow \text{SelectPredicate}(vR(B))
\]

On comparing with the sufficient conditions listed in [KRS06], we can see that we additionally assume that the semi-join predicate always equates the necessary attributes i.e. it is an equi-semijoin. We believe that this is a reasonable assumption for real-life queries.

5.2.5 Extensions to the Transformation Rules-Set

As previously mentioned, each operator has a set of transformational rules which are applied exhaustively in the expansion phase. Due to the presence of semi-joins, we extended the rules-set to include the following, broadly speaking, semi-join pull-up rules:

- \( A \bowtie (B \bowtie C) \equiv (A \bowtie B) \bowtie C \)
- \( \sigma(A \bowtie C) \equiv \sigma(A) \bowtie C \)
- \( \prod_L(A \bowtie_\theta C) \equiv (\prod_L A) \bowtie_\theta C, \text{ if } A(\theta) \subseteq L \)

In addition, we added a semi-join subsumption rule:

- \( A \bowtie_\theta B \equiv \sigma_{\text{true}} A, \text{ if } B \text{ subsumes } A \)

Note that we had to introduce a \( \sigma_{\text{true}} \) because of the LQDAG structure where the parent of an equivalence node has to be a logical operator node. Moreover, as this is an equivalence rule, we should not unify the old and new equivalence nodes i.e. the node representing \( A \bowtie_\theta B \) and \( A \). But unfortunately due to this rule, coupled with the rules-set already present in the optimizer, a cycle may be introduced in the DAG. To see this, consider the situation as depicted in Figure 5.2.

In Figure 5.2(a), we have applied the simplification rule on the semi-join between classes \( C \) and \( D \). As a result we get the \( \sigma_{\text{true}} \) operator under class \( B \). After this, the Select-Push-Predicate rule is applied and we get the new expression \( \sigma_{\alpha}(C) \) under class \( A \). Now, we can clearly see that the Semi-Join Pull-up rule can not be applied on \( \sigma_{\alpha} \) and the semi-join below it as this will introduce a cycle in the DAG - the parent of the new semi-join would have been the same as its first input! Hence, to avoid this problem, we turn-off the application of any pull-up rule if we detect such a situation. Note that this does not prevent a favourable plan from being discovered. This is so because of the fact that if both the equivalence classes representing the first input and the parent are the same for a semi-join, it implies that subsumption would anyway have taken place had the semi-join was actually in place. Hence we do not lose out on any possible plan.
5.2.6 Redundancy Removal

We will now touch upon extensions specifically for redundancy removal. Recall that in conditioned authorization we can do redundancy removal before the transformation phase. And for making this more efficient, [KRS06] have suggested the pull-up of semi-joins. Note that this phase is pretty much the same as the semi-join pull-up rules (Section 5.2.5), except one important difference. In Section 5.2.5 the rules were equivalence rules, hence the old form was still retained - the new operator just became another child of the equivalence node. On the other hand, now we have to discard the old form and retain only the pulled-up form. We apply the following skeleton algorithm for doing this:

Algorithm Semi-Join Pull-up
---------------------------
flag = true;
while(flag) {
    op = next logical operator child;
    // check for the feasibility of an immediate
    // semi-join pull-up
    // operator dependent according to standard rules
    feasible = op->PullupSJFeasibility();
    if(feasible!=-1) {

Figure 5.2: Example of Cycle Generation
/* we can immediately pull-up a semi-join
   hence, now
   1. Create the new semi-join operator
   2. Delete op (the old operator)
*/
} else {
  // no immediate pullup, hence recurse for searching
  flag = op->PullupSJ();
}

Thus, this algorithm keeps on pulling-up semi-joins until no further pull-up is possible.

The second issue is that of the actual redundancy removal. Again, compare this with
the semi-join subsumption rule in Section 5.2.5. Here, we need to unify the old and new
equivalence nodes as this is a simplification rule. So, $A \times B \rightarrow A$. Both the semi-join
pull-up and semi-join simplify are done as separate passes over the entire initial query DAG
produced after the view based re-writing) before the transformation phase.

## 5.2.7 Safe-Plan Generation

We discuss here extensions for generation of safe-plans. Firstly, all the transformational
rules were modified such that no predicates containing Unsafe UDFs (USFs) are pushed
on non-authorized equivalence nodes. Determining which nodes are “authorized” is de-
pendent on which notion of authorization we are using. To enable such a check, we define
a IsReadyForUSF() function which determines whether a given predicate can be placed
over a given equivalence node. Recall that in conditioned authorization, an equivalence
node is unconditionally authorized only when the conditional authorization is EMPTY. On
the other hand, in normal authorization, a simple boolean marking of the node results in
unconditional authorization.

\[
\text{IsReadyForUSF(Predicate } p, \text{ Equivalence } eq) \{ \\
  \text{ // checks whether } p \text{ has a USF} \\
  \quad \text{flag} = p->\text{ContainsUSF}(); \\
  \text{ if( conditioned Auth. is enabled) } \\
  \quad \quad \text{return !flag || (flag && (eq->CondAuth.IsEmpty()));} \\
  \text{ else } \\
  \quad \quad \text{return !flag || (flag && (eq->NormAuth == 1));}
\}
\]

## 5.2.8 Propagation of Authorization

Finally, we deal with code for propagation of respective authorizations. Note that we need
to fire a new computation of any authorization (conditioned or normal) whenever a new
logical operator is made a child of an equivalence node. This is so because it is only now
that the authorization status of the equivalence node can change - we may or may not be able to infer from the new operator node new authorization for the node. If the status does change, we simply need to propagate the changes to its ancestors. As we maintain the parents’ list of an equivalence node, this is easy to do. Hence, we plug the inference check and propagation in the AddLogExprs() function of an equivalence node object. The exact inference check and propagation are dependent on which notion we use. For conditioned authorization, the rules have already been discussed in the previous chapter. For normal authorization, we simply use the validity propagation rule of Chapter 3. In both the cases however, care has to be taken so that propagations do not take place in infinite recursion and propagation travels up the DAG reaching each relevant node exactly once.

This concludes a brief description of the additions made to the optimizer for enabling redundancy removal and safe-plan generation. To summarize, Figure 5.3 shows the execution flow in case normal authorization is enabled. Figure 5.4 shows the flow when conditioned authorization is set.
Figure 5.4: **Execution Flow in case of Conditioned Authorization**
Chapter 6

Experiments

We ran several experiments to measure the performance characteristics of using Conditioned Authorization and Normal Authorization for generation of safe plans. Recall that in Normal Authorization, we do redundancy removal during the transformation phase while in Conditioned Authorization, it is done during the simplification stage itself. Also in Normal Authorization, we use the validity propagation (Chapter 3) rule to infer authorization while in Conditioned Authorization, we use the rules developed in Chapter 4.

We consider the following parameters while measuring performance:

1. **Optimization Time**: This is the total time spent by the optimizer before it outputs the optimal plan. Note that for conditioned authorization it also includes the time to do the initial semi-join pull-ups and simplifications.

2. **Size of the Final LQDAG**: We measure the size by simply counting the number of nodes (equivalence classes and logical operators) in the final expanded DAG.

3. **Cost of the Optimal Plan Chosen**: This is the cost of the final optimal physical query plan as output by the optimizer. Recall that we do not change anything in the physical DAG generation module of Roy’s [Roy00] optimizer.

6.1 Experimental Setup

The code for both the algorithms was in C++ (with standard STL). All experiments were performed on dual Intel Xeon 3 GHz Processors with EM64T, 4 GB RAM, 2 X 300 GB SATA in RAID-1 hard disk running Debian Linux. The experiments were conducted on the following queries (in all the cases, `usf` denotes an arbitrary unsafe user-defined function returning numeric values):

- **Q1**:

  ```sql
  SELECT emp.ename, dep.empid
  FROM emp, dep
  WHERE emp.empid = dep.empid AND
        dep.depid < 20 AND usf(emp.empid) < 24
  ```
The authorization view for *emp* was defined as:

```
CREATE AUTHORIZATION VIEW A1 AS
SELECT * FROM emp
WHERE emp.empid IN (SELECT dep.empid FROM dep
                      WHERE dep.depid <$user-dept)
```

We instantiate $A1$ with $\$user-dept = 26$. Semantically, the view restricts each user-employee to see information for only those employees whose department-ids are less than that of the user.

- **Q2**: This query is based on the standard TPC-H schema.

```
SELECT supplier.s_suppkey
FROM supplier, lineitem, orders, customer, nation, region
WHERE region.r_name <= 3 AND
  nation.n_regionkey = region.r_regionkey AND
  customer.c_nationkey = nation.n_nationkey AND
  orders.o_custkey = customer.c_custkey AND
  lineitem.l_orderkey = orders.o_orderkey AND
  supplier.s_suppkey = lineitem.l_suppkey AND
  usf(orders.o_totalprice) < 25
```

As in [KRS06], we define the following views ($\theta = region.r_name <= 3$):

- **authNation**: nation $\bowtie$ $\sigma_\theta$region
- **authCustomer**: customer $\bowtie$ nation $\bowtie$ $\sigma_\theta$region
- **authOrders**: orders $\bowtie$ customer $\bowtie$ nation $\bowtie$ $\sigma_\theta$region
- **authLineitem**: lineitem $\bowtie$ orders $\bowtie$ customer $\bowtie$ nation $\bowtie$ $\sigma_\theta$region
- **authSupplier**: supplier $\bowtie$ lineitem $\bowtie$ orders $\bowtie$ customer $\bowtie$ nation $\bowtie$ $\sigma_\theta$region

To see how the parameters mentioned before vary, we ran Q1 with different sizes of the *emp* table. We also ran Q2, Q3 and Q4, which were the TPC-H schema based queries. Q3 was similar to Q2 but was a join of lineitem, orders, customer, nation, region only. In the same vein, query Q4 was a join of orders, customer, nation, region). Recall that we had already modified the rules-set for safety. Hence, both these methods always generated safe-plans during the experiments.

### 6.2 Varying Data Sizes

Figure 6.1 shows how the plan cost varies with increasing size of the *emp* table. Clearly the authorization view is redundant in the query; indeed the simplification stage of conditioned authorization simplifies the query to a simple join. In this case, the entire plan space generated during conditioned authorization is also being generated during normal authorization.
Hence, same plans are chosen by both the methods. However, a slight difference in the cost is observed - because of the extra $\sigma_{true}$ being introduced during normal authorization. We also observed significant (about 1.6 times) difference in the final LQDAG sizes. This of course remains constant with change in data size. The optimization time for normal authorization is also larger - clearly as it has to deal with the extra logical plan space.

### 6.3 Varying Query Sizes

Figures 6.2, 6.3 and 6.4 show how the parameters vary with decreasing size of the TPC-H schema based queries i.e. w.r.t. the queries Q2, Q3 and Q4. Firstly, note that these queries contain a large amount of redundancy. Infact, all of the semi-joins introduced by authorization views are redundant. The simplification phase of conditioned authorization is also able to remove them. Hence, the difference between the sizes of the DAGs generated by the two techniques is very large. In particular we can see that for Q2, the difference is as large as 4.5 times. Analogously, optimization times also follow the same pattern. In fact, shapes of graphs for both these parameters are almost the same - expected as they are correlated.

We also observe (Figure 6.3) that the estimated plan costs are almost equal for these queries. As in the previous section, this is because both normal and conditioned authorization techniques are able to detect the redundancies and also infer authorization equivalently. The rules developed in Chapter 4 are adequate here to infer authorization so as to generate the optimal safe plan. However, we might see differences when the conditioned authorization rules are not powerful enough to deduce an equivalence node to be authorized but normal authorization would allow us to infer the same.

But surprisingly, there is a noticeable difference in the plan costs for Q2. In this case,
normal authorization is not able to detect a redundancy. This can be explained if we look at the semi-join pull-up rules. Recall that we modified all rules for safe-plan generation such that predicates containing USF’s are not pushed on unauthorized expressions. So, $A \bowtie_p (B \bowtie C) \equiv (A \bowtie_p B) \bowtie C$ would fail if predicate $p$ is unsafe for $B$. As a result, in this particular case, we are not able to pull-up that semi-join. Hence, as a side-effect, we are in turn unable to deduce subsumption between $(A \bowtie_p B)$ and $C$, due to which this semi-join is not detected to be redundant. Note that in case of conditioned authorization, we pull-up all the semi-joins irrespective of these safety concerns in the simplification stage itself. Hence, while we are able to remove redundancies in conditioned authorization, we are unable to do so in normal authorization.

6.4 Comparison with No Redundancy Removal

Finally as a sanity check, we also optimized these queries (Q2, Q3, Q4) without using any redundancy removal techniques. But as safe plan generation is not just desired but rather required, we kept the safety checks in place. The best plan cost (for Q4) is around 1.3 times the cost of the plan produced by optimization with redundancy removal. As already seen in [KRS06], redundancy removal indeed (both NA/CA) gives real gains in final query plan costs.

Another observation (see Figures 6.5 and 6.6) was that the DAG size as well as optimization time in case of optimizing without any redundancy removal was smaller as compared to normal authorization. This can be explained if we remember that we have in effect removed a transformation rule (Semi-Join Subsumption) from the rules-set - hence the logical space explored is larger. This is undesirable as it imposes an overhead on optimization. On the other hand, conditioned authorization in fact reduces optimization time.
Figure 6.3: Varying Query Size (B) - Plan Cost

Figure 6.4: Varying Query Size (B) - Optimization Time
Figure 6.5: Comparison with No Redundancy Removal (A) - Optimization Time

Figure 6.6: Comparison with No Redundancy Removal (B) - DAG Size
as the query after simplification is generally much smaller than the original query. This is a benign characteristic as it allows us to get better and faster plans and at the same time reduce optimization time. Recall that in case of normal authorization, we do not simplify the query initially at all and hence supply the original query itself for DAG expansion.

All these results clearly prove what we expected theoretically; both about benefits of redundancy removal in general and conditioned authorization in particular. To conclude, Conditioned Authorization is the superior method for integrating safe-plan generation and redundancy removal. The key feature of conditioned authorization which makes it more efficient is that we perform redundancy removal during simplification stage itself - thus giving a smaller query to the logical transformation phase. It takes lesser optimization time, removes greater redundancies and takes less memory resources to run.
Chapter 7

Conclusions and Future Work

To summarize, relational databases have recognized the importance of providing Fine-Grained Access Control (FGAC) in a principled and efficient manner. There are broadly two completely different approaches for providing FGAC - the Truman Model (or the view-replacement model) and the Non-Truman Model [RMS04]. [KRS06] investigated two niggling issues present in all Truman models - redundancy in re-written queries and susceptibility of the model to information leakage through channels other than that of query result itself. They characterized redundancy in terms of detection of redundant semi-joins and also classified individual query plans as safe or unsafe depending on the placement of Unsafe UDFs in the plan. They also gave a method for generating safe-plans in which one needs to infer the authorization state of expressions. For this purpose, they use the validity propagation rule which is hampered by the removal of authorization views in the simplification stage. Hence, to overcome this problem, [KRS06] suggested delaying simplifications to the transformation stage (Normal Authorization) so that redundancy removal and safe-plan generation can take place together with maximum benefit.

Normal Authorization suffers from the drawback that the search space blows up due to the initial non-simplification of the query. A more elegant solution namely Conditioned Authorization was briefly motivated in [KRS06]. In conditioned authorization, we do simplifications as before before transformation. But as the original authorized views may have been deleted, we define expressions to be conditionally authorized on these original views. The intuition is that we might still be able to infer an expression to be authorized even though the original views have been deleted. Note that we now need a set of propagation rules which enable us to infer unconditionally authorized expressions. We analyzed this concept more deeply and formalized it by giving a number of propagation rules. The rules used structure on authorized views to simplify propagation of conditioned authorization. After this, we implemented the algorithms on a Volcano style query optimizer. Although the basic structure was not changed, the extensions were significant. We also saw how to handle some tricky implementation issues such as cycle generation during transformations etc. Finally, we did a performance study comparing the two approaches by testing them on a number of queries. It confirmed the fact that conditioned authorization is clearly a superior approach then normal authorization for integration of redundancy removal techniques with safe-plan generation. Both the expanded DAG sizes and the optimization overhead in conditioned authorization are very small compared to both normal authorization and optimization without redundancy removal. It is also able to detect a larger number of
redundancies and at the same time we saw that its power to infer authorization is as good as normal authorization. Thus, to conclude we observed that conditioned authorization takes lesser optimization time, lesser memory resources to run and produces better plans.

As part of future work, we realize that the propagation and simplification rules for conditioned authorization are not complete. We can extend them to handle more operators and also more complex structures of the authorization views. We would also want to handle duplicate tuples in relations as allowed by the SQL standard. Also, recall that [KRS06] base their solutions on a specific form of authorization views. We can extend these techniques for optimizing in presence of other types of authorization views too which can not be reduced to simple semi-joins - these include views which aggregate information from base relations etc. In addition, we can work on integrating these methods with keyword searching algorithms for databases. In systems such as [SJ01], the user keyword query is translated to full-fledged SQL queries which run on the DBMS. The results of these SQL queries are then translated back and presented to the user. Integration of redundancy removal and safe-plan generation techniques with such systems might provide efficient security against possible information leaks through the intermediate SQL queries. Although it is not clear at present about how this can be done or if it is even applicable, the idea itself of integrating FGAC with keyword based querying is very attractive.
Bibliography


Appendix

We will here provide the definition of the new classes that have been created.

**Class UDFExpr**

class UDFExpr : public Expr
{
    public:
        char *fname; // the function name
        ExprList* paramList; // the parameter list
    private:
        void Constructor(char* name_a, ExprList* list_a);
    public:
        static UDFExpr *create(char* name, ExprList* list);
        UDFExpr *Copy();
        bool CheckLocalArgBounds(int lowerBound_a, int upperBound_a);
        void ShiftLocalArgs(int howMuchToShift_a);
        ArgType typeCheck(TupleDef &bindings);
        int print(FILE * file);
        Expr *rewrite(TupleDef &bindings_a, ArgSource source_a);
        UDFExpr *Substitute(Expr **exprList);
        static int IsEquivalent(UDFExpr*, UDFExpr*);
};

**Class UDFExpr_t**

class UDFExpr_t : public Expr_t {
    char *fname;
    Expr_t** paramList;
    int noParams;

    public:
        UDFExpr_t(char* name, int number);
        UDFExpr_t(char* name, Expr_t** list, int number);
        char* FName(void) const;
    };
Expr_t** ParamList(void) const;
Expr_t* Param(int i) const;
int NoParams(void) const;
Expr_t *Copy(void) const;
void PrintExpr(int isFullExp = 0) const;
int IsEquivalent(const Expr_t *e) const;

// is it a USF?
int ContainsUSF(void);

bool IsUnsafe(void) const;
~UDFExpr_t(void) {}
void Print(void);
};

Class RelViews_t
class RelViews_t {
  char* relName;
  Predicate_t* pred;
  Equivalence_t* eq;
public:
  RelViews_t(char* name, Equivalence_t* e, Predicate_t* p);
  RelViews_t();
  char* Name(void) const;
  Equivalence_t* EqNode(void) const;
  Predicate_t* Predicate(void) const;
  ~RelViews_t();
};

Class SemiJoin_t
Note that all the functions which are used in the physical DAG are just dummy functions
doing nothing as this operator is used only till the logical plan expansion stage.
class SemiJoin_t : public Filter_t {
  LogProp_t *ComputeLogicalProp(void) const;
  char isTransDisabled;
public:
  SemiJoin_t(Predicate_t *pred_a);
  LogicalOp_t *PushInPred(Predicate_t *inPred, Memo_t& memo)
  {return NULL;}
  LogicalOp_t *PushInProj(Project_t *inProj, Memo_t& memo)
  {return NULL;}
  void ApplyAlgorithms(PlanGroup_t *planGroup, Volcano_t *opt,
                        Cost_t *totalCost, Plan_t *bestPlan, CostVal_t *costLimit) { }
}
// apply the transformations
void ApplyTransformations(Memo_t& memo);

// compute View rep
ViewRep_t* ComputeViewRep(void);

// decomposeSJ
int DecomposeSJ(Memo_t& memo);

// simplifies redundancies
Equivalence_t* Simplify(Memo_t& memo);

// creates a new copy for pullupSJ with the appropriate
// input changed
Equivalence_t* CreateCopyForPullupSJ(int index, Equivalence_t* eq, Memo_t& memo);

// propagates the search for a suitable SJ for pulling up
bool PullupSJ(Memo_t& memo);

// checks whether an *immediate* pullup of a semijoin is possible
int PullupSJFeasibility(void);

// lower bound on any algorithm corresponding to this logop
CostVal_t CostLB(void) const {return 0;}

void PrintName(void) const;

// to avoid generating duplicate expressions
void DisableTrans(void) { isTransDisabled = 1; }
int IsTransDisabled(void) { return isTransDisabled; }

~SemiJoin_t(void) { }
};