PAQO: Preference-Aware Query Optimization for Decentralized Database Systems

Nicholas L. Farnan [#1], Adam J. Lee [#2], Panos K. Chrysanthis [#3], Ting Yu [*‡4]

[#] *Department of Computer Science, University of Pittsburgh*
Pittsburgh, PA, USA
{ [1]nlf4, [2]adamlee, [3]panos }@cs.pitt.edu

[*] *Department of Computer Science, North Carolina State University*
Raleigh, NC, USA
[‡] *Qatar Computing Research Institute*
Doha, Qatar
[4]yu@csc.ncsu.edu

***Abstract*—The declarative nature of SQL has traditionally been a major strength. Users simply state *what* information they are interested in, and the database management system determines the best plan for retrieving it. A consequence of this model is that should a user ever *want* to specify some aspect of how their queries are evaluated (e.g., a preference to read data from a specific replica, or a requirement for all joins to be performed by a single server), they are unable to. This can leave database administrators shoehorning evaluation preferences into database cost models. Further, for distributed database users, it can result in query evaluation plans that violate data handling best practices or the privacy of the user. To address such issues, we have developed a framework for declarative, user-specified constraints on the query optimization process and implemented it within PosgreSQL. Our Preference-Aware Query Optimizer (PAQO) upholds both strict requirements and partially ordered preferences that are issued alongside of the queries that it processes. In this paper, we present the design of PAQO and thoroughly evaluate its performance.**

I. INTRODUCTION

The declarative nature of SQL has traditionally been a major boon to users querying relational databases. It allows users to avoid dealing with the complexities of query planning and optimization. They are able to simply specify *what* information they want and then allow the system to devise a plan for *how* to retrieve it. Unfortunately, this simplicity leaves users unable to control how their queries are evaluated when they need or want to. The growing popularity of distributed database systems makes this lack of control increasingly problematic.

Consider a corporate user operating on data from several of her company's partners who is instructed not to reveal to either of the partners how she is combining their data. Traditionally, she would need to make separate queries to each partner database and then combine the results herself. Even if both servers were part of a distributed system this user could query, doing so might result in a condition for joining different partner data tables being revealed to one of the partner database servers. As another example, consider a user that issues queries containing sensitive or private information (e.g., queries containing medical patient names or illnesses) to a distributed database system. Issuing such queries via a traditional query optimizer could violate the user's privacy by revealing sensitive information to a server needed to resolve the query, but not fully trusted (e.g., an insurance company server, or a database server in a different hospital from the user). These cases are made even more problematic by the fact that these users would not have known *how* their queries were evaluated (they would only receive the results of their query). They would be left unaware if any problems occurred.

We have developed a Preference-Aware Query Optimizer (PAQO) and implemented it within PostgreSQL [22]. PAQO allows users to specify declarative preferences on the query optimization process and resulting query plans. We refer to these declarative preferences as *constraints*. For example:

- A doctor issuing a query over data stored at several different hospitals can constrain the optimizer to produce a plan that only executes operations involving patient names at the doctor's own hospital.
- When issuing a query that operates on a relation replicated across several different sites, users could state a preference for accessing this relation via the primary server while still allowing the optimizer to fall back to eventually-consistent replicas if needed.
- A user could require that specific portions of her query are evaluated by different database servers to uphold separation of duty policies (e.g., any site that performs a scan on a relation it hosts should not perform any joins, as needed by our corporate user mentioned above).

We have developed extensions to the SQL `SELECT` statement that allow users to express constraints as either requirements (constraints that *must* be upheld by any plan to evaluate their query), or preferences (constraints the user would prefer to be upheld but does not consider necessary) [7]. PAQO interprets constraints expressed through these SQL extensions and upholds them *during* the query optimization process. By doing so, PAQO ensures that user-specified constraints are upheld by plans for evaluating their corresponding queries at a minimal cost to the cost of evaluating the query.

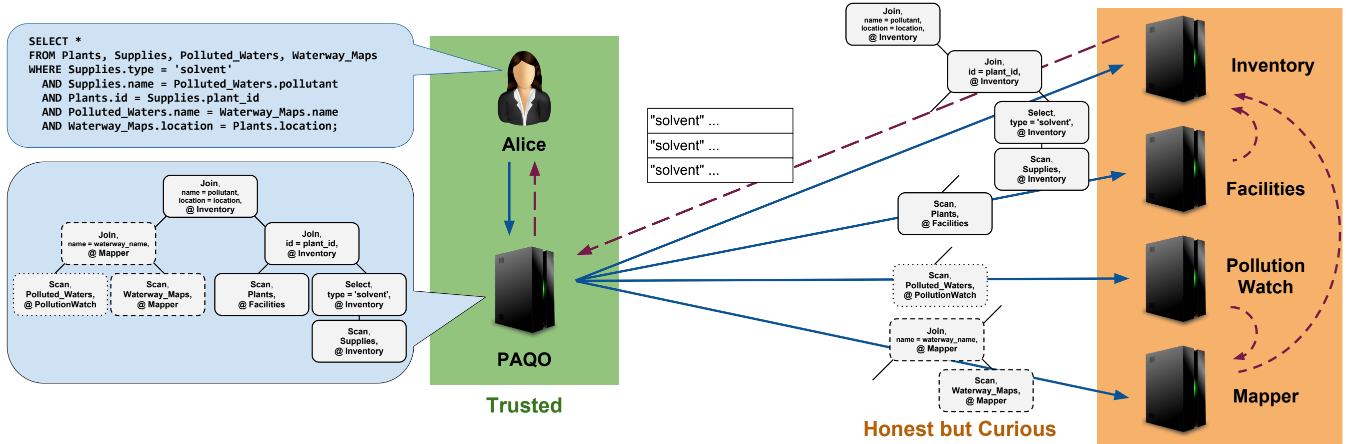


Fig. 1. A visualization of how PAQO would process Alice's example query from Section II-A. Solid lines indicate Alice passing her query to PAQO, and PAQO distributing the partial evaluation plans to each server involved in the query. The final result of the query is passed back to Alice along the dashed line.

Contributions In this work, we present the design of PAQO, and an extensive experimental evaluation. Specifically:

- We have designed and implemented what is, to the best of our knowledge, the first query optimizer to leverage user requirements and preferences as optimization metrics.
- We have developed and implemented a heuristic algorithm for efficiently upholding complex user preferences during query optimization.
- To demonstrate the utility of preference-aware optimization, we have developed a PAQO prototype by extending the PostgreSQL query optimizer. This required extending PostgreSQL to be able to reason about distributed locations during optimization and cost modeling, making for a rich experimental platform.
- We have performed and present a thorough experimental evaluation of PAQO's performance, showing that it incurs only modest overhead to optimization costs.

Roadmap Sec. II presents the system model that PAQO operates in. Sec. III overviews our extensions to SQL. Sec. IV describes the design of PAQO, while Sec. V presents its experimental evaluation. Sec. VI overviews related work and Sec. VII concludes with our future work.

II. System Model and Example

In this section, we first describe an example that will be used throughout this work and our assumed system model.

A. Running Example Scenario

Alice is a low ranking corporate executive who wishes to investigate possible illegal pollution by her employer ManuCo. As a first step in this investigation, she wishes to join data stored by her employer with that of an environmental watchdog group (Pollution Watch) and a waterway mapping service (Mapper) to see if there is any correlation between where her company has plants holding industrial solvents and where those chemicals are appearing as waterway pollutants. To do so, she constructs a query over records describing hazardous solvents owned by ManuCo (stored in the `Supplies` table on ManuCo's `Inventory` database server), details of ManuCo's manufacturing plants (stored in the `Plants` table on ManuCo's `Facilities` server), water pollution data (stored in Pollution Watch's `Polluted_Waters` table), and waterway location data (stored in Mapper's `Waterway_Maps` table) as follows:

```
SELECT *
FROM Plants, Supplies, Polluted_Waters, Waterway_Maps
WHERE Supplies.type = 'solvent'
    AND Supplies.name = Polluted_Waters.pollutant
    AND Plants.id = Supplies.plant_id
    AND Polluted_Waters.name = Waterway_Maps.name
    AND Waterway_Maps.location = Plants.location;
```

In issuing this query, however, Alice would not want either ManuCo or Pollution Watch to become aware of the portion of her query that was issued to the other, or the join condition between the `Supplies` and `Polluted_Waters` tables. Such a revelation could easily cost her her job, either because her employer felt that she "knew too much," or because the watchdog group applied external pressure to the company after learning of the content of her query.

B. System Model

An illustration of the flow of processing Alice's query through PAQO is shown in Fig. 1: Alice issues her query to PAQO, which optimizes and produces an evaluation plan. Each operation in this plan is designated to be evaluated by a specific database server in the system. PAQO splits the plan into the subplans to be evaluated by each server involved in resolving the query, and distributes these plans to their respective servers. The servers then evaluate their subplans, combine the intermediate results as needed, and return the final result to Alice via the machine running PAQO.

System We assume that PAQO produces query plans to be issued over a system of distributed, autonomous, and heterogeneous database servers. These servers can be located anywhere on the Internet, though we assume that all entities in the system (i.e., users, PAQO instances, and database servers) are able to establish private and authenticated communication

channels with one another. Messages sent along these channels should be protected from eavesdropping, modification, reordering, and replay (e.g., through the use of TLS [6]). We further assume that all relations in the system are protected by access controls that are specified and enforced by the database systems that host them (e.g., using the industry standard RBAC [9], [23]). As such, we assume that users only obtain the results that they are authorized to see.

We assume that PAQO knows a priori all of the servers participating in the system via an expanded catalog that includes cached metadata about these remote servers. In addition to listing participating servers, this catalog details the relations they make available, and metadata about these relations (e.g., cardinality, attribute selectivities). This relational metadata mirrors that stored in the catalogs of the remote servers so that PAQO can intelligently process distributed queries. The catalog is kept current with remote servers via periodic polling or pushed updates since on-demand polling could reveal aspects of a user's query to remote servers.

Trust Model We assume that the user has access to a running copy of PAQO on either a personal machine or a trusted server. Throughout this work we will refer to the PAQO instance being used to optimize a given query as the *querier*. We assume the querier to be fully trusted by a user. We consider this assumption reasonable, not only due to the user's ability to run and maintain their own personal PAQO instance, but further because users must reveal their queries to some piece of software in order for them to be evaluated.

Database servers in the system, on the other hand, we consider to be honest-but-curious passive adversaries. That is, database servers will correctly evaluate the query subplans assigned to them, and will return the correct results to the user, but in doing so, they will attempt to learn the query issued by the user. Techniques for verifiable computation can be used to ensure correct query evaluation [2]. Though we assume all servers to be honest-but-curious, users may trust subsets of the servers in the system to learn the makeup of their queries (e.g., patient names can be revealed to a doctor's own hospital severs). Our model allows individual users to decide which servers should be trusted to handle sensitive portions of their queries, and which should be treated as untrusted adversaries.

Input to PAQO As shown in Fig. 1, processing distributed database query via PAQO begins with the user specifying her query to PAQO. In addition to supporting queries issued in SQL, PAQO is built to optimize queries specified in Preference-Aware SQL (PASQL). PASQL extends SQL to support specification of constraints on the optimization process and the query evaluation plans produced by that process. Specifically, it extends the SQL `SELECT` statement with two clauses, the `REQUIRING` and `PREFERRING` clauses. These clauses were first introduced in [7], and are discussed in Section III-B. Constraints expressed via these clauses can either be appended to individual queries as they are issued, or placed into profiles to be applied to all outgoing queries. PAQO is able to parse these extensions and ensure that the constraints they express are upheld during query optimization.

It should be noted that PASQL support is only needed by PAQO instances. Database servers in the system can remain unmodified, as they need only evaluate query plans sent to them by a remote query optimizer. The `REQUIRING` and `PREFERRING` statements are only processed by the optimizer, where they are used as optimization metrics. As such, these constraints are *locally-enforceable*: they do not require server-side support to be used. A user wishing to issue queries with PASQL constraints attached can simply install and set up a PAQO instance on her personal machine.

III. CONSTRAINT LANGUAGE

In this section, we describe the SQL extensions supported by PAQO [7].

A. Constraints

Constraints are user requirements or preferences on the types of evaluation plans that PAQO produces to evaluate their queries. Traditionally, query performance (specifically the time required to return a result to the user) has been the primary metric used by query optimizers. As such, the fastest plan to evaluate a query was the one chosen. Constraints allow the user to ensure that other conditions are accounted for in addition to query run time. Through constraints, users can instruct PAQO to, instead of just emitting the *fastest* plan, emit the fastest plan *that upholds their specified constraints*.

Our approach to constraint specification consists of two steps. First, a user identifies the portions of her query that she wishes to have specially handled during query optimization. This could be, e.g., pointing out which parts of the query contain patient names, which parts need to be evaluated by separate sites, or which tables should be specifically scanned from servers hosting the freshest copies. In Alice's case, she may want to identify all query plan operations involving the `pollutant` attribute from the `Polluted_Waters` table.

Node descriptors are used to identify these portions of the query that should be specially handled. We consider query plans to be trees of nodes representing relational algebra operations. Each of these nodes is a ternary $\langle\ op, params, p\ \rangle$, where op is the operation represented by the node, $params$ represents the parameters to that operation, and p is the principal (e.g., a database server or the querier) assigned to evaluate the operation. We define node descriptors as a mirror to query tree nodes, also ternaries consisting of op, $params$, and p. Node descriptors are used to "match" query tree nodes that the user would like to have evaluated in a specific way. To accomplish this matching, a "*" is used as a general wildcard. Setting op, $params$, or p to be "*" in a node descriptor will cause that portion of the node descriptor to match any value in the corresponding portion of a query plan node. To construct a node descriptor matching any query tree node that operates on the attribute `Polluted_Waters.pollutant`, for example, Alice could instantiate op and p as "*" while defining $params$ as `Polluted_Waters.pollutant` as follows:

$$\langle *, \{(Polluted_Waters.pollutant)\}, *\rangle$$

Once a portion of the query plan has been identified via a node descriptor, the user can move on to the second step of authoring a constraint: creating a condition that must be upheld by query tree nodes matching the node descriptor. Node descriptors are tied to conditions through the use of free variables. The "@" character is used to mark the use of free variables in node descriptors. Like the wildcard marker, free variables will match any value in the corresponding position of a query tree node. Conditions are written over the potential values found in positions corresponding to free variables in matched query tree nodes. For example, to keep all operations on the attribute `Polluted_Waters.pollutant` from be executed by ManuCo's `Inventory` server, Alice would first include a free variable (@site) in the p position as follows:

$$\langle *, \{(Polluted_Waters.pollutant)\}, @site\rangle$$

She could then ensure that any query plan node matching that node descriptor should not have the Inventory server as the value of p by specifying that $@site <> Inventory$. For the full syntax of node descriptors and constraints, see [7].

B. SQL Extensions

Our extensions to the SQL `SELECT` statement combine node descriptors and conditions and pass them to PAQO. While we only demonstrate queries with either a `REQUIRING` or `PREFERRING` clause attached, these clauses can be combined by a user to reflect her needs for evaluating a single query.

Required constraints take the following general form:

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

Alice's need to keep ManuCo's `Inventory` server from evaluating operations on `Polluted_Waters.pollutant` could be added to her query using a `REQUIRING` clause as follows:

```
SELECT *
FROM Plants, Supplies, Polluted_Waters, Waterway_Maps
WHERE Supplies.type = 'solvent'
    AND Polluted_Waters.pollutant = Supplies.name
    AND Plants.id = Supplies.plant_id
    AND Polluted_Waters.name = Waterway_Maps.name
    AND Waterway_Maps.location = Plants.location
REQUIRING @p <> Inventory HOLDS OVER
        < *, {(Polluted_Waters.pollutant)}, @p >;
```

As another example, let us say that Alice would like her query to be evaluated even if none of her constraints can be upheld, and that she has the following preferences for evaluating her query:

P1 Most importantly, the `Inventory` server should not evaluate operations on the attribute `Polluted_Waters.pollutant`

P2 The `Inventory` server should not evaluate operations on the attribute `Polluted_Waters.name`

P3 The `Facilities` server should not evaluate operations on the attribute `Polluted_Waters.name`

In order to allow users to express complex and partially ordered hierarchies of constraints such as this, PASQL includes the `PREFERRING` clause. 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 those listed after that `CASCADE`.

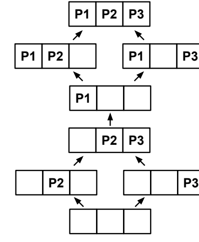


Fig. 2. A preference lattice with the most preferred sets of upheld preferences (Pi) at the top, and the least preferred at the bottom.

Returning to our example, Alice could express her preferential constraints as follows:

```
SELECT *
FROM Plants, Supplies, Polluted_Waters, Waterway_Maps
WHERE Supplies.type = 'solvent'
    AND Polluted_Waters.pollutant = Supplies.name
    AND Plants.id = Supplies.plant_id
    AND Polluted_Waters.name = Waterway_Maps.name
    AND Waterway_Maps.location = Plants.location
PREFERRING @p <> Inventory HOLDS OVER
        < *, {(Polluted_Waters.pollutant)}, @p > -- P1
    CASCADE @p <> Inventory HOLDS OVER
        < *, {(Polluted_Waters.name)}, @p >      -- P2
    AND @p <> Facilities HOLDS OVER
        < *, {(Polluted_Waters.name)}, @p >;     -- P3
```

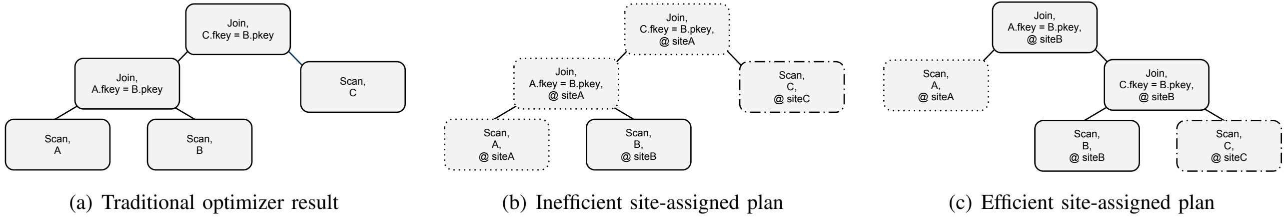
This specification forms a partial order of preferred constraints. Specifically, keeping `Pollution_Watch.pollutant` from Inventory is the most important constraint to be considered, while the other two constraints should be given equal weight. Using this partial order, a ranking of query plans by the query constraints they uphold can be constructed as shown in Fig. 2. Of the most preferred plans found by PAQO during query optimization (i.e., the highest ranking plans), the one with the lowest cost will be the one emitted to evaluate the query. In Sec. IV-B, we will show how such a ranking is used by PAQO to efficiently generate highly-preferred evaluation plans for queries with an attached `PREFERRING` clause.

IV. Design and Implementation

In this section, we first demonstrate how a strawman approach can lead to woefully inefficient query plans, and describe the design and implementation of PAQO in detail.

A. A Strawman Approach

An intuitive approach to support most user constraints on query evaluation plans would be to optimize the query using an off-the-shelf query optimizer, and then post-process the



(a) Traditional optimizer result (b) Inefficient site-assigned plan (c) Efficient site-assigned plan

Fig. 3. Candidate query plans for the example query in Sec. IV-A.

resulting plan to enforce user constraints. [4] uses this general approach to ensure that their optimizer generates plans that uphold tuple-level access controls (as defined by the hosts of the relations involved in the query) during query evaluation. To apply this approach to uphold user-specified constraints, REQUIRING and PREFERRING clauses would first need to be stripped from incoming queries. The resulting pure SQL query would then be optimized and a plan without any execution locations would be produced. A second optimization phase would then be performed on this plan, applying execution locations to each operation according to the user-specified constraints in these REQUIRING and PREFERRING clauses.

However, this disconnect between optimization and constraint consideration opens the door for the creation of unnecessarily inefficient query plans. Consider this example:

```
SELECT * FROM A, B, C
WHERE A.fkey = B.pkey AND C.fkey = B.pkey;
REQUIRING @p <> siteB HOLDS OVER < *, {(A.fkey)}, @p >;
```

Here, a user wishes to join Tables A, B, and C (stored at siteA, siteB, and siteC, respectively) while ensuring that siteB does not evaluate any operations on A.fkey. We assume that A has a cardinality of 100 tuples, B 5,000,000 tuples, and C 200 tuples. We further assume that the joins to be performed link attributes in Tables A and C that are foreign keys to B.

Given the pure SQL version of this query (everything preceding the REQUIRING clause) a traditional query optimizer (the first phase of a two-phase approach) would first join Tables A and B, as A is half the size of C, resulting in the query plan shown in Fig. 3(a). Post-processing this plan to support its attached requirement would disallow the join tables A and B at siteB. Hence, the second phase would logically choose siteA to evaluate this join (as in Fig. 3(b)), though this comes at a great cost to query performance. Under this plan, all of B (5,000,000 tuples) must be shipped over the network to siteA, incurring a great cost to not only total query evaluation time, but also network bandwidth utilization. Further, this prevents all indices for Table B (assumedly) maintained at siteB from being used during the evaluation of this query.

To compare the costs of different optimizer results, we assume that all three servers are capable of downloading tuples at 50 Mbit/second, that all three tables have a tuple width of 128 bytes, that scan operations read tuples at a rate of 120 microseconds/tuple, and that join comparisons take 0.028 microseconds/tuple.[1] Further, we assume that operations executed at different sites occur in parallel, and tuples are pipelined from one operation to the next as they are generated.

Given these assumptions, the plan shown in Fig. 3(b) would take 711.66576 seconds to execute. Without considering constraints, the general plan from Fig. 3(a) could be evaluated in 0.02847 seconds. In addition to not having to ship all 5,000,000 tuples of B across the network, this unconstrained plan can utilize indices on the primary key of B to quickly scan only the tuples needed in the process of joining A and B.

Ideally, this preference-aware query would be evaluated using the plan shown in Fig. 3(c). By joining B and C first at siteB, the large network transfer shown in Fig. 3(b) can be avoided while upholding the user's specified constraint. Further, by joining B and C together at siteB, indices on B can again be utilized to speed up the join and avoid scanning all of table B. This approach takes 0.03038 seconds to evaluate. Such a plan can only be discovered, however, by considering constraints when determining the join order of a query plan.

B. PAQO

To avoid the shortcomings of two-phase optimization discussed above, we take synergistic approach to query optimization, accounting for both user preference and query performance during the query optimization process. We now outline the challenges inherent in adopting this approach, and then describe our implementation of PAQO within PostgreSQL's query optimizer. Finally, we highlight how these challenges were overcome in our implementation.

1) Challenges in Implementing PAQO:

Distributed Processing Support The optimizers of most open-source DBMSs (e.g., PosgreSQL, MySQL) have no support for optimizing queries issued across a system of autonomous, distributed, heterogeneous database servers as we assume in our system model. Hence, we had to adapt the optimizer to be able to reason about execution locations for individual operations in a query plan.

Early Pruning Any subplan that violates a required constraint can not be used as the basis for any further plans as no plan can be emitted that violates a required user constraint. This makes user requirements natural pruning rules: any plan that violates one can be safely discarded as soon as it is realized. We needed to ensure that PAQO enforces this.

Efficient Preference Support Producing optimally-preferred plans can lead to an increase in the state space that need be explored during optimization. Consider the following:

[1]These values were gleaned from the experimental setup used in Sec. V.

Algorithm 1 PAQO's dynamic programming optimization pseudocode.

```
Require: A parsed representation of a user query, Q
Require: A list of requirements, R
Require: A list of preferences, P
 1: subplans ← EMPTY_LIST
 2: subplans[1].add(EMPTY_LIST)
 3: for table in Q do
 4:     subplans[1].add(scan plan for table)
 5: num_tabs = number of tables in Q
 6: for level = 2 to num_tabs do
 7:     subplans[level].add(EMPTY_LIST)
 8:     possibles ← ENUMERATE_JOINS(level, subplans)
 9:     for plan in possibles do
10:         for each possible site do
11:             set site to evaluate the root of plan
12:             if VIOLATES_REQUIREMENT(plan, R) then
13:                 continue to next iteration
14:             CHECK_PREFERENCES(plan, P)
15:             accept_new = True
16:             for other in subplans[level] do
17:                                   ▷ Compare prefs upheld by plan and other
18:                 if plan ⪰ other then
19:                     subplans[level].remove(other)
20:                 else if plan ⪯ other then
21:                     accept_new = False
22:                     break
23:                 else if plan has a better sort order then
24:                     if cost(plan) < cost(other) then
25:                         subplans[level].remove(other)
26:                 else if cost(plan) > cost(other) then
27:                     accept_new = False
28:                     break
29:             if accept_new then
30:                 subplans[level].add(plan)
    return the fastest plan in subplans[num_tabs]
```

Algorithm 2 Pseudocode for the functions handling requirements and preferences.

```
 1: function VIOLATES_REQUIREMENT(plan, R)
 2:     for req in R do
 3:         fvar_vals ← EMPTY_LIST
 4:         for node in plan do
 5:             for descriptor attached to requirement do
 6:                 if node matches descriptor then
 7:                     fvar_vals.add(values from node)
 8:         for value in fvar_vals do
 9:             if condition of req is False for value then
10:                 return True
        return False
11: end function
12: function CHECK_PREFERENCES(plan, P)
13:     prefs_upheld ← EMPTY_LIST
14:     for pref_level in P do
15:         prefs_upheld[pref_level] ← EMPTY_BITMAP
16:         for pref in pref_level do
17:             fvar_vals ← EMPTY_LIST
18:             for node in plan do
19:                 for descriptor attached to preference do
20:                     if node matches descriptor then
21:                         fvar_vals.add(values from node)
22:             for value in fvar_vals do
23:                 if condition of pref is True for value then
24:                     prefs_upheld[pref_level].add(pref)
25:     Attach prefs_upheld to plan
26: end function
```

```
PREFERRING @P = Mapper HOLDS OVER
      < *, {(Polluted_Waters.pollutant)}, @p >
   AND @p <> @q HOLDS OVER
      < *, {(Polluted_Waters.pollutant)}, @p >,
      < *, {(Plants.location)}, @q >;
```

If a join on `Plants.location` needs to be performed before a join on `Polluted_Waters.pollutant`, in order to ensure that both of constraints are upheld, PAQO would have to maintain multiple plans with differing execution sites for the join on `Plants.location`. PAQO would be unable to simply select the most efficient subplan as, if that subplan required the evaluation of the join on `Plants.location` at `Mapper`, maintaining only this subplan would cause the constraints to be unnecessarily violated. Unfortunately, maintaining multiple such subplans leads to an increase in the state space to be searched, and hence, and increase in query optimization time. We needed to *efficiently* produce highly preferred query plans, that is, support preferences and emit highly preferred plans while avoiding this increase in searchable state space.

2) Implementation Overview: We will use Algorithms 1 and 2 to describe our implementation of PAQO. Unless otherwise noted, all line numbers refer to Algorithm 1. To make our discussion concrete, our description below is based on an implementation of PAQO on PostgreSQL. It is not hard to see that it can be easily adapted to other database systems.

Similar to most query optimizers, PAQO takes as input a parsed representation of a query and returns a plan to evaluate that query. PAQO also takes two additional inputs: a list of required constraints, and a list of lists of preferred constraints (where each inner list represents constraints in a different preference level). Note that either or both of these constraint lists may be empty. With these inputs, PAQO begins a bottom-up dynamic programming approach to find an evaluation plan for the query that upholds its specified constraints.

To begin, PAQO initializes nested lists that will be used to store the sub-plans that are realized over the course of query optimization (lines 1-2). These lists are seeded with plans for reading the data needed to evaluate the query under consideration (lines 3-4). PAQO then proceeds to iteratively find plans by joining increasing number of tables (lines 5-32). Note that to conserve space, we abbreviate the process for discovering potential joins at each join level with a single function call at line 8 (ENUMERATE_JOINS()).

For each of these join plans, we check the viability of a large number of potential evaluation sites to ensure that an efficient plan upholding user constraints can be found. Specifically, PAQO considers the querier, all sites hosting data required by the query, and further third-party sites offering to perform computation for queries as potential evaluation sites. To perform this site assignment, we augmented PostgreSQL's query plan data structures to hold additional state representing execution location. The entire site assignment process is represented in lines 10 and 11. Note that only the root of a query plan is assigned a site at any given time. This is due to PAQO's bottom-up construction of query plans: each newly created node is added as the root of two previously-realized query plans that already have assigned evaluation sites.

Once an evaluation site is assigned to the root node of a new plan, PAQO checks to ensure that this plan does not violate any required constraints (lines 12 and 13). To perform this check, PAQO iterates through all requirements attached to the query (lines 2-11 of Algorithm 2) finding all nodes

of the current query plan that match each requirement (lines line 4-7 of Algorithm 2). If a match is found (lines 6-7 of Algorithm 2), the appropriate free variable values are stored (e.g., if the descriptor that was just matched has a single free variable in place of p, the execution location of the matched query plan node is stored). The current requirement's condition is then checked using all of the matched free variable values stored in traversing the query plan (lines 8-10 of Algorithm 2). If a violation is found, the function returns, and the plan is pruned from further consideration (lines 12-13).

Assuming that the current query plan does not violate any required constraints, PAQO checks which preferences it upholds (line 14). Checking for preferred constraint adherence is done in a manner very similar to checking for required constraint adherence. The first key difference is that preferred constraints are divided into several levels of constraints that must be iterated through (lines 14 and 16 of Algorithm 2). When the `PREFERRING` clause is parsed, every time a `CASCADE` keyword is encountered, a new level of preferences is instantiated and populated with all constraints that come before the next `CASCADE` (or the end of the query, in the case of the final level). Hence, the levels are ordered from most to least-preferred. Checking preference adherence also differs from checking requirements in that the goal is not to find a single violation, but to record the constraints upheld in a list of bitmaps (lines 13-24 of Algorithm 2). These bitmaps are used to efficiently compare the relative preference of two plans.

Once PAQO has checked which preferences a given plan upholds, it is compared with previously stored plans joining the same set of relations to see if either or both can be pruned (lines 16-32). Each plan is assumed to be kept until PAQO determines that it is dominated by another plan. As shown in lines 17-22, PAQO uses a heuristic that more preferred plans dominate lesser preferred. Comparisons of plan preference are shown on lines 18 and 20 via the $\succeq$ and $\preceq$ operators. For plans that are equally preferred, PAQO falls back to PostgreSQL's standard domination checks to see if either of the two should be discarded. Specifically, one of the plans will be discarded if it does not produce a different tuple sort order that could be used to the benefit of future operations and it is estimated to cost more than the other (lines 24-30). If no plan is found to dominate the current plan, it is stored for later use.

Our assumption of distributed query evaluation necessitated a wide-ranging reworking of query plan cost estimation functions. First and foremost, the cost of shipping data between sites needed to be accounted for. To do this, we extended PostgreSQL's calculation of costs to read data from disk, scaling the calculation up to network transfer speeds. To calculate disk read costs, PostgreSQL simply multiplies the number of tuples to be read by a disk read cost constant. We similarly defined a network transfer cost constant, scaled it appropriately relative to the disk read cost constant, and calculate network transfer costs by multiplying the number of tuples to be transferred by this constant.

Using multiple execution sites further allows for parallel execution of query plans. While PostgreSQL is a threaded database server, each query is evaluated within a single thread, and hence, the cost estimation functions assume the total cost of evaluating a query to be the sum of the costs of evaluating all of its operations. We modified these functions to account for parallel execution of operations annotated for different sites. Further, we allow for operations to process streams of tuples from their children where appropriate (e.g., the smaller relation involved in a hash join must be realized and hashed before the join can begin, though the larger relation can be streamed, probing the join attribute of each tuple as it is received).

Once all potential n-way joins have been processed, only the most preferred plans will remain due to our heuristic. PAQO then returns the n-way join plan with the lowest cost.

3) Challenges Revisited:

Distributed Processing Support By including evaluation location state in optimization data structures and rewriting PostgreSQL's cost estimation functions to account for parallelizable and distributed query evaluation, we enable PAQO to optimize queries over a distributed set of database servers.

Early Pruning As soon as an execution location is assigned to a new operation (and hence a new subplan), the plan rooted by that operation is checked for adherence to all user requirements. Any plans that fail to pass this check are immediately pruned from the search space.

Efficient Preference Support By maintaining only the most preferred plans at each join level, we limit the number of plans maintained over the course of optimization. This heuristic enables us to avoid the state space blowup discussed in Sec. IV-B1, and efficiently support user preferences.

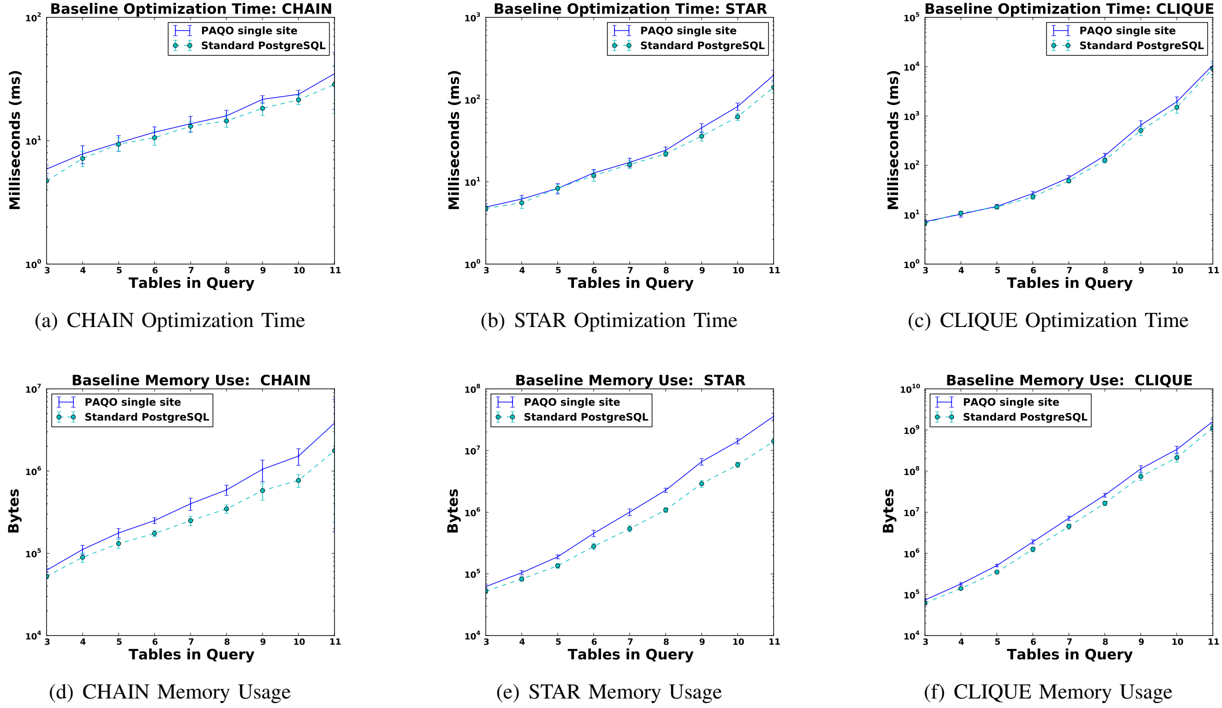
V. Experimental Evaluation

In this section, we begin by describing our experimental setup, and then present the results of several experiments evaluating the performance of PAQO.

A. General Experimental Setup

All of the experiments presented in this section were conducted on a single machine running Arch Linux with an Intel i5-2500 processor, 16GB of RAM, a 2TB hard disk dedicated to database tables and configuration files, and a 500GB hard disk to hold everything else on the machine (e.g., OS files and database binaries). For experiments comparing PAQO's performance to PostgreSQL's standard optimizer, the optimizer from PostgreSQL version 9.1.1 (which serves as the basis for PAQO) was used. Optimization times, memory utilization statistics, and query plan makeups are gathered from logs produced by PAQO and PostgreSQL during optimization. Due to the difficulties of gathering accurate measurements of PostgreSQL's memory usage (see [1], [15]), the memory utilization statistics presented here were gathered via modifications to PostgreSQL's internal memory allocation functions. Hence, they precisely show the amount of memory allocated by the optimizer to evaluate the presented queries.

As most of our experiments optimize for data stored at multiple sites, a distributed database system was simulated within the PostgreSQL DBMS. Each relation used by PAQO



(a) CHAIN Optimization Time (b) STAR Optimization Time (c) CLIQUE Optimization Time

(d) CHAIN Memory Usage (e) STAR Memory Usage (f) CLIQUE Memory Usage

Fig. 4. Results of comparing the optimization times and memory usages of PAQO and the optimizer from PostgreSQL

is assumed to be stored at a single site in the system, and this site is annotated as the execution location of all scans of the relation. We treat PostgreSQL's system catalog as the catalog of metadata from remote databases maintained by PAQO. Because we aim only to evaluate the performance of our optimizer, this simulation does not affect the results we present here. Optimization occurs at a single site with access to a single metadata catalog before distributed query evaluation.

B. Experiments

We first demonstrate the negligible effects of our code on the optimization of queries without constraints by comparing PAQO with PostgreSQL' standard optimizer. From there, we demonstrate the effect that requirements and preferences have on query optimization performance with both randomly generated queries, and a case study using the example scenario from Sec. II-A. Finally, we examine the plans generated by the case study queries to show that PAQO is not only efficient, but also makes correct choices during optimization.

1) Comparison to PostgreSQL: These experiments were performed on randomly generated queries over randomly generated relations following the experimental model used in [13], [18], [20]. The relations over which these queries are defined are sized according to the distribution shown in Table I, and composed of attributes whose size follows the distribution shown in Table II. The experiments were run on queries over an increasing number of relations (from three to eleven). Queries were generated according to three join topologies: CHAIN, STAR, and CLIQUE. Each query was run once to warm caches, and again to be averaged into the data

TABLE I
DISTRIBUTIONS USED TO GENERATE RELATION CARDINALITIES.

Class	Relation Size (Cardinality)	Distribution
S	10 - 1,000	15%
M	1,000 - 10,000	30%
L	10,000 - 1,000,000	35%
XL	1,000,000 - 100,000,000	20%

TABLE II
DISTRIBUTIONS USED TO GENERATE RELATION SCHEMAS.

Class	Attribute Domain Size (Bytes)	Distribution
S	2 - 10	5%
M	10 - 100	50%
L	100 - 500	30%
XL	500 - 1,000	15%

point value. Each of these data points represents the average of 20 runs of different randomly generated queries.

For a fair comparison, PAQO assumes that all relations are stored at a single site. As such, all points where multiple execution locations must be considered are effectively avoided. Further, in order to be processed by PostgreSQL's standard optimizer, none of these queries had any constraints attached. Figs. 4(a), 4(b), and 4(c) show the optimization time (in milliseconds) required by queries of CHAIN, STAR, or CLIQUE topology (respectively) while Figs. 4(d), 4(e), and 4(f) show the memory (in bytes) required to optimize these queries.

These graphs clearly demonstrate the negligible overhead that PAQO incurs on the optimization process. With this data, we establish that any increases in optimization time and memory usage shown in later experiments are a result of site

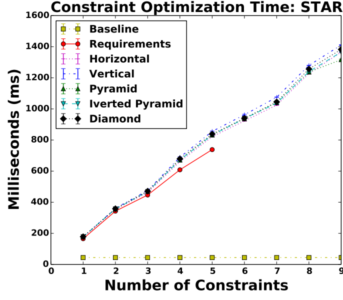


Fig. 5. Optimization times of queries with varying shapes of constraints.

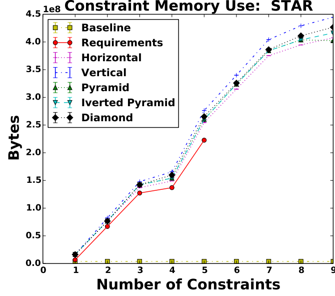


Fig. 6. Memory utilization by queries with varying shapes of constraints.

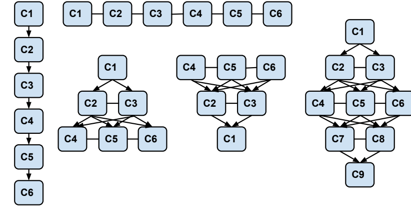


Fig. 7. A visualization of the different preference shapes used in Sec. V-B2's experiments (clockwise from the left): Vertical, Horizontal, Diamond, Inverted Pyramid and Pyramid. Each node represents a constraint. Equally preferred nodes (joined by an `AND`) are shown on the same level, while nodes higher in a given structure are considered more preferred than those lower (i.e., each change in level represents the use of a `CASCADE`.

TABLE III
REQUIREMENTS APPLIED TO ALICE'S QUERY FROM SEC. II-A AS PART OF OUR CASE STUDY EVALUATION

1	@p <>Inventory HOLDS OVER ⟨ *, {(Polluted_Waters.pollutant)}, @p ⟩
2	@p <>Pollution_Watch HOLDS OVER ⟨ *, {(Waterway_Maps.name)}, @p ⟩
3	@p <>@q HOLDS OVER ⟨ *, {(Polluted_Waters.name)}, @p ⟩, ⟨ *, {(Plants.location)}, @q ⟩
4	@p <>Facilities HOLDS OVER ⟨ *, {(Polluted_Waters.name)}, @p ⟩
5	@p <>Inventory HOLDS OVER ⟨ *, {(Polluted_Waters.name)}, @p ⟩
6	@p = Querier HOLDS OVER ⟨ *, {(Polluted_Waters.name)}, @p ⟩
7	@p = Querier HOLDS OVER ⟨ join, *, @p ⟩;

assignment and constraint processing. Our implementation of PAQO does not inherently necessitate an increase in optimization cost, it only requires more time and space to perform additional processing that PostgreSQL cannot provide.

2) Processing Constraints: To demonstrate the effects of constraint processing on optimization time and memory utilization, we randomly generated a query over six relations and nine constraints on the optimization of that query. This query is issued over the same randomly generated relations used in the previous experiments. Here, though, the relations are considered to be stored at different sites, allowing for meaningful use of constraints. We optimized this query by itself and with the generated constraints attached to produce the graphs displayed in Figs. 5 and 6. Each data point is the average of five runs of this query with a given number of constraints attached (six runs were performed in total for each data point, one to warm caches and the five to gather data). As the similarities across topologies shown in Fig. 4 are also displayed in this experiment, only the STAR topology results are presented here in the interest of space.

In these graphs, the Baseline curve shows the performance of PAQO when given the query without any constraints. The Required curve shows the results of adding each of the nine constraints as a requirement. The rest of the graphs show the results of adding these constraints as preferences according to different shapes. Different shapes were created by varying the use of the `AND` and `CASCADE` keywords to for each additional constraint. Fig. 7 illustrates the shapes used in this experiment.

It should be noted that the graph of required constraints shows no data for any more than five constraints because the sixth constraint generated conflicts with a previous constraint, and hence it is impossible for PAQO to generate query plan that upholds all user-specified requirements. Note that this conflict is only an attribute of the specific constraints generated, this is not an inherent limit in the number of requirements that can be applied to a given query. As these constraints are randomly generated, however, there is an increasing probability of conflict between required constraints as the number of random constraints applied to a query increases. The inclusion of conflicting constraints in this experiment further showcases PAQO's ability to handle conflicting preferred constraints.

While this experiment does confirm the expectation that optimization cost increases with the number of constraints attached to a query, more importantly, it shows that optimization time and space requirements increase linearly with the number of constraints. Hence, while an overhead is incurred to account for user constraints during query optimization, this overhead scales well as the number of constraints specified increases.

3) Case Study Performance: We have further run a series of experiments evaluating the performance of Alice's query from Sec. II-A with the requirements listed in Table III. The results of these experiments are shown in Figs. 8 and 9. The constraints from Table III are applied one at a time as requirements, and each data point is the average of five runs (preceded by a cache-warming run). Fig. 8 further presents the optimization time needed to process each requirement alone in addition to combining it with previous requirements (i.e., the results for optimizing both Alice's query with just the second requirement attach and with the first and second requirements

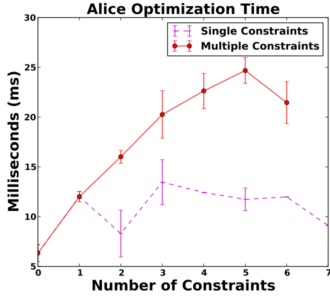


Fig. 8. Optimization times of Alice's query with a varying number of requirements.

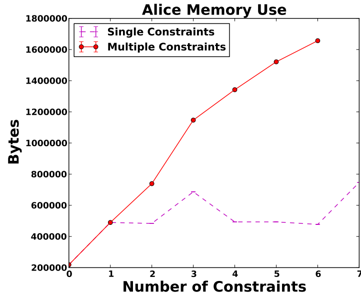


Fig. 9. Memory utilization of Alice's query with a varying number of requirements.

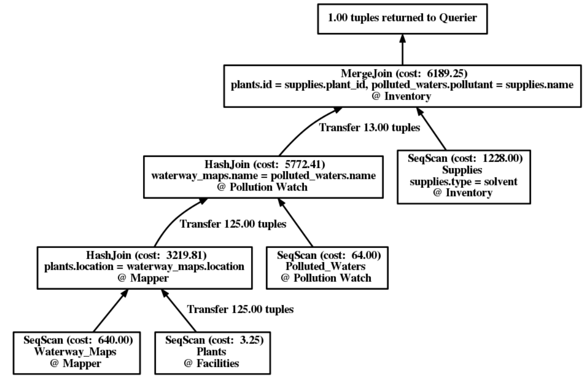


Fig. 10. A plan to evaluate Alice's base query.

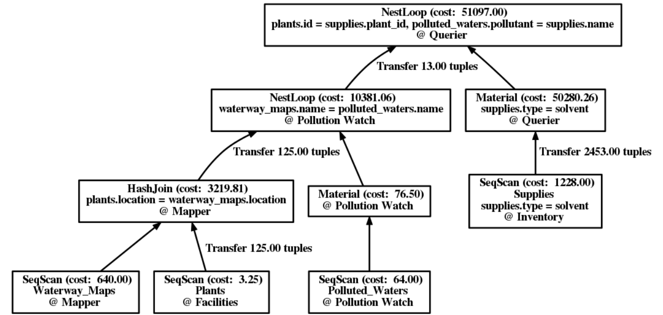


Fig. 11. A plan to evaluate Alice's query with requirement 1.

attached are both presented, as are the results for the third and the first, second, and third, etc.). The sixth requirement, limiting all joins on `Polluted_Waters.name` to be evaluated by the `Querier` is highly restrictive. With this constraint, PAQO is able to prune large portions of the optimization search space (i.e., all plans with another site annotated to evaluate an operation on `Polluted_Waters.name`), resulting in a clear decrease in optimization time.

4) Query Plan Cost and Correctness: As the queries and requirements used in previous experiments had no semantic meaning, comparing the tradeoff between constraint adherence and the estimated cost of executing a query plan similarly would have no real meaning. Hence, we also utilize this case study to demonstrate the cost and correctness of plans generated by PAQO. PAQO was similarly demonstrated in [8].

The query plans presented in this section are a direct representation of the plans produced by PAQO. As such, the nodes in these query plans display the method for evaluating the relational algebra operation needed as opposed to just the general relational algebra operation (e.g., sequential scan as opposed to scan, hash join as opposed to join). The first line of each node lists the method and the estimated cost of evaluating the entire subtree rooted at that node (hence the cost of the entire plan is displayed in the root node). It should be noted that these costs are given in PostgreSQL's computational units. The real world time required to evaluated these plans will depend on the computation power of the servers they are evaluated by, the speeds of network connections between these servers, etc. The middle lines of each node represent the parameters to the operation while the final line indicates the execution location assigned to that operation. Note that PAQO will choose to evaluate some operations together by combining select and project operations with other operations. This is a performance improving technique that saves making an extra pass over the intermediate relation. As a final note, network transfers are indicated via annotations to the right of edges linking two operations to be performed at different sites.

First, we show the result of optimizing Alice's base query (from Sec. II-A), in Fig. 10. The first requirement from Table III requires that operations on `Polluted_Water.pollutant` are hidden from the `Inventory` server. To accommodate this, PAQO chooses to evaluate the root of the query plan at the `Querier` as shown in Fig. 11. This plan has a higher estimated cost, though that is exactly why it was not selected as the plan to evaluate Alice's query without constraints. With this requirement, Alice is adding an optimization metric that is considered more important than the otherwise assumed "lowest cost" metric. As such, this tradeoff between constraint adherence and plan cost is expected. As future work, we will develop constraints that allow users to bound the overheads incurred by upholding their constraints (see Sec. V-C).

Applying both requirements 1 and 2 similarly shifts the evaluation of the join on the condition `waterway_maps.name = polluted_waters.name` from `Pollution_Watch` to `Mapper` as shown in Fig. 12. It is interesting to note that, though this requires another large network transfer of tuples from `Pollution_Watch` to `Mapper`, this transfer can be performed

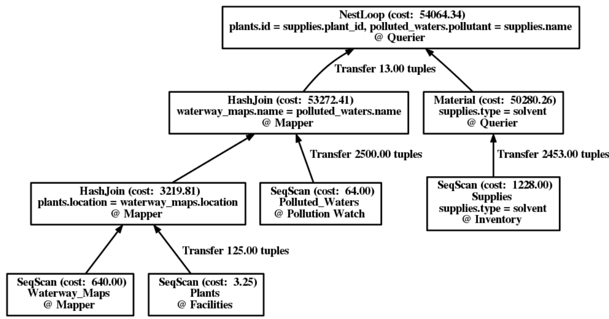


Fig. 12. A plan to evaluate Alice's query with requirements 1 and 2.

in parallel with the transfer from Inventory to the `Querier`, and hence has only a small impact on the overall plan cost.

Requirement 3 explicitly disallows Mapper from performing both of the operations that it does in Fig. 12, and hence the evaluation of the join on the condition `waterway_maps.name = polluted_waters.name` is shifted again to the `Facilities` server. Requirements 4 and 5 similarly adjust the site selected to evaluate this join to the `Inventory` server and a third party computation server, before requirement 6 finally forces it to be evaluated by the `Querier` as well.

The inclusion of requirement 7 creates a conflict among the requirements. If all joins are to be performed by the `Querier`, requirement 3 cannot be upheld as the attributes it covers are both used as join conditions. Given this query, PAQO returns an error informing the user that it is unable to produce an evaluation plan for the supplied query.

C. Discussion and Future Work

Requirements as Pruning Rules While PAQO incurs a negligible overhead on plans with no constraints attached, optimization time of constrained queries increases with the number of constraints applied to the queries. As requirements are used as additional pruning rules, queries with larger numbers of requirements should be able to use them to decrease the number of potential plans created during query optimization, and hence, decrease optimization time. Our selection of randomly generated constraints (from the experiments presented in Sec. V-B2), by their random nature, rarely affected the query evaluation plans generated by PAQO (e.g., they prohibit joins at sites that were not selected to perform joins anyway as there was another site that could perform joins more efficiently). As such, the optimization time results from Sec. V-B2 serve as a worst-case experimental result. Lacking sufficient actual pruning, these queries are subject to the additional overheads imposed by constraint checking while achieving only minimal benefits. While we were able to clearly show how the use of requirements as pruning rules can reduce optimization time and overcome constraint checking overhead in Sec. V-B3, not all users will be able to glean those benefits (e.g., users who issue queries with only preferences attached). To address this, we will be investigating smarter approaches to constraint checking and constraint pruning as future work.

Effect of Preference Shape In contrast to requirements, preferences have two factors that could contribute to their complexity: the number of preferences attached and their shape. Fig. 5, however, clearly shows that shape has a negligible effect on optimization time. This is an intuitive conclusion, as in checking preference adherence, all preferences are iterated over without regard to the ranking relative to one another as shown in Algorithm 2. The shape of the ranking of preferences attached to a query only affects the comparison of the relative preference of two plans in that each `CASCADE` keyword used in a `PREFERRING` clause necessitates the use of another bitmap to store the preferences upheld by a given plan. This effect is negligible, however, due to the efficiency of bitmap comparisons and the fact that comparisons can stop as soon as a difference in the preferences upheld at a given level is found (i.e., an increased number of `CASCADE`s does not have as much of an impact on the average case plan comparison times as it does on the worst case).

Efficient Preference Support Fig. 5 further shows PAQO is able to optimize queries with attached preferences with performance competitive to queries with the same number of requirements. Recall that, though the cost of constraint checking takes its toll on optimization with requirements, they are used as pruning rules when exploring the optimization search space. Because of this, additional requirements can only decrease the size of the search space that must be explored during query optimization. By showing that PAQO's optimization of queries with preferences takes little more time than for those with requirements, we can validate our intuition that our use of a heuristic for emitting highly-preferred query plans limits the number of query plans maintained duration optimization, and hence, maintains lower optimization times. This does lead us to question what tradeoffs in optimization time would be required to support better heuristics. We will be investigating such tradeoffs as future work on PAQO.

VI. Related Work

Preference SQL Our support for preferred constraints closely mirrors that of [11] in both the use of partially-ordered preference structures, and the syntax of the SQL extensions used to express them. Here, these techniques are used to drastically different effect: we use user preferences as optimization metrics, while in [11], preferences are used to order query result tuples from most to least-interesting.

Distributed Query Processing The optimization and processing of queries over distributed database systems has been an area of active research for several decades (for a survey, see [12]). There are two main techniques for evaluating queries over distributed database systems: data shipping (transferring the data being queried to the querier for processing) and query shipping (having servers hosting data process it as needed to evaluate the query and return the result to the querier). The combination of these techniques is known as hybrid shipping [10]. While previous work on distributed database query optimization has focused primarily on decreasing optimization time and improving the plans generated [5], [18], [21], here,

we present the first distributed query optimizer to include user-specified constraints as additional optimization metrics.

While query hints [17] also allow users to guide the query optimization process, they are orthogonal and complimentary to query constraints. Hints specify the use of certain physical operators for relational operations (e.g., the use of hash join for evaluating a certain condition). We allow users to constrain the evaluation site of any portion of their SQL query.

Authorization Enforcement in Database Systems In [4], the authors propose a theoretical framework for optimizing distributed database queries such that when evaluating the resulting plan, tuples are only processed by sites that are explicitly allowed to see them according to authorizations specified as profiles for each relation in the system. To enact these controls, the authors utilize the strawman two-phase optimization approach presented in IV-A. They first optimize the query without regard to data distribution, and then assign sites to the operations in the resulting query plan according to the authorization profiles. Here, we present techniques for including not only site assignment, but further user-specified constraints directly in the query optimization process.

The authors of [3] consider a very different adversary: passive malware with access to the memory of a database server for a limited period of time. The authors modify MySQL to optimize queries over centralized, encrypted databases so as to limit the number of tuples that are stored decrypted in memory as well as the number of decryption keys stored in memory. While their approach of modifying MySQL's query optimizer to include new optimization metrics is, indeed, similar to that presented here, they focus strictly on protecting database contents from a compromised system. We empower users to express new optimization metrics for the queries they issue.

Private Information Retrieval (PIR) techniques allow a user to retrieve some information form a remote database without revealing to the database server specifically what information was retrieved. Though the practical feasibility of PIR has been called into question [19], it has received quite a bit of research attention over the past several years [14], [16]. In [7], we prove that the use cases for PIR are a subset of those that can be expressed by the user via PASQL. Hence, PIR techniques could be used to evaluate PASQL queries in the case that both the client and server support PIR, and using PIR would be the most efficient option. PAQO allows users to protect their privacy in a locally-enforceable manner, however, and further supports protecting any aspect of an SQL query.

VII. Conclusion

In this paper, we present PAQO, a distributed query optimizer with support for declarative, user-specified constraints on the optimization process. Such constraints enable users to express a variety of requirements and preferences ranging from privacy aspects to data quality. To the best of our knowledge, PAQO is the first query optimizer to include user-specified requirements and preferences as optimization metrics. We have thoroughly experimentally evaluated PAQO to establish the overheads incurred on the optimization process by upholding such constraints during query optimization.

In addition to the previously outlined future work, we are working to expose internal optimizer state for the user to constrain. This type of constraint would empower users to inform PAQO that they would like some set of constraints upheld, provided that doing so does not significantly impact the estimated runtime of their query. We are also developing an interactive query optimization interface that will make PASQL constraints more user friendly by allowing users to graphically select unacceptable execution site choices as they are realized by the optimizer.

Acknowledgments

This work was supported in part by NSF awards CCF–0916015, CNS–0964295, CNS–0914946, CNS–0747247, OIA–1028162, and CNS–1253204, and EMC/Greenplum. We thank the reviewers for their feedback.

References

[1] Performance Analysis Tools. http://wiki.postgresql.org/wiki/Performance_Analysis_Tools, May 2013.
[2] S. Benabbas, R. Gennaro, Y. Vahlis. Verifiable delegation of computation over large datasets. In *CRYPTO*, 2011.
[3] M. Canim et al. Building disclosure risk aware query optimizers for relational databases. *VLDB*, 2010.
[4] S. De Capitani di Vimercati et al. Authorization enforcement in distributed query evaluation. *JCS*, 2011.
[5] A. Deshpande, J. Hellerstein. Decoupled query optimization for federated database systems. In *ICDE*, 2002.
[6] T. Dierks, E. Rescorla. RFC 5246: The Transport Layer Security (TLS) protocol ver. 1.2. http://tools.ietf.org/html/rfc5246, August 2008.
[7] N. L. Farnan, A. J. Lee, P. K. Chrysanthis, T. Yu. Don't reveal my intension: Protecting user privacy using declarative preferences during distributed query processing. In *ESORICS*, 2011.
[8] N. L. Farnan, A. J. Lee, P. K. Chrysanthis, T. Yu. PAQO: A preference-aware query optimizer for PostgreSQL. In *VLDB*, 2013.
[9] D. Ferraiolo, R. Kuhn. Role-based access control. In *NIST-NCSC*, 1992.
[10] M. J. Franklin, B. T. Jónsson, D. Kossmann. Performance tradeoffs for client-server query processing. *SIGMOD Rec.*, 1996.
[11] W. Kießling, G. Köstler. Preference SQL: design, implementation, experiences. In *VLDB*, 2002.
[12] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 2000.
[13] D. Kossmann, K. Stocker. Iterative dynamic programming: a new class of query optimization algorithms. *ACM TODS*, 2000.
[14] C. A. Melchor et al. High-speed private information retrieval computation on GPU. In *SECURWARE*, 2008.
[15] B. Momjian. Measuring free memory and kernel cache size on linux. http://momjian.us/main/blogs/pgblog/2012.html, May 2013.
[16] F. G. Olumofin, I. Goldberg. Privacy-preserving queries over relational databases. In *PETS*, 2010.
[17] Oracle. Using optimizer hints. http://docs.oracle.com/cd/B19306_01/server.102/b14211/hintsref.htm, Oct 2013.
[18] F. Pentaris, Y. Ioannidis. Query optimization in distributed networks of autonomous database systems. *ACM TODS*, 2006.
[19] R. Sion, B. Carbunar. On the practicality of private information retrieval. In *NDSS*, 2007.
[20] M. Steinbrunn, G. Moerkotte, A. Kemper. Heuristic and randomized optimization for the join ordering problem. *VLDB J.*, 1997.
[21] M. Stonebraker et al. Mariposa: A wide-area distributed database system. *VLDB J.*, 1996.
[22] The PostgreSQL Global Development Group. PostgreSQL. http://www.postgresql.org/, Dec. 2012.
[23] S. Tran, M. Mohan. Security information management challenges and solutions. http://www.ibm.com/developerworks/data/library/techarticle/dm-0607tran/index.html, July 2006.