Privacy-preserving Queries over Relational Databases

Femi Olumofin
Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
fgolumof@cs.uwaterloo.ca

Ian Goldberg
Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
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.**

I. 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) [4], [13] 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* [26]. The registrar is motivated to engage in front running because of the revenue to be derived from reselling the domain at an inflated price, and from placing ads on the domain's landing page. 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. Although ICANN claims the practice of domain front running has subsided [26], 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. [35], 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 [17] 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.

To address the protection of the query, we study how client applications embed personal information into queries, particularly for systems that use SQL for data access. We focus on the protection of SQL queries over relational databases because such databases are widely deployed.

Our goal of preserving the privacy of sensitive information within an SQL query requires an extension to the rudimentary data access model of PIR. These models are limited to retrieving a single bit, a block of bits [4], [13], [28], or a textual keyword [12]. These theoretical primitives are a limiting factor in deploying successful PIR-based systems. 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.

Dynamic SQL is an incomplete SQL statement within a software system, meant to be fully constructed and executed at runtime [39]. It provides a flexible, efficient, and secure way of using SQL in software systems. The flexibility enables systems to construct and submit SQL queries to the database at runtime. Dynamic SQL is efficient because it requires only a single compilation that *prepares* the query for its subsequent executions. In addition, dynamic SQL is more secure because malicious SQL code injection is much more difficult. 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. Of note is the related observation made between parameterized SQL queries and parse tree validation [9], [23]. In this context, runtime parse trees obtained from combining user inputs with parameterized queries are validated to ensure consistency with parse trees for programmer-specified queries, thereby defeating SQL injection. Unlike valid inputs which only alter the semantics of a parse tree, SQL injection attempts to change both the syntax and semantics of a parse tree [24].

Listing 1 Example Dynamic SQL query (see Appendix A for the corresponding database schema)

```
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 [12] 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 [35], problems on querying Database-as-a-Service [25], [22], those that require an encrypted database before permitting private data access [38], and those restricted to simple keyword search on textual data sources [6]. 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], [40]; 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 II provides background information on PIR, the relational model, SQL, and database indexing. Section III discusses related work, while Section IV details the threat model,

security, and assumptions for the paper. Section V provides a description of the approach for hiding sensitive constants within an SQL query. We provide detailed discussions of the algorithm in Section VI. Section VII gives an overview of the prototype implementation and microbenchmarking results of this prototype privacy mechanism. Section VIII highlights results and discussions of the experiment used to evaluate the prototype in greater depth. Section IX concludes the paper and suggests some future work.

II. Preliminaries

A. 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, privacy and non-triviality [14]. The requirement of correctness ensures that the scheme returns the correct block X_i to the user. The requirement of privacy assures the scheme does not leak any information to the database about the user's private input i and the retrieved block X_i. The non-triviality requirement expects a communication complexity that is better than the trivial solution; that is, sublinear in n. An additional requirement, which is not often addressed in the published literature, is implementation efficiency. In fact, the literature has dedicated most attention to reducing communication complexity at the expense of computational complexity [1], [40]. While the performance of information-theoretic PIR schemes are generally better [21], this neglect of computational overhead has led to single-database PIR schemes that are slow for large databases [40]. 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 resources required for the trivial download of data. Recent attempts at building practical single-database PIR [45] using general-purpose secure co-processors 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.

When the PIR problem was first introduced in 1995 [13], it was proven that a better-than-trivial solution with information-theoretic privacy is impossible to achieve with a single database. Information-theoretic privacy ensures that the adversary cannot learn the user's query, regardless of its current or future computational abilities. Using at least two replicated databases, however, PIR schemes with information-theoretic privacy are possible, and sometimes hold attractive properties like robustness and byzantine robustness [21]. The first single-database PIR proposal was in 1997 [11]. This PIR scheme assures privacy against an adversary with limited computational capability only; i.e., polynomially bounded attackers. This type of privacy protection is known as computational privacy, and is a weaker notion than information-theoretic privacy. However, computational PIR (CPIR) [11], [28] offers the benefit of being able to field a single database, unlike information-theoretic PIR [4], [13] that requires replication and some form of restriction on how the databases can communicate.

Basic PIR schemes place no restriction on information leaked about other items in the database, which are not of interest to the user. However, an extension of PIR, known as *Symmetric* PIR (SPIR) [29], adds that restriction by insisting that a user learns *only* the result of her query. The restriction is crucial in situations where the database privacy is equally of concern.

Another cryptographic construction related to PIR is *oblivious transfer* (OT) [30], [31]. 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 oblivious or unaware of which particular items it received. OT and SPIR 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.

Freedman et al. [18] provides a solution for database search with keywords in various settings including OT, using oblivious polynomial evaluation and homomorphic encryption. However, each database tuple, which they referred to as a payload, still needs to be tagged with an appropriate keyword. The key improvements over earlier results [30], [31] is the preservation of privacy against a fixed number of queries after an initial setup, a fixed number of rounds for oblivious query evaluation, and the ability to deal with exponential domain sizes.

B. The relational model and SQL

The *relational model* forms the basis for data storage in many database systems. Data in this model is organized as a collection of tables and the relationships between them. Tables are also called *relations*. Each record or row of a relation is a *tuple*, and each column represents an *attribute*. SQL is a language for manipulating and retrieving data from the relations of a database. [39]

The basic form of an SQL query consists of the *SELECT*, *FROM*, and *WHERE* clauses (see Listing 2). The SELECT clause produces a relation consisting of the attributes in the set $a_1, a_2, ..., a_n$. The FROM clause performs a *cross product* (or *Cartesian product*) operation on the relations,

by combining each tuple of R_1 with each tuple of R_2 (and similarly for $R_3, \ldots, R_n$); each of the resulting tuples has all the attributes of all of the relations. The WHERE clause selects the tuples from the cross product that satisfy a given condition or predicate P. The predicate P is a boolean expression on constants and the attributes of $R_1, R_2, ..., R_n$ and involves comparison operators $=$, $<>$, $<$, $>$, $<=$, $>=$ as well as logical operators AND, OR, and NOT. Often, the predicate includes a join that further constrains the tuples for the cross product.

Listing 2 Basic form of an SQL query.

```
SELECT a_1, a_2, ..., a_n
FROM R_1, R_2, ..., R_n
WHERE P
```

Another clause of interest to our work is the *HAVING* clause. This clause is similar to the WHERE clause; however, it allows aggregate expressions, such as $SUM(*)$ and $COUNT(*)$, in its predicate expressions. In practice, the predicates of these two clauses constrain the result of a query to some selected tuples. For example, a predicate "domain = 'somedomain.com' " restricts the tuples of some selection to those with the domain value 'somedomain.com'.

C. Indexing

A database index is a supplementary data structure used to efficiently access data from the database. Data are indexed either directly by the values of one or more attributes or by hashes (generally not cryptographic hashes) of those values. The attributes used to define an index form the *key*. Indices are typically organized into tree structures, such as B^+ trees. The number of nodes between the root and any leaf of a B^+ tree is constant, because the tree is balanced. 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 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 an index attribute value 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) [25] techniques to hash a range of values to a single bucket, instead of to individual buckets. Recent advances [8] in *perfect hash functions (PHF)* have produced a family of hash functions that can efficiently maps a large set of n key values to a set of m integers without collisions, where n is less than or equal to m. A perfect hash function is *minimal* when $n = m$. These PHF that can work with large sets of keys (on the order of billions), unlike earlier developments, such as gperf [37], that can only manage small sets of keys.

Performance parameters of PHF are generation or construction speed to index a set of keys, representation size or bits stored per key and evaluation time. The state-of-the-art construction [5] takes linear time; the representation size can be as low as 0.67 bits per key for $m = 2n$. The evaluation time is $O(1)$. In addition to point queries, an order-preserving PHF [15] can be useful for evaluating range queries over a B^+ tree index.

III. 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. [12] 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 [12] neither explores B^+ trees nor considers executing SQL queries using keyword-based PIR.

Reardon et al. [35] 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.

Private matching and private set intersection schemes [19], [27] consider the problem of computing the intersection of two private sets from two users, such that each user only learns the sets' intersection. Our work is significantly different from private intersection schemes because SQL queries are richer and more complex than simple set intersection. In addition, an SQL query describes the expected result of a query, which may not contain any itemized listing of the data, whereas private set intersection schemes require the exact data to be the input of a query. The differences of these schemes from our work remain if one considers a modified private matching scheme, where only one party (the user) needs to learn of the result of the intersection.

Significant effort has been devoted to the problem of searching on encrypted data [3], [41], [47]. Shi et al. [38] considers the problem of storing encrypted data in an untrusted repository. To retrieve a subset of the encrypted data,

the user must possess a key that will only decrypt the data matching some preauthorized attributes or keywords. They considered the encryption and auditing of network flows, but the approach is also applicable to financial audit logs, medical privacy, and identity-based biometric encryption systems. Our work is different from encrypted data search in three ways. First, we do not require encryption of the data, regardless of the assumption of the adversary being an insider to the database server. The privacy provided with PIR aims to hide the particular data that is of interest, in the midst of the entire unencrypted data set. Second, the type of query supported with our approach is much more extensive. Third, encrypted data search is typically performed on unstructured or textual data, whereas our approach deals with structured data in the repository of relational databases.

A closely related research stream is the problem of privately searching an encrypted index over an outsourced database in the computing context of Database-as-a-Service [25], [22]. Hacigümüş et al. [22] presents a technique for executing SQL queries over a user-encrypted database hosted in a service provider's server. The goal is to protect the data from the service provider, but still enable the user to query the database. The context of use for the Database-as-a-service paradigm differs from that of PIR. The service provider typically owns the data that multiple users query with PIR. The goal is not to hide data from the server, but to hide data access patterns, which could leak information about users' requests.

A related problem to PIR is that of privately searching an unencrypted stream of documents [6], [33]. In these schemes, the client selects some keywords, and then encrypts them before sending it to a server. The server performs a search using the keywords over a stream of unencrypted documents and returns the list of documents containing the keywords back to the client. The server remains oblivious of which particular document it returns, and the confidentiality of the keywords is preserved. Existing constructions are limited to returning documents that give exact matches on a keyword list, or two keyword lists combined with logical "OR" or "AND". These types of queries are much simpler than a relational database query, which may contain multiple operators — comparison, logical, and so on. In addition, range queries are not presently possible with private stream searching because exact keywords must be specified for the search. The performance of private stream searching constructions is also comparable with that of a single database PIR, because most such schemes rely on homomorphic encryption using the Paillier cryptosystem [34].

An interesting attempt to build a practical pseudonymous message retrieval system using the technique of PIR is presented in [36]. 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.

IV. Threat Model, Security and Assumptions

A. 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 [4], [13]. 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 [4], [21]. 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.

B. 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.

C. 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 [10] and Tor [17]. 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], [45] offer considerable efficiency improvements over previous work in the area.

V. Hiding Sensitive Constants

A. 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 3, the domain name is sensitive because it could presumably be used for domain name front running, and the creation date may be sensitive as well.

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 client. The client subsequently performs PIR retrievals on the index and finally combines the retrieved items to build the result for the original query. An alternative approach to storing materialized tuples or subquery results in an index

Listing 3 Example SQL query with a WHERE clause featuring sensitive domain name information.

```
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')
```

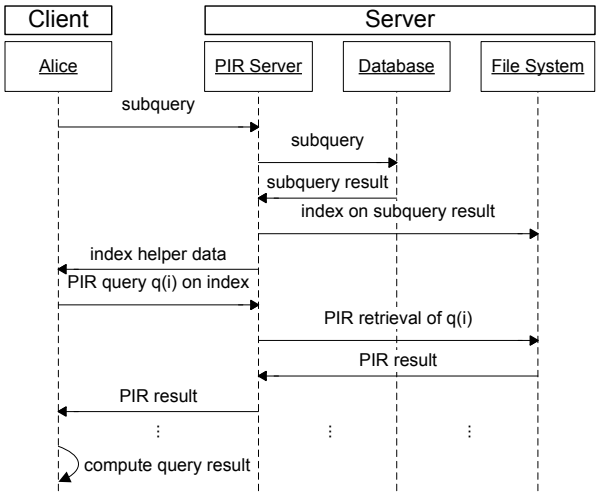

Figure 1. A sequence diagram for evaluating Alice's private query over a PIR-enabled relational database.

is to maintain index entries as references to actual database tuples. In other words, each index entry will simply store keys and reference database tuples. An index built using this approach can be considered as maintaining a 'view' of the subquery result (i.e., no data materialization). The approach offers space savings, but will incur considerable performance overhead. PIR queries over such indices necessarily require individual fetching of all tuples in the original subquery result (at worst), or systematic range-based fetches (at best). These operations will be slow and much more complex to implement. For these reasons, our approach explores indices built on materialized data.

The important benefits of this approach as compared with the previous approach [35] 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 V-B, 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 the users' privacy (see Section IV).

B. 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 3 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)
- 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 and the constraint list for the input query become:

- *Attribute list:* {`t1.contact`, `t1.email`, `t2.created`, `t2.expiry`, `t2.domain`}
- *Constraint list:* {`(t2.created, 20090101, >)`, `(t2.domain, 'anydomain.com', =)`}

The subquery, which is a SELECT query with reduced conditions, is shown in Listing 4.

Listing 4 Example subquery with reduced conditions.

```
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 to each server

- the subquery
- a key attribute name
- an index file type

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
- 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, unlike native database indices.

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 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 [36].

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.

[1]Using the CMPH Library [7] 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.

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. This is an open area of work closely related to database optimization.

VI. Discussion

In this section, we discuss important architectural components and design decisions related to the algorithm presented in Section V.

A. Parsing SQL queries

The algorithm parses an input query — the WHERE and HAVING clauses in particular. Other subclauses of the SELECT statements, such as GROUP BY and ORDER BY, can either be processed as part of a subquery or applied on the result obtained with PIR. Specific implementations can adopt the mature parsers developed with open source and commercial databases.

The expression tree provides an easy way to construct the desensitized query and the constraint list. The parsing process builds an expression tree representation for the WHERE clause conditions. The internal nodes of this expression tree typically contain arithmetic, relational, and logical operators, while the leaf nodes consist of attribute names and constants. Any WHERE clause predicate expression can be a join, a non-private condition or a private condition. The latter contains a sensitive constant value, whereas the former two do not. Our parser allows the user to tag sensitive constants with the symbol "#" to differentiate them from public constants. For example, the sensitive constant `'20090511'` is tagged in this query: `SELECT * FROM table WHERE n = 20090605 AND p = #20090511`. Each WHERE clause condition is related to another condition with the logical AND. Logical OR conditions are not considered as expression delimiters, but disjunct multiple subexpressions in the same condition. Typically, relational databases convert the WHERE clause conditions in the input query to an equivalent set of conditions in the conjunctive normal form, to facilitate query optimization.

For an example of AND and OR, consider the two SELECT queries below, which differ only in their WHERE clause conditions.

(i) `SELECT * FROM table WHERE a = 'SQL' AND b = 'LEX'`

(ii) `SELECT * FROM table WHERE a = 'SQL' OR b = 'LEX'`

The client can compute the result for (i) using either one or two indices, whereas it requires two indices to compute the result for (ii). To compute the result for (i) with a single index, the client requests an index for `a` or `b` because both of the conditions in the WHERE clause can only be true if one of them is true. If it requests an index for `a`, it will first perform keyword-based PIR using the literal `'SQL'` over this index, and then filter the result obtained with the second condition `b = 'LEX'`. To compute either (i) or (ii) with two indices, the client requests indices for both `a` and `b`, and then performs two keyword-based PIR searches using the string literals `'SQL'` and `'LEX'` over the respective indices. Finally, the client computes the intersection of the tuples in the two PIR results to obtain the result for (i), or it computes the union to obtain the result for (ii).

We note that a worst case query scenario having several private conditions combined with an OR operator will have storage and computational costs linear in the number of unique attribute names used with the private conditions. In certain circumstances, it may be possible to eliminate the storage cost by maintaining references to the tuples data in the database rather than maintaining a materialized copy in an index.

Currently, logical NOT conditions cannot be processed with PIR. We are unable to find any practical PIR scenario to justify its use. For example, performing PIR queries on a patent database will generally not require a NOT operator. We prescribe client-side processing for NOTs, after the data required for evaluating the condition are retrieved with PIR.

This expression tree is traversed twice. The first traversal lists the desensitized query's WHERE conditions, which includes all joins and all non-private conditions. The logical AND operator combines the joins and the non-private conditions. The boolean true value can serve as a placeholder for every private condition. For example, the actual WHERE clause for the subquery in Listing 4 can be `WHERE (t1.reg_id = t2.reg_id) AND true AND true`, which can be subsequently optimized. The second traversal lists the private conditions, which are used to build the constraint list.

B. Indexing subquery results

For many general purposes, it may be impractical to execute the desensitized query and generate an index on the query result for every request. The use of an index cache addresses some of the cost, because the database can use the same cached index to serve multiple PIR queries (with the same private attributes, though not necessarily the same private constants) from multiple users. This mitigates the computational costs for generating indices. An exception for

the use of a cache is when the shape of the input query is unpredictable, especially in an environment where the users make *ad hoc* queries. In this case, a separate index must be generated for each unique query.

C. Database servers

Practical implementations could use any commercial or open-source database server to execute the desensitized query. The client does not need to install database client programs to query the database server in the privacy-friendly manner we describe; however, the client will need an installation of the private SQL client that implements the client-side logic of the algorithm. Similarly, a program that implements the server-side logic of the algorithm must be installed at the server.

D. Processing specific conditions

We provide an overview on how to deal with private constants in specific conditions of the WHERE clause. In particular, we consider simple conditions, as well as specialized conditions such as BETWEEN, LIKE, and IN.

A simple WHERE clause condition consists of the general form $column$ ***relop*** $literal$ or $literal$ ***relop*** $column$, where *relop* is a relational operator, such as $=, <>, <, >, <=$, and $>=$. If the $column$ is used to index a query result, then the $literal$ will be used as input to the keyword-based PIR. The operator "$=$" indicates a point query. If the key attribute $column$ is unique, then a single result is expected; either a hash or a B^+ tree index is appropriate. On the other hand, a B^+ tree is preferred for non-unique key values, since there may be multiple tuples in the query result. The other operators, which imply range queries, require B^+ tree indices. The $literal$ or its next or previous neighbours from the domain of values for the data type, in sorted or lexicographical order, provide one of the values for the range search. The other value is determined from the smallest or largest value in the domain for the data type. The input values for range search for the condition `t2.created > 20090101`, for example, are `(20090102, 99991231)`.

A BETWEEN condition has the general form $column\ BETWEEN\ literal_1\ AND\ literal_2$, which is equivalent to the condition $column >= literal_1\ AND\ column <= literal_2$. This condition is processed as a range query on the two literal values.

A LIKE condition has the form $column\ LIKE\ literal$, where $literal$ is a search condition that involves one or more wildcards, such as % and __. The __ allows for the matching of a single character, while the % allows for matching strings of any length, including zero-length strings. Prefix-based conditions, such as `domain LIKE 'some%'`, and suffix-based ones, such as `domain LIKE '%main.com'` can easily be processed with a B^+ tree index over the attribute, or the reverse of the attribute, respectively. Other variants are more easily processed in the client; the client would first retrieve the data from the server with PIR, and then perform a more sophisticated filtering on the result, using the wildcard expression.

An IN condition has the general form $column\ IN$ $(literal_1$,$literal_2, ...)$. If the attribute $column$ has unique values, then the tuple associated with each literal can be retrieved with a point query on the same index over the $column$ attribute. Some PIR implementations, such as [20], can simultaneously retrieve multiple blocks for a set of literal values in a single query. Otherwise, a combination of range and point queries will be required. The client optimizer can be built to intelligently combine literal values to reduce the overall number of PIR queries.

Client-side support for database function evaluation is required when private constants are used as function parameters in a WHERE clause expression. Such functions can be evaluated before the data required are retrieved with PIR, or afterwards. The latter follows for functions that take private constants *and* attribute names as parameters.

We note that special WHERE clause conditions, such is $IS\ NULL$ and $IS\ NOT\ NULL$, do not require any private constants. It would suffice to include them in the desensitized query in many situations. Alternatively, they could be processed locally, especially for *ad hoc* queries, if they are considered to reveal sensitive information about the tuples of interest.

Finally, an implementation may decide to localize the processing of all the above conditions, as well as other conditions of the WHERE clause. The approach to adopt depends on the amount of optimization the client is capable of performing and the requirements of the application domain.

VII. Implementation and Microbenchmarks

A. 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 [20], [21], 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 [7], [8], 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 [16], [44]. Finally, we base the implementation

on the MySQL [42] relational database, version 5.1.37-1ubuntu5.1.

B. Experimental setup

We began evaluating our prototype implementation using a set of six whois-style queries from Reardon et al. [35], 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) [43] benchmarks, and open-source benchmarking kits such as Open Source Development Labs Database Test Suite (OSDL DTS) [46], 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 [35]. 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 evaluation queries and the two database relations in Appendices B and C. We choose the predicate parameters for the benchmark queries to ensure query selectivity values (ratio of the number of matching tuples to the total number of tuples) similar to those used in the original benchmarking of TransPIR [35]. The respective values for benchmark queries Q1 through Q6 for the small data set are 1.00×10^{-6}, 2.00×10^{-5}, 4.20×10^{-5}, 5.90×10^{-5}, 1.33×10^{-6}, and 3.87×10^{-2}. For the large data set they are 2.50×10^{-7}, 2.00×10^{-5}, 4.20×10^{-5}, 5.90×10^{-5}, 2.50×10^{-7}, and 4.20×10^{-5}.

The measurements for all test queries are based on the default behaviour of the TPIE Library with respect to determining the branching factor λ for B^+ tree indices. The following expression shows the computation of branching factor with this default configuration:

$$\lambda = \left\lfloor \frac{\gamma \times size(os\_block) - size(BID) - size(size\_t)}{size(Key) + size(BID)} \right\rfloor$$

Where γ, os_block, BID, $size\_t$, and Key are respectively the data logical blocking factor, operating system block, block ID, C++ $size\_t$ data type, and the key. $size(x)$ is the size of x in bytes. Specifically for our experimental setup for the large data set, the branching factor for indices over integer keys is 2730, and it is 409 for indices over character keys. For the small data set, these values are respectively 1634 and 215. Our implementation stores integer keys in 8 bytes, and character keys in 72 bytes. The branching factor values are based on a block size of 32 KB ($\gamma = 8$), and 16 KB ($\gamma = 4$), where $size(os\_block) = 4096$ bytes. The actual fill factor (again, the default for the TPIE Library) is 0.6 for internal B^+ tree nodes. We report the disk sizes for indices built for our experiment on the queries with complex conditions (see Table II).

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.

C. Result overview

The results from the benchmark tests 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 Table I later in this section and Table II in Section VIII). 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. [35] — 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 [35]. 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. We initially attempted to run the microbenchmarking tests for the larger data with TransPIR on the development hardware platform for our prototype, but was limited by this commodity hardware because TransPIR requires a 64-bit processor and a minimum of 6 GB RAM to index or preprocess the larger data set. The development hardware had a 32-bit processor and only 3 GB RAM.

D. Microbenchmark experiment

We executed the six whois-style benchmark queries over the data sets and 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 I 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,

Table I
EXPERIMENTAL RESULTS FOR BENCHMARK TESTS ON THE SMALL DATA SET COMPARED WITH THOSE OF REARDON ET AL. [35]. **BTREE** = RESULT FOR OUR B^+ TREE PROTOTYPE, **HASH** = RESULT FOR OUR HASH TABLE PROTOTYPE, AND **TransPIR** = RESULT FROM TRANSPIR [35]; **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 CACHED INDEX GENERATION, **Xfer** = TOTAL DATA TRANSFER BETWEEN THE CLIENT AND THE TWO PIR SERVERS.

Small database with .75 M contact records, 1 M registration records, and 16 KB blocks

Query	Approach	Time (s)	PIRs	Tuples	QI (s)	Xfer (KB)
Q1	**HASH**	0	1	1	4	64
	BTREE	6	4	1	9	256
	TransPIR	7	2	1	120	128
Q2	**BTREE**	3	3	20	7	192
	TransPIR	76	23	20	120	1,472
Q3	**BTREE**	3	3	42	7	192
	TransPIR	149	45	42	120	2,880
Q4	**BTREE**	13	3	59	8	256
	TransPIR	217	62	59	120	3,968
Q5	**BTREE**	5	4	1	13	256
	TransPIR	10	3	1	120	192
Q6‡	**BTREE**	5	3	29	13	192
	TransPIR	558	111	42	—‡	7,104

Large database with 3 M contact records, 4 M registration records, and 32 KB blocks

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

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.

E. 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 timing for query evaluation with PIR (Time), and sometimes on the index generation timing (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 on both data sets, followed by our B^+ tree (BTREE) prototype; it achieves better performance for the large set. 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. Again, our BTREE prototype outperforms TransPIR by a significant margin for both data sets; the improvement is most noticeable for the large data set. 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 key attributes only; it can only return a single tuple in its query result.

Query Q3 is a range query on the `expiry_date` attribute. Our BTREE prototype respectively was approximately 50 and 411 times faster than TransPIR for the small and large data sets. Of note is the large number of PIR queries that TransPIR needs to evaluate the query; for the large data set, our BTREE prototype requires only 2% of that number. We observed a similar trend for Q4, where BTREE was 17 and 480 times faster for the small and large sets respectively. 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 10 and 67 times faster than the time it takes TransPIR to execute the query alone.

Query Q5 is a point query with a single join. For the large data set, 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.

VIII. COMPLEX QUERY EVALUATION

In addition to the above microbenchmarks, we performed two other experiments to evaluate our prototype. The first of these studies 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 run the first experiment on the same hardware configuration as the microbenchmark tests, and the second experiment

‡We reproduced TransPIR's measurements from [35] for query Q6 because we could not get TransPIR to run Q6 due to program errors. The '—' under QI indicates measurements missing from [35]

Listing 5 Experimental SQL queries indicating private constants or conditions with "#".

CQ1 – Private point query on a range with sensitive domain name information.

```
SELECT domain, name, address,
email, reg_date, expiry_date
FROM registration, contact
WHERE (contact_id = registrant) AND
(reg_date > 20090501) AND
(domain = #'somedomain.org')
```

CQ2 – Private range query with sensitive registrar ID range.

```
SELECT domain, name,
address, email, expiry_date
FROM registration, contact
WHERE (contact_id = registrant) AND
(status IN (1,4,5,7,9)) AND
(registrar #BETWEEN 198542 AND 749999) AND
(expiry_date BETWEEN 20090101 AND 20091031)
```

CQ3 – Private aggregate point query with sensitive registrar ID value.

```
SELECT registrar, count(domain)
FROM registration, contact
WHERE (contact_id = registrar) AND
(registrar = #635393)
GROUP BY registrar
HAVING count(domain) > 0
ORDER BY registrar ASC
```

CQ4 – Non-private LIKE query revealing only the prefix of a domain name.

```
SELECT domain, name, address,
email, reg_date, expiry_date
FROM registration, contact
WHERE (contact_id = registrant) AND
(domain LIKE 'some%') AND
(domain = #'somedomain.com')
```

CQ5 – Private LIKE query with domain name prefix as wildcard.

```
SELECT domain, name, address,
email, reg_date, expiry_date
FROM registration, contact
WHERE (contact_id = registrant) AND
(domain #LIKE 'some%')
```

on the developmental hardware platform. For the test data, query selectivity for complex queries CQ1 through CQ5 are 3.70×10^{-6}, 5.05×10^{-2}, 2.11×10^{-6}, 1.30×10^{-3}, and 5.34×10^{-6} for the small data set. Similarly for the large data set, the values are 5.70×10^{-7}, 5.12×10^{-2}, 1.58×10^{-6}, 9.52×10^{-7}, and 1.50×10^{-6}.

In addition to the above microbenchmarks, we performed two other experiments to evaluate our prototype. The first of these studies 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. The second experiment tests whether our prototype leverages database optimization. Both experiments were performed on the same commodity hardware configuration as the microbenchmark tests.

A. Result overview

The results obtained from the experiments demonstrate the benefits of our approach for dealing with complex queries. While the storage and computational costs (over non-private querying) remain, the overall performance and resource requirements are still reasonable for the added privacy benefit. The prototype requires additional storage for two types of indices used for PIR operations. The first type of index is generated for a particular shape of query, over one or more key attributes or combinations of attributes. These types of indices need permanent storage in the same manner as native indices for relational databases. The second type of index is used in an *ad hoc* environment, where the tuples in a subquery result can be constrained in an unpredictable manner, with one or more WHERE clause conditions and joins. These latter indices must be generated as needed. In most practical situations, it should be possible for indices to be based on the former type, just like most software systems rely on indices prebuilt by the database to efficiently run their queries.

B. Experiments on queries with complex conditions

We describe and present the results of experiments that examined the behaviour of our prototype when supplied with SQL queries that are more complex than the above microbenchmarks. We provide a number of synthetic query scenarios having different requirements for privacy, the corresponding SQL queries with appropriate tagging for the condition involving sensitive data, and the measurements. As mentioned above, our SQL parser uses the "#" character to tag private conditions; we include that tag in the SQL queries we present in Listing 5. We used the same database schema (see Appendix C) as the microbenchmarks. The measurements show execution duration for the original query without privacy provision over the MySQL database, the same query after removal of conditions with sensitive information over the MySQL database, and several other measurements taken from within our prototype using a B^+ tree index. All of the measurements are reported in Table II.

Private point query (CQ1). The task is to obtain a domain name record from the whois server without revealing the sensitive domain name information.

Private range query (CQ2). The ICANN Security and Stability Advisory Committee may be interested in performing an investigation on some registrars, with IDs ranging from 198542 to 749999. The task is to privately obtain some domain name information without revealing the range of IDs for the registrars. We show the query to obtain registration records with status in the set (1, 4, 5, 7, 9), and expiration

Table II

MEASUREMENTS TAKEN FROM EXECUTING FIVE COMPLEX SQL QUERIES WITH VARYING REQUIREMENTS FOR PRIVACY. **oQm** = TIMING FOR EXECUTING ORIGINAL SQL QUERY DIRECTLY AGAINST A MYSQL DATABASE, **BTREE** = OVERALL TIMING FOR MEETING PRIVACY REQUIREMENTS WITH OUR B^+ TREE PROTOTYPE, **rQp** = SUBQUERY EXECUTION DURATION WITHIN BTREE, **cI** = TIMING FOR GENERATING CACHED INDEX WITHIN BTREE, **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** = TEMPORARY STORAGE SPACE FOR CACHED INDEX.

Small database with .75 M contact records, 1 M registration records, and 16 KB blocks

Query	oQm (s)	BTREE (s) =	rQp (s) +	cI (s) +	Time (s)	PIRs	Tuples	rTuples	Xfer (KB)	Size (MB)
CQ1	**0**	**8**	4	2	**2**	3	1	328,805	192	110.57
CQ2	**0**	**2**	0	0	**2**	17	686	13,594	1,088	4.57
CQ3	**0**	**13**	9	2	**2**	3	1	473,646	192	157.82
CQ4	**0**	**0**	0	0	**0**	2	1	768	128	0.32
CQ5	**0**	**17**	9	5	**3**	4	4	749,472	256	251.82

Large database with 3 M contact records, 4 M registration records, and 32 KB blocks

Query	oQm (s)	BTREE (s) =	rQp (s) +	cI (s) +	Time (s)	PIRs	Tuples	rTuples	Xfer (KB)	Size (MB)
CQ1	**2**	**31**	19	10	**2**	3	1	1,753,144	384	579.63
CQ2	**1**	**15**	2	0	**13**	41	3,716	72,568	5,248	25.13
CQ3	**0**	**80**	74	3	**3**	3	1	631,806	384	209.38
CQ4	**2**	**25**	12	7	**5**	3	1	1,050,300	384	348.63
CQ5	**2**	**69**	42	24	**3**	3	6	4,000,000	256	1,324.13

dates between 20090101 and 20091031, without revealing the registrar ID range.

Private aggregate point query (CQ3). The task is to privately compute the total number of registrations sponsored by a particular registrar. The registrar ID is sensitive.

Non-private LIKE query (CQ4). The task is to efficiently retrieve a single domain name record from a whois server with some amount of privacy. In other words, a user wants to reveal a prefix of the domain name to improve performance, while still preventing the adversary from learning the exact textual domain name. Since many long domain names have a common prefix, the user intends to leverage that knowledge to improve query performance.

Private LIKE query (CQ5). The task is to retrieve registration records from a whois server without revealing the LIKE wildcard.

Results. We see from Table II 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 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 on the large data set. Moreover, this trivial download of data is impractical for devices with low bandwidth and storage (e.g., mobile devices).

C. Database optimization experiments

We studied the overall response of our prototype to determine the benefits accrued from database optimization. The experimental MySQL database runs mostly with the default settings. The only change made was reducing the default number of user connections to free up memory for other processes running on the machine. In other words, we did not tune the database for optimal performance in the course of our previous experiments.

Most databases cache query plans and small-sized query results when a query is executed for the very first time. Subsequent executions of the same query will be more responsive by reusing the cached plan and result. For this experiment, we disabled cache usage by flushing the relations in our database before running each query. Flushing relations in MySQL closes the open relations and flushes the query cache. This ensures the database obtains a fresh query plan and result set every time.

We ran this experiment on the less powerful hardware platform we used to develop the prototypes, because the time to build a fresh query plan does not take a significant time for the more powerful hardware platform we used for running the previous tests.

Table III presents the measurements taken for CQ1 through CQ5 over the large data set under default database behaviour, and the measurements taken when the relations of the database are flushed. The result obtained for these queries validates the claim that our approach leverages database optimization to improve performance. The most interesting measurements taken in this experiment are the subquery execution durations (rQp). For CQ1, CQ3, and CQ5, the difference in measurements is obvious. However, the effect is not quite obvious for CQ2 and CQ4. For the latter pair, the fraction of the overall timing spent for PIR

Table III
EFFECTS OF DATABASE OPTIMIZATION ON QUERY RESPONSIVENESS, OVER THE LARGE DATA SET. **BTREE** = OVERALL TIMING FOR MEETING PRIVACY REQUIREMENTS WITH OUR B^+ TREE PROTOTYPE, **rQp** = SUBQUERY EXECUTION DURATION WITHIN BTREE, **cI** = TIMING FOR GENERATING CACHED INDEX WITHIN BTREE, **Time** = TIME TO EVALUATE PRIVATE QUERY WITHIN BTREE.

	With optimization: default database settings				**Without optimization: query cache disabled**			
Query	**BTREE (s) =**	**rQp (s) +**	**cI (s) +**	**PIR (s)**	**BTREE (s) =**	**rQp (s) +**	**cI (s) +**	**PIR (s)**
CQ1	104	**50**	52	2	122	**69**	50	3
CQ2	22	**3**	1	18	29	**4**	2	23
CQ3	375	**347**	22	5	454	**434**	15	6
CQ4	66	**25**	32	9	66	**26**	29	11
CQ5	214	**90**	118	6	436	**310**	120	6

queries is nonnegligible: 79% and 17% respectively. For CQ1, CQ3, and CQ5, the portion of the time spent for PIR is respectively 2%, 1%, and 1%. The results for CQ1, CQ3, and CQ5 clearly indicate the contributions of database optimization to query responsiveness with our approach.

D. Improving performance by revealing keyword prefixes.

The performance of a query may be improved by revealing a prefix or suffix of the sensitive keyword in the query. Revealing a substring of a keyword helps to constrain the result set that will be indexed and retrieved with PIR. We have demonstrated the feasibility of this technique with complex query CQ4 (Listing 5 and Table II). While this technique may be infeasible in some application domains, due to the sensitive nature of the keyword, it does improve performance in others. This technique does, of course, trade off improved performance for some loss of privacy, though it is in fact the user (who can best make this trade-off decision) who can decide to what extent to use it. Making the best trade-off decision 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 actually considering this extension as part of our future work.

The processing of queries that allow users to reveal either a prefix or suffix of their private constant will proceed as follows on a prebuilt index. A user would first request the root to a particular subtree in a prebuilt B^+ tree index (indexed either on the attribute or the reverse of the attribute, as above), by supplying a substring for that root. The server would search for and return the requested root, without PIR. Subsequent PIR queries by the user will be based on the subtree with the retrieved root, instead of the entire B^+ tree. In other words, revealing a substring of a user's private keyword reveals the portion of the data that is of interest to the user. However, the level of privacy protection may still be sufficient for many user and application purposes.

The only realistic situations where performance cannot be easily improved with this technique are when users must make *ad hoc* queries that are unknown to the server before a system is deployed. In such situations, it is difficult to make a single index generic enough to serve the diversity of the constraints in an *ad hoc* query.

E. 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'), is beyond the current capability of keyword-based PIR. Similarly, negated WHERE clause conditions, such as (NOT registrant = 45444), are infeasible to compute with 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.

IX. 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 utilize 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. Private information retrieval is only the first step for preserving a user's query privacy. An extension to this work can explore private information storage (PIS) [32], and how to use it for augmenting the privacy of users in real-world scenarios. An interesting focus would be to extend PIS to SQL in the manner of this paper, in order to preserve the privacy of sensitive data within SQL INSERT, UPDATE and DELETE data manipulation statements.

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] J. Baek, R. Safavi-Naini, and W. Susilo. On the Integration of Public Key Data Encryption and Public Key Encryption with Keyword Search. In S. K. Katsikas, J. Lopez, M. Backes, S. Gritzalis, and B. Preneel, editors, *ISC*, volume 4176 of *Lecture Notes in Computer Science*, pages 217–232, 2006.

[4] A. Beimel and Y. Stahl. Robust Information-Theoretic Private Information Retrieval. *J. Cryptol.*, 20(3):295–321, 2007.

[5] D. Belazzougui, F. C. Botelho, and M. Dietzfelbinger. Hash, Displace, and Compress. In *ESA 2009: Proceedings of the 17th Annual European Symposium, September 7-9, 2009*, pages 682–693, 2009.

[6] J. Bethencourt, D. Song, and B. Waters. New Techniques for Private Stream Searching. *ACM Trans. Inf. Syst. Secur.*, 12(3):1–32, 2009.

[7] F. C. Botelho, D. Reis, and N. Ziviani. CMPH: C minimal perfect hashing library on SourceForge. http://cmph.sourceforge.net/.

[8] F. C. Botelho and N. Ziviani. External perfect hashing for very large key sets. In *ACM CIKM*, pages 653–662, 2007.

[9] G. Buehrer, B. W. Weide, and P. A. G. Sivilotti. Using parse tree validation to prevent SQL injection attacks. In *SEM*, pages 106–113, 2005.

[10] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–90, 1981.

[11] B. Chor and N. Gilboa. Computationally private information retrieval (extended abstract). In *STOC '97: Proceedings of the twenty-ninth annual ACM Symposium on Theory of Computing*, pages 304–313, New York, NY, USA, 1997.

[12] B. Chor, N. Gilboa, and M. Naor. Private information retrieval by keywords. Technical Report TR CS0917, Dept. of Computer Science, Technion, Israel, 1997.

[13] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *FOCS*, pages 41–50, Oct 1995.

[14] G. D. Crescenzo. Towards Practical Private Information Retrieval. Achieving Practical Private Information Retrieval (Panel @ Securecomm 2006), Aug. 2006.

[15] Z. J. Czech, G. Havas, and B. S. Majewski. An optimal algorithm for generating minimal perfect hash functions. *Inf. Process. Lett.*, 43(5):257–264, 1992.

[16] Department of Computer Science at Duke University. The TPIE (Templated Portable I/O Environment). http://madalgo.au.dk/Trac-tpie/.

[17] R. Dingledine, N. Mathewson, and P. Syverson. Tor: the second-generation onion router. In *USENIX Security Symposium*, pages 21–21, 2004.

[18] M. J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold. Keyword search and oblivious pseudorandom functions. In J. Kilian, editor, *TCC*, volume 3378 of *Lecture Notes in Computer Science*, pages 303–324. Springer, 2005.

[19] M. J. Freedman, K. Nissim, and B. Pinkas. Efficient Private Matching and Set Intersection. In C. Cachin and J. Camenisch, editors, *EUROCRYPT*, volume 3027 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2004.

[20] I. Goldberg. Percy++ project on SourceForge. http://percy.sourceforge.net/.

[21] I. Goldberg. Improving the Robustness of Private Information Retrieval. In *IEEE Symposium on Security and Privacy*, pages 131–148, 2007.

[22] 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.

[23] R. J. Hansen and M. L. Patterson. Guns and butter: Towards formal axioms of input validation. In *Black Hat USA 2005*, Las Vegas, July 2005.

[24] R. J. Hansen and M. L. Patterson. Stopping injection attacks with computational theory. In *Black Hat USA 2005*, Las Vegas, July 2005.

[25] B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. In *VLDB*, pages 720–731, 2004.

[26] ICANN Security and Stability Advisory Committee (SSAC). Report on Domain Name Front Running, February 2008.

[27] S. Jarecki and X. Liu. Efficient Oblivious Pseudorandom Function with Applications to Adaptive OT and Secure Computation of Set Intersection. In *TCC '09: Proceedings of the 6th Theory of Cryptography Conference on Theory of Cryptography*, pages 577–594, Berlin, Heidelberg, 2009.

[28] E. Kushilevitz and R. Ostrovsky. Replication is not needed: single database, computationally-private information retrieval. In *FOCS*, page 364, 1997.

[29] S. K. Mishra and P. Sarkar. Symmetrically Private Information Retrieval. In *INDOCRYPT*, pages 225–236, 2000.

[30] M. Naor and B. Pinkas. Oblivious transfer and polynomial evaluation. In *ACM Symposium on Theory of Computing*, pages 245–254, 1999.

[31] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *ACM-SIAM SODA*, pages 448–457, 2001.

[32] R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *STOC '97: Proceedings of the twenty-ninth annual ACM Symposium on Theory of Computing*, pages 294–303, New York, NY, USA, 1997.

[33] R. Ostrovsky and W. E. Skeith, III. Private Searching on Streaming Data. *J. Cryptol.*, 20(4):397–430, 2007.

[34] P. Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Advances in Cryptology—Eurocrypt '99, Lecture Notes in Computer Science 1592*, pages 223–238, 1999.

[35] J. Reardon, J. Pound, and I. Goldberg. Relational-Complete Private Information Retrieval. Technical report, CACR 2007-34, University of Waterloo, 2007.

[36] L. Sassaman, B. Cohen, and N. Mathewson. The Pynchon Gate: a Secure Method of Pseudonymous Mail Retrieval. In *ACM WPES*, pages 1–9, 2005.

[37] D. C. Schmidt. *More C++ gems*, chapter GPERF: a perfect hash function generator, pages 461–491. Cambridge University Press, New York, NY, USA, 2000.

[38] 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.

[39] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, Inc., New York, NY, USA, 5th edition, 2005.

[40] R. Sion and B. Carbunar. On the Computational Practicality of Private Information Retrieval. In *Network and Distributed Systems Security Symposium*, 2007.

[41] D. X. Song, D. Wagner, and A. Perrig. Practical Techniques for Searches on Encrypted Data. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 44, Washington, DC, USA, 2000.

[42] Sun Microsystems. MySQL. http://www.mysql.com/.

[43] Transaction Processing Performance Council. Benchmark C. http://www.tpc.org/.

[44] 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.

[45] P. Williams and R. Sion. Usable PIR. In *Network and Distributed System Security Symposium*. The Internet Society, 2008.

[46] M. Wong and C. Thomas. Database Test Suite project on SourceForge. http://osdldbt.sourceforge.net/.

[47] S. S. Yau and Y. Yin. Controlled privacy preserving keyword search. In *ASIACCS '08: Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security*, pages 321–324, New York, NY, USA, 2008.

APPENDIX

A. Database schema for examples

```
CREATE TABLE registrar (
reg_id int(11) NOT NULL,
contact char(60) default NULL,
phone char(80) default NULL,
address char(80) default NULL,
email char(60) default NULL,
PRIMARY KEY (reg_id));

CREATE TABLE regdomains (
id int(11) NOT NULL,
domain char(80) default NULL,
created int(8) default NULL,
expiry int(8) default NULL,
reg_id int(11) NOT NULL,
status varchar(2) default NULL,
PRIMARY KEY (reg_id));
```

B. Microbenchmark queries from [35]

Q1 – Point query with single result

```
SELECT domain, reg_date
FROM registration WHERE domain = ?
```

Q2 – Point query with multiple results

```
SELECT domain FROM registration
WHERE expiry_date = ?
```

Q3 – Range query with single condition

```
SELECT domain, status FROM
registration WHERE expiry_date > ?
```

Q4 – Range query with multiple conditions

```
SELECT * FROM registration
WHERE expiry_date > ? AND reg_date < ?
```

Q5 – Point query with join

```
SELECT domain, name, email
FROM contact, registration
WHERE domain=? AND registrant = contact_id
```

Q6 – Range query with join

```
SELECT * FROM contact,registration WHERE
expiry_date>? AND registrar=contact_id
```

C. Database schema for microbenchmarks and experiments

```
CREATE TABLE contact (
contact_id int(11) NOT NULL,
name char(60) default NULL,
address char(80) default NULL,
email char(60) default NULL,
PRIMARY KEY (contact_id));

CREATE TABLE registration (
reg_id int(11) NOT NULL,
domain char(80) default NULL,
expiry_date int(8) default NULL,
reg_date int(8) default NULL,
registrant int(11) default NULL,
registrar int(11) default NULL,
status varchar(2) default NULL,
PRIMARY KEY (reg_id));

ADD FOREIGN KEY fk_registrant (registrant)
REFERENCES contact(contact_id);
ADD FOREIGN KEY fk_registrar (registrar)
REFERENCES contact(contact_id);
```