Redundancy and Information Leakage in Fine-Grained Access Control

Govind Kabra [1]
Univ. of Illinois, Urbana-Champaign
gkabra2@uiuc.edu

Ravishankar Ramamurthy
Microsoft Research
ravirama@microsoft.com

S. Sudarshan [2]
I.I.T. Bombay
sudarsha@cse.iitb.ac.in

ABSTRACT

The current SQL standard for access control is coarse grained, in that it grants access to all rows of a table or none. Fine-grained access control, which allows control of access at the granularity of individual rows, and to specific columns within those rows, is required in practically all database applications. There are several models for fine grained access control, but the majority of them follow a view replacement strategy. There are two significant problems with most implementations of the view replacement model, namely (a) the unnecessary overhead of the access control predicates when they are redundant and (b) the potential of information leakage through channels such as user-defined functions, and operations that cause exceptions and error messages. We first propose techniques for redundancy removal. We then define when a query plan is safe with respect to UDFs and other unsafe functions, and propose techniques to generate safe query plans. We have prototyped redundancy removal and safe UDF pushdown on the Microsoft SQL Server query optimizer, and present a preliminary performance study.

1. INTRODUCTION

The current SQL standard for access control is coarse-grained, in that it grants access to all rows of a table or none at all. Fine-grained access control, which allows control of access at the granularity of individual rows/columns, is required in practically all database applications, for example to ensure that employees can see only their own data, and relevant data of other employees that they manage. Fine-grained access control has traditionally been performed at the level of application programs. However, implementing security at the application level makes management of authorization quite difficult, in addition to presenting a large surface area for attackers --- any breach of security at the application level exposes the entire database to damage, since every part of the application has complete access to the data belonging to every application user. There is, therefore, an increasing need to support fine-grained access control at the database level.

Several models for fine-grained access control have been proposed in the recent past [8][9][10][11][12]; two of these, Oracle's Virtual Private Database (VPD) [9], and Sybase's row level security model [10], are implemented as part of commercial database systems. With the exception of [8] and [11], all these models, in effect, replace each relation in a user query by a view of the relation that the user is authorized to see; the replacement view includes authorization checks. More details about these models are provided in Section 2.

There are two significant problems that must be addressed by any implementation of fine-grained access control using the view replacement model:

1. The first issue is a question of efficiency: The original query usually includes predicates/joins that restrict access to only authorized data. The authorization checks performed in the replacement views are often redundant, including not only cheap comparisons, but also expensive semi-joins. The introduction of complex authorizations also increases the complexity of the rewritten query significantly, resulting in increased optimization time and worsened execution time.
2. The second issue is a question of effectiveness of the access control implementation under a general model of attack, which may include users submitting arbitrary SQL queries. Although applications generally do not allow users to submit arbitrary SQL queries, hackers who break into an application may be able to submit such queries. Further, in a hosted application, a single database schema may be shared by a large number of independent users who may be authorized to submit SQL queries on their views of the data.
   Even if the fine-grained authorization implementation ensures that the query result contains only authorized information, it is possible for malicious users to gain access to unauthorized information based on *information leakage* through channels other than the query result. These channels include:
   i. Leakage of information through exceptions and error messages.
   ii. Leakage of information through user-defined functions (UDFs). In this case, the malicious user needs the ability to define user defined functions in addition to the ability to run SQL queries.

As an example of leakage through UDFs, consider a relational schema employee(emp_id, name, dept_id, salary) and a view

```
CREATE VIEW myemployees AS
SELECT * FROM employee
WHERE dept_id in Q1
```

which allows a user to see only the employee tuples with dept_id in the result of some query Q1.

[1] Work done during a summer internship at Microsoft Research
[2] Work done during a sabbatical at Microsoft Research

SIGMOD 2006, June 27-29, 2006, Chicago, Illinois, USA.

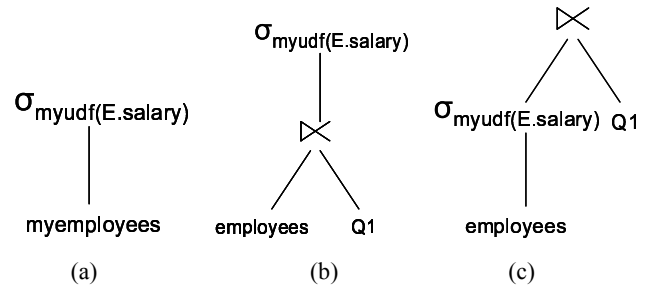

Figure 1: (a) User query on an authorized view. (b) A safe plan for this query, and (c) A result-equivalent plan that can potentially leak unauthorized information.

Suppose the user issues the following query:

```
SELECT * FROM myemployees
WHERE myudf(salary)
```

(Or, equivalently, suppose the user writes the query on the relation employees, which is rewritten by replacing the relation by the authorized view myemployees.) Figure 1(a) shows the original user query. The query preprocessor replaces the view myemployees by its definition. Figure 1(b) and 1(c) illustrate two alternative plans for the rewritten query. In the plan in Figure 1(b), every tuple for which myudf is executed has passed through the authorization check of subquery Q1. However, in the plan in Figure 1(c), myudf is executed on every tuple in employee relation, even those that would not pass the authorization check of subquery Q1.

The behavior of myudf is not under the control of the authorization system, and it can potentially print out (or save in a separate relation) the salary values it is passed. While both the plans produce same result set, the plan in Figure 1(c) can potentially leak the salary information of all the tuples in employee relation.

Such leakage can also occur with system defined functions that throw exceptions, or display error messages. We use the term *unsafe function (USF)* to denote user-defined functions, system-defined functions and operators that may reveal information, whether directly or by indirect means such as exceptions, error messages or timing. We provide more concrete examples of these leakage channels in Section 5.1. For any fine-grained authorization implementation to be effective, it has to protect against information leakage.

One possible solution is to pull up USFs to the top of query plans, thereby ensuring that they are executed only on data that the user is authorized to see. While correct, this solution can have poor performance in many situations, especially if the USF is used in a predicate that allows very few tuples out. Sandboxing is another standard technique to protect database systems from potentially harmful effects of UDFs. However, sandboxing cannot handle some leakage channels, and has other limitations, as discussed in Section 5.1.

In this paper we address the problems of redundancy and information leakage. The contributions of this paper are as follows:

- We describe how redundancy checks can be integrated into a rule based optimizer with low overheads. We note that algorithms for detecting redundant joins have been available from the early days of databases; our contribution is in engineering an existing optimizer to introduce redundancy detection, exploiting existing functionality that supports materialized view matching.
- We target the problem of information leakage as follows:
  - We first explore what plans can be guaranteed to not leak information, and define a class of safe plans that are guaranteed to not leak information through unsafe functions.
  - We then describe techniques to find the optimal safe plan. In our performance study, we demonstrate the benefits of finding optimal safe plans, instead of settling for heuristics such as pulling USFs to the highest level in a query plan.

The rest of the paper is organized as follows. Section 2 describes related work, while Section 3 presents a query rewriting model which we use to describe our techniques. Section 4 covers removal of redundant semijoins. In Section 5 we formally define when a query plan is safe in the presence of USFs, and describe how to extend a query optimizer to find an optimal safe plan. The above algorithms have been prototyped on Microsoft SQL Server, and in Section 6, we present a performance study. In Section 7 we present our conclusions and directions for future work.

2. RELATED WORK

The basic goal of fine-grained access control is to grant a particular user access to only specific columns of a subset of the tuples in a relation. In contrast, although the current SQL standard supports column level authorization, it does not provide any way of giving different authorizations to different rows.

Oracle's Virtual Private Database (VPD) model [9] supports fine-grained access control through functions that return strings containing predicates. A function is associated with each relation, and when invoked returns a string containing predicates that enforce fine-grained access control; the function takes as input the mode of access and an application context which includes information such as user-id of the end user.

The policy based security management feature of Sybase Adaptive Server Enterprise [10] allows the specification of predicates that are added to where clauses. Different policies can be specified on different columns, and are automatically combined. The idea of access control by query modification dates back to Ingres [13].

Cell-level access control is described by LeFevre et al. [4]. In their model, values of particular columns of particular tuples may be replaced by null, if the user is not authorized to see those values. Their target application is the handling of privacy policies, not general purpose authorization. However, their approach of nullification is useful in a general purpose authorization model. A proposal to use predicated grants to manage cell-level access control is described by Agrawal et al. [1].

All the above models fall into the class of "Truman models", in the terminology of [11]. The non-Truman model described in [11] has a different approach: under this model, a query is valid if it can be rewritten using only authorized views (there are two notions of validity, unconditional and conditional: see [11] for details). Queries submitted to the system are checked for validity; if a query is valid it is executed with no modifications, otherwise it is rejected. Such behavior matches the authorization checking in standard SQL, but works with fine-grained authorizations specified by authorized views. The Non-Truman model is attractive for several

reasons, such as guaranteeing correctness; that is, if a query is accepted, it will give the same result as if the user had full authorizations on all relations. In contrast, in the class of Truman models, the result of a query can be changed by the authorization mechanism.

However, any non-Truman model implementation is likely to be unpredictable in the following sense: the model requires a powerful query inferencing mechanism and since inferencing can never be complete, a query that is accepted by one database implementation may be rejected by another (perhaps even a different version of the same database system). Such unpredictability is highly undesirable for applications, and inference procedures are expensive and far from complete; as a result the class of Truman models is used in practice, in preference to the non-Truman model.

Brodsky et al. [2] provide a survey of secure databases, which includes coverage of inference channels. Attacks on database applications using SQL injection coupled with inferencing using exceptions and error messages are well known in the hacker community; see for example, Litchfield [7]. These attacks were designed to subvert application level security, not fine-grained access control in the database, but as our examples in Section 5.1 show, exceptions and error messages can easily be used to leak information in the context of fine-grained access control.

As far as we are aware, the problem of leakage through exceptions and error messages have not been addressed in the past. We are also not aware of any published work on ensuring safety with respect to UDFs.

3. QUERY REWRITING MODEL

The techniques we describe in this paper are applicable to authorization models that are (conceptually) based on query rewriting, where references to database relations are replaced by references to corresponding authorized views; these include [1][4][9][10].

Consider the TPC-H schema, and a user who is only authorized to see orders placed by that user. Such an authorization can be specified by the following authorized view on the relation Orders.

```
CREATE VIEW authOrders AS
SELECT * FROM Orders
WHERE (o_custkey = userId())
```

Here, userId() is a function that (at runtime) returns the user identifier of the current user. Note that applications built on top of databases often have thousands to millions of users; since databases cannot efficiently support that many users, application programs usually run under a single database user-id, and maintain their own notion of application users. The userId() function above returns the user identifier at the application level, which must be provided to the database system by the application. (In Oracle VPD, the application context provides information about the application user, as well as other application level information.)

In general, the authorized view can contain a subquery. For instance, the following view grants a user access to the Lineitem tuples that the user has ordered.

```
CREATE VIEW authLineitem AS
SELECT  * FROM Lineitem
WHERE EXISTS (SELECT * FROM Orders
                    WHERE l_orderkey = o_orderkey
                        AND o_custkey = userId( ) )
```

Although an authorized view can be arbitrarily complex, we believe the most useful class of authorized views is one where the authorized views return a subset of the tuples of a relation, where the subset is defined by a predicate (which may include a subquery). The view may additionally project away some columns.

In addition a view may replace a column value by some function. For example, in cell-level access control using nullification [4], in the SELECT clause of a view on relation R, a column A1 may be replaced by the expression

(CASE WHEN P1 THEN A1 ELSE null) AS A1

The AS clause above ensures the name A1 is retained for the result of the expression.

In this paper we consider a class of authorized views of the form

```
CREATE VIEW auth_Ri AS
SELECT Li  FROM  Ri WHERE Pi
```

where Pi may include subqueries, and Li is a list of attribute names or expressions, or may be * (to allow access to all columns). For simplicity, our presentation of redundancy removal techniques assumes that Li is just *, although we briefly outline how to handle the case where Li contains expressions implementing cell-level access control. The case where Li contains only a subset of attributes of Ri is easier to handle: it is trivial to check if a query accesses a column not in Li, and to reject such a query. For simplicity, we do not consider such projections hereafter.

Authorized views of the above form, with Li being * can be represented as a semi-join ($Ri \ltimes_{\theta i} Ai$), where Ai is an expression containing the subqueries in Pi. For notational convenience, we assume that selection conditions in Pi are folded into the semi-join condition θi.

Given a query, the access control component of the database system replaces the relations with the definitions of the corresponding authorized views. Thus, a query of the form ($R1 \bowtie R2 \bowtie \ldots \bowtie Rn$) would get rewritten to the form (($R1 \ltimes_{\theta 1} A1$) $\bowtie$ ($R2 \ltimes_{\theta 2} A2$) $\bowtie$ $\ldots$ ($Rn \ltimes_{\theta n} An$)).

Although we use the syntax of authorized views, the problems we describe as well as the solutions we present in this paper apply equally to these alternative models:

- The Oracle VPD model, where for each Ri in a query, an authorization predicate Pi is added to the where clause of the query. This model is equivalent to defining authorized views where Li in the SELECT clause is "*".
- The use of views, where the user query references the views instead of the actual relation (as we did in the examples in Section 1). This is the traditional SQL authorization model, except that the views may use application context information such as userId().
- Models based on authorization grants, where the grant may have a grant predicate, such as [1]. Such grants can be translated into authorized views.

The problems of redundancy and information leakage arise with all these approaches, and our techniques for redundancy removal and safe plan generation are applicable to these approaches.

A user can have different authorized views for different operations on the relations, such as select, insert, delete or update. In this paper we concentrate on the select authorization, and on queries,

since these present the main challenges in terms of redundancy and information leakage.

We note, however, that the approach of using authorized views to implement fine-grained access control can handle inserts, updates and deletes by defining corresponding authorized views for each operation; the views are of the same form as we use for the select operation, with Li restricted to being "*". As in Oracle VPD, such authorizations can be handled by ensuring that Pi is satisfied by the old and new values of updated tuples, and inserted and deleted tuples. Relations used in subqueries of an insert, delete or update statement are subject to view replacement using the authorized view for the select operation. With this model, updates, inserts and deletes do not introduce any additional issues of redundancy and information leakage, beyond those introduced by select queries, and we ignore them in the remainder of this paper.

4. REDUNDANCY REMOVAL

In the view replacement approach, the base relations in a query submitted by a user are replaced by authorized views. The original query usually includes predicates/joins that ensure that the query accesses only authorized tuples. In such cases the additional authorization checks introduced by view replacement would be redundant. In particular, checks which involve semi-joins can be quite expensive, and should be removed if they are redundant. In this section, we study how such redundancy can be removed by leveraging the existing view matching infrastructure of the query optimizer.

4.1 Motivating Example

We first outline an example to motivate redundancy removal. Consider the TPCH schema and assume the following authorized view for the Lineitem table which authorizes a customer to see only the lineitems corresponding to an order placed by him.

```
CREATE VIEW authLineitem AS
SELECT * FROM Lineitem   WHERE
      EXISTS (SELECT * FROM Orders
            WHERE l_orderkey = o_orderkey
                  AND o_custkey = userId())
```

Suppose a customer with userId = 123 issues the following query:

```
SELECT  Lineitem.*  FROM  Lineitem, Orders
WHERE  l_orderkey = o_orderkey AND o_custkey = '123'
```

Fine-grained access control would replace the relation Lineitem with the corresponding authorized view. The rewritten query (after view expansion) is shown below

```
SELECT  lineitem.*  FROM  Lineitem, Orders O1
    WHERE  l_orderkey = O1.o_orderkey
           AND O1.o_custkey = '123' and
           EXISTS (SELECT * FROM Orders O2
                WHERE l_orderkey = O2.o_orderkey
                      and O2.o_custkey = userId())
```

Note that the rewritten query includes an additional semi-join with the Orders table. We assume that the function userId() is evaluated at optimization time, and replaced by its return value, which would be 123 in the above example. Notice that the query has a selection o_custkey = 123, and thus accesses only tuples that are authorized. Thus, for this example the additional semi-join introduced by the rewriting is actually redundant. In general the rewritten query could include many semi-joins that are redundant. Indeed, we believe that it is fairly common for rewritten queries to include redundant semi-joins. This could potentially result in additional optimization as well as execution times for these queries. In order to optimize for the "common" case, we investigate techniques for redundancy removal and look at how they can be integrated in an existing query optimizer.

Although removal of redundant joins has been studied for many years (see, e.g. [3]), current generation commercial optimizers have only very limited forms of redundancy removal. The main reason is that queries have thus far rarely had redundancy, and the optimization effort spent to detect redundancy did not have worthwhile payoffs. The introduction of redundancy due to fine-grained access control motivates redundancy detection and removal. Our contributions in this context are as follows:

- We show how to detect and remove redundancy by exploiting existing code for matching (parts of) queries with materialized view definitions.
- We show how to implement redundancy removal by means of a set of transformation rules, enabling easy deployment in an optimizer based on the Volcano/Cascades framework [5][6], such as the SQL Server query optimizer. Since the redundancy introduced by authorized views is typically redundant semi-joins, we consider removal of redundant semi-joins, rather than joins. Interaction of these rules with other transformation rules is another issue that we address.
- We have implemented the transformation rules and show that we can get good performance benefits.

4.2 Detecting Redundancy

In general, the problem of redundancy removal can be rephrased as a query minimization problem; query minimization is NP-complete for conjunctive queries [3]. In fact, if we consider arbitrary arithmetic expressions, query containment (and minimization) is undecidable; however, in the special case where relations are not repeated, query minimization can be done in polynomial time for conjunctive (SPJ) queries.

For the purpose of fine grained authorization, we are primarily interested in detecting redundancy in semi-joins introduced by authorized views, which may be part of a more complex query involving other operations such as grouping and aggregation. We use the notion of subsumption in order to detect redundancy, which we explain next.

Definition (Subsumption). An expression E2 *subsumes* expression E1 if $E1 \rtimes_\theta E2 = E1$.

Thus if E2 subsumes E1, we can infer that E2 is redundant and transform $E1 \rtimes_\theta E2$ to E1. Although it is difficult to get necessary conditions for subsumption, we can use sufficient conditions to test for subsumption.

Consider the following transformation rule that replaces an expression with a selection over a materialized view

$(R1 \bowtie R2 \bowtie \ldots \bowtie Rn) \rightarrow (\sigma_\theta(V1))$

The above transformation is valid as long as the view V1 subsumes the expression $(R1 \bowtie R2 \bowtie \ldots \bowtie Rn)$. Thus, view matching in existing optimizer already uses subsumption checks. Optimizers typically normalize the expressions in order to check for subsumption. A SPGJ normal form has a cross product of relations, on which a

selection is applied, on top of which a projection is applied, and finally an optional group-by/aggregation operation is applied on top. Using the SPGJ normal form is much more efficient than attempting to match expression trees to determine subsumption. Of course, not all expressions have an SPGJ representation; our redundancy removal techniques apply only to those subexpressions of the query where such a representation is possible.

Our techniques are a minor variant of the conditions for rewriting using a materialized view. The primary change is that the subsumption test used for view matching requires that E1 and E2 have the same set of relations – we relax this to allow E2 to have a subset of the relations that E1 has. We outline the procedure for testing subsumption below.

Using the SPGJ representation of E1 and E2, we can test for subsumption of E1 by E2 in E1 $\ltimes_\theta$ E2 by checking that E2 has a subset of the relations that E1 has, and there is a mapping from the relations in E2 to those in E1 such that the following conditions hold:

1. The predicates in the selection in the SPGJ-representation of E2 are weaker than the corresponding predicates in E1, that is the predicates in E1 imply the predicates in E2, and
2. The semi-join condition in $\ltimes_\theta$ equates columns of E1 and E2 that are equivalent under the mapping.

Using the above procedure for testing subsumption between two expressions, the following two transformation rules can be used to detect and remove redundant semi-joins:

- At a semi-join E1 $\ltimes_\theta$ E2, check if E2 subsumes E1. If so, transform E1$\ltimes_\theta$ E2 to E1.
- Consider a query where rewriting using an authorized view results in a disjunction of subquery expressions, such as:

```
SELECT * FROM E1
  WHERE (A IN (SELECT …))
        OR (B IN  (SELECT …))
```

  The first phase of the SQL Server optimizer transforms the where-clause expression into (in-effect) a disjunction of semi-joins. The subsumption test is applied to each of the disjuncts. If any one of the disjuncts is found to subsume the expression E1, discard the entire set of semi-joins in the disjunction.

Consider a rewritten query (after every relation is replaced by the corresponding authorized view) of the form

$$(\ldots((R1 \ltimes_{\theta 1} A1) \bowtie (R2 \ltimes_{\theta 2} A2)) \bowtie \ldots (Rn \ltimes_{\theta n} An))$$

The rules for redundancy removal check for patterns of the form E1 $\ltimes_\theta$ E2. If applied to the above expression, it would check for subsumption only between the pairs (Ri, Ai). Of course, if we apply the redundancy removal rules during the transformation phase, rules that push/pull semi-joins through joins would ensure that all possibilities for detecting redundancy would be explored. But in the worst case, the number of times the redundancy removal rules is fired could be exponential in the number of relations. In order to have a more efficient solution, we use the following technique. During the simplification phase, we use a normalized form of the above expression in which all the authorization semi-joins are pulled up in the query tree. This is implemented as a simple set of transformation rules, which pull semijoins up through joins and selections. For the above example, the normalized version of the expression obtained by semi-join pull up would be:

$$((..(R1 \bowtie R2 \bowtie \ldots \bowtie Rn) \ltimes_{\theta 1} A1) \ldots \ltimes_{\theta n} An).$$

Now the redundancy removal rules would check for subsumption between the Ai's and the original query expression (and not just the corresponding Ri's) resulting in better detection of redundancy. The number of times the redundancy removal rules are fired would also now be at most linear in the number of authorizations. This approach is just a heuristic and does not guarantee elimination of all redundant authorizations. For instance, the set of authorizations removed could vary based on the order of the Ai's in the normalized version of the query. But this scheme seems to work well in practice and is easy to integrate in an existing optimizer infrastructure. In Section 6, we present an experimental evaluation that shows the significant benefits of these techniques.

Note also that redundancy detection rules above are best applied before query decorrelation transformations are applied, since decorrelation may translate semi-joins into outer-joins. Redundancy detection on joins and outerjoins is harder, since we have to deal with duplicate counts, which can be ignored for semijoins.

4.3 Discussion

The transformation rules for redundancy removal described above are tailored for the case when the authorized view is a semi-join view, with all columns selected unchanged. Further transformation rules are required to effectively handle the case of expressions, such as nullification, in the select clause of the authorized view; we omit details for brevity.

It is interesting to note that redundancy removal can be used as a sufficient condition to test for query validity in the non-Truman model of [11] Given a query, we replace each relation by a semi-join with its authorizations, and then perform redundancy removal. If the query after redundancy removal is equivalent to the original query without the added authorization predicates, then the original query can be inferred to be valid.

The above rule can supplement other sufficient conditions for inferring validity that are presented in [11]. Although the above inference rule is not a necessary condition, we found that in many queries that accessed only authorized data, the added authorization predicates were all detected to be redundant and removed.

As noted earlier, query rewriting can change the semantics of a query. However, if we apply redundancy removal on a rewritten query, and get back the original query, we can infer that query rewriting does not change the semantics of the query.

5. INFORMATION LEAKAGE THROUGH UNSAFE FUNCTIONS

In this section, we first outline leakage channels from unsafe functions (whether system defined or user defined). We then define when a query plan can be judged as safe with respect to USF invocations; we then consider how an optimizer can be extended to find optimal safe plans. We end the section with a discussion on further optimizations for handling exceptions in Section 5.6.

5.1 Unsafe Functions

Recall the example in Section 1; the rewritten query (after incorporating the authorized view) was

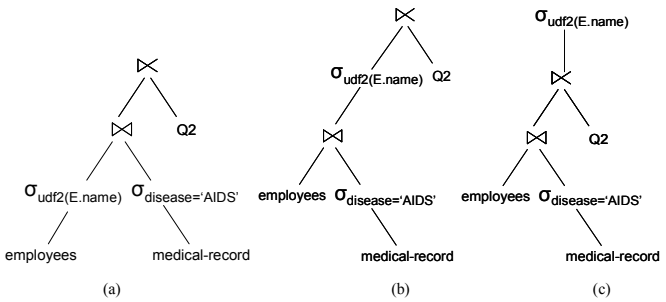

Figure 2: All the plans satisfy the naïve approach to determining safety, however, the UDF placement in (b) can potentially leak information.

```
SELECT * FROM myemployees
WHERE myudf(salary)
```

where myemployees represents an authorized view that involves a subquery. The optimizer could however pick a plan for evaluating this query in which myudf() is pushed below the access control check (Figure 1c). In such cases, the UDF has access to tuples that it is not authorized to see.

The code defining the UDF is not under the control of the authorization system and it could leak information about values passed to it in one of a number of ways, such as printing out the value, storing it in a database relation, generating an error message, raising an exception, or even through timing, by varying the execution time of the function depending on the values passed to the UDF.

Leakage can occur even with system defined functions that can throw exceptions. Consider the following example. Assume an employee database and that managers are authorized to see the salaries of employees in their department. Thus, the authorization predicate on the employee relation uses a semi-join (subquery) with the relation manager (managerid, deptid). Consider the following query

```
SELECT * FROM employee
WHERE empid = 'XYZ'  AND  1 / (salary – 100K) = 0.23
```

Assume the query is issued by someone who is not a manager of XYZ and hence is not authorized to see his salary. The employee relation would be replaced with the corresponding authorized view. The selection predicate involving the salary attribute and the predicate on empid column could however be pushed below the access control semi-join; and if there is an divide by zero exception encountered during query execution, the information that the salary of XYZ is 100K can be inferred.

Error messages generated by some functions are another source of information leakage. For example a system-defined function to_integer(), which converts strings to integers, would output an error message containing the string, if the string were not a valid integer; such error messages are important for finding erroneous data. But if a query uses a to_integer() function on a string which is not an integer, the error message can leak information about the actual value of the string, just like a user-defined function.

Sandboxing is a standard technique to protect database systems from potentially harmful effects of UDFs, such as accessing or corrupting system data. Sandboxing can be used to prevent some side effects such as I/O, but exceptions and error messages would provide leakage channels. Exceptions could perhaps be caught and hidden, but that may change the behavior of queries, and more importantly, update transactions. Error messages can be blocked, but that may make it hard for a genuine user to trace errors. Worse, even if exceptions and error messages are blocked, timing can be used to leak information. For example, a UDF which takes significantly longer if the salary of user XYZ is less than some cutoff, say 100K, can leak information through timing. In fact, it can be used repeatedly, with different cutoffs, to determine the exact salary of XYZ.

We say that a UDF or system-defined function/operator is *safe* if it has been (manually) verified to not leak information about parameter values passed to it, through any means such as those outlined above. All other functions/operators are said to be **unsafe functions**, or **USF**s. For example, the function userId(), which returns the identifier of the current user, is a UDF in the SQL sense, but we know it will not cause exceptions or leak information in any other fashion. The userId() function can therefore be treated as a safe function. We assume that UDFs invoked in the authorized views are safe, since they are defined by the security administrator, not a user.

One straightforward solution to the information leakage problem is to ensure that invocations of USFs happen only after all authorization checks have been carried out, by pulling USFs to the top of the query plan. Such an approach clearly ensures that USFs only see authorized information. However, pulling USFs to the top of the query plan can lead to inefficient query plans. To allow other plans to be considered, we first need to define when a query plan is safe with respect to USFs.

5.2 Safety of plans with respect to USFs

Consider a naïve (and as we shall see, incorrect) approach to determining safety of a query plan: suppose a USF invocation in a query plan is judged to be safe if all its parameters are from relations that have their access control checks enforced. It may appear that since each value passed to the USF is in the result of an authorized view, no unauthorized information can be revealed.

In reality, information can be leaked not only by values that are revealed, but also by values that are *not* revealed, as the following example shows. Suppose a particular user is allowed to access all tuples from the employee relation and only those tuples from the relation medical_record(emp_id, disease) that appear in the following view:

```
CREATE VIEW auth_medical_record AS
  SELECT * FROM medical_record
  WHERE emp_id in Q2
```

where Q2 is a subquery that determines the set of employees whose medical records a particular user is allowed to see. Suppose this user executes the following query:

```
SELECT *
FROM employee E, auth_medical_record  A
WHERE E.emp_id = A.emp_id
        AND A.disease='AIDS' AND udf2(E.name)
```

Of the several alternative plans for this query, we illustrate three plans in Figure 2. As the user has full access to employee relation, the parameter E.name to udf2() in all three plans trivially satisfies the above naïve condition. However, in the plan in Figure 2(b), names of all employees having AIDS disease reach udf2(), some of which may not have qualified subquery Q2, thereby leaving a channel open for information leakage.

Thus, the naïve approach to inferring safety described above does not actually guarantee safety with respect to USF invocations.

Given that the above naïve approach does not work, we now present a correct definition of safety. Before we do so, we point out one more detail that has to be handled, namely parameters to a correlated subquery. If a correlated subquery in a query plan has a USF in it (or nested anywhere below it), an invocation of the subquery can reveal information about the parameter values. For example, if we had a query

```
SELECT * FROM employee E, auth_medical_record A
WHERE E.emp_id = A.emp_id
      AND disease='AIDS'
      AND EXISTS
            (SELECT * FROM R WHERE  udf2(E.name))
```

where R is any non-empty relation. If in the query plan the condition disease=AIDS is checked first, and then the subquery with udf2() is invoked, and the authorization test (using Q1) is performed last, udf2() in the subquery can reveal unauthorized information about which employees have AIDS. This problem can be handled by treating the invocation of a subquery containing a USF in the same fashion as the invocation of a USF.

The Microsoft SQL Server optimizer represents correlated evaluation plans algebraically using an *apply* operator E1 *A* E2; the apply operator *A* invokes its right input E2 for each tuple generated by its left input E1. Correlation variables are bound by E1 and used by E2.

Definition 1: (Authorized Expression) An (algebraic) expression is authorized if it is equivalent to an expression defined using only authorized views.

Clearly, the query obtained by replacing each relation in the original user query by its authorized view is authorized since it is an expression defined using only authorized views. However, transformations applied to the query during query optimization can generate a number of different expressions. A key issue is to efficiently infer which of these expressions is authorized. We return to this issue in Section 5.3.

We now define when a plan is safe. Note that the property of being authorized is different from plan safety: authorization is a logical property of an expression, regardless of the plan used to compute the expression, whereas safety is a property of a particular query plan, which ensures that it cannot leak information (using channels other than the query result itself).

Definition 2 (**Safety w.r.t. USFs**) A node in a query plan is safe w.r.t. USFs if:

1. it there are no USFs in the node, and all inputs (if any) of the node are safe, or
2. the node has a USF, it is not an apply operator, and all its inputs are safe and authorized (treating correlation variables defined by ancestor apply operators as constants), or
3. the node is an apply operator, both its children are safe, and either (a) the right child (subquery) does not have any USF invocations, or (b) the left child is authorized (treating any correlation variables defined by ancestor apply operators as constants)

A plan is safe if its root node is safe (or, equivalently, all its nodes are safe).□

It should be clear that in a safe plan, USFs are invoked only on the results of expressions which can be computed using only authorized views. For cases where some correlation variables may be defined higher up, by part 3 of the above definition, these correlation variables are themselves generated by authorized expressions. As a result, unauthorized information is never passed to a USF.

Note also that the above definition of safety does not depend on the form of the authorized view. As a result it can be used with arbitrary views, including those which perform nullification.

5.3 Inferring Authorization

We now consider how to infer if an expression is authorized. To do so we need to find an equivalent expression that has as leaves authorized views. Unfortunately, this is not an easy task: first, the problem of query equivalence is NP complete as discussed earlier in Section 4.2. Further, we don't even know which expression on authorized views to compare the subexpression to. Several inference rules to check for authorization are presented in [11]; the goal there was to check for authorization of the given query, whereas we are trying to apply authorization tests to a potentially large number of subexpressions generated by a query optimizer (Section 5.4).

We use the "validity propagation" approach of [11] to infer authorization, since it can be used to infer authorization for multiple subexpressions at a low cost. The intuition behind the approach is as follows: authorized views (which replace the relations) in a query plan are all marked as authorized, and any expression generated during optimization is marked as authorized if all its inputs have been marked as authorized.

More formally, the validity propagation approach modifies the optimizer to infer authorization of expressions as follows. In the Volcano/Cascades optimization framework [5][6], an expression is represented by a group (or equivalence node), representing a group of equivalent expressions. A group may have multiple children, each of which is an operation node (such as join or selection); the children of the operation node are in turn equivalence nodes. Transformation rules may add more operation node children to a group node.

For illustration, we apply this mechanism on the example query in Section 5.1 in Figure 3. For simplicity, we remove UDF from this query, to focus on validity propagation instead of safety.

In the validity propagation approach, authorization is maintained as a group property in the optimizer's memo structure. This property is independent of which plan is chosen to implement the expression. We start with the rewritten form of the query in which each relation Ri is replaced by its authorized view (with semi-join authorizations, $Ri \ltimes_{\theta 1} Ai$). The groups corresponding to the individual authorized views ($Ri \ltimes_{\theta 1} Ai$) are initially marked as authorized. As shown in Figure 3, we mark G1 and G5 as authorized as they correspond to the authorized expressions.

The following inference rule IA is then used repeatedly: Rule IA: *If all the children group nodes of an operation node are marked as authorized, the group node which is the parent of that operation node is also marked as authorized.*

In the example in Figure 3, by applying IA we infer that G6 is also authorized. A subtle point in our setting is the fact that a Group G1 might not be inferred as authorized when it is created, but later if a new expression is added as a child of G1, it allows us to infer that

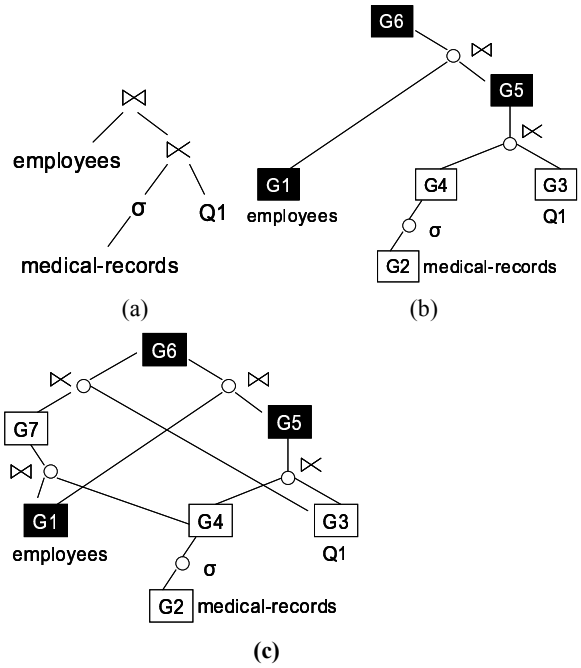

Figure 3: (a) The initial query tree for query in Section 5.1 without UDF. (b) The DAG representation of query. (c) The expanded query tree after applying transformation rules. Commutativity and Selection pull up not shown. Black boxes represent the authorized groups, and white boxes represent the unauthorized groups.

G1 is authorized. This information may in turn allow us to deduce that a parent (or ancestor) group is authorized. As a result, whenever we infer a group to be authorized we need to propagate the authorization property changes up to parent and ancestor groups in the memo structure.

A key difference with [11] is the fact that they assume that transformation rules have been completely applied before checking for validity (authorization), whereas in our context the inference has to take place while transformations are being applied (Section 5.4).

It may appear that this inference rule is overly simplistic. In fact, it can be quite powerful, as the following theorem shows.

Theorem 1. *If the entire search space is explored by optimizer, every authorized expression that is generated will be marked as authorized using the inference rule IA.* □

We omit the proof for lack of space. We note that [11] does not consider completeness of the authorization propagation rule. In general optimizers prune the search space, so there is no guarantee that all authorized expressions are marked as authorized, but our performance study shows this rule works quite well in practice. We note that the results in this section are independent of the form of the authorized view, and can therefore also handle nullification based cell-level authorization.

5.4 Approaches to Generating Safe Plans

We now elaborate how a top-down optimizer based on Volcano/Cascades [5][6], such as the query optimizer used in Microsoft SQL Server, can be modified to generate safe query plans involving USFs.

It is easy to devise heuristics to find a safe plan. A naïve method is to ensure that the USF invocations are never pushed down (by disabling the corresponding optimizer rule for UDF/selection/projection push down). Since the USFs stay at the top of the query plan, all the necessary access control checks will be applied before USF invocation and the resulting query plan would be safe. However, the plan obtained can be far from optimal, especially if the USFs are used in predicates that are very selective (i.e. eliminate most tuples). This point is substantiated in our performance study in Section 6.

Another possible approach is to just tweak the optimal unsafe plan (by pulling up USFs) till the plan becomes safe. This is a simple technique that requires relatively little modification of the query optimizer. Unfortunately, the resultant plan may not be the optimal safe plan, and may in fact be quite inefficient compared to the optimal safe plan. Moreover, to check if a node in the query plan is authorized we need the memo structure after the query is expanded, and the memo structures of the authorized views after expansion.

Given that the memo structures are needed anyway, it is natural to ask if there is a principled way to extend the search algorithm of the query optimizer to find the optimal safe plan. It turns out we can achieve this by enforcing safety either at every transformation rule (that involve USFs) or only when we pick the final plan. The two strategies are discussed below

1. One approach is to modify all the optimizer rules such that USFs are only pushed on top of authorized expressions. The transformation rule that pushes USF to a query expression succeeds only if the query expression is known to be authorized. It is easy to see this approach would generate a safe query plan. A potential drawback is that the rule may fail if a group which is actually authorized has not yet been inferred to be authorized. Some candidate plans may be missed as a result.
2. The second approach allows unsafe transformations but enforces safety when picking the optimal plan. In this approach, USFs are pushed down (as a transformation, not a substitution) even to potentially unauthorized query expressions. When the optimal plan is picked, the safety property can be enforced using an existing optimizer feature called required/derived properties. The physical operators derive the safety property from the their corresponding logical operators, while the procedure that finds the optimal plan ensures that only plans satisfying the safety property are considered. This is similar to the notion of enforcing a sort order to support an order-by clause.

The two approaches are related, we can in fact show that in certain cases the two approaches would in fact explore the same set of safe plans, as the following theorem states.

Theorem 2. For the case where queries as well as authorized views are conjunctive (SPJ) queries, if the entire search space of the optimizer is explored, all the safe plans obtained by approach 2 can also be obtained by approach 1.□

In our implementation, we currently chose approach 1 above, disallowing USF pushdown into expressions that have not been inferred to be authorized. We intend to further explore the relationship between the two approaches as part of future work.

5.5 Integrating Redundancy Removal and Safety

We illustrated how redundancy removal could be implemented by transformation rules that pull up semi-joins and detect redundancy using the view matching infrastructure (Section 4.2). We had mentioned earlier that these rules are executed as part of query simplification (since redundancy removal is almost always guaranteed to produce a better plan, much like pushing down selections).

However, it turns out there is a subtle interaction between redundancy removal and safety inference of UDFs. The key point is the fact that in order to infer if sub-expressions are authorized using the validity propagation approach, we need the query expression which is input to the transformation phase to be in the canonical form shown below, with authorization checks intact.

(.. ((R1 $\ltimes$ A1) $\bowtie$ (R2 $\ltimes$ A2)) $\bowtie$… (Rn $\ltimes$ An))

If redundancy removal were applied prior to the transformation phase, we may have eliminated some of the Ai's; the validity propagation approach would then not be able to infer many intermediate results to be authorized even if they actually are: for if any Ai is found to be redundant and deleted, notice that Ri will not be part of any authorized expression. There are a couple of ways to circumvent this problem, which we now describe.

5.5.1 Redundancy Removal During Transformation

One simple approach to solving this problem is to apply redundancy removal during the transformation phase (instead of the simplification phase), so that the Ai's are preserved and can be used to infer authorization.

To prevent other simplification rules from affecting the normal form, we introduce *authorization-anchor* operation nodes before the simplification stage, which prevent any transformations that pull up any of the Ri's or Ai's, or push down any operation into the Ri $\ltimes$ Ai expressions. At the start of the transformation phase, we remove the authorization-anchor nodes and mark as authorized the corresponding groups. Propagation of authorization is done as usual during optimization. (Propagating the authorization marking up on the initial query tree actually results in all the parent/ancestor groups being marked as authorized. However, only some of groups added subsequently may be authorized.)

Redundancy removal rules are then applied during the transformation phase. Because groups have alternatives with the authorization predicates in place, we can infer authorization of groups using the validity propagation approach. At the same time, the group can have other equivalent expressions with redundant parts removed, allowing more efficient evaluation. Thus, we are able to get an optimal safe plan.

We note that the redundancy removal rules in our prototype are substitution rules, which allow partial pruning of the search space when redundancy is detected; we omit details for brevity. This can reduce the optimization overhead compared to an approach that does not remove redundancy, or which does not do the partial pruning mentioned above. However, the overhead in this case is larger than when redundancy removal is applied in the simplification stage, because the redundancy of semi-joins gets tested multiple times, and because of the increased search space due to late detection of the redundant parts.

An interesting challenge in applying redundancy removal as part of the transformation phase is the fact that this could lead to potential cycles in the memo structure. For instance, when we eliminate a semi-join, we generate a child node that is equivalent to the parent and is part of the same group. The SQL Server optimizer which we used for our prototype had partial support for handling cycles in the memo, which we had to extend. Details are beyond the scope of this paper.

5.5.2 Conditioned Authorization

A better approach to integrating safe plan generation is to perform redundancy removal at simplification time, but use an extended notion of authorization, which we call *conditioned authorization*. (Note: this notation is not to be confused with the conditional validity notion of [11].)

With conditioned authorization, instead of making an expression as authorized, we may mark it as authorized conditioned on a join/semijoin with another expression. For example suppose we have a relation Ri with authorization Ai. Ai could possibly be deleted by redundancy removal or moved elsewhere in the expression during simplification. We mark Ri as authorized conditioned on Ai, that is, conditioned on it being joined/semi-joined with Ai. (We can add the join condition to the authorization, but for simplicity we omit the join condition in our discussion.) Authorization conditioned on an empty expression is the same as unconditioned authorization.

Groups (or equivalence nodes in the memo) derive their authorizations as follows. If any child is unconditionally authorized, so is the group. Otherwise, the authorization condition for a group is the disjunction of the authorization conditions of its children; the disjunction can be suitably simplified.

The rule for propagating authorization is then modified. For example, if expression E is of the form E1 $\bowtie$ E2, where E1 is authorized conditioned on Ai, while E2 is unconditionally authorized, if E2 is equivalent to Bj $\bowtie$ Ai, we can infer that the resultant expression is unconditionally authorized. The extended propagation rule is as follows:

1. If an operation has two child groups E1 and E2 that are each authorized conditioned on A1 and A2 respectively, the result of the operation is authorized conditioned on A1 and A2.
2. The above condition is then simplified as follows: if A1 subsumes E2, we drop A1 (and similarly for A2) from the condition.

If simplification results in an empty condition, we can infer that the expression is unconditionally authorized. Note that if authorization condition A1 (on relation R1) were dropped as redundant during initial query simplification, surely the query would have had an expression that is subsumed by A1; this expression will be joined with R1 at some point in the query plan; at this point, the authorization condition A1 would get dropped.

For lack of space we omit the complete formalization of conditioned authorization.

5.6 Handling Exceptions and Error Messages

We have seen how to handle USFs by moving them to safe locations. The most common USFs are built-in functions and operations (such as division or conversion to integer) that can cause exceptions or error messages (such as divide by zero or a bad input error message), such as the examples we saw in Section 1. We can improve execution costs significantly for built-in functions, as

follows. For each built-in function (or operation), we create a safe version of the function that ignores exceptions, does not output error messages, and does not have any side effects. For instance, we can create a safe version of the division function, which catches exceptions, and returns a null value.

Predicates using unsafe functions are rewritten using the corresponding safe versions of the functions, in such a way that the rewritten predicate is weaker than the original one. In the example from Section 1, we can create a safe version of the predicate (1 / (salary-100K) = 0.23) by using the safe division. However, replacing an unsafe function in a predicate by a safe function may allow tuples through that would not have passed the original predicate (e.g. the predicate 1/ (salary-100K) is null) and vice-versa. We can however rewrite the predicate using safe functions in such a way that it is weaker than the original condition. The rewriting has to deal with negations and null/unknown results; we omit details for brevity.

We can then push down the safe version of the predicate while retaining the unsafe version on top, above authorization predicates. In general, let the original predicate be θ, and the rewritten safe one be $\theta 1$. We can then transform a selection $\sigma_\theta(E)$ to $\sigma_\theta(\sigma_{\theta 1}(E))$, and push the rewritten selection $\theta 1$ down into the query. The original predicate θ remains on top, to filter out tuples that get erroneously included by $\theta 1$.

We note that catching exceptions can alter the behavior of the query, in terms of what it executes before an exception prevents further execution; this is especially true for updates that are not transactional. However, even in current SQL implementations the behavior in such situations is not defined by the SQL standard, and depends on the query plan chosen.

6. PERFORMANCE EVALUATION

We have built a prototype that incorporates our techniques for redundancy removal, and for generating safe plans. We added the transformation rules for implementing redundancy removal to the SQL Server query optimizer, and also modified it to take safety with respect to USFs into account. (Unsafe system functions can be handled by modifying the code to recognize USFs such as arithmetic operations and to handle them in the same way as UDFs). For queries that require both redundancy removal and safe USF placement, our prototype adopts the technique described in Section 5.5.1.

6.1 Benefits of Redundancy Removal

In order to illustrate the benefits of redundancy removal, we present a sample scenario using the TPCH schema. Consider a user who is trying to analyze data that have been shipped in the last ten years. For this user, the DBA may create authorized views in order to ensure that the user is not allowed to access data from an earlier period. A possible set of authorized views are shown below

authLineitem: $\sigma_{l\_shipdate > '1995\text{-}01\text{-}01'}$ (Lineitem)

authOrders: (Orders $\ltimes$ authLineitem)

authSupplier: (Supplier $\ltimes$ authLineitem)

authCustomer: (Customer $\ltimes$ (Orders $\ltimes$ authLineitem)

The views essentially restrict access in each of the tables to only those tuples that correspond to a lineitem that has been shipped after 1995. Input queries will be rewritten by replacing each of the base relations with the corresponding authorized views. We modified the query predicates in the TPCH queries to restrict the access to only those lineitems that have been shipped after 1995. Thus the authorization checks for the purpose of this experiment are all redundant.

TPCH Query	Execution Time Without RR	Execution Time With RR
Query 3	100.00	48.28
Query 6	56.03	38.79
Query 10	94.83	55.45
Query 12	77.57	43.97
Query 14	49.14	38.79

Table 1: Results with redundancy removal

Table 1 shows the benefits of performing redundancy removal for a few queries of varying complexity from the TPCH suite. For instance, Query 6 is a single table query, while the rest of the queries involve multiple joins. Query 10 is a join between 4 tables and the rewritten version of the query has 3 additional (redundant) joins. For each query, the execution times with and without redundancy removal is shown. The execution times shown have been normalized. The transformation rules that we discussed in Section 4.2 manage to detect and eliminate all the redundant semi-joins which were added as a result of the authorized views. The additional optimization overheads (due to redundancy removal) were around 10-15%, which is certainly reasonable, given the sizable gains in execution time. As the numbers indicate, redundancy removal can lead to a significant improvement.

6.2 Plan Safety and Redundancy Removal

In this section, we illustrate the importance of combining redundancy removal and safe USF pushdown, using an example. We construct a hypothetical scenario using the TPCH schema, where authorizations are created for managers in Europe. The authorized views restrict access to information that is only pertinent to regions in Europe, by using appropriate semi-joins.

The authorized views for the different relations are shown below (the selection predicate θ on the region table selects regions that are in Europe). The example shows that even such a simple scenario could result in complex authorized views.

authNation: access to nation information for nations that are in Europe: (Nation $\ltimes$ (σ_θ (Region))

authCustomer: provides access to customer information that are in some nation in Europe: (Customer $\ltimes$ (Nation $\ltimes$ (σ_θ (Region))

authOrders: provides access to orders that have been placed by customers from a nation in Europe.

(Orders $\ltimes$ (Customer $\ltimes$ (Nation $\ltimes$ (σ_θ (Region)))

authLineitem*:* provides access to lineitems that have been ordered by some customer in Europe:

(Lineitem $\ltimes$ (Orders $\ltimes$ (Customer $\ltimes$ (Nation $\ltimes$ (σ_θ (Region))))

authSupplier: provides access to suppliers that have supplied some lineitems for an order placed by some customer in Europe.

(Supplier $\ltimes$ (Lineitem $\ltimes$ (Orders $\ltimes$ (Customer $\ltimes$ (Nation $\ltimes$ (σ_θ (Region)))))

Redundancy Removal Phase	USF placement	Exec time
No removal	USF on Top	100.00
Simplification phase	USF on Top	47.83
No removal	Optimal safe	52.25
Transformation phase	Optimal safe	23.25

Table 2: Combinations of optimization alternatives

As the view definitions indicate, the authorized views include multiple semi-joins to restrict access to only those tuples that are relevant. Now, consider a query issued by a manager in Europe in order to examine the details of suppliers who supplied the lineitem's for the "sensitive" orders that were placed by customer in Europe. Suppose a UDF sensitiveOrder() is used to define which orders are sensitive, and it is not verified and may be unsafe, The preprocessor will replace the relations in the user query by the corresponding authorized views to obtain the following query.

```
SELECT *
FROM authSupplier, authLineitem, authOrders
WHERE  s_suppkey = l_suppkey
   AND l_orderkey = o_orderkey
   AND o_custkey  IN
     (SELECT c_custkey FROM authCustomer
      WHERE  c_nationkey  IN
       (SELECT n_nationkey FROM authNation
        WHERE n_regionkey  IN
            (SELECT r_regionkey FROM authRegion
             WHERE  r_name='Europe')))
  AND dbo.sensitiveOrder(o_totalprice)
```

The above query is then rewritten by expanding the authorized views. It is easy to see that the rewritten query would some redundant authorization checks since the query is only evaluated on European regions anyway.

We study the performance of the techniques we present in this paper for the above query. Table 2 illustrates the tradeoffs between using redundancy removal and/or safe USF pushdown in terms of execution times (which have been normalized). The corresponding query plans are illustrated in Figure 4. Notice that all the alternatives would generate safe query plans. We also measured the execution cost of the optimal unsafe plan, and found that it was the same as that of the optimal safe plan, since the UDF was not as selective as the join, and was pulled up above the join based on cost considerations. This may not happen always, of course.

In the absence of the techniques we present in this paper, an implementation of fine-grained access control would process this query without removing any redundancies and by pulling the UDF to the highest level in query plan. The resulting plan, given in Figure 4 (a), executes in 100 units of time (after normalization). We find that either of the techniques we propose in this paper – redundancy removal or safe placement of USFs – helps us reduce the execution times by nearly 50%. The corresponding query plans are illustrated in Figure 4 (b) and Figure 4 (c).

Finally, we find that when we apply both the techniques, we are able to find the plan, shown in Figure 4 (d), that further reduce the execution time by another 50%.

Thus, both redundancy removal and safe USF pushdown are essential components in implementing fine grained access control.

Redundancy Removal Phase	USF Placement	Normalized Optimization Time
Transformation	USF on Top	100.00
Transformation	Optimal Safe	43.29
Simplification	USF on Top	7.62

Table 3: Normalized optimization time

6.3 Optimization Cost

Redundancy removal can be performed either at the simplification phase or the transformation phase. As described in section 5.5.1, the key difference is the overheads involved. Table 3 shows the normalized optimization time cost of various alternative combinations of approaches for redundancy removal and USF placement from the example in the previous section.

As can be seen from Table 3, performing redundancy removal during transformations is quite expensive compared to performing redundancy removal during simplification. This is not surprising, since redundancy removal at simplification time can reduce the size of the query very significantly. The ideal solution would employ redundancy removal in the simplification stage and still use safe USF placement. This can be achieved using the notion of conditioned authorization inference (Section 5.5.2). Since simplification would have the same effect in this case, and conditioned authorization inference is similar in cost to materialized view matching, which is quite efficient, we believe the optimization

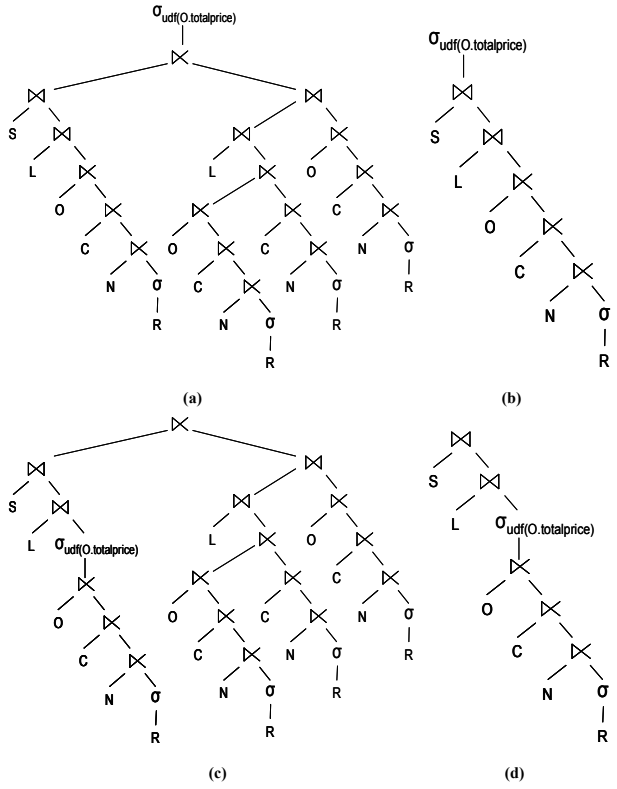

Figure 4: Optimal plans for query in Section 6.2 with (a) No redundancy removal and pulling USFs to highest level. (b) Applying redundancy removal. (c) Pushing USFs to safe places. (d) Both redundancy removal and pushing USFs to safe places.

costs will be similar to the third row (simplification+USF at top), which is quite low.

To summarize our performance results, (a) redundancy removal can give very significant performance improvements with low optimization overhead, and can in fact reduce optimization costs greatly, (b) although pulling USFs to the top works well in several cases, optimal placement of USFs can give significantly better results in many cases, and (c) it is feasible to modify an optimizer to generate safe plans. As part of ongoing work we are implementing conditioned authorization inference, and we believe that this would reduce the overheads to a reasonable fraction of the optimization costs for the same query (with redundancy removal).

7. CONCLUSIONS

Any fine-grained authorization implementation has to address the problem of redundant authorization checks, and the problem of information leakage. We showed how redundancy removal can be done effectively and efficiently as a query simplification step. We then defined when a query expression can be considered safe with respect to unsafe function invocations. We illustrated how redundancy removal and safe USF pushdown can be incorporated in an existing rule based query optimizer by adding new transformation rules. Our performance study shows that redundancy removal is feasible, and gives significant performance benefits with only a very small effect on optimization cost. The study also shows that leakage of information through USFs, exceptions and error messages can be efficiently tackled by choosing good safe plans.

As part of future work, we intend to extend our prototype to include conditioned authorization, which will reduce optimization time, and carry out a more detailed performance study. We also need to add rules to better optimize authorized views more complex than semi-join views, in particular to support nullification. We also plan to study and address other potential sources of information leakage, such as timing based on subquery execution time.

Acknowledgements: We wish to thank Paul Larson for discussions on SQL Server query optimizer and on view matching, and for his help with the implementation, Surajit Chaudhuri and Tanmoy Dutta for discussions on fine-grained authorization, and the anonymous referees for their insightful suggestions.

8. REFERENCES

[1] R. Agrawal, P. Bird, T. Grandison, J. Kiernan, S. Logan, W. Rjaibi: Extending Relational Database Systems to Automatically Enforce Privacy Policies. In *ICDE*, pages 1013–1022, 2005.

[2] A. Brodsky, C. Farkas, and S. Jajodia. Secure databases: Constraints, inference channels, and monitoring disclosures. *IEEE Trans. on Knowl. and Data Engg.,* 12(6):900–919, 2000.

[3] A.K Chandra and P.M Merlin, Optimal Implementation of conjunctive queries in relational databases. *STOC*, 1977

[4] K. LeFevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu and D. DeWitt, Limiting disclosure in Hippocratic databases, In *VLDB*, 2004

[5] G. Graefe, W. McKenna, The Volcano Optimizer Generator: Extensibility and Efficient Search, In *ICDE*, 1993

[6] G. Graefe, The Cascades Optimization Framework, *Data Engg. Bulletin*, 1995

[7] D. Litchfield, Web Application Disassembly with ODBC Error Messages, 2001, http://www.blackhat.com/presentations/win-usa-01/Litchfield/BHWin01Litchfield.doc

[8] A. Motro. An access authorization model for relational databases based on algebraic manipulation of view definitions. In *ICDE*, pages 339–347, 1989.

[9] The Virtual Private Database in Oracle9ir2: An Oracle Technical White Paper http://otn.oracle.com/deploy/security/oracle9ir2/pdf/vpd9ir2twp.pdf.

[10] New Security Features in Sybase Adaptive Server Enterprise. Sybase Technical White Paper, 2003.

[11] S. Rizvi, A. Mendelzon, S. Sudarshan and P. Roy, Extending query rewriting techniques for fine-grained access control. In *SIGMOD*, 2004

[12] A. Rosenthal and E. Sciore. Abstracting and Refining Authorization in SQL. In Secure Data Management (SDM) workshop, In *VLDB*, 2004.

[13] M. Stonebraker and E. Wong. Access control in a relational database management system by query modification. In *Procs of the ACM Annual Conference*, pages 180-186, 1974.