Don't Reveal My Intension: Protecting User Privacy using Declarative Preferences during Distributed Query Processing

Nicholas L. Farnan[1], Adam J. Lee[1], Panos K. Chrysanthis[1], and Ting Yu[2]

[1] Department of Computer Science, University of Pittsburgh
[2] Department of Computer Science, North Carolina State University

Abstract. In a centralized setting, the declarative nature of SQL is a major strength: a user can simply describe *what* she wants to retrieve, and need not worry about *how* the resulting query plan is actually generated and executed. However, in a decentralized setting, two query plans that produce the same result might actually reveal vastly different information about the *intensional description* of a user's query to the servers participating its evaluation. In cases where a user considers portions of her query to be sensitive, this is clearly problematic. In this paper, we address the specification and enforcement of *querier* privacy constraints on the execution of distributed database queries. We formalize a notion of intensional query privacy called (I, A)-privacy, and extend the syntax of SQL to allow users to enforce strict (I, A)-privacy constraints or partially ordered privacy/performance preferences over the execution of their queries.

1 Introduction

Applications are increasingly becoming more decentralized, relying on data exchange between autonomous and distributed data stores. This reliance is in some cases a design decision motivated by the need for scalability; for example, when user demand for a service exceeds what a system at a single site would be able to provide and replication becomes a necessity. In other instances, it is a fact of life that must be dealt with. Such is the case when multiple partnering corporate entities wish to share data with one another, or when developers create "mashups" leveraging data gathered from multiple sources as a value-added service.

To date, the declarative nature of SQL has been one of its major strengths: a user can simply describe *what* her queries should retrieve and she need not worry about *how* they are converted into query execution plans and materialized. This has given rise to a rich literature concerning query optimization techniques that allow the system to explore relationally-equivalent representations of a user's query with the end goal of finding the *best possible* query execution plan (e.g., [9, 13]). Traditionally, the term *best* has been defined in reference to minimizing some combination of overall query execution time or data transmission requirements [13]. Unfortunately, optimizing purely for query performance can
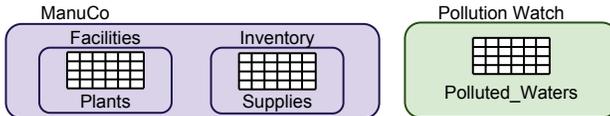
Fig. 1. Our example DB system. Three servers (`Facilities`, `Inventory`, and `Pollution Watch`), each store a single table (`Plants`, `Supplies`, and `Polluted_Waters`).

violate end-user privacy in a decentralized setting. Specifically, two query plans that produce the same result might actually reveal vastly different information about the intensional description of a user's query to the distributed servers participating in its evaluation.

Example 1 Consider Alice, a low ranking corporate executive, who wishes to investigate possible illegal pollution by her employer ManuCo. She could potentially do so by joining information stored in her employer's databases (e.g., records describing hazardous chemicals that are stored in the `Supplies` table on ManuCo's `Inventory` database server, and the locations of manufacturing plants stored in the `Plants` table on ManuCo's `Facilities` server, as in Fig. 1) with relevant information maintained by an environmental watchdog group (e.g., a database of polluted waterways). In issuing such a query, Alice would not want either party to become aware of the portion of her query that was issued to the other, or even that some part of her query was being evaluated by the other. Such a revelation could easily cost her her job, either because her employer felt that she "knew too much," or because the watchdog group applied external pressure to the company after learning of the intension of her query. ∎

This type of scenario clearly points to the need to protect end-user privacy during distributed query evaluation. In this paper, we strive to protect the privacy of a user *querying* a set of distributed databases. This is in contrast to most existing work on user privacy in database systems, which has focused on the privacy of users whose data is *contained* in a database [5, 15, 16, 21].

We believe that protecting the intension of user queries can be accomplished by modifying distributed query optimizers to optimize for query privacy in addition to query performance. We posit that through careful site selection and the use of a wide variety of query evaluation techniques (such as private information retrieval (PIR) [3, 14, 19]), query optimizers can be made to produce plans that effectively balance users' need for privacy with their desire for performance. In this paper, we work towards this goal by developing a formal framework for representing user preferences for privacy and performance on top of which such query optimizers can be built. As end-user privacy is an inherently personal and context-dependent notion that undoubtedly varies from user to user and query to query, our proposed approach allows users to express hard privacy constraints, establish preference relations between collections of possibly-competing privacy constraints, and explicitly balance the often competing interests of privacy-preservation and query efficiency. *It is our hypothesis that such*

a declarative model for preferences is a natural extension to declarative query languages such as SQL, and that a well defined semantics for balancing privacy and execution cost can be achieved using partially-ordered preference structures. By investigating this hypothesis, this paper makes the following contributions:

- We develop a formal definition of intension-based user privacy called (I, A)-privacy, which allows users to specify that some subset of the intension of their query I (which we will refer to as an "intensional region") is kept hidden from some set of adversarial principals A. (Sect. 3)
- We propose a syntax for augmenting SQL with the capability of expressing partially-ordered (I, A)-privacy constraints, which future *privacy-aware* distributed query optmizers can utilize to optimize for both privacy and performance. We further demonstrate that (I, A)-privacy is flexible enough to express PIR privacy constraints. (Sects. 4 and 5)
- We develop a preference algebra capable of prioritizing and balancing competing (I, A)-privacy constraints, thereby allowing users to build complex privacy profiles that can be applied to their future queries. This preference algebra also allows users to explicitly balance the often competing goals of efficient query processing and intensional privacy. (Sect. 5)
- To demonstrate the utility of (I, A)-privacy, we explore its expressive power and show that it is flexible enough to encode common idioms such as discretionary access control policies, mandatory access control policies, separation of duty constraints, PIR constraints, and data provenance preferences (presented in Appendix D).
- We develop a proof of concept implementation of our model for identifying sensitive aspects of a query plan done with XPath and XML, and further discuss how our proposed system could be implemented in existing query processors (presented in Appendix E).

It should be noted that the specific notion of (I, A)-privacy explored in this paper is syntactic, in that we define (I, A)-privacy to be violated only if some sensitive piece of query intension is directly revealed to another principal. However, the definition of (I, A)-privacy presented is sufficiently general to support more semantic notions of end user privacy. Such a semantic definition for the underpinnings of (I, A)-privacy is the subject of future work.

In the next section, we describe our system and threat model. In Sects. 3-6, we present our contributions. We overview related work in Sect. 6, and finally, discuss future work in Sect. 7.

2   Background and Assumptions

Here we will give a brief overview of query processing and relational algebra, and then present the assumptions we make for this work.

Query Processing and Relational Algebra The basic processing of a user-input 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 work we deal exclusively with SQL) 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 work), this representation is a tree of *relational algebra* operators leafed by read operations on the source database relations for the query.

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*, which returns only tuples from the input relation that match some given selection criteria; *projection*, which reduces the arity of the tuples it processes by eliminating unwanted attributes; *Cartesian product*, which returns all possible combinations of tuples from two input relations; *rename*, which changes the labels of the components of the tuples it processes; *set union*; and *set difference*. One notable operator that can be defined in terms these primitives operators is *join*. *Join* combines two input relations using selection criteria, 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 to ensure its efficient evaluation by the database engine(s). After being sufficiently optimized, the query is transformed yet again to a representation that can be evaluated by the database engine through *code generation*. Finally, the result of the query is realized through *plan execution*. For a more in-depth review of query processing and relational algebra, we refer the reader to [6].

System Model We tackle the problem of protecting the privacy of users who issue queries against a distributed system of autonomous relational database servers. In this paper, we assume the distributed system to be comprised of a countable set $\mathcal{P}$ of principals (i.e., servers and users) cooperating to process distributed queries. Each principal participating in the system as a *server* may host its own data tables locally, as well as export views defined in terms of queries over its own data tables, or tables/views hosted at other servers. When a server exports a view, it is not required to also export the definition of this view, which it may consider to be private [10]. Further, data tables may be replicated across multiple servers. Hence, in addition to exchanging meta-data (e.g., exported schemas and statistics) for query planning and optimization, servers inter-operate to maintain consistency across any replicated data tables.

Users are the subset of principals that issue queries within the system. A user is not assumed to have complete trust in every server that may see some part of her query, but is assumed to have (at least) one trusted server responsible for planning and executing queries on her behalf. Trivially, this trusted server—which we refer to as a *querier*—might be operated by the user herself on her local machine. Upon receiving an SQL query from a user, the querier uses its locally-available meta-data to generate an execution query plan derived from the

user's query. It then schedules the execution of the query plan by decomposing it into sets of sub-plans by projecting on the servers selected to execute each operation in the query plan, while also making explicit the communication flow of tuples between sub-plans materialized by different servers. Finally, the querier dispatches the sub-plans to their corresponding servers, materializes its part of the query, and prepares the final result to return to the user. Note that a participating server may further expand and optimize their sub-plans, e.g., due to the expansion of remote views. As such, query evaluation is a recursive process that may traverse many servers.

Threat Model We assume that each server controls access to the data stored within its tables and views using Role-Based Access Controls [8] defined at the table or view level, as implemented by most commercial database systems [23]. We further assume that any pair of principals $p_1, p_2 \in \mathcal{P}$ have the ability to set up a private and authenticated channel (e.g., a TLS tunnel [4]) to protect against eavesdropping and message modification, reordering, and replay attacks. We assume that all servers in the system (aside from the querier) to be *honest-but-curious* passive adversaries. Specifically, each server will correctly execute the query evaluation protocol, but may try to infer information regarding the user's original query from the sub-plan that it received. Furthermore, a set of colluding servers may work together to make inferences based upon their combined collection of sub-plans.

The honest-but-curious adversary model makes sense within the context of our work, given that we focus on preserving user privacy and not on ensuring the correctness of query results. Finally, although we assume a colluding adversary model, we do not consider the notion of a global passive adversary. That is, all information used by colluding servers to uncover querier intension must be gathered exclusively from the execution of distributed queries.

3 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, our focus lies in the protection of end-user privacy during the optimization and execution of distributed queries. In this section, we develop the notion of (I, A)-privacy, in which intensional regions of a user's query that the user considers to be sensitive are protected from disclosure to a set of colluding adversaries. While query intension is generally represented using a declarative language like SQL (and, indeed, knowledge of SQL is all that users would require in order to author protections over the privacy of such intension through the use of our model), we focus in this paper 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 gleaned by adversaries according to our threat model.

Definition 1 (Query Plan) *A* query plan $Q = \langle N, E \rangle$ *is a directed, acyclic, and fully-connected graph with a single root where* $N \subseteq \mathcal{N}$ *and* $E \subseteq N \times N$.

$\langle join, \{(pullutant, =, name), (AND), (location, =, location)\}, localhost\rangle$

$\langle scan, \{(Polluted\_Waters)\}, PollutionWatch\rangle$ $\langle join, \{(id, =, plant\_id)\}, localhost\rangle$

$\langle scan, \{(Plants)\}, Facilities\rangle$ $\langle select, \{(type, =, \text{“}solvent\text{”})\}, Inventory\rangle$

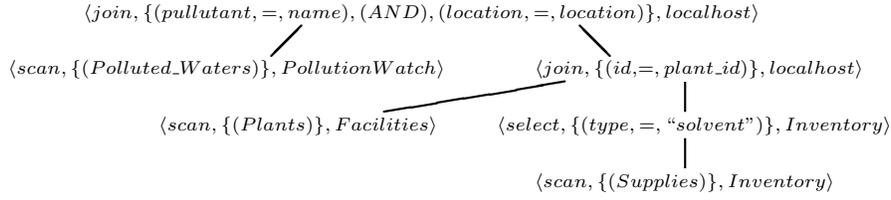$\langle scan, \{(Supplies)\}, Inventory\rangle$

Fig. 2. Query plan for Alice's query. The text in each node is a comma-seperated list of the operation type of that node, the arguments to that operation, and the execution location of that node.

An element n of the node set $\mathcal{N}$ is a ternary $n = \langle op, params, p \rangle$ that describes a relational operator (op), the parameters to this operator (params), and the principal at which this operator is scheduled to be executed (p). The edge set E describes the producer/consumer relation between relational operators.

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

Assuming that $\mathcal{S}$ denotes the set of all SQL queries, we can then formally represent a query planner as a function $\mathsf{plan} : S \to \mathcal{Q}$ that generates query plan $Q \in \mathcal{Q}$ from an SQL query $q \in \mathcal{S}$. Fig. 2 is a graphical representation of a well-formed query plan corresponding to Alice's query from our motivating example in Sect. 1. We now look at two specific types of query plans:

Definition 3 (Locally-Expanded Query Plan) *A well-formed query plan Q is said to be* locally-expanded *with respect to a principal $p \in \mathcal{P}$ if every leaf node ℓ of Q represents either (i) an operation annotated for execution at some remote principal $p' \neq p$, or (ii) the scan of a table managed locally by the principal p. We denote the set of all locally-expanded query plans for a query $q \in \mathcal{S}$ as $\mathscr{L}(q)$.*

Note that the term *scan* is used to denote the retrieval of tuples from a database relation without specifying a specific access method (e.g., the use of a particular index). Given a query $q \in S$, the output of a query planner plan at a specific principal p is a locally-expanded query plan $Q \in \mathscr{L}(q)$ with respect to the principal p. 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 4 (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 principal $p \in \mathcal{P}$. We denote the set of all globally-expanded query plans for a query $q \in \mathcal{S}$ as $\mathscr{G}(q)$.*

Note that a globally-expanded query plan is trivially a locally-expanded query plan relative to every principal $p \in \mathcal{P}$. That is, no principal 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 principal in the system: each principal 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 query, 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 5 (Intensional Region) *An* intensional region *is a countable subset I of the node set $\mathcal{N}$.*

Example 2 To illustrate this concept, we can refer back to Alice's query plan from Fig. 2. Consider the case in which Alice wants to keep the selection criteria (i.e., solvents) used over the `Supplies` table private from all principals involved in the processing of her query. An intensional region which contains the single select 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 also flexible enough to identify broader descriptions of intension considered to be sensitive like "all selection operations that may be executed at the human resources server." Given such intensional regions, we can informally say that a query plan Q maintains a user's privacy if no adversarial principal can learn the 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 6 (Intensional Knowledge) *Given a globally-expanded query plan $Q = \langle N, E \rangle$, we denote by $\kappa_p(Q) \subseteq N \cup E$ the* intensional knowledge *that principal $p \in \mathcal{P}$ has of the query encoded by the plan Q. At a minimum, $\kappa_p(Q)$ contains the set of all locally-expanded query plans for each node $n \in N$ annotated for execution by the principal p, and further all edges leaving or entering such nodes.*

Given a colluding set of principals $P = \{p_1, \ldots, p_k\}$, we can define the combined intensional knowledge of the colluding principals P in the natural way, i.e., $\kappa_P(Q) = \bigcup_{p_i \in P} \kappa_{p_i}(Q)$. For example, the intensional knowledge of the three other principals participating in the evaluation of the query in Fig. 2 can be broken down as follows:

$$\begin{array}{ll} \kappa_{PollutionWatch}(Q) & \langle \mathtt{scan}, \{(\mathtt{Polluted\_Waters})\}, \mathtt{Pollution\ Watch} \rangle \\ \kappa_{Facilities}(Q) & \langle \mathtt{scan}, \{(\mathtt{Plants})\}, \mathtt{Facilities} \rangle \\ \kappa_{Inventory}(Q) & \langle \mathtt{select}, \{(\mathtt{type}, =, \text{``solvent''})\}, \mathtt{Inventory} \rangle \\ & \text{and } \langle \mathtt{scan}, \{(\mathtt{Supplies})\}, \mathtt{Inventory} \rangle \end{array}$$

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

Definition 7 ((I, A)-privacy) *Given an intensional region I and a set of colluding adversaries $A \subseteq \mathcal{P}$, 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 paper, we will focus our attention on the syntactic condition of the containment relation $\sqsupseteq$ defined below. 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.

Definition 8 ($\sqsupseteq$) *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 $N \cap I \neq \emptyset$.*

We feel that 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 in Sect. 4.2, and a formal proof of it is presented in Appendix C.

In the next section, we will present extensions to SQL that allow users to concisely identify the intensional regions considered sensitive within queries that they may issue. Sect. 5 then shows how privacy requirements over these regions can be combined using partially-ordered preference structures to develop a flexible formal foundation for providing (I, A)-privacy in distributed database systems.

4 Matching Query Plan Nodes

In this section, we will first illustrate how users can use the node matching in order to specify constraints on the evaluation of a query. We then formalize our method for identifying intensional regions through the use of descriptors of nodes in N, and a matching operator, $\pitchfork$.

4.1 Matching Syntax

In order for users to be able to express their notions of privacy to a query planner when issuing a query, we propose a REQUIRING clause (REQUIRING ...

`HOLDS OVER ...`) as an extension to SQL. We are proposing 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`). In the case that 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 Appendix A.

This clause allows a user to identify one or many intensional regions that match against *node descriptors* in a *match statement*, and further specify *constraints* over how nodes in those intensional regions are treated over the course of executing a query. Node descriptors can be considered similar to regular expressions in that they specify a pattern to be matched against nodes. They are made up of descriptors of the same three attributes present in query plan nodes in $\mathcal{N}$: *op*, *params*, and *p*. They are differentiated from query plan nodes in that the values of these attributes do not have to be grounded, they could instead be expressed as a wildcard (*) or free variable (which we prefix with a "$"). Constraints can then be expressed as statements over the values of the free variables present in corresponding node descriptors. These constraints can be used to express (I, A)-privacy constraints by defining an intensional region with a node descriptor, and constraining all nodes matching that description such that they can not be executed by principals in A. Constraints can be tests for any of the following: equality ($=$), inequality ($\neq, <, >, \leq, \geq$), or set membership ($\in$).

Example 3 Given the ability to express clauses of the form shown in Appendix A as a part of her SQL queries, Alice could have written her query shown in Fig. 2 to ensure that any operation that she considers private is executed by her trusted server. The adversarial set in this example would hence consist of all principals in the system aside from the querier, denoted as `localhost`.

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

Such a query would uphold (I, A)-privacy by keeping an intensional region consisting of all nodes operating on the constant "solvent" from being disclosed to any remote database servers, as any nodes in that intensional region would be constrained to be executed at `localhost`. Note that the query plan shown in Fig. 2 does not adhere to this requirement, and would not be an acceptable query plan for this new query. ∎

Examples of the expressive power of the `REQUIRING` clause are given in Appendix D by showing how it can be used to express common policy idioms.

4.2 Matching Operator

While we have presented the practical applications of node descriptors, we now formally define node descriptors:

Definition 9 (Node Descriptor) *A node descriptor $d \in \mathcal{D}$, is a triple $d = \langle op, params, p \rangle$ describing the relational operator (op), a set of parameters to the given operator (params), and the principal at which this operator shall be executed (p). A node descriptor is well formed if all of the following hold true:*

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

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

Definition 10 (Matching) *Given a node descriptor $d \in \mathcal{D}$, and a node $n \in N$, $d \pitchfork n$ if and only if*

$$(d.op = n.op \vee d.op \in \mathcal{V} \vee d.op = *) \ \wedge \ (d.p = n.p \vee d.p \in \mathcal{V} \vee d.p = *) \ \wedge$$

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

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

While matching on *op* or *p* is rather intuitive, matching on *params* requires a bit of explanation. *params* is a set of sets in both 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 node based on *params* if every ordered set in the descriptor's *params* is either contained directly in the node's *params* or is a subset of an ordered set in the node's *params*.

Example 4 Our definition of $\pitchfork$ allows nodes with complex *params* attributes, such as $\langle$ select, { (at1, =, 42), (AND), (at2, <, 10) }, example_principal $\rangle$, to be easily matched. Any of the following node descriptors will match this node based on our definition of $\pitchfork$: $\langle$ *, { (at1) }, * $\rangle$; $\langle$ *, { (at1, =, 42) }, * $\rangle$; $\langle$ *, { (at1, 42) }, * $\rangle$; $\langle$ *, { (AND) }, * $\rangle$; $\langle$ *, { (at1, 42), (AND) }, * $\rangle$; $\langle$ *, { (at1), (AND), (at2) }, * $\rangle$ ∎

As has been previously stated, privacy is an inherently personal property, and hence, we must ensure that our method of specifying node descriptors and matching nodes against them 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 $q \in \mathcal{S}$, 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. This proof is included in Appendix B. Furthermore, 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 $\pitchfork$ operator.*

Since it is possible to specify a set of node descriptors $D \subseteq \mathcal{D}$ such that the matching construct $\pitchfork$ 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 $\pitchfork$ is sufficiently expressive to encode any (I, A)-privacy constraint.*

As should now be apparent, (I, A)-privacy is capable of expressing a wide variety of user privacy constraints. One set of such constraints that we would like to highlight (as alluded to in Sect. 3) is that offered by private information retrieval (PIR). The problem addressed by PIR is formulated as follows: given some k servers that store replicated copies of a database that is viewed as a length n binary string $x = x_1 \ldots x_n$, 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 [3]. Initially, k was specified to be $k \geq 2$, however [14] details a technique for achieving PIR from a single computationally-bounded server.

In essence, 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). 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 (I, A)-privacy to express PIR privacy constraints on their queries. This notion is precisely stated in the following theorem, and further formally proven in Appendix C.

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

4.3 Constraining Multiple Node Descriptors

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. Node descriptors have a dual-purpose in that they not only serve to identify nodes (through their grounded attributes), but also establish which attributes of the nodes in

the intensional region that they identify can be constrained. 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 for the values of that node which correspond to the free variables in the node descriptor.

When a condition is specified over multiple node descriptors, however, we allow for two possibilities. Either all node descriptors use the same free variable, or different variables are used in different descriptors. In the case that the same variable is used, it must be ensured that the constraint holds for any node matching any of the descriptors. This is essentially a shortcut for writing multiple identical conditions for different node descriptors. In the case that different variables are used, however, ensuring that a condition holds over a query plan is slightly more complicated, as it must be ensured that for all combinations of nodes that match the independent descriptors, the condition holds. Examples such node descriptor/condition pairs are shown in Appendix D.

5 Preference Algebra

The syntactic and logical constructs defined in Sect. 4 are sufficient for upholding strict (I, A)-privacy requirements that users may define for their queries. However, users may need to consider the enforcement of many potentially-competing privacy constraints, or explicitly balance the desire for private query evaluation with the real-life performance implications of private query evaluation techniques. In this section, we develop a formal preference algebra that allows users to establish complex preference structures over the privacy preservation and performance characteristics of query plans generated from their SQL queries.

5.1 Background

In [11, 12], 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 [11, 12] 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 [11] define a preference as follows:

Definition 11 (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 [11] for descriptions of a range of other numeric and non-numeric base preference constructors.

Similarly, [11] 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:

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

Complex preferences in which one preference is strictly more important than the other can be defined using the *prioritized preference* operator &. The definitions of other complex preference constructors can be found in [11].

5.2 Preferences for Query Plan Execution

We now extend the above preference formalism to enable the expression of preferences over the (I, A)-privacy properties and performance characteristics of a query plan. To establish preferences over query plans—rather than relational tuples—we must redefine the notion of preferences to operate over a set of *query plan evaluation functions* $\mathcal{F}$. Functions within this set might, for instance, check a query plan's compliance with an (I, A)-privacy condition, evaluate the predicted runtime of a query plan, or estimate the amount of data that will need to be transmitted during the evaluation of a query plan. To allow preferences to be specified over (I, A)-privacy conditions, $\mathcal{F}$ contains at least the function check : $\mathcal{Q} \times 2^{\mathcal{D}} \times \mathcal{C} \rightarrow \mathbb{B}$, where $2^{\mathcal{D}}$ is the power set of all node descriptors D. We define check as follows:

Definition 12 (check) *Let $Q = \langle N, E \rangle$ be a query plan, D be a set of node descriptors, c be a constraint over the free variables describing node locations in D, A be the set of principals not permitted to be assigned to a free variable in a node descriptor in D by the constraint c, and $I = \{n \mid n \in N \wedge \exists d \in D : d \pitchfork n\}$ be the intensional region of Q matched by the set D of node descriptors. Now, check$(Q, D, c) \leftrightarrow \kappa_A(Q) \not\models I$.*

That is, check examines whether a given query plan upholds (I, A)-privacy as defined by a particular node descriptor, constraint, and inference operator $\models$.

Example 5 Consider the query plan shown in Fig. 2. If we refer to this query plan as Q, then check$(Q, \{\langle *, \{(\text{"solvent"})\}, \$p \rangle\}, \$p = \texttt{localhost})$ evaluates to $\texttt{false}$ (when using $\sqsubseteq$ for $\models$), as the SELECT statement is executed at the Inventory Server, not Alice's trusted server. Similarly, check$(Q, \{\langle \text{join}, *, \$p \rangle\}, \$p = \texttt{localhost})$ evaluates to $\texttt{true}$, since all joins happen on Alice's server. $\blacksquare$

In order to allow users to balance (I, A)-privacy preferences with the estimated performance of query, we can augment $\mathcal{F}$ with additional functions that estimate the performance characteristics of a query plan. For instance, we could include a function runtime : $\mathcal{Q} \to \mathbb{R}$ that associates a query plan with its estimated runtime (e.g., in seconds). Given such a set $\mathcal{F}$ of query plan evaluation functions, we can formally define a query plan preference as follows:

Definition 13 (Preference $P = (\mathcal{F}, <_P)$) *Let $\mathcal{F}$ be a set of query plan evaluation functions, and let $V = \{range(f) \mid f \in \mathcal{F}\}$. A query plan preference P is a strict partial ordering $P = (\mathcal{F}, <_P)$ where $<_P \subseteq V \times V$.*

Preferences over quantitative evaluations of the *performance* of a query plan can be easily defined through the use of the base numerical preference constructors presented in [11] (e.g., LOWEST). However, the expression of preferences over (I, A)-privacy constraints requires a new base preference constructor, which we will call HOLD to mirror the HOLDS OVER keywords presented in Sect. 4.1.

Definition 14 (HOLD(Q, D, c)) *Given a query plan Q, a set of node descriptors D, and a constraint c over free variables declared node descriptors in D, it is preferable for check(Q, D, c) to evaluate to `true` as opposed to `false`.*

Complex preferences over both privacy and performance can now be constructed using base numerical preference constructors and the above defined HOLD constructor in addition to the complex preference constructors presented in [11].

Example 6 Alice considers it of paramount importance for her query to run in the least amount of time, but also prefers that all join operations and any operations involving the constant "solvent" be executed by her trusted query processing software, which runs on `localhost`. She considers the latter two preference to be equally desirable, but less desirable than her runtime preference. This complex preference can be represented as follows:

$$LOWEST(runtime) \ \& \ (HOLD(q, \langle *, \{(\text{``solvent''})\}, \$p\rangle, \$p = localhost)$$
$$\otimes \ HOLD(q, \langle join, *, \$p\rangle, \$p = localhost)) \qquad \blacksquare$$

Since query plan preferences are defined over the range of *every* function in $\mathcal{F}$, complex preference can be expressed over base preference operators with differing types. This allows preferences to be established between multiple (I, A)-privacy constraints, between multiple performance constraints, or between a mix of (I, A)-privacy and performance constraints (as in the above example).

5.3 Preference Syntax

In order to enable users to make use of the query plan preference constructs defined in Sect. 5.2, we now describe another extension to SQL, the PREFERRING clause —modeled after the extensions proposed in [12]— that similarly applies to both SELECT statements and set operators. We maintain the same notation

for preferences over the results of functions with numeric ranges in that such preferences are stated explicitly through the base preference constructor that is to be used and the name of the function that a preference is to be expressed over. We further maintain the use of the keyword `AND` to represent the complex preference constructor $\otimes$ and `CASCADE` to represent the complex preference constructor &. In the case of expressing privacy preferences over intensional regions, however, this clause will take a similar form to the `REQUIRING` clause presented in Sect. 4.1. The full syntax of the `PREFERRING` clause is presented in Appendix A. To demonstrate the use of this syntax for including query plan preferences within a query, we now consider the following example the combines strict requirements on a query plan with more flexible preferences:

Example 7 Assume that Alice *requires* that any nodes whose operations make use of the constant "solvent" must be annotated for execution at `localhost`. Further, she *prefers* to execute joins on `localhost` as well, given that they do not increase the execution time of her query. 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 Polluted_Waters.location = Plants.location,
    AND Plant.id = Supplies.plant_id
REQUIRING $p = localhost HOLDS OVER <*, {("solvent")}, $p>
PREFERRING LOWEST{runtime}
    CASCADE $p = localhost HOLDS OVER <join, *, $p>;
```
∎

The notion of query plan preferences can be used to express a wide range of privacy and performance constraints over the execution of user queries. We present examples of common policy idioms encoded using this approach in Appendix D.

5.4 Implementation Considerations

A practical method implementing an (I, A)-privacy aware query optimization would be the direct inclusion of our model in an existing query optimizer. While this work is not focused on the implementation of the model which we describe, here we briefly discuss how the changes that would have to be made to a query optimizer would affect its performance.

Such an implementation could be accomplished with little modification to existing dynamic programming based query optimizers by maintaining separate lists of plan costs for each set of plans that is equally preferred throughout the course of optimization. While this may result in an increase in query optimization time, it should be noted that such lists need only be maintained for query execution restrictions in `PREFERRING` clauses. Restrictions from `REQUIRING` clauses, on the other hand, could be utilized to speed up query optimization by pruning query plans from the search space that do not uphold them. Hence, users making use of only `REQUIRING` clause constraints can only lessen the time that is required to optimize their queries in the average case. Optimization time penalties are only incurred by users wishing to express complex controls over the intension of

their queries. Given the expressive power that our model affords users, however, we feel it is quite reasonable to assume that in these cases they will be willing to accept this increased optimization time.

6 Related Work

We now discuss areas of closely related work: private information retrieval and distributed query processing.

Private Information Retrieval The problem that PIR techniques work to solve was described in Sect. 4.2. PIR was originally proposed in [3] through an approach that required multiple non-colluding servers to host replicas of the database that a user wished to access. This approach has come to be known as information-theoretic PIR. PIR using only a single server was established using computational techniques in [14]. Though the practical feasibility of computational PIR has been called into question [22], such issues can be assuaged through the use of either secure co-processors [25], or general purpose computing on graphics processing units (GPGPU) [17]. Further, a method for performing information-theoretic PIR over widely implemented database access methods (hash indices and B^+ tree indices) was recently demonstrated [19].

Systems implementing (I, A)-privacy support would be able to utilize any of these techniques in the special case that (i) the user specifies privacy constraints on the execution of her query that can be achieved through the use of PIR and (ii) 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 only rudimentary query processing capabilities (specifically those outlined in Sect. 2), while still providing the capability for users to take advantage of more advanced techniques such as PIR, when they are available and applicable.

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) [13]. These techniques can further be combined as a form of hybrid shipping [9]. Mutant query plans [20] can also provide a form of combined shipping that allows asynchronous query evaluation. In the same manner that our model proposed here is able to glean the advantages of PIR when possible, it can also utilize all of the these query processing techniques to construct query plans that sufficiently balance user privacy preferences with query performance, realizing the proposed hybrid query processor from [7].

7 Conclusions and Future Work

In this paper, we have formally defined a notion of intension-based query privacy called (I, A)-privacy. This type of privacy is designed to allow the user querying

a database to express constraints on the portions of her intensional query that should not be leaked to the servers involved in executing her query. We have further shown that private information retrieval is a special case of (I, A)-privacy and can thus be used as a building block for systems seeking to preserve certain types of (I, A)-privacy. We have presented a framework for representing complex user preferences balancing query privacy and performance. Further, we developed a syntax for extending SQL to express such preferences.

Future work will first and foremost include the implementation of our framework within a distributed query optimizer. Future work will also investigate the expression of preferences not only over the dissemination of query plan metadata, but also the flow of extensional query results over the course of query execution. We will explore other possible relations for the $\models$ operator that could take into account both extensional flows of query results and a more powerful adversarial model, and further include semantic notions of end user privacy.

References

1. D. E. Bell and L. J. Lapadula. Secure computer system: unified exposition and multics interpretation, March 1976.
2. R. A. Botha and J. H. P. Eloff. Separation of duties for access control enforcement in workflow environments. *IBM Syst. J.*, 40:666–682, March 2001.
3. B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *FOCS*, 1995.
4. T. Dierks and E. Rescorla. Rfc 5246: The transport layer security (tls) protocol version 1.2, August 2008.
5. C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*, 2006.
6. R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* Addison-Wesley, 2007.
7. N. L. Farnan, A. J. Lee, and T. Yu. Investigating privacy-aware distributed query evaluation. In *WPES*, 2010.
8. D. Ferraiolo and R. Kuhn. Role-based access control. In *NIST-NCSC*, 1992.
9. M. J. Franklin, B. T. Jónsson, and D. Kossmann. Performance tradeoffs for client-server query processing. *SIGMOD Rec.*, 25:149–160, 1996.
10. Information technology - database language sql, 1992.
11. W. Kießling. Foundations of preferences in database systems. VLDB, 2002.
12. W. Kießling and G. Köstler. Preference sql: design, implementation, experiences. VLDB, 2002.
13. D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.
14. E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *FOCS*, 1997.

15. N. Li, T. Li, and S. Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and l-diversity. In *ICDE*, 2007.

16. A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam. L-diversity: Privacy beyond k-anonymity. *ACM TKDD*, 1(1):3, 2007.

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

18. N. C. S. C. (NCSC). "glossary of computer security terms" (ncsc-tg-04). `http://csrc.nist.gov/publications/secpubs/rainbow/tg004.txt`, October 1988.

19. F. G. Olumofin and I. Goldberg. Privacy-preserving queries over relational databases. In *PET*, 2010.

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

21. P. Samarati. Protecting respondents' identities in microdata release. *IEEE TKDE*, 13:1010–1027, November 2001.

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

23. 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.

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

25. P. Williams and R. Sion. Usable pir. In *NDSS*, 2008.

A   SQL Extension Syntax

It should be noted that in the following syntax we leave <literal> ungrounded. We use <literal> as a placeholder for strings of characters which could represent either names of free variables, constants (e.g., "solvent"), relation names, relational attribute names, or principals in the system (e.g., `Facilities`, `localhost`).

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


<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>


  <dlist> ::= <dnode> ["," <dlist>]
  <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> | "*"
```

Fig. 3. Syntax for the REQUIRING and PREFERRING clauses

B Proof of Theorem 1

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 ⇑, 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:

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

By the definition of ⇑, 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:

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

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

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

By the definition of ⇑, 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` `*` 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:

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

By the definition of ⇑, 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 Sect. 4.1, 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`):

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

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 ⇑, 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 indentifies 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 ⇑, 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 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`):

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

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 ⋔, 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 trivailly be identified by one of the following node descriptors:

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

In the above, 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 names (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` attribute), 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:

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

By the definition of ⋔, the above (simple) node descriptor matches exactly the set of all occurances 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.

C Proof of Theorem 4

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>,
                                   <*, {(=)}, $p>,
                                   <*, {(const)}, $p>;
```

The above query specifies the following set of node descriptors:

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

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 constraint `$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 Definition 8, we have that $\kappa_S(Q) \not\sqsupseteq I$, and thus q' preserves (I, A)-privacy (by Definition 7). Thus, the notion of (I, A)-privacy can be used to encode the above (and therefore any) PIR constraint.

D Expressive Capabilities

The examples that we have presented in the body of this work 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 work 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 paper.

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 [18]. 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 the paper have encoded very specific DAC policies restricting access to intensional regions to just the querier. A user could just as easily have required that certain intensional regions be executed by some 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 `REQUIRING` block would force all matching query nodes be evaluated by some server in the set $\{P, Q, R\}$ of trusted servers.

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 [1] 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 allow the user 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 preference framework articulated in this paper can allow MAC and DAC constraints to co-exist, as is often the case in environments using of MAC constraints [1].

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 [24]. 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.

Separation of Duty Separation of duty (SoD) policies are used to require that multiple principals cooperate to carry out a particular action [2]. 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 by our model through the use of a single constraint over multiple node descriptors:

```
REQUIRING $p1 != $p2 HOLDS OVER
    <scan,{("Salary")},$p1>, <scan,{("EmploymentHistory")},$p2>
```

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

Data Source Preference In addition to privacy-related constraints, the preference model developed in this paper 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 $p = P HOLDS OVER <scan,{(A)},$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.

E XPath-based Prototype

In order to concretely demonstrate that policies specified in `REQUIRING` and `PREFERRING` statements are easily enforced, we have developed a proof-of-concept implementation using XML and XPath which can be found on the web[3]. With XML, we are able to represent query plans (trees of nodes) as a heirarchy of XML tags, and then identify specific tags within that tree through the use of XPath.

---

[3] `http://www.cs.pitt.edu/~nlf4/xml_prototype.html`

Example 8 We could represent a select operation on the result of a scan of some table T on the condition that an attribute a is equal to the constant 10 as:

```
<select>
  <execution_location> localhost </execution_location>
  <constraint>
    <loperand>
      <attribute> a </attribute>
    </loperand>
    <operator> = </operator>
    <roperand>
      <constant> 10 </constant>
    </roperand>
  </constraint>
  <scan>
    <execution_location> http://thost.example.com </execution_location>
    <database> t_database </database>
    <table> t </table>
  </scan>
</select>
```

We could then check that this query plan upholds the constraint that any operation that expresses a constraint on the attribute a is executed at `localhost` ($\langle$ *, { (a) }, * $\rangle$) with the following XPath query:

```
//*[constraint//attribute=\"a\" and execution_location!=\"localhost\"]
```

As the above query plan does not contain an XML tag that matches this XPath query, the query plan upholds the constraint. If any tags were to match this XPath query, it would not uphold the constraint. ■

This implementation is, however, somewhat limited in that it has no notion of query performance, and further cannot process constraints over multiple node descriptors.

Though we present this implementation purely as a proof of concept, it could be utilized to protect query intension given a user already had a complete enumeration of all possible query plans for a given query. In such a situation, a user could convert these query plan to XML, convert any constraints and node descriptors for that query to XPath, and check, for each XPath expression, which query plans contain no nodes matching against it (we strive for empty results here as we have defined XPath expression to contain a representation of node descriptors and the negation of a constraint, hence any matching portions of an XML query plan indicate a violation of a user preference). After weeding out query plans that violate a required constraint, the remaining query plans could then be ranked according to which preferred constraints they uphold. The top ranked results are then candidates for execution.

An implementation of this style, however, is rather naive. As was stated, it requires the user to have an enumeration of all possible query plans for a given query, which would be incredibly time-consuming where it is not intractable all together. We do not present this as a model for actual implementations. We present this prototype simply for the purposes of showing that our the matching and constraint constructs put forth in this paper are *enforceable*.