Privacy-preserving Queries over Relational Databases*

Femi Olumofin and Ian Goldberg

Cheriton School of Computer Science
University of Waterloo
Waterloo, ON, Canada N2L 3G1
{fgolumof,iang}@cs.uwaterloo.ca

Abstract. We explore how Private Information Retrieval (PIR) can help users keep their sensitive information from being leaked in an SQL query. We show how to retrieve data from a relational database with PIR by hiding sensitive constants contained in the predicates of a query. Experimental results and microbenchmarking tests show our approach incurs reasonable storage overhead for the added privacy benefit and performs between 7 and 480 times faster than previous work.

Keywords: Private information retrieval, relational databases, SQL

1 Introduction

Most software systems request sensitive information from users to construct a query, but privacy concerns can make a user unwilling to provide such information. The problem addressed by private information retrieval (PIR) [3, 9] is to provide such a user with the means to retrieve data from a database without the database (or the database administrator) learning any information about the particular item that was retrieved. Development of practical PIR schemes is crucial to maintaining user privacy in important application domains like patent databases, pharmaceutical databases, online censuses, real-time stock quotes, location-based services, and Internet domain registration. For instance, the current process for Internet domain name registration requires a user to first disclose the name for the new domain to an Internet domain registrar. Subsequently, the registrar could then use this inside information to preemptively register the new domain and thereby deprive the user of the registration privilege for that domain. This practice is known as *front running* [17]. Many users, therefore, find it unacceptable to disclose the sensitive information contained in their queries by the simple act of querying a server.

Users' concern for query privacy and our proposed approach to address it are by no means limited to domain names; they apply to publicly accessible databases in several application domains, as suggested by the examples above.

* An extended version of this paper is available [22].

Although ICANN claims the practice of domain front running has subsided [17], we will, however, use the domain name example in this paper to enable head-to-head performance comparisons with a similar approach by Reardon et al. [23], which is based on this same example.

While today's most developed and deployed privacy techniques, such as onion routers and mix networks, offer anonymizing protection for users' identities, they cannot preserve the privacy of the users' queries. For the front running example, the user could tunnel the query through Tor [12] to preserve the privacy of his or her network address. Nevertheless, the server could still observe the user's desired domain name, and launch a successful front running attack.

The development of a practical PIR-based technique for protecting query privacy offers users and service providers an attractive value proposition. Users are increasingly aware of the problem of privacy and the need to maintain privacy in their online activities. The growing awareness is partly due to increased dependence on the Internet for performing daily activities — including online banking, Twittering, and social networking — and partly because of the rising trend of online privacy invasion. Privacy-conscious users will accept a service built on PIR for query privacy protection because no currently deployed security or privacy mechanism offers the needed protection; they will likely be willing to trade off query performance for query privacy and even pay to subscribe for such a service. Similarly, service providers may adopt such a system because of its potential for revenue generation through subscriptions and ad displays. As more Internet users value privacy, most online businesses would be motivated to embrace privacy-preserving technologies that can improve their competitiveness to win this growing user population. Since the protection of a user's identity is not a problem addressed by PIR, existing service models relying on service providers being able to identify a user for the purpose of targeted ads will not be disabled by this proposal. In other words, protection of query privacy will provide additional revenue generation opportunities for these service providers, while still allowing for the utilization of information collected through other means to send targeted ads to the users. Thus, users and service providers have plausible incentives to use a PIR-based solution for maintaining query privacy. In addition, the very existence of a practical privacy-preserving database query technique could be enough to persuade privacy legislators that it is reasonable to demand that certain sorts of databases enforce privacy policies, since it is possible to deploy these techniques without severely limiting the utility of such databases.

However, the rudimentary data access model of PIR is a limiting factor in deploying successful PIR-based systems. These models are limited to retrieving a single bit, a block of bits [3, 9, 18], or a textual keyword [8]. There is therefore a need for an extension to a more expressive data access model, and to a model that enables data retrieval from structured data sources, such as from a relational database. We address this need by integrating PIR with the widely deployed SQL.

Dynamic SQL is an incomplete SQL statement within a software system, meant to be fully constructed and executed at runtime [26]. It requires only a single compilation that *prepares* it for subsequent executions. It is therefore a flexible, efficient, and secure way of using SQL in software systems. We observe that the shape or textual content of an SQL query prepared within a system is not private, but the constants the user supplies at runtime are private, and must be protected. For domain name registration, the textual content of the query is exposed to the database, but only the textual keyword for the domain name is really private. For example, the *shape* of the dynamic query in Listing 1 is not private; the question mark `?` is used as a placeholder for a private value to be provided before the query is executed at runtime.

Listing 1 Example Dynamic SQL query (database schema as in [22])

```
SELECT t1.domain, t1.expiry, t2.contact
FROM regdomains t1, registrar t2
WHERE (t1.reg_id = t2.reg_id) AND (t1.domain = ? )
```

Our approach to preserving query privacy over a relational database is based on hiding such private constants of a query. The client sends a *desensitized* version of the prepared SQL query appropriately modified to remove private information. The database executes this public SQL query, and generates appropriate cached indices to support further rounds of interaction with the client. The client subsequently performs a number of keyword-based PIR operations [8] using the value for the placeholders against the indices to obtain the result for the query.

None of the existing proposals related to enabling privacy-preserving queries and robust data access models for private information retrieval makes the noted observation about the privacy of constants within an otherwise-public query. These include techniques that eliminate database optimization by localizing query processing to the user's computer [23], problems on querying Database-as-a-Service [16, 15], those that require an encrypted database before permitting private data access [25], and those restricted to simple keyword search on textual data sources [4]. This observation is crucial for preserving the expressiveness and benefits of SQL, and for keeping the interface between a database and existing software systems from changing while building in support for user query privacy. Our approach improves over previous work with additional database optimization opportunities and fewer PIR operations needed to retrieve data. To the best of our knowledge, we are the first to propose a practical technique that leverages PIR to preserve the privacy of sensitive information in an SQL query over existing commercial and open-source relational database systems.

Our contributions. We address the problem of preserving the privacy of sensitive information within an SQL query using PIR. In doing this, we address two obstacles to deploying successful PIR-based systems. First, we develop a generic data access model for private information retrieval from a relational database using SQL. We show how to hide sensitive data within a query and how to use

PIR to retrieve data from a relational database. Second, we develop an approach for embedding PIR schemes into the well-established context and organization of relational database systems. It has been argued that performing a trivial PIR operation, which involves having a database send its entire data to the user, and having the user select the item of interest, is more efficient than running a computational PIR scheme [1, 27]; however, information-theoretic PIR schemes are much more efficient. We show how the latter PIR schemes can be applied in realistic scenarios, achieving both efficiency and query expressivity. Since relational databases and SQL are the most influential of all database models and query languages, we argue that many realistic systems needing query privacy protection will find our approach quite useful.

The rest of this paper is organized as follows: Section 2 provides background information on PIR and database indexing. Section 3 discusses related work, while Section 4 details the threat model, security, and assumptions for the paper. Section 5 provides a description of our approach. Section 6 gives an overview of the prototype implementation, results of microbenchmarking and the experiment used to evaluate this prototype in greater depth. Section 7 concludes the paper and suggests some future work.

2 Preliminaries

2.1 Private Information Retrieval (PIR)

PIR provides a means to retrieve data from a database without revealing any information about which item is retrieved. In its simplest form, the database stores an n-bit string X, organized as r data blocks, each of size b bits. The user's private input or query is an index $i \in \{1, ..., r\}$ representing the i^{th} data block. A trivial solution for PIR is for the database to send all r blocks to the user and have the user select the block of interest at index i (i.e., X_i), but this carries a very poor communication complexity.

The three important requirements for any PIR scheme are correctness (returns the correct block X_i to the user), privacy (leaks no information to the database about i and X_i) and non-triviality (communication complexity is sublinear in n) [10]. An additional requirement, which is not often addressed in the published literature, is implementation (i.e., computational) efficiency [1, 27]. While the performance of information-theoretic PIR schemes are generally better [14], this neglect of computational overhead has led to single-database PIR schemes that are slow for large databases [27]. On the other hand, multi-server information-theoretic PIR schemes are much more efficient than the trivial solution and their use is justified in situations where the user lacks the bandwidth and local storage for the trivial download of data. Recent attempts at building practical single-database PIR [31] using general-purpose secure coprocessors offers several orders of magnitude improvement in performance. Nevertheless, the potential application of PIR in several practical domains has been largely unrealized with no "fruitful" or "real world" practical application.

A related cryptographic construction to PIR is *oblivious transfer* (OT) [20, 21]. In OT, a database (or sender) transmits some of its items to a user (or chooser), in a manner that preserves their mutual privacy. The database has assurance that the user does not learn any information beyond what he or she is entitled to, and the user has assurance that the database is unaware of which particular items it received. OT and the related *Symmetric* PIR (SPIR) [19] can thus be seen to be generalizations of PIR. Those protocols could easily be used in place of PIR in our work, with the concomitant extra computational cost.

2.2 Indexing

Data can be indexed by a key formed either from the values of one or more attributes or from hashes (generally not cryptographic hashes) of those values. Indices are typically organized into tree structures, such as B^+ trees where internal or non-leaf nodes do not contain data; they only maintain references to children or leaf nodes. Data are either stored in the leaf nodes, or the leaf nodes maintain references to the corresponding tuples (i.e., records) in the database. Furthermore, the leaf nodes of B^+ trees may be linked together to enable sequential data access during range queries over the index; *range queries* return all data with key values in a specified range.

Hashed indices are specifically useful for *point queries*, which return a single data item for a given key. For many situations where efficient retrieval over a set of unique keys is needed, hashed indices are preferred over B^+ tree indices. However, it is challenging to generate hash functions that will hash each key to a unique hash value. Many hashed indices used in commercial databases, for this reason, use data partitioning (bucketization) [16] techniques to hash a range of values to a single bucket, instead of to individual buckets. Recent advances [5, 6] in *perfect hash functions (PHF)* have produced a family of hash functions that can efficiently map a large set of n key values (on the order of billions) to a set of m integers without collisions, where n is less than or equal to m.

3 Related Work

A common assumption for PIR schemes is that the user knows the index or address of the item to be retrieved. However, Chor et al. [8] proposed a way to access data with PIR using keyword searches over three data structures: binary search tree, trie and perfect hashing. Our work extends keyword-based PIR to B^+ trees and PHF. In addition, we provide an implemented system and combine the technique with the expressive SQL. The technique in [8] neither explores B^+ trees nor considers executing SQL queries using keyword-based PIR.

Reardon et al. [23] similarly explore using SQL for private information retrieval, and proposed the TransPIR prototype system. This work is the closest to our proposal and will be used as the basis for comparisons. TransPIR performs traditional database functions (such as parsing and optimization) locally on the

client; it uses PIR for data block retrieval from the database server, whose function has been reduced to a block-serving PIR server. The benefit of TransPIR is that the database will not learn any information even about the textual content of the user's query. The drawbacks are poor query performance because the database is unable to perform any optimization, and the lack of interoperability with any existing relational database system.

An interesting attempt to build a practical pseudonymous message retrieval system using the technique of PIR is presented in [24]. The system, known as the Pynchon Gate, helps preserve the anonymity of users as they privately retrieve messages using pseudonyms from a centralized server. Unlike our use of PIR to preserve a user's query privacy, the goal of the Pynchon Gate is to maintain privacy for users' identities. It does this by ensuring the messages a user retrieves cannot be linked to his or her pseudonym. The construction resists traffic analysis, though users may need to perform some dummy PIR queries to prevent a passive observer from learning the number of messages she has received.

4 Threat Model, Security and Assumptions

4.1 Security and adversary capabilities

Our main assumption is that the shape of SQL queries submitted by the users is public or known to the database administrator. Applicable practical scenarios include design-time specification of dynamic SQL by programmers, who expect the users to supply sensitive constants at runtime. Moreover, the database schema and all dynamic SQL queries expected to be submitted to, for example, a patent database, are not really hidden from the patent database administrator. Simultaneous protection of both the shape and constants of a query are outside of the scope of this work, and would likely require treating the database management system as other than a black box.

The approach presented in this paper is sufficiently generic to allow an application to rely on any block-based PIR system, including single-server, multi-server, and coprocessor-assisted variants. We assume an adversary with the same capability as that assumed for the underlying PIR protocol. The two common adversary capabilities considered in theoretical private information retrieval schemes are the curious passive adversary and the byzantine adversary [3, 9]. Either of these adversaries can be a database administrator or any other insider to a PIR server.

A curious passive adversary can observe PIR-encoded queries, but should be incapable of decoding the content. In addition, it should not be possible to differentiate between queries or identify the data that makes up the result of a query. In our context, the information this adversary can observe is the desensitized SQL query from the client and the PIR queries. The information obtained from the desensitized query does not compromise the privacy of the user's query, since it does not contain any private constants. Similarly, the adversary cannot obtain any information from the PIR queries because PIR protocols are designed to be resistant against an adversary of this capability.

A byzantine adversary with additional capabilities is assumed for some multi-server PIR protocols [3, 14]. In this model, the data in some of the servers could be outdated, or some of the servers could be down, malfunctioning or malicious. Nevertheless, the client is still able to compute the correct result and determine which servers misbehaved, and the servers are still unable to learn the client's query. Again, in our specific context, the adversary may compromise some of the servers in a multi-server PIR scenario by generating and obtaining the result for a substitute fake query or executing the original query on these servers, but modifying some of the tuples in the results arbitrarily. The adversary may respond to a PIR request with a corrupted query result or even desist from acting on the request. Nevertheless, all of these active attack scenarios can be effectively mitigated with a byzantine-robust multi-server PIR scheme.

4.2 Data size assumptions

We service PIR requests using indexed data extracted from relational databases. The size of these data depends on the number of tuples resulting from the desensitized query. We note that even in the event that this *desensitized* query yields a small number of tuples (including just one), the privacy of the *sensitive part* of the SQL query *is not compromised*. The properties of PIR ensure that the adversary gains no information about the sensitive constants from observing the PIR protocol, over what he already knew by observing the desensitized query.

On the other hand, many database schemas are designed in a way that a number of relations will contain very few rows of data, all of which are meant to be retrieved and used by every user. Therefore, it is pointless to perform PIR operations on these items, since every user is expected to retrieve them all at some point. The adversary does not violate a user's query privacy by observing this public retrieval.

4.3 Avoiding server collusion

Information-theoretic PIR is generally more computationally efficient than computational PIR, but requires that the servers not collude if privacy is to be preserved; this is the same assumption commonly made in other privacy-preserving technologies, such as mix networks [7] and Tor [12]. We present scenarios in which collusion among servers is unlikely, yielding an opportunity to use the more efficient information-theoretic PIR.

The first scenario is when several independent service providers host a copy of the database. This applies to naturally distributed databases, such as Internet domain registries. In this particular instance, the problem of colluding servers is mitigated by practical business concerns. Realistically, the Internet domain database is maintained by different geographically dispersed organizations that are independent of the registrars that a user may query. However, different registrars would be responsible for the content's distribution to end users as well as integration of partners through banner ads and promotions. Since the registrars are operating in the same line of business where they compete to win users and

deliver domain registry services, as well as having their own advertising models to reap economic benefits, there is no real incentive to collude in order to break the privacy of any user. In this model, it is feasible that a user would perform a domain name registration query on multiple registrars' servers concurrently. The user would then combine the results, without fear of the queries revealing its content. Additionally, individual service agreements can foreclose any chance of collusion with a third party on legal grounds. Users then enjoy greater confidence in using the service, and the registrars in turn can capitalize on revenue generation opportunities such as pay-per-use subscriptions and revenue-sharing ad opportunities.

The second scenario that offers less danger of collusion is when the query needs to be private only for a short time. In this case, the user may be comfortable with knowing that by the time the servers collude in order to learn her query, the query's privacy is no longer required.

Note that even in scenarios where collusion cannot be forestalled, our system can still use any computational PIR protocol; recent such protocols [1, 31] offer considerable efficiency improvements over previous work in the area.

5 Hiding Sensitive Constants

5.1 Overview

Our approach is to preserve the privacy of sensitive data within the WHERE and HAVING predicates of an SQL query. For brevity, we will focus on the WHERE clause; a similar processing procedure applies to the HAVING clause. This may require the user (or application) to specify the constants that may be sensitive. For the example query in Listing 2, the domain name and the creation date may be sensitive.

Our approach splits the processing of SQL queries containing sensitive data into two stages. In the first stage, the client computes a public subquery, which is simply the original query that has been stripped of the predicate conditions containing sensitive data. The client sends this subquery to the server, and the server executes it to obtain a result for the subquery. The desired result for the original query is contained within the subquery result, but the database is not aware of the particular tuples that are of interest.

In the second stage, the client performs PIR operations to retrieve the tuples of interest from the subquery result. To enable this, the database creates a cached index on the subquery result and sends metadata for querying the index to the

Listing 2 Example query with a WHERE clause featuring sensitive constants.

```
SELECT t1.contact, t1.email, t2.created, t2.expiry
FROM registrar t1, regdomains t2
WHERE (t1.reg_id = t2.reg_id) AND (t2.created > 20090101) AND
      (t2.domain = 'anydomain.com')
```

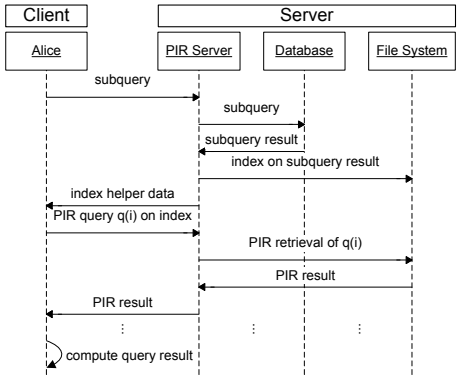

Fig. 1. A sequence diagram for evaluating Alice's private SQL query using PIR.

client. The client subsequently performs PIR retrievals on the index and finally combines the retrieved items to build the result for the original query.

The important benefits of this approach as compared with the previous approach [23] are the optimizations realizable from having the database execute the non-private subquery, and the fewer number of PIR operations required to retrieve the data of interest. In addition, the PIR operations are performed against a cached index which will usually be smaller than the complete database. This is particularly true if there are joins and non-private conditions in the WHERE clause that constrain the tuples in the query result. In particular, a single PIR query is needed for point queries on hash table indices, while range queries on B^+ tree indices are performed on fewer data blocks. Figure 1 illustrates the sequence of events during a query evaluation.

We note that often, the non-private subqueries will be common to many users, and the database does not need to execute them every time a user makes a request. Nevertheless, our algorithm details, presented next in Section 5.2, show the steps for processing a subquery and generating indices. Such details are useful in an *ad hoc* environment, where the shape of a query is unknown to the database *a priori*; each user writes his or her own query as needed. Our assumption is that revealing the shape of a query will not violate users' privacy (see Section 4).

5.2 Algorithm

We describe our algorithm with an example by assuming an information-theoretic PIR setup with two replicated servers. We focus on hiding sensitive constants in the predicates of the WHERE clause. The algorithm details for the SELECT query in Listing 2 follows. We assume the date `20090101` and the domain `anydomain.com` are private.

Step 1: The client builds an attribute list, a constraint list, and a desensitized SELECT query, using the attribute names and the WHERE conditions of the input query. We refer to the desensitized query as a *subquery*.

To begin, initialize the attribute list to the attribute names in the query's SELECT clause, the constraint list to be empty, and the subquery to the SELECT and FROM clauses of the original query.

- *Attribute list:* {`t1.contact`, `t1.email`, `t2.created`, `t2.expiry`}
- *Constraint list:* {}
- *Subquery:*
```
SELECT t1.contact, t1.email, t2.created, t2.expiry
FROM registrar t1, regdomains t2
```

Next, consider each WHERE condition in turn. If a condition features a private constant, then add the attribute name to the *attribute list* (if not already in the list), and add (attribute name, constant value, operator) to the *constraint list*. Otherwise, add the condition to the subquery.

On completing the above steps, the attribute list, constraint list, and subquery with reduced conditions for the input query become:

- *Att. list:* {`t1.contact`, `t1.email`, `t2.created`, `t2.expiry`, `t2.domain`}
- *Con. list:* {`(t2.created,20090101,>),(t2.domain,'anydomain.com',=)`}
- *Subquery:*
```
SELECT t1.contact,t1.email,t2.created,t2.expiry,t2.domain
FROM registrar t1, regdomains t2 WHERE (t1.reg_id = t2.reg_id)
```

Step 2: The client sends the subquery, a key attribute name, and an index file type to each server.

The key attribute name is selected from the attribute names in the constraint list — `t2.created, t2.domain` in our example. The choice may either be random, made by the application designer, or determined by a client optimizer component with some domain knowledge that could enable it to make an optimal choice. One way to make a good choice is to consider the *selectivity* — the ratio of the number of distinct values taken to the total number of tuples — expected for each constraint list attribute, and then choose the one that is most selective. This ensures the selection of attributes with unique key values before less selective attributes. For example, in a patent database, the patent number is a better choice for a key than the author's gender. A poor choice of key can lead to more rounds of PIR queries than necessary. Point queries on a unique key attribute can be completed with a single PIR query. Similarly, a good choice of key will reduce the number of PIR queries for range queries. For the example query, we choose `t2.domain` as the key attribute name.

For the index file type, either a PHF or a B^+ tree index type is specified. Other index structures may be possible, with additional investigation, but these are the ones we currently support. More details on the selection of index types is provided below.

Step 3: Each server: executes the subquery on its relational database, generates a cached index of the specified type on the subquery result, using the key attribute name, and returns metadata for searching the indices to the client.

The server computes the size of the subquery result. If it can send the entire result more cheaply than performing PIR operations on it, it does so. Otherwise,

it proceeds with the index generation. For hash table indices, the server first computes the perfect hash functions for the key attribute values. Then it evaluates each key and inserts each tuple into a hash table. The metadata that is returned to the client for hash-based indices consists of the PHF parameters, the count of tuples in the hash table, and some PIR-specific initialization parameters.

For B^+ tree indices, the server bulk inserts the subquery result into a new B^+ tree index file. B^+ tree bulk insertion algorithms provide a high-speed technique for building a tree from existing data [2]. The server also returns metadata to the client, including the size of the tree and its first data block (the root). Generated indices are stored in a disk cache external to the database.

Step 4: The client receives the responses from the servers and verifies they are of the appropriate length. For a byzantine robust multi-server PIR, a client may choose to proceed in spite of errors resulting from non-responding servers or from responses that are of inconsistent length.

Next, the client performs one or more keyword-based PIR queries, using the value associated with the key attribute name from the constraint list, and builds the desired query result from the data retrieved with PIR.

The encoding of a private constant in a PIR query proceeds as follows. For PIR queries over a hash-based index, the client computes the hash for the private constant using the PHF functions derived from the metadata[1]. This hash is also the block number in the hash table index on the servers. This block number is input to the PIR scheme to compute the PIR query for each server. For a B^+ tree index, the user compares the private value for the key attribute with the values in the root of the tree. The root of the tree is extracted from the metadata it receives from the server. Each key value in this root maintains block numbers for the children blocks or nodes. The block number corresponding to the appropriate child node will be the input to the PIR scheme.

For hash-based indices, a single PIR query is sufficient to retrieve the block containing the data of interest from the hash table. For B^+ tree indices, however, the client uses PIR to traverse the tree. Each block can hold some number m of keys, and at a block level, the B^+ tree can be considered an m-ary tree. The client has already been sent the root block of the tree, which contains the top m keys. Using this information, the client can perform a single PIR block query to fetch one of the m blocks so referenced. It repeats this process until it reaches the leaves of the tree, at which point it fetches the required data with further PIR queries. The actual number of PIR queries depends on the height of the (balanced) tree, and the number of tuples in the result set. Traversals of B^+ tree indices with our approach are oblivious in that they leak no information about nodes' access pattern; we realize retrieval of a node's data as a PIR operation over the data set of all nodes in the tree. In other words, it does not matter which particular branch of a B^+ tree is the location for the next block to be retrieved. We do not restrict PIR operations to the subset of blocks in the subtree rooted

[1] Using the CMPH Library [5] for example, the client saves the PHF data from the metadata into a file. It reopens this file and uses it to compute a hash by following appropriate API call sequences.

at that branch. Instead, each PIR operation considers the set of blocks in the entire B^+ tree. Range queries that retrieve data from different subtrees leak no information about to which subtree a particular piece of data belongs. The only information the server learns is the number of blocks retrieved by such a query. Therefore, specific implementations may utilize dummy queries to prevent the server from leaning the amount of useful data retrieved by a query [24].

To compute the final query result, the client applies the other private conditions in the constraint list to the result obtained with PIR. For the example query, the client filters out all tuples with `t2.created` not greater than `20090101` from the tuple data returned with PIR. The remaining tuples give the final query result.

Capabilities for dealing with complex queries can be built into the client. For example, it may be more efficient to request a single index keyed on the concatenation of two attributes than separate indices. If the client requests separate indices, it will subsequently perform PIR queries on each of those indices, using the private value associated with each attribute from the constraint list. Finally, the client combines the partial results obtained from the queries with set operations (union, intersection), and performs local filtering on the combined result, using private constant values for any remaining conditions in the constraint list to compute the final query result. The client thus needs query-optimization capabilities in addition to the regular query optimization performed by the server.

6 Implementation and Microbenchmarks

6.1 Implementation

We developed a prototype implementation of our algorithm to hide the sensitive portions of SQL queries using generally available open source C++ libraries and databases. We developed a command-line tool to act as the client, and a server-side database adapter to provide the functions of a PIR server. For the PIR functions, we used the Percy++ PIR Library [13, 14], which offers three varieties of privacy protection: computational, information theoretic and hybrid (a combination of both). We extended Percy++ to support keyword-based PIR. For generating hash table indices for point queries, we used the C Minimal Perfect Hash (CMPH) Library [5, 6], version 0.9. We used the API for CMPH to generate minimum perfect hash functions for large data sets from query results; these perfect hash functions require small amounts of disk storage per key. For building B^+ tree indices for range queries on large data sets, we used the Transparent Parallel I/O Environment (TPIE) Library [11, 30]. Finally, we base the implementation on the MySQL [28] relational database, version 5.1.37-1ubuntu5.1.

6.2 Experimental setup

We began evaluating our prototype implementation using a set of six whois-style queries from Reardon et al. [23], which is the most appropriate existing

microbenchmark for our approach. We explored tests using industry-standard database benchmarks, such as the Transaction Processing Performance Council (TPC) [29] benchmarks, and open-source benchmarking kits such as Open Source Development Labs Database Test Suite (OSDL DTS) [32], but none of the tests from these benchmarks is suitable for evaluating our prototype, as their test databases cannot be readily fitted into a scenario that would make applying PIR meaningful. For example, a database schema that is based on completing online orders will only serve very limited purpose to our goal of protecting the privacy of sensitive information within a query.

We ran the microbenchmark tests using two whois-style data sets, similar to those generated for the evaluation of TransPIR [23]. The smaller data set consists of 10^6 domain name registration tuples, and 0.75×10^6 registrar and registrant contact information tuples. The second data set similarly consists of 4×10^6 and 3×10^6 tuples respectively. We describe the two database relations and the evaluation queries, as well as the results for the smaller data set, in the extended version [22].

In addition to the microbenchmarks, we performed an experiment to evaluate the behaviour of our prototype on complex input queries, such as aggregate queries, BETWEEN and LIKE queries, and queries with multiple WHERE clause conditions and joins. Each of these complex queries has varying privacy requirements for its sensitive constants.

We ran the all experiments on a server with two quad-core 2.50 GHz Intel Xeon E5420 CPUs, 8 GB RAM, and running Ubuntu Linux 9.10. We used the information-theoretic PIR support of Percy++, with two database replicas. The server also runs a local installation of a MySQL database.

6.3 Result overview

The results from our evaluation indicate that while our current prototype incurs some storage and computational costs over non-private queries, the costs seem entirely acceptable for the added privacy benefit (see Tables 1 and 2). In addition to being able to deal with complex queries and leverage database optimization opportunities, our prototype performs much better than the TransPIR prototype from Reardon et al. [23] — between 7 and 480 times faster for equivalent data sets. The most indicative factor of performance improvements with our prototype is the reduction in the number of PIR queries in most cases. Other factors that may affect the validity of the result, such as variations in implementation libraries, are assumed to have negligible impact on performance. Our work is based on the same PIR library as that of [23]. Our comparison is based on the measurements we took by compiling and running the code for TransPIR on the same experimental hardware platform as our prototype. We also used the same underlying PIR library as TransPIR.

Table 1. Experimental results for microbenchmark tests compared with those of Reardon et al. [23]. **BTREE** = timing for our B^+ tree prototype, **HASH** = timing for our hash table prototype, and **TransPIR** = timing from TransPIR [23]; **Time** = time to evaluate private query, **PIRs** = number of PIR operations performed, **Tuples** = count of rows in query result, **QI** = timing for subquery execution and index generation, **Xfer** = total data transfer between the client and the two PIR servers.

Query	Approach	Time (s)	PIRs	Tuples	QI (s)	Xfer (KB)
Q1	**HASH**	2	1	1	16	128
	BTREE	4	3	1	38	384
	TransPIR	25	2	1	1,017	256
Q2	**BTREE**	5	4	80	32	512
	TransPIR	999	83	80	1,017	10,624
Q3	**BTREE**	5	4	168	32	512
	TransPIR	2,055	171	168	1,017	21,888
Q4	**BTREE**	6	5	236	37	640
	TransPIR	2,885	240	236	1,017	30,720
Q5	**BTREE**	5	3	1	67	384
	TransPIR	37	3	1	1,017	384
Q6[†]	**BTREE**	5	4	168	66	512
	TransPIR	3,087	253	127	—[†]	32,384

6.4 Microbenchmark and complex query experiments

For the benchmark tests, we obtained measurements for the time to execute the private query, the number of PIR queries performed, the number of tuples in the query results, the time to execute the subquery and generate the cached index, and the total data transfer between the client and the two PIR servers.

Table 1 shows the results of the experiment. The cost of indexing (QI) can be amortized over multiple queries. The indexing measurements for BTREE (and HASH) consist of the time spent retrieving data from the database (subquery execution), writing the data (subquery result) to a file and building an index from this file. Since TransPIR is not integrated with any relational database, it does not incur the same database retrieval and file writing costs. However, TransPIR incurs a one-time preprocessing cost (QI) which prepares the database for subsequent query runs. Comparing this cost to its indexing counterpart with our BTREE and HASH prototypes shows that our methods are over an order of magnitude faster.

For the experiment on queries with complex conditions, we used a number of synthetic query scenarios having different requirements for privacy (see [22] for details). The measurements, as reported in Table 2, show execution duration for the original query without privacy provision over the MySQL database, and several other measurements taken from within our prototype using a B^+ tree index.

[†] We reproduced TransPIR's measurements from [23] for query Q6 because we could not get TransPIR to run Q6 due to program errors. The '—' under QI indicates measurements missing from [23]

6.5 Discussion

The empirical results for the benchmark tests reflect the benefit of our approach. For all of the tests, we mostly base our comparison on the timings for query evaluation with PIR (Time), and sometimes on the index generation timings (QI). The time to transfer data between the client and the servers is directly proportional to the amount of data (Xfer), but we will not use it for comparison purposes because the test queries were not run over a network.

Our hash index (HASH) prototype performs the best for query Q1, followed by our B^+ tree (BTREE) prototype. The query of Q1 is a point query having a single condition on the domain name attribute.

Query Q2 is a point query on the `expiry_date` attribute, with the query result expected to have multiple tuples. The number of PIR queries required to evaluate Q2 with BTREE is 5% of the number required by TransPIR. A similar trend is repeated for Q3, Q4 and Q6. Note that the HASH prototype could not be used for Q2 because hash indices accept unique keys only; it can only return a single tuple in its query result.

Query Q3 is a range query on `expiry_date`. Our BTREE prototype was approximately 411 times faster than TransPIR. Of note is the large number of PIR queries that TransPIR needs to evaluate the query; our BTREE prototype requires only 2% of that number. We observed a similar trend for Q4, where BTREE was 480 times faster. This query features two conditions in the SQL WHERE clause. The combined measured time for BTREE — the time taken to both build an index to support the query and to run the query itself — is still 67 times faster than the time it takes TransPIR to execute the query alone.

Query Q5 is a point query with a single join. It took BTREE only about 14% of the time it took TransPIR. We observed the time our BTREE spent in executing the subquery to dominate; only a small fraction of the time is spent building the B^+ tree index.

Our BTREE prototype similarly performs faster for Q6, with an order of magnitude similar to Q2, Q3, and Q4.

In all of the benchmark queries, the proposed approach performs better than TransPIR because it leverages database optimization opportunities, such as for the processing of subqueries. In contrast, TransPIR assumes a type of block-serving database that cannot give any optimization opportunity. Therefore, in our system, the client is relieved from having to perform many traditional database functions, such as query processing, in addition to its regular PIR client functions.

Results for queries with complex conditions. We see from Table 2 that in most cases, the cost to evaluate the subquery and create the index dominates the total time to privately evaluate the query (BTREE), while the time to evaluate the query on the already-built index (Time) is minor. An exception is CQ2, which has a relatively small subquery result (rTuples), while having to do dozens of (consequently smaller) PIR operations to return thousands of results to the overall range query. Note that in all but CQ2, the time to privately evaluate the query on the already-built index is at most a few seconds longer than performing

Table 2. Measurements taken from executing five complex SQL queries with varying requirements for privacy. **oQm** = timing for executing original query directly against the database, **BTREE** = overall timing for meeting privacy requirements with our B^+ tree prototype, **Time** = time to evaluate private query within BTREE, **PIRs** = number of PIR operations performed, **Tuples** = number of records in final query result, **rTuples** = number of indexed records in subquery result, **Xfer** = total data transfer between the client and the two PIR servers, **Size** = storage for index.

Query	oQm (s)	BTREE (s)	Time (s)	PIRs	Tuples	rTuples	Xfer (KB)	Size (MB)
CQ1	**2**	**31**	**2**	3	1	1,753,144	384	579.63
CQ2	**1**	**15**	**13**	41	3,716	72,568	5,248	25.13
CQ3	**0**	**80**	**3**	3	1	631,806	384	209.38
CQ4	**2**	**25**	**5**	3	1	1,050,300	384	348.63
CQ5	**2**	**69**	**3**	3	6	4,000,000	384	1,324.13

the query with no privacy at all; this underscores the advantage of using cached indices.

We note from our results that it is much more costly to have the client simply download the cached indices. We observe, for example, that it will take about 5 times as long, for a user with 10 Mbps download bandwidth, to download the index for CQ5. Moreover, this trivial download of data is impractical for devices with low bandwidth and storage (e.g., mobile devices).

One way to improve query performance is by revealing a prefix or suffix of the sensitive keyword in a query. Revealing a substring of a keyword helps to constrain the result set that will be indexed and retrieved with PIR. Making this trade-off decision in a privacy-friendly manner necessarily requires some knowledge of the data distribution in terms of the number of tuples there are for each value in the domain of values for a sensitive constant. These information can be included in the metadata a server sends to the client and the client can make this trade-off decision on behalf of the user based on the user's preset preferences. We are considering this extension as part of our future work.

6.6 Limitations

Our approach can preserve the privacy of sensitive data within the WHERE and HAVING clauses of an SQL query, with the exception of complex LIKE query expressions, negated conditions with sensitive constants, and SELECT nested queries within a WHERE clause. The complexity of complex search strings for LIKE queries, such as (LIKE 'do%abs%.c%m'), and negated WHERE clause conditions, such as (NOT registrant = 45444) are beyond the current capability of keyword-based PIR. Our solution to dealing with these conditions in a privacy-friendly manner is to compute them on the client, after the data for the computation has been retrieved with PIR; converting NOT = queries into their equivalent range queries is generally less efficient than our proposed client-based evaluation method. In addition, our prototype cannot process a nested query within a WHERE clause. We propose that the same processing described for

a general SQL query be recursively applied for nested queries in the WHERE clause. The result obtained from a nested query will become an input to the client optimizer, for recursively computing the enclosing query for the next round. There is need for further investigation of the approach for nested queries returning large result sets and for deeply nested queries.

7 Conclusion and Future Work

We have provided a privacy mechanism that leverages private information retrieval to preserve the privacy of sensitive constants in an SQL query. We described techniques to hide sensitive constants found in the WHERE clause of an SQL query, and to retrieve data from hash table and B^+ tree indices using a private information retrieval scheme. We developed a prototype privacy mechanism for our approach offering practical keyword-based PIR and enabled a practical transition from bit- and block-based PIR to SQL-enabled PIR. We evaluated the feasibility of our approach with experiments. The results of the experiments indicate our approach incurs reasonable performance and storage demands, considering the added advantage of being able to perform private SQL queries. We hope that our work will provide valuable insight on how to preserve the privacy of sensitive information for many existing and future database applications.

Future work can improve on some limitations of our prototype, such as the processing of nested queries and enhancing the client to use statistical information on the data distribution to enhance privacy. The same technique proposed in this paper can be extended to preserve the privacy of sensitive information for other query systems, such as URL query, XQuery, SPARQL and LINQ.

Acknowledgments

We would like to thank Urs Hengartner, Ryan Henry, Aniket Kate, Can Tang, Mashael AlSabah, John Akinyemi, Carol Fung, Meredith L. Patterson, and the anonymous reviewers for their helpful comments for improving this paper. We also gratefully acknowledge NSERC and MITACS for funding this research.

References

1. C. Aguilar-Melchor and P. Gaborit. A Lattice-Based Computationally-Efficient Private Information Retrieval Protocol. Cryptol. ePrint Arch., Report 446, 2007.
2. L. Arge, O. Procopiuc, and J. S. Vitter. Implementing I/O-efficient Data Structures Using TPIE. In *Annual European Symposium on Algorithms*, pages 88–100, 2002.
3. A. Beimel and Y. Stahl. Robust Information-Theoretic Private Information Retrieval. *J. Cryptol.*, 20(3):295–321, 2007.
4. J. Bethencourt, D. Song, and B. Waters. New Techniques for Private Stream Searching. *ACM Trans. Inf. Syst. Secur.*, 12(3):1–32, 2009.
5. F. C. Botelho, D. Reis, and N. Ziviani. CMPH: C minimal perfect hashing library on SourceForge. http://cmph.sourceforge.net/.

6. F. C. Botelho and N. Ziviani. External perfect hashing for very large key sets. In *ACM CIKM*, pages 653–662, 2007.
7. D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–90, 1981.
8. B. Chor, N. Gilboa, and M. Naor. Private information retrieval by keywords. Technical Report TR CS0917, Dept. of Computer Science, Technion, Israel, 1997.
9. B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *FOCS*, pages 41–50, Oct 1995.
10. G. D. Crescenzo. Towards Practical Private Information Retrieval. Achieving Practical Private Information Retrieval (Panel @ Securecomm 2006), Aug. 2006.
11. Department of Computer Science at Duke University. The TPIE (Templated Portable I/O Environment). http://madalgo.au.dk/Trac-tpie/.
12. R. Dingledine, N. Mathewson, and P. Syverson. Tor: the second-generation onion router. In *USENIX Security Symposium*, pages 21–21, 2004.
13. I. Goldberg. Percy++ project on SourceForge. http://percy.sourceforge.net/.
14. I. Goldberg. Improving the Robustness of Private Information Retrieval. In *IEEE Symposium on Security and Privacy*, pages 131–148, 2007.
15. H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *ACM SIGMOD*, pages 216–227, 2002.
16. B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. In *VLDB*, pages 720–731, 2004.
17. ICANN Security and Stability Advisory Committee (SSAC). Report on Domain Name Front Running, February 2008.
18. E. Kushilevitz and R. Ostrovsky. Replication is not needed: single database, computationally-private information retrieval. In *FOCS*, page 364, 1997.
19. S. K. Mishra and P. Sarkar. Symmetrically Private Information Retrieval. In *INDOCRYPT*, pages 225–236, 2000.
20. M. Naor and B. Pinkas. Oblivious transfer and polynomial evaluation. In *ACM Symposium on Theory of Computing*, pages 245–254, 1999.
21. M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *ACM-SIAM SODA*, pages 448–457, 2001.
22. F. Olumofin and I. Goldberg. Privacy-preserving Queries over Relational Databases. Technical report, CACR 2009-37, University of Waterloo, 2009.
23. J. Reardon, J. Pound, and I. Goldberg. Relational-Complete Private Information Retrieval. Technical report, CACR 2007-34, University of Waterloo, 2007.
24. L. Sassaman, B. Cohen, and N. Mathewson. The Pynchon Gate: a Secure Method of Pseudonymous Mail Retrieval. In *ACM WPES*, pages 1–9, 2005.
25. E. Shi, J. Bethencourt, T.-H. H. Chan, D. Song, and A. Perrig. Multi-Dimensional Range Query over Encrypted Data. In *IEEE SSP*, pages 350–364, 2007.
26. A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, Inc., New York, NY, USA, 5th edition, 2005.
27. R. Sion and B. Carbunar. On the Computational Practicality of Private Information Retrieval. In *Network and Distributed Systems Security Symposium*, 2007.
28. Sun Microsystems. MySQL. http://www.mysql.com/.
29. Transaction Processing Performance Council. Benchmark C. http://www.tpc.org/.
30. D. E. Vengroff and J. Scott Vitter. Supporting I/O-efficient scientific computation in TPIE. In *IEEE Symp. on Parallel and Distributed Processing*, page 74, 1995.
31. P. Williams and R. Sion. Usable PIR. In *Network and Distributed System Security Symposium*. The Internet Society, 2008.
32. M. Wong and C. Thomas. Database Test Suite project on SourceForge. http://osdldbt.sourceforge.net/.