Raising Authorization Awareness in a DBMS

Abhijeet Mohapatra [1], Ravi Ramamurthy [2], Raghav Kaushik [2],

[1]*Stanford University,* [2]*Microsoft Research*

ABSTRACT

Fine-grained authorization (FGA) is a critical feature of many database applications. The general approach to FGA both in research and practice is the following: FGA policies are enforced by rewriting queries as a function of the current user. This query rewriting suffices for supporting the functionality of FGA but essentially treats FGA as a second-class citizen — most of the DBMS is *unaware* of authorizations; all the authorization logic is encapsulated in a small component that performs query rewriting.

In this paper, we argue that in order to engineer good performance, it is essential to treat FGA as a first-class citizen by making the core components of the DBMS authorization-aware. As concrete evidence, we propose a novel index structure that we call an *authorization index* and show how an optimizer can exploit it to generate plans that are significantly better than the plans obtained using the rewriting approach. We also discuss how the tightly-coupled integration of authorizations provides an interesting case for revisiting other query processing problems.

1. INTRODUCTION

The current authorization model in the SQL standard allows authorizations only to coarse grained objects such as tables and views. Such coarse grained authorizations are inadequate for many applications. For example, a payroll application accessing Employee information might want to let each employee access only their own data. Such fine-grained authorizations have been traditionally implemented in the application with little database support. Some commercial database systems such as Oracle VPD [1] have recently started providing native support for fine-grained authorizations [2]. The idea is that each user has access only to a subset of rows in each table. The authorized subset is specified using an authorization policy. We illustrate through an example.

Example 1.1 Consider a database that has the tables `submissions(paperID, title, track)`,`users(email, userID)` and `authors(paperID, userID)`. Consider an authorization policy that lets authors access their submissions. This policy can be specified as follows [3]. We assume that the function $userID()$ provides the identity of the current (application) user.

```
GRANT select on submissions
where paperID in
  (select paperID from authors
   where userID = userID())
to public
```

1.1 Rewrite-then-Optimize Approach

The authorization policy is enforced by explicitly *rewriting* queries [4] to go against the authorized subset. The key advantage of this approach is that existing applications need not be modified since the query rewriting is completely transparent to the application as shown in the following example.

Example 1.2 Consider the database in Example 1.1. The query:

```
select * from submissions
```

gets rewritten to:

```
select * from submissions
where paperID in
     (select paperID from authors
      where userID = userID())
```

In terms of implementation, there is a *query rewriting* module that is aware of the authorization policy. Any query posed to the system is rewritten by the module to enforce the authorizations. After the rewriting, the rest of the query processing proceeds as usual — the rewritten SQL query is optimized and executed. This *rewrite-then-optimize* architecture has the advantage that it separates the authorization subsystem from the query processing subsystem. The query rewriter that enforces the authorization policy does not have to take execution costs and physical design into account. Similarly, the query optimization and execution sub-system does not have to be modified in order to incorporate support for (fine-grained) authorizations — as far as they are concerned, the rewritten query is yet another SQL query.

We observe in Example 1.2 that even though the original query has no joins, the query rewriter adds a join in order to enforce the authorization policy. In general, the complexity of the rewritten query increases with the complexity of the authorization policy.

Example 1.3 We extend the database in Example 1.1 to add the tables `trackChair(track, userID)`, `reviewers(paperID, userID)` and `conflicts(paperID, userID)`. We also extend the authorization policy to let reviewers have access to papers assigned to them and track chairs have access to all non-conflicting papers in the respective track. Then the query `select * from submissions` gets rewritten as:

7th Biennial Conference on Innovative Data Systems Research (CIDR '15) January 4-7, 2015, Asilomar, California, USA.

```
select * from submissions
where paperID in
     (select paperID from authors
      where userID = userID())
union
select * from submissions
where paperID in
    (select paperID from reviewers
     where userID = userID())
union
select * from submissions
where track in
    (select track from trackChair
     where userID = userID()
     and not exists
      (select *
      from conflicts
      where userID = trackChair.userID
       and paperID = submissions.paperID)))
```

Thus, what seems like a very simple operation namely looking up all relevant submissions for a particular user gets rewritten to a complex query that joins submissions with four other tables and takes the union of the results. We note that Example 1.3 illustrates how even the simplest single table queries can result in complex rewritten queries for non-trivial policies. For queries that involve multiple joins and subqueries as in the TPC-H benchmark, the complexity of the rewritten query increases significantly — we need to add authorization checks as in Example 1.3 for *every table* referenced in the query. Not surprisingly, the complexity of the rewritten queries can adversely impact the query performance.

The first natural optimization to consider is to "tune" the query by selecting an appropriate physical database design. Many commercial database systems support automatic physical design tools (see [5] for an overview) that suggest appropriate physical designs for an input query workload. However, the state of the art physical design mechanisms offer only limited optimizations for the above query. This is because commercial systems support a restricted class of materialized views that includes select-project-join queries with grouping and aggregation [1] and does not cover the above expression which involves a union of three queries including a nested subquery with a negation. Thus, the performance problems of the rewritten queries are unlikely to be "tuned" away for all policies. The question arises if we can do better or is this complexity intrinsic to enforcing authorizations.

1.2 Authorization Aware Query Processing

Note that the "rewrite-then-optimize" approach forces a separation between the authorization component and the query engine— components such as the optimizer, execution engine and physical design are not designed or optimized to account for authorization predicates.

Instead, consider the following strategy for executing the query in Example 1.3. Assume we can pre-compute the "capability list" that lists, for each user, all submissions to which they have access. Given the query `select * from submissions` issued by a specific user, we look up the "capability list" for the user similar to an index-seek plan and obtain all the submissions that they have access to. The above strategy accesses only two database objects — the `submissions` table and the "capability list"; this is in contrast with the complex query involving five tables implied by the rewriting shown in Example 1.3. The key question is how we can implement such a "capability list" efficiently. Obviously, materializing the capability-lists as a materialized view is not feasible for reasons previously described. Instead, consider the following alternative. Suppose that the query rewriter that enforces authorizations is aware of the presence of the "capability lists". It can then choose to rewrite the query to go against the capability lists (to enforce the authorizations) thereby eliminating the need for view matching; this could result in an execution plan that is much better than the plan obtained by explicitly enforcing the authorization through query rewriting.

[1]"matchable" views typically do not include constructs such as union — we define matchable views more precisely in Section 6.

In this paper, we explore the importance of raising authorization awareness in a DBMS. As a concrete example, we propose adding a new auxiliary structure that we term an *authorization index* (Section 2) to the database system, that essentially maintains the capability list for the table. Just as regular indexes provide a fast path for retrieving tuples that satisfy a particular predicate, authorization indexes provide a fast path for retrieving tuples that are *authorized for a particular user*. We briefly discuss challenges in integrating such indexes in a traditional optimizer (Section 3) and present a preliminary experimental evaluation that points to the benefits of such index structures (Section 4). We believe this is an initial step — opening up the DBMS to incorporate authorization as a first-class citizen provides an interesting opportunity to revisit well known query processing problems which we discuss in Section 5.

2. AUTHORIZATION INDEXES

Authorization indexes are a new auxiliary structure that provide a "fast-path" for accessing data that is authorized for a particular user. As in the case of regular indexes, the key issues that arise are: 1) creating and maintaining the indexes and 2) leveraging the indexes during query processing. In this section, we define authorization indexes and discuss how they can be created and maintained. We discuss how authorization indexes can be leveraged by the query rewriter/query optimizer in the following section.

DEFINITION 2.1. *Consider a table $\mathcal{T}$. We assume that each tuple in the table is uniquely identified by a surrogate (t_i). Given an access control policy $\mathcal{P}$ on $\mathcal{T}$, the set of userids $\mathcal{U}$ and the predicate $access(u_i, t_i)$ which returns true if user u_i is authorized to access tuple t_i under policy $\mathcal{P}$, an authorization index $\mathcal{I}$ on table $\mathcal{T}$ is defined as:*
$\mathcal{I}$ = { (u_i, t_i) | $u_i \in \mathcal{U}$ and $access(u_i, t_i)$ is true }

The surrogate (t_i) can be the RID of the tuple or the key value corresponding to any clustered index on the table $\mathcal{T}$. Thus, authorization indexes maintain the mapping between users and the corresponding RIDs that they are authorized to access in a table.

In general, an authorization index is a database object with the following properties.

1. An authorization index is a DDL construct. An authorization index is created via a `create authorization index` statement that specifies a table. The authorization index stores the mapping between users and the corresponding RIDs that they are allowed to access. Like a regular index, an authorization index is an auxiliary structure (it is not necessary to create an authorization index in order to enforce authorizations). It is similar to indexes and different from materialized views in that applications are not permitted to reference the authorization index by name.

2. As the name suggests, an authorization index is authorization-aware. This is a key difference from traditional indexes and materialized views (and hence motivates

the new terminology). In order to create the index, in addition to the table which is being indexed, we also need the authorization policy (that is stored as part of the system catalog) and a view that specifies the set of userIDs. By binding the index to the authorization policy, the query processing subsystem leverages an authorization index without any need for complex view matching. An "index-seek" operation performed over the authorization index to retrieve the authorized tuples of a particular user is equivalent to applying the authorization predicate *independent of its complexity*. For instance, in Example 1.3, an index-seek plan that fetches the RIDs corresponding to the current userID could be much more efficient that evaluating a query with four joins.

3. Just like any other auxiliary structure, an authorization index needs to be used in a cost-based manner by the query rewriter/query optimizer (see Section 3.1 for more details).

4. In order to maintain an authorization index as the data changes, we leverage previous work on incremental maintenance of views. This implies that authorization indexes can only be constructed for a limited class of authorizations. However, this is a much larger class than the set of materialized views that can be efficiently *matched*. We discuss index creation and maintenance in this section.

2.1 Index Creation

Recall that an authorization policy on a table is specified using a predicated grant. In general, consider the grant statement on a table T of the following form (where predicate P is parameterized using the userID() function).

```
grant select on T where P
```

A `create authorization index` statement for a table requires as input: 1) the corresponding predicated grant on the table and 2) a list of userIDs who can access this table. We assume these are supplied in a view (USERS(uid)). In some cases the set of userids directly corresponds to the set of values in a database table column (e.g, the o_custkey column) in which case the USERS table can be a view that points to the appropriate column. An authorization index needs to be maintained incrementally with updates. This is in general not feasible for arbitrary authorization predicates P. A `create authorization index` statement first checks if the predicated grant for the table falls in the class of predicates that are incrementally maintainable (Section 2.2) and fails otherwise.

The straightforward way of creating the index is to iterate over all uids in the USERS table and compute the "capability list" corresponding to predicate P and obtain the corresponding RIDS to create the index entry for each uid.

However, we can further optimize this step (similar to the notion of bulk-loading a traditional index). We assume that predicate P contains a reference to a table S that contains the predicate parameterized by the userID() function (say S.attr = userID()). For such a predicated grant, we could "bulk load" the authorization index I for this user-group by executing the following query (we assume the function RID() returns the RID/key column for the input row in table T).

```
insert into I
select distinct uid, RID(t)
from T  t, USERS u
where P′
```

In the above query, predicate P' is a modified version of the original predicate P that removes the occurrence of $S.attr = userID()$ in predicate P and replaces it with $S.attr = u.uid$. Note that the above procedure essentially outputs the userIDs in addition to the tuple RIDs. Since $S.attr$ may not be a key attribute of relation S, we filter duplicates from the join between table T and table S (hence the distinct clause).

In addition, we may wish to build an authorization index for only a subset of the users for the table. For instance, consider a CEO of a company who is authorized to see all employee data; the authorization index needs to keep track of this mapping for each tuple in the employee table. Excluding such an user from the index can help improve its storage efficiency and maintenance costs. We can extend authorization indexes to be defined over roles instead of users. This is similar to the notion of a partial index [6]. Assume each user belongs to an unique role. Thus, we have the option of creating an authorization index for users who belong in the "employee" role but not the "CEO" role. When a user logs in, the database system checks the role that he currently belongs to (in general a user can have multiple roles and log in using only a subset of those roles) and can thus determine if an authorization index can be used. Our techniques naturally extend for the case for the multiple roles.

2.2 Index Maintenance

As with regular indexes, it is necessary for authorization indexes to be incrementally maintained with updates. Note that updates can include updates made either: (a) to the base relations on which authorizations are defined or (b) to the set of users that have access to the table (the USERS table described in the previous section) or (c) to the authorization policy to the table.

If the authorization policy corresponding to a table is changed, the authorization index has to be dropped. This is similar to the case where the definition of a materialized view is changed; the materialized view is no longer valid. Unlike regular indexes, authorization indexes also need to be maintained when the set of users that have access to the table changes (this could happen when the set of users assigned to a particular role changes or through GRANT/REVOKE statements) — we note that the bulk-loading scheme for the index can be extended to incrementally maintain the index for this case.

For updates made to the base relation, we incrementally maintain the authorization index by leveraging prior work [7] that maintains views by using delta propagation rules.We need to add appropriate additional metadata (e.g., certain partial counts) to maintain the indexes as defined in [7]. The class of predicated grants that can be maintained using the techniques is described by the following grammar.

```
P := P and P | P or P  Q := SPJ query
  := <attr> op <attr>     := SPJ query where P
  := <attr> op <val>
  := <attr> op agg(Q)
  :=  exists(Q)
  :=  true | false
  :=  not P
```

As mentioned in Section 3.1, we disallow any `create authorization index` statement for a table whose predicated grant does not fall in the above class of predicate. In spite of restricting the class of predicates for which we can use authorization indexes, we note that the above language covers a rich class of authorization predicates including the examples discussed in the introduction.

Note that the key insight we leverage in authorization indexes is the fact that the class of maintainable materialized views far exceeds the class of matchable views (see Section 6) — while there is previous work [7] showing how we can incrementally maintain views containing union and limited forms of negation such as the expression above, it is unclear how to efficiently extend view *matching* technology to also capture union and negation. View matching is implemented in commercial optimizers by checking equivalence of query sub-expressions to a view expression. Even in the case of a single conjunctive view, checking equivalence has the same complexity as graph isomorphism [8] and it is known [9] that the worst-case complexity of view matching increases sharply when we add other operations such as unions. Note that authorization indexes sidestep the need to match views and only rely on incremental view maintenance algorithms.

3. AUTHORIZATION AWARE QUERY OPTIMIZATION

In order to effectively support authorization indexes, the query optimizer needs to be authorization aware — this is contrast to the "rewrite-then-optimize" approach in which the optimizer optimizes the rewritten query as a regular query. An authorization aware optimizer essentially integrates the query rewriting (to add authorizations) and cost-based optimization in a single step. Extending an optimizer to reason about authorization predicates has several dimensions; we focus on the following aspects: 1) why the choice of authorization indexes should be cost-based 2) how to integrate authorization indexes in a memo-based query optimizer 3) some interesting properties of authorization indexes. While we focus the discussion on a single query, we note that authorization indexes are also applicable to stored procedures which can contain multiple queries — the authorization checks and optimization are still enforced for each query individually.

3.1 Plans using Authorization Indexes

Example 3.1 Consider TPC-H Query 14 which is a join of the Lineitem and Part tables followed by an aggregate computation. Consider a policy in which a customer is only allowed to see lineitems corresponding to his orders and parts corresponding to those lineitems. In this case the original query (see Figure 1(a)) gets rewritten to the following query shown in Figure 1(b) (the additional semi-joins are added to enforce the authorizations).

Recall that authorization indexes essentially maintain the mapping between users and the corresponding RIDs that they are authorized to access for a table. Thus, for the join query discussed in the above example, a query plan that uses the authorization indexes on Lineitems and Parts (Figure 1(c)) to fetch the authorized tuples need not evaluate the additional semi-joins that are introduced due to the authorization predicates and could potentially result in a much better plan. As shown in Figure 1, authorization indexes provide an alternate means for enforcing the predicated grant on a table (instead of adding the authorization predicates).

A simple way to leverage authorization indexes is always consider using an authorization index for a table if it were available. However, just like the case of regular indexes, this may not be the best choice as shown in the following example.

Example 3.2 Figure 1 shows two query plans, one that uses the authorization predicates (Figure 1(b)) for both tables and another plan (Figure 1(c)) that uses the authorization indexes for both tables. Using an authorization index to retrieve all authorized tuples is likely

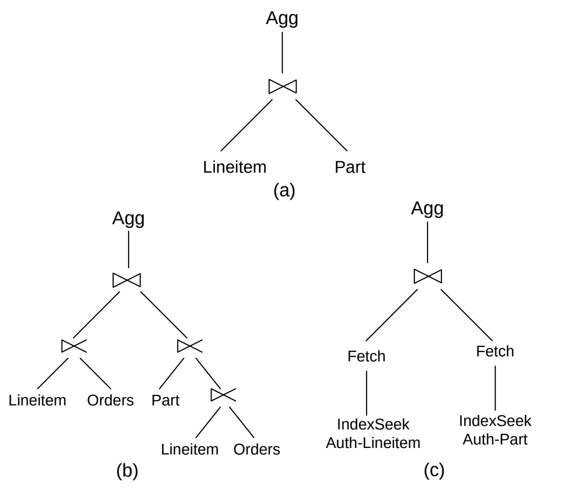

Figure 1: Different Query Plans for Example 4.1

to be efficient if the fraction of tuples in the table that the user is authorized to see is small (otherwise it could result in a large number of random I/Os). This is similar to the case of a regular index which is likely to be useful only when the predicate is highly selective. For instance, if the customer has ordered only a few lineitems and their corresponding parts are also a small fraction of the part table, then the plan that uses both authorization indexes is likely to be the best plan. However, consider scenarios where the number of lineitems is small but the number of corresponding parts is a large fraction of the part table (or vice-versa). In such cases, alternate plans as shown in Figure 2 that use authorization indexes for one table and adds the authorization predicates for another table could potentially be better choices than the two plans shown in Figure 1. This points to the fact that the choice of authorization indexes must be made in a cost based manner.

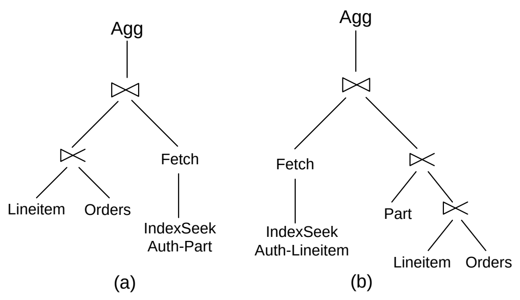

Figure 2: Alternate Query Plans for Example 4.1

3.2 Integrating Authorization in a Memo

We assume a transformation rule-based query optimizer(see [10] for an overview), but our techniques can be extended for other optimizer architectures as well. The key data structure used by transformation rule-based optimizers to keep track of different logical and physical plans is called a memo. Figure 3 shows a snapshot of the memo for TPC-H Query 14. Each group represents a set of equivalent logical expressions and keeps track of different implementation choices (shown shaded in Figure 3) available for it. For instance, Group 3 in Figure 3 represents the join between the two tables. It has two logical expressions corresponding to the two possible join orderings and an implementation using the hash join algorithm. Transformation rules are used to generate different logical alternatives and implementation algorithms for each group. Each plan (sub-plan) is costed by invoking suitable cost functions and suboptimal plans (sub-plans) are suitably pruned. Once all the rules have been applied, the memo is finally traversed top-down to pick the optimal plan.

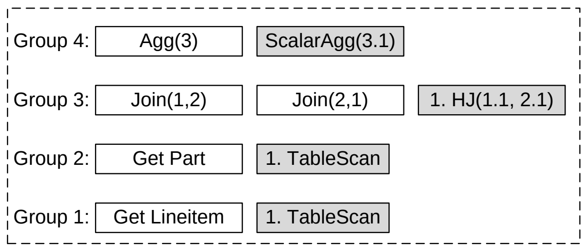

Figure 3: Fragment of Memo for TPC-H Query 14

Authorization indexes are in many ways similar to traditional indexes, there are implementation rules (covering index-seek plans) which need to be added to the memo [10]. Likewise, the issues in costing plans using such indexes are quite similar. However, a key difference is semantics : a index-seek plan is equivalent to a base table scan to which the authorization predicates have been applied. This is *independent* of the predicate complexity (e.g., it could include sub-queries) and this makes the integration non-trivial.

In order to extend the optimizer to incorporate authorization indexes, we need the following changes.

- We need to maintain a new logical property for every group: *IsAuthorized*
- We need to add a new logical rule for adding authorization predicates for a table
- We need to add a new implementation rule for authorization indexes

We now explain each of the above modifications. It is essential to keep track of whether each group is authorized or not. This is because the query optimizer has to now enforce the correctness of a execution plan with respect to the predicated grants (recall that the query rewriter does not exist in this architecture). We can track this by adding a new logical property for a group: *IsAuthorized*. Note that a group being authorized is a logical property; it does not depend on the specific algorithms used for implementation.

A group in the memo (that corresponds to a base table) can be explicitly authorized by adding the authorization predicate (for the appropriate table). This can be implemented by adding a new logical rule to add authorization predicates for a table based on the appropriate predicate grant. The *IsAuthorized* property can be propagated by using the rule that a group is authorized if and only if all its children are authorized. The key point is that authorization is explicitly enforced by the query optimizer while creating the appropriate groups in the memo thus bypassing view matching to identify the authorization predicates. While traversing the memo to pick the final plan, we need to pick the authorized query execution plan with the cheapest cost.

Authorization indexes can be incorporated in such an optimizer by adding a new implementation rule. Whenever an optimizer creates an authorized group for a single table relation by adding the authorization predicate, the implementation rule would add the index seek plan on an authorization index for the table (if it exists) as a possible implementation alternative.

Consider the join query discussed in example 4.1. A snapshot of the memo for an optimizer modified as stated above is shown in Figure 4. The authorized groups (Groups 4-7 in Figure 4) are marked with a '*'. Recall that the authorizations on the lineitem and parts table required additional semi-joins with the orders table in order to restrict the access to a particular customer's information. Notice that the optimizer adds these semi-joins in order to enforce the authorizations in Groups 4 and 5. The authorization index also serves as an implementation alternative for these groups. For instance, for the Group 5, the implementation alternatives are to use hash joins for enforcing the semi-joins (the HJ(2.1,4.1) alternative) or to use the authorization index. Depending on the cost of the alternatives, either of these plans can be chosen.

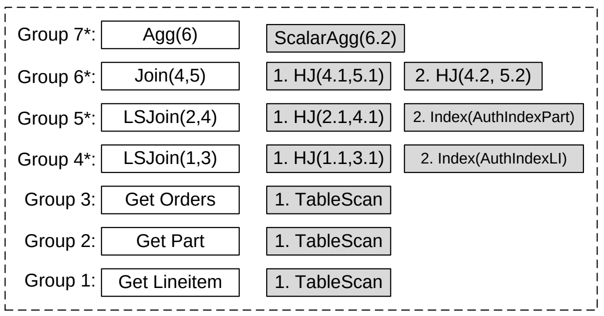

Figure 4: Fragment of Memo in Modified Optimizer

We note that there are several additional issues to be addressed — these include issues such as extending the cost-model for authorization indexes and appropriate pruning techniques for the new transformation rules. We defer a more detailed examination and evaluation of the proposed architecture of an authorization-aware optimizer to future work.

3.3 Side Channel Attacks and Safe Plans

It has been previously shown that fine-grained access control is amenable to side channel attacks [11]. For instance, user defined functions or predicates could involve operations with side effects. For instance 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 (which could be a join with another table). The selection predicate involving the salary attribute and the predicate on empid column could however be pushed below the access control semi-join due to classic optimizations considered by the query optimizer (e.g., select predicates are typically pushed down) . Note if there is an divide by zero exception encountered during query execution, the information that the salary of XYZ is 100K can be inferred. Thus, a query optimizer that is unaware of authorizations can potentially pick an "unsafe" plan. This again points to the fact that the rewrite-then-optimize architecture can result in subtle problems in enforcing authorization.

There has been previous work [11] that extends a query optimizer to choose only *safe* plans that avoid this problem. Interestingly, authorization indexes can naturally prevent such side channel attacks. The key reason is that the index access is *atomic* - an index seek plan for an authorization index (a leaf level operator in a plan) obtains *all authorized tuples* and no other predicate can be pushed into this operation [2] and this guarantees that only authorized tuples are even input to other operators such as filters. Thus, authorization indexes provide a simple way to derive safe plans [11].

4. EXPERIMENTAL EVALUATION

In this section we provide a preliminary evaluation of the effectiveness of using authorization indexes. We use a simple client based implementation to simulate query plans that use authorization indexes — we defer a detailed integration in a query optimizer and its evaluation to future work.

4.1 Implementation and Experimental Setup

We used the following experimental setup. We simulated an authorization index for a relation by materializing the corresponding userid and tuple id/key value mappings for that relation as a table (indexed on the userid column). By using a client side SQL parsing tool, we implemented an "authorization-aware" query rewriter that can rewrite the query to include the appropriate combination of authorization predicates and authorization indexes. We implemented a simple greedy algorithm to choose appropriate authorization indexes. The algorithm works as follows: we start with the rewritten query that includes all authorization predicates. As long as we can improve the query cost (using the optimizer estimated cost of the rewritten query), we consider rewritings that adds the "next-best" authorization index that improves the cost. Note when we add an authorization index we can drop the corresponding authorization predicate. We believe that this represents a simple strawman algorithm to simulate a good plan that uses authorization indexes — we note that a thorough integration with an optimizer would likely result in better plans.

We use the TPC-H database (1 GB version) and the TPC-H queries as our workload. We note that for the fine-grained authorization policies used in the experiments, some queries resulted in empty results. We only report the results for the queries that returned non-empty results for all the policies. We evaluate the effectiveness of authorization indexes with respect to the following two key parameters: (a) complexity of the authorization policy and (b) physical database design.

4.1.1 Authorization Policies

In order to study the effectiveness of authorization indexes under a variety of authorization policies, we used the following different authorization policies on the TPC-H database:

- P1: Policy P1 constrains a customer to see his own orders, the lineitems corresponding to his orders and so on (the policy discussed in Example 2.1). As discussed in Section II, the query rewriter will produce rewritten queries involving subqueries to enforce the authorizations.

- P2: Policy P2 is a complex policy that uses negations (similar to the example in the introduction). A particular user (e.g., an analyst) is allowed to see his records but is not allowed to any customer records that correspond to a particular nation. For policy P2, we generate a relation called AuthNations(c_custkey, n_name). A tuple $(c, n) \in$ AuthNations if customer c is not permitted to view the order corresponding to the nation n. We note that policy P2 cannot be re-written as a SPJ query.

```
GRANT select on orders
where o_orderkey IN
select o_orderkey
from orders  o, customer c, nation n
and o.o_custkey = c.c_custkey
and c.c_nationkey = n.n_nationkey
and c.c_custkey = user_id()
and n.n_name not in
     ( select * from AuthNations as a
       where  a.c_custkey = c.c_custkey)
```

- P3: Policy P3 is a complex policy that uses unions. A particular user (e.g., an analyst) is constrained to view the lineitems purchased in a fixed set of regions. The set of regions for a particular user will access is specified using UNIONS (we vary the number of unions to study increasing complex authorizations).

4.1.2 Physical Design

Another important parameter for consideration is the baseline for comparison. In order to defend against the best possible baseline, we carefully considered "tuned" versions of the database for each policy. For instance, authorization policy P1 may be rewritten as an equivalent SPJ policy (that represent the decorrelated versions of the rewritten queries which involve subqueries). For policy P1 , we consider a physical design that materializes all possible primary key-foreign key join-paths in the TPC-H database. However, exact materialized views are not feasible for policies P2 and P3 for since the authorizations are typically more complicated (with sub-queries, negations and unions) than the language of materialized views supported by commercial database systems. As an approximation, we tune the workload by using an automated physical design tool that is part of the database system we use. Such tools (see [5] for an overview) take an input query workload and a storage bound and return a set of indexes/materialized views that best optimize the workload while respecting a storage bound. We did not place constraints on the storage bound required. Thus, this physical design in some sense is fully "optimized" for the workload.

4.2 Results for Policy P1

We first report the results for policy P1 over the basic TPC-H database. We compare the query times of the rewritten TPC-H queries with authorization predicates to the query rewritings generated by the greedy algorithm. The graphs plot relative speedup as a function of the query — note that the Y axis is in log (base 2) scale.

Evaluation on Tuned Database: Authorization policy P1 is a SPJ policy. Hence, the rewritten queries may be potentially answered using the views that materialize the primary key-foreign key joins, we materialized views corresponding to all primary key-foreign key joins in the TPC-H database. We first compare the query times of the rewritten TPC-H queries in the tuned database to the query times of the query rewritings generated by the greedy algorithm. The results are reported in Figure 5. Only three queries in the benchmark (queries Q4, Q6 and Q8) could be matched with the materialized views. As mentioned in the introduction, view matching uses a lot of heuristics and there is no guarantee that the rewritten queries (involving subqueries) will be matched to the corresponding un-nested/de-correlated version of the materialized views. On an average, the benchmark queries rewritten using the authorization indexes ran 90x faster than the ones matched against

[2] note that authorization indexes do not support SARGable predicates

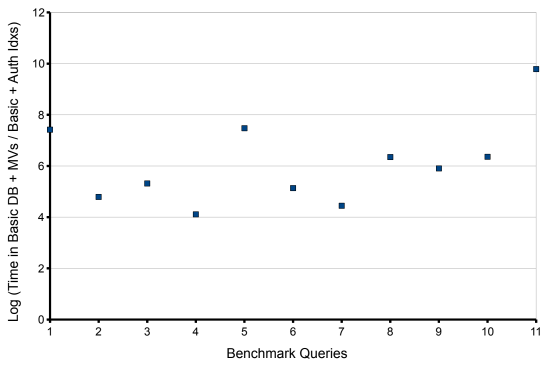

Figure 5: Performance of authorization indexes over Policy P1 in the presence of all materialized joins

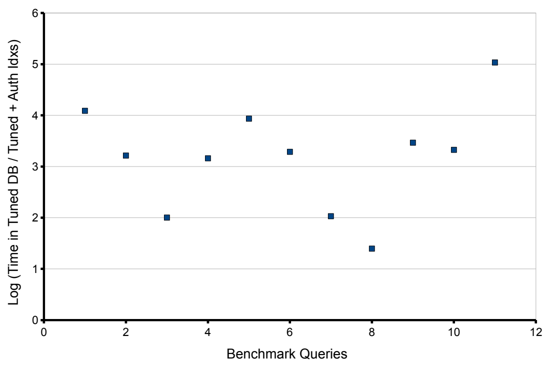

Figure 6: Performance of authorization indexes over Policy P1 in the tuned TPC-H DB

materialized views. Thus, authorization indexes can be very effective even for policies that fall within the class of supported materialized views.

As an additional data point, we also obtained an optimized physical design for the rewritten queries using the automated physical design tool that ships as a part of SQL Server (we did not give the design tool any bounds in storage requirement). The space required to store the suggested indexes was around 2 Gb (the space required to store the authorization indexes was around 875 MB). In order to evaluate if there is any additional benefit in having authorization indexes, we compared the execution time of the tuned rewritings with the rewritings generated by the greedy algorithm on the tuned database. The results are shown in Figure 6. The average speed up in execution time for the greedy rewritings (even over the tuned rewritings) was a factor of 11x.

We observed that there were interesting interactions between authorization indexes and the indexes suggested by the physical design tool. For instance, for TPC-H queries 3 and 9, the greedy algorithm picked an index intersection plan between a regular index (suggested by the physical design tool) and an authorization index in order to intersect the RIDS before the relevant tuples are fetched from the base relations. The plans (an example is shown in Figure 7) led to a speed up of 5x and 13x in the execution times of queries 3 and 9 respectively.

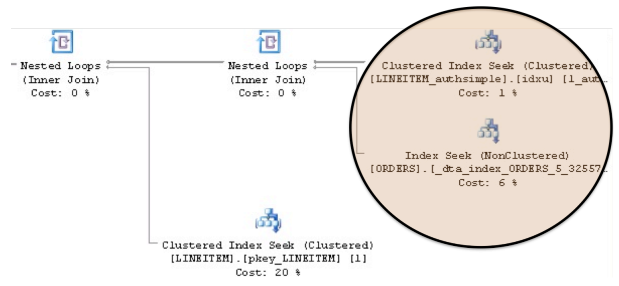
Nested Loops
(Inner Join)
Cost: 0 %
Nested Loops
(Inner Join)
Cost: 0 %
Clustered Index Seek (Clustered)
[LINEITEM_authsimple].[idxu] [l_au
Cost: 1 %
Index Seek (NonClustered)
[ORDERS].[_dta_index_ORDERS_5_32557
Cost: 6 %
Clustered Index Seek (Clustered)
[LINEITEM].[pkey_LINEITEM] [l]
Cost: 20 %

Figure 7: Snapshot of the query execution plan for Query 3 showing an index intersection plan between an authorization index and a regular index

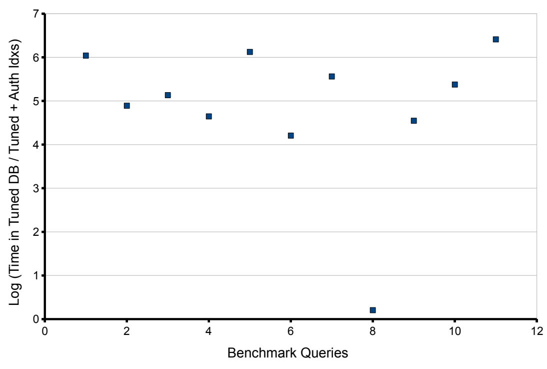

Figure 8: Performance of authorization indexes over Policy P2 in the tuned TPC-H DB

4.3 Results on complex policies - P2 and P3

Policy P1 was a simple authorization policy which could be equivalently rewritten as a SPJ query. To evaluate the effectiveness of authorization indexes we vary the complexity of the authorization policy by introducing negations (P2) and unions (P3). We first report the results for the authorization policy P2 which contains negations — we only report the results on the tuned database.

Results on Tuned Database: As mentioned earlier, as we cannot create "exact" materialized views for the complex policies, we use the automated physical design tool to suggest an appropriate physical design. The space required to store the suggested indexes was $\sim$ 500 Mb. We compared the execution time of the queries rewritten using authorization predicates on the tuned database with the rewritings generated by the greedy algorithm. The results are shown in Figure 8. The average speed up in execution time for the greedy rewritings over the tuned rewritings was by a factor of $\sim$ 30x.

Evaluation on Policy P3 Policy P3 constrains the users to view the lineitems corresponding to their orders where the parts are purchased from a set of regions. We vary the complexity of the policy by varying the number of regions by adding additional UNION clauses. In this experiment, we fix the query to be *select * from lineitem* and show the results for five "users" who can see the re-

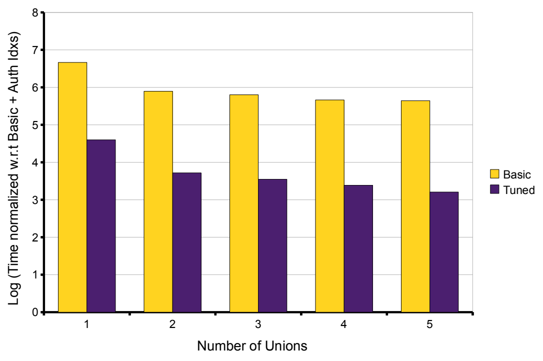

Figure 9: Performance of authorization indexes over Policy P3 with respect to Basic and Tuned TPC-H DB

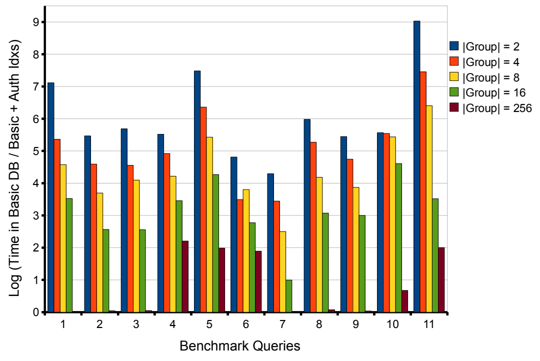

Figure 10: Performance of authorization indexes over Policy P1 with varying selectivities

sults corresponding to one to five regions - (i.e., user three's authorization policy would result in a rewritten query that has a union of three subqueries). The results are reported in Figure 9. On an average, the speed up in execution time of the local greedy rewritings over the rewritings with authorization predicates on the basic database and the database tuned using the physical design tool are 100x and 10x respectively. The space required to store the authorization indexes for policy P3 with 1, 2, 3, 4 and 5 regions is 64, 128, 192, 256, and 320 Mb respectively. The physical design tool was only able to select only a few relevant indexes with a total space of 14 Mb.

4.4 Selectivity of Authorization Predicates

If an user can view only a small fraction of the base relation's tuples, authorization indexes are likley to provide faster access to the records. As this fraction grows we would expect authorization indexes to have diminishing returns (as is the case with regular indexes). In this section, we evaluate the effectiveness of authorization indexes by varying the selectivity of the authorization policy P1. Policy P1 constrains a customer to view his own order. We vary the selectivity the authorization predicate on the policy P1 by grouping customers and allowing a customer to view the records of all customers in his group. We report the results for groups of size 2, 4, 8, 16, and 256. the results are reported in Figure 10. For groups of size 2 and 4, authorization indexes were used in almost every query (9 of 11 queries). However, for groups of size 256, less than a half of the benchmark queries were rewritten using authorization indexes. The authorization index on the Customer table had the most diminishing return and was excluded from every rewriting with more than 4 groups.

4.5 Summary of Results

The experimental results illustrating the average speed-up and maximum speed-up (across the TPC-H queries) for all the policies is summarized in Table 1.

The experimental results demonstrate that authorization indexes can significantly reduce the execution times of complex queries (with fine-grained authorizations) even when compared to a physical design that is highly tuned for the rewritten queries. For instance, for Policy P2 the maximum speed-up was close to two orders of magnitude (for TPC-H Query 5). Of course, this is not meant to be an exhaustive study and there are other important dimensions to consider (such as the cost of storing and maintaining the indexes). But, the results clearly demonstrate that authorization indexes merit more study.

Policy	Average speed-up	Maximum speed-up
P1	11x	32x
P2	34x	85x
P3	10x	24x (Policy with 5 unions)

Table 1: Summary of results

5. OPEN PROBLEMS

In this section, we briefly comment on some interesting research problems that arise from the tightly-coupled integration of authorizations and a query engine.

Compression Techniques: We have focused our discussion on SELECT queries. Users could have different authorization for SELECT and UPDATES (e.g., an analyst can read data but not update it). This would imply having multiple authorization indexes for different operations. Thus the space requirements of using authorization indexes could be non-trivial. It is interesting to examine appropriate compression techniques for such indexes. For instance, one can consider a bitmap index [12] like representation for the RIDS. Likewise, other compression techniques like run length encoding could also prove to be effective in compressing RID lists.

Authorization based Optimizations: Reasoning about authorization also opens the avenue for interesting optimizations of regular query processing. Consider a set of reporting queries that need to be run for a set of users (e.g., a set of managers in a hierarchy). Note that the rewritten queries for different users could have large sub-queries that are identical based on the "org-chart". A multi-query optimization scheme that is aware of the authorizations can compute a "minimal" subset of queries to run in order to cover all the original reporting queries. In general, the interaction of query processing and the complexity of authorization schemes (e.g., involving recursion) is relatively unexplored.

Authorization Semantics: Current FGA semantics are based on the "truman" model [13] which enables any query to be run on a selected subset of rows of the database (the rows being identified by appropriate predicates on each table). Ideally, the authorizations would also include: 1) restrictions on appropriate columns and 2) restrictions on computations that are allowed on that subset of rows and columns (e.g., can aggregate these rows but not filter them). We can enable this functionality using additional views but this loses an important advantage of enforcing authorizations using rewriting, namely transparency. Another important problem is the issue of information leakage — as discussed in Section 3.3, side channel attacks can be used to subvert authorizations. While a certain class of attacks can be handled by using the notion of safe plans, it is not clear how to generalize this notion for other side channels such as timing information. Designing an uniform FGA model that can permit only specific computations on data while taking into account any information leakage using side channels is an intersting avenue for future work.

Authorization and Encryption: There has been recent work on integrating encryption as a first class citizen in a database (e.g., [14, 15]) — there has not been much work that examines the integration of fine-grained authorizations and encryption. For instance, does a rewriting based approach suffice or can fine-grained authorizations be directly enforced using encryption (e.g., by using appropriate keys or a specific partial homomorphic encryption technique to restrict computations). Again, developing a unified FGA model that integrates (fine-grained) encryption with authorization is an interesting open problem.

6. RELATED WORK

The original query rewriting model for authorizations was proposed in [4]. Extensions for fine-grained authorizations are discussed in [1, 2, 3]. Oracle VPD [1] is a commerical offering that implements FGA using predicated grants.

In addition, the Oracle DBMS supports the notion of application contexts[16] which is a set of name-value pairs that is available to the application (e.g., the current employee id) — this can be potentially used to optimize the rewritten query for simple authorization policies in a similar fashion as authorization indexes. But for more complex policies (e.g., the query in 1.3), the application would have to reason about query equivalence between complex subexpressions in order to guarantee correctness and this is best handled by an authorization aware query optimizer (Section 3). Caching application contexts could still provide an interesting optimization for authorization indexes in certain cases — for instance, we could choose partial authorization indexes on the basis of which application contexts are cached.

As discussed previously, commerical systems support materialized views for only a limited class of expressions. For instance, the algorithms described in [17] are indicative of the state of the art and handle the following class of expressions — a view must be defined by a single- level SQL statement containing selections, (inner) joins, and an optional group-by. The FROM clause cannot contain derived tables, i.e. it must reference base tables, and subqueries are not allowed. The output of an aggregation view must include all grouping columns as output columns (because they define the key) and a count column. Aggregation functions are limited to sum and count. Thus, materialized views do not cater the complex expressions that arise more naturally for fine-grained authorizations.

7. CONCLUSIONS

The rewrite-then-optimize architecture for integrating authorization in a database introduces subtle performance problems that are hard to "tune" away. We argue the need for raising authorization awareness in a DBMS [3]. In particular, we introduced the notion of an authorization index. Our initial experiments demonstrate that such an index structure (along with an authorization aware optimizer) can provide significant benefits for complex TPC-H queries. We consider this paper as a first step — we believe the tightly-coupled integration of authorization and query processing can be a rich source for interesting problems.

8. REFERENCES

[1] Oracle Corporation, "Oracle virtual private database," http://www.oracle.com/.

[2] Q. Wang *et al.*, "On the correctness criteria of fine-grained access control in relational databases," in *VLDB*, 2007.

[3] S. Chaudhuri, T. Dutta, and S. Sudarshan, "Fine grained authorization through predicated grants," in *ICDE*, 2007.

[4] M. Stonebraker and E. Wong, "Access control in a relational database management system by query modification," in *ACM CSC-ER*, 1974.

[5] "Special issue on self-managing systems," *IEEE Data Eng. Bull.*, vol. 29, no. 3, 2006.

[6] M. Stonebraker, "The case for partial indexes," *SIGMOD Rec.*, vol. 18, no. 4, pp. 4–11, 1989.

[7] T. Griffin and L. Libkin, "Incremental maintenance of views with duplicates," in *SIGMOD*, 1995.

[8] S. Chaudhuri and M. Vardi, "Optimization of real conjunctive queries," in *PODS*, 1993.

[9] Y. Sagiv and M. Yannakakis, "Equivalences among relational expressions with the union and difference operators," *J. ACM*, vol. 27, no. 4, 1980.

[10] G. Graefe, "The Cascades Framework for Query Optimization," *IEEE Data(base) Engineering Bulletin*, vol. 18, pp. 19–29, 1995.

[11] G. Kabra, R. Ramamurthy, and S. Sudarshan, "Redundancy and information leakage in fine-grained access control," in *SIGMOD*, 2006.

[12] P. O'Neil and D. Quass, "Improved query performance with variant indexes," in *SIGMOD*, 1997.

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

[14] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "Cryptdb: protecting confidentiality with encrypted query processing," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, 2011.

[15] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan, "Orthogonal security with cipherbase," in *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*, 2013.

[16] Oracle Corporation, "Implementing application context and fine-grained access control," http://docs.oracle.com/.

[17] J. Goldstein and P. åke Larson, "Optimizing queries using materialized views: A practical, scalable solution," 2001, pp. 331–342.

[3] If you support our cause, you can help by citing this paper. We also welcome donations.