EFFICIENT, LOCALLY-ENFORCEABLE QUERIER PRIVACY FOR DISTRIBUTED DATABASE SYSTEMS

by

Nicholas L. Farnan

B.Sc. in Computer Science, University of Pittsburgh, 2007

Submitted to the Graduate Faculty of

the Kenneth P. Dietrich School of Arts and Sciences in partial

fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2014

UNIVERSITY OF PITTSBURGH

KENNETH P. DIETRICH SCHOOL OF ARTS AND SCIENCES

This dissertation was presented

by

Nicholas L. Farnan

It was defended on

November 21, 2014

and approved by

Adam J. Lee, Associate Professor, University of Pittsburgh

Panos K. Chrysanthis, Professor, University of Pittsburgh

Daniel Mossé, Professor, University of Pittsburgh

Ting Yu, Associate Professor, North Carolina State University

Dissertation Director: Adam J. Lee, Associate Professor, University of Pittsburgh

EFFICIENT, LOCALLY-ENFORCEABLE QUERIER PRIVACY FOR DISTRIBUTED DATABASE SYSTEMS

Nicholas L. Farnan, PhD

University of Pittsburgh, 2014

Traditionally, the declarative nature of SQL is viewed as a major strength. It allows database users to simply describe *what* they want to retrieve without worrying about *how* the answer to their question is actually computed. However, in a decentralized setting, two different approaches to evaluating the same query may reveal vastly different information about the query being asked (and, hence, about the user) to participating servers. In the case that a user's query contains sensitive or private information, this is clearly problematic.

In this dissertation, we address the problem of protecting *query issuer* privacy. We hypothesize that by extending SQL to allow for declarative specification of constraints on the attributes of query evaluation plans and accounting for such constraints during query optimization, users can produce efficient query evaluation plans that protect the private intensional regions of their queries without explicit server-side support. Towards supporting this hypothesis, we formalize a notion of intensional query privacy that we call (I, A)-privacy, and present PASQL, a set of extensions to SQL that allows users to specify (I, A)-privacy constraints to a query optimizer. We explore tradeoffs between the expressiveness of several PASQL variants, optimization time requirements, and the optimality of plans produced. We present two algorithms for optimizing queries with attached (I, A)-privacy constraints and formally establish their time and space complexities. We prove that one is capable of producing optimal results, though at the cost of greatly increased time and space requirements. We use the other as the basis of PAQO, our implementation of an (I, A)-privacy-aware query optimizer. We present an extensive experimental evaluation of PAQO to show

that is is capable of efficiently generating plans to evaluate PASQL queries, and to confirm the results of our formal complexity analysis.

Keywords Privacy-aware Query Optimization, Private Query Processing.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

1.0 INTRODUCTION

1.1 MOTIVATION

Increasingly, people are moving from operating solely on data stored at a single site to processing data combined from many different sources. This shift is occurring for a multitude of reasons. The creation of "mashups" of data and services offered by web sites has become a popular value-added service that creates a richer experience for users. Statisticians are embracing meta-analysis of large number of polls and samples from varied sources to increase the accuracy of predictive models. Different corporations have adopted information sharing programs for mutual benefit on specific projects and initiatives. The research questions surrounding how to perform distributed query processing have been extensively studied in the literature [44]. While research efforts continue to find new and more efficient ways to perform the types of processing exemplified here, the feasibility of such operations is well established. The drastically different privacy concerns presented by this distributed data processing paradigm have received relatively little attention by the research community, however.

User queries to relational database management systems consist of a list of base data relations from which data will need to be fetched and a series of specifications about what subset of the data from each relation should be returned (e.g., return all of the information on red cars from a relation storing lists of cars for sale), how data from different relations should be combined (e.g., cross-reference car sale listings with information about the dealership selling them), and what further operations should be performed on this data to produce the result desired by the user (e.g., count the number of red cars for sale in a given zip code). Collectively, we will refer to this makeup of a query as the *intension* of a query. When using centralized database systems, users issue their queries to a single site that will perform all required optimization and processing to generate the

results of that query. As such, when a user issues her query, she knows that the whole intension of her query will be revealed to this single server. More importantly, she knows that it is the only server that will learn the intension of her query. In a distributed environment, however, though the user may issue her query to one server at a single site, many servers that are owned and managed by different entities could participate in the evaluation of her query. If a user's query requires data stored and maintained at multiple different sites, these sites may need to work together to jointly compute its results. For each operation these sites perform in evaluating the query, they learn more of its intension.

Query intension could be, in part, comprised of information that users consider to be sensitive or private in nature. Revealing such private information through the dissemination of query intension during distributed query processing can very easily lead to violations of user privacy. This situation is made even more grave in that users are typically left unaware of how their queries will be evaluated when they issue their queries. Popular relational database systems process queries specified in *declarative* languages (e.g., SQL [40]). The declarative model allows users to simply state what information they would like, and the system processing their query will determine the best way to retrieve it. While this model is a boon to users in that it frees them from having to manually determine the fastest way to retrieve the information they desire, in the distributed setting, it carries the consequence that users are left unaware of what servers, at which sites, are processing the operations specified in their queries. Without the knowledge of how the intension of their queries is implicitly revealed to remote servers during query processing, users cannot know if their privacy will be violated in issuing a query, let alone the extent of the violation that will occur.

To concretely establish how user privacy can be violated over the course of distributed query processing, the next section presents an example scenario that will be referenced throughout this dissertation.

1.2 EXAMPLE SCENARIO

Consider Alice to be a low-ranking corporate executive at a large manufacturing company called `ManuCo`. Recently, Alice has become concerned that her company may not be following the

proper procedures for disposing of the industrial solvents used in their many manufacturing plants located around the world. Specifically, she is worried that her employer may be dumping large quantities of toxic pollutants into rivers and lakes in the vicinity of ManuCo's plants. As a first step in her investigation to determine if her concerns are valid, 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). Alice expresses her query in SQL 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;
```

An illustration of how this query could be processed is shown in Figure 1. Solid lines indicate Alice issuing her query to the query processing system, and the distribution of partial plans to evaluate her query to each server involved. The final result of the query is passed back to Alice along the dashed line.

Clearly, Alice would consider portions of the intension of this query to be sensitive. If her employer were to become aware that she was combining their inventory data with data describing where the chemicals ManuCo uses are showing at waterway pollutants, her job could be put at risk because she "knew too much" or was "asking the wrong questions." Similarly, if Pollution Watch learned that Alice was cross-referencing her employer's chemical stocks with known in-

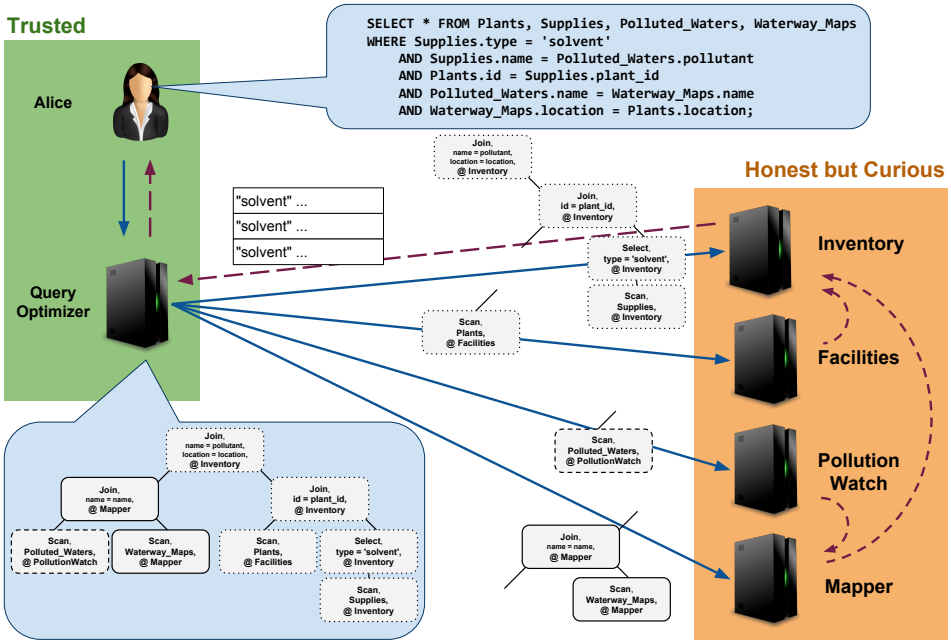


Figure 1: An illustration of processing Alice's example query from Section 1.2.

stances of chemical pollution, her job could again be put at at risk by external pressure from `Pollution Watch`.

If either `Pollution Watch` or `ManuCo` were to perform the combination of both parties' data sets as specified by Alice's query, Alice would be risking her job simply by issuing this query. Unfortunately, Alice has no way to express her desire that this sensitive operation should not be carried out by either `ManuCo` or `Pollution Watch`. A traditional query processing approach would simply devise a plan to return results to Alice in the shortest amount of time and follow this plan to produce the results to her query. However, doing so could well have `ManuCo` or `Pollution Watch` evaluating sensitive portions of Alice's query, violating her privacy and putting her job at risk. The traditional notion of optimizing user queries purely to find the fastest plan would not work out well for Alice. In this scenario, Alice needs her privacy notions to be taken into account when a plan for evaluating her query is devised. The best plan from Alice's point of view is one that upholds the privacy of the intension of her query.

1.3 PROBLEM STATEMENT AND SHORTCOMINGS OF EXISTING APPROACHES

Trivially, Alice could produce the desired result of her query and protect her privacy by using a technique described in the literature as *pure data shipping* [32, 44]. Using this approach, Alice would simply request all of the data stored by each server hosting data that Alice needs to resolve her query. With all of this data on her personal machine, Alice could combine and process the data however she wants without revealing her query to any third party.

While this approach will provide the best case in terms of privacy protections, it comes at a terrible cost to query performance. As the size of the datasets being accessed increases, pure data shipping quickly becomes impractical as the cost to evaluate the query becomes dominated by the cost to transfer entire data sets to the user for processing.

Private information retrieval (PIR) has been proposed as an approach to protecting query intension in very specific circumstances [14, 47, 7, 8, 70, 80, 54, 60, 59, 22]. PIR techniques allow users to retrieve specific items from a single dataset without revealing to the server or servers hosting that dataset what item the user was interested in. While PIR has been shown provide the same privacy guarantees as pure data shipping, it can only protect the intension of relational select operations. Users that wish to issue general relational queries in a privacy preserving manner are left to process all operations themselves after retrieving results from the server via PIR.

As another approach to protecting the privacy of users accessing data stored on remote systems, Oblivious RAM (ORAM) focuses on hiding the sequence of memory locations accessed during the execution of a program [35, 63, 36, 67, 4, 17, 38, 46, 71]. Through ORAM, users can keep the actions that they perform on remotely stored data private.

Both of these approaches require explicit server-side support to protect user privacy. Even if a user wished to issue a simple selection query that could be protected using PIR or ORAM, if the server she is querying has no vested interest in protecting user privacy (e.g., `ManuCo`'s servers from the above example), neither PIR nor ORAM can be of any practical use.

This state of affairs in distributed query processing leaves users with two uncomfortable extremes: an efficient but potentially privacy-violating approach that has several distributed sites evaluate different portions of user's query, and the privacy-preserving but expensive approach pre-

sented by pure data shipping. This dichotomy leads to the central problem addressed by this dissertation:

The transition from centralized to distributed database management systems introduces novel privacy concerns for users in regards to the leakage of the query intension. Without a way to identify sensitive portions of their queries and specify how intensional revelations could violate their privacy, users must either sacrifice the efficiency and ease of use that relational database management systems have offered users for the last 40 years, or suffer privacy violations that they will not know the existence or extent of.

In this dissertation, we propose to use distribution to solve the very problem that it creates. We present the novel notion of (I, A)-privacy. Given a user's specification of what portions of her query are considered to be sensitive and what remote servers this sensitive query intension should be kept from, an (I, A)-privacy-aware query optimizer would produce an evaluation plan upholding the user's privacy constraints.

Even with a method to communicate notions of privacy to a distributed database system, however, care must be taken in how such notions are accounted for during query processing to ensure that user queries are evaluated in both an efficient and privacy-preserving manner. A simple approach to accounting for user privacy during query processing would be to use a traditional database query optimizer to determine the ordering of how a user's query should be evaluated (i.e., the order in which different datasets should be combined, and when other processing operations should occur), and then simply assign different servers to perform the operations in this ordering according to the user's notion of privacy (e.g., Alice would not want either `Pollution Watch` or `ManuCo` to combine the `Polluted_Waters` and `Supplies` tables). A similar approach is taken in [18] to enforce access controls on relational data during distributed query processing. We will show, however, that even a simple example can result in unnecessarily expensive query evaluation using this approach. In Section 6.2, we will present a counterexample that shows that users' privacy notions must be accounted for when constructing a plan to produce the results of a user's query. This underpins the hypothesis of this dissertation.

1.4 ACCOUNTING FOR USER SPECIFIED CONSTRAINTS DURING DISTRIBUTED QUERY OPTIMIZATION

It is the hypothesis of this dissertation that, by extending SQL to allow for declarative specification of constraints on the attributes of query evaluation plans and accounting for such constraints during query optimization, users can produce efficient query evaluation plans that protect the private intensional regions of their queries without explicit server-side support.

In this dissertation, we present an approach to allow users to protect the intension of queries that they issue to distributed database systems. Our approach empowers users with the ability to specifically identify the portions of their queries that they consider to be private or sensitive in nature, and denote which servers or sites in the system should be given access to this sensitive intension over the course of query evaluation. As users' ideas of what is considered to be private information will vary widely not only from person to person, but also from query to query, we ensure that our approach enables users to mark *any* portion of their query as sensitive.

This dissertation further presents an analysis of how such user constraints on query evaluation plans can be enforced. We show that our approach is *locally-enforceable*; i.e., users can ensure the protection of the intension of their queries themselves, without the explicit support of remote servers. We further develop and analyze algorithms to provide this utility to users.

Finally, we present our work towards making such protections a reality for distributed database users. To this end, we have developed PAQO, an SQL distributed query optimizer that supports user-specified constraints.

1.5 CONTRIBUTIONS

To support the hypothesis stated above, in this dissertation we make the following specific contributions:

1. In Chapter 4, we develop a formal definition of intension-based user privacy called (I, A)-privacy, which allows users to specify that some subset, I, of the intension of their query

(which we will refer to as an "intensional region") is kept hidden from some set of adversarial principals, A.

2. In Chapter 5, we present Privacy-Aware SQL (PASQL), a set of extensions to SQL with the capability of expressing (I, A)-privacy constraints. We show that that (I, A)-privacy is flexible enough to express private information retrieval (PIR) privacy constraints (Section 5.1.5). We also develop a preference algebra capable of prioritizing and balancing competing (I, A)-privacy constraints, thereby allowing users to specify complex privacy preferences in addition in addition to hard requirements (Section 5.1.3 and 5.1.4). Further, we use this preference algebra to allow users to explicitly balance the often competing goals of privacy and performance (Sections 5.3).
3. In Chapter 6, we first formally establish that (I, A)-privacy constraints can be *locally enforced*. Users wishing to protect their privacy through using (I, A)-privacy techniques can do so without explicit server-side support (Section 6.1). We then examine the tradeoff space of PASQL constraint expressiveness, query optimization performance, and query plan optimality (Section 6.3). Finally, we prove that under certain limits to expressiveness, we can use dynamic programming-based query optimization to find the fastest of the most preferred query plans for a given user query with (I, A)-privacy constraints and formally establish a worst case optimization time complexity for our algorithm to do so (Sections 6.5 and 6.6).
4. In Chapter 7, we present our Privacy-Aware Query Optimizer, PAQO. PAQO is capable of optimizing distributed queries with attached PASQL constraints using a heuristic-based algorithm that is capable of efficiently producing highly-preferred query plans (Sections 7.1 and 7.3). We formally and experimentally show that this heuristic approach requires only linear overhead to optimization time (Sections 7.2 and 7.4).

In establishing a syntax and semantics for PASQL, we create a mechanism for users to express their concerns for the privacy of the intension of their queries such that this privacy can be enforced by users alone. We show that not only is the problem of finding the fastest plan that best matches a user's privacy constraints solvable, but we provide an algorithm that provably solves this problem. Finally, we implement heuristic support for PASQL queries in PAQO to show that enforcing (I, A)-privacy constraints is practically feasible. Together, these contributions validate the hypothesis of this dissertation.

1.6 OUTLINE

The rest of this dissertation is structured as follows: Chapter 2 overviews background information on query processing and optimization. Chapter 3 presents a summary of related work to frame the contributions of this dissertation within the bodies of literature on user privacy and distributed query processing. Chapter 4 presents (I, A)-privacy, our notion of intension-based user privacy. Chapter 5 presents PASQL, our set of extensions to SQL that enable users to express constraints over the queries they use to protect the intension of their queries according to (I, A)-privacy. Chapter 6 analyzes the enforcement of PASQL constraints, exploring the tradeoffs presented by different approaches to enforcement, and showing that any approach can be taken without the need for explicit server-side support. Chapter 7 presents our implementation of a Privacy-Aware Query Optimizer, PAQO, as well as a comprehensive performance evaluation of this approach. Finally, Chapter 8 concludes this dissertation with a summary of its contributions, an overview of directions for future work, and a discussion of its impact on both the research community and society as a whole.

2.0 BACKGROUND

In this chapter, we will provide a brief overview of the background knowledge on query processing that will be assumed throughout the rest of this dissertation. Within this summary we present a classic optimization algorithm for relational queries that is used as the basis for the novel algorithms proposed in later chapters of this dissertation for enforcing (I, A)-privacy constraints.

2.1 QUERY PROCESSING AND RELATIONAL ALGEBRA

The basic processing of an SQL query involves the following steps: *parsing*, *reorganization*, *optimization*, *code generation*, and *plan execution*. The user query is first *parsed* based on whatever input query language was used (in this dissertation we deal exclusively with PASQL, which extends SQL [40]) and transformed to a representation that the query processor can operate on directly. In most query processing systems (and further what we will consider for the purposes of this dissertation), this representation is a tree of *relational algebra* operators leafed by read operations on the source database relations for the query [26].

Although databases must deal with bags of tuples, relational algebra formally operates on sets of tuples. Relational algebra can be defined in terms of six primitive operators: *selection* (σ), which returns only tuples from the input relation that match some given selection criteria; *projection* (π), which reduces the arity of the tuples it processes by eliminating unwanted attributes; *Cartesian product* ($\times$), which returns all possible combinations of tuples from two input relations; *rename* (ρ), which changes the labels of the components of the tuples it processes; *set union* ($\cup$); and *set difference* ($\setminus$). One notable operator that can be defined in terms these primitives operators is *join* ($\bowtie$). *Join* combines two input relations using selection criteria (e.g., combining a relation

containing tuples about cars for sale with a relation describing dealerships such that tuples with matching dealership ID value are combined), as opposed to *Cartesian product* which does so exhaustively.

Once the query processor has converted the user query to an internal representation, the query is *reorganized* and *optimized* according to available meta-data (e.g., relation cardinality and attribute selectivities) to ensure its efficient evaluation by the database engine(s). This optimization is possible because the relational algebra operations used to resolve database queries have great flexibility in their relative ordering. Different relations needed by a query can be combined in any order and still produce a correct result. For example, to combine three tables A, B, and C, the same result will be produced whether A and B are joined first, B and C are joined first, or A and C are joined first (e.g., $(\text{A} \bowtie \text{B}) \bowtie \text{C} = \text{A} \bowtie (\text{B} \bowtie \text{C}) = (\text{A} \bowtie \text{C}) \bowtie \text{B}$). Optimization is required because this ordering will drastically affect the resources required (e.g., CPU time, disk bandwidth, network bandwidth) required to evaluate the query. If A and B are both very large compared to C, for example, using C in the first join operation could result in a quicker overall plan by producing a much smaller intermediate result.

The optimization phase produces a *query plan* that describes the steps that an execution engine should take to efficiently compute the result of a query. Formally, we define a query plan as follows:

Definition 1 (Query Plan)**.** *A* query plan $Q = \langle N, E \rangle$ *is a directed, acyclic, fully-connected graph with a single root where* $N \subseteq \mathcal{N}$ *and* $E \subseteq N \times N$*. An element* n *of the node set* $\mathcal{N}$ *is a ternary* $n = \langle op, params, s \rangle$ *that describes a relational operator (*op*), the parameters to* op *(*$params$*), and the server at which this operator is scheduled to be executed (*s*). The edge set* E *describes the data flow, i.e., the producer/consumer relation, between relational operators.*

Similarly, we define a *well-formed query plan* as follows:

Definition 2 (Well-Formed Query Plan)**.** *A query plan* Q *is* well-formed *iff (1)* Q *corresponds to a valid relational algebra expression and (2) for each node* $n = \langle op, params, s \rangle \in Q$*, the server* s *is capable of both executing the operation described by* n *and transmitting the result of that operation to the server annotated to execute the parent node of* n *in the query plan* Q*.*

Figure 2 is a graphical representation of a well-formed query plan corresponding to Alice's query from our motivating example in Section 1.2.

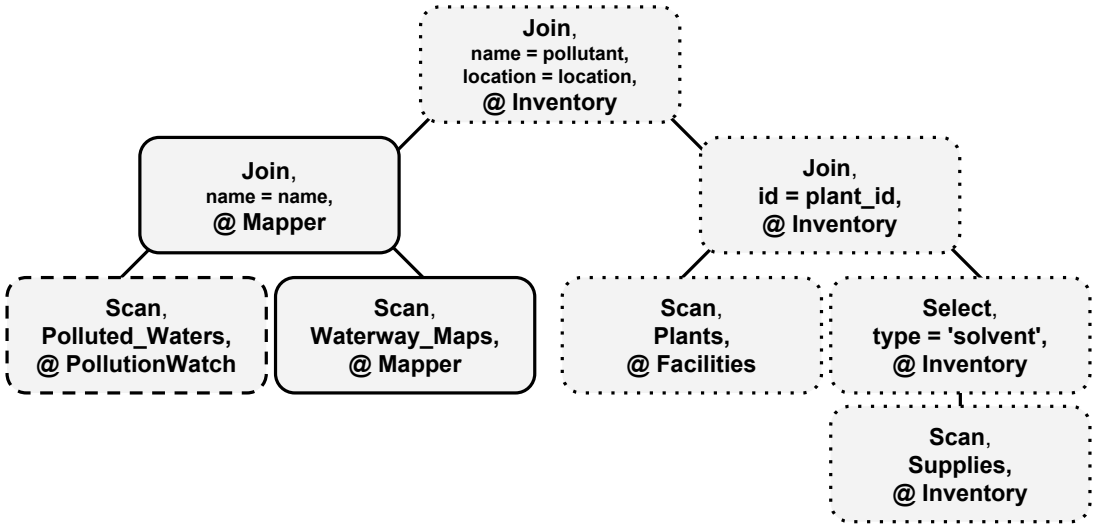


Figure 2: A query plan for Alice's query. Node borders correspond to evaluation sites.

To complete the processing of a query, a database management system must transform the query plan generated during optimization into a representation that can be directly evaluated by the database execution engine through the process of *code generation*. This code is then evaluated by the database system to produce the result of the query. This final step is known as *plan execution*. For further review of query processing and relational algebra, we refer the reader to [26].

2.2 CLASSIC QUERY OPTIMIZATION ALGORITHM

To describe the classic dynamic programming [10, 16] based query optimization algorithm from the literature [69, 44, 45], we first present the general approach in Algorithm 1, and then detail the sub-functions that it calls in Algorithms 2, 3, and 4.

As can be seen in Algorithm 1, the classic dynamic-programming based query optimization algorithm proceeds as follows. First, the algorithm enumerates all of the base relations containing data needed to evaluate the query, and then considers all of the ways that this data can be read from disk (lines 1-3 of Algorithm 1). In some cases, the entire relation will need to be read from disk and

Algorithm 1 "Classic" dynamic programming query optimization algorithm.

Require: SPJ query q on relations $R_1..R_n$, with selection/projection/join conditions $C_1..C_m$.

```
 1: for i = 1 to n do do
 2:     optPlan[{R_i}] ← AccessPlans(R_i)
 3:     PrunePlans(optPlan[{R_i}])
 4: for i = 2 to n do do
 5:     for all P ⊂ {R_1..R_n} such that |P| = i do do
 6:         optPlan[P] ← ∅
 7:         for all O ⊂ P do do
 8:             l ← optPlan[O]
 9:             r ← optPlan[P/O]
10:             cs ← {C_1..C_m}
11:             jps ← JoinPlans(l, r, cs)
12:             optPlan[P] ← optPlan[P] ∪ jps
13:             PrunePlans(optPlan[P])
14: PrunePlans(optPlan[{R_1..R_n}])
    return optPlan[{R_1..R_n}]
```

considered to produce the result of a query. When only a subset of a relation is needed, however, *indexes* on base relations can be used to greatly speed up disk reads. An index provides pointers to on-disk locations of tuples with certain attribute values (e.g., an index on the color attribute of a relation describing cars for sale would greatly speed up a query that was only looking up red cars). We refer to these considered approaches for reading data as *access paths* for the base relations specified in the query.

The best access paths for the current query are then used to bootstrap the bottom up dynamic programming approach to generate query plans. Iterating over increasing join levels (the number of base relations involved in a query plan, line 4), the algorithm combines all feasible previously-realized subplans towards constructing a plan to evaluate the entire query. For example, the algorithm will combine all possible access plans to find all 2-way joins. It will then look at all feasible combinations of 2-way joins and access plans to produce 3-way joins. From there it will proceed to examine all feasible combinations of 2-way joins (bushy joins) and all feasible combinations of 3-way joins and access plans (left/right deep plans) to produce 4-way join plans and so on until all base relations in the query are joined together. Once all n-way joins have been established, the fastest plan to resolve the query is returned.

Algorithm 2 ACCESSPLANS: Access plan enumeration.

Require: A relation R_i to enumerate access plans for.
Require: The conditions $C_1..C_M$ specified as part of the query.
1: $aPlans \leftarrow \emptyset$
2: $c \leftarrow$ POSSIBLE_CONDS$(R_i, \{C_1..C_m\})$
3: **for** each physical scan operator, po **do**
4: $\quad New \leftarrow$ NEW_SCAN(po, R_i, c)
5: $\quad aPlans \leftarrow aPlans \cup \{New\}$
return $aPlans$

The possible access plans are rather simply enumerated by the ACCESSPLANS function (called on line 2 of Algorithm 1, and shown in Algorithm 2), while the possible ways of performing the join of each pair of subplans are enumerated through the use of the JOINPLANS function (called on line 11 of Algorithm 1, and shown in Algorithm 3). In this classic algorithm, both functions enumerate all applicable SQL conditions that can be applied to the result of the join (e.g., selections, projections, aggregates) and iterate through all possible physical operators for performing the corresponding relational operations on their targets (e.g., nested loop joins, merge joins, or hash joins on two subplans in JOINPLANS; sequential scans, index scans, bitmap scans on individual base relations for ACCESSPLANS).

In either case, after enumerating potential plans, *dominated* plans must be pruned so that fewer options need be investigated in constructing plans at later join levels (lines 3 and 13 of Algorithm 1). As shown in Algorithm 4, one plan dominates another that joins the same set of relations if it is faster and has at least an equally *interesting* sort order. Operations producing specific sorted orders of tuples are said to be interesting if they can be used to speed up later operations in the query plan (e.g., if tuples are already sorted according to a `name` attribute, a merge join on the `name` attribute can be performed without an explicit sort operation). PRUNEPLANS is also called at the end of Algorithm 1 to ensure only the best plan to evaluate the query remains. In the classic case, the *best* plan is the *fastest* plan. We formally define the notion of plan domination used in the Classic algorithm as follows:

Algorithm 3 JOINPLANS: Join enumeration psuedocode for the classic dynamic programming query optimization algorithm.

Require: A set of plans that will make up the left side of a new root join $Left_1..Left_u$.
Require: A set of plans that will make up the right side of a new root join $Right_1..Right_v$.
Require: The conditions $C_1..C_M$ specified as part of the query.
1: $joinPlans \leftarrow \emptyset$
2: **for** $i = 1$ to u **do**
3: **for** $j = 1$ to v **do**
4: $c \leftarrow$ POSSIBLE_CONDS$(Left_i, Right_j, \{C_1..C_m\})$
5: **for** each physical join operator, po **do**
6: $New \leftarrow Left_i \bowtie_{po,c} Right_j$
7: $joinPlans \leftarrow joinPlans \cup \{New\}$
return $joinPlans$

Definition 3 (Classic Plan Domination)**.** *A plan p is said to* dominate *another plan p' in the Classic algorithm if:*

- *p and p' join the same set of tables AND*
- *p sorts tuples based on an equal or extended list of attributes compared to p' AND*
- *p is faster than p'*

This notion of plan domination ensures that any plan pruned from the search space can be completely replaced by another discovered plan anywhere it could have been utilized as the basis for future plans. Note that the Classic algorithm uses a heuristic of "pushing down" SQL conditions (e.g., selection criteria, join conditions, projection operations) as far down in the tree as possible. As soon as one of these conditions can be evaluated, it is added to the plan (lines 2 and 4 of Algorithm 2 and lines 4 and 6 of Algorith 3). This makes it tivial to see how access plans for the same table can replace one another. Any plan to read data from a base relation will apply any necessary selection and projection conditions when scanning the data and, hence, any plan to access a given table will produce the same resulting tuples.

For all other plans, we note that the subproblems of joining two base relations together, then three, then four, etc., exhibit optimal substructure. For example, the optimal plan for A $\bowtie$ B can always be extended to the optimal plan for (A $\bowtie$ B) $\bowtie$ C. As such, there is no harm in pruning

Algorithm 4 PRUNEPLANS: A function to prune dominated plans from a given join level according to classic domination metric.

Require: A list Q of query plans joining the same number of base relations.

```
1:  for all p ∈ Q do
2:      reject ← False
3:      for all other ∈ Q|p! = other do
4:          if ROOT_SITE(p) == ROOT_SITE(other) then
5:              if p has a more interesting sort order then
6:                  if cost(p) < cost(other) then
7:                      Q.remove(other)
8:              else if cost(plan) > cost(other) then
9:                  reject ← True
10:                 break
11:     if reject then
12:         Q.remove(p)
```

suboptimal partial plans. Further, any plan that joins the same relations will input all of the same base relations, and (at varying points) apply the same SQL conditions. Hence, any plans that join the same base relations will produce the same results. By finding the best plans for joining all pairs of base relations together, then all sets of three base relations together and so on, the classic algorithm can cut large number of potential plans from the search space that it must explore while still arriving at the optimal solution.

We will refer to the overall algorithm realized through the combined pseudocode presented in Algorithms 1, 2, 3, and 4 as the *Classic algorithm*.

A simple approach to implementing support for (I, A)-privacy constraints in a distributed database system would be to take any centralized database management system that implements this classic query optimization algorithm, and then perform site selection for each operation in the resulting plan to support the constraints attached to a given query. [18] uses such a two-phase optimization approach to enforce access controls on relational tuples as they are processed during query evaluation. As we show in Section 6.2, however, this post-processing approach can result in woefully inefficient query plans compared to an approach that accounts for constraints during query optimization (taking, in one case, an estimate of over 24,000 times as long to execute). Hence, constraint adherence should be accounted for during query optimization.

2.3 SUMMARY

In this chapter, we overviewed necessary background information assumed throughout the rest of this dissertation. We formally established the concept of a query plan, which will be used and referenced throughout this dissertation. We also presented the classic dynamic programming algorithm which is used as the basis for the novel optimization algorithms presented in this dissertation.

3.0 RELATED WORK

We now overview several areas of recent work related to this dissertation. Specifically, we discuss the relation of our work to (i) data privacy technologies, (ii) techniques for private database operations, (iii) private information retrieval, (iv) oblivious RAM, (v) distributed query processing, and (vi) query optimizers built for other security and privacy goals.

3.1 DATA PRIVACY

Often, discussions of database privacy concerns focus on the privacy of those whose information is *stored* in the database. This area of work was founded in response to the discovery that most Americans can be uniquely identified by their 5-digit ZIP code, gender, and date of birth [68, 74]. This troubling revelation entailed that data releases previously considered to present anonymized data could be easily deanonymized.

k-anonymity [68, 74] was presented as the first approach to solve this problem of properly anonymizing data releases. k-anonymity is underpinned by the concept of *quasi-identifiers*: attributes of the data that could be used to identify the individuals that the data describes. k-anonymity is a syntactic privacy condition such that a data release is said to be k-anonymous if there are no fewer than k entries in the release that share a unique set of quasi-identifiers.

A key weakness of k-anonymity is that if all k entries in the release exhibit the same sensitive trait (e.g., a set of k entries sharing the same quasi-identifiers in a k-anonymized release of medical records are all shown to have cancer), then the privacy of the people represented by those k entries could still be violated. To address this concern, l-diversity was proposed in [51]. l-diversity improves upon k-anonymity by stating that all blocks of k similar entries in a data release should

contain l sufficiently different values for sensitive attributes of the release. This notion was again refined in [49] to give rise to t-closeness. A release is said to exhibit t-closeness if the l different values for a sensitive attribute within a block of k entries exhibit a difference from the distribution of sensitive attribute values for the table as a whole that is no greater than a threshold, t.

A contrasting approach towards a similar goal is presented by work on differential privacy [24, 25]. Differentially private databases aim to maintain the utility of information stored in a database while ensuring that the answer to a query returned by a database containing information pertaining to a certain user is indistinguishable to the answer that would be returned had that user's data *not* been in the database. This is accomplished by adding noise to query results according to some carefully chosen distribution to provide strong, semantic guarantees of privacy for people whose data is stored in the database. Specifically, a randomized function K gives ϵ-differential privacy if, for all data sets D and D' differing by at most one row, and all $S \subseteq Range(K)$:

$$Pr[K(D) \in S] \leq exp(\epsilon) \times Pr[K(D') \in S]$$

Where the probability is over the coin flips of K.

While protecting the privacy of users whose information is contained within released or available datasets is still an active and interesting area of research, it solves a problem orthogonal to that which this dissertation addresses. In this dissertation, we protect the privacy of users who are querying distributed database systems.

3.2 PRIVACY-PRESERVING JOINS

Work has also been done to allow for set operations or joins to be performed on data sets from two independent parties while revealing to each party no more about the other's input data set than can be inferred from the output of the operation. In [3], it was established that set intersection, set intersection result size, equijoin, and equijoin result size can be performed and nearly meet such data privacy-preserving assumptions (the equijoin-size protocol can leak the existence of duplicates

or even the intersection set). Secure co-processors have been utilized to eliminate such leakages and allow for more general operations [2].

Similarly, these works are not concerned with protecting the operation being performed or the arguments to that operation (i.e., the intension of the query), only relational data provided as input but not present in the result.

3.3 PRIVATE INFORMATION RETRIEVAL

In essence, Private Information Retrieval (PIR) is a technique for retrieving some information from a database without revealing to the server(s) hosting that database the criteria for selecting items from that database (the indices of the bits that the querier is interested in). The problem addressed by PIR was originally formulated as follows: given some l servers that store replicated copies of a database that is viewed as a length n binary string $x = x_1 \dots x_n$, the goal of PIR is to allow a user to learn the value of some desired bit x_i without allowing any server to gain information about the value of i [14]. Initially, l was specified to be $l \geq 2$ to achieve a communication complexity of less than $\mathcal{O}(n)$ bits, where n is the size of the dataset being queried. The approach presented in [14] requires multiple non-colluding servers to host replicas of the database that a user wishes to access, and necessitates a communication complexity of $\mathcal{O}(\sqrt{n})$ bits. This approach provided strong information-theoretic privacy guarantees to users, and has appropriately come to be known as *information-theoretic PIR*. Recent advances in information-theoretic PIR have focused on not only improving the required communication complexity, but also improving the robustness of the schemes in the face of Byzantine adversaries [48]. The issue of robust PIR was first addressed in [7, 8]. These works established that for l servers, if k of the servers respond to a PIR request and v of those k respond incorrectly, their scheme will tolerate t colluding servers without revealing the user's query if $v \leq t < k/3$. In [22], a scheme exhibiting the theoretically maximum Byzantine robustness is demonstrated ($v < k - t - 1$) with a communication complexity of $\mathcal{O}(v(k + l))$. A method for performing information-theoretic PIR using widely implemented database access methods (hash indices and B^+ tree indices) has also been demonstrated in [59].

PIR using only a single server was established in [47]. Given the proof from [14] that information-theoretic PIR with a single server requires a communication complexity of $\mathcal{O}(n)$ bits, however, single server approaches instead relax their security assumptions to assume a computationally-bounded adversary. Hence, the single-server is also referred to as *computational PIR*. In [47], the security of the presented scheme is based on the quadratic residuosity problem.

While the practical feasibility of single-server has been called into question [70], more efficient techniques for performing computational PIR schemes have since been proposed. In [55], an approach to PIR is proposed that bases its security in linear algebra. The authors introduce a novel security problem, the Hidden Lattice Problem, and show that it is related to NP-complete coding-theory problems. This approach is especially amenable to implementation on general purpose computing on graphics processing units (GPGPU) as demonstrated in [54]. The authors of [60] demonstrate that this approach, as well as advanced approaches to multi-server PIR, could perform up to an order of magnitude faster than a trivial transfer of the entire database for users operating on sufficiently large databases over an average consumer-grade network connection. Efficient approaches to computational PIR have also been demonstrated through the use of secure co-processors [80].

A hybrid information-theoretic/computational PIR scheme is proposed in [21]. This scheme combines the computational efficiencies established in [55] with the robust, communication-efficient properties of [22]. The resulting scheme maintains the robustness of [22] and offers performance superior to either of the base schemes.

Systems implementing (I, A)-privacy support can utilize any of these techniques in the special case that (1) the user specifies privacy constraints on the execution of her query that can be achieved through the use of PIR, and (2) the database servers providing the required data support a practical technique for PIR. This requirement in and of itself also highlights an advantage of our work: it allows users to protect their privacy when interacting with database severs that have only rudimentary query processing capabilities (see Proposition 1 in Section 6.1), while still providing the capability for users to take advantage of more advanced techniques such as PIR, when they are available and applicable.

3.4 OBLIVIOUS RAM

Oblivious RAM (ORAM) algorithms hide the sequence of memory locations accessed during the execution of a program [35]. These algorithms can be used to protect the privacy of users that wish to access data that they store on remote servers. By continually shuffling and re-encrypting data as it is accessed on a remote server, the actions the user performs on that data can be kept private. Since its initial proposal, a large body of ORAM approaches have been presented in the literature [63, 36, 67, 38, 46, 71]. The efficiency of these schemes is measured by the amount of data that must be stored by a client to run an ORAM algorithm, the storage overhead for the server hosting the data, and the amortized overhead to read and write data to ORAM. While the security of most ORAM schemes relies on the client having access to a stream a truly random data (a random oracle), approaches that offer information-theoretic security guarantees have also bee proposed [4, 17].

Just as was the case for PIR schemes, however, protecting user privacy through ORAM requires active participation by the server. In this dissertation, we show that our approach to protecting user privacy can be enforced without explicit server-side support.

3.5 DISTRIBUTED QUERY PROCESSING

Distributed query processing is typically performed by either shipping the data required for the query back to the site that issued the query for processing (data shipping), or shipping pieces of the query out to the sites holding the data for parallel processing, returning only the result to the issuing site (query shipping) [44]. These techniques can further be combined as a form of hybrid shipping [32]. To date, however, there has not been a realization of a system to evaluate queries over systems of fully autonomous, heterogeneous, distributed databases.

The closest example of our system model in the literature is that of mutant query plans [65]. Mutant query plans are an extension to XML queries that are partially optimized and evaluated by several remote servers over the course of their processing. A user may issue a relatively simple mutant query to a single server. That server would then attempt to evaluate what it could of the query,

reoptimize the query around what remains (i.e., determine a faster plan to evaluate everything else in the query if such a plan exists), and then forward the query to another server that "knows more" about the portion of the query yet to be evaluated. Consider the following example. A user wishes to know the list of highly-reviewed sci-fi movies playing near them. The user could construct a mutant query plan and send it to a local theater's server. That server would add information about the shows playing at the theater and then pass the mutant query plan off to a movie review site. The review site would then add its information about reviews of sci-fi movies to the plan, combine this data with what was received from the theater and return the result to the user.

Note that mutant query plans make no effort to protect the intension of user queries. In fact, they reveal to remote servers *more* of the intension of the user's query than the servers need to know for the query to be evaluated.

In the same manner that our model proposed here is able to glean the advantages of PIR when possible, however, it can also utilize all of these query processing techniques to construct query plans that sufficiently balance user privacy preferences with query performance, realizing the proposed hybrid query processor from [30].

3.6 QUERY OPTIMIZATION FOR PRIVACY AND SECURITY

In [18], the authors assume that relational tables in a distributed database system carry strict access controls specifying what servers in the system should be able to access the table's data during distributed query evaluation. To uphold these access controls, the authors propose to modify the optimization process to be two-phased. First, an off-the-shelf query optimizer is used to determine join ordering and produce a query evaluation plan as if the query were to be evaluated in a centralized database system. Evaluation sites are then assigned to each of the operations in this plan such that the table access controls are upheld. The authors do not provide a working implementation of this system or optimizer, they simply present a framework for enabling such access-control aware optimization.

The authors of [13] also strive to protect data stored in their system, but from a very different adversary. The adversary here is a passive piece of malware with access to the memory of a

database server for a limited period of time. Thus, the authors deal with encrypted databases, and optimize queries 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 both of these works use a similar approach to the one presented in this dissertation—i.e., modifying the optimization process to address security and privacy concerns—they solve very different problems and are thus orthogonal to the work presented in this dissertation.

4.0 (I, A)-PRIVACY

We begin this chapter by formally establishing the system model that is assumed throughout this dissertation. With this in hand, we move on to define our novel notion of querier privacy: (I, A)-privacy. We then define (I, A)-privacy constraints as constructs that users can issue alongside their queries to specify what portions of their queries should be considered to be sensitive and what servers this sensitive intension should be kept from. We conclude this section with a formal overview of how a query plan can be determined to either support or violate (I, A)-privacy constraints.

4.1 SYSTEM MODEL

In this section, Alice's example query from Section 1.2 is used to assist in defining our system model. An illustration of the flow of processing Alice's query through a distributed database system is shown in Figure 1: Alice issues her query to a query optimizer, 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. The optimizer 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 the optimizer.

4.1.1 System

We assume that query optimizers in this model produce query plans to be issued over a loosely federated 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, optimizer 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 [23]). 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 [31, 77]). As such, we assume that users only obtain the results that they are authorized to see.

We further assume that query optimizers know 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 that 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 query optimizers can intelligently process distributed queries. We assume 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.

These assumptions comprise a standard approach to distributed query processing discussed in the literature [44].

4.1.2 Trust Model

We assume that the user has access to a trusted query optimizer running on either a personal machine or a trusted server. Throughout this disseration we will refer to the optimizer instance being used for a given query as the *querier*. We assume the querier to be fully trusted by the user. We consider this assumption reasonable, not only due to the user's ability to run and maintain her own personal optimizer instance, but further because users must reveal their queries to some piece of software in order to have them 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 [11]. 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's 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.

4.2 QUERIER PRIVACY

The vast majority of research in database privacy deals with preventing the disclosure or inference of sensitive tuples stored in database systems. By contrast, the focus of this dissertation lies in the protection of end-user privacy during the execution of distributed queries. In this section, we develop the notion of (I, A)-privacy, in which sensitive portions of a user's query are protected from disclosure to a set of colluding adversaries. While query intension is generally represented using a declarative language like SQL (and, indeed, working knowledge of SQL is the only background users need in order to author protections over the privacy of such intension through the use of our model), we focus in this dissertation on how this knowledge is encoded in the actual query plan used during the distributed query evaluation process, as this is the information that would actually be gleaned by adversaries according to our threat model.

Assuming that $\mathcal{L}$ denotes the set of all SQL queries, we can then formally represent a query planner as a function $\mathsf{plan} : \mathcal{L} \rightarrow \mathcal{Q}$ that generates query plan $Q \in \mathcal{Q}$ from an SQL query $q \in \mathcal{L}$.

Note that our system model leads to two additional specifications of a query plan to be considered. First is that of a *locally-expanded query plan*:

Definition 4 (Locally-Expanded Query Plan)**.** *A well-formed query plan Q is said to be* locally-expanded *with respect to a server $s \in \mathcal{S}$ if every leaf node ℓ of Q is either (1) an operation*

annotated for execution at some remote server $s' \neq s$, or (2) the scan of a table managed by the server s. We denote the set of all locally-expanded query plans for a query $q \in \mathcal{L}$ as $\mathscr{L}(q)$.

We use the term *scan* to denote the retrieval of tuples from a database relation without specifying an access method (e.g., the use of a particular index). Given a query $q \in \mathcal{L}$, the output of a query planner plan at a specific server s is a locally-expanded query plan $Q \in \mathscr{L}(q)$ with respect to the server s. The annotated leaves of the resulting query plan may be further expanded by other nodes in the network during query processing (e.g., during the expansion of a remote view). This leads to the following definition:

Definition 5 (Globally-Expanded Query Plan)**.** *A well-formed query plan Q is said to be* globally-expanded *if every leaf node ℓ of Q represents the scan of a relational table managed by some server $s \in \mathcal{S}$. We denote the set of all globally-expanded query plans for a query $q \in \mathcal{L}$ as $\mathscr{G}(q)$.*

Note that a globally-expanded query plan is trivially a locally-expanded query plan relative to every server $s \in \mathcal{S}$. That is, no server can further expand a globally-expanded query plan. Furthermore, given some initial query, a corresponding globally-expanded query plan may not ever be learned by any one server in the system: each server generates a locally-expanded query plan, but may perhaps be unaware of how its plan is further expanded by others.

In general, a user may only consider certain parts of her query to be sensitive. In our example, Alice may not mind if people know that she is interested in data from the `Facilities` server's `Supplies` table, but she would certainly want the fact that it will be combined with data from `Pollution Watch` to be kept private. To formalize this notion, we introduce the following term:

Definition 6 (Intensional Region)**.** *An* intensional region *is a countable subset I of the node set $\mathcal{N}$.*

Example 1. *To illustrate this concept, we can refer back to Alice's query plan from Figure 2. Consider the case in which Alice wants to keep her use of the `Polluted_Waters` table's `pollutant` attribute secret from `ManuCo`'s `Inventory` server. An intensional region which contains the single node in her query plan could then be defined to represent the specific portion of the intension of Alice's query that she wishes to protect. Similarly, if Alice wanted all parts of her query in which data from different relations is joined to be kept private, an intensional region containing all join nodes could be defined.*

The above definition of intensional region is flexible enough to identify very specific descriptions of intension considered to be sensitive, like "all selection operations that involve the constant 'solvent'." Given such intensional regions, we can informally say that a query plan Q maintains a user's privacy if no adversarial principal can learn information from nodes in intensional regions identified by the user as sensitive. To continue to reason about this notion of user privacy, however, we must first define how query intension is revealed to principals in the system during the evaluation of a user's query.

Definition 7 (Intensional Knowledge)**.** *Given a query plan $Q = \langle N, E \rangle$, we denote by $\kappa_s(Q) \subseteq N \cup E$ the* intensional knowledge *that server $s \in \mathcal{S}$ has of the query encoded by the plan Q. At a minimum, $\kappa_s(Q)$ contains the set of all locally-expanded query plans for each node $n \in N$ annotated for execution by the server s, and further all edges leaving or entering such nodes.*

Given a colluding set of principals $A = \{s_1, \ldots, s_k\}$, we can define the combined intensional knowledge of the colluding servers A in the natural way, i.e., $\kappa_A(Q) = \bigcup_{s_i \in A} \kappa_{s_i}(Q)$. For example, the intensional knowledge of the four principals participating in the evaluation of the query in Figure 2 is as follows:

$$
\begin{aligned}
\kappa_{PollutionWatch}(Q) \quad &= \langle \texttt{scan}, \{(\texttt{Polluted\_Waters})\}, \texttt{Pollution Watch} \rangle \\
\kappa_{Mapper}(Q) \quad &= \langle \texttt{scan}, \{(\texttt{Waterway\_Maps})\}, \texttt{Mapper} \rangle \\
&\quad \text{and } \langle \texttt{join}, \{(\texttt{name}, =, \texttt{name})\} \texttt{Mapper} \rangle \\
\kappa_{Facilities}(Q) \quad &= \langle \texttt{scan}, \{(\texttt{Plants})\}, \texttt{Facilities} \rangle \\
\kappa_{Inventory}(Q) \quad &= \langle \texttt{select}, \{(\texttt{type}, =, \text{“solvent”})\}, \texttt{Inventory} \rangle \\
&\quad \text{and } \langle \texttt{scan}, \{(\texttt{Supplies})\}, \texttt{Inventory} \rangle \\
&\quad \text{and } \langle \texttt{join}, \{(\texttt{id}, =, \texttt{plant\_id})\}, \texttt{Inventory} \rangle \\
&\quad \text{and } \langle \texttt{join}, \{(\texttt{name}, =, \texttt{pollutant}), \\
&\qquad (\texttt{location}, =, \texttt{location})\}, \texttt{Inventory} \rangle
\end{aligned}
$$

The above definitions enable us to precisely define the notion of privacy that we will explore in the remainder of this dissertation as follows:

Definition 8 ((I, A)-privacy)**.** *Given an intensional region I and a set of colluding adversaries $A \subseteq \mathcal{S}$, a globally-expanded query plan Q is said to be* (I, A)-private *iff $\kappa_A(Q) \not\models I$, where $\models$*

denotes an inference procedure for extracting intensional knowledge from a collection of query plans.

In the above definition, there are many possible candidates for the $\models$ relation. For the purposes of this dissertation, we will focus our attention on the syntactic condition of the containment relation $\sqsupseteq$, which we formally define as follows:

Definition 9 ($\sqsupseteq$)**.** *Let Q be a globally-expanded query plan, I be an intensional region, A be a set of colluding adversaries, and $\kappa_A(Q) = (N, E)$ be the intensional knowledge that the adversary set A has about the query plan Q. Then $(N, E) \sqsupseteq I$ iff $N \cap I \neq \emptyset$.*

This relation denotes whether an intensional region explicitly overlaps some collection of adversarial knowledge. We will be exploring a semantic relation for $\models$ as a subject of future work. Using the $\sqsupseteq$ relation as our inference procedure $\models$ is a natural first step for exploring end-user privacy as it is expressive enough to encode private information retrieval (PIR) constraints. This claim will be explored further and proved in Section 5.1.5.

We next overview our approach to empowering users to protect their privacy using (I, A)-privacy constraints, and show how users can define intensional regions through the use of node descriptors.

4.3 PROTECTING QUERIER PRIVACY

Our approach empowers users to specifically identify portions of their queries that require special handling during query evaluation, and then to author conditions that must be upheld by any query plan nodes operating on an identified portion of the query (e.g., a user could want to ensure that no operations on the `pollutant` attribute of the `Polluted_Waters` table are evaluated by `ManuCo`'s `Inventory` server). We refer to such a notion as an (I, A)-privacy constraint:

Definition 10 ((I, A)-privacy constraint)**.** *An* (I, A)-privacy constraint *is a three-tuple $\langle I, \Diamond, A\rangle \in 2^{\mathcal{N}} \times \{+, -\} \times 2^{\mathcal{S}}$. Such a tuple indicates that the intensional region I is either required to flow $(+)$ or forbidden from flowing $(-)$ to principals in the set A.*

In order to enforce these constraints during query evaluation, users must be able to express their specifications and conditions to the optimizer. Having users direct express intensional regions as they are formally defined (e.g., by having users enumerate the nodes that make up their desired intensional region) would prove impractical and error prone. Hence, we need some way that users can succinctly define the intensional regions they wish to use in their (I, A)-privacy constraints. We have developed the *node descriptor* for this precise reason.

4.3.1 Defining Intensional Regions

A node descriptor is a matching construct that can be used to define an intensional region. For a given node descriptor, any query plan node $n \in (N)$ that is matched by the node descriptor is a member of the intensional region defined by that node descriptor. A user could, for example, define a node descriptor to match all nodes that operate on the `pollutant` attribute of the `Polluted_Waters` table. This node descriptor's intensional region would contain all query plan nodes that contain `pollutant` as part of their $params$. We formally define node descriptors as follows:

Definition 11 (Node Descriptor)**.** *A node descriptor $d \in \mathcal{D}$, is a triple $d = \langle op, params, s \rangle$ describing the relational operator (op), a set of parameters to the given operator ($params$), and the principal at which this operator shall be executed (s). A node descriptor is well formed if all of the following hold true:*

- *op is either a valid relational algebra operation, free variable, or $*$ (matches any operation)*
- *$params$ is either a set of sets (any of which may have a free variable) or $*$ (matches any parameters)*
- *s is either the location of some server in the system, a free variable or $*$ (matches any server)*

We will consider $\mathcal{V}$ to be the set of all free variables that can be declared. To determine whether some query plan node n matches a given descriptor, we define a matching operator ($\pitchfork$) as follows:

Definition 12 (Matching)**.** *Given a node descriptor $d \in \mathcal{D}$, and a node $n \in N$, $d \Cap n$ if and only if*

$$(d.op = n.op \vee d.op \in \mathcal{V} \vee d.op = *) \ \wedge \ (d.s = n.s \vee d.s \in \mathcal{V} \vee d.s = *) \wedge$$
$$[\forall a \in d.params : (a \in \mathcal{V} \ \vee \ (a \in n.params \vee \exists a' \in n.params : a \subseteq a')) \ \vee d.params = *]$$

Hence, an intensional region I can be defined in terms of a node descriptor d as a subset of $\mathcal{N}$ as: $I = \{n \mid n \in \mathcal{N} \wedge d \Cap n\}$.

While matching on op or s is rather intuitive, matching on $params$ requires a bit of explanation. $params$ is a set of sets in both query plan nodes and node descriptors that represents the arguments to a relational algebra operator. To allow for easy and concise expression of node descriptors, we state that a node descriptor matches a query plan node based on $params$ if every ordered set in the descriptor's $params$ is either contained directly in the query plan node's $params$ or is a subset of an ordered set in the query plan node's $params$.

Example 2. *Our definition of $\Cap$ allows nodes with complex $params$ attributes to be easily matched. Consider the following query plan node:*

⟨ *select,* { *(*`at1`*, =, 42), (*`AND`*), (*`at2`*, <, 10)* }*, example_server* ⟩

Any of the following node descriptors will match this node based on our definition of $\Cap$:

- ⟨ **,* { *(*`at1`*)* }*, ** ⟩
- ⟨ **,* { *(*`at1`*, =, 42)* }*, ** ⟩
- ⟨ **,* { *(*`at1`*, 42)* }*, ** ⟩
- ⟨ **,* {*(*`AND`*)* }*, ** ⟩*;* ⟨ **,* { *(*`at1`*, 42), (*`AND`*)* }*, ** ⟩
- ⟨ **,* { *(*`at1`*), (*`AND`*), (*`at2`*)* }*, ** ⟩

Note that this is not an exhaustive list of all node desciptors that would match the node described above; they merely present an illustrative sample of possible node descriptors.

With this, we show how node descriptors can allow users to succinctly specify intensional regions. However, in order for this construct to be useful, we must further demonstrate that node descriptors offer sufficient utility so as to allow users to protect any portion of the intension of their queries.

4.3.2 Proof of Completeness

As has been previously stated, privacy is an inherently personal property, and hence, we must ensure that our approach of using node descriptors to define intensional regions is sufficiently general to allow users to define regions based on *any* part of the intension of their query.

Theorem 1. *For any SQL query $q \in \mathcal{L}$, it is possible to specify a node descriptor $d \in \mathcal{D}$ that identifies any clause of q and/or its components (i.e., table or view names, constraints, constraint operators, attributes, or constants). Then, for any query plan $Q = \langle N, E \rangle$ that materializes q, the set of nodes $C = \{n \mid n \in N \wedge d \pitchfork n\}$ contains* exactly *those nodes corresponding the specified clause and/or components of q.*

The proof of this theorem is a case-by-case analysis showing that for any valid SQL operator with any valid set of arguments, there exists a node descriptor that matches the corresponding node in a query plan:

Proof. To prove our claim, we will first demonstrate the ability of our annotation and matching scheme to identify any and all query plan nodes resulting from node descriptors flagging specific clauses of an SQL query q. We then show that this scheme can also annotate any other semantically-meaningful portion of an SQL query that may span multiple clauses. Specifically, we show how to define node descriptors annotating a particular table or view name, constraint operator, attribute, or constant as being of interest.

SQL queries can consist of multiple `SELECT` statements combined through the use of one of three set operators (`UNION`, `INTERSECT`, or `MINUS`), each of which consists up to six different clauses: `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING`, and `ORDER BY`. We now demonstrate the ability to match against any portion of any clause of a `SELECT` statement, and against the inclusion of any operators combining select statements.

SELECT: The `SELECT` clause consists of the optional keyword `DISTINCT` and a list of relational attributes and aggregate functions over relational attributes or *.

Case 1. The `DISTINCT` keyword would add a deduplication operator into the resulting query plan, and hence a user wishing define an intensional region over query plan nodes resulting from the inclusion of `DISTINCT` could identify such a region by a node descriptor matching all deduplication operations as such:

$$\langle \texttt{deduplicate}, *, * \rangle$$

By the definition of $\pitchfork$, this node descriptor would match only the `deduplicate` nodes in any query plan materializing q, which are present *iff* the `DISTINCT` keyword is used in q.

Case 2. Should a `SELECT` clause include any relational attribute names, an intentional region for each attribute could be identified through a node descriptor of the following form:

$$\langle \texttt{project}, \{ \text{(}\textit{attribute\_name}\text{)} \}, * \rangle$$

By the definition of $\pitchfork$, this node descriptor would match only the `project` nodes for a specific attribute in any query plan materializing q. Such nodes are only induced if a projection attribute list is passed to the `SELECT` keyword in q.

Case 3. In the case that `SELECT` clause includes an aggregate function, an intensional region containing all nodes operating with that function can be identified by:

$$\langle \texttt{aggregate}, \{ \text{(}\textit{function}\text{)} \}, * \rangle$$

Any relations that are aggregated by a function can be identified by:

$$\langle \texttt{aggregate}, \{ \text{(}\textit{relation\_name}\text{)} \}, * \rangle$$

By the definition of $\pitchfork$, this node descriptor would match only the `aggregate` nodes for a specific function or relation name in any query plan materializing q. Such nodes are only present in a query plan if q utilizes an aggregate function in its selection list.

Case 4. Note that a specification of `SELECT` $\star$ would not add a specific projection to the resulting query plan, and hence there would be no node in the resulting query tree to match against. This lack of an explicit project operation does not, however, leak user intension as any remote server evaluating some portion of a query plan would have no way of knowing whether or not the querier would perform a project operation as a final step in evaluating a query upon receiving all intermediate results of that query. Since we are optimizing for privacy as well as performance, query plans containing such project operations that had not been pushed down the plan closer to operations scanning base relations are quite possible.

FROM: The `FROM` clause lists the relations that should supply the data for a query to operate on. As such, any user intension specified in a `FROM` clause can be easily identified by a node descriptor of the following form:

$$\langle \texttt{scan/view}, \{ \text{(}\textit{relation\_name}\text{)} \}, * \rangle$$

By the definition of $\pitchfork$, this node descriptor will match all nodes representing the scan of the specified view or table for any query plan materializing q. Further, `scan/view` nodes are only present in a query plan if the corresponding table or view is listed as a data source in the `FROM` clause of q.

WHERE: The `WHERE` clause of an SQL query q specifies conditions to be applied by the `select` operations present in any query plan materializing q. These conditions can be quite complex, consisting of several subconstraints combined through the use of the operators `AND` and `OR`. In the syntax presented in Section 5.1.1, we enumerate all operators that can be involved in such conditions in the <pop> term. Without loss of generality, we describe how to match a constraint consisting of a left operand, an operator, and a right operand (e.g., `salary > 75,000`):

$$\langle *, \{ \text{(}\textit{operand}, \textit{operator}, \textit{operand}\text{)} \}, * \rangle$$

A * is used in place of the operation in the node descriptor above, since conditions may appear in either `select` or `join` operations in some query plan materializing q. By the definition of $\pitchfork$, the above node descriptor will exactly match the identified constraint, regardless of the operational node in which it appear.

GROUP BY: A `GROUP BY` clause's presence in an SQL query identifies the relational attributes that will be used to group tuples resulting from an aggregate operation. Node operators that identify intensional regions containing these relational attributes are formed as:

$$\langle \text{aggregate}, \{ \text{(attribute\_name)} \}, * \rangle$$

By the definition of $\pitchfork$, the above node descriptor will match all nodes within any query plan materializing the query q in which the specified `GROUP BY` clause was specified. Note that this does not match `aggregate` nodes induced by aggregate functions included as part of a `SELECT` statement.

HAVING: `HAVING` clauses are used to specify conditions on the tuples resulting from an aggregate operation. As a result, their parameters can be identified in a very similar manner to those of the `WHERE` clause. Without loss of generality, we demonstrate how to identify nodes expressing a `HAVING` constraint consisting of a function, and argument, an comparison operator, and a constant value (e.g., `AVG(salary) > 50,000`):

$$\langle\ *,\ \{\ (\textit{function}, \textit{attrib}, \textit{operator}, \textit{constant})\ \},\ *\ \rangle$$

By an argument similar to that given for the `WHERE` clause, the above node descriptor matches exactly the nodes corresponding to the identified `HAVING` of any query plan materializing the query q.

ORDER BY: An `ORDER BY` clause in an SQL statement would directly result in an ordering operation being added to a query plan. As the `ORDER BY` clause takes as parameters pairs of attribute names and one of the keywords `ASC` or `DESC`, the intension represented by either of these pieces of a parameter can be explicitly identified by one of the following node descriptors:

$$\langle \texttt{sort},\ \{\ (\textit{attribute\_name})\ \},\ *\ \rangle$$
$$\langle \texttt{sort},\ \{\ (\textit{attribute\_name}, \texttt{ASC})\ \},\ *\ \rangle$$
$$\langle \texttt{sort},\ \{\ (\textit{attribute\_name}, \texttt{DESC})\ \},\ *\ \rangle$$

By the definition of $\pitchfork$, the above node descriptors will match all nodes within any query plan materializing the query q in which the specified `ORDER BY` clause was specified. Further, the `sort` operator will be present in a query plan *iff* an `ORDER BY` clause is present in the query that the plan materializes.

Further Operators: As a query consisting of multiple `SELECT` statements joined by either a `UNION`, `INTERSECT`, or `MINUS` operator would directly cause the corresponding relational algebra operator to be included in a query plan, the intension represented by any of these operators can trivially be identified by one of the following node descriptors:

$$\langle\ \texttt{union}, *, *\ \rangle$$
$$\langle\ \texttt{intersection}, *, *\ \rangle$$
$$\langle\ \texttt{difference}, *, *\ \rangle$$

In the above cases, we have demonstrated that it is possible to construct a node identifier matching the nodes corresponding to any well-formed clause within a valid SQL query. To complete this proof, we now show how to express constraints on other semantically meaningful intensional regions. Specifically, to identify a specific view or table name (e.g., all queries to the `salary` table), constraint operator (e.g., any use of the function `AVG`), attribute (e.g., any reference to the `age` at-

tribute), or constant (e.g., a particular cutoff threshold) as being of interest, the following node descriptor can be used, where *atom* indicates the item of interest:

$$\langle\ *,\ \{\ (atom)\ \},\ *\ \rangle$$

By the definition of $\Cap$, the above (simple) node descriptor matches exactly the set of all occurrences of *atom* in any query plan materializing the query q.

We have thus demonstrated that for any semantically-meaningful fragment of a query q, it is possible to specify a node descriptor that matches *exactly* the query plan nodes corresponding to this query fragment in *any* query plan Q materializing q. □

With this proof, and given that any single node within a query plan can be matched by some node descriptor, we immediately have the following:

Corollary 2. *Any intensional region $I \subseteq \mathcal{N}$ can be specified using a collection $D \subseteq \mathcal{D}$ of node descriptors, and detected within any query plan $Q \in \mathcal{Q}$ using the $\Cap$ operator.*

Since it is possible to specify a set of node descriptors $D \subseteq \mathcal{D}$ such that the matching construct $\Cap$ identifies the corresponding intensional regions contained within a query q, and constraints can be placed on the location field of any node matching a set of node descriptors D, we trivially have the following:

Corollary 3. *The matching operator $\Cap$ is sufficiently expressive to encode any (I, A)-privacy constraint.*

4.3.3 Constraining Query Execution Plans

In addition to passing node descriptors to a query optimizer in order to specify intensional regions that require special handling, users must also be able to state what sites should or should not be allowed to evaluate nodes in that region. Users can do this by specifying a *condition* on the free variable used in the node descriptor. Given a node n that matches a node descriptor, a condition states what values are allowable for the portion of that node that is matched by free variable in a node descriptor.

Example 3. *Consider nd = $\langle$ *, {`(Polluted_Waters.pollutant)`}, @f $\rangle$ as a node descriptor paired with the condition $cond$ = @f <> `Inventory`.*

For the node $n1$ = ⟨ *join, (*{*(*`Polluted_Waters.pollutant`*,* =*,* `Supplies.name`*), (AND), (*`Mapper.location`*,* =*,* `Plants.location`*)*}*,* `Inventory` ⟩*, the condition would evaluate to false because the free variable in* nd *in the* s *position and the* s *position in* $n1$ *is* `Inventory`*, the exact value that is specified not to be allowable by* $cond$*.*

For the node $n2$ = ⟨ *join, (*{*(*`Polluted_Waters.pollutant`*,* =*,* `Supplies.name`*), (AND), (*`Mapper.location`*,* =*,* `Plants.location`*)*}*,* `Mapper` ⟩*, the condition would evaluate to true because there is a free variable (@f) in* nd *at the* s *position and the value of the* s *position in* $n2$ *is* `Mapper`*, which is allowable by* $cond$*.*

Such pairings of node descriptors and conditions is the underpinning for our group of extensions to SQL, Privacy-Aware SQL (PASQL). We formally refer to this simplest example of node descriptor/constraint pairings as basic PASQL constraints:

Definition 13 (Basic PASQL constraint)**.** *A basic PASQL constraint,* c *is a combination of a node descriptor,* c.nd $\in \mathcal{D}$*, and a condition,* c.cond*.* c.nd *must contain a single free variable as either* $c.nd.op$*,* $c.nd.s$*, or as part of* $c.nd.params$*.* c.cond *must be a comparison that can be evaluated to either true or false consisting of two operands and a comparison operator. The operator can be one of the following:* $=, \neq, \in, \notin$*. One of the operands must be the free variable from c.nd, while the other can be either a literal constant or a set of literal constants.*

Given a basic PASQL constraint, we can evaluate its condition on a node matching its node descriptor using the following function:

Definition 14 (evaluate(n, c))**.** *Given the value v from n that corresponds to the free variable in c.nd,* ***evaluate****(n, c) returns True if substituting v for the free variable in c.cond causes c.cond to be a true statement, and* ***evaluate****(n, c) returns False if it causes it to be a false statement.*

Note that only nodes that match the node descriptor of a basic PASQL constraint can violate that constraint's condition (and hence violate the constraint). Given that we define $\mathcal{C}$ to be the set of all basic PASQL constraints, we define the violation of a constraint by a query plan node according to the function violates : $\mathcal{N} \times \mathcal{C} \rightarrow \mathbb{B}$:

Definition 15 (violates(n, c))**.** *Let* n *be a node in* $\mathcal{N}$ *and* c *be a basic PASQL constraint in* $\mathcal{C}$*,* n violates *c iff* $n \Cap c.nd \wedge \neg$***evaluate***(n, c).

We say query plan qp violates a basic PASQL constraint if any node $n \in qp$ violates that constraint. Any query plan node that does not violate a constraint is considered to support it:

Definition 16 (supports(n, c))**.** *A query plan node* n *supports a constraint* c *so long as that node does not violate* c*:* $\neg$*violates*(n, c)*, or* $n \not\pitchfork c.nd \vee$ *evaluate*(n, c)*, or,* $n \pitchfork c.nd \implies$ *evaluate*(n, c).

Finally, we state that a query plan upholds a constraint so long as every one of its nodes supports it:

Definition 17 (upholds(qp, c))**.** *A query plan* qp *upholds a constraints* c *iff* $\forall n \in qp :$ *supports*(n, c). *This can be equivalently stated as* $\forall n \in qp : \neg$*violates*(n, c)*, or,* $\forall n \in qp : n \pitchfork c.nd \implies$ *evaluate*(n, c)

4.4 SUMMARY

In this chapter, we first formally defined the system model that we assume users to be working within throughout this dissertation. With this in hand, we formally established concepts of intensional privacy, and how the privacy of the intensions of users' queries can be violated. This leads to our formal definition of our novel notion of privacy, (I, A)-privacy. In order to uphold (I, A)-privacy, we presented the concept of a node descriptor, which users can wield in order to identify intensional regions of their query that they consider to be sensitive. We showed how node descriptors can be used to match any query plan operations that are members of the intensional regions that the node descriptors represent, and further proved that node descriptors are powerful enough to identify *any* portion of an SQL query. Finally, we showed how users can author conditions on free variables they include in node descriptors to create (I, A)-privacy constraints and conclude the section by formally defining how a query plan can uphold or violate a user-specified (I, A)-privacy constraint. In the next chapter, we will describe extensions to SQL that allow the expression of these types of constraints.

5.0 PASQL: PRIVACY-AWARE SQL

In this chapter, we describe three different variants of Privacy-Aware SQL (PASQL): PASQL0, PASQL1, and PASQL2. Each provides an increasingly expressive syntax. As we discuss in Section 6.3, these variants represent interesting points in the tradeoff space between the variety of constraints that can be expressed, optimization time complexity, and the optimality of query plans produced.

It should be noted that we define PASQL1 to be a superset of PASQL0, and similarly, PASQL2 to be a superset of PASQL1. Any constraint specification that can be interpreted by an optimizer supporting PASQL0 will can be correctly parsed by an optimizer supporting PASQL1 (just as any PASQL0 or PASQL1 constraint specification will be correctly parsed and optimized by an optimizer supporting PASQL2). Note that constraints unique to PASQL2 do not inherently depend on those of PASQL1 (i.e., it is not necessary for PASQL2 to be a superset of PASQL1). We define PASQL2 as a superset of PASQL1 to simplify our presentation of PASQL variants and establish a continuum of increasing expressiveness.

5.1 PASQL0

PASQL0 allows users to express only basic PASQL constraints. As such, it is the least expressive variant of PASQL presented here. With PASQL0, basic PASQL constraints can be presented to the optimizer as either hard *requirements* that *must* be supported by any query plan generated to evaluate the query or as *preferences* that the user would like to see upheld by resulting query plans but do not need to be upheld by the plan chosen to evaluate the query (e.g., if two constraints

conflict with one another, one of them can be ignored, or if a constraint cannot possibly be upheld, a query plan should still be generated and evaluated).

5.1.1 Requirement Syntax

<rclause> ::= "REQUIRING" <holds>
<holds> ::= <hold> [", AND" <holds>]
<hold> ::= <cons> "HOLDS OVER" <dnode>

<pclause> ::= "PREFERRING" <prefs> [<cascade>]
<cascade> ::= "CASCADE" <prefs> [<cascade>]
<prefs> ::= <pref> | <hold> ["AND" <prefs>]
<pref> ::= <num>"("<f>")"

<cons> ::= <operand> <cop> <operand>
<operand> ::= <fvar> | <literal> | <set>
<fvar> ::= ""<literal>
<cop> ::= "=" | "<>" | "IN" | "NOT IN"
<set> ::= "{" <items> "}"
<items> ::= <literal> | <items>

<dnode> ::= "⟨" <op> "," <param> "," <p> "⟩"
<op> ::= <fvar> | "scan" | "select" | "project"
| "join" | "product" | "rename" | "aggregate"
| "sort" | "deduplicate" | "union" | "intersection"
| "difference" | "*"
<param> ::= "{" <pset> "}" | "*"
<pset> ::= "(" <pitems> ")"
<pitems> ::= <pitem> | <pitems>
<pitem> ::= <pop> | <literal> | <set> | <agg>
| <fvar> | "ASC" | "DESC"
<agg> ::= "MIN" | "MAX" | "AVG" | "SUM" | "COUNT"
<pop> ::= "=" | "<>" | "<" | ">" | "<=" | ">="
| "IN" | "NOT IN" | "BETWEEN" | "LIKE"
| "IS NULL" | "IS NOT NULL" | "AND" | "OR"
<p> ::= <literal> | <set> | <fvar> | "*"

Figure 3: PASQL0 REQUIRING and PREFERRING clause syntax.

In order for users to be able to express constraints that must be upheld by a plan to evaluate a query, we propose a REQUIRING clause (REQUIRING <condition> HOLDS OVER <node de-

scriptor>) as an extension to SQL. We propose this clause for use in two locations: first as an addition to the SQL `SELECT` statement, and also as an addition to SQL's set operators (`UNION`, `INTERSECT`, and `MINUS`). If it is used in conjunction with a set operator, the requirements expressed by this clause will apply to both the operator itself, and the two `SELECT` statements that it combines. If it is used at the end of a `SELECT` statement, its requirements apply only to that `SELECT` statement. The full syntax for the `REQUIRING` clause is defined in Figure 3.

This clause explicitly lists the two portions of a basic PASQL constraint, and in parsing the query, an optimizer supporting PASQL can simply construct basic PASQL constraints directly from the node descriptor/condition pairs specified in the `REQUIRING` clause.

Example 4. *Alice can use a `REQUIRING` clause to write her query from Figure 2 to ensure that any operations on `Pollution_Watch`'s `pollutant` attribute are not evaluated by the `Inventory` server as follows:*

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

Such a query upholds (I, A)-privacy for Alice by keeping an intensional region consisting of all nodes operating on `Pollution_Watch`'s `pollutant` attribute from being evaluated by the `Inventory` database server. Note that the query plan shown in Figure 2 does not adhere to this requirement because the `Inventory` server evaluates the root join which contains a condition on `Polluted_Waters.pollutant`. Hence, it would not be an acceptable query plan for this new, constrained version of Alice's query.

Now, this notion of hard requirements for basic PASQL constraints may not always be in the best interest of the user. What a user considers private and the importance maintaining the privacy of different aspects of a query can vary widely, not only from user to user but also from query to

query. To accommodate this, PASQL additionally allows users to specify constraints as *preferences* that a PASQL supporting optimizer will attempt to uphold during optimization, will not necessarily uphold.

5.1.2 Preference Background

In [42, 41], the authors develop a formalism for expressing preferences over the tuples returned by a relational database query. Rather than requiring that an SQL selection specify an *exact match* criteria, the preference SQL described in [42, 41] allows the user to specify a partially-ordered preference structure over the tuples returned. This is particularly helpful when exact match criteria cannot be found. Formally, the authors of [41] define a preference as follows:

Definition 18 (Preference $P = (R, <_P)$)**.** *Given a set R of relational attribute names, a* preference P *is a strict partial order $P = (R, <_P)$, where $<_P \subseteq domain(R) \times domain(R)$.*

Given this definition, "$x <_P y$" is interpreted as "I like y better than x." For example, a user querying a database for the cheapest car could express her preference for tuples with the lowest value for the price attribute as: LOWEST(price), where LOWEST is a base preference defined such that $x <_P y$ *iff* x.price $> y$.price and y.price is as low as possible. Using this base preference constructor, tuple t will be preferred to tuple t' *iff* t represents a lower cost car than tuple t'. We refer the reader to [41] for descriptions of a range of other numeric and non-numeric base preference constructors.

Similarly, [41] defines *complex preferences* through the use of complex preference constructors. For example, two preferences that are equally preferred can be combined through the use of a *Pareto* preference constructor. Given two preferences $P1$ and $P2$ over attributes $A1$ and $A2$, respectively, such a preference is defined for two items x and y containing attributes from both $A1$ and $A2$ as:

$$x <_{P1 \otimes P2} y \textit{ iff } (x_1 <_{P1} y_1 \wedge (x_2 <_{P2} y_2 \vee x_2 = y_2)) \vee$$
$$(x_2 <_{P2} y_2 \wedge (x_1 <_{P1} y_1 \vee x_1 = y_1))$$

Complex preferences in which one preference is strictly more important than the other can be defined using the *prioritized preference* operator, &. For the definitions of other complex preference constructors, see [41]. As a part of PASQL0, we augment this notion of preferences to function over the makeup of query *plans* as opposed to query *results*.

5.1.3 Preference Syntax

In order for users to express preferred constraints, we now describe the `PREFERRING` clause—modeled after the extensions proposed in [42]—that applies to both `SELECT` statements and set operators, following a `REQUIRING` clause if both are used in a given query. Similar to the `REQUIRING` clause, the `PREFERRING` clause allows users to express constraints over the makeup of plans that evaluate their queries. Preferred constraints however, need not all be upheld for a plan to be used to evaluate a query. Plans that uphold more preferred constraints, or that uphold preferred constraints that are more important to users should be used to evaluate their queries over plans that uphold fewer or less important preferred constraints.

We use of the keyword `AND` to represent the complex preference constructor ⊗ and `CASCADE` to represent the complex preference constructor &. For each basic PASQL constraint listed in a `PREFERRING` clause, a plan is considered more preferred if it violates that constraint fewer times. Consider the following `PREFERRING` clause:

```
PREFERRING @p <> Pollution_Watch HOLDS OVER
        < *, {(Waterway_Maps.name)}, @p >                (1)
    CASCADE @p <> Facilities HOLDS OVER
        < *, {(Polluted\_Waters.name)}, @p >             (2)
        AND @p <> Inventory HOLDS OVER
        < *, {(Polluted\_Waters.name)}, @p >             (3)
```

In this example, constraint (1) is stated to be more highly preferred than constraints (2) and (3). Further, constraints (2) and (3) are stated to be equally preferred. Given this specification, a PASQL query optimizer would be able to construct the lattice shown in Figure 4 to rank query plans according to the subset of these three constraints they support. A plan that supports all three constraints is considered to be more highly preferred than a plan that does not, and a plan

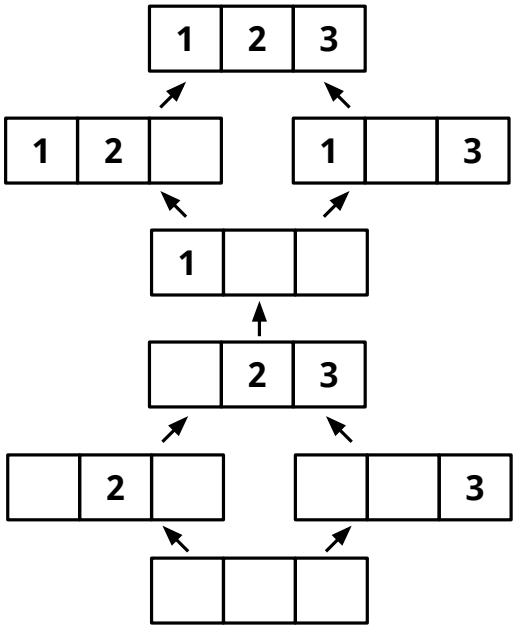


Figure 4: A lattice to rank query plan by preference adherence.

that supports constraint (1) is considered to be more highly preferred than any that do not. We formalize our notion of relative preference between query plans in Section 5.1.4.

The full syntax of the `PREFERRING` clause is presented in Figure 3. To demonstrate the use of this syntax, let us consider the following example the combines strict requirements on a query plan with more flexible preferences:

Example 5. *Assume that Alice does not want her query to be evaluated if any nodes whose operations make use of* `Polluted_Water`*'s* `pollutant` *attribute are evaluated at the* `Inventory` *server. Further, she* prefers *to execute joins on the* `Inventory` *server, and she* prefers *that no nodes operating on* `Supplies'` `name` *attribute are evaluated by the* `Inventory` *server. She has a stronger preference for executing joins on the* `Inventory` *server than for keeping* `name` *from the* `Inventory` *server. Alice could express this combination of preferences using both the* `REQUIRING` *and* `PREFERRING` *clauses:*

```
SELECT * FROM Plants, Supplies, Polluted_Waters
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
REQUIRING @p <> Inventory HOLDS OVER
        < *, {(Polluted_Waters.pollutant)}, @p >
PREFERRING @p = Inventory HOLDS OVER
        < join, *, @p >
    CASCADE @p <> Inventory HOLDS OVER
        < *, {(Supplies.name)}, @p >;
```

In this example, there is clear conflict in Alice's constraint specifications. As it stands, a PASQL query optimizer would be forced to ignore Alice's preference to perform joins at the `Inventory` server in order to ensure that her requirement that operations that make use of `Polluted_Waters.pollutant` are kept from the `Inventory` server (note that Alice's second preference could still be supported, however). If this query were issued sans the `REQUIRING` clause (i.e., with only the `PREFERRING` clause listed above), the optimizer should choose to perform all joins at the `Inventory` server at the cost of keeping operations on `Supplies.name` from the `Inventory` server.

5.1.4 Preference Adherence

Here, we formally define the notion of relative preference between two query plans. We first need to be able to determine how many times a query plan qp violates a single constraint c using the function single_counts : $\mathcal{Q} \times \mathcal{C} \rightarrow \mathbb{Z}$:

Definition 19 (single_counts(qp, c))**.** *This function returns the following result:*

$\sum_{n \in qp}$ *1 if* violates(n, c) *or 0 if* supports(n, c)

Users can specify queries with many attached constraints, and hence we need to consider all constraints specified for a given query when investigating the relative preference of two query plans. A constraint specification C is a total order of m lists of equally preferred constraints $(C_1, C_2, ..., C_m)$. We refer to constraints in the same list as being part of the same *preference level*. All preferred constraints listed before the first `CASCADE` in a query are considered to be most important and make up list C_1. All constraints between the first and second `CASCADE`s make up list C_2, and so on with all constraints listed after the last `CASCADE` making up list C_m. A constraint in any given preference level i is just as important as all other constraints of list C_i, less important than constraints in lists preceding i (i.e., in C_j where $j < i$) and more important than constraints of lists following i (i.e., in C_k where $k > i$). Note that, overall, C describes a partial order of all of the constraints attached to a query.

Now, in comparing the relative preference of two query plans, since constraints at a given preference level are equally important to the overall preference of a plan, we account for how many times constraints in a given preference level are violated by that plan with the function $\mathsf{level\_counts} : \mathcal{Q} \times 2^{\mathcal{C}} \rightarrow \mathbb{Z}$:

Definition 20 ($\mathsf{level\_counts}(qp, C_i)$)**.** *This function returns the following result:*

$$\sum_{c \in C_i} \mathit{single\_counts}(qp, c)$$

Hence, we consider a query plan qp_1 to be more preferred than another query plan qp_2 if qp_1 contains fewer violations of more important preferences than qp_2. We formally define this notion of preference using the function $\mathsf{preferred} : \mathcal{Q} \times \mathcal{Q} \times 2^{2^{\mathcal{C}}} \rightarrow \mathbb{B}$:

Definition 21 ($\mathsf{preferred}(qp_1, qp_2, \mathrm{C})$)**.** *$qp_1$ is more preferred than qp_2 iff:*

$$\exists C_i \in C : \mathit{level\_counts}(qp_1, C_i) < \mathit{level\_counts}(qp_2, C_i)$$
$$\wedge\, (\forall C_j \in C : C_i \neq C_j \implies (j > i \vee \mathit{level\_counts}(qp_1, C_j) = \mathit{level\_counts}(qp_2, C_j)))$$

5.1.5 Subsumption of PIR

In this section, we show that an important subset of (I, A)-privacy constraints can be easily expressed using PASQL0: PIR constraints. By constructing node descriptors that define an intensional region as all nodes which contain some part of the selection criteria of a query, users can constrain this intensional region to ensure that no database servers gain intensional knowledge of the selection criteria of their queries, and hence use PASQL0 to express PIR privacy constraints on their queries. This notion is more precisely formalized below.

Theorem 4. *Let the inference procedure $\models$ be defined using the containment relation $\sqsupseteq$. In this case, PASQL0 can be used to express* any *private information retrieval (PIR) constraint.*

Proof. (By construction.) Without loss of generality, consider a user Alice issuing the following query, q, to an untrusted server S using PIR techniques:

```
q: SELECT * FROM t WHERE attr = const;
```

In executing this query, the server S should *not* be able to learn any part of Alice's private selection criteria (`attr = const`). We now show that this is equivalent to the following (I, A)-privacy aware SQL query q':

```
q': SELECT * FROM t WHERE attr = const
    REQUIRING @p != S HOLDS OVER <*, {(attr)}, @p>,
        AND @p != S HOLDS OVER <*, {(=)}, @p>,
        AND @p != S HOLDS OVER <*, {(const)}, @p>;
```

The above query specifies the following set of node descriptors:

$$D = \{\langle *, \{(\texttt{attr})\}, \texttt{@p}\rangle, \langle *, \{(=)\}, \texttt{@p}\rangle, \langle *, \{(\texttt{const})\}, \texttt{@p}\rangle\}$$

Given any query plan $Q = (N, E)$ materializing the query q', Theorem 1 tells us that the set $I = \{n \mid n \in Q \wedge \exists d \in D : d \Cap n\}$ contains exactly the set of nodes in Q defining the intensional region within Alice's query that she considers sensitive. Further, the condition `p != S` ensures that the server S does not learn any nodes in $I \subset N$, thereby enforcing the PIR constraint on

Alice's selection criteria. Now, by Def. 9, we have that $\kappa_S(Q) \not\supseteq I$, and thus q' preserves (I, A)-privacy (by Def. 8). Thus, (I, A)-privacy can be used to encode the above (and therefore any) PIR constraint. □

5.2 PASQL1 - CONSTRAINING MULTIPLE NODE DESCRIPTORS

To expand upon the expressiveness presented by PASQL0, PASQL1 allows users to author constraints with conditions that compare two different free variables from two different node descriptors. We refer to such constraints as a complex PASQL constraint:

Definition 22 (complex PASQL constraint)**.** *A* complex PASQL constraint*,* c *is a combination of two node descriptors,* c.nd1 $\in \mathcal{D}$ *and* c.nd2 $\in \mathcal{D}$*, and a condition,* c.cond*.* c.nd1 *and* c.nd2 *must each contain a single different free variable as either* op*,* s*, or as part of their respective* $params$*.* c.cond *must be a comparison that can be evaluated to either true or false consisting of two operands and and a comparison operator. The operator can be one of the following:* $=, \neq, \in, \notin$*, while the operands must be the free variables from* c.nd1 *and* c.nd2*.*

Note that supporting complex PASQL constraints would require the syntax presented in Figure 3 to be amended to allow for multiple node descriptors to be specified.

We allow free variables to be included in the specification of a node descriptor so that constraints can be placed on the values of those variables. In the case that a constraint is written over the free variables in a single node descriptor, ensuring that such a condition holds over a given query tree is relatively simple: for each node in the query tree that matches the node descriptor, ensure that the condition holds. In the case that multiple node descriptors are used, however, ensuring that a condition holds over a query plan is slightly more complicated, as it must be ensured that the condition holds for all combinations of nodes that match the independent descriptors. This increase in expressiveness allows users to author new types of policies such as separation of duty (e.g., any site that performs a scan operation should not also perform a join operation).

5.3 PASQL2 - PREFERENCES FOR QUERY PLAN EXECUTION

We propose PASQL2 in order to show how the `PREFERRING` clause could be used to allow users to balance their privacy with the performance of their queries. We have already established the relative preference of two plans according to basic PASQL constraints (and implicitly complex PASQL constraints): a plan that violates more important preferred constraints fewer times is more preferred. To account for query performance, users must be able to construct constraints using optimizer estimations about their query plans (e.g., runtime, network traffic generated), and also preference constructors to define how query plans should be ranked given the respective results of those optimizer estimation functions (e.g., a LOWEST preference constructor would state that lower values of a optimizer estimation function would be preferable to higher values). If we consider $\mathcal{F}$ to be the set of all optimizer estimation functions, and $\mathcal{PC}$ to be the set of all numerical preference constructors, we can define *PASQL performance constraints* as follows:

Definition 23 (PASQL performance constraint)**.** *A* PASQL performance constraint*,* c *is a combination of a preference constructor* $c.pc \in \mathcal{PC}$ *and optimizer estimation function* $c.f \in \mathcal{F}$*.*

Example 6. *Assume that Alice would like to keep all operations on the* `name` *attribute of the* `Supplies` *table from being evaluated by the* `Inventory` *server so long as doing so does not cause her query to take any longer to run. Alice can succinctly state this preference using PASQL2 as follows:*

```
SELECT * FROM Plants, Supplies, Polluted_Waters
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
PREFERRING LOWEST(runtime)
    CASCADE @p <> Inventory HOLDS OVER
        < *, {(Supplies.name)}, @p >;
```

As can be seen, PASQL performance constraints allow users to explicitly balance the privacy and performance of their queries. Users can specify different levels of preferences to establish a performance penalty that they are willing to pay to provide certain privacy protections. In the previous example, we assume that Alice would not be willing to sacrifice *any* performance to keep `Supplies.name` from being evaluated by the `Inventory` server. However, PASQL performance constraints do not limit users to only this all-or-nothing approach. Using PASQL performance constraints, users establish a sliding scale of performance overheads that they are willing to pay to uphold their (I, A)-privacy constraints.

Example 7. *Assume that Alice would like to keep all operations on the* `name` *attribute of the* `Supplies` *table from being evaluated by the* `Inventory` *server and is willing to pay a cost to performance such that her query could take up to two minutes to run so that this constraint can be upheld. Alice can succinctly state this using PASQL2 as follows:*

```
SELECT * FROM Plants, Supplies, Polluted_Waters
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
PREFERRING LESSTHAN(runtime, 2)
    CASCADE @p <> Inventory HOLDS OVER
        < *, {(Supplies.name)}, @p >;
```

A tradeoff between privacy and performance is to be expected. Extra care must be taken to protect the privacy of users' query intension during distributed query evaluation. Through the use of PASQL performance constraints, users can directly manage this tradeoff and weight the performance value of each of their privacy preferences.

5.4 EXPRESSIVE CAPABILITIES OF PASQL

The examples that we have presented in this dissertation so far have served mostly explanatory purposes, illustrating the mechanics of (I, A)-privacy and further motivating the need for the protections that it offers. However, the preferences model described in this dissertation is capable of expressing much more powerful controls over the execution of user queries than have so far been demonstrated. This section will demonstrate a range of common policy idioms that can be expressed within the privacy and execution preference framework developed in this dissertation.

5.4.1 Discretionary Access Control (DAC)

In the access control literature, DAC policies allow users to explicitly list the identities of the other users permitted to access their files [57]. The notion of DAC policies has natural applications to user privacy in distributed query execution, as users might wish to white- or black-list individual servers from learning about their queries. In fact, all of the examples presented in previous sections of this dissertation have encoded very specific DAC policies restricting access to intensional regions. Users can also require that certain intensional regions be executed by a specific remote server:

```
REQUIRING @p = Inventory HOLDS OVER <*, {("solvent")}, @p>
```

The above requires that any nodes matching the specified node descriptor be executed by the Inventory server. It is also possible to allow any remote server explicitly identified as belonging to some *set* of trusted servers to handle a particular intensional region:

```
REQUIRING @p IN {P, Q, R} HOLDS OVER <*, {("solvent")}, @p>
```

This constraint would force all matching query nodes be evaluated by some server in the set $\{P, Q, R\}$ of trusted servers.

5.4.2 Mandatory Access Control (MAC)

In contrast to DAC systems, MAC systems rely on a centrally-defined security policy that cannot be overridden. For instance, the Bell-LaPadula model [9] is a MAC system that enforces access

controls based on centrally-managed security clearances: e.g., users can read any file with a security level lower than their security clearance, but cannot read documents with a higher security level. To enforce MAC constraints, the client software from which queries are issued could automatically apply `REQUIRING` clauses to all outgoing queries. For ease of use in such cases, we could allow the use of macros to define collections of principals (denoted here by the prefix "#") which could be parsed and replaced with a static list of principals by the trusted query processor as a first step in parsing.

To illustrate this point, consider an intelligence analyst using a top-secret clearance workstation looking over field agent reports concerning a certain date (say, 01-01-10). To ensure proper compartmentalization of the data from those reports, the query issuing client software on that workstation could ensure that all queries sent out are sent only to servers cleared to handle top-secret data with the following `REQUIRING` clause:

```
REQUIRING @p IN #top-secret-clearance HOLDS OVER
    <*,{("01-01-10")},@p>
```

Note that the framework articulated in this dissertation can allow MAC and DAC constraints to co-exist, as is often the case in environments using of MAC constraints [9].

5.4.3 Attribute-based Access Control (ABAC)

ABAC policies allow access decisions to be made based upon the attributes of principals in the system, rather than their identities [78]. The macro mechanism described to support MAC policies could be leveraged by users—rather than the query issuing client—to enforce ABAC policies. For instance, Alice could require that any operation on the `Salary` table only be visible by servers run by the finance group:

```
REQUIRING @p IN #finance-group-servers HOLDS OVER
    <*,{("Salary")},@p>
```

In addition to supporting static, user-defined macros to encode server attributes, an interesting avenue of future work would be enabling support for dynamic macros to be built based upon unforgeable digital attribute credentials stored in a server's meta-data catalog. This would allow

for more flexible ABAC support in which users can rely on the attestations of trusted certifiers to make attribute-based judgments regarding a server's characteristics.

5.4.4 Separation of Duty

Separation of duty (SoD) policies are used to require that multiple principals cooperate to carry out a particular action [12]. In the context of distributed database systems, one could use SoD to explicitly limit information flow when querying multiple tables replicated across a collection of servers by forcing each table scan to be performed by a different server. This is easily expressed in PASQL1 through the use of a single condition over multiple node descriptors:

```
REQUIRING @p != @q HOLDS OVER <scan,{("Salary")},@p>,
    <scan,{("EmploymentHistory")},@q>
```

The above example would force the scans of the `Salary` and `EmploymentHistory` tables to occur at different sites.

5.4.5 Data Source Preference

In addition to privacy-related constraints, the preference model developed in this dissertation can also be used to enforce other execution preferences during query evaluation. For instance, users can specify preferences over the sources used to obtain replicated data:

```
PREFERRING @p = Q HOLDS OVER <scan,{(A)},@p>
    CASCADE @p != R HOLDS OVER <scan,{(A)},@p>
```

This preference says that a user would prefer to get table `A` from the server `P`. If this fails, the table could be retrieved from any replica other than `R`. Such preferences would clearly benefit users who, even though a table is available from multiple sources, wish it to be acquired from a given source. This type of preference could arise, e.g., due to differing consistency guarantees offered by various sources. For instance, in the above, we could imagine `P` being the primary copy of a relational table, and `R` being an eventually consistent—and thus potentially out of date—replica.

This example highlights how PASQL can be used not only as a tool for protecting querier privacy, but also as a tool for use by database administrators to easily tune system performance.

5.5 SUMMARY

In this chapter, we presented the syntax and specifications of the three variants of PASQL: PASQL0, PASQL1, and PASQL2. In describing PASQL0, we overviewed the concepts of preferences as used in the database literature, and how we adapted this formalization of preferences to the specification of constraints on the makeup of query evaluation plans. Through this formalization, users can flexibly specify constraints that they would like to see upheld by plans to evaluate their queries but whose support is not necessary for the query to be evaluated. We further proved that PASQL0 is capable of subsuming private information retrieval (PIR): (I, A)-privacy constraints can be used to encode PIR constraints. With PASQL1, we presented a method for users to author conditions over the relative values of two different free variables, allowing for the creation of separation of duty contraints. With PASQL2, we provided a way for users to explicitly balance the privacy and performance of their queries. Following our descriptions of these three variants of PASQL, we presented several example policies that can be encoded using PASQL. In the next chapter, we will discuss how a PASQL query optimizer can enforce PASQL constraints during optimization.

6.0 ENFORCING (I, A)-privacy

We have so far focused on defining a syntax and semantics for (I, A)-privacy, and on describing how these types of constraints—and partially-ordered preference structures over many such constraints—can be modeled and specified using PASQL. We now turn our attention to the enforcement of (I, A)-privacy constraints. First, we prove that (I, A)-privacy constraints can be enforced directly by users without explicit server-side support. We next explore the possibilities available in implementing an optimizer for queries with attached PASQL constraints. We first present a straw-man post-processing approach to illustrate the need to account for both querier privacy and performance *during* query optimization. With this in mind, we examine the space of tradeoffs to be considered in implementing such an optimizer to support (I, A)-privacy constraints. Finally, we conclude this chapter by presenting an algorithm to find the best performing of the most highly preferred query plans and analyzing its time and space complexity.

6.1 LOCAL ENFORCEMENT OF (I,A)-PRIVACY CONSTRAINTS

The constructions discussed in Chapter 5 certainly make it possible to specify constraints that cannot be enforced in practice. For instance, one could require a table be retrieved from a principal that does not maintain a copy of that table. We now discuss how to identify sets of (I, A)-privacy constraints that can be enforced, and then show that it is always possible for the querier to enforce these constraints without requiring any server-side support beyond the ability to process standard SQL queries. To be enforceable, this set of constraints (1) should not preclude the retrieval of any table required to materialize the query, and (2) must not contain any conflicting constraints.

Definition 24 (Source Possible)**.** *Let* $\mathsf{tables}(q)$ *denote the set of relations and views required to materialize the results of an SQL query* q*. We say that it is* source possible *to uphold a set of* (I, A)*-privacy constraints* C *for a query* q *iff* $\forall_{t \in \mathsf{tables}(q)} \exists_{s \in \mathcal{S}}$: (s *maintains a copy of* t) $\wedge$ $(\langle scan, \{(t)\}, * \rangle, -, \{\ldots, s, \ldots\}\rangle \notin C)$.

Definition 25 (Conflict Free)**.** *Without loss of generality, consider two* (I, A)*-privacy constraints* $c_1 = \langle I_1, \Diamond_1, A_1 \rangle$ *and* $c_2 = \langle I_2, \Diamond_2, A_2 \rangle\}$*. We say that* c_1 *and* c_2 *are conflicting iff* $(I_1 \cap I_2 \neq \emptyset) \wedge (A_1 \cap A_2 \neq \emptyset) \wedge (\Diamond_1 \neq \Diamond_2)$*. We say that a set* C *of* (I, A)*-privacy constraints is* conflict free *iff* $\forall_{\{c_j, c_k\} \subseteq C, c_j \neq c_k}$: c_j *and* c_k *do not conflict.*

The notion of source possibility ensures that each relation or view necessary for materializing a query q can be obtained from at least one server in the network without violating any constraint in C. On the other hand, conflict freeness ensures that no logical contradictions requiring a given intensional region to be both *visible to* and *hidden from* the same server occur within the specified constraint set.

Definition 26 (Enforceable)**.** *We say that a set* C *of* (I, A)*-privacy constraints is* enforceable *for a query* q *iff* C *is source possible for* q *and* C *is conflict free.*

(I, A)-privacy is a highly personalized notion of privacy: users are free to define their own notions of privacy and can explicitly balance various privacy and performance trade-offs. As such, it is interesting to note that any enforceable set of (I, A)-privacy constraints can be *locally enforced* by the querier via intelligent query planning. Informally, local enforceability implies that the querier can ensure that her (I, A)-privacy constraints are enforced without any server-side support beyond that required by the SQL standard. As such, (I, A)-privacy constraints can be enforced by queriers when interacting with *any* distributed database system. This is in contrast to existing notions of querier privacy—e.g., PIR ([14, 47, 80, 54, 59]) —which require non-standard, server-side cryptographic support. We now formalize this notion of local enforceability and prove that all enforceable (I, A)-privacy constraints are locally enforceable.

Proposition 1 (Local Enforceability)**.** *Let us use the set* $\mathcal{R} = \{\sigma, \pi, \rho, \cup, \setminus, \times\}$ *to denote the six primitive operations comprising the relational algebra, and* $\mathsf{servers}(Q)$ *to denote the set of server nodes participating in the execution of a query plan* Q*. Given an SQL query* q *and a set* C *of enforceable* (I, A)*-privacy constraints, there exists a query plan* $Q_C = \langle N, E \rangle$ *for* q *that upholds*

all constraints in C *in which* $\forall_{s \in \mathsf{servers}(Q_C)} \forall_{\langle op, param, s \rangle \in N} : op \in \mathcal{R}$*. As such, we say that any enforceable set of* (I, A)*-privacy constraints is* locally enforceable.

Proof. (*Sketch, by construction.*) Given a query q and a set of (I, A)-privacy constraints C, the querier can trivially construct a plan Q_C that upholds all constraints in C by using a pure data shipping approach [44]. Specifically, each table $t \in \mathsf{tables}(q)$ can be retrieved in its entirety by the querier, and all operations on these base relations can be executed locally by the querier. □

Although the construction above is likely to be incredibly inefficient in practice, it is sufficient for proving the local enforceability of (I, A)-privacy constraints. To improve performance beyond this baseline, interior operations in the query plan tree must be assigned to be evaluated at sites sites other than the querier. Plans must be explored that allow partial query processing to occur on servers that host the relations needed to resolve a query so that intermediate results of the query can be shipped between sites instead of full base relations. Clearly, this approach to PASQL query optimization would require great care in performing site assignments to ensure that (I, A)-privacy constraints are not violated.

6.2 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 resulting plan to enforce user constraints. This general approach is used in [18] 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 (I, A)-privacy 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.

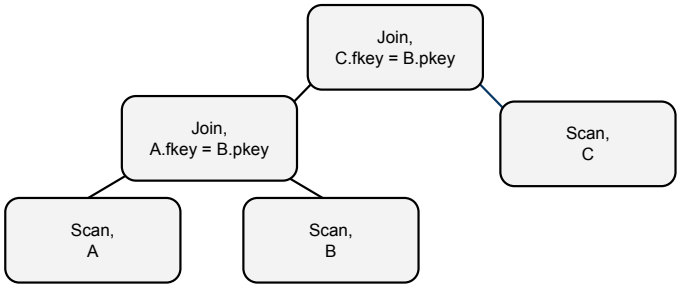


Figure 5: Traditional optimizer result for the example query from Section 6.2.

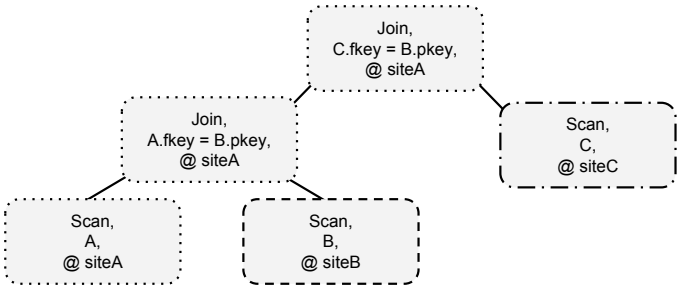


Figure 6: Inefficient site-assigned plan for the example query from Section 6.2.

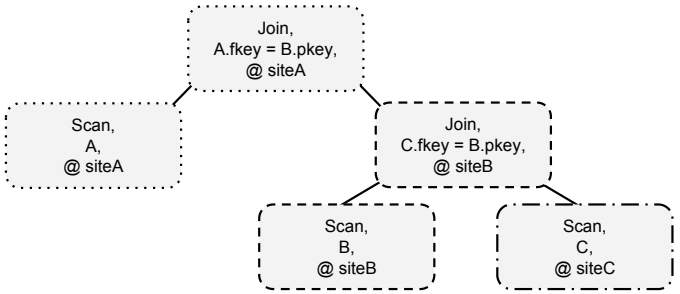


Figure 7: Efficient site-assigned plan for the example query from Section 6.2.

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 Figure 6.2. 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 Figure 6.2), 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 at further cost to evaluation time.

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 (assumptions on scan and join operation times were gleaned from the experimental setups described in Section 7.4). 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 Figure 6.2 would take 711.66576 seconds to execute. Without considering constraints, the general plan from Figure 6.2 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 Figure 6.2. By joining `B` and `C` first at `siteB`, the large network transfer shown in Figure 6.2 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 in a query plan.

To avoid the shortcomings of two-phase optimization discussed above, we must take a synergistic approach to query optimization, accounting for both (I, A)-privacy constraints and query performance during the query optimization process.

6.3 TRADEOFFS IN CONSTRAINT ENFORCEMENT

We consider the tradeoffs present in optimizing for both privacy and performance. In order to support (I, A)-privacy constraints, constraint adherence must supplant estimated query performance as the primary optimization metric for distributed query optimization. This does not make performance irrelevant, however. Users will still wish to have low cost plans to evaluate their queries, we have just shifted the goal of query optimization to be finding the fastest of the most highly preferred plans for evaluating a user's query that violates no required constraints. As such, query optimization must occur over two metrics, both *constraint adherence* and *performance*. This shift leads to the development of an interesting tradeoff space between *optimization time*, *quality of plans* produced, and the *expressiveness* of the constraint language.

One interesting point in this tradeoff space is the sacrifice of query optimization time to maximize constraint expressiveness and plan quality. Dynamic programming cannot be used to produce optimal plans for queries supporting PASQL1 or PASQL2 as the use of such constraints breaks the optimal substructure properties of SQL query optimization, a requirement for a dynamic programming approach to query optimization. Consider the constraint that any site that performs a scan should not also preform a join. This constraint can be easily expressed in PASQL1. It is trivial

to imagine a case where the optimal plan to evaluate a query with such a constraint attached is built using a seemingly suboptimal 2-way join plan (i.e., a 2-way join plan that pulls data from a slower replica). PASQL2 exclusive constraints can also break optimal substructure. Consider the following `PREFERRING` clause:

```
PREFERRING LESSTHAN(runtime, 2min)
    CASCADE @p = S1 HOLDS OVER < scan, {(T1)}, @p >
```

If *all* plans that are leafed by scans `T1` at site `S1` will take more than two minutes to run, an optimizer could unknowingly prune all of subplans that form the basis of an overall optimal plan because they are dominated by plans performing scans of `T1` at `S1`. Because of this, we do not have an optimal domination metric for PASQL1 and PASQL2 optimization and must take a brute force approach to optimizing these queries.

By sacrificing expressiveness, however, we can show that optimal plans can be produced without having to resort to brute force optimization. In Section 6.5, we prove that using the algorithm presented in Section 6.4 (an extension of the Classic algorithm), optimal substructure can be maintained throughout the optimization of PASQL0 queries.

The final notable point in this tradeoff space involves minimizing the overhead to query optimization while allowing queries to be expressed using PASQL1 and PASQL2 by sometimes producing suboptimal plans. This is the approach that we have taken in implementing PASQL1 support in PAQO [29]. We sacrifice the guarantee that optimal plans will be produced to maintain constraint language expressiveness and optimization performance. Note that PAQO supports only PASQL1 not because PASQL2 support is infeasible, but merely because we leave the implementation of PASQL2 support to future work.

6.4 ALGORITHM FOR OPTIMAL CONSTRAINT SUPPORT

Before detailing our algorithm for producing the best performing of the most highly preferred query plans for evaluating a user's query, we must highlight an important case to be considered

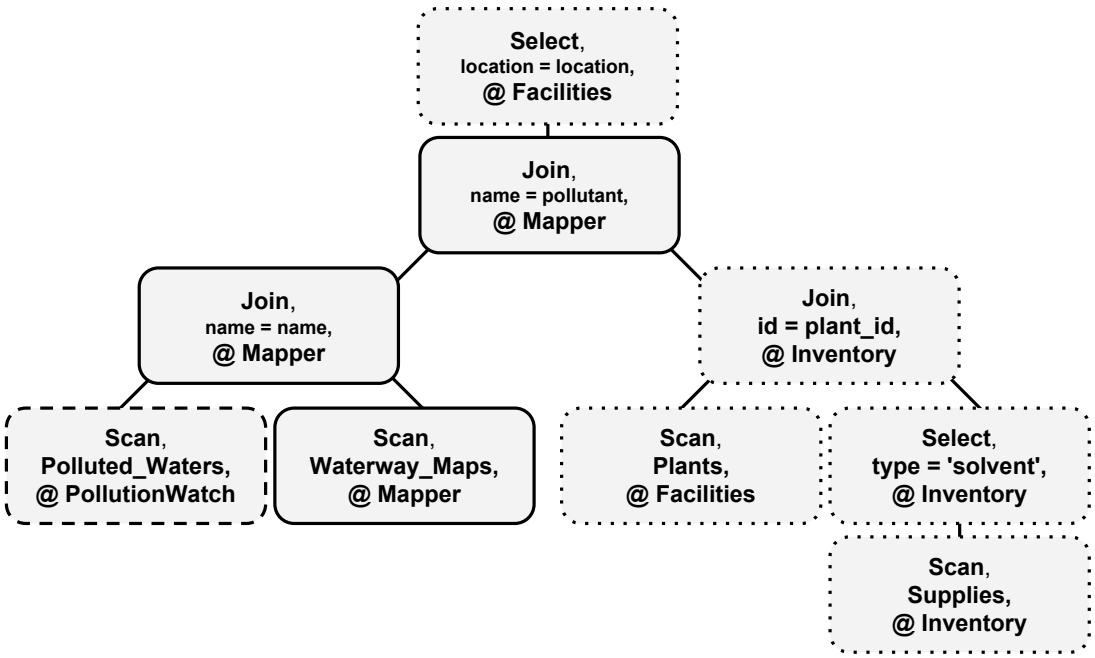


Figure 8: An example plan to evaluate Alice's query while respecting a specific set of constraints.

in optimizing queries with attached PASQL constraints. Consider Alice's example query with the following two constraints:

```
SELECT * FROM Plants, Supplies, Polluted_Waters
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
REQUIRING @p = Mapper HOLDS OVER
        < *, {(Polluted_Waters.pollutant)}, @p >,
    AND @p = Facilities HOLDS OVER
        < *, {(Plants.location)}, @p >;
```

Given these constraints, the plan shown in Figure 2 cannot be produced as a result of optimization as each of its two join conditions (`Polluted_Waters.pollutant` = `Supplies.name`, and `Waterway_Maps.location` = `Plants.location`) are required to be evaluated by different sites (`Mapper` and `Facilities`, respectively). In fact, if we assume the plan shown in Figure 8 to be the optimal plan to evaluate this query, it becomes clear that in optimizing queries with attached PASQL constraints, we can no longer leverage a long-standing heuristic for query optimization. Pushing selection/join conditions down as close to the base relation scans in a query plan weeds out unwanted tuples early on in evaluation, avoiding unnecessary processing of those tuples and resulting in faster query execution time. In accounting for constraint adherence as the primary optimization metric, however, this example demonstrates that this approach is no longer sound. Note that a query optimizer following this push-down heuristic could still produce a query plan that supported both of the specified constraints by producing a query plan with a different join ordering. However, if we consider the join ordering used by the plan shown in Figure 8 to be the best performing, plans with a different join ordering would most likely be suboptimal as they would not be the fastest of the most highly-preferred plans for evaluating Alice's query.

Algorithm 5 presents a modified version of the classic dynamic programming-based query optimization algorithm (discussed in Chapter 2) that includes user-specified PASQL constraints in the optimization process and also accounts for the use of multiple, distributed evaluation sites in constructing query plans. The set of usable evaluation sites within the distributed system is taken as input to the optimization process while constraints are, of course, specified as part of the query using PASQL. In addition to being needed to prune out unusable access paths (line 4 of Algorithm 5), required constraints are needed by the JOINPLANS function (line 15), while preferred constraints are needed by the PRUNEPLANS function (lines 17 and 18).

Note that the changes to the Classic algorithm shown in Algorithm 5 do not in and of themselves ensure that optimal query plans will be produced (in fact, Algorithm 5 is also used as part of PAQO's heuristic optimization algorithm). These changes simply ensure that the functions called throughout it (ACCESSPLANS, JOINPLANS, and PRUNEPLANS) have access to the constraints specified with the query and the list of possible evaluation sites. To ensure production of optimal plans, each of these functions must be modified, and an additional function must be

Algorithm 5 Dynamic programming algorithm that accounts for PASQL constriants.

Require: SPJ query q on relations $R_1..R_n$, with selection/projection/join conditions $C_1..C_m$, required constraints $REQ_1..REQ_r$, and preferred constraints $PREF_1..PREF_p$
Require: Possible evaluation sites $S_1..S_s$
1: $subplans \leftarrow$ EMPTY_LIST
2: $subplans[1]$.add(EMPTY_LIST)
3: **for** $i = 1$ to n do **do**
4: $optPlan[\{R_i\}] \leftarrow$ ACCESSPLANS$(R_i, \{REQ_1..REQ_r\})$
5: PRUNEPLANS$(optPlan[\{R_i\}])$
6: **for** $i = 2$ to n **do**
7: **for** all $P \subset \{R_1..R_n\}$ such that $|P| = i$ do **do**
8: $optPlan[S] \leftarrow \emptyset$
9: **for** all $O \subset P$ do **do**
10: $l \leftarrow optPlan[O]$
11: $r \leftarrow optPlan[P/O]$
12: $rs \leftarrow \{REQ_1..REQ_r\}$
13: $cs \leftarrow \{C_1..C_m\}$
14: $ss \leftarrow \{S_1..S_s\}$
15: $jps \leftarrow$ JOINPLANS(l, r, rs, cs, ss)
16: $optPlan[P] \leftarrow optPlan[P] \cup jps$
17: PRUNEPLANS$(optPlan[P], \{PREF_1..PREF_p\})$
18: PRUNEPLANS$(optPlan[\{R_1..R_n\}], \{PREF_1..PREF_p\})$
return $optPlan[\{R_1..R_n\}]$

added to account for the deferral of constraints. To accomplish this, we present ACCESSPLANS$_{opt}$, JOINPLANS$_{opt}$, EXTEND, and PRUNEPLANS$_{opt}$ in Algorithms 6, 7, 8, and 9, respectively.

First, ACCESSPLANS$_{opt}$ must iterate through all possible evaluation locations in order to ensure that efficient, highly preferred plans are found (line 3). Note that the list of sites used here could be easily shortened to only the sites hosting a copy of the relation currently under consideration. For the sake of simplicity, and to ease the worst case runtime analysis done in Section 6.6, however, we assume all sites host a replica of all relations.

An iteration through all evaluation sites is similarly performed in JOINPLANS$_{opt}$. In this case, however, there is no easy, practical way to cut down the list of potential sites. All sites must be explored, not just the sites that evaluate the children of a new join node. Consider two subplans p_1 evaluated at s_1 and p_2 evaluated at s_2. Let us assume that p_1 and p_2 are *significantly* faster than any other plans realize their respective relations. Let us further assume that the condition to be used in joining p_1 and p_2 is required by a PASQL constraint to be evaluated at s_3. If we make the

Algorithm 6 ACCESSPLANS$_{opt}$: Access plan enumeration.

Require: A relation R_i to enumerate access plans for.
Require: A list of requirements, $REQ_1..REQ_r$
Require: The conditions $C_1..C_M$ specified as part of the query.
Require: Possible evaluation sites $S_1..S_s$
1: $aPlans \leftarrow \emptyset$
2: $conds \leftarrow$ POSSIBLE_CONDS$(R_i, \{C_1..C_m\})$
3: **for** $k = 1$ to s **do**
4: $\quad cond\_sets \leftarrow 2^{conds}$
5: $\quad$ **for** each set of conditions $c \in cond\_sets$ **do**
6: $\quad\quad$ **for** each physical scan operator, po **do**
7: $\quad\quad\quad New \leftarrow$ NEW_SCAN(po, R_i, c)
8: $\quad\quad\quad aPlans \leftarrow aPlans \cup \{New\}$
9: $\quad\quad\quad rs \leftarrow \{REQ_1..REQ_r\}$
10: $\quad\quad\quad$ **if** not VIOLATES_REQ(New, rs) **then**
11: $\quad\quad\quad\quad aPlans \leftarrow aPlans \cup \{New\}$
12: $\quad\quad\quad\quad lo \leftarrow conds/c$
13: $\quad\quad\quad\quad$ **if** $|lo| > 0$ **then**
14: $\quad\quad\quad\quad\quad ss \leftarrow \{S_1..S_s\}$
15: $\quad\quad\quad\quad\quad exps \leftarrow$ EXTEND(New, lo, rs, ss)
16: $\quad\quad\quad\quad\quad aPlans \leftarrow aPlans \cup exps$
$\quad$ **return** $aPlans$

reasonable assumption that the cost to ship the results of both p_1 and p_2 to s_3 and perform a join is less than the cost to ship only one of the results and perform a crossproduct, then the fastest plan upholding required user constraints cannot be found by iterating only through sites evaluating the children of a prospective join.

To further enumerate all new access plans and join plans with all combinations of conditions applied and deferred for later processing, ACCESSPLANS$_{opt}$ and JOINPLANS$_{opt}$ must both compute the power set of applicable conditions that can be applied to the new plan (lines 4 and 7 of Algorithms. 6 and 7, respectively), and iterate through them to ensure that no split of conditions is missed. Then, for each newly realized plan, any usable conditions that are not applied to it should be added as another parent node atop the new plan (lines 12-16 of Algorithm 6, and lines 14-18 of Algorithm 7). Adding further operational nodes on to new join plans is accomplished though the use of the EXTEND function presented in Algorithm 8. JOINPLANS$_{opt}$ must further ensure that no pair of potential left and right subplans to be joined evaluate the same condition (line 4 of Algorithm 7), as doing so would incur unnecessary overall processing in the query plan. Fi-

nally, ACCESSPLANS$_{PAQO}$ and JOINPLANS$_{PAQO}$ must check to make sure each newly produced plan does not violate any required constraints (lines 7 and 8). Any plans that violate a required constraint are promptly pruned from the search space. By iteratively applying all sets of applicable conditions and then recursively extending each plan with additional operational nodes when needed, we ensure that this algorithm will explore all possible placements of conditions within the query plan, and hence, be able to find the fastest optimally preferred plans to evaluate a given query.

Finally, PRUNEPLANS$_{opt}$ must be modified. To start, checks must be made to ensure that two plans are to be executed at the same site (i.e., both of the root joins are to be evaluated by the same site)(line 5 of Algorithm 12) and evaluate the same set of SQL conditions (line 5 of Algorithm 9) before one can be considered to dominate the other. The metric for domination must also be changed to primarily consider the relative preference of two plans (lines 6-11). In the case that one plan does not clearly dominate another based on preferred constraint adherence, PRUNEPLANS$_{opt}$ falls back to the domination metric used by PRUNEPLANS (lines 12-17).

We will refer to the overall algorithm realized through the combining pseudocode presented in Algorithm 5 with Algorithms. 6, 7, 8, and 9 as the *Optimal algorithm*.

6.5 PROOF OF OPTIMAL SUBSTRUCTURE USING PASQL0

In this section, we will prove that the Optimal algorithm upholds the optimal substructure properties of traditional SQL query optimization while upholding user preferences. Key to this proof is the notion of a *preferred coverage set* of query plans for number of tables.

Definition 27 (Preferred coverage set)**.** *Given a query q and an integer s, the preferred coverage set $pcs(q, s)$ contains all query plans p for joining any s tables as a basis for evaluating q that do not violate any requirements and are not dominated by another plan p'. We say that a plan p dominates another plan p' if:*

- *p and p' join the same set of tables*

  AND
- *p and p' evaluate the same selection and join conditions*

  AND
- *The root nodes of p and p' are evaluated at the same site*

  AND EITHER
  - *p is more preferred than p'*

    OR
  - *p and p' are equally preferred*

    AND p sorts tuples based on an equal or extended list of attributes compared to p'

    AND p is faster than p'

In short, pcs(q,s) contains all of the fastest, preference optimal plans for joining s tables in q and applying all combinations of applicable selection and join conditions at each site, while also maintaining all interesting tuple sort orderings.

Note that the fastest of the most preferred plans joining all tables in s and applying all selection and join conditions applicable to s to evaluate q must be a member of the preferred coverage set for s. Hence the fastest of the most preferred plans to evaluate a query is a member of the preferred coverage set for all of the tables involved in the query.

Our proof will further make use of the following Lemma and its Corollaries:

Lemma 1. *Every $n \in \mathcal{N}$ can be checked for preference support in isolation. When any $n \in \mathcal{N}$ is used as a component in a query plan, the selection of nodes to make up its descendants and ancestors in that query plan cannot affect the constraints upheld by n.*

Proof. Whether or not a node upholds a given constraint is determined as specified in Definition 15 (upholding a constraint is simply the negation of violating a constraint). Note that Definition 15 relies on only the matching operator ($\pitchfork$) and the function evaluate. Both of which use only the constraint and the values of the ternary n. As stated in Definition 1, the ternary representation of a node contains only: the relational algebra operation represented by that node, the parameters to that operation, and the execution location assigned to the node. The descendants and ancestors of a node are not described in this ternary; the relationships between nodes are only encoded in the

query plan structure (as stated in Definition 1). Hence, the descendants and ancestors of a node cannot affect what constraints that node violates (and hence also the constraints that is supports) as they are used nowhere in our definition of constraint violation (Definition 15). □

From Lemma 1, we get the following Corollaries:

Corollary 5. *A node $n \in \mathcal{N}$ will uphold and violate the same constraints regardless of the subtrees that are assigned to it as children in query plan.*

Corollary 6. *A node $n \in \mathcal{N}$ will uphold and violate the same constraints regardless of what nodes are assigned as its ancestors in a query plan.*

With these in hand, we can present our proof by induction over the height of the query plans produced to optimize some query q.

Theorem 7. *The Optimal algorithm will always find the preferred coverage set of i tables according to some PASQL0 query q given the preferred coverage sets of less than i tables according to q.*

Proof. **Base Case** To find $pcs(q, 2)$ for a query q, the Optimal algorithm must first construct $pcs(q, 1)$. To find $pcs(q, 1)$, all access paths to each table needed to resolve q must first be enumerated (line 3-5 of Algorithm 5). Each of these plans is checked to ensure that it does not violate a required constraint in the ACCESSPLANS$_{opt}$ and EXTEND functions (line 10 of Algorithm 6 and line 8 of Algorithm 8). All that do not violate any requirements are added to the dynamic programming data structure for join level 1 (note that this data structure will eventually hold $pcs(q, 1)$). From here, all combinations of applicable selection conditions are applied to each of these access paths, and the (non-requirement violating) resulting access path/selection combinations are added to the dynamic programming data structure. At this point, all possible single node plans have been enumerated by the Optimal algorithm. To determine $pcs(q, 1)$, however, plans containing multiple nodes evaluated at different sites must be considered (e.g., a plan that accesses a table at one site, then performs a selection operation at another). Hence, each of the single node plans enumerated must be extended to apply all combinations of applicable conditions not applied to the single node via another node evaluated at a different site (and further, each of these extended plans is recursively extended until all applicable conditions are evaluated in the plan). Each extension will

attempt to apply all combinations of these leftover conditions to selection nodes evaluated at each possible site. Using this approach, by line 5 of Algorithm 5, all possible single leaf plans will have been enumerated by the Optimal algorithm. The Optimal algorithm checks whether the plan to be added dominates or is dominated by any plan already in the dynamic programming data structure in PRUNEPLANS$_{opt}$. Note that the domination metric used in PRUNEPLANS$_{opt}$ implements the exactly the domination metric proposed in Definition 27. In the case that a plan already in the data structure is dominated, it will be removed (lines 8 and 15 of Algorithm 9). In the case that the plan to be added is dominated by an existing plan, the appropriate flag is set to ensure that the new plan is not added to the dynamic programming data structure (lines 10, 17, and 19 of Algorithm 9). By enumerating all possible plans, the Optimal algorithm must find the fastest of the most highly preferred plans evaluating all possible sets of conditions and realizing all possible sort orders. By pruning only plans that violate requirements or are dominated by another plan according to the domination metric from Definition 27, the Optimal algorithm realizes $pcs(q, 1)$ by line 5 of Algorithm 5.

From here, the Optimal algorithm will combine all feasible pairs of plans (i.e., plans that scan different tables) in $pcs(q, 1)$ to form 2-way join plans at all possible sites. To show that $pcs(q, 2)$ can be constructed from $pcs(q, 1)$, we must show that any potential 2-way join plans that could have been constructed from single table plans pruned during the construction of $pcs(q, 1)$ would be dominated by a 2-way join plan constructed from plans in $pcs(q, 1)$.

Since the Optimal algorithm enumerates all possible single table plans and prunes only those that violate requirements or are dominated by another plan, by showing that using one of these pruned plans cannot lead to a faster or more preferred 2-way join plan than exclusively using plans in $pcs(q, 1)$, we will prove that $pcs(q, 2)$ can be generated purely based only plans in $pcs(q, 1)$. The Optimal algorithm enumerates all possible single table plans, and prunes only those that violate requirements or are dominated by another plan. Further, by our domination metric, any plan that is dominated by another can be universally replaced with the dominating plan. Both plans must join the same sets of tables, evaluate the same selection and join conditions, and be assigned to be evaluated by the same site, and hence, wherever one can be used as a building block for a new plan, the other can as well. No plan that is dominated by another could form the basis of a faster,

equally-preferred plan at later join levels. A dominating plan is defined to be universally better than a plan it dominates.

By Corollary 6, any constraints violated by a node will continue to be violated no matter what plans it is used as the basis of. Hence, the use of requirement violating single table plans will be of no help in constructing 2-way joins as any such 2-way join plans would still violate at least one required constraint. Similarly, using a plan that was pruned during the construction of $pcs(q, 1)$ because it was less preferred as the basis of a 2-way join plan would be disadvantageous compared to the more preferred plan that dominated it.

While the use of a lesser preferred plan could lead to the creation of a faster 2-way join plan than could be realized using the plans in $pcs(q, 1)$, this is of no consequence because speed is a less important optimization metric compared to relative preference (i.e., a more highly preferred plan will dominate a faster, less preferred, plan). Also, by Corollary 5, it is impossible for the use of a less preferred single table plan to allow for the creation of more preferred 2-way join node (or extension to a 2-way join node).

Given that no single table plan pruned in constructing $pcs(q, 1)$ can be used to construct a 2-way join plan that is not dominated by a plan generated using only $pcs(q, 1)$, the Optimal algorithm will find $pcs(q, 2)$ using $pcs(q, 1)$.

Inductive Hypothesis Assume that the Optimal algorithm will correctly produce $pcs(q, i)$ for some $i < n$.

Inductive Step We now show that given $pcs(q, i)$, the Optimal algorithm will produce $pcs(q, i+1)$. In the general case (i.e., forming plans to join i tables), the Optimal algorithm will continue to build plans rooted with nodes evaluated at all possible sites and evaluating all possible combinations of selection and join conditions. What must be shown here is that does not exist a plan to join i tables constructed from one or more subplans pruned in generating $pcs(q, 1), pcs(q, 2), ..., pcs(q, i-2), pcs(q, i-1)$ that would not be dominated by a plan in $pcs(q, i)$. Similar to the Base case, the use of a requirement violating subplan, a lesser preferred subplan, or a slower subplan cannot be of benefit to the construction of i-way join plan. Though we do not exhaustively enumerate all subplans joining less than i tables as was done in the base case, this is irrelevant as the PCS plans that are used to construct i-way join plans consist of plans that cannot be dominated by other plans by definition.

It is trivial to see that our arguments for joining i tables will also hold in joining $i+1$ tables. Hence, our argument holds in the general case, and the proof of our theorem by induction over the the number of tables being joined in a query plan is complete.

This completes our proof of Theorem 7.

□

6.6 COMPLEXITY OF THE OPTIMAL ALGORITHM

The literature has previously established that the Classic algorithm has a time complexity of $\mathcal{O}(3^n)$ and a space complexity of $\mathcal{O}(2^n)$ for queries over n base relations in a centralized setting. It has further been shown that moving this algorithm to a distributed setting (with s possible evaluation sites) requires a time complexity of $\mathcal{O}(s^3 * 3^n)$, and a space complexity of $\mathcal{O}(s * 2^n + s^3)$ [45]. In this section, we will formally establish the time and space complexities of the Optimal algorithm.

For each of the proofs in this section, we assume that all sites in the system host copies of all tables and are capable of evaluating any operation in a query plan. In the interests of avoiding the discussion and presentation of spurious constants, we further assume that there is only one physical scan operator and only one physical join operator available to the optimizer, and that no interesting sort orders can be taken advantage of during optimization. All of these assumptions are similarly made in [45]. We only add the assumption that no user-specified required constraints can be violated by any query plans to evaluate their respective queries. That is, considering that pruning requirement violating plans can decrease the practical time and space need of the running of our presented algorithms, we make this assumption to aid in defining the worst case time and space complexities of the Optimal algorithm.

Theorem 8. *The* time complexity *of the Optimal algorithm for queries over n base relations with r required constraints, p preferred constraints, and c SQL conditions (e.g., selection, projection, join conditions) that can be evaluated by s potential servers is* $\mathcal{O}(2^{c^2+3c} * s^{c+3} * c * 3^n * (r+p))$.

Proof. Because the domination metric used in PRUNEPLANS$_{opt}$ has been rewritten such that a plan cannot dominate another if they are both to be evaluated at different sites and, in the worst case, all

s sites host a replica of all tables involved in the query, each entry in $optPlan$ must correspond to at least s plans in the worst case (note there cannot be multiple plans in the same entry that differ only by sort order or physical operator due to our assumptions). s different scan plans for each relation, and, s different join plans for each set of base relations. The domination metric was also changed such that no plan can dominate another if they evaluate different sets of conditions. This, combined with accounting for evaluation at different sites, requires that each entry in $optPlan$ to corresponds to, in the worst case, $s * 2^c$ different plans. Each of the s different sites must be considered, and later joins must be able to draw upon a plan upholding any combination of SQL conditions at any possible site.

The increased size of $optPlan$ further entails that, for each call to JOINPLANS$_{PAQO}$, all $s * 2^c$ plans in $optPlan[O]$ must be combined with all $s * 2^c$ plans from $optPlan[P/O]$ with all s sites considered for evaluation of the new root join, and all 2^c possible combinations of SQL conditions applied to the join, increasing the runtime complexity by a factor of $(s * 2^c)^3$, or $2^{3c} * s^3$.

The only other additional runtime cost incurred by the Optimal algorithm is that required to call the EXTEND function. In the worst case, EXTEND is called to extend a node with no SQL conditions applied. In this case all c conditions are passed to EXTEND. Hence, we can upper bound the cost of all calls to extend as taking in c conditions. Inside of the extend function, two nested loops are run: one over all possible evaluation sites ($\mathcal{O}(s)$) and one over every set in the powerset of all conditions passed to EXTEND ($\mathcal{O}(2^c)$). Inside of the innermost loop, a new query plan is created that applies the current set from the powerset of conditions. All leftover conditions are then passed to a recursive call to extend. At this point, at least one condition must be applied to the plan created in extend, and hence, we can upper-bound the number of conditions passed to the recursive call to EXTEND by $c - 1$. Hence, we can represent the runtime of EXTEND by the following recurrence relation: $T(c) = s * 2^c * T(c-1) + \Theta(1)$.

To solve this recurrence relation, we consider the recursion tree that can be used to represent it. This tree would be of height c as each recursive call decrements c by 1. A recursive call at level i of the tree would further make $s * 2^{c-1}$ recursive calls, itself in addition to doing a constant amount of work (creating a new query plan and adding it to the list of plans to be pruned). Hence, we can determine the runtime of the EXTEND function as $T(c) = \Sigma_{i=0}^{c}(s * 2^{c-i})^i$, which we can bound as $T(c) < c * s^c * 2^{c^2}$.

As per our assumptions, no plans can be pruned due to requirement violation, and hence such pruning has no effect on the time complexity. Our use of relative plan preference will further have no effect on the time complexity of our algorithm. Using relative preference as part of the domination metric will only change *which* plans are stored in $optPlan$, it can not effect the number of plans stored for each entry in $optPlan$. Constraint adherence must be checked for each plan that is realized, however, and that does have a slight effect on the runtime complexity. For each plan realized, it must be determined if it violates any of the r required constraints or any of the p preferred constraints. This leads to the addition of the $(r + p)$ term in our runtime.

With this, we can bound the runtime of the Optimal algorithm as $\mathcal{O}(c * s^c * 2^{c^2} * 2^{3c} * s^3 * 3^n * (r + p))$. Or, equivalently: $\mathcal{O}(2^{c^2+3c} * s^{c+3} * c * 3^n * (r + p))$ □

The space complexity for our optimal algorithms follows rather simply from the previous proofs.

Theorem 9. *The* space complexity *of the Optimal algorithm for queries over n base relations with r required constraints, p preferred constraints, and c SQL conditions (e.g., selection, projection, join conditions) that can be evaluated on s potential servers is* $\mathcal{O}((2^c * s * 2^n + 2^{c^2+3c} * s^{(c+3)} * c) * (r + p))$.

Proof. The optimal algorithm must, for each entry in $optPlan$, store $2^c * s$ times as many plans as the classic dynamic programming algorithm. Further, additional overhead is incurred in the JOINPLANS$_{opt}$ function to iterate through all $2^c * s$ left child plans, $2^c * s$ right child plans, 2^c sets of conditions, $c * s^c * 2^{c^2}$ extended plans, and s sites. Finally, the Optimal algorithm must save, for each plan, a record of which constraints it upholds/violates, state which grows in size according to the number of constraints specified for the query being optimized, specifically $(r + p)$. □

6.7 SIMULATED ANALYSIS OF THE OPTIMAL ALGORITHM

As shown in Section 6.6, the Optimal algorithm has time and space complexities that contain an additional exponential term compared to the classic algorithm for distributed query processing previously demonstrated in the literature. To further illustrate the effect of this increased complexity

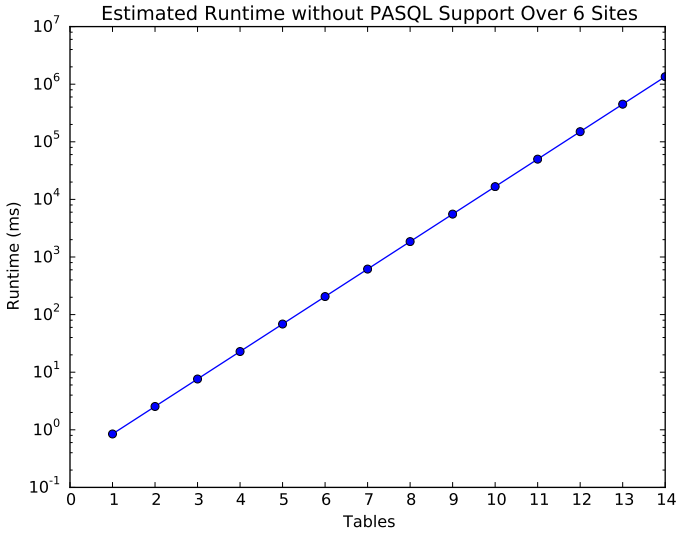


Figure 9: Log scale graph of runtimes for the classic distributed query optimizer from the literature.

and motivate the development of a greedy heuristic approach to constraint support, we now present a simulated analysis of the runtime of the Optimal algorithm.

Figure 9 presents a log scale graph of estimated runtimes for the variant of the Classic algorithm capable of optimizing distributed queries that has been proposed in the literature [45]. This algorithm has been shown to have a runtime complexity of $\mathcal{O}(s^3 * 3^n)$. The graph presented in Figure 9 simply plots the result of this growth function (assuming 6 possible evaluation sites) multiplied by a per-realized plan time cost that we gleaned from the experimental setup described in Section 7.4. We vary the number of tables needed by a query from 1 to 14, as using more tables results in estimated optimization times of over an hour, which we feel is certainly impractical.

Figure 10 presents a similar simulated evaluation for the Optimal algorithm. Here, both the number of tables involved in the query and the number of SQL conditions it contains are varied from 1 to 14. For this experiment, we again assumed the existence of 6 possible evaluation locations, and further assume that each query has 3 PASQL constraints applied. The results are again presented in log scale, with all runtime results over an hour shown in red. Just over 6% (12/196) of the queries simulated would require under an hour to optimize: queries over up to 9 tables with

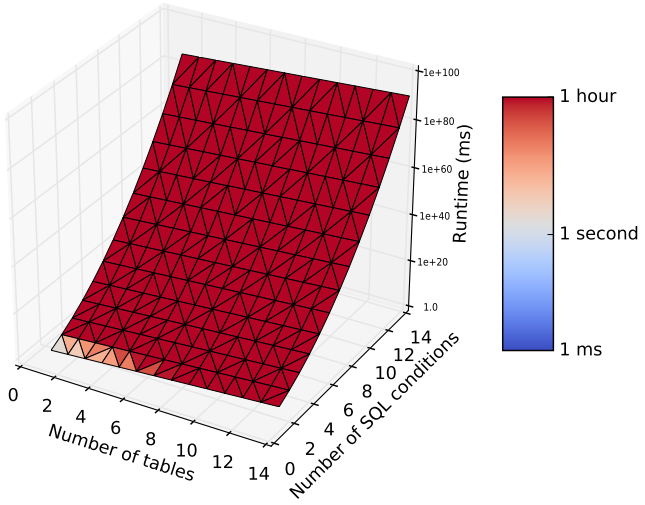


Figure 10: Log scale heatmap of runtimes for the Optimal algorithm.

1 SQL condition applied, and queries over up to 3 tables with 2 SQL conditions applied. Note that most queries with more than 1 more table than SQL condition can prove functionally impractical, as there would not be enough join conditions specified as part of the query to join all of the tables. Queries over n tables with any fewer than $n - 1$ SQL conditions applied must contain cross products in their result. Because of this, our simulated experiments show the Optimal algorithm to be bounded in the worst case to optimizing queries over a maximum of 3 tables in a reasonable amount of time.

6.8 SUMMARY

We began this chapter by formally establishing that (I, A)-privacy is locally-enforceable; that is, using this privacy notion, users can protect the privacy of the intension of their queries without the need of explicit server-side support. With this established, we presented a strawman two-

phase approach to (I, A)-privacy-aware query optimization to illustrate the need to account for intensional privacy *during* query optimization. Following this, we discussed the possible tradeoffs to be considered when implementing such an optimizer, and presented an algorithm for producing the fastest of the most preferred plans for a PASQL0 queries. We then proved the optimality of this algorithm, its runtime complexity, and its space complexity. This chapter concluded with a simulated analysis illustrating the worst case real world performance of this Optimal algorithm. This simulated analysis demonstrated the intractability of the Optimal algorithm, estimating even relatively simple queries with three PASQL constraints applied would take millenia to optimize. This result motivates the need for a practically efficient approach to optimizing PASQL queries, which we will present in the next chapter.

Algorithm 7 JOINPLANS$_{opt}$: Join enumeration psuedocode.

Require: A set of plans that will make up the left side of a new root join $Left_1..Left_u$.
Require: A set of plans that will make up the right side of a new root join $Right_1..Right_v$.
Require: A list of requirements, $REQ_1..REQ_r$
Require: Possible evaluation sites $S_1..S_s$

```
1:  jPlans ← ∅
2:  for i = 1 to u do
3:      for j = 1 to v do
4:          if APP_CONDS(Left_i) ∩ APP_CONDS(Right_j) = ∅ then
5:              for k = 1 to s do
6:                  conds ← POSSIBLE_CONDS(Left_i, Right_j, {C_1..C_m})
7:                  cond_sets ← 2^conds
8:                  for each set of conditions c ∈ cond_sets do
9:                      for each physical join operator, po do
10:                         New ← Left_i ⋈_{po,c} Right_j at S_k
11:                         rs ← {REQ_1..REQ_r}
12:                         if not VIOLATES_REQ(New, rs) then
13:                             jPlans ← jPlans ∪ {New}
14:                             lo ← conds/c
15:                             if |lo| > 0 then
16:                                 ss ← {S_1..S_s}
17:                                 exps ← EXTEND(New, lo, rs, ss)
18:                                 jPlans ← jPlans ∪ exps
        return jPlans
```

Algorithm 8 EXTEND: A recursive function for applying leftover conditions.

Require: A child plan, $Plan$
Require: A set of conditions, to_apply
Require: A set of requirements, $REQ_1..REQ_r$
Require: A list of possible evaluation sites, $S_1..S_s$

```
1:  newPlans = ∅
2:  for k = 1 to s do
3:      conds = POSSIBLE_CONDS(Plan, to_apply)
4:      cond_sets = 2^conds
5:      for each set of conditions c ∈ cond_sets do
6:          New ← NEW_ROOT(Plan, c) at S_k
7:          rs ← {REQ_1..REQ_r}
8:          if not VIOLATES_REQ(New, rs) then
9:              newPlans = newPlans ∪ {New}
10:             lo = conds/c
11:             if |lo| > 0 then
12:                 ss ← {S_1..S_s}
13:                 exps ← EXTEND(New, lo, rs, ss)
14:                 newPlans = newPlans ∪ exps
        return newPlans
```

Algorithm 9 PRUNEPLANS$_{opt}$: A function to prune dominated plans from a given join level according to a domination metric that allows for the construction of optimally preferred plans.

Require: A list Q of query plans joining the same number of base relations.
Require: A list of preferences, $PREF_1..PREF_p$.

```
 1: for all p ∈ Q do
 2:     CHECK_PREFERENCES(p, P)
 3:     reject ← False
 4:     for all other ∈ Q|p! = other do
 5:         if CONDS_APPLIED(p) == CONDS_APPLIED(other) then
 6:             if ROOT_SITE(p) == ROOT_SITE(other) then
 7:                 if p ≻ other then
 8:                     Q.remove(other)
 9:                 else if p ≺ other then
10:                     reject ← True
11:                     break
12:                 else
13:                     if p has a more interesting sort order then
14:                         if cost(p) < cost(other) then
15:                             Q.remove(other)
16:                     else if cost(plan) > cost(other) then
17:                         reject ← True
18:                         break
19:     if reject then
20:         Q.remove(p)
```

7.0 PAQO: PRIVACY-AWARE QUERY OPTIMIZER

In this chapter we present PAQO, our implementation of an (I, A)-privacy-aware query optimizer. By taking a greedy, heuristic approach to optimizing PASQL queries, PAQO is able to efficiently produce highly preferred plans. This approach allows PAQO to protect the intension of reasonably-sized user queries, something that is highly impractical for the Optimal algorithm as illustrated in Section 6.7.

We begin by describing the algorithm that underpins PAQO and analyzing its complexity. Following this, we present a review of additional implementation challenges that we faced in realizing PAQO. Finally, we conclude the chapter with an experimental evaluation of PAQO.

7.1 PAQO'S ALGORITHM

PAQO's algorithm generally makes use of the same (I, A)-privacy supporting scaffold that was used by the Optimal algorithm (Algorithm 5). To achieve efficient preference support that can be practically usable, however, the ACCESSPLANS, JOINPLANS, and PRUNEPLANS functions used by PAQO differ from those used by the Optimal algorithm. Further, PAQO's algorithm does not account for the deferral of SQL conditions so as to avoid the heavy cost of recursive calls to the EXTEND function. ACCESSPLANS$_{PAQO}$, JOINPLANS$_{PAQO}$, and PRUNEPLANS$_{PAQO}$ are presented in Algorithms 10, 11, and 12, respectively.

As can be seen, ACCESSPLANS$_{PAQO}$ differs only minorly from the ACCESSPLANS function used by the Classic algorithm. It adds only an iteration through all possible evaluation locations. Note that just as with the Optimal algorithm, the list of sites used here could be easily shortened to only the sites hosting a copy of the relation currently under consideration. Again, to simplify our

Algorithm 10 ACCESSPLANS$_{PAQO}$: Access plan enumeration with constraint checking.

Require: A relation R_i to enumerate access plans for.
Require: A list of requirements, $REQ_1..REQ_r$
Require: The conditions $C_1..C_M$ specified as part of the query.
Require: Possible evaluation sites $S_1..S_s$
1: $aPlans \leftarrow \emptyset$
2: $c \leftarrow$ POSSIBLE_CONDS$(R_i, \{C_1..C_m\})$
3: **for** $k = 1$ to s **do**
4: **for** each physical scan operator, po **do**
5: $New \leftarrow$ NEW_SCAN(po, R_i, c)
6: $aPlans \leftarrow aPlans \cup \{New\}$
7: **if** not VIOLATES_REQ$(New, \{REQ_1..REQ_r\})$ **then**
8: $aPlans \leftarrow aPlans \cup \{New\}$
return $aPlans$

analysis, we assume all sites host a replica of all relations. An iteration through all evaluation sites is similarly performed in JOINPLANS$_{PAQO}$. And again, just as in the Optimal algorithm, there is no easy, practical way to cut down the list of potential sites.

Both of these functions differ from their counterparts in the Optimal algorithm in that they do not attempt to defer potentially applicable SQL conditions. PAQO's algorithm trades off some measure of quality in the plans produced in order to make use of the classic heuristic of "pushing down" SQL conditions in a query plan to be as close to the leaves of the query plan tree as possible.

ACCESSPLANS$_{PAQO}$ and JOINPLANS$_{PAQO}$ still ensure that all produced plans do not violate any required constraints (lines 7 and 8). Any plans that violate a required constraint are promptly pruned from the search space. This ensure that, even though PAQO's algorithm may not output the fastest or most preferred plan for evaluating a PASQL query, it will never output a plan that violates a required constraint.

PRUNEPLANS$_{PAQO}$ shows that just as in the Optimal algorithm, one plan cannot dominate another in PAQO's algorithm if they are evaluated at different sites (line 5 of Algorithm 12). The relative preference of two plans is again used as the primary focus of plan domination (lines 6-11), falling back to the classic domination determination for equally preferred plans (lines 12-17). However, there is no need for PRUNEPLANS$_{PAQO}$ to differentiate between plans that uphold different SQL constraints because, again, PAQO's algorithm pushes down all SQL conditions in the query plan. Hence, any plans that could otherwise dominate one another because they join the

Algorithm 11 JOINPLANS$_{PAQO}$: Join enumeration psuedocode for PAQO.

Require: A set of plans that will make up the left side of a new root join $Left_1..Left_u$.
Require: A set of plans that will make up the right side of a new root join $Right_1..Right_v$.
Require: A list of requirements, $REQ_1..REQ_r$
Require: The conditions $C_1..C_M$ specified as part of the query.
Require: Possible evaluation sites $S_1..S_s$

```
1: jPlans ← ∅
2: for i = 1 to u do
3:     for j = 1 to v do
4:         c ← POSSIBLE_CONDS(Left_i, Right_j, {C_1..C_m})
5:         for k = 1 to s do
6:             for each physical join operator, po do
7:                 New ← Left_i ⋈_{po,c} Right_j at S_k
8:                 if not VIOLATES_REQ(New, {REQ_1..REQ_r}) then
9:                     jPlans ← jPlans ∪ {New}
       return jPlans
```

same sets of relations would evaluate the same set of SQL conditions. As we show in Section 7.2, maintaining only the fastest, most highly preferred plan for each site in $optPlan$ will lead to drastically better time and space complexities compared to the Optimal algorithm.

We refer to the overall algorithm realized through the combined pseudocode presented in Algorithms 5, 10, 11, and 12 as *PAQO's algorithm*.

7.2 COMPLEXITY OF PAQO'S ALGORITHM

Just as in our analysis of the Optimal algorithm, here, we assume that all sites in the system host copies of all tables and are capable of evaluating any operation in a query plan. We further again assume that there is only one physical scan operator and only one physical join operator available to the optimizer, that no interesting sort orders can be taken advantage of during optimization, and that no user specified required constraints can be violated by any query plans to evaluate their respective queries.

As described in Section 7.1, the only changes to the Classic algorithm needed to support PASQL constraints are the addition of distributed evaluation sites, the use of requirements to prune violating plans, and the use of preferences to refine the algorithm's domination metric. Hence, in

establishing how these changes affect the complexity of the PAQO's algorithm compared to the Classic algorithm, we can establish the overall complexity of PAQO's algorithm.

Theorem 10. *The* time complexity *of PAQO's query optimization algorithm for queries over n base relations with r required constraints and p preferred constraints that can be evaluated on s potential servers is $\mathcal{O}(s^3 * 3^n * (r + p))$.*

Proof. Because, in the worst case, all s sites host a replica of all tables involved in the query, and the domination metric used in PRUNEPLANS$_{PAQO}$ has been rewritten such that a plan cannot dominate another if they are to be evaluated at different sites, each entry in $optPlan$ corresponds to at most s plans (note there can not be multiple plans in the same entry that differ only by sort order or physical operator due to our assumptions). s different scan plans for each relation, and, s different join plans for each set of base relations. Hence, for each call to JOINPLANS$_{PAQO}$, all s plans in $optPlan[O]$ must be combined with all s plans from $optPlan[P/O]$ with all s sites considered for evaluation, increasing the runtime complexity by a factor of s^3.

As per our assumptions, no plans can be pruned due to requirement violation, and hence such pruning has no effect on the time complexity. Our use of relative plan preference will further have no effect on the time complexity of our algorithm. Using relative preference as part of the domination metric will only change *which* plans are stored in $optPlan$, it can not effect the number of plans stored for each entry in $optPlan$. Constraint adherence must be checked for each plan that is realized, however, and that does have a slight effect on the runtime complexity. For each plan realized, it must be determined if it violates any of the r required constraints or any of the p preferred constraints. This leads to the addition of the $(r + p)$ term in our runtime complexity. □

PAQO's algorithm imposes only a linear overhead of checking constraint adherence for each plan generated over the $\mathcal{O}(s^3 * 3^n)$ bound established in the literature for dynamic programming based optimization of distributed queries. We experimentally verify this linear overhead to optimization costs in Section 7.4.2.

Theorem 11. *The* space complexity *of PAQO's query optimization algorithm for queries over n base relations with r required constraints and p preferred constraints that can be evaluated on s potential servers is $\mathcal{O}((s * 2^n + s^3) * (r + p))$.*

Proof. Trivially, PAQO's algorithm must, in general, store s times as many plans as the classic dynamic programming algorithm (the s plans in each entry of $optPlan$), and must further have s^3 memory available for the JOINPLANS$_{PAQO}$ function. PAQO's algorithm must further save, for each plan, a record of which constraints it upholds/violates, specifically $(r + p)$. □

As [45] presents a space complexity of $\mathcal{O}(s * 2^n + s^3)$, we are again able to show the only overhead incurred by PAQO's algorithm is linear (one bit required per constraint). Again, the linear overhead to optimization space requirements is verified in Section 7.4.2.

7.3 CHALLENGES IN IMPLEMENTING PAQO

In order to realize the efficient support for (I, A)-privacy constraints described so far in this chapter, we implemented PAQO through extensive modifications to the optimizer from PostgreSQL [75]. PostgreSQL is a widely-used, open-source, transactional, object-relational database management system. PostgreSQL does not support queries over systems of autonomous, heterogeneous, distributed database systems, however.

Our first steps towards implementing PAQO was to establish support for distributed query optimization. All query plan nodes were augmented to maintain state about *where* they were assigned to evaluated. The join planning algorithm was expanded to iterate though all potential evaluation sites as described in Section 7.1. Most importantly, the cost estimator from PostgreSQL was rewritten to account for network transfers of data between query plan operations and to consider the parallel execution of operations occurring at different sites (while PostgreSQL is a threaded database system, it assigns each query to its own thread, hence, it does not consider parallel operation execution within the same query during optimization).

With this established, we moved on to enabling support for PASQL1 constraints. The parser was modified to detect the `REQUIRING` and `PREFERRING` clauses and extract the constraints they contain. We created mechanisms within the optimizer to detect query plan nodes that match constraint node descriptors. We needed to track down all points where new query subplans are created within the optimizer to ensure that a plan can be checked for requirement violations and removed from the search space as soon as possible. Finally, we had to modify the functions to

check for the domination of one plan by another to primarily consider preference support as outline in Algorithm 12. To efficiently keep track of what preferences a plan supports, we store a list of bitmaps with each plan representation. Each bit in these bitmap stores whether a query plan supports a given preference, and each bitmap corresponds to a different preference level (i.e., the first bitmap represents all of the constraints listed before the first `CASCADE` keyword, the second bitmap represents all of the constraints listed between the first and second `CASCADE` keywords, and so on). The relative preference of two query plans can then be quickly compared by iteratively comparing their bitmaps.

7.4 EXPERIMENTAL EVALUATION

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 [56, 1]), 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. Estimated query execution costs are presented in generic units. These units relate directly to the time required to evaluated the query plans produced, though the exact relationship between the estimation units and real time will depend on the machines evaluating the query (i.e., processor speeds, disk speeds, etc.). Note that this is the same way that estimated costs are reported by PostgreSQL's EXPLAIN statement, and for more information we direct the reader to [76]. The query plan visualizations presented in Figures 19, 20, and 21 were generated using a GraphViz ([39]) script operating on PAQO's logs.

Table 1: 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%

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 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.

7.4.1 Baseline Comparison to PostgreSQL

These experiments were performed on randomly generated queries over randomly generated relations following the experimental model used in [72, 45, 66]. The relations over which these queries are defined are sized according to the distribution shown in Table 1, and composed of attributes whose size follows the distribution shown in Table 2. 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 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

Table 2: 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%

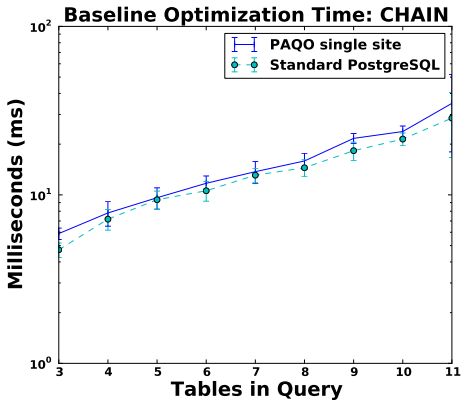
any constraints attached. Figures 11(a), 12(a), and 13(a) show the optimization time (in milliseconds) required by queries of CHAIN, STAR, or CLIQUE topology (respectively) while Figures 11(b), 12(b), and 13(b) show the memory (in bytes) required to optimize these queries.

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 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.

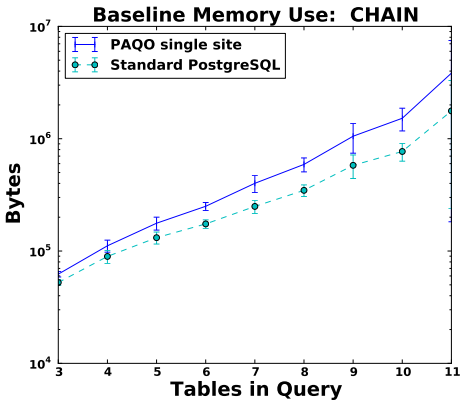
7.4.2 Processing Constraints

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 Figures 15(a) and 15(b). 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).

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 con-



(a) CHAIN Optimization Time

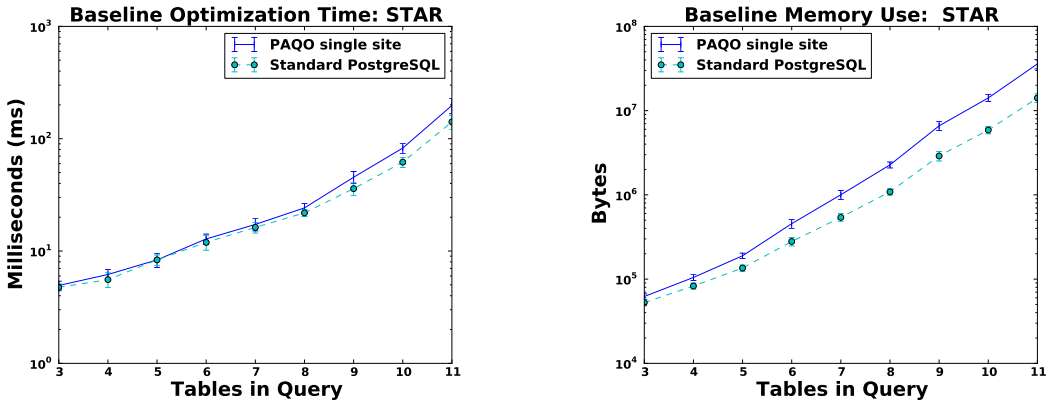


(b) CHAIN Memory Usage

Figure 11: Results of comparing CHAIN optimization times and memory usages of PAQO and the optimizer from PostgreSQL

straints 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. Figure 17 illustrates the shapes used in this experiment. Each node in Figure 17 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`).

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.



(a) STAR Optimization Time

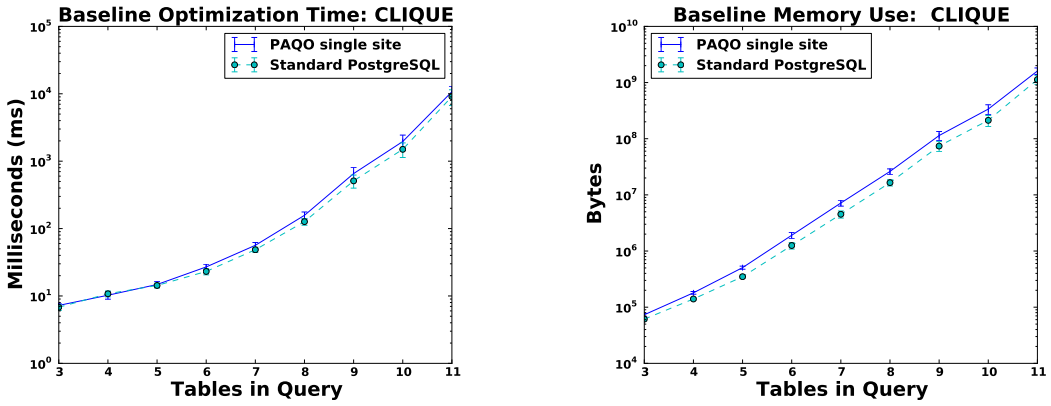
(b) STAR Memory Usage

Figure 12: Results of comparing STAR optimization times and memory usages of PAQO and the optimizer from PostgreSQL

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.

7.4.3 Case Study Performance

We have further run a series of experiments evaluating the performance of Alice's query from Section 1.2 with the requirements listed in Table 3. The results of these experiments are shown in Figures 18(a) and 18(b). The constraints from Table 3 are applied one at a time as requirements, and each data point is the average of five runs (preceded by a cache-warming run). Figure 18(a) 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 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



(a) CLIQUE Optimization Time

(b) CLIQUE Memory Usage

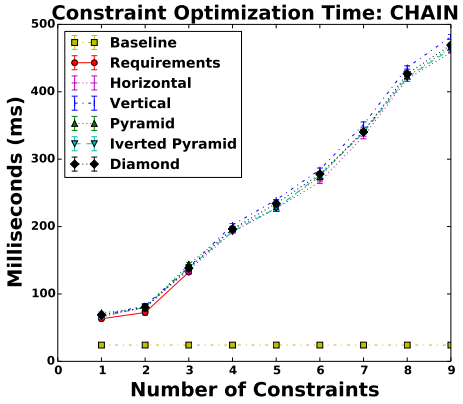
Figure 13: Results of comparing CLIQUE optimization times and memory usages of PAQO and the optimizer from PostgreSQL

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.

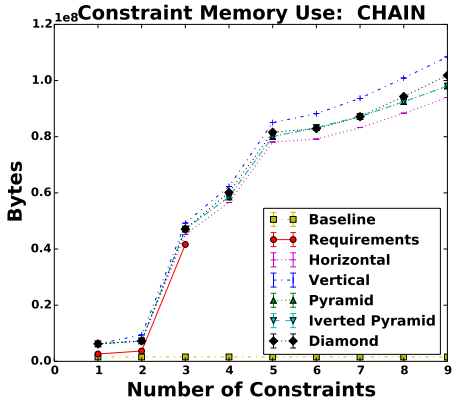
7.4.4 Query Plan Cost and Correctness

None of the queries and requirements used in our baseline and constraint experiments had any semantic meaning. Hence, for those experiments, comparing the tradeoff between constraint adherence and the estimated cost of executing a query plan similarly would have no real meaning. To explore this tradeoff, we further utilize the case study to demonstrate the cost and correctness of plans generated by PAQO.

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 again that these costs are given



(a) CHAIN Optimization Time



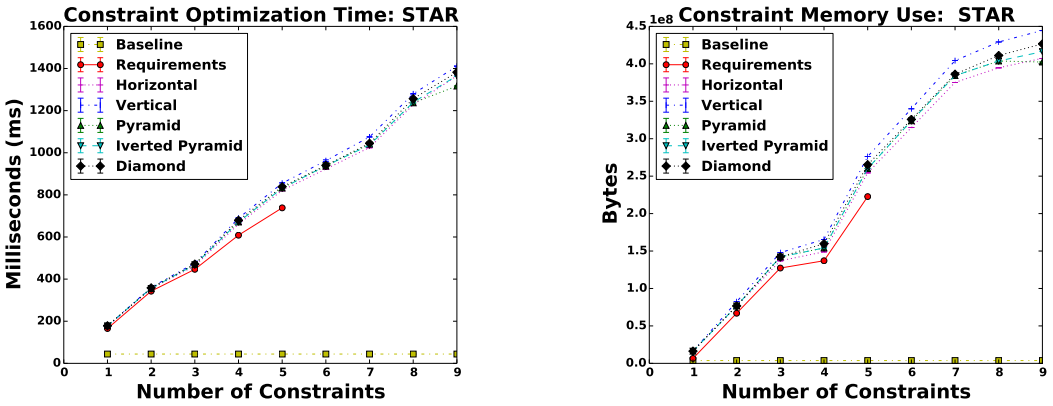
(b) CHAIN Memory Usage

Figure 14: Results of comparing CHAIN optimization times and memory usages of queries with varying shapes of constraints

in PostgreSQL's computational units. The real world time required to evaluated these plans will depend on the computational 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 Section 1.2), in Figure 19. PAQO's estimated cost to evaluate this query is 6189.25, while, on average, optimizing this query with PAQO took 6.34ms and utilized 218680 bytes of memory.

The first requirement from Table 3 states that operations on `Polluted_Water.pollutant` should be hidden from the `Inventory` server. To accommodate this, PAQO chooses to evaluate the root of the query plan at the `Querier` as shown in Figure 20. PAQO took an average of 12.013ms and 488808 bytes of memory to produce this plan and it has an estimated cost of 51097.00. This plan has a higher estimated cost than the base plan, though that is exactly why it



(a) STAR Optimization Time

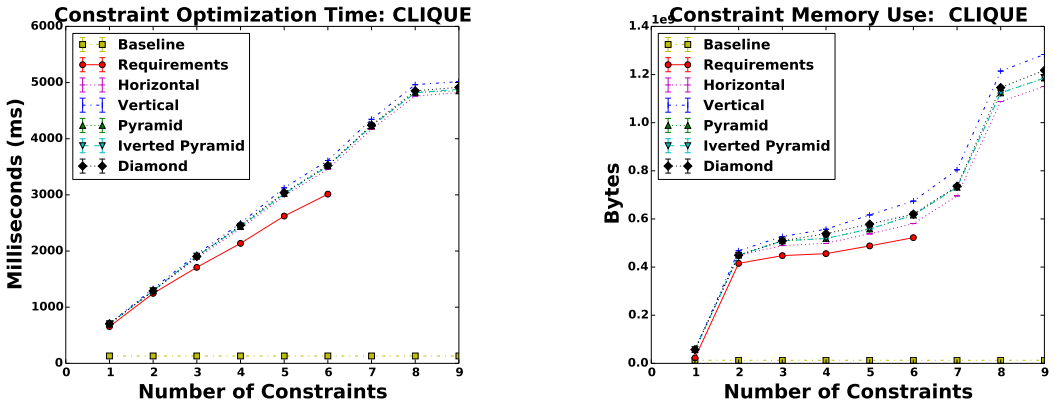
(b) STAR Memory Usage

Figure 15: Results of comparing STAR optimization times and memory usages of queries with varying shapes of constraints

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.

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 Figure 21. 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 in parallel with the transfer from Inventory to the `Querier`, and hence has only a small impact on the overall plan cost. This plan has an estimated cost of only 54064.34. PAQO is able to produce this query plan in an average of 16.02ms using 738052 bytes of memory.

Requirement 3 explicitly disallows Mapper from performing both of the operations that it does in Figure 21, 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`.



(a) CLIQUE Optimization Time

(b) CLIQUE Memory Usage

Figure 16: Results of comparing CLIQUE optimization times and memory usages of queries with varying shapes of constraints

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.

7.4.5 Discussion of Experimental Results

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 Section 7.4.2), 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 Section 7.4.2 serve as a worst-case experimental result. Lacking sufficient actual pruning, these

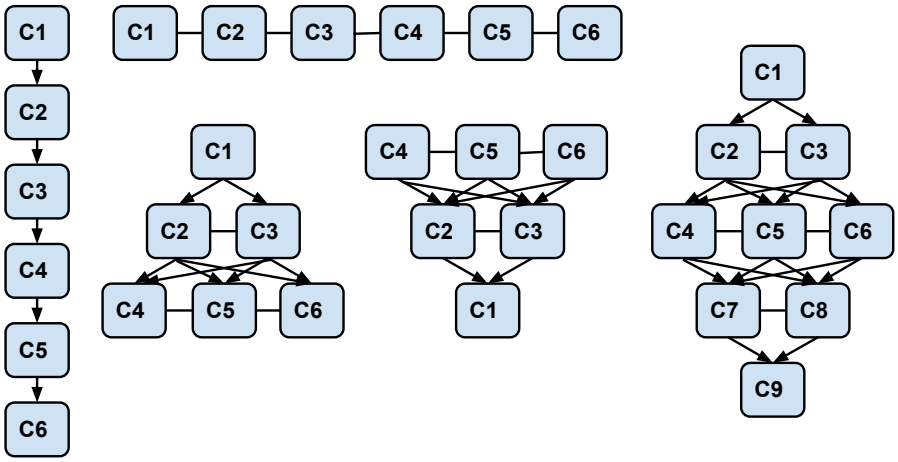
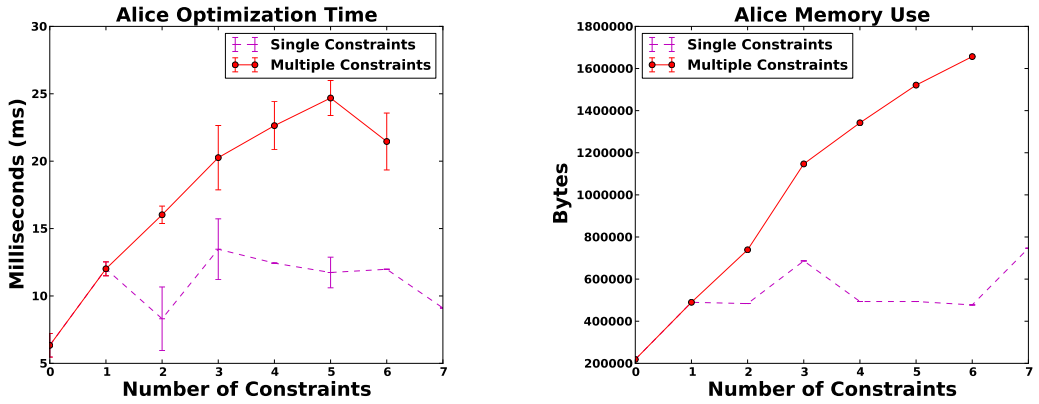


Figure 17: A visualization of the different preference shapes used in Section 7.4.2's experiments (clockwise from the left): Vertical, Horizontal, Diamond, Inverted Pyramid and Pyramid.

Table 3: Requirements applied to our case study of Alice's query from Section 1.2

Number	Constraint
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 ⟩;



(a) Optimization times of Alice's query.

(b) Memory utilization of Alice's query.

Figure 18: Case study performance results with varying numbers of requirements

queries are subject to the additional overheads imposed by constraint checking while achieving only minimal benefits.

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. Figure 15(a), 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. 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 Figure 15(a) 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

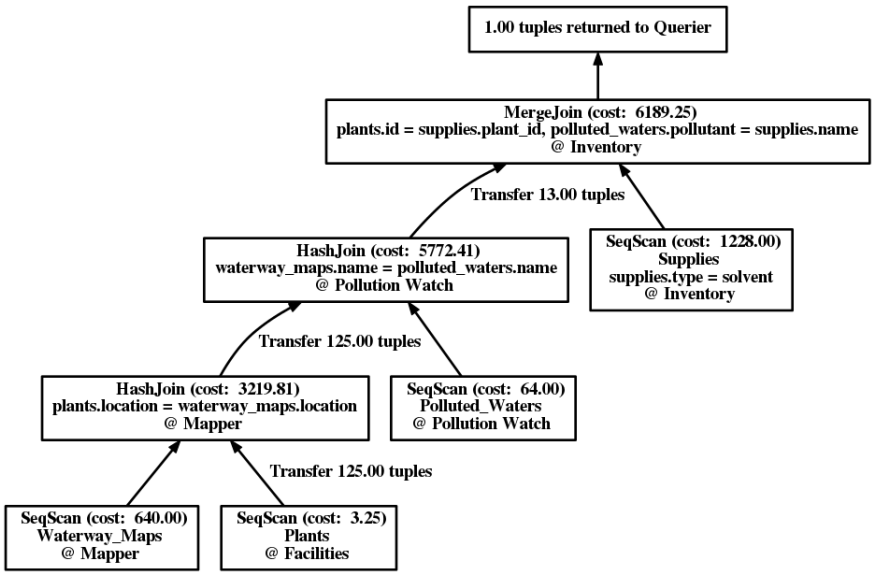


Figure 19: Query plan to evaluate Alice's query produced by PAQO with no constraints.

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 during optimization, and hence, maintains lower optimization times.

7.5 SUMMARY

In this chapter, we presented PAQO, our implementation of an (I, A)-privacy-aware query optimizer. We began by describing the algorithm that underpins PAQO and analyzing the complexity of this algorithm. We detail the challenges in rewriting the optimizer from PostgreSQL in order to realize PAQO. We presented an extensive experimental evaluation of PAQO. This evaluation verified the linear overhead of PAQO's algorithm compared to the Classic algorithm that we the-

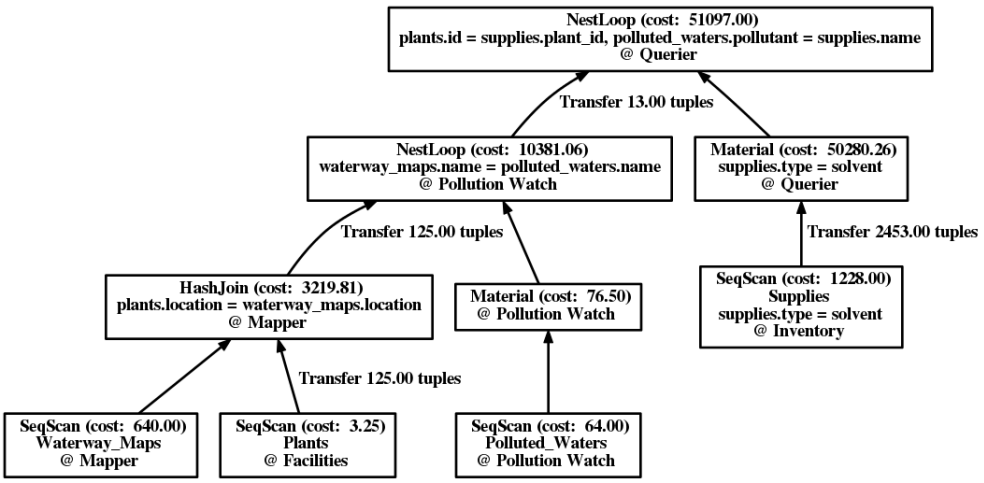


Figure 20: Query plan to evaluate Alice's query produced by PAQO with a single constraint.

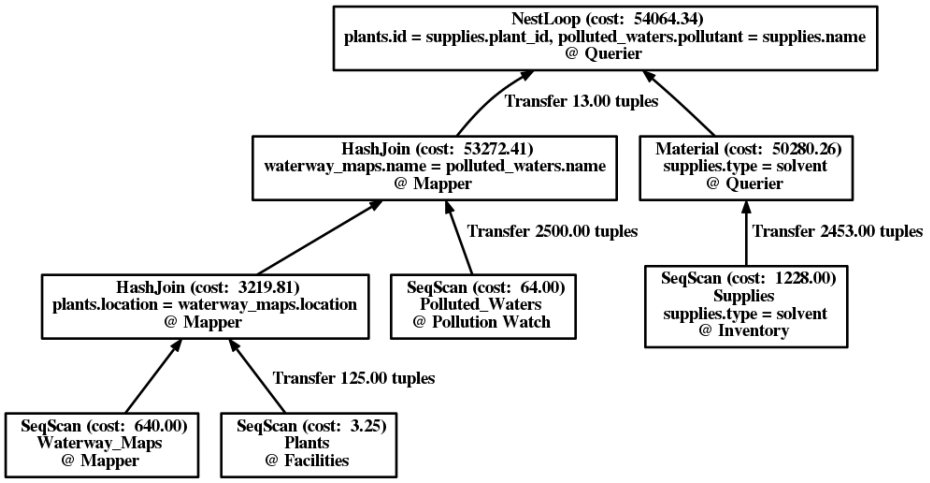


Figure 21: Query plan to evaluate Alice's query produced by PAQO with two constraints.

oretically established. It further showed that PAQO is able to efficiently produce highly preferred plans to evaluate PASQL1 queries, and that the "shape" of a preference specification does not affect optimization time. Finally, we presented the results of a case study analysis optimizing Alice's query with a number of different constraints attached to show that highly restrictive requirements can cut down on optimization time. Together, these results showed that PAQO is capable of efficiently optimizing PASQL queries to provide distributed database users with practical protection of their query intension.

Algorithm 12 PRUNEPLANS$_{PAQO}$: A function to prune dominated plans from a given join level according to PAQO's domination metric.

Require: A list Q of query plans joining the same number of base relations.
Require: A list of preferences, $PREF_1..PREF_p$.

```
 1: for all p ∈ Q do
 2:     CHECK_PREFERENCES(p, P)
 3:     reject ← False
 4:     for all other ∈ Q|p! = other do
 5:         if ROOT_SITE(p) == ROOT_SITE(other) then
 6:             if p ≻ other then
 7:                 Q.remove(other)
 8:             else if p ≺ other then
 9:                 reject ← True
10:                 break
11:             else
12:                 if p has a more interesting sort order then
13:                     if cost(p) < cost(other) then
14:                         Q.remove(other)
15:                 else if cost(plan) > cost(other) then
16:                     reject ← True
17:                     break
18:     if reject then
19:         Q.remove(p)
```

8.0 CONCLUSIONS AND FUTURE WORK

8.1 SUMMARY OF CONTRIBUTIONS

In this dissertation, we address the following hypothesis:

By extending SQL to allow for declarative specification of constraints on the attributes of query evaluation plans and accounting for such constraints during query optimization, users can produce efficient query evaluation plans that protect the private intensional regions of their queries without explicit server-side support.

To support this hypothesis, we first needed to formalize an approach for users to specify privacy constraints on plans to evaluate their queries. We then needed to show that not only could these constraints be enforced, but that they could be enforced *locally*. Finally, we needed to show that users could practically realize the privacy protections offered by our approach. These goals are achieved by the following specific contributions of this dissertation:

Chapter 4 formally established the concept of intensional privacy and how the privacy of the intensions of users' queries can be violated. It presented a formal definition of our novel notion of privacy, (I, A)-privacy, and contained a proof that our constructs for describing (I, A)-privacy constraints are identify *any* portion of an SQL query.

This proof allowed us to state that our approach can be used to protect the intensional privacy of user queries. Privacy is an inherently personal concept that can vary greatly not only from person to person but also from query to query. By ensuring that our formalization of (I, A)-privacy constraints is flexible enough to cover any portion of user query, we ensured

that regardless of what piece of a query a user would consider to be sensitive they can author an (I, A)-privacy constraint to protect it.

Chapter 5 presented a set of extensions to SQL that allows users to express (I, A)-privacy constraints over the queries that they issue. Specifically, the syntax of three variants of PASQL are presented: PASQL0, PASQL1, and PASQL2. It contained a proof that PASQL0 (and hence PASQL1 and PASQL2) is capable of subsuming private information retrieval (PIR): (I, A)-privacy constraints can be used to encode PIR constraints. A preference algebra was presented that is utilized by all three variants of PASQL to prioritize and balance competing (I, A)-privacy constraints. These preference constructs are further used by PASQL2 to allow users to explicitly balance privacy and performance when evaluating their queries.

Formally establishing a syntax and semantics for the three variants of PASQL is a necessary step towards enforcing (I, A)-privacy and supporting the hypothesis of this dissertation. In order for (I, A)-privacy constraints to be enforced, users must have a way to communicate their privacy concerns to the system processing their queries. PASQL provides exactly this functionality in declarative way, maintaining the boon that declarative languages have provided to database users for the last 40 years.

Further, by showing that PIR constraints can be expressed using PASQL, we ensured that PASQL users will be able to take advantage of PIR techniques whenever they are applicable to the user's query and supported by the remote servers being queried, while still being able to utilize the novel privacy enforcement techniques presented in this dissertation anytime PIR's necessary pre-conditions are not met.

Chapter 6 first formally established that (I, A)-privacy is locally-enforceable. That is, using this privacy notion, users can protect the privacy of the intension of their queries without the need of explicit server-side support. The fact that (I, A)-privacy constraints are locally-enforceable is a key distinguishing advantage of our approach to protecting querier intension, and a direct condition of supporting the hypothesis of this dissertation.

Chapter 6 further illustrated the need to account for intensional privacy *during* query optimization via a strawman approach to (I, A)-privacy support. The tradeoffs to be considered in accounting for (I, A)-privacy during query optimization were described, and, with them in mind, our novel Optimal algorithm for producing the fastest of the most preferred plans for

PASQL0 queries was presented. We proved this algorithm to be optimal, and formally establish its runtime complexity and its space complexity. Chapter 6 concluded with a simulated analysis illustrating the worst case real world performance of this Optimal algorithm.

We not only showed that the problem of finding the fastest optimally-preferred plan for evaluating a PASQL0 query solvable, but further presented an algorithm for solving this problem. Combined, these two points clearly established that users can generate efficient plans that protect the private intensional regions of their queries, the final component of the hypothesis of this dissertation. In analyzing the time and space requirements of running this algorithm, however, we showed that such optimal support comes at a great cost. We further established a space of tradeoffs to be considered in enforcing (I, A)-privacy constraints that presents a large potential for exploration in future work.

Chapter 7 presented PAQO, our implementation of an (I, A)-privacy-aware query optimizer. The novel algorithm that underpins PAQO was described, and its time and space complexities were analyzed. It further presents an extensive experimental evaluation of PAQO. This evaluation verified the linear overhead of PAQO's algorithm compared to the Classic algorithm that was theoretically shown in our analysis. It further showed that PAQO is able to efficiently produce highly preferred plans to evaluate PASQL1 queries, and that the "shape" of a preference specification does not affect optimization time.

By implementing heuristic support for PASQL1 in PAQO, we showed that not only is it *possible* for users to efficiently protect their query intension, but further that it is *practical*.

With these contributions, we are able to support the hypothesis of this dissertation. In Chapters 4 and 5, we showed how users can express (I, A)-privacy constraints using PASQL. In Chapter 6, we proved that these constraints can be locally enforced and provided an algorithm for producing the fastest, most highly preferred plan for a given PASQL query. Finally, we presented PAQO, a practical implementation of an (I, A)-privacy-aware query optimizer, in Chapter 7.

8.2 FUTURE WORK

By presenting the novel privacy notion (I, A)-privacy and further exploring the potential tradeoffs to be considered in enforcing (I, A)-privacy, this dissertation sets the stage for a large body of future work.

8.2.1 Distributed Query Execution Engine

PAQO makes relatively simple assumptions about the capabilities of remote database servers processing the query plans that it generates. They do not need to take any role in actively ensuring that users' intensional privacy is being protected, or even know whether or not the queries that they are processing have PASQL constraints attached to them. Remote servers are only assumed to accept and process query evaluation plans optimized elsewhere, accept relations streamed to them from other servers/sites that require further processing, and ship generated results either to other remote servers or back to the querier. However, there are currently no such deployed systems that can accomplish these tasks.

While motivating use cases for such a system were clearly outlined in Chapter 1, and the research community has developed the techniques for enabling distributed query processing, a distributed system of heterogeneous and fully autonomous database servers has yet to materialize. By ensuring that the intensions of users' queries can be protected throughout the query evaluation process, we have removed one hurdle to the adoption of such a distributed database system.

Aside from the adapting the inputs and outputs of the execution engine to receive and transmit query plans and tuples from/to the network in addition to disk, each entity in the system must also be modified to allow for the maintenance of a distributed catalog of all of the data stored in the distributed system. Data storage servers will have to export their locally stored catalogs and statistics (e.g., selectivities and cardinalities) so that remote query optimizers such as PAQO can make accurate runtime estimations during query optimization. As mentioned in Section 4.1, these updates should occur either as pushed updates in response to changes to data stored in the system, or by having optimizers perform periodic polling of all data hosts in the system.

8.2.2 PASQL2 Implementation

Implementing support for PASQL2 within PAQO would allow users to practically bound the cost in terms of estimated query performance that they are willing to pay for the privacy-preserving evaluation of their queries. Examples of how users can wield PASQL performance constraints to directly manage this tradeoff between privacy and performance are presented in Section 5.3.

To enable support for PASQL2 within PAQO, the sets of exposed optimizer estimation functions, $\mathcal{F}$, and numerical preference constructors, $\mathcal{PC}$, would need to first be established. In [41], the authors present a number of numerical preference constructors that can be used to create PASQL performance constraints, and hence populate $\mathcal{PC}$. With this in hand, PAQO could be modified to check the corresponding metadata generated during query optimization to check whether any specified PASQL2 constraints are upheld on the plans it generates. While there are no theoretical obstacles to enabling such checks, some usability concerns need to be accounted for. As described in Section 5.3, PASQL performance constraints on estimated query runtime would require users to specify a direct bound for the overall runtime of their queries (e.g., that it be less than two minutes). While it is straightforward to see how this could be implemented, it would require users to be aware of reasonable bounds for each of their queries. It may be more accessible to have users establish preferences over *relative* performance metrics (e.g., that a query should take less than twice as long to run).

The use of relative performance constraints raises several interesting questions. First, what query plan should be used as a basis for comparison? A plan to evaluate the base query without any PASQL constraints may be the intuitive interpretation of a plan taking x times as long to execute for many users. In this case, the constraint would be interpreted as performance relative to issuing the query to a system that is fully trusted. Another interpretation would be to consider performance relative to a plan that supports all constraints that are more important the given performance constraint. Through this lens the performance constraint indicates the cost the user is willing to pay just to support a specific set of constraints. This second interpretation more closely models the use case we outline when describing PASQL2 throughout this dissertation.

A second question to be considered is how this baseline measure should be determined: should PAQO perform multiple optimizations of a single query to account for PASQL2 constraints? These questions will be the focus of future work on implementing PASQL2 support within PAQO.

8.2.3 Further Analysis of the Runtime of the Optimal Algorithm

In Section 6.6, we clearly establish the extreme performance cost to optimal support of PASQL constraints using, to the best of our knowledge, the most efficient algorithm for finding the fastest of the most highly preferred plans for evaluating PASQL0 queries. We have not, however, produced a lower-bound on the time required to produce optimal plans for queries with attached PASQL constraints. While it is well-established that query optimization in and of itself requires exponential time to perform, it is important to also bound the time requirements of the overheard to query optimization necessitated by optimal PASQL constraint support. Such a bound could formally confirm that the Optimal algorithm presented here is the best approach for finding the fastest of the most highly preferred plans for evaluating PASQL0 queries. A different result from this analysis, on the other hand, could open the door for further work in developing optimal approaches for PASQL support.

8.2.4 Analysis of the Quality of Plans Produced by PAQO's Algorithm

PAQO is able to efficiently produce highly-preferred query plans for queries with attached PASQL constraints. However, we have yet to establish a bound on how close the quality of these plans comes to the quality of the optimal plans for user queries. The results from Section 6.6 establish that implementing the optimal algorithm for an experimental comparison of output plans would be infeasible for reasonably sized queries. Hence, a theoretical bound must be established on the quality of plans produced by PAQO's algorithm compared to the optimal.

8.2.5 Alternative Implementations of a Privacy-Aware Query Optimizer

In parallel with efforts to analyze the quality of plans produced by PAQO's algorithm, future work will explore the possibility of implementing a Privacy-Aware Query Optimizer based on query

optimization approaches other than dynamic programming. Being based on dynamic programming, PAQO is subject to the same limitations in terms of query size as the Classic algorithm. PostgreSQL, for example, only uses the Classic algorithm for queries over fewer than 12 tables. After this hard cutoff, PostgreSQL switches to a heuristic approach (a genetic query optimization algorithm). Through the use of a randomized query optimization algorithm, for example, it may be possible to efficiently produce fast, highly-preferred query plans. To provide practical protections for the intensional privacy of large queries, alternative heuristic approaches to (I, A)-privacy constraint support need to be explored.

8.2.6 Evaluation of Interactive Query Optimization Interfaces

There are two concerns that could hamper the widespread adoption use of PASQL. First, users would have to learn the syntax and semantics of PASQL in addition to SQL. Incurring such a burden detracts from the usability of our proposed approach and its practical ability to protect user privacy. Second, users would need to anticipate possible intensional leaks that could violate their privacy. To address both of these concerns, a more user-friendly approach to specify (I, A)-privacy concerns is needed.

Towards this ends, we have developed an interactive approach to query optimization that allows users to ensure that the intension of their queries is protected by (I, A)-privacy without the need to learn PASQL. Our general approach to interactive query optimization is illustrated in Figure 22. To begin the interactive query optimization process, users simply issue their queries in SQL to a interactive query optimization client. This client will then proceed to optimize the user's query and produce a plan to evaluate that query. Before passing this plan off for evaluation, however, the client will present a representation of that plan to the user. This allows users to directly see how the intension of their queries will be disseminated during the evaluation of the displayed query plan. In response to this information, users can inform the client of what portions of the generated query plan are unacceptable (i.e., users can identify portions of the query plan that they feel violate their privacy and ensure that the client will correct them before issuing the query plan to be evaluated). The client will then take this collection of user constraints and the originally specified query, produce a new query plan that does not violate the constraints, and present this

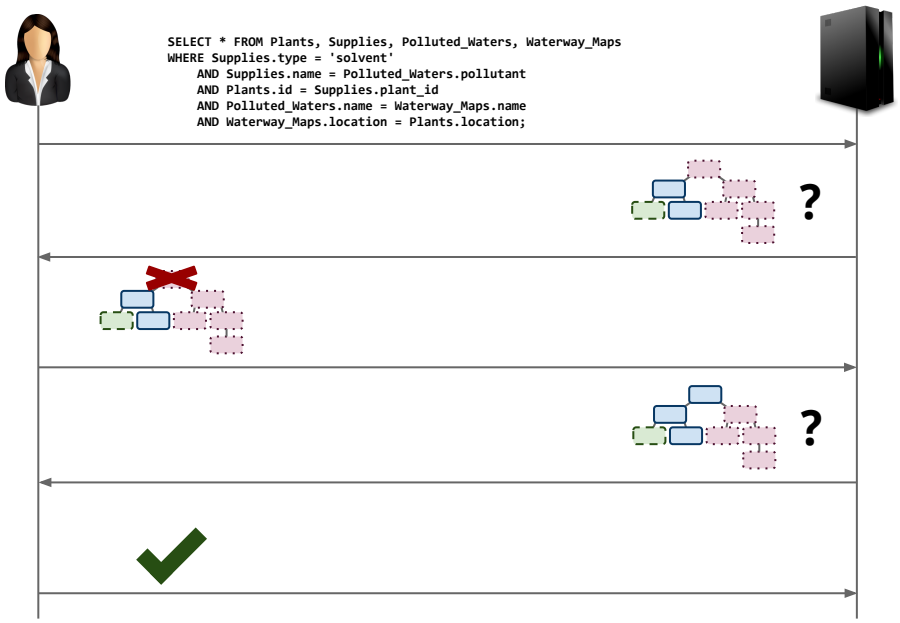


Figure 22: An overview of our proposed interactive optimization process.

new query plan to the user for review. This interactive process will continue until an acceptable query plan is produced (i.e., the user does not feel it violates her privacy). Once such an acceptable query plan is generated, the client will issue this plan to all of the remote sites needed to evaluate it and return the result to the user.

Currently, we have developed two different client applications for performing interactive query optimization [61]. Both of these clients take user feedback and generate PASQL constraints accordingly. These constraints are then attached to the original SQL query and passed to PAQO for optimization. Each of the two clients then takes a different approach to presenting the plan to the user. The two main graphical user interfaces presented by these clients are show in Figures 23 and 24.

Figure 23 presents our *query view* approach to interactive query optimization. After a query plan is produced, the user is presented with an interactive version of their original query. The user can highlight any portion of the query and be shown what site (or sites) is assigned to evaluate operations on involving that portion of the query. Specifically, Figure 23 shows a user highlighting the

relational attribute `Supplies.name` while the client indicates that only `Pollution Watch`'s server is assigned to evaluate any operation on that attribute in the current plan. Similarly, if a user were to highlight a server in the system, the client would be shown what portions of the query that server is assigned to evaluate.

Our second client interface, the *hierarchical view* interface is shown in Figure 24. With this interface, users are presented with lists of all of the servers in the system and and all of the parameters (e.g., attribute names, selection conditions, join conditions) of the query. Users can either start by selecting a server or a parameter, and be shown either a list of all of the parameters operated on by that server, or all of the servers that evaluate an operation on that parameter. Selecting both a server and parameter will present the user with the list of operations in the current query plan that operate on the selected parameter at the selected site. In Figure 24, for example, the user has selected the `Mapper` server from the "Sites" hierarchy, and the `location` attribute of the `plants` table from the "Parameters" hierarchy. In response to this, the client displays the only operation node form the current plan that operates on the `location` attribute of the `plants` table, and is evaluated by the `Mapper` server (far right). If the user changed the selection under the "Sites" hierarchy to be the `Pollution_Watch` server, the entire parameters hierarchy would update to reflect all of the relations and attributes evaluated at the `Pollution_Watch` server, and the "Query Plan Nodes" pane would be blank until the user selected an item from the "Parameters" hierarchy.

Both of these interfaces further assist users in crafting constraints in response to their viewing of the current query plan. Figures 25 and 26 present the dialogues for constraints creation for the query view and hierarchical view interfaces (respectively). After specifying new constraints to be applied to their query, the user can instruct either interface to produce a new query plan. After applying PASQL constraints to the user's query and having PAQO reoptimize the query with the new constraints, the interfaces not only present the new query plan to the user, but further highlight the differences between the previous and current query plans (as shown in Figure 27).

While these interfaces represent a represent a very strong first step towards ensuring the usability of (I, A)-privacy, we currently have no evaluation to verify their usability. As the subject of ongoing work, feedback from demonstrating these interfaces will be used to drive a future user study to evaluate the usability that they provide.

8.3 IMPACT OF THIS DISSERTATION

The impact of this dissertation can be described according to two different metrics:

Impact on science and technology. This dissertation opens a new area of research in privacy-preserving distributed query processing, and poses several avenues of future work in this area as described above. First, this dissertation presents a novel notion of user privacy, (I, A)-privacy. (I, A)-privacy adds to the body of literature describing how users can protect their privacy in online interactions with remote parties. By incorporating (I, A)-privacy into the design of software systems and programs, designers and developers are provided a new approach to ensuring that the privacy of the users of their systems and applications will be protected.

In proposing (I, A)-privacy, this dissertation also describes the problem of enforcing (I, A)-privacy constraints on user queries during query optimization. This dissertation presents not only algorithms to solve this problem, but theoretical analysis and bounds on the performance of algorithms for solving this problem. This dissertation takes the important first steps to exploring the solutions to the problem of enforcing user-specified (I, A)-privacy constraints.

Impact on society. This dissertation takes a rather forward-thinking approach of addressing the privacy concerns of distributed database systems that have yet to achieve widespread popular use. This is a major strength of the work. History is littered with examples of widely used computer systems and protocols that were deployed and widely adopted without concern for security and user privacy. Attempting to address these concerns after widespread adoption can be an arduous process if successful at all [6, 79]. By analyzing the privacy issues involved in querying distributed systems of fully autonomous and heterogeneous database system now, we allow for such systems to account for these issues and protect users from the start.

This dissertation additionally furthers this goal by ensuring the usability of our approach to protecting querier privacy. First of all, the techniques presented here are locally-enforceable. While this has the clear benefit to users of being able to protect their privacy independently, without explicit server-side support, it also bodes well for widespread adoption of a privacy-preserving system. Entities that will host the database servers described in our system model can focus entirely on creating a service that provides utility to users without the additional concern or justification

of cost to support user privacy. The database servers can be set up purely to process incoming relational query plans without any concern for supporting (I, A)-privacy.

To ease user adoption of (I, A)-privacy support, this dissertation presents an initial look at ongoing work to ensure the user-friendliness of (I, A)-privacy constraint specification. In proposing interactive query optimization, this dissertation takes the first steps towards easing adoption of (I, A)-privacy by users, ensuring that users will easily be able to make use of the approaches presented here to protect their privacy.

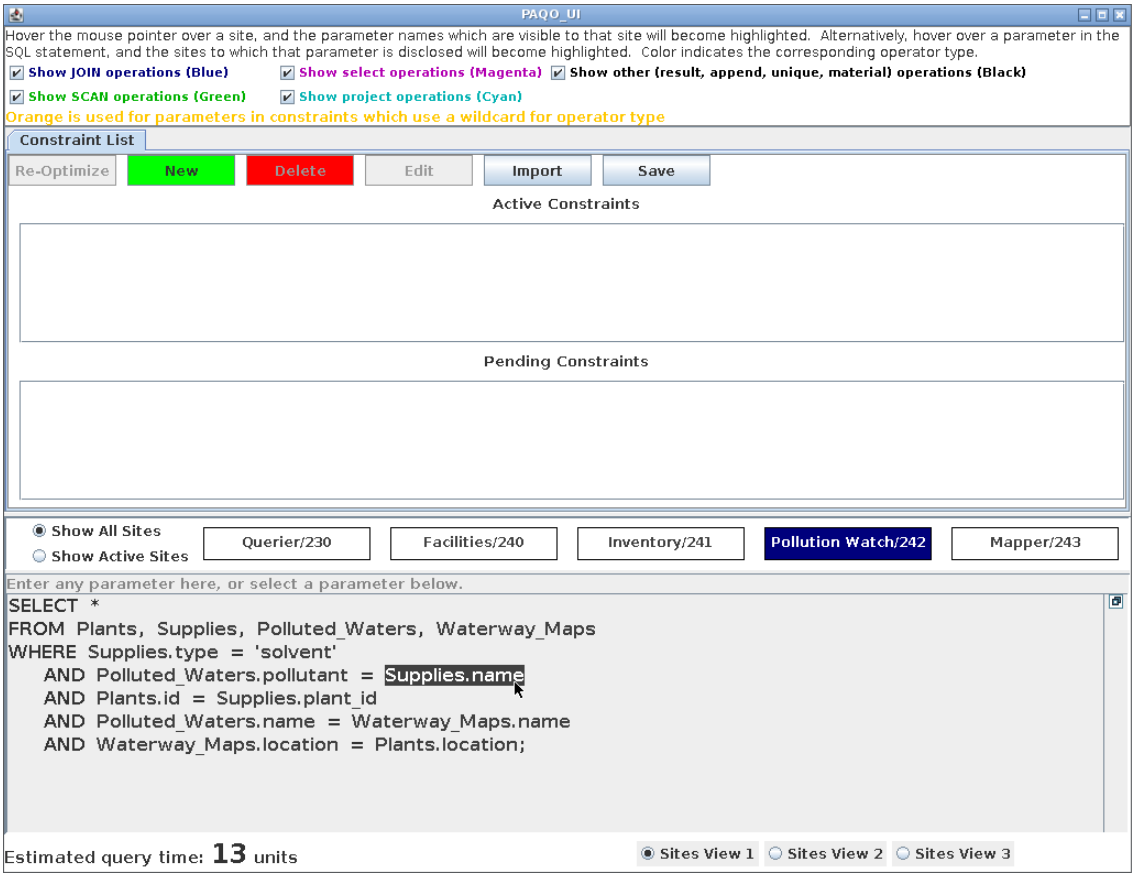


Figure 23: The main view of our query view interface for interactive query optimization.

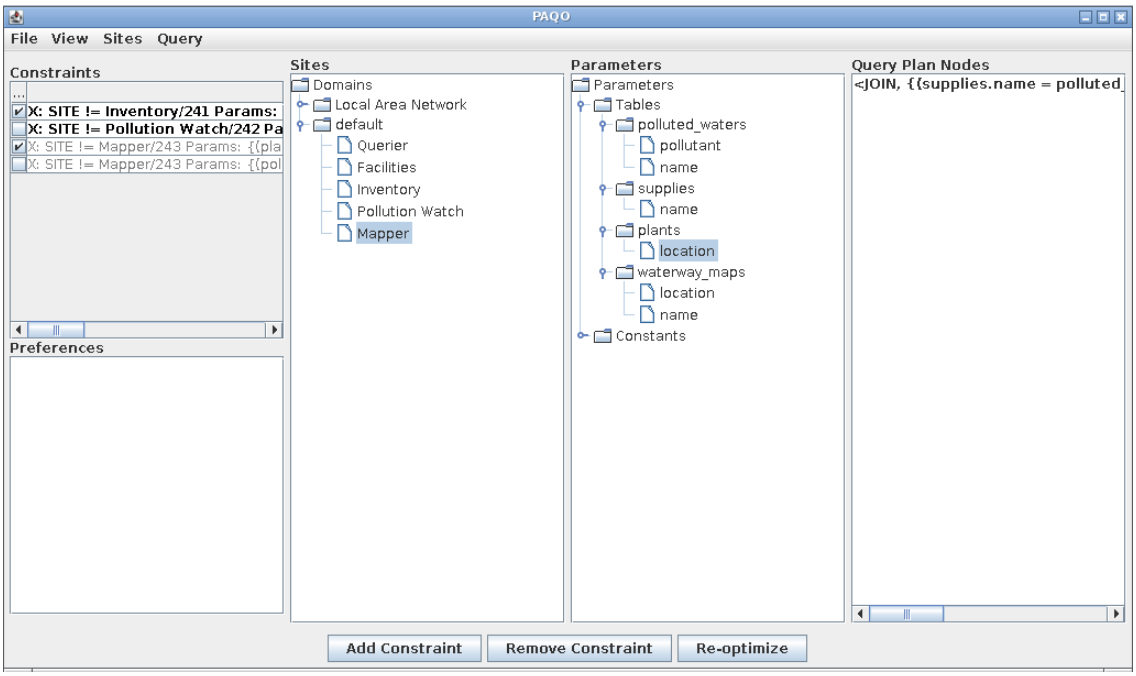


Figure 24: The main view of our heirarchical view interface for interactive query optimization.

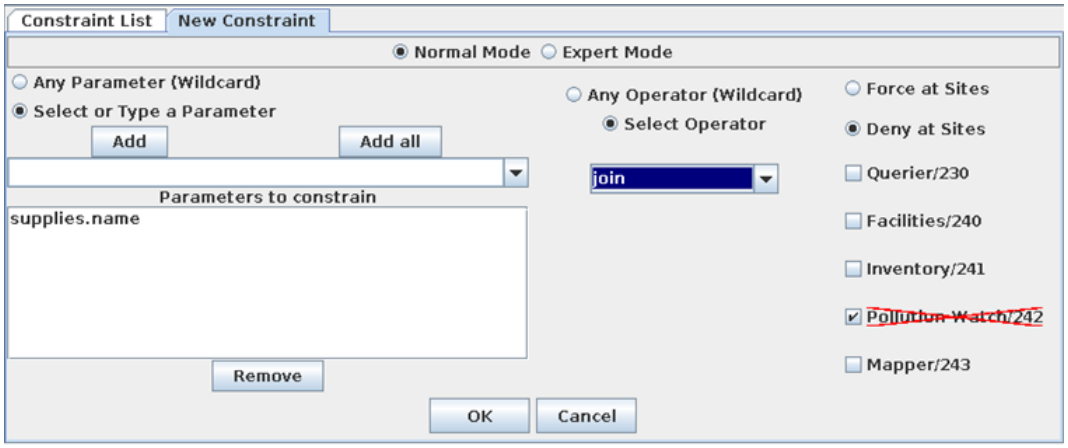


Figure 25: Our query view interface's dialog for creating a new (I, A)-privacy constraint.

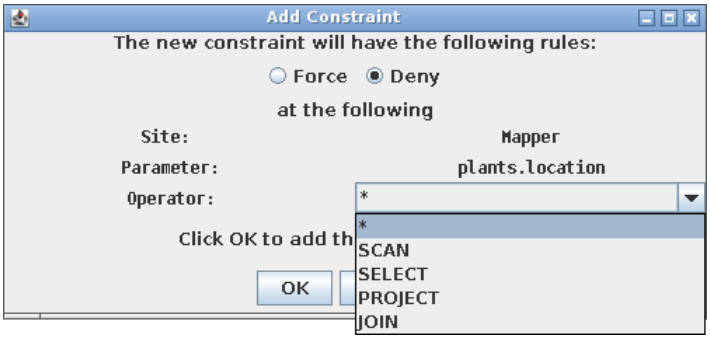


Figure 26: Our hierarchical view interface's dialog for creating a new (I, A)-privacy constraint.

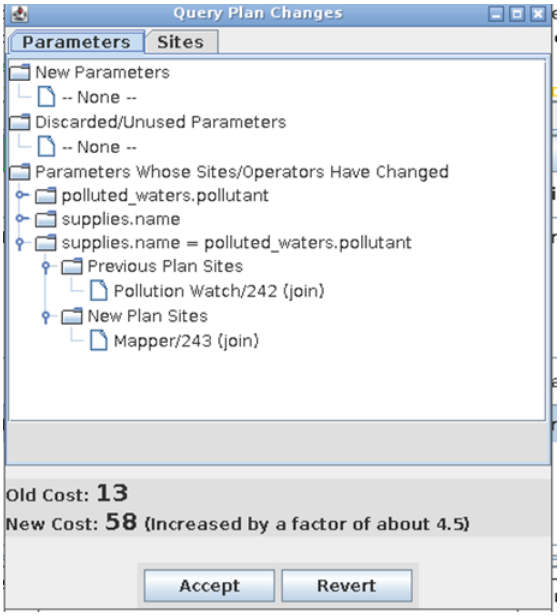


Figure 27: Our query view interface's display reporting the changes between a past query plan and the currently displayed query plan.

BIBLIOGRAPHY

[1] Performance Analysis Tools. http://wiki.postgresql.org/wiki/Performance_Analysis_Tools, May 2013.

[2] R. Agrawal, D. Asonov, M. Kantarcioglu, and Y. Li. Sovereign joins. In *ICDE*, 2006.

[3] R. Agrawal, A. Evfimievski, and R. Srikant. Information sharing across private databases. In *SIGMOD*, 2003.

[4] M. Ajtai. Oblivious RAMs without cryptogrpahic assumptions. In *STOC*, 2010.

[5] I. Altintas, C. Berkley, E. Jaeger, M. B. Jones, B. Ludäscher, and S. Mock. Kepler: An extensible system for design and execution of scientific workflows. In *SSDBM*, 2004.

[6] D. Atkins and R. Austein. RFC 3833: Threat analysis of the Domain Name System (DNS). http://tools.ietf.org/html/rfc3833, August 2004.

[7] A. Beimel and Y. Stahl. Robust information-theoretic private information retrieval. In *SCN*, 2002.

[8] A. Beimel and Y. Stahl. Robust information-theoretic private information retrieval. *J. Cryptology*, 2007.

[9] D. E. Bell and L. J. Lapadula. Secure computer system: unified exposition and Multics interpretation, 1976.

[10] R. Bellman. On the theory of dynamic programming. *Proceedings of the National Academy of Sciences*, 1952.

[11] S. Benabbas, R. Gennaro, and Y. Vahlis. Verifiable delegation of computation over large datasets. In *CRYPTO*, 2011.

[12] R. A. Botha and J. H. P. Eloff. Separation of duties for access control enforcement in workflow environments. *IBM Syst. J.*, 2001.

[13] M. Canim, M. Kantarcioglu, B. Hore, and S. Mehrotra. Building disclosure risk aware query optimizers for relational databases. *VLDB*, 2010.

[14] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *FOCS*, 1995.

[15] S. Clark, T. Goodspeed, P. Metzger, Z. Wasserman, K. Xu, and M. Blaze. Why (special agent) Johnny (still) can't encrypt: A security analysis of the APCO Project 25 two-way radio system. In *USENIX*, 2011.

[16] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2009.

[17] I. Damgård, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious RAM without random oracles. In *TCC*, 2011.

[18] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Authorization enforcement in distributed query evaluation. *JCS*, 2011.

[19] A. Deshpande and J. Hellerstein. Decoupled query optimization for federated database systems. In *ICDE*, 2002.

[20] K. Developers. Kepler project. http://www.kepler-project.org/, 2012.

[21] C. Devet and I. Goldberg. The best of both worlds: Combining information-theoretic and computational PIR for communication efficiency. In *PETS*, 2014.

[22] C. Devet, I. Goldberg, and N. Heninger. Optimally robust private information retrieval. In *USENIX*, 2012.

[23] T. Dierks and E. Rescorla. RFC 5246: The Transport Layer Security (TLS) protocol ver. 1.2. http://tools.ietf.org/html/rfc5246, August 2008.

[24] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*, 2006.

[25] C. Dwork, M. Naor, T. Pitassi, and G. N. Rothblum. Differential privacy under continual observation. In *STOC*, 2010.

[26] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 2007.

[27] 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*, 2011.

[28] N. L. Farnan, A. J. Lee, P. K. Chrysanthis, and T. Yu. PAQO: A preference-aware query optimizer for PostgreSQL. In *VLDB*, 2013.

[29] N. L. Farnan, A. J. Lee, P. K. Chrysanthis, and T. Yu. PAQO: Preference-aware query optimization for decentralized database systems. In *ICDE*, 2014.

[30] N. L. Farnan, A. J. Lee, and T. Yu. Investigating privacy-aware distributed query evaluation. In *WPES*, 2010.

[31] D. Ferraiolo and R. Kuhn. Role-based access control. In *NIST-NCSC*, 1992.

[32] M. J. Franklin, B. T. Jónsson, and D. Kossmann. Performance tradeoffs for client-server query processing. *SIGMOD Rec.*, 1996.

[33] M. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. In *EUROCRYPT*, 2004.

[34] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: outsourcing computation to untrusted workers. In *CRYPTO*, 2010.

[35] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, 1987.

[36] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.

[37] S. Goldwasser and S. Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 1984.

[38] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Practical oblivious storage. In *CODASPY*, 2012.

[39] GraphViz. GraphViz. http://www.graphviz.org/, July 2014.

[40] Information technology - database language SQL, 1992.

[41] W. Kießling. Foundations of preferences in database systems. In *VLDB*, 2002.

[42] W. Kießling and G. Köstler. Preference SQL: design, implementation, experiences. In *VLDB*, 2002.

[43] L. Kissner and D. Song. Privacy-preserving set operations. In *CRYPTO*, 2005.

[44] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 2000.

[45] D. Kossmann and K. Stocker. Iterative dynamic programming: a new class of query optimization algorithms. *ACM TODS*, 2000.

[46] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, 2012.

[47] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *FOCS*, 1997.

[48] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 1982.

[49] N. Li, T. Li, and S. Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and l-diversity. In *ICDE*, pages 106–115, 2007.

[50] B. Ludäscher, M. Weske, T. M. McPhillips, and S. Bowers. Scientific workflows: Business as usual? In *BPM*, 2009.

[51] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam. L-diversity: Privacy beyond k-anonymity. *ACM TKDD*, 2007.

[52] M. Mackall. smem memory reporting tool. http://www.selenic.com/smem/, May 2013.

[53] M. M. Masud and M. A. Hossain. Dynamic query plan for efficient query processing in peer-to-peer environments. *INFOCOMP Journal of Computer Science*, 2008.

[54] C. A. Melchor, B. Crespin, P. Gaborit, V. Jolivet, and P. Rousseau. High-speed private information retrieval computation on GPU. In *SECURWARE*, 2008.

[55] C. A. Melchor and P. Gaborit. A lattice-based computationally-efficient private information retrieval protocol. *IACR Cryptology ePrint Archive*, 2007, 2007.

[56] B. Momjian. Measuring free memory and kernel cache size on linux. http://momjian.us/main/blogs/pgblog/2012.html, May 2013.

[57] National Computer Security Center (NCSC). Glossary of computer security terms (NCSC-TG-04). http://csrc.nist.gov/publications/secpubs/rainbow/tg004.txt, October 1988.

[58] 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*, 2012.

[59] F. G. Olumofin and I. Goldberg. Privacy-preserving queries over relational databases. In *PETS*, 2010.

[60] F. G. Olumofin and I. Goldberg. Revisiting the computational practicality of private information retrieval. In *Financial Cryptography*, 2011.

[61] N. R. Ong, S. E. Rojcewicz, N. L. Farnan, A. J. Lee, P. K. Chrysanthis, and T. Yu. Interactive preference-aware query optimization. In *ICDE*, 2015.

[62] Oracle. Using optimizer hints. http://docs.oracle.com/cd/B19306_01/server.102/b14211/hintsref.htm, Oct 2013.

[63] R. Ostrovsky. Efficient computation on oblivious RAMs. In *STOC*, 1990.

[64] V. Papadimos and D. Maier. Distributed queries without distributed state. In *WebDB*, 2002.

[65] V. Papadimos, D. Maier, and K. Tufte. Distributed query processing and catalogs for peer-to-peer systems. In *CIDR*, 2003.

[66] F. Pentaris and Y. Ioannidis. Query optimization in distributed networks of autonomous database systems. *ACM TODS*, 2006.

[67] B. Pinkas and T. Reinman. Oblivious RAM revisited. In *CRYPTO*, 2010.

[68] P. Samarati. Protecting respondents' identities in microdata release. *IEEE TKDE*, 2001.

[69] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, 1979.

[70] R. Sion and B. Carbunar. On the practicality of private information retrieval. In *NDSS*, 2007.

[71] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: An extremely simple oblivious RAM protocol. In *CCS*, 2013.

[72] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *VLDB J.*, 1997.

[73] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A wide-area distributed database system. *VLDB J.*, 1996.

[74] L. Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(5), 2002.

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

[76] The PostgreSQL Global Development Group. Using EXPLAIN. http://www.postgresql.org/docs/9.1/static/using-explain.html, July 2014.

[77] S. Tran and M. Mohan. Security information management challenges and solutions. http://www.ibm.com/developerworks/data/library/techarticle/dm-0607tran/index.html, July 2006.

[78] L. Wang, D. Wijesekera, and S. Jajodia. A logic-based framework for attribute based access control. In *FMSE*, 2004.

[79] A. Whitten and J. D. Tygar. Why Johnny can't encrypt: a usability evaluation of PGP 5.0. In *USENIX*, 1999.

[80] P. Williams and R. Sion. Usable PIR. In *NDSS*, 2008.