Relational-Complete Private Information Retrieval

Joel Reardon Jeffrey Pound Ian Goldberg

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

Abstract

In the typical client/server model for data retrieval, the content of the client's query is exposed to allow the server to evaluate the query and return the result. However, in some situations, it may be desirable to keep the content of the query confidential. To achieve this, systems can make use of private information retrieval (PIR) technology, which allows the retrieval of data without compromising the information contained in the query. Currently, the use of PIR comes with a great restriction on the data access language, severely limiting the practicality of such systems. In this paper we show how to extend PIR systems so that SQL-like queries can be privately evaluated over relational databases. We present TRANSPIR, an implementation of our extensions, and analyze its performance. Our work brings relational-complete query capabilities to traditional private information retrieval schemes, which gives greater utility to a private information retriever without impacting the confidentiality of her query.

Keywords: Privacy, Private Information Retrieval, Relational Algebra

1 Introduction

The problem of Private Information Retrieval (PIR) [6] is to enable a client to request information from a server such that the server learns no information about what the client has requested. Such functionality could have many practical uses where the information in a query is confidential, such as querying a patent database without eavesdroppers discovering the (private) nature of your work, or privately searching messages from a public host, without anyone associating your identity to your message box by the content of the query.

We consider a motivating example that highly relevant to today's e-commerce sector: the searching of a *whois* database. A *whois* database contains registration information for world-wide web (WWW) domain names. If a person is interested in registering a domain name, she would first query a *whois* server to see if the domain is currently registered; if not, she may register it.

The problem arises when the server administrator (or a third-party observer of the network traffic) views the content of the *whois* query. The fact the query is being issued establishes that someone is interested in registering the domain, which increases its potential value. The eavesdropper can then register the domain before the original query issuer has the opportunity, and subsequently charge the original query issuer an inflated price to purchase the domain. This practice is known as *domain front running* [14].

A PIR-based query interface to *whois* data would remedy this problem, allowing people to query the availability of domain names without the servers ever knowing the content of the query. Most PIR schemes, however, are restricted to retrieving blocks at known offsets in the database. This restrictive data access model makes practical use of PIR technology very limited. Thus, support for more expressive query languages over PIR databases is needed before such systems can be realistically deployed.

Relational algebra provides the underlying theory for today's most prevalent query languages for data access, such as SQL [13] and XQuery [22]. As such, it is an obvious starting point for building an expressive data access model for private information retrieval. In this paper we propose TRANSPIR (Transparent Relational Algebra eNabled Storage for Private Information Retrieval), a system for privately evaluating relational queries with the goal of minimizing communication complexity. Relational algebra allows for a more expressive query language which increases the utility of PIR. We discuss how to translate relational algebra queries into *query plans* comprised of function calls to a virtual database interface. The virtual database in turn responds to these calls by fetching the appropriate PIR blocks from the PIR server based on the layout of the database. To reduce the communication complexity, we construct indices over the relations in the database that aid in executing queries while attempting to minimize the requests for data required.

Our experimental results indicate such a system is feasible for performing queries with small output sizes, such as point searches, small ranges searches, or a small set of tuples satisfying multiple constraints on indexed attributes. The processing time for a sequence of PIR queries is equal to the processing time for a single PIR request times the number of blocks being retrieved, and so for cases where entire relations need to be retrieved, or to retrieve a large number of arbitrary tuples from distinct blocks, the processing time becomes a significant burden to operation.

The main contributions of this work are as follows:

- We improve the state of the art of private information retrieval by enabling SQL-like queries to be privately evaluated over relational databases.
- We evaluate our experimental implementation of the system, illustrating the feasibility of relational query processing over PIR.

The remainder of the paper is organized as follows. Section 2 reviews PIR and the relational model. Section 3 presents our system model and the encoding of relational databases as a PIR array, as well as discussing the query optimizer and the problems introduced when querying private databases. Section 4 contains details about our implementation and experimental results. Section 5 offers some future work on this topic, and Section 6 concludes.

2 Preliminaries

2.1 Private Information Retrieval

Private information retrieval is the problem of allowing clients to obtain information from a server without revealing to any observer, including the server, what information has been requested. This field was initiated with the study of oblivious transfer [18], which aimed to allow servers to send information while remaining oblivious as to what was sent. Initial lower bounds by Chor et al. proved that if PIR was being provided by a single server then information-theoretic privacy, meaning privacy against computationally unbounded adversaries, is only possible if the client requests the entire database and performs the query off-line [6]. To circumvent this bandwidth-intensive solution, the authors presented a PIR system that information-theoretically separated the query between two servers. The client is given a protocol to convert their desired bit into structured server queries and combine the results to retrieve their desired bit privately [6]. Their security model assumed that the servers will not collude, but will record any information they happened to come across, known as the honest-but-curious security model in the literature.

Concisely, the client would request the exclusive-or of a set of bits from both servers. These sets were redundant in all but the bit of interest, and so the client would take the exclusive-or of the results to answer the query. If the size of the requests were proportional to half the database size, and neither server knew if their request included the bit of interest, then neither server could learn statistical information about what bit the client has requested. Since queries in this model have complexity $\Theta(n)$ for a database of size n, their paper then provided ways to reduce the size of queries to $O\left(n^{\frac{1}{3}}\right)$ without impacting the unconditional privacy of the scheme [6]. Chor and Gilboa suggested computational PIR, where a client's query is protected by problems used in cryptography that are believed to be computationally hard for an adversary to compute [4]. This is weaker security in contrast with information-theoretic security, and known techniques are much more computationally expensive than information-theoretic PIR, but it can yield protocols with smaller communication complexity, and fewer servers.

The honest-but-curious security model has been considered insufficiently secure by some papers [1, 11, 12], and consequently the robustness of PIR has been addressed against stronger adversaries. Beimel and Stahl advance the notion of servers that fail to respond or intentionally or accidentally respond incorrectly [1], and propose schemes that are robust against such adversaries. Gertner et al. consider that keeping replicated copies of the database to be a security vulnerability in itself, and considered schemes where the database operator information-theoretically could not determine the contents of the database it was storing [11]. A recent result by Goldberg [12] improved the robustness of PIR against both these adversaries and introduced the notion of hybrid PIR, where the client's privacy is protected by information-theoretic security against a subset of colluding servers, and further protected by computational security against all servers colluding.

2.1.1 Useable PIR

The usefulness of privately retrieving a single bit is dubious, and many PIR systems are designed to allow a client to retrieve sequences of bits, known as blocks [6]. The servers in such systems are modelled as a database partitioned into a sequence of equal-sized blocks. Communication efficiency is improved when requesting an entire block as more traffic contains useful data, and the server can also greatly increase throughput (with a cost in latency) if it receives multiple queries and computes the responses concurrently. In subsequent work, Chor, Gilboa, and Naor acknowledged that assuming that clients know a priori which PIR block they wish to retrieve may be unreasonable [5]. It is particularly unlikely when the client is not personally managing the database. They introduce the notion of retrieving data based on keywords, and name their system PERKY, for PrivatE Retrieval by KeYwords. Hence, a client need only know keywords for the data they wish to retrieve, and the server is capable of replying usefully without learning what keywords are being sought. They give a general reduction for performing a query by having PIR blocks contain the required results for the possible paths of search keywords. This technique is later substantiated by giving the example of indexing data using elementary data structures, such as binary search trees and hash tables. This is the technique upon which we will build in our development of a complete relational algebra PIR system. Our work differs from that of Chor et al. in three ways. First, we consider the problem of encoding our index structures for arbitrarily sized PIR blocks to minimize the number of blocks needed to traverse the index. This problem was beyond the scope of the foundational work of Chor. Second, the keyword based hash index of Chor depends on finding a perfect hashing scheme; we relax this requirement by allowing bucket chaining in our hash index. Lastly, we implement and empirically evaluate our PIR based data structures. Saint-Jean, in a technical report for a PIR implementation [20], considered as future work the possibility of supporting string types, non-sequential keys, existential queries, and joins; our proposal supports all of these features and more.

To simulate the notion of a mail server, Kissner et al. [15] introduced push and pull by keywords. This work provided for stacks indexed by keywords, where three main operations could be performed privately: pushing an item onto a keyword's stack, pulling an item off of a keyword's stack, and peeking at the top item of the stack. This system naturally lends itself to a private mail server. Mail is encrypted for a user with their public key, and pushed into a stack based on some non-identifying pseudonymous keyword. The recipient will occasionally peek and pull any mail that has been pushed into their mailbox.

Recently, the Pynchon Gate [21] was designed, arguably the first attempt to seriously develop and deploy a useable pseudonymous mail retrieval system. The authors suggest that existing mail anonymizing systems had bandwidth problems, were vulnerable to traffic analysis, or were too burdensome, thus encouraging users to take shortcuts that degrade their privacy. Their system used multiple privacy-enhancing components, including the PIR system of Chor [6] and anonymous remailers [9] for communication in both directions. Since mail retrieval may take a variable number of PIR operations, each user performed extra *dummy* PIR requests, up to a system-defined maximum. This prevented mail servers from learning how much mail was being retrieved.

2.2 Relational Algebra

The relational model provides a logical abstraction for the underlying data representation in a data management application [7, 8]. Relational algebra provides the formal operators needed to access the underlying data through its abstraction.

In the following subsections, we review the operators that make up the relational algebra query language, and informally explain their semantics. We start with a brief and informal description of the definitions and notation we use to discuss the relational model.

2.2.1 Definitions

A *relation* is a named set of *tuples* (*records*) that constitute the data being represented. Each tuple has a fixed structure of *attribute* values, as dictated by the relation's *schema*. A schema consists of a list of attribute names and types, as well as *primary keys* and *foreign key constraints*. A primary key is a set of attributes that uniquely identify a tuple. A foreign key constraint restricts values of attributes in a relation to reference values of attributes in another relation. A *predicate* is a binary expression of the form $attribute \diamond constant$ or $attribute \diamond attribute$, where $\diamond$ denotes a binary operator over the domain of the attributes, generally one of $\{=, >, \geq, <, \leq, \neq\}$.

2.2.2 The Primitive Relational Operators

Relational algebra is based on six primitive operators, from which all other operators are derived. These operators are rename, selection, projection, union, difference, and cross product. Each operator takes one or two relations as arguments, and generates a relation as output. Some operators may also take an additional argument such as a predicate over the attributes of the relation.

- ***Rename*** (ρ) The rename operator is used to rename an attribute in a relation. This operator is used to avoid naming conflicts in cross products and joins. We omit a discussion of this operator as it can be trivially supported in our model.
- ***Selection*** (σ) The selection operator is used to select particular records from a relation R that satisfy a given predicate p and is denoted as:

  $$\sigma_p(R)$$

  We note that predicates may be arbitrary conjunctions or disjunctions of atomic predicates; the following relational algebra equivalences allow us to consider p to be atomic for convenience in later sections. This first equivalence shows that conjuncts of predicates in a selection are equivalent to a nesting of selections with atomic predicates.

  $$\sigma_{(a_1 \diamond k_1) \wedge (a_2 \diamond k_2) \wedge \cdots \wedge (a_n \diamond k_n)}(R) \equiv$$

  $$\sigma_{(a_1 \diamond k_1)}(\sigma_{(a_2 \diamond k_2)}(\cdots(\sigma_{(a_n \diamond k_n)}(R))\cdots))$$

A disjunction of predicates can be expressed as a union of the selection of each atomic predicate.

$$\sigma_{(a_1 \diamond k_1) \vee (a_2 \diamond k_2) \vee \cdots \vee (a_n \diamond k_n)}(R) \equiv$$
$$\sigma_{(a_1 \diamond k_1)}(R) \ \cup \ \sigma_{(a_2 \diamond k_2)}(R) \ \cup \cdots \cup \ \sigma_{(a_n \diamond k_n)}(R)$$

- ***Projection*** (π) The projection operator is used to extract desired attributes $a_1, a_2, \cdots, a_n$ from a relation R using the following syntax:

  $$\pi_{a_1, a_2, \cdots, a_n}(R)$$

  The result is a relation containing one tuple for each tuple in R, with only the attributes specified in the argument to the projection operator.
- ***Union*** $(\cup)$ The union operator is analogous to the set union operator. The result of $R \cup S$ is a single relation containing all tuples from both R and S. Note that the union operator is only defined for relations that are *union compatible*, that is, they have the same number of attributes, and the domain of the attributes in R (from left to right) is the same as those of S.
- ***Difference*** $(-)$ The difference operator is analogous to the set difference operator. The result of $R - S$ is a relation containing all tuples from R that do not appear in S. The difference operator is only defined for relations that are union compatible.
- ***Cross Product*** $(\times)$ The cross product (or Cartesian product) operator is a binary operator over relations. The result of $R \times S$ produces a relation containing a concatenation of each record in R with every record in S. Each tuple in the resulting relation has all attributes of both R and S.

The two most significant operators that can be derived from the primitive relational algebra operators are intersection and join. While these operators can be evaluated by the definition of their construction, in many situations alternate evaluations can be more efficient. We describe these derived operators below.

- ***Intersection***: Intersection can be expressed using the primitive operators in the following way:

  $$A \cap B \equiv A - (A - B)$$

- ***Join***: The join operator can be implemented by the definition of its primitive components as

  $$A \bowtie_p B \equiv \sigma_p(A \times B)$$

  or in the natural join case as

  $$A \bowtie B \equiv \sigma_{(A.x_1 = B.x_1 \wedge A.x_2 = B.x_2 \wedge \ldots \wedge A.x_n = B.x_n)}(A \times B)$$

  where the predicate is equality of all common attributes of A and B.

As an example of how relational algebra is used, consider the following SQL query and a relational algebra translation. The query finds all active domain names and registration dates in the *whois* database, such that the domain will expire before 2008-02-01.

SQL	**Relational Algebra**

`SELECT domain, date`
`FROM Registrations`
`WHERE expiry < 20080201`
`AND status = “ACTIVE”`
	$\pi_{\{domain,\ date\}}($
	$\sigma_{status=\text{“}\texttt{ACTIVE}\text{”}}($
	$\sigma_{expiry<20080201}($
	$Registrations)))$

3 System Design

3.1 Model

This paper builds relational-complete PIR on top of private block retrieval. As such, we allow for any implementation of a PIR protocol—information-theoretic or computational, single or multiple server, etc.—provided it can compute the single function *getBlock(i)* without the server learning any information about i. This function is provided a block number i and returns the data contained in that block. The function is implemented in a PIR proxy that performs whatever work is necessary to fulfill this request.

In the following subsections, we discuss a relational data encoding for PIR, a virtual database that maps relation information into PIR blocks, and a query processor that translates relational algebra requests into function calls implemented by the virtual database. This can be viewed as an implementation of a relational database, with the limitation of being able to retrieve a single block at a time during query evaluation and without the aid of the server in selecting the block based on its content.

We note that in implementation, it is worthwhile to parallelize the query plan and dispatch multiple concurrent block requests. This is a consequence of the nature of most PIR systems, including the one used in our implementation, to perform computation on every block in the database during each block request. By examining every block, the protocol ensures the privacy of requests against the responding servers. It is thus possible for a server to achieve greater throughput if multiple requests are processed concurrently during a single pass over the database.

We aim to minimize the number of blocks that must be retrieved in satisfying a query. This is in contrast to traditional relational databases which aim to minimize wall-clock time, potentially resulting in more data retrieval in order to optimize disk access. We discuss this issue further in Section 3.3. An overview of our system is illustrated in Figure 1.

3.2 Encoding Relational Databases for PIR

In this section we describe an encoding of a relational database, including metadata, relations, and indices, as a large array of *words* partitioned into fixed-size blocks; a typical block size for a 1 gigabyte database would be around 32 kilobytes. This is significantly larger than the block sizes

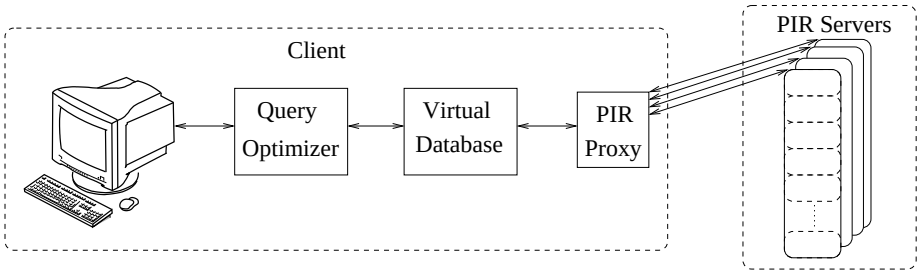

Figure 1: TRANSPIR system diagram. The user, query processor, virtual database and PIR proxy are all located on the client side. The only exposed data are the communications with the PIR servers, which are protected by the underlying PIR block retrieval protocol.

used in traditional database applications, however to use PIR blocks effectively it is necessary to use a block size on the order of the square root of the database size. Each word is 8 bits and attribute values are marshalled as compounds of words as in typical byte-aligned architectures; we use the term *array position* to refer to a single word index in the PIR array. Both the client and server messages in a private block query are typically around the size of one block, and so both can be quickly transmitted over high-speed Internet connections.

The problem of encoding this database is analogous to encoding resources onto disk pages in traditional relational databases, with a few distinguishing characteristics. First, our optimization goal for query answering is to minimize network traffic, so clients incur no penalty for random access to PIR blocks. Their relative physical storage location is not an issue considered during query evaluation; this differs from the traditional database setting in which data layout can greatly affect the performance of query evaluation. For example, clustered indices optimize disk access by making the index scan order consistent with the page order on disk; i.e., the number of read-head seeks is minimized when scanning the index. Second, depending on the underlying PIR block protocol, the PIR block may be considerably larger than a typical disk page. This presents interesting issues in deciding how to encode resources, particularly tree indices, which need to be traversed and scanned with a minimal number of PIR block accesses. In this context clustered indices can be beneficial to our system by clustering sequential records of an attribute value on the same block, meaning discrete ranges of records are stored on a minimal (or near minimal) number of blocks. Finally, observe that the PIR server is unable to aid in the evaluation of queries as it is oblivious to the data values being accessed. We must encode our resources such that retrieving the necessary data in order to process a query accesses the fewest number of PIR blocks possible.

Indices are used to index records over the values of a single attribute. This permits rapid evaluation of predicates and determination of the cardinality of potential result sets. There are two kinds of predicates that are considered: equality predicates (point searches) and inequality predicates (range searches). An equality predicate specifies that the value of the attribute must be equal to some provided value, such as all domains assigned to a certain owner. A range predicate permits the attribute to occur within a contiguous range of allowed values, such as searching for

domains registered between two dates. To evaluate a predicate on an index means to return a set of *attribute-index pairs* that correspond to all records that satisfy the predicate. To determine the cardinality means to return the size of the set of matching tuples.

We implement two types of indices: binary search trees (BST) and hash tables. BSTs are used to index attributes with total linearly ordered domains, and are able to determine cardinality and evaluate either equality or range predicates. The hash indices are used to index attributes in any domain, and are able to determine the cardinality or evaluate equality predicates. Hash table indices also permit a set of keywords to be associated with a record, building an index that maps a keyword to a set of related records. We do not consider the standard B+ tree structure in our model since our system does not perform updates to the database. We discuss updates in the future work section.

3.2.1 Database Metadata

The first thing a client must retrieve is information regarding the layout of the relational database, known as the *catalog*. As this is a necessity for accessing the system, we can allow the client to retrieve the catalog non-privately. This is reasonable because the only loss of privacy that occurs is that the server and external observers become aware that the client intends to access *some* content within the database. This is not a privacy loss that PIR intends to assuage; the client can use anonymity software such as Tor [10] to send its PIR requests to the database servers if this is in fact a concern.

Foremost, the catalog contains the layout of the database: the relations, the schemas as an ordered set of attributes, and a list of the attributes that have indices. Relations and indices are virtually arranged by the client using the same method as was used by the server during the initial construction of the database, and so the catalog contains the sizes of the relations and the indices that are needed to construct the virtual layout. While currently not in our implementation, the full catalog will also contain host names and TLS public keys for the servers that are hosting the database. Once the client has obtained an authentic copy of the catalog, she will be able to lay out the virtual database, devoid of actual values, and query the servers for the missing values only when the query processor indicates they are needed.

Additionally, the database may want to store statistical information about the relations, such as an approximation of the distribution of attribute values in the relations (e.g. a histogram). This information can be used during query evaluation to estimate the selectivity of predicates, which aids in computing efficient query plans.

3.2.2 Ordered Index Structures

Our system implements ordered indices to map attribute values to their corresponding tuples in the relation. They make use of a full ordering function of the attribute values to construct a BST. Tree indices are used to execute three functions: cardinality searches, point searches, and range searches.

Ordered indices are implemented as sequential access BSTs, where data appears only in the

leaves; internal nodes are used to guide the search. The leaf layer of the tree is stored as a sorted list following the tree to simplify sequential data access, and to group data values on PIR blocks. Each search node contains an attribute value that is used to guide the search to the data layer; the references to other tree nodes are implicit by the structure of the tree. Once the client resolves which block contains her desired value, she simply retrieves the entire block and locates the desired value within. Ranges of values can be found by iterating over the sequential leaf layer until she completes her query. Because the search nodes are only used to guide the client to the correct block in the data layer, most BST searches can be completed with only one or two block retrievals.

Trees are initially partitioned into PIR blocks during the construction phase. An index-building algorithm constructs a balanced BST, and partitions all the nodes into PIR blocks so that binary searches require the fewest PIR queries in the worst case, similar to cache-oblivious search trees [2]. It begins by constructing breadth-first subtrees that fill a PIR block, and then merges unfilled blocks where possible. The clients execute this algorithm to lay out the virtual tree during initialization, and when later using it to navigate they populate the nodes they visit with attribute values via PIR block requests.

Determining the cardinality of a predicate on binary tree indices is implemented with two binary searches. The searches locate the beginning and end positions of the elements that tightly bound the desired range, and their relative offsets in the PIR array indicate the size of the matching set. Not yet implemented in our design is fetching heuristic information about the cardinality of a set, which would operate by navigating simultaneously along two binary search paths until it can bound the cardinality within a tolerance based on the number of levels that the two search paths differ.

Point and range searches are computed by binary searching until the smallest array index whose attribute satisfies the predicate is reached. The client then retrieves the PIR block that contains the attribute-index pairs for matching values. Since these are stored sequentially by increasing attribute value, the client iterates over increasing array positions, retrieving sequential PIR blocks when needed, until the results no longer satisfy the predicate.

3.2.3 Keyword Index Structures

Relations may contain attributes whose values have no meaning when sorted. For the *whois* database this would be the domain name. We need to be able to query for the existence of a domain name in the table, and retrieve its full record if present.

Our system implements a hash-table based index to maintain records for maps from keywords to lists of pertinent records, which forms a one-to-many relationship. We can extend this to many-to-many relationships formed by annotating a record from a set of keywords. There are two routines that these index structures are designed to execute: cardinality searches and point searches.

Building a hash table begins by computing the reverse index of keywords to records; keywords are then represented as integers by using some well-known collision resistant hash function to map keywords to numbers. We use Knuth's hash function for strings [16], the identity function for integers, and compositions of these for constructed types, such as dates. Note that this is *not*

intended to be a cryptographic hash function. For the pragmatic purposes of our implementation, we do not compute individual perfect hash functions suited to the data as suggested by Chor [5], but rather use simple and well-known hash functions.

Data is stored in a *hash table* organized by sequential *hash buckets*, each a PIR block. Hash buckets are uniquely associated with integers corresponding to both the offset in PIR blocks from the beginning of the index and the hash value modulo the number of buckets. Each bucket manages an index of the full hash value of each keyword that maps to the bucket, the length of the list of matching records, and each list of tuples that correspond to a contained keyword.

Cardinality searches on hash tables are computable with a single PIR request. The client hashes the keyword for which they are searching locally and retrieves the corresponding hash index block. Our implementation places the list of keyword hashes and result lengths at the beginning of the block in a table of contents, and so the client simply searches that table to determine the cardinality of her search. Point searches are computed by first looking up the keyword in the appropriate hash bucket's table of contents. The client then examines the length delimited set of matching records. Any records that cannot fit inside the primary hash block are instead represented by a position in the database where the set can be found. Each hash table manages a single *hash heap* where all extra records are placed. This means that point searches can execute either in a single PIR request, or in one request for a table of contents block and one for a hash heap block.

3.3 Query Evaluation

In this section, we discuss how an expression in relational algebra can be compiled into a *query plan* consisting only of PIR block requests. A query plan is a relational algebra parse tree, annotated with access methods for the relations (at the leaves of the tree) and algorithms for performing the intermediate query processing tasks such as joins. As such, the actual data access is either a tuple retrieval from a relation or an index scan. We discussed index traversal in Section 3.2.2 and Section 3.2.3 and we omit the specific details for mapping the low-level data access, such as tuple retrieval, to PIR block requests for brevity. It follows that selection (σ) and projection (π) can be supported by an index search or a sequence of tuple retrievals. Union ($\cup$), set difference ($-$), and cross products ($\times$) can be materialized on the client side following the necessary data access. Thus, the remainder of this section is dedicated to the discussion of generating query plans for arbitrary relational expressions in terms of tuple and index access, with the goal of minimizing data retrieval by exploiting index search in place of relational scans when possible, and by making use of specialized join ($\bowtie$) algorithms which reduce data access by exploiting indices.

3.3.1 Query Optimization

Query optimization is the problem of compiling a query into an equivalent query evaluation plan that is optimal (or near optimal) by some cost measure. In traditional database systems, the optimization target is an estimate of the actual time the query plan will require to execute. This estimate is based on the number of records needed at various stages of computation, as well as

other characteristics such as disk access costs.

For example, a traditional database management system may scan a relation to evaluate a predicate as part of a query plan, instead of using an index to retrieve only the records satisfying the predicate. The initial plan may be deemed optimal despite accessing more records than the index approach. The reason for this is because a predicate can have *low selectivity*, in which many records are expected to satisfy the predicate. Because the records are likely to be stored contiguously on disk, a scan of the entire relation will optimize disk access. An index traversal, on the other hand, could access records in an arbitrary order with respect to physical location on disk, resulting in a longer time to evaluate the query.

The formulation of our problem presents a different optimization target. We pay a constant cost for each PIR block access, independent of storage location, so the additional types of temporal optimization considerations taken in traditional systems do not apply in our situation. Instead, we aim to minimize the total amount of network traffic, which is directly proportional to the number of PIR requests in a query plan since retrieval blocks are of a fixed size. Also, because the database system is oblivious to the actual data values being retrieved, it is not able to evaluate any part of the queries. Thus the query optimization and plan evaluation must be done on the client side.

Computing Query Plans

There are two sides to the query optimization problem. The first entails a manipulation of the relational algebra statement using common heuristics to produce an equivalent statement that reduces the size of intermediate results. For brevity, we omit a thorough discussion of these optimizations as they can be found in most relational database texts, such as [19]. The general idea is to perform projections and selections as early as possible, thus reducing the amount of data considered at intermediary processing phases. The second side of query optimization involves choosing an access plan for each relation referenced in the query, and possibly further manipulating the form of the query to alter the order in which relations are considered. One then needs to decide which algorithms will be used to join relations. The space of all possible query plans is therefore all possible ways of accessing and combining the relations, in addition to the unary relational algebra operators. An overview of these challenges and common approaches in query optimization can be found in [3].

Estimating the cost of a plan often requires information such as the cardinality of a predicate which is not available without querying the database (thus adding to the overall cost of query evaluation). A database may store statistics about the data, which can be used to estimate the cardinality of intermediate results without actually accessing the data. In many cases a plan that is optimal for one database instance can be sub-optimal for a different database instance because of differences in the distributions of attribute values; for example, a range predicate may be satisfied by all tuples of one database instance, and no tuples of another. Because of this property, it is not possible to have an optimal query optimization algorithm without perfect knowledge of the data.

In our model we consider the space of *left-deep* plans to reduce the search space, a common approach taken by many relational query optimizers. This means that the right child of any node in

the plan will correspond to data access for a relation. We do not currently maintain statistics about the database, but opt for some simple heuristics to estimate the relative selectivity of predicates. Predicates in the query are analyzed to determine when a full relation scan can be replaced by an index scan that simultaneously satisfies the predicate. We choose indices based on the following simple estimates of selectivity: predicates on primary key attributes are considered the most selective, followed by point predicates, then range predicates. Predicates on clustered indices—that is, indices whose order is the same as the order of the relation—are always used in the case of multiple alternatives. Our system supports two basic join operations: nested loop join, and index nested loop join. A nested loop join is a simple approach for materializing a full join in arbitrary situations. If a join condition on an indexed attribute is detected in the query, then our optimizer will use the index nested loop join algorithm, a specialized join procedure which reduces data access by *searching* for the joining tuples, rather than scanning an entire relation. Our optimizer will further arrange for the input with smaller cardinality to be in the outer loop, resulting in fewer required searches.

3.3.2 Aggregate Operators

Relational algebra with aggregates adds to the utility and expressiveness of the query language. Implementing the most common aggregates is a straightforward extension of the work presented thus far. We summarize how to evaluate some common aggregates below.

- ***Count***: The count operator simply returns the cardinality of a query result, rather than the result itself. The obvious implementation is to evaluate the query and perform a linear scan through the result set to obtain a count of the number of records. In the case of a range query over an indexed attribute however, the cardinality searches discussed in Sections 3.2.2 and 3.2.3 are a much more efficient approach.
- ***Min/Max***: The min operator returns the minimal value in a query result. The obvious implementation here is to evaluate the query, sort the result, and return the first element in the sorted result. In the presence of a tree index however, the min operator can be evaluated by returning the leftmost leaf-node in the tree that satisfies the query predicates (or just the first leaf in the tree if no such predicates exist in the query). The evaluation of the max operator is analogous.
- ***Sum/Average***: The sum operator computes the sum of a given attribute over a query result. A linear scan through the result is thus unavoidable, however, in the case where a sum is to be computed over a tree indexed attribute, we can save by computing the sum over the data stored in the leaf index nodes, rather than the relation's records. This is beneficial since the size of index nodes will generally be much smaller than the size of records, resulting in less data being accessed to compute the sum. The same approach applies to the average operator as well, since we can count the number of results during the scan to compute the average.

While our implementation does not currently support aggregates in the query interface, our backend query processing infrastructure is equipped to handle them. It can perform the cardinality searches

previously discussed for counting and tree index navigation for min/max. Computing sums and averages is a simple extension of the data access our system currently supports.

4 Implementation and Evaluation

We built an experimental implementation of the proposed system using the *Percy++* PIR library [12]. The query processor is written in Python and the virtual database and the PIR proxy are written in C++. We report on some experiments in Section 4.2 using this implementation, and discuss our future plans in Section 5.

4.1 Implementation

Our implementation consists of four main components: a query processor, a virtual database, a PIR proxy, and a PIR block server. The query processor maps SQL statements into database operations. The virtual database caches data that has been retrieved from the PIR block server, and maintains a virtual model of the database layout so that it knows which blocks must be retrieved in order to satisfy requests. It also contains the PIR proxy that dispatches requests for blocks. The PIR block server holds a raw binary copy of the database, which it uses to respond to requests for data.

Combining these components yields a system which greatly improves the usefulness of PIR, allowing complex SQL queries instead of simple address-based block (or bit!) retrieval. This improvement informs the choice of the name TRANSPIR for our work: TRANSPIR goes beyond the capabilities of traditional PIR. TRANSPIR will operate with any PIR system that can compute the function *getBlock(i)*. We chose the open-source PIR system Percy++ which provides a C++ interface to client-server PIR protocols [12]. The author has extended the functionality of Percy++ since its initial publication; it can now process multiple private block requests simultaneously and requires only a couple of seconds to respond to a private block request on a one gigabyte database.

Our query processor takes as input relational algebra queries in an SQL-style syntax. The queries are then compiled into query plans as detailed in Section 3.3.1. By adhering to this simple query optimization procedure, we generate optimal or near-optimal plans in all of the tested cases.

As with any query optimization procedure, there is always room for improvement. We currently do not make use of database statistics. While this is not an issue for the simple *whois* example database, it could be much more relevant in other situations. Also, we use a small set of fairly simple heuristics to navigate the space of query plans. It would be worthwhile to explore alternative heuristics which could take advantage of new evaluation algorithms, such as index merging procedures. Lastly, the largest restriction of our query processor is its in-memory processing design. Because of this, we can not evaluate queries that have intermediate results larger than the available memory. We plan to address these issues in future revisions to TRANSPIR.

TRANSPIR's virtual database's interface is through the catalog object. On construction, the catalog is initialized with a catalog file containing the data that was discussed in Section 3.2.1. The catalog implements the interface for the virtual database, exposing methods to retrieve tuples and search indices to the relation.

The raw block data, once retrieved from the PIR block server, is cached on the client side in pages, each corresponding to a single PIR block. The caching class, known as the AtomFactory, manages all requests and dispatches for data. Each byte of data, or atom, is associated to its position in the database, and atoms are accessed by requesting their positions from the AtomFactory. Groups of atoms, forming the basic types in our system, are called compounds. When the user wishes to operate on a raw sequence of data, she creates a compound of the appropriate type, initialized with its position in the virtual map. Compounds are equipped with all of the functionality needed to interact usefully with raw bytes from the database, such as marshalling, demarshalling, comparing, and hashing. Aggregates are responsible for managing compounds; they span large ranges of the virtual map, such as relations and indices. Aggregates are responsible for knowing the ranges spanned by their data, the method of constructing their components, and the types of compounds at each position they manage.

4.2 Experimental Design

We performed experiments on our system using a series of randomly generated *whois*-style datasets of varying sizes (see Section 4.3). The data consists of two relations: a *people* relation, consisting of information about the individuals or companies who register domains or are registrars of domains, and a *registrations* relation, containing all of the registered domain names, their registration and expiration dates, a reference to the registering contact, and a reference to the registrar. Both the contact and registrar fields have foreign key constraints to the *people* relation. Both relations also store an *id* field to uniquely identify each record. The detailed schema can be found in Appendix B.

We create a hash index on the *registrations* relation's *domain* attribute, and the *people* relation's *id* attribute. A tree index is created on the *registrations* relation's *registration date* and *expiration date* attributes.

4.3 Evaluation

To test the core functionality of relational query evaluation, we designed a suite of nine micro-benchmarks exercising our index structures. These correspond to searching, scanning, and evaluating cardinality queries. Additionally, we tested six typical SQL queries for our whois data. These test the interaction between the query processor and the back-end database holistically. These queries test various types of predicates (Q1-Q4), as well as joins (Q5, Q6). Details of our SQL benchmarks can be found in appendix A.

These benchmarks were evaluated over three datasets. The small dataset consists of 1,000,000 registration tuples and 750,000 people tuples, approximately yielding a 256 MB database (including indices), with a block size of 16 KB. The medium dataset consists of 2,000,000 registration tuples and 1,500,000 people tuples, approximately yielding a 512 MB database with a block size of 24 KB. The large dataset consists of 4,000,000 registration tuples and 3,000,000 people tuples approximately yielding a 1 GB database with a block size of 32 KB. No special care is take to align

tuples to block boundaries; block sizes are chosen for Percy++ as discussed in [12].

Our experiment computes the following results: the number of tuples returned by the query, the time required to execute the query, the number of PIR requests performed, and the total bandwidth required to compute the result. The following linear relationships should hold in the result set. First, the number of block requests will be linear with the elapsed time, as server PIR block retrieval is the dominant factor in query processing. The great majority of the elapsed time is server-side processing of the query by the Percy++ PIR backend. Second, the total data transfer will be the number of blocks retrieved times the size of a block times the number of servers (two in our case) times two flows (since client requests are of equal size as server responses).

The micro-benchmarks are divided into three categories: hash index benchmarks, tree index benchmarks, and cardinality search benchmarks (note that we query the cardinality of arbitrary predicates, not entire relations which can be trivially answered using the catalog data). We omit individual record retrieval benchmarks as these are analogous to the simple block retrieval experiments reported in [12], with consideration for the recent improvements already discussed. The results of the micro-benchmark experiments are presented in Table 1.

The labels in Table 1 describe the type of test being performed for each benchmark. These benchmarks cover a range of fundamental query processing operations. It is evident from these results that tree and hash based index searches over PIR have a tolerable overhead when the result set is a small number of tuples. In particular, our results scale well in all measured situations; increasing the size of the database does not impact the number of blocks required to perform operations such as cardinality searches and fetching single tuples.

As can be seen in the tree point and tree range multi-result benchmarks, fetching a set of tuples at arbitrary locations in the relation has a worst-case block access equal to the number of tuples being returned (bounded by the number of blocks in the relation). This was expected a priori, since retrieving a set of data from the database requires, at minimum, retrieving the smallest set of blocks that spans all the desired data. Improving the database schema and data layout based on anticipated query needs would mitigate this poor performance. This could be done, for example, by clustering records on commonly queried attributes.

Cardinality searches on hash tables take a single block request as expected. The client simply retrieves the table of contents immediately from the appropriate hash bucket and returns the cardinality of the keyword if it exists. Tree cardinality searches take more block requests because they must retrieve two disjoint blocks in the leaf layer to compute the exact cardinality of the range.

Table 2 shows the results for evaluating the SQL queries over all three databases. The evaluation of an SQL query may involve multiple index searches, retrieval of individual tuples, along with client side processing of predicates and materialization of joins. The total time reported includes query parsing, optimization, and evaluation for an individual query operating with a clear cache. The other columns of the table are measured the same way as described for the micro-benchmark experiments. Again we see that the dominant factor of processing time is in the processing of PIR requests. Because we did not design the data layout for the particular workload, these queries exercise worst case scenarios in which an entire block must be retrieved for each tuple. Our general observations for the micro-benchmarks extend to full SQL query processing in that queries with

256 MB Database (16 KB Blocks)

Access Type	Result Size	Time	PIR Requests	Data Transfer
Hash single result	1 Tuple	6.3 s	2 Blocks	128 KB
Hash no result	0 Tuples	3.1 s	1 Block	64 KB
Hash multi-result	1 Tuple	6.2 s	2 Blocks	128 KB
Tree point multi-result	16 Tuples	55.7 s	18 Blocks	1152 KB
Tree no result	0 Tuples	3.1 s	1 Block	64 KB
Tree range multi-result	56 Tuples	175.9 s	57 Blocks	3648 KB
Clustered tree point single result	1 Tuple	3.1 s	1 Block	64 KB
Hash cardinality search (point)	2 (count)	3.1 s	1 Block	64 KB
Tree cardinality search (range)	81 (count)	6.2 s	2 Blocks	128 KB

512 MB Database (24 KB Blocks)

Access Type	Result Size	Time	PIR Requests	Data Transfer
Hash single result	1 Tuple	18.5 s	3 Blocks	288 KB
Hash no result	0 Tuples	6.1 s	1 Block	96 KB
Hash multi-result	3 Tuples	24.3 s	4 Blocks	384 KB
Tree point multi-result	33 Tuples	212.5 s	35 Blocks	3360 KB
Tree no result	0 Tuples	6.1 s	1 Block	96 KB
Tree range multi-result	86 Tuples	528.0 s	87 Blocks	8352 KB
Clustered point single result	1 Tuple	6.0 s	1 Block	96 KB
Hash cardinality search (point)	5 (count)	6.1 s	1 Block	96 KB
Tree cardinality search (range)	599253 (count)	18.14 s	3 Blocks	288 KB

1 GB Database (32 KB Blocks)

Access Type	Result Size	Time	PIR Requests	Data Transfer
Hash single result	1 Tuple	26.3 s	2 Blocks	384 KB
Hash no result	0 Tuples	11.9 s	1 Block	128 KB
Hash multi-result	3 Tuples	47.8 s	4 Blocks	512 KB
Tree point multi-result	58 Tuples	691.9 s	58 Blocks	7424 KB
Tree no result	0 Tuples	11.9 s	1 Block	128 KB
Tree range multi-result	167 Tuples	1987.3 s	167 Blocks	21376 KB
Clustered point single result	1 Tuple	11.9 s	1 Block	128 KB
Hash cardinality search (point)	5 (count)	11.9 s	1 Block	128 KB
Tree cardinality search (range)	1344288 (count)	35.7 s	3 Blocks	384 KB

Table 1: TRANSPIR micro-benchmark results

256 MB Database (16 KB Blocks)

Query	Results	Time (s)	PIR Reqs	Data Xfer
Q1	1	9.9 s	3	192 KB
Q2	16	61.3 s	19	1216 KB
Q3	20	74.1 s	23	1472 KB
Q4	13	73.9 s	23	1472 KB
Q5	1	12.9 s	4	256 KB
Q6	20	141.4 s	44	2816 KB

512 MB Database (24 KB Blocks)

Query	Results	Time	PIR Reqs	Data Xfer
Q1	1	12.9 s	2	192 KB
Q2	34	232.1 s	37	3552 KB
Q3	56	369.2 s	59	5664 KB
Q4	31	369.1 s	59	5664 KB
Q5	1	18.8 s	2	192 KB
Q6	56	712.3 s	114	10944 KB

1 GB Database (32 KB Blocks)

Query	Results	Time	PIR Reqs	Data Xfer
Q1	1	25.1 s	2	256 KB
Q2	73	927.0 s	76	9728 KB
Q3	127	1588.7 s	130	16640 KB
Q4	65	1587.8 s	130	16640 KB
Q5	1	36.5 s	3	384 KB
Q6	127	3086.8 s	253	32384 KB

Table 2: TRANSPIR SQL query results

small result sizes have a tolerable overhead in both time and data transfer. However, as the query result sizes and database grow larger, the processing time starts to become prohibitively large. Applications that require this type of query processing on large databases must strongly consider how much they value the privacy of their queries.

Performance of an entire workload is not the sum of individual query performance as clients may greatly benefit from cached PIR blocks in the proxy when running a sequence of consecutive queries. This is because some queries may share predicates or traverse the same tree index (such as Q2, Q3, Q4, and Q6). Queries can benefit from PIR blocks cached by previous queries and some blocks may be frequently used, such as a hash index header or the blocks encoding the top regions of a tree index. Table 3 shows the total time to process the entire workload over each database without clearing the cache between queries. These times show a substantial improvement over the sum of individual query times, indicating that clients processing consecutive queries may benefit if their queries share predicates or search the same index structures. We note that this improvement is highly dependent on the workload.

4.4 Query Size

The use of PIR ensures that the contents of each block query are hidden from the database servers. However, one piece of leaked information is the *number* of PIR queries that were performed. Performing multiple successive queries reveals no information about any particular query, but the total number of queries may fingerprint a popular request. One approach, used by the Pynchon Gate [21], is to top up the number of queries made to some system-defined maximum. In general, however, it is possible for clients to execute queries that retrieve large amounts of data, such as an entire relation, or the join of multiple relations. Hence, there is no fixed maximum that can be used for all relational algebra queries. Individual applications, on the other hand, may certainly exploit domain-specific knowledge in order to confound an attacker's ability to guess the query from its size.

Even in the general case, however, we argue that the number of possible queries greatly devalues the information that is leaked by the client by briefly considering the space of queries. Recall query plans as discussed in Section 3.3; the leaf layer corresponds to data accesses from relations or indices. Because each PIR block request from a query plan could correspond to any type of data access from one of any of the available database resources, there is no way to reliably fingerprint a query based on the number of PIR blocks requested. For example, the server is unable to differentiate between a large data access (such as a single relation scan) and a number of smaller data accesses (such as multiple index scans).

We further argue that the server cannot use knowledge of a popular query to fingerprint the number of block requests in order to break privacy. PIR servers should be entirely unable to identify popular queries over unpopular ones, as PIR is used exactly to ensure that servers cannot collect information on what queries are being performed. Indeed, performing a query that the server has already identified as popular obviates the use of PIR entirely. For instance, if a whois server knew a particular domain request was popular and yet unregistered, then it might act on that knowledge

DB Size	Time	PIR Requests	Data Transfer
256 MB	210.7 s	65	4160 KB
500 MB	957.0 s	153	14688 KB
1 GB	4023.2 s	330	42240 KB

Table 3: TRANSPIR SQL workload results

independently of any further requests. This is not the problem we are attempting to solve, and it would be better addressed by a client-privacy system, such as Tor [10], as opposed to a query-privacy system, such as TRANSPIR.

5 Future Work

We are in the process of evolving our experimental TRANSPIR implementation into a deployable and functional API extending the Percy++ project. Some additional avenues for future work include extending our query language to include richer constructs, such as support for the complete SQL query language. We note that many of the remaining SQL constructs can be supported in our framework by additional client side operations. Future considerations for database encoding are database compression and column-oriented storage. By including database compression techniques we can further reduce network traffic. Column-oriented storage stores the relations by column, rather than by rows. In this layout, all of the data values for the same attributes of subsequent rows get grouped in the same PIR blocks, allowing queries over non-indexed attributes to be answered with far fewer PIR requests than the case where entire rows need to be retrieved. The disadvantage of this approach is that retrieving entire records becomes very expensive. One must have knowledge of the query workload for the particular application before making this type of design decision.

Allowing updates of the database by a client with write privileges is an interesting direction for future work. Support for client updates would require an extension of the PIR interface as proposed, for example, by [17]. Additionally the database would have to employ some sort of transactional model and concurrency control. This opens up new issues in privacy, since explicitly locking resources for writing leaks information about the nature of the update. One would have to explore the idea of a *private lock* to avoid locking the entire database. Such locking information would be retrieved privately by clients before initiating a transaction. The clients would then have to employ the locking protocol on the client side, much like the query optimization and query evaluation phases.

6 Conclusion

We have proposed and implemented an improvement to private information retrieval to allow expressive relational algebra queries to be made privately. Our design consists of four components: the query processor, the virtual database, the PIR proxy and the PIR block server. We used the Percy++ PIR library for our system and analyzed the performance of our implementation for different types of queries. Our experimental results suggest that private relational querying is a feasible technology to deploy in certain real-world applications.

References

[1] A. Beimel and Y. Stahl. Robust information-theoretic private information retrieval. In *Proceedings of the 3rd Conference on Security in Communication Networks*, 2002.

[2] M. Bender, E. Demaine, and M. Farach-Colton. Cache-Oblivious B-Trees. In *IEEE Symposium on Foundations of Computer Science*, 2000.

[3] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM symposium on Principles of database systems*, pages 34–43, New York, NY, USA, 1998. ACM.

[4] B. Chor and N. Gilboa. Computationally private information retrieval (extended abstract). In *Proceedings of the twenty-ninth annual ACM symposium on Theory of Computing*, 1997.

[5] B. Chor, N. Gilboa, and M. Naor. Private Information Retrieval by Keywords. Technical report, TR CS0917, Department of Computer Science, Technion, 1997.

[6] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private Information Retrieval. *Journal of the ACM*, 1998.

[7] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[8] E. F. Codd. *The relational model for database management: version 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[9] G. Danezis, R. Dingledine, and N. Mathewson. Mixminion: Design of a Type III Anonymous Remailer Protocol. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, 2003.

[10] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second-Generation Onion Router. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[11] Y. Gertner, S. Goldwasser, and T. Malkin. A Random Server Model for Private Information Retrieval or How to Achieve Information Theoretic PIR Avoiding Database Replication. In *Proceedings of the Second International Workshop on Randomization and Approximation Techniques in Computer Science*, 1998.

[12] I. Goldberg. Improving the Robustness of Private Information Retrieval. *IEEE Security and Privacy*, 2007.

[13] International Standards Organization. ISO/IEC 9075-11:2003 Information technology - Database languages - SQL - Part 11: Information and Definition Schemas (SQL/Schemata), 2003.

[14] Internet Corporation for Assigned Names and Numbers Security and Stability Advisory Committee. SAC 022: SSAC Advisory on Domain Name Front Running. http://www.icann.org/committees/security/sac022.pdf.

[15] L. Kissner, A. Oprea, M. Reiter, D. Song, and K. Yang. Private keyword-based push and pull with applications to anonymous communication. *Applied Cryptography and Network Security*, 2004.

[16] Donald E. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)*. Addison-Wesley Professional, April 1998.

[17] R. Ostrovsky and V. Shoup. Private Information Storage. *ACM Symposium on Theory of Computing*, 1997.

[18] M. Rabin. How to exchange secrets by oblivious transfer. Technical report, Tech. Memo TR-81, Aiken Computation Laboratory, Harvard University, 1981.

[19] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, New York, NY, USA, 2003.

[20] F. Saint-Jean. Java Implementation of a Single-Database Computationally Symmetric Private Information Retrieval (cSPIR) protocol. Technical report, YALEU/DCS/TR-1333, Department of Computer Science, Yale University, July 2005.

[21] L. Sassaman, B. Cohen, and N. Mathewson. The Pynchon Gate: A Secure Method of Pseudonymous Mail Retrieval. In *Proceedings of the Workshop on Privacy in the Electronic Society (WPES 2005)*, 2005.

[22] World-Wide Web Consortium. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, January 2007.

A Workload

Q1 – point query (hash-index), single result

```
SELECT domain, registration_date
FROM registrations WHERE domain =
?x
```

Q2 – point query (tree-index)

```
SELECT domain FROM registrations
WHERE expiration_date = ?x
```

Q3 – range query (tree-index)

```
SELECT domain, status FROM
registrations
WHERE expiration_date > ?x
```

Q4 – range predicate (tree-index), range predicate (no index)

```
SELECT * FROM registrations
WHERE expiration_date > ?x AND
registration_date < ?y
```

Q5 – join (index nested loop) with point predicate

```
SELECT domain, name, email FROM
people,
registrations WHERE domain=?x AND
contact=p_id
```

Q6 – join (index nested loop) with range predicate

```
SELECT * FROM people,registrations
WHERE
expiration_date > ?x AND registrar
= p_id
```

B Experimental Data Schema

Registrations:	
reg_id	UInt32 (PRIMARY KEY)
domain	VARCHAR(64)
exp_date	UInt16
reg_date	UInt16
contact	UInt32
registrar	UInt32
Status	UInt8
FOREIGN KEY contact REFERENCES PEOPLE.ID	
FOREIGN KEY registrar REFERENCES PEOPLE.ID	

People:	
p_id	INT
email	VARCHAR(64)
name	VARCHAR(64)
addr	VARCHAR(512)