PAQO: A Preference-Aware Query Optimizer for PostgreSQL

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

Ting Yu[2,3]
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
[3]Qatar Computing Research Institute, Doha, Qatar

ABSTRACT

Although the declarative nature of SQL provides great utility to database users, its use in *distributed* database management systems can leave users unaware of which servers in the system are evaluating portions of their queries. 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 with unintended consequences to the user (e.g., violating user privacy by revealing sensitive portions of a user's query to untrusted servers, or impacting result freshness by pulling data from stale data stores). To address these types of issues, we have created a framework that empowers users with the ability to specify constraints on the kinds of plans that can be produced by the optimizer to evaluate their queries. Such constraints are specified through an extended version of SQL that we have developed which we call PASQL. With this proposal, we aim to demonstrate PAQO, a version of PostgreSQL's query optimizer that we have modified to produce plans that respect constraints specified through PASQL while optimizing user-specified SQL queries in terms of performance.

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 (DBMS) will determine the *best* plan for accessing that data. 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 DBMSs, 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. These variances in where portions of a query are evaluated can cause a number of problems for the user.

Consider, for example, a user's wish to uphold a separation of duty restriction on certain portions of her query, i.e., the query contains operations that should not be evaluated by the same server. If a distributed database system were to construct and execute a plan that evaluated any such operations on the same server, the user's separation of duty requirement would be violated. This situation would be especially problematic if this requirement is dictated by the user's employer and violating it could put her job at risk. As another example, consider a user issuing a query that contains sensitive information (e.g., interest in a medical condition or personally identifiable information). Generating a plan that reveals such information to a server the user considers untrustworthy would violate the user's privacy. Issues such as these are further complicated by the fact that after issuing an SQL query, users are left completely unaware of how it is disseminated and evaluated. This leaves users unaware if a separation of duty requirement is violated or sensitive information is divulged to untrusted sites.

To assuage problems such as these, we have developed a preference-aware query optimizer (PAQO) that can uphold declarative, user-specified constraints on the query plans generated during the optimization process. Such constraints are capable of capturing user privacy concerns and are specified by users via an extended versions of SQL we have developed called Preference-Aware SQL (PASQL). PASQL adds two extension clauses to the SQL SELECT statement that have been formally established in [1].

In this demonstration, we illustrate the functionality of PAQO. Specifically, we show how the use of constraints specified via PASQL affects the optimization process. Towards this end, we visualize the incremental sub-plans constructed by the optimizer and used as the building blocks in generating a final query plan for queries with and without PASQL constraints attached. The visualizations of these *intermediate* plans demonstrate how PAQO is able to accommodate user preferences in a declarative manner and ensure that the plans it generates adhere to such preferences.

In the next section, we describe our assumed system model and present a motivating example that forms the core of our demonstration. Section 3 describes our framework, the extensions to SQL that we have developed as an interface to it, and our implementation of this framework within PostgreSQL's optimizer. Section 4 describes our demonstration.

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 submits a query to the system which the system

Proceedings of the VLDB Endowment, Vol. 6, No. 12
Copyright 2013 VLDB Endowment 2150-8097/13/10... $ 10.00.

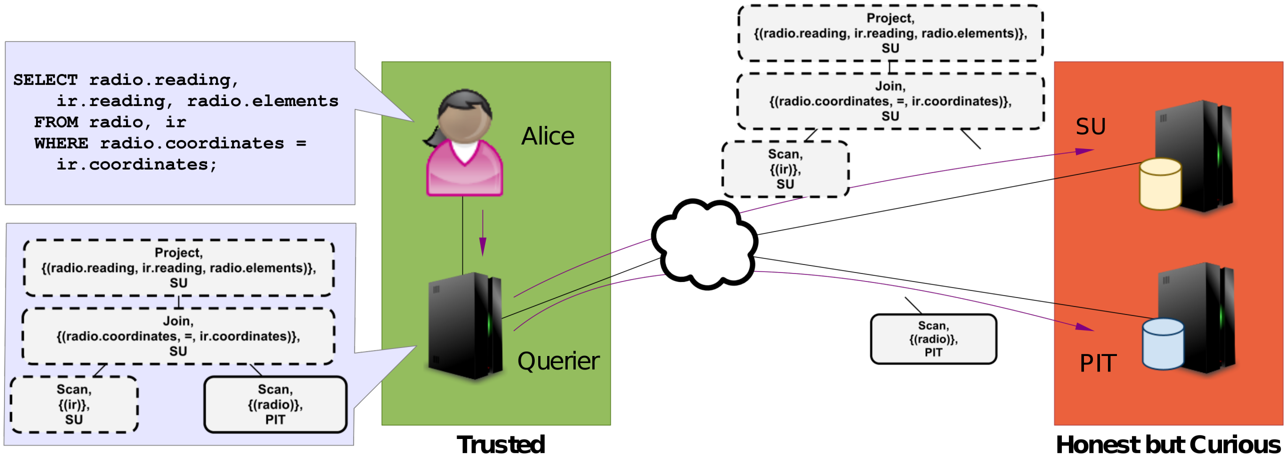
SELECT radio.reading,
ir.reading, radio.elements
FROM radio, ir
WHERE radio.coordinates =
ir.coordinates;
Alice
Querier
Trusted
Project, {(radio.reading, ir.reading, radio.elements)}, SU
Join, {(radio.coordinates, =, ir.coordinates)}, SU
Scan, {(ir)}, SU
Scan, {(radio)}, PIT
SU
PIT
Honest but Curious

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. Query plan nodes are shown as ternaries consisting of the relational algebra operation to be performed, the arguments to that operation, and the site assigned to evaluate the operation.

validates and passes on to a query optimizer. The optimizer then determines the best plan for evaluating this query, and distributes portions of this plan to each server needed to evaluate the query. These servers will evaluate their assigned portions of the query, combine their intermediate results as needed, and return the final query result to the user.

In this traditional setting, the user has no part in deciding to disseminate partial query plans to database servers. This provides the opportunity for issues such as those outlined in Section 1 to occur. To highlight such issues, 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                                      (1)
  WHERE radio.coordinates = ir.coordinates;
```

Without the aid of our framework, an optimizer could produce a query evaluation plan like the one shown in Figure 2. The nodes of this query plan represent the relational algebra operations to be performed. Specifically, they are ternaries of the form $\langle$ *op*, *params*, *p* $\rangle$. In these ternaries, *op* represents the relational operation to be performed; *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.

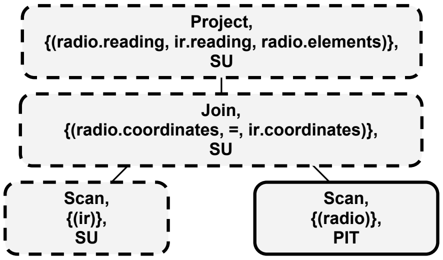
Project, {(radio.reading, ir.reading, radio.elements)}, SU
Join, {(radio.coordinates, =, ir.coordinates)}, SU
Scan, {(ir)}, SU
Scan, {(radio)}, PIT

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.

It can be seen that the example plan from Figure 2 reveals to SU sensitive aspects 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 such problems for users like Alice, we empower users to constrain the types of plans generated to evaluate their queries.

3. OUR APPROACH

PAQO differs from the query optimizer shown in our system model in that it accepts not only user queries, but also constraints on the plans that can be generated to evaluate those queries. These constraints are then effectively utilized as optimization metrics.

3.1 SQL Extensions

The constraints supported by PAQO can be specified as either *requirements* (constraints that *must* be upheld by any plan to evaluate the query) or *preferences* (constraints that the use does not consider necessary but would like to have upheld by a plan to evaluate her query). To express each of these to the optimizer, in [1] we developed the two extensions to the SQL `SELECT` for use in PASQL: the `REQUIRING` and `PREFERRING` clauses.[1]

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 1* `HOLDS OVER` *node descriptor 1*
[`AND` *condition 2* `HOLDS OVER` *node descriptor 2*]

Node descriptors are used to identify the portions of the query that the user wishes to constrain the evaluation of. 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 the user would prefer to have evaluated in a specific way. 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          (2)
  REQUIRING @p <> SU HOLDS OVER
    <*, {(radio.reading, ir.reading)}, @p>;
```

User preferences are not always so straightforward, however. In order to allow users to express complex and partially ordered hierarchies of constraints, PASQL includes the `PREFERRING` clause. The `PREFERRING` clause is made up of the same basic *constraint* `HOLDS OVER` *node descriptors* building blocks as the `REQUIRING` clause. It is differentiated in that the optimizer may choose a plan to evaluate the query that does not uphold some (or any) constraints from a `PREFERRING` clause. Users can also indicate that certain `PREFERRING` clause constraints as more important than others.

Returning to our 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 aspects of her query were 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 her interest in either radio or 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>                     (3)
    AND @p <> SU HOLDS OVER
     <*, {(ir.reading)}, @p>;
```

Note that the `AND` keyword in a `PREFERRING` clause indicates to PAQO that it separates two constraints that the user considers equally preferred. Each may be independently selected by PAQO to be upheld over the other. PAQO will consider a plan that upholds both to be better than a plan that upholds only one (and similarly, a plan that upholds one is better than a plan that upholds neither). A plan that upholds only the first constraint is considered just as good as a plan that upholds only the the second, however. If Alice wished to add another constraint that should be considered less important, that all join operations are performed by PIT's database server, for example, she could use the `CASCADE` keyword instead of `AND` before this constraint:

```
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                         (4)
     <*, {(ir.reading)}, @p>
    CASCADE @q == PIT HOLDS OVER
     <Join, *, @q>;
```

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`. PAQO further allows the use of multiple `CASCADE`s in a `PREFERRING` clause to denote multiple levels of preference. 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. 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 not 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.

3.2 Implementation

PostgreSQL is a widely-used, open-source, object-relational DBMS [4]. It consists of over 700,000 lines of code. PostgreSQL serves as the basis for PAQO. We have extensively

[1] Though we utilize similar syntax to that presented in [2], we apply user preferences to query plan generation while [2] worked on ordering query results.

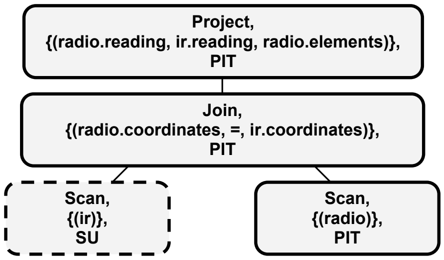
Project,
{(radio.reading, ir.reading, radio.elements)},
PIT
Join,
{(radio.coordinates, =, ir.coordinates)},
PIT
Scan,
{(ir)},
SU
Scan,
{(radio)},
PIT

Figure 3: A privacy-preserving plan for evaluating Alice's query.

modified PostgreSQL's parser and optimizer to support PASQL constraints. Our modifications to PostgreSQL touch a subset of the code base totaling over 60,000 lines of code.

Parsing and Data Structures. To be able to process PASQL constraints during optimization, we first need to parse these constraints out of incoming queries. As such, we extended PostgreSQL's query parser to support PASQL's `REQUIRING` and `PREFERRING` clauses. This further necessitated the creation of new data structures to house the additional output of this modified parser. Further, as PostgreSQL is not a distributed DBMS, we needed to extend all query plan data structures to include execution location state.

Cost Estimation. We have rewritten the cost estimation functions of PostgreSQL to account for distributed query plan execution. As a first step toward this goal, we included network transfer costs to PostgreSQL's cost estimations for all operations on relations coming from a different site. This was accomplished by scaling its calculation of disk read costs to Internet speeds. We further account for the parallel execution of applicable sub-trees of the query plan at different sites. Additionally, we assume that tuples can be streamed from site to site as they are generated.

Optimization. With all of the above modifications in hand, we were able to implement preference-aware query optimization through two changes to the generation of new intermediate query plans (though these same changes also apply to the generation of final query plans). First, 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. Second, 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 as optimization metrics while offering optimization performance near that of the unmodified optimizer. It is enforced in two steps during the optimization process. For each intermediate plan generated, the optimizer will determine which preferred constraints (if any) it upholds. This information is stored in a list of bitmaps (each representing a different preference level) that is attached to the internal representation of each plan. These plan bitmap lists are then compared to efficiently determine the relative preference of different intermediate query plans. In general, a plan is considered more preferred if it upholds more constraints at a higher level of preference.

Evaluation. We have established the correctness of PAQO by conducting a case study utilizing the example scenario we demonstrate here.

4. DEMONSTRATION

In this demonstration, we first illustrate how easily a user with knowledge of SQL and site topology can specify constraints on her queries and second, the effect of such constraints on the query optimization process. We show how, when passed to PAQO alongside a user query, user preferences can alter the optimization process to ensure they are supported by resulting plans. We present the example scenario and query shown in Section 2, and a look at the different intermediate plans generated by optimizing Alice's query with and without PASQL constraints applied.

To set up this demonstration, we consulted with the authors of the Astroshelf project [3] to create database tables simulating PIT and SU's stores of telescope readings. Due to practical limitations of the demonstration, we store relatively low cardinality tables on the demonstration machine and scale table size meta data read in during optimization to table sizes of 1TB for PIT and 4TB for SU. This allows us to effectively simulate the optimization of queries over large scale astronomical data.

As a baseline, our demonstration presents the optimization of Alice's query without any constraints on the resulting plan. We show graphical representations of the intermediate plans constructed by PAQO during the optimization process, as well as their estimated costs. In parallel with this, we similarly present an overview of the process for optimizing Queries 2 and 3 from Section 3.1. We use Query 2 to show how required constraints are used to trim the search space by dismissing any further processing on intermediate plans that violate a required constraint. We further demonstrate (using Query 3) how, in a manner similar to PostgreSQL's maintenance of different plans for generating the same intermediate relation but in a different sorted order, PAQO maintains different plans that uphold different user preferences. After exploring the optimization processes for these three queries, we show that our optimizer generates final query evaluation plans that respect Alice's specified constraints. We further discuss the minimal impact of our modification on query optimization time. Finally, we invite participants to submit queries if time permits.

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. It 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] W. Kießling and G. Köstler. Preference SQL - design, implementation, experiences. VLDB, pages 990–1001, 2002.

[3] 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 *SIGMOD*, pages 713–716, 2012.

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