Investigating Privacy-Aware Distributed Query Evaluation

Nicholas L. Farnan[†]
nlf4@cs.pitt.edu

Adam J. Lee[†]
adamlee@cs.pitt.edu

Ting Yu[‡]
yu@csc.ncsu.edu

[†]Department of Computer Science, University of Pittsburgh
[‡]Department of Computer Science, North Carolina State University

ABSTRACT

Historically, privacy and efficiency have largely been at odds with one another when querying remote data sources: traditional query optimization techniques provide efficient retrieval by exporting information about the intension of a query to data sources, while private information retrieval (PIR) schemes hide query intension at the cost of extreme computational or communication overheads. Given the increasing use of Internet-scale distributed databases, exploring the spectrum between these two extremes is worthwhile. In this paper, we explore the degree to which query intension is leaked to remote data sources when a variety of existing query processing and view materialization techniques are used. We show that these information flows can be quantified in a concrete manner, and investigate the notion of privacy-aware distributed query evaluation. We then propose two techniques to improve the balance between privacy and efficiency when processing distributed queries, and discuss a number of interesting directions for future work.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed databases*; H.2.4 [**Database Management**]: Systems—*distributed databases, query processing*; K.4.1 [**Computers and Society**]: Public Policy Issues—*privacy*

General Terms

Security, Performance

Keywords

privacy, distributed query processing, p2p, database, mutant query plan

1. INTRODUCTION

The widespread use of database management systems coupled with the explosive growth of the Internet have brought about the capability for vast amounts of information stored at independent sites to be accessed collectively by individual users. Turning this capability into a usable system, however, has forced the development of new models for distributed query processing. Traditional distributed database applications make use of a global schema that would be infeasible to maintain at Internet-scale and currently deployed peer-to-peer (P2P) systems lack the rich querying options that database users rely upon, offering instead only information-retrieval styled string matching and containment.

In response to this state of affairs, there is a growing body of work that aims to overcome these difficulties in order to make a distributed computing platform that can truly take advantage of the resources available to it. Work on using schema matching to evaluate queries in traditional distributed databases has been adapted to function in pure P2P systems [2]. There have also been attempts to work in the opposite direction by simply adapting existing P2P technologies (e.g., BitTorrent [7]) to handle more complex queries. A truly novel solution to the problem has also been proposed in the form of mutant query plans, which allow for nearly stateless query processing, in-network reoptimization, and distributed view materialization [16, 17].

In the process of providing the ability to query multiple autonomous and heterogeneous data sources, these approaches have also drastically changed the flow of information during the query evaluation process. In traditional approaches to distributed query processing, the queries issued to a site concern *only* data local to that site. These new schemes, however, spread more information about the overall query to more sites in the system in order to provide the desired functionality. The listed schema-matching techniques may require a (sub-)query to be passed through a chain of servers (being repeatedly translated along the way) in order for it to be interpreted by its target. For example, as a mutant query plan makes its way through the P2P network, it will reveal to a given site all of the sub-queries that have yet to be evaluated, as well as the results of all previously-evaluated sub-queries. In optimizing these types of systems for performance alone, this increased amount of information about the query's intension that is released to the query processing network has historically been overlooked. In this paper, we argue that this need not be the case and that the querier privacy should also be considered as a point of optimization in distributed data management systems.

In particular, our goal is to intuitively present both an initial analysis of the flow of information during the evaluation of distributed queries and further a concise metric for comparing the information revealed to participants in different query processing schemes. For the latter point, we show that the ability of remote nodes to reconstruct the intension of a query is one such metric that can be used to concretely assess the privacy loss associated with a given query plan. We will further propose both a privacy aware variant of mutant query processing and an efficient privacy-preserving distributed query processing scheme.

To these ends, the rest of this paper is structured as follows. Section 2 presents an overview of distributed query processing techniques. In Section 3, we describe our metric for assessing the privacy loss of a given query strategy and use it to compare the relative tradeoffs between the query processing techniques discussed in Section 2. Section 4 presents our privacy aware extension to mutant query processing as well as a hybrid query processing scheme. Section 5 discusses a number of interesting open questions in the privacy-aware distributed query processing space. In Section 6, we overview related work, and we present our conclusions and directions for future work in Section 7.

2. DISTRIBUTED QUERY PROCESSING

In this section, we overview the query processing workflow, as well as several methods for distributed query processing. We then informally highlight differences in the amount of information revealed about the query by each method. We begin by detailing an example scenario that will be used throughout this paper.

2.1 Example Scenario

Alice is a low-ranking executive for the manufacturing corporation ManuCo. ManuCo maintains three separate database servers for their Facilities, Inventory, and Human Resources departments, respectively. On their Facilities server, information about all of ManuCo's many manufacturing plants (namely, each plant's internal id, name, and location) is stored in the aptly named `Plants` table. Their Inventory server hosts the `Supplies` table, which stores the name, type, and quantity of all of ManuCo's manufacturing supplies in addition to a plant_id which indicates the plant at which the supplies represented by a given tuples are stored. The Human Resources' department server keeps a list of plants that should undergo an internal company audit in the near future. This list is not maintained as a base table, but instead as a view called `Audit_Watch` of ManuCo's other corporate database tables, defined by the following query:

```
SELECT Plants.id FROM Plants, Supplies
WHERE Supplies.type = "solvent"
      AND Supplies.plant_id = Plants.id
GROUP BY Plants.plant_id ORDER BY COUNT(*) DESC
```

This view presents the plants that are currently storing solvents in order from the plant storing the most variety of solvents to the plant storing the least. Finally, ManuCo also maintains an `Employees` table to keep track of the name and title of each employee, as well as the plant_id of the plant in which each employee works. For efficiency reasons, the `Employees` table is replicated across all three of ManuCo's database servers.

Recently, Alice has become concerned that her company may be illegally dumping industrial solvents into nearby waterways, and wishes to use the information stored in these corporate databases in conjunction with statistics on contaminant levels of public waterways that is maintained by the environmental watchdog group Pollution Watch to justify her concerns. The `Polluted_Waters` table of Pollution Watch's database stores the names of contaminants, their observed concentration levels, the bodies of water they were observed in, and the locations of said bodies of water.

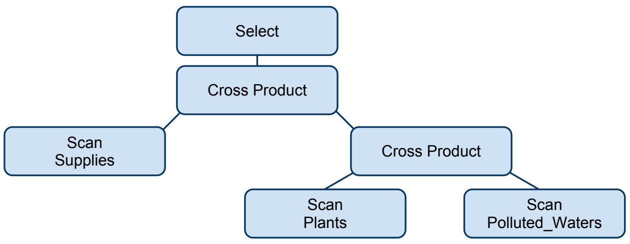
2.2 General Query Processing

A "textbook" architecture for query processing typically consists of five steps: *parsing*, *reorganization*, *optimization*, *code generation*, and *plan execution*. These five steps can be applied to query processing in any incarnation of a database system (centralized or distributed). To illustrate these five steps, we will refer to our example scenario. To see if there is even a correlation between the solvents stored at ManuCo's plants and the pollution present in the surrounding bodies of water, Alice might issue the following SQL query:

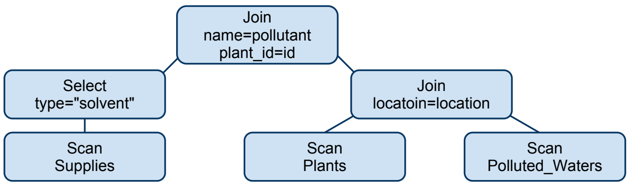
```
SELECT * FROM Plants, Supplies, Polluted_Waters
WHERE Supplies.type = "solvent",
      AND Supplies.name = Polluted_Waters.pollutant,
      AND Polluted_Waters.location = Plants.location,
      AND Plant.id = Supplies.plant_id
```

To *parse* this query, it must be translated from whatever query language it was issued in (in this case, SQL) to an internal representation that the query processor can operate on directly. In existing database systems, the representation is almost always a tree of operators (selects, projects, joins, etc.). The tree that would be generated by parsing our example query can be found in Fig. 1(a). From here, the query is *reorganized* according to prescribed rules. This step will, e.g., un-nest sub-queries, simplify expressions where possible, and remove redundant information from the query. The reorganization of our example query tree is shown in Fig. 1(b). After this first pass of general streamlining is complete, *optimization* for the current state of the system can take place. Optimization will result in a number of different plans for the execution of the query. The optimizer will select which indexes should be used to access data from the base tables, what algorithms should be used to execute certain operations (i.e., joins), and where each of the operators in the plan should be executed. Note that in a distributed setting, optimization will require knowledge of the metadata catalogs from remote databases that will be needed to evaluate the query. All of the plans generated in the first stage of optimization will need to be checked against cost estimations (bandwidths and latencies between given nodes in the system are of particular importance in a distributed environment) and compared against one another. The final, optimized tree for our example query is shown in Fig. 1(c). This tree would then be passed off to be converted into a plan that can actually be executed by the underlying database engine (*code generation*), which would in turn released to the system to be *executed.*

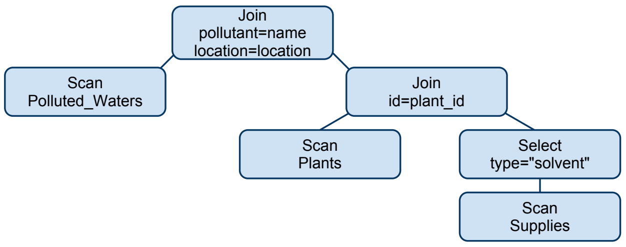
The above provides only a brief overview of query processing. For a more in-depth review of query processing in general, as well as a survey of work in distributed query processing, we refer the reader to [12].



(a) Tree generated by the *parsing* phase.



(b) Tree generated by the *reorganization* phase.



(c) Tree generated by the *optimization* phase.

Figure 1: Query trees generated over the course of processing an example query. Assume the condition on the Select node in (a) to be the full condition of the WHERE clause in the example query

2.3 Traditional Execution Methodologies

In order to actually execute a query in a distributed environment, a querier must pass a request for information to a data source and receive back an answer. One method to accomplish this is to have the querier request from every data source all data that it could possibly need to execute the overall query (i.e., request the full contents of all tables listed in all scan operators in the query tree that are annotated to be executed at a given site) and then perform all of the processing on this data that is required. This technique is known as *data shipping*, as the query never moves from the initiator; instead, all of the data comes to it. If the example query from Section 2.2 was executed via data shipping, Alice's computer would request the full contents of the `Plants` table from ManuCo's Facilities server, the `Supplies` table from the Inventory server, and the `Polluted_Waters` table from Pollution Watch's server. Note that if one node requests a scan of a view defined by another node, the node defining the view would materialize the view and send the results of this evaluation back to the requesting node.

For a great many queries, however, data shipping will result in the network transmission of far more data than is actually necessary to evaluate the query. A complementary technique is that of *query shipping*. In query shipping, the sub-trees of a query plan that have all of their operators annotated for execution at the same remote site are sent to the site as a whole. Remote sites can then transfer back only the results of the execution of that sub-tree, instead of the full contents of the tables or views requested. This technique is handy as a great many relational operators (e.g., selects, projects, and in certain cases joins) will have results of *lower* cardinality than their inputs. Hence, in many cases the use of query shipping will result in a reduced amount of information having to be transferred over the network. In the case of our example query, the Inventory server would be asked not only to scan the `Supplies` table, but also to select from that table only the tuples that have "solvent" as their type to be sent back to Alice. In the event that one node requests the scan of a view defined by another node, the node defining the view would send the query that defines the view back to the requesting node for further processing.

Neither data shipping nor query shipping is strictly preferable in all cases. For example, evaluating sub-trees consisting of only cross products using query shipping will result in vastly more data than is provided as input, whereas using data shipping would result in much more efficient evaluation. In *hybrid shipping*, a querier can choose to use either data shipping or query shipping in its communication with any given remote site depending on which technique could more efficiently execute a sub-tree that requires data from that site. Though quite a bit more information about the query is revealed in the process of query shipping (as compared with data shipping), it should be noted that hybrid shipping, as it has been proposed in the literature, is used only to optimize query execution for efficiency [10].

2.4 Mutant Query Plans

Mutant query plans (MQPs) have been proposed as a form of "combined shipping," as they are data structures that contain both a representation of the query to be evaluated, as well as intermediate results from partial evaluation of the query [16]. A user wishing to evaluate some query first initializes a new MQP with the full query tree and then sends it to one of the servers that will evaluate part of that query tree. That server then evaluates what it can of the query and attaches the results to the MQP in place of the nodes in the tree that it evaluated. In the case of a remote view expansion, these "results" will be a sub-query tree generated from the query that defines the view being expanded. In all other cases, the server will evaluate a sub-tree of the query and replace it with the data tuples that were the actual results of its evaluation. As such, the querier is able to maintain only negligible local state during the evaluation of a query (e.g., a query ID), and can offload the computational burden of the entire query to remote database servers.

For example, if Alice were to use a MQP to evaluate the query from Section 2.2, she would first send the complete query tree to ManuCo's Facilities server. The server would scan all of the tuples from its `Plants` table and remove the corresponding scan from the MQP, replacing it with the resulting data tuples. The MQP would then be sent to the Inventory server, which would scan its `Supplies` table, selecting from it the tuples with "solvent" as their type. The server would then join these tuples with the results that the Facilities server had previously added to the MQP, and replace the sub-tree rooted by the corresponding join node with these results. At this point, the MQP would consist of a node that represents a scan of the `Polluted_Waters` table, a node representing the final join in the query, and the intermediate result of the rest of the query tree. From here, the Inventory server would send the MQP to Pollution Watch's server, which would perform the final scan and join and send the final result of the query back to Alice.

Over the course of its execution, it may be necessary for servers that evaluate some portion of an MQP to reorganize and re-optimize its query tree. For example, if a server finds itself evaluating multiple separate sub-trees of a query, it will first attempt to combine them into one sub-tree if that will minimize the size of the intermediate results in the MQP. Conversely, if a server needs to evaluate one sub-tree, but the results of that sub-tree could be more compactly stored as two sets of results under a binary operator (a cross product, for example), the server can defer the evaluation of that operator in order to minimize the size of the MQP.

2.5 Wigan

Wigan [7] is a variant of query shipping that uses BitTorrent (`http://www.bittorrent.com/`) as an overlay upon which to build a peer-to-peer database system. In Wigan, queries are sent to a BitTorrent tracker that indexes database tables. Servers hosting these tables acts as seeders, to use BitTorrent terminology. The tracker responds to a query by directing the querier to a list of servers that can resolve their query. The querier will then send their entire query to each of these servers, along with a request for some chunk of the data the querier requires. By requesting different chunks from each server in parallel, the querier is able to take advantage the speed in data transmission that BitTorrent can provide. The tracker will remember what subset of the requested table the querier downloaded and list the querier as a data provider for future requests for the same data. This is a variant of query shipping, as the combination of data from different sources must be performed by the querier, but the querier receives only the necessary subset of the tables he is interested in from each data source.

3. INFORMATION FLOW ANALYSIS

As should be readily apparent from the previous section, the choice of execution methodology for a query in a distributed system has a noticeable effect on the information that is revealed about that query to the servers hosting the data that the query is operating on. In particular, there exist two main classes of information flow that must be accounted for: the *explicit* flow of result tuples from data sources to queriers, and the *implicit* flow of the query intension (i.e., *Why is this query being issued?*) from the querier to the data sources. In this paper, we assume that all result tuples explicitly flowing from a data source are permitted by the source's authorization policy. We further assume that these policies are specified on a per-table basis, and place no restrictions on the form that these policies take (e.g., role-based access controls, access control lists, or logical policies). In this section, we seek to quantify the implicit flow of query intension and compare the relative privacy loss incurred by existing distributed query processing techniques.

3.1 Measuring Implicit Information Leakage

In order to analyze this implicit flow of information, the intension of a query must be defined:

DEFINITION 1 (QUERY INTENSION). *The* intension *of a query is the set of all operations and data sources that are used to generate the result of that query.*

Note that the intension of a query can be concisely represented by a query tree. Hence, by limiting the ability of participants in a distributed query evaluation to reconstruct the initial query tree, the amount of implicit information that is leaked can be limited and queriers' privacy can be guarded. There exist both lower and upper bounds to the query intension that must be revealed to a given database server while still allowing the query to be resolved. It is bounded below by revealing to each server only the leaves of the query tree that are to be executed by that server (i.e., the scans associated with tables stored by that server). This can be accomplished (at the cost of potentially massive amounts of network traffic) through the exclusive use of data shipping.

Trivially, the upper bound on query intension leakage is to reveal to a database server the full intension of the query. Materializing the entire contents of this query tree, however, is not entirely trivial. It is possible that no node in the network (including the initial querier) will actually know the true full query tree. In the case that some distributed view needs to be expanded in order to evaluate the query, the initial querier could be left unaware of the definition of that view (if, for example, either data shipping or a mutant query plan were used to evaluate the query). This view, however, is still a part of the intension of the overall query as the operations and data sources that define it are used to generate the results of the query. As such, the server that defines a view that is used in the evaluation of a query is a supplemental querier whose privacy (namely the intension of their view) must also be protected. In between these two bounds, two truths about the amount of information revealed about the intension of a query are readily apparent.

AXIOM 1. *If two servers, A and B, are used to evaluate a given query are shown exactly the same sub-tree S of the overall query tree, they learn the same amount of information about the intension of that query.*

AXIOM 2. *Assume that some server A is used to evaluate a sub-tree S of some query Q (and hence has S revealed to it). Further, let another server, B, be shown a sub-tree S' of Q that contains S in its entirety in addition to other operational nodes of the query. In this case, B learns more information about Q than A.*

In all other cases, while we can trivially compare the total number of nodes revealed to data servers as a measure of leaked intension, the actual impact of such revelations is far more dependent on the semantic value of the data that is revealed, than on the simple count of tables accessed. Looking back to our example query from Section 2.2, Alice's request for a table on polluted waters is something she would much rather keep private than her request for all of the solvents currently being kept by ManuCo. As an executive for ManuCo, her interest in waterway pollution is much more semantically interesting than her desire to check up on company supplies, even though the sub-tree that is revealed to the Inventory department would contain more operational nodes than the query revealed to Pollution Watch. From this we can infer that the privacy impact of revealing an individual node in a query tree can vary widely. The above

1	`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`

2	`SELECT * FROM Employees, Plant`
	`WHERE Employees.title = "manager",`
	`AND Employees.plant_id = Plant.id,`
	`AND Plant.location IN <list of polluted locations>`

3	`SELECT * FROM Employees, Audit_Watch`
	`WHERE Employees.name = "Charlie"`
	`AND Employees.plant_id = Audit_Watch.plant_id`

Table 1: The example queries Alice issues to investigate a connection between ManuCo and waterway pollution.

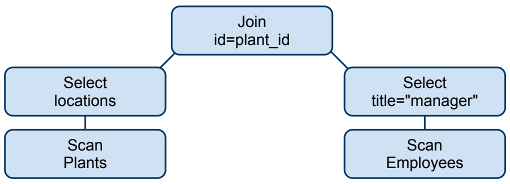


Figure 2: Optimized query tree for Alice's second query.

axioms hold despite this observation, as they operate on shared sub-structures.

3.2 Examples

To demonstrate the utility of this seemingly-simple measure of privacy loss, we will analyze three queries that Alice might make to further her investigation of waste disposal at ManuCo (see Table 1). The first of these queries is the query from Section 2.1. The second query (see Figure 2) identifies the managers of plants near polluted waterways, while the third query (see Figure 3) checks to see whether the manager of several polluting plans, Charlie, happens to be listed on an internal company watchlist.

Query 1. The use of data shipping to evaluate query 1 would be reveal to each of the three sites only a single scan node. The downside to this privacy is that all three tables will need to be transferred over the network in their entirety. The time needed to process the query will be determined by the longest of the three transfers, as they can be parallelized.

Query shipping would lessen the amount of network traffic generated as the Inventory department will only have to transmit the tuples from its `Supplies` table that have "solvent" listed as their type, easing the total amount of network traffic generated. Though, by Axiom 2, this implies that the Inventory department will learn more about the intension of her query through the use of query shipping than it would through data shipping. The Facilities department and Pollution Watch, however, will learn no more than they did through data shipping (by Axiom 1), as they are still asked to deliver the full contents of their respective tables to Alice.

A mutant query plan would, for all sites, reveal more of

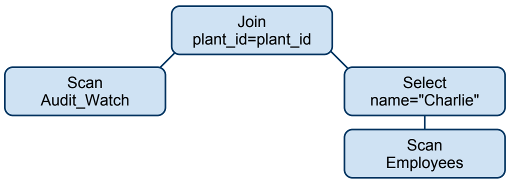


Figure 3: Optimized query tree for Alice's final query.

the intension of the query than either data or query shipping. The Facilities department would be shown the full query, and hence would realize Alice's intentions to show a link between waterway pollution and ManuCo's plants. The Inventory department should also be able to come to this conclusion as even though by the time that the MQP reaches their server the sub-tree that shows Alice's interest in ManuCo's plants will have been replaced with only data tuples, the fact that Pollution Watch is included in the MQP as a data source in addition to the fact that only tuples representing potential pollutants are being selected from the `Supplies` table should provide ample clues as to Alice's reasons for issuing the query. In this paper, we will not be concerned with the fact that all intermediate results of the query that are attached to the MQP will be revealed to all future servers that evaluate that MQP as this is explicit information, and we assume that when the server that produced those results attached them to the MQP, the database access control policy allowed for such a release.

Though the network transmissions made through the use of an MQP must be done in serial, it provides many opportunities to cut down the total amount of data transmitted that neither data shipping nor query shipping can offer. The first transmission of explicit data (from the Facilities server to the Inventory server) would contain just as many data tuples as were transmitted from the Facilities server to Alice using query shipping. The next hop, however, would contain only the results of joining those data tuples with selections from the the `Supplies` table. Depending on the selectivity of this join, this could result in far less data being transferred from the Inventory server to Pollution Watch than was transmitted from the Inventory server to Alice in either of the previous two schemes. The final transmission of data would then be the transfer of the result of the overall query from Pollution Watch to Alice. Assuming that the results of the query are smaller in size than Pollution Watch's `Polluted_Waters` table, this will also result in less network traffic over either data or query shipping.

If evaluated using Wigan, the full intension of the query will be revealed to all parties involved. As compared with MQPs, this reveals the same information about the intension of the query to the Inventory department, and more information to both the Facilities department and Pollution Watch. Further, it should be noted that the set of parties involved in the evaluation of this query using Wigan is not at all constrained to just Alice and the servers providing data to answer her query. Any other server or user that is known to the Wigan tracker to have at least the data that Alice needs to answer her query will be brought in to assist in providing Alice with that data in order to maximize the parallelism of data transfer. This could potentially mean that Alice would be revealing her entire query to one of her superiors at ManuCo, if they had previously requested either the `Plants` table, Pollution Watch's `Polluted_Waters` table, or all of the solvents listed in the `Supplies` table. The amount of network traffic generated by the use of Wigan is the same as that generated through the use of query shipping.

In summary, while this query could be evaluated most efficiently using an MQP, this execution methodology comes at a cost to privacy surpassed only by Wigan. As was noted, these violations to privacy are, in this case, due in large part to the fact that both of ManuCo's database servers that were used to evaluate the MQP were requested to do their part of its evaluation before Pollution Watch, and as a result were able to see that Pollution Watch was included as data source for the query. If Pollution Watch were asked to evaluate is part of the MQP first, this violation could be assuaged. However, the order in which an MQP is sent to servers has an effect on not only the privacy of that query, but also on the amount of network traffic generated over the course of evaluating a query, as all intermediate results are passed along with the MQP on the rest of its journey. To minimize the amount of network traffic generated, the MQP should first be sent to the server that is estimated to generate the intermediate results with the lowest cardinality (which we assume in this case to be the scan of the `Plants` table), and from there to the other servers in the order that is estimated to minimize the cardinality of all intermediate results stored in the MQP at any given time.

Query 2. In the case of data shipping, Alice will request data from the `Employees` table from either ManuCo's Inventory or Human Resources department. Though the Facilities department also maintains a copy of the `Employees` table as well, since Alice will need this server to provide her with its `Plants` table, grabbing `Employees` from one of ManuCo's other database servers will decrease the total time needed to transfer both tables by transferring them in parallel.

In contrast, if query shipping or an MQP were used, the replication of the `Employees` table would cause the Facilities server to be the only one to receive this query, as it could evaluate the *entire* query. In these cases, the Facilities server learns (by Axiom 2) more than either of the two servers used in the case of data shipping.

Wigan would pull the `Employees` table from all available data sources (including the Facilities server), maximizing the parallelism of the transfer while revealing the intension of the query to more parties than any of the other methodologies.

While data shipping, query shipping, and MQPs all make use of replication to improve the performance of query evaluation (in fact, query shipping and mutant query processing generate the minimum amount of network traffic required to resolve the query in that the exact result of the query is all that is sent over the network). The approach taken by data shipping, though, has the added benefit of better protecting the intension of the query by minimizing how much is revealed to any given database server. Though either the Inventory or Human Resources learns more by being included in the evaluation of query 2, no server learns as much about the intension of query 2 as the Facilities server would in the case that it was issued both scans (by Axiom 2).

Query 3. The evaluation of query 3 requires the Human Resources department server to be a supplemental querier through the inclusion of its `Audit_Watch` view. Using data shipping, the intension of the `Audit_Watch` view is partially revealed to the Facilities and Inventory department servers (in that the Human Resources server requests their `Plants` and `Supplies` tables). By the fact that the Human Resources server maintains the `Audit_Watch` view, however, it can be reasonably expected that such revelations are a regular occurrence that have a minimal impact on its privacy.

Query shipping, on the other hand, would call for the Human Resources server to reveal to Alice the intension of its `Audit_Watch` table, so that she could evaluate the query that defines it. To allow for this, the Human Resources server would have to place access controls on not only the contents of its database tables and views, but also the definitions of its views. As the Human Resources department server must allow for the definition of `Audit_Watch` to be attached to an MQP to be used to evaluate this query, similar concerns would be raised through the use of a mutant query plan.

Wigan, however, would be unable to even evaluate this query, as the expansion of a distributed view would require a more sophisticated tracker than the one proposed in [7].

3.3 Comparison of Evaluation Methodologies

From these examples, it is quite clear that, of the methodologies involved, data shipping always reveals the least about the intension of the query, while Wigan reveals the most. Further, query shipping will always reveal at least as much intension as data shipping. In all cases, query shipping will request of each data source a scan of that source's tables that are needed to evaluate the query. In many cases, though, query shipping will also request that the server perform some other operation on the results of those scans. Query shipping also requires the release of view definitions to queriers, something that data shipping is effectively able to hide.

In the case of mutant query plans, however, these examples may be misleading. In each of the cases presented, MQPs reveal more of the intension of the initial query than query shipping would. Intuitively, this makes sense, as a server should be annotated in the MQP to evaluate the same sub-tree presented to it in query shipping, but that server may also see other nodes of the query tree that are present in the MQP. This would indicate that the intension revealed by MQPs is bounded below by query shipping and above by Wigan. However, none of these rather simple queries required any of the database servers to perform any reorganization of the query tree to streamline the query's processing. In the face of such reorganization, it is quite difficult to make strong comparisons about the intension revealed, as the reorganization would reform sub-trees that would have been presented to servers in any of the other execution methodologies, invalidating the basis for both of our axioms for comparing intension leakage. Note that deferral does not detract from our ability to compare intension leakage with other methodologies. In order to defer the execution of some sub-tree to another server, the server initially annotated to evaluate it must first be shown that sub-tree.

4. PRIVACY-AWARE QUERY EXECUTION

As the examples presented in the previous section have shown, there is a wide range in both the privacy and efficiency that is provided by available distributed query execu-

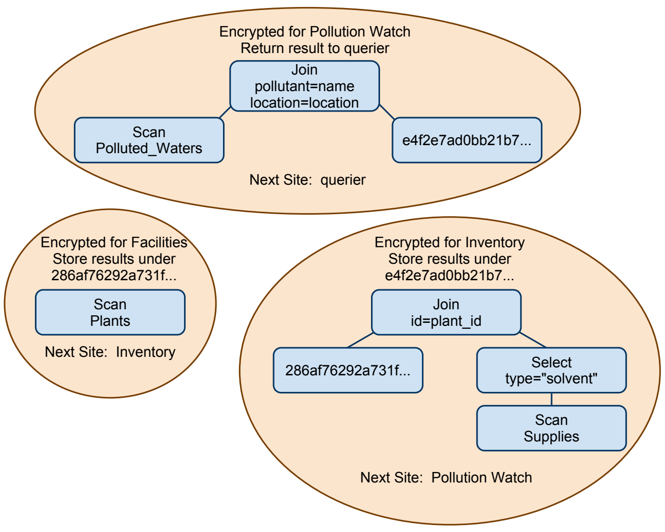


Figure 4: An illustration of a hardened mutant query plan.

tion methodologies. In this section, we will propose two different, independent, extensions to existing query execution methodologies with the aim of striking a balance between privacy and performance. Hardened mutant query plans offer the performance provided by vanilla MQPs sources only the nodes of the query tree that they will actually evaluate. Our proposed hybrid query evaluation scheme, on the other hand provides much more powerful query privacy controls than HMQPs while offering better performance than either query shipping or data shipping alone.

4.1 Hardened Mutant Query Plans

One way of attaining this goal involves using encryption to harden mutant query plans to better protect the intension of the query they contain. Instead of passing a whole query tree from server to server, such a hardened mutant query plan (HMQP) could be sent as a stack of encrypted portions of the query tree and a hash table mapping these encrypted subtrees to their intermediate results.

Description. To create an HMQP, the querier must break up the query tree into sections that will be executed by a given site, replacing sub-trees to be evaluated by another site with the hash of that sub-tree. In the case of our first example query, this would mean that Alice would encrypt the scan of the `Plants` table, as this is the only section of the query to be evaluated by this server, and directions to send the HMQP to the Inventory server next with the session key established with the Facilities server. She would then replace this node in the query tree with its hash. Next, she would take the lower join in the query tree and its children, replace them in the query tree with their hash, and encrypt them for the Inventory server along with instructions to subsequently send the HMQP to Pollution Watch. The remainder of the query tree would then be encrypted for Pollution Watch. An illustration of the results of this process is presented in Figure 4. Alice would then push each of these encrypted blobs onto the stack in the reverse of their intended order of execution (Pollution Watch's first, then the Inventory server's, and finally the Facilities server's).

To evaluate this hardened MQP, Alice would first send both the stack and empty hash table to the Facilities server. The Facilities server would then pop its blob from the stack, decrypt it, evaluate it, store the results in the hash table, and send the HMQP to the Inventory server. The Inventory server would similarly pop its blob and decrypt it, though it would first have to look up the results of the Facilities server's section of the sub-tree in the hash table before evaluating its own sub-tree, adding those results to the hash table, and sending the HMQP to the Pollution Watch to complete the HMQP's execution.

Costs and benefits. Given that the portions of the query tree to be encrypted are quite small—consisting of only operation names and arguments—and symmetric encryption keys can be established between nodes, the cryptographic overhead imposed by HMQPs is minimal. Further, this minimal overhead allows the query to be processed with the efficiency of an MQP while revealing to any evaluating server only the portion of the query that it needs to evaluate. The use of an HMQP to evaluate query 1, for example, would result in the generation of the same amount of network traffic as the use of a standard MQP (which Section 3.2 showed to be less than that generated using other techniques). Further, the first two servers to evaluate the HMQP would learn much less of the query intension; e.g., the Facilities server would see only the scan of its `Plants` table, where as a standard MQP would have revealed the entire query.

The only significant performance hit from the use of HMQPs (as opposed to standard MQPs) would be incurred through the expansion of remote views. Should the addition of such a view definition to the HMQP create a query tree that is sub-optimal, the server defining the view would not be able to reorganize and re-optimize the whole of the HMQP to account for this. This shortcoming is only a *potential* for suboptimal evaluation in the face of remote view expansion, though, not a *guarantee* of it. In Section 5, we discuss one potential method to allow for partial reorganization and re-optimization of HMQPs.

Though deferral must be abandoned in the use of HMQPs as it reveals query intension encrypted for one server to a different server, this should not be a problem for an intelligent query plan (where, for example, cross products are annotated for evaluated by the querier as the last step).

4.2 A Hybrid Scheme

Though hardened mutant query plans provide better privacy than standard MQPs while maintaining efficient query evaluation, query shipping and data shipping can provide better privacy than even HMQPs at the cost of maintaining more local state. If a query optimizer could know which portions of a query that the user considers private, privacy and performance can be maintained by executing a query using a combination of data shipping, query shipping, and MQPs. An obvious challenge, though, lies in capturing this notion of *private*. One way to accomplish this is to modify the query language being used to support privacy annotations when a query is being issued. In SQL, this could be accomplished, e.g., by enclosing private attributes in square brackets. Figures 5 and 6 present pseudo-code for a hybrid query processor that takes a query tree resulting from a

```
define mark_tree(to_check)
  while len(to_check) > 0:
    cur = to_check.pop()
    if cur.op == "SCAN":
      if cur.is_exclusive:
        rv = climb(cur, QUERY_SHIPPING)
        if rv != null:
          to_check.append(rv)
      else:
        rv = climb(cur, MOST_EFFICIENT)
        if rv != null:
          to_check.append(rv)
    else:
      // Binary op on two non-private sub-trees,
      // mark ancestors for most efficient evaluation
      if (!cur.is_private &&
          cur.child[0].method == MOST_EFFICIENT &&
          cur.child[1].method == MOST_EFFICIENT):
        rv = climb(cur, MOST_EFFICIENT)
        if rv != null:
          to_check.append(rv)
      // Binary op on two sub-trees exclusive to the same site
      else if (!cur.is_private &&
               cur.child[0].method == QUERY_SHIPPING &&
               cur.child[1].method == QUERY_SHIPPING &&
               cur.child[0].site == cur.child[1].site):
        rv = climb(cur, QUERY_SHIPPING)
        if rv != null:
          to_check.append(rv)
      // Either the binary op is private, or sub-trees
      // have incompatible privacy requirements
      else:
        while cur != null:
          cur.method = LOCAL
          cur = cur.parent
```

Figure 5: Pseudo-code for a hybrid query processor. Note that this code does not return a new data structure, but instead further annotates an already-existing query tree.

```
define climb(node, exec_meth):
  node.method = exec_meth
  node = node.parent
  while node != null:
   // ancestors require more privacy than nodes
   // are currently being marked for, so stop
    if node.method < exec_meth:
      return null
    else if node.is_private:
      while node != null:
        node.method = LOCAL
        node = node.parent
      return null
    // Binary op will need to be checked again once
    // both subtrees have been marked
    else if len(node.children) > 1:
      return node
    else
      node.method = exec_meth
      node = node.parent
  return null
```

Figure 6: Pseudo-code for a function needed by the hybrid query processor.

1	`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`

2	`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`

Table 2: Two examples privacy annotations for example query 1.

privacy-annotated SQL query, and determines which nodes of the tree should be evaluated locally by the querier, which should be evaluated using query shipping, and which nodes can be safely be evaluated by the most efficient method available. This code requires a queue, `to_check`, that is initialized to contain all of the leaves of the query tree in question, and also the three global constants: `MOST_EFFICIENT`= 2, `QUERY_SHIPPING`= 1, and `LOCAL`= 0.

If an attribute in the selection list of a query is enclosed in square brackets, any projects that contain that attribute would be annotated as private in the query tree. Such nodes and their ancestors in the query tree would then be kept at the querier for local evaluation once their descendants have been evaluated. Private annotations on conditions in the `WHERE` clause would similarly cause any node in the query tree (join or select) that contains that condition to be marked for privacy-preserving local evaluation. On the other hand, an annotation on any of the table names in the `FROM` clause would cause the associated scans to be marked as *exclusive*. As scans must be revealed to the server hosting the table to be scanned, an annotation on a scan will keep that scan and any node of the query tree that operates on data from that scan from being revealed to any server aside from the one who will execute it through the use of query shipping. The portions of the query tree that are unaffected by any privacy annotation could then be executed using the most efficient methodology.

Two potential privacy annotations for our first example query are presented in Table 2. The first would cause the `Polluted_Waters` table to be read via data shipping and the root join of the tree would be evaluated by the querier once the requisite data had been received. The rest of the tree could then be evaluated as an MQP. This would generate slightly more network traffic than evaluating the entire query using an MQP and less network traffic than using either query shipping or data shipping to evaluate the entire query. The second example would cause the entire query to be evaluated using data shipping. By marking the condition on the data from the `Supplies` table as private, the select node of the query tree and both of its join nodes would be executed locally (the private condition is used in the select node, and both of the joins are ancestors of the select), leaving only the scan nodes to be evaluated by remote servers.

So far, this scheme protects only the privacy of the initial querier. Protecting the privacy of supplemental queriers could be similarly accomplished given that they make note of which of their view definitions should be kept private. When a server receives a request for a view with a private definition, it would execute the query defining the view and return to the querier the results (or attach them to an MQP, if the query was received in that fashion) as if the view were actually a locally defined table. The server could further respond with requests for other views by returning (or attaching) the definition of the query, and offloading the burden of re-evaluating the view.

5. OPEN PROBLEMS

We now discuss several interesting directions for future research on the privacy-aware evaluation of distributed queries.

Syntax Versus Semantics. One of the primary strengths of standard query optimization techniques is that they operate largely without user involvement. In particular, most of these techniques involve the application of structural transformations to the query being optimized that are influenced by statistics (e.g., predicate selectivity) that are automatically collected by the DBMS. Unfortunately, the privacy protection that should be afforded to a given query is largely a semantic issue that will vary from user to user. In the examples discussed in this paper, e.g., Alice felt that her queries to ManuCo's databases were sensitive and could cost her a job if they were leaked. However, if she were an auditor working for ManuCo's regulatory oversight committee, these types of queries would not seem out of place.

In this paper, we proposed an initial—largely syntactic—measure of the privacy loss associated with a given query plan: the degree to which nodes within the system can reconstruct the intension of the querier. Although it is very easy to show that limiting this ability restricts the implicit flow of information within the system, it is not always the case that such privacy optimizations are sensible. For instance, splitting a join of two sensitive tables across two servers to limit the amount that each server learns makes sense; doing the same for two non-sensitive tables does not. Capturing the definition of *sensitive* on a per-user or system-wide basis is likely to be a significant challenge.

Query Annotation Techniques. In Section 4.2, we proposed one method to gather such notions of sensitivity via a modification to SQL that allows the user to mark certain sections of his query as "private". These very coarse-grained annotations were then used to guide the query optimization process. At one extreme, one interpretation of a *private* join condition or selection predicate could be that it should not be revealed to external servers at *any* cost. On the other hand, such a condition could also be viewed as a *preference*: e.g., only reveal this condition if doing so would result in at least a 200% faster query. Ideally, it should be possible to allow users to specify hard privacy constraints that should not be violated, as well as a range of suggestions that enable an explicit balance efficiency and privacy loss.

Another interesting concept would be to include the concepts of collusion and separation of duty. For instance, it could be beneficial to mark portions of a query as private with respect to some group of nodes who are thought to collude (e.g., closely-related departments in a company) or to indicate that a single node should not learn about two or more different portions of a query. This could benefit both HMQPs and the hybrid scheme in Section 4.2. By encrypting sections of a query tree to multiple servers, these servers would be able to reorganize and re-optimize amongst themselves. Developing query annotation languages that can capture these and other important privacy constraints without overwhelming users is an interesting challenge.

A complementary area potentially worth exploring is that of interactive query plan generation. Rather than asking users to annotate queries with privacy preferences, these preferences could be learned over time by summarizing the information flow associated with several query plans and asking them to rank the options presented.

Incorporating History. While we have up until this point primarily examined only the privacy of individual queries, users may also have a keen interest in protecting the privacy of their data access patterns at large. In the case that a user were to issue multiple queries at once, ideally, all of those queries would be optimized for privacy together, as opposed to piecemeal. Even given a delay between queries, however, it would be ideal to make use of privacy decisions made on past query evaluations. If a given query had already been issued by the same user, the system should evaluate it in the same manner to avoid leaking more of the query's intension via the union of the two executed plans. Further, if a given query tree contains some sub-tree that had been previously evaluated on behalf of the issuing user, the system should similarly evaluate this sub-tree in the same manner as had been previously used.

Correctness and Completeness Concerns. Any execution methodology that offloads the evaluation of a remote view raises concerns about the correctness and completeness of the information returned. When the server that defines a view evaluates it, it passes its own credentials to the data sources needed by that view for checks against the data sources' access controls. Whenever the evaluation of a remote view is passed off to another party, that other party will only be able to use their own credentials. In the case that the data sources involved use table-level controls on their database (parties are either allowed or denied access to whole tables), other parties could be unable to evaluate the view if they lack the credentials needed to access any of the tables involved in the query. Even more troublesome, though, is the case where data sources make use of row- or column-level access controls. In such a situation, the evaluating party may still get a result for the view, though it may not be the same result the the server that defines the view would have received. Such a discrepancy would clearly have an impact on the results of the overall query, but the querier would be unaware of this inconsistency.

6. RELATED WORK

The privacy of query intension has been previously studied in the form of private information retrieval (PIR). Information-theoretic PIR can be achieved through the replication of database tables [1, 5] or the use of auxiliary computational servers at query time [11]. In contrast, we focus on the privacy-aware use of existing query execution methodologies to benefit both performance and privacy. While computational PIR [4,13] can be used in a single-server environment, such schemes have been shown to be orders of magnitude slower than the trivial transfer of an entire database table [19]. The approaches we investigate strike a balance between performance and privacy which in the worst case will use such a database transfer through data shipping.

Historically, performance alone had been the goal of distributed query optimization [10, 12]. On the other hand, we propose the addition of privacy as a metric for distributed query optimization to balance privacy with efficiency.

k-anonymity and l-diversity [15, 20] seek to protect the privacy of individuals whose personal data is included in the release of tuples from some data store. By releasing only sufficiently large and similar groups of tuples from the original database, the privacy of any individual who is represented in that database is hoped to be protected from de-anonymization attacks. Differential privacy [8, 9] aims towards similar goals by perturbing output from statistical databases according to a carefully chosen distribution. By contrast, this paper deals with the privacy of individuals who are issuing queries to database systems. This line of reasoning also differentiates our proposed methodology from recent work that maintains the privacy of data providers by outsourcing their data [6, 18].

7. CONCLUSION

We present a preliminary investigation into the implicit leakage of potentially private information during the evaluation of queries to multiple, heterogeneous, distributed database systems. Recent proposals for accomplishing this task have trended towards increased performance at a high cost to querier privacy. In response, we propose several variants on existing distributed query execution methodologies that aim to maximize performance while maintaining querier privacy. As this is only an initial investigation in the querier privacy in distributed query processing, we have forgone strictly formal definitions and discussions in favor of an intuitive presentation. Our future work in this area will present a more formal treatment of the concepts presented here.

In addition to the problems presented in Section 5, future work would strive to integrate privacy concerns into the query optimization process itself. Our proposed hybrid evaluation scheme operates only on trees that are the result of query optimization. Taking privacy into account during optimization could allow our hybrid scheme to take advantage of replication of database tables to improve privacy.

This work has implications outside of just distributed databases. Declarative networking [14], and trust management [3] both function as distributed sets of rules and facts. In order to route traffic or make trust decisions, queries must be made to distributed data stores in the system. In these cases both privacy and performance are especially desired properties.

Acknowledgements. This research was supported in part by the National Science Foundation under awards CCF-0916015, CNS-0964295, CCF-0914946 and CNS-0747247.

8. REFERENCES

[1] A. Beimel, Y. Ishai, and T. Malkin. Reducing the servers' computation in private information retrieval: Pir with preprocessing. *J. Cryptol.*, 17(2):125–151, 2004.

[2] P. A. Bernstein, F. Giunchiglia, A. Kementsietsidis, J. Mylopoulos, L. Serafini, and llya Zaihrayeu. Data management for peer-to-peer computing: A vision. pages 89–94, 2002.

[3] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *IEEE Symposium on Security and Privacy*, pages 164–173, 1996.

[4] Y.-C. Chang. Single database private information retrieval with logarithmic communication. In *ACISP*, pages 50–61, 2004.

[5] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, 1998.

[6] V. Ciriani, S. D. C. di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Keep a few: Outsourcing data while maintaining confidentiality. In *ESORICS*, pages 440–455, 2009.

[7] J. Colquhoun and P. Watson. Evaluating a peer-to-peer database server based on bittorrent. In *Proceedings of the 26th British National Conference on Databases: Dataspace: The Final Frontier*, pages 171–179, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*, pages 265–284, 2006.

[9] C. Dwork, M. Naor, T. Pitassi, and G. N. Rothblum. Differential privacy under continual observation. In *STOC*, pages 715–724, 2010.

[10] M. J. Franklin, B. T. Jónsson, and D. Kossmann. Performance tradeoffs for client-server query processing. *SIGMOD Rec.*, 25:149–160, June 1996.

[11] Y. Gertner, S. Goldwasser, and T. Malkin. A random server model for private information retrieval or how to achieve information theoretic pir avoiding database replication. In *RANDOM*, pages 200–217, 1998.

[12] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[13] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *FOCS*, pages 364–373, 1997.

[14] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: language, execution and optimization. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 97–108, New York, NY, USA, 2006. ACM.

[15] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam. L-diversity: Privacy beyond k-anonymity. *ACM Trans. Knowl. Discov. Data*, 1(1):3, 2007.

[16] V. Papadimos and D. Maier. Distributed queries without distributed state. In *In WebDB*, pages 95–100, 2002.

[17] V. Papadimos, D. Maier, and K. Tufte. Distributed query processing and catalogs for peer-to-peer systems. In *In CIDR*, pages 5–8, 2003.

[18] P. Samarati and S. D. C. di Vimercati. Data protection in outsourcing scenarios: issues and directions. In *ASIACCS*, pages 1–14, 2010.

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

[20] L. Sweeney. k-anonymity: a model for protecting privacy. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 10(5):557–570, 2002.