Enabling Intensional Access Control via Preference-aware Query Optimization

Nicholas L. Farnan, Adam J. Lee, Panos K. Chrysanthis[1]
{nlf4, adamlee, panos}@cs.pitt.edu

Ting Yu[2]
yu@csc.ncsu.edu

[1]Department of Computer Science, University of Pittsburgh, Pittsburgh, PA, USA
[2]Department of Computer Science, North Carolina State University, Raleigh, NC, USA

ABSTRACT

Although the declarative nature of SQL provides great utility to database users, its use in *distributed* database management systems can result in unintended consequences to user privacy over the course of query evaluation. By allowing users to merely say *what* data they are interested in accessing without providing guidance regarding *how* to retrieve it, query optimizers can generate plans that leak sensitive query intension. To address these types of issues, we have created a framework that empowers users with the ability to specify access controls on the intension of their queries through extensions to the SQL SELECT statement. In this demonstration, we present a version of PostgreSQL's query optimizer that we have modified to produce plans that respect these constraints while optimizing user-specified SQL queries in terms of performance.

Categories and Subject Descriptors

K.4.1 [**Computers and Society**]: Public Policy Issues—*Privacy*; H.2.4 [**Database Management**]: Systems—*Distributed databases, Query processing*

Keywords

query optimization, privacy, distributed databases, preference SQL

1. INTRODUCTION

The declarative nature of SQL has been a major strength of relational database systems: users can simply specify *what* data they are interested in accessing and the database management system will determine the *best* plan for accessing that data. Plans for query evaluation detail what operations need to be performed to produce the result of an SQL query, what order these operations should be completed in, and what server should actually execute each operation. Traditionally, the *best* plan has been simply the plan that returns results to the user in the shortest amount of time. When user queries are issued to distributed database management systems, however, two plans generating the same results for the same query can vary greatly in how they disseminate portions of that query during its evaluation.

In issuing a query to a database management system, the user is specifying exactly what information she wishes to retrieve from the system. This description is called the *intension* of a user's query. Users may consider some of the intension of their queries to be sensitive and wish to keep such sensitive intension hidden from remote servers. Hence, in order to protect their privacy, users must be empowered with the ability to control who is granted access to the intension of their queries.

In centralized database systems, the whole of the intension of a user's query is disclosed to a single system that optimizes and evaluates the query itself. As such, *how* the system optimizes and evaluates the query has no effect on the intensional privacy afforded to the user. With distributed database systems, however, though the whole intension of a user query is still disclosed to the optimizer, the optimizer may construct an evaluation plan for the query that distributes portions of query intension to a large number of remote servers needed to evaluate some part of that plan. Further, the user is left completely unaware of how the intension of her query is disseminated during its evaluation. Though these servers are trusted to store data and correctly process queries, the user may not trust them to learn sensitive portions of the intension of her query. Hence, access controls over the intension of SQL queries are needed to uphold user privacy.

Towards this goal, we have developed a framework for users to author declarative constraints on the query evaluation plans generated by a query optimizer for resolving SQL queries. The goal of our demonstration is to show how our framework can be used to establish access controls and protect the intension of SQL queries in distributed database systems, such as by keeping sensitive join conditions from being revealed to untrusted servers.

The remainder of this demonstration paper is structured as follows: Section 2 will describe the system model that we assume for this work and present a motivating example that will be used as the core of our demonstration. Section 3 will describe our framework, the extensions to SQL that we have developed as an interface to it, and further our implementation of this framework within PostgreSQL's optimizer. Finally, Section 4 will describe the details of the demonstration.

SACMAT'13, June 12–14, 2013, Amsterdam, The Netherlands.

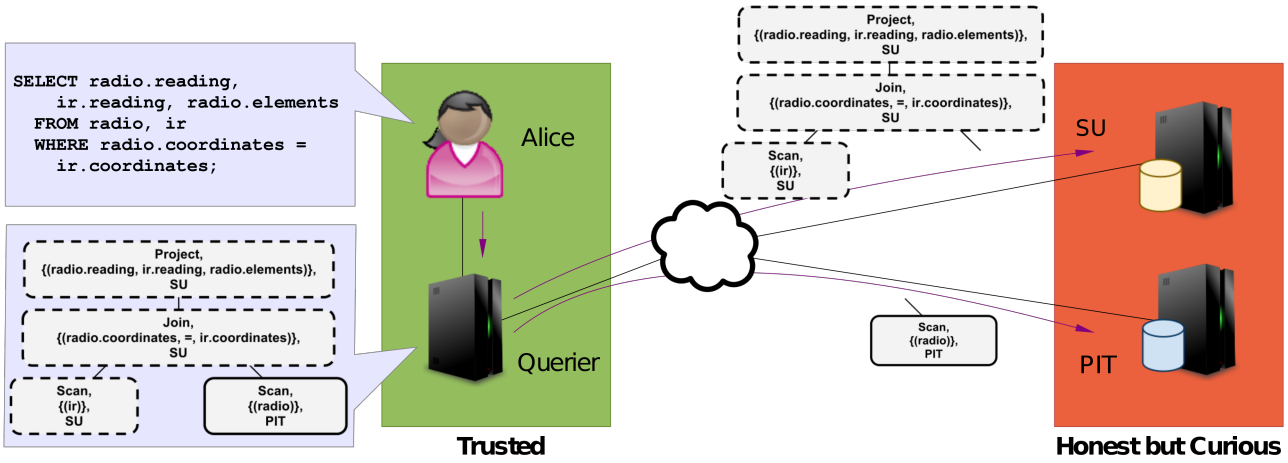

Figure 1: An illustration of the workflow of distributed query processing. Here, Alice's issuance of Query 1 and the dissemination of the query plan presented in Figure 2 is shown.

2. SYSTEM MODEL AND USE CASE

In our work, we consider the system model illustrated in Figure 1. The distributed query evaluation process begins when a user constructs the query she wishes to issue and passes it to a query optimizer. The optimizer then determines the best plan for evaluating the user's query, and distributes a portion of this plan to each server needed to evaluate the query. These database servers will evaluate their assigned portions of the query, combine their intermediate results as needed, and return the final query result to the user.

As can be seen here, the user has no part in deciding to or actually disseminating partial query plans to database servers. This provides the opportunity for violations of the user's privacy to occur. To highlight this issue, consider the following example:

Example 1. Alice is an astronomy researcher working at the Polytechnic Institute of Technology (PIT). Alice has recently decided to investigate whether combined readings from radio and infrared telescopes viewing the same stellar object can be used to efficiently predict the elements that make up that object in a novel way. Towards investigating this theory, she will query a small database of radio telescope readings and the elements known to be found in those objects that is maintained by PIT (in a table called simply "radio"). To get infrared readings to work with as well, however, Alice will have to query a much larger database maintained by State University (SU), specifically their "ir" table. For the greater good, the PIT and SU astronomy departments allow each other access to their respective databases. In spite of this, though, Alice would like to keep her new theory secret from researchers at SU to ensure she is the first to publish it in the case that she is, indeed, on to something.

Alice is interested in the result of the following SQL query:

```
SELECT radio.reading, ir.reading, radio.elements
  FROM radio, ir
  WHERE radio.coordinates = ir.coordinates;
```
(1)

Without the aid of our framework, an optimizer could produce a plan like the one shown in Figure 2. Here (and

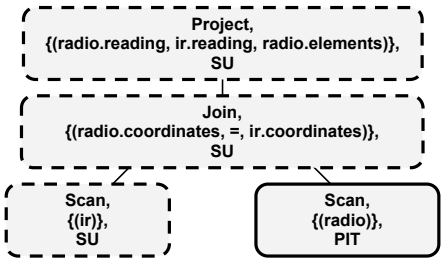

Figure 2: A potential plan for evaluating Alice's query that reveals sensitive intension to the adversarial server SU. The execution location of each node is represented by its border. Nodes with a dashed border are to be executed by SU, while those with a solid border are assigned to PIT.

throughout this demonstration paper) we represent a query plans as trees of relational algebra operations where the leaves of this tree scan the base tables containing needed data, the root produces the result of the query, and intermediate nodes process data from the relations scanned at the leaves. The relational algebra operations that make up the nodes of this tree can be represented as ternaries of the following form:

$$\langle op, params, p \rangle$$

Here, *op* represents the operation to be performed. We consider valid operators to be either one of the core relational algebra operators (selection, projection, rename, set union, set difference, and cross product), a scan, or a join. *params* is a set of sets that represents the parameters to that operation (e.g., the table to be scanned, the condition on a join of two relations, the attributes that tuples should be projected down to, etc.); and p represents the site annotated to evaluate this operation.

It can be seen in Figure 2 that this example plan reveals to SU sensitive aspects of the intension of Alice's query: by having SU evaluate the join of data from the radio and ir tables, SU learns that Alice is interested in both radio and ir telescope readings, the very information that she wished to

keep private. To solve this problem for Alice and users like her, we provide a way for users to specify what portions of the intension of their queries they consider to be sensitive, and whom that sensitive intension should be kept from.

3. OUR APPROACH

To give users control over what parties are granted access to sensitive portions of the intension of their queries, we modify the query optimizer from our system model to accept not only user queries, but also constraints on the plans that can be generated to evaluate those queries. The optimizer will then utilize these constraints during the optimization process and produce plans that uphold them. Through this approach, we can effectively include privacy as an optimization metric.

3.1 SQL Extensions

Constraints can be specified as either *requirements* (constraints that *must* be upheld by any plan to evaluate the query) or *preferences* (constraints that may not be upheld if they conflict with other constraints or render the optimizer unable to generate a feasible query plan). To express each of these to the optimizer, in [1] we developed two extensions to the SQL `SELECT` statement: the `REQUIRING` and `PREFERRING` clauses.

Required constraints are fairly straightforward: Alice (from Example 1) could require that any plan produced by the optimizer for her query presented in Section 2 keeps her interest in radio and infrared telescope readings from SU. The `REQUIRING` clause takes the following general form:

`REQUIRING` *condition* `HOLDS OVER` *node descriptors*
`AND` *another condition* `HOLDS OVER` *node descriptors*

Node descriptors are used to identify the intensional region that the user wishes to constrain. Node descriptors are defined as a mirror to our representation of query tree nodes, ternaries consisting of *op*, *params*, and *p*. They are used to "match" query tree nodes that contain sensitive pieces of query intension. A node descriptor designed to specify the mention of radio and infrared telescope readings as sensitive, for example, would match against the project operation in Figure 2 as this node operates on both the `radio` table's `reading` attribute and the `ir` table's `reading` attribute. In node descriptors, "*" is used as a general wildcard for portions of the ternary that the user wishes a given node descriptor to match against any value of. To construct a node descriptor matching any query tree node that operates on radio and infrared readings, for example, the user could instantiate *op* and *p* as "*" while defining *params* as a combination of these two attributes. Finally, the "@" character is used to identify free variables over which conditions can be authored. By stating the *p* part of her node descriptor to be a free variable, and authoring a condition that that free variable should not have the value SU, Alice can inform the optimizer of her constraint that query tree nodes matching her node descriptor are not evaluated by SU. This constraint could be expressed through our `REQUIRING` clause as follows:

```
SELECT radio.reading, ir.reading, radio.elements
  FROM radio, ir
  WHERE radio.coordinates = ir.coordinates
  REQUIRING @p <> SU HOLDS OVER
    <*, {(radio.reading, ir.reading)}, @p>;
```
(2)

Privacy notions are rarely so straightforward, however. User conceptions of privacy are inherently personal and situationally dependent. The `PREFERRING` clause allows our framework to capture such complex privacy notions. While the `PREFERRING` clause is made up of the same basic *constraint* `HOLDS OVER` *node descriptors* building blocks as the `REQUIRING` clause, it makes use of an additional keyword to bind them together. Where the `REQUIRING` clause uses only `AND` to join individual constraints together, constraints in the `PREFERRING` clause can also be joined using `CASCADE`. This second keyword is needed to establish the priority of different constraints relative to one another to form partially ordered preference structures. While two constraints joined by `AND` are considered equally preferred, any constraint before a given `CASCADE` is considered more important by the optimizer than any that are listed after that `CASCADE`.

As an example, let us say that Alice would prefer to keep SU from learning that she is interested in both infrared telescope readings and radio telescope readings. Alice would like it if neither of those pieces of intension is revealed to SU, but if that is not possible, she would prefer that either one or the other is revealed as opposed to revealing her interest in both. This relatively complex notion of query privacy can be succinctly captured in our framework using simply the `AND` keyword (as Alice does not consider the revelation of either her interest in radio readings or her interest in infrared readings to be more important than the other).

```
SELECT radio.reading, ir.reading, radio.elements
  FROM radio, ir
  WHERE radio.coordinates = ir.coordinates
  PREFERRING  @p <> SU HOLDS OVER
      <*, {(radio.reading)}, @p>
    AND @p <> SU HOLDS OVER
      <*, {(ir.reading)}, @p>;
```
(3)

If Alice further wanted all join operations to be performed by PIT's database server, she could add this as a less important constraint using the `CASCADE` keyword:

```
SELECT radio.reading, ir.reading, radio.elements
  FROM radio, ir
  WHERE radio.coordinates = ir.coordinates
  PREFERRING @p <> SU HOLDS OVER
      <*, {(radio.reading)}, @p>
    AND @p <> SU HOLDS OVER
      <*, {(ir.reading)}, @p>
    CASCADE @p == PIT HOLDS OVER
      <Join, *, @p>;
```
(4)

Upon receiving this extended query specification with preferred constraints, the optimizer will attempt to construct a plan that upholds the constraints for keeping `radio.reading` and `ir.reading` from SU (or both). If it can construct a plan that further executes all joins at PIT, all the better. The optimizer will only emit a plan that evaluates all joins at PIT and reveals sensitive information to SU if revealing such information is unavoidable in any plan the optimizer is able to build. Because Alice states that keeping her interests secret from SU is more important to her than executing joins at PIT, the optimizer will never trade off revealing information to SU in favor of executing joins at PIT.

Such an optimization process could result in the query plan shown in Figure 3. This query plan upholds all of the example constraints mentioned in this section.

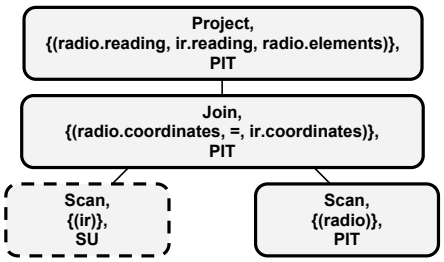

Figure 3: A potential plan for evaluating Alice's query that protects her privacy. The execution location of each node is represented by its border. Nodes with a dashed border are to be executed by SU, while those with a solid border are assigned to PIT.

3.2 Implementation

PostgreSQL is a widely-used, open-source, object-relational database management system [3]. In all, it consists of over 700,000 lines of code. To support our extensions to SQL, we have modified the optimizer contained within PostgreSQL.

As PostgreSQL is not a distributed database management system, the first step in adapting its optimizer to our needs was to modify the optimizer to reason about execution locations of the individual operations that make up an overall query plan. This required modifying all query plan data structures to incorporate execution location state, iterating through possible execution locations of all operations added to a query plan, and revising the plan cost estimator to account for parallel operations occurring at different sites and data transfer times.

Once this was done, we adapted the optimization process to utilize `REQUIRING` clauses to prune the optimization search space. Any sub-plan that violates a required constraint cannot be emitted as part of a final query plan and can hence be discarded from further consideration during plan construction. To support the `PREFERRING` clause, we have modified PostgreSQL's dynamic programming approach to query optimization to maintain only the most highly preferred plans over the course of optimization. This heuristic allows us include user preferences and intensional access controls as optimization metrics while offering optimization performance near that of the unmodified optimizer. All together, our modifications to PostgreSQL touched a subset of the code base totaling over 60,000 lines of code.

4. DEMONSTRATION

In this demonstration, we first illustrate the how easily a user with knowledge of SQL can specify constraints on her queries and second, the effect of such constraints on the query optimization process. We show how, when passed into our modified query optimizer alongside a user query, intensional access controls specified through such constraints can drastically change the makeup of the plan devised by the optimizer to protect user privacy with little overhead to optimization time. We present the example scenario and query shown in Section 2, and the different query plans produced by our optimizer when different constraints are issued with the query. In doing so, we present only a small subset of the expressive power available through our framework.

To set up this demonstration, we establish databases simulating PIT and SU's stores of telescope readings. Specifically, we seed two databases with samples of data gleaned from the Astroshelf project [2], and then adjust their catalog metadata to simulate the optimization of queries over large scale astronomical databases without having to maintain a large scale data store for this demonstration. We assume that SU stores around 4TB of data, while PIT maintains only 1TB, and scale the metadata accordingly.

As we demonstrate only the query optimization process, only the catalog metadata for these databases is accessed in our demonstration. By seeding these databases with sample data from Astroshelf, we can ensure that our demonstration presents realistic examples of query optimization.

We begin our demonstration by presenting the optimization of Alice's query without any constraints on the resulting plan. We present a graphical representation of the resulting evaluation plan, as well as the time required to optimize the query and the estimated run time of the evaluation plan. With this baseline in hand, we then show how the addition of the required constraints shown in Query 1 in Section 3.1 causes the generation of structurally different plans and the minimal effect that the addition of such constraints has on optimization time and estimated runtime. We further demonstrate how our optimizer can detect the presence of conflicting requirements (e.g., Alice requires that all joins happen at PIT and also that no joins happen at PIT), and inform the user of such.

From here, we show the effect of preferred constraints on the optimization process by optimizing Queries 3 and 4 from Section 3.1. We further show how conflicting preferred constraints are of no consequence to our optimizer. If they are separated by a `CASCADE`, the optimizer will produce a plan upholding the one given before the `CASCADE`. If they are separated by an `AND`, the optimizer will support whichever allows for the creation of a faster evaluation plan.

Finally, we demonstrate how other policies idioms can be expressed using our framework. For example, we show how separation of duty controls can be implemented through our constraints (e.g., do not allow any server that scans the infrared readings table to perform any join operations).

Acknowledgments. This work was supported in part by the National Science Foundation under awards CCF–0916015, CNS–0964295, CNS–0914946, CNS–0747247, and OIA-1028162. This work was further partially supported by a gift from EMC/Greenplum.

5. REFERENCES

[1] N. L. Farnan, A. J. Lee, P. K. Chrysanthis, and T. Yu. Don't reveal my intension: Protecting user privacy using declarative preferences during distributed query processing. In *ESORICS*, pages 628–647, 2011.

[2] P. Neophytou, R. Gheorghiu, R. Hachey, T. Luciani, D. Bao, A. Labrinidis, E. G. Marai, and P. K. Chrysanthis. Astroshelf: understanding the universe through scalable navigation of a galaxy of annotations. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 713–716, 2012.

[3] The PostgreSQL Global Development Group. Postgresql. `http://www.postgresql.org/`, Dec. 2012.