Evaluating Private Information Retrieval on the Cloud

Casey Devet

University ofWaterloo
cjdevet@cs.uwaterloo.ca

Abstract

The goal of Private Information Retrieval (PIR) is for a client to query a database in such a way that no one, including the database operator, can determine any information about the desired database record. Many PIR schemes have been developed, but are generally not deemed fast enough for client needs in today's fast-paced world or do not scale well to large databases. We implement a parallelized version of one PIR scheme and evaluate it in a cloud computing environment. We show that our parallelization improves the performance and scalability of this PIR scheme. We also discuss what should be done if multiple queries are needed and how to split the database to get the most out of this improvement.

1 Introduction

There are many situations where people or institutions do not wish for others to learn about private information. One such situation is when a user wishes to fetch private information from a database. For the user to maintain the privacy of the fetched information, the information must be kept secret in transit. This can be done simply using one of a variety of common communication encryption techniques. The user must also be assured that the operator of the database cannot determine any information about the data fetched. This is the goal of Private Information Retrieval (PIR). PIR has applications in many proposed privacy-sensitive applications including patent databases, anonymous email and domain name registration [4].

One approach to PIR is for the user to request the entire database from the server. In 2007, Sion and Carbunar suggested that this approach, called *trivial download*, would always be faster than any other PIR scheme [10]. However, a more recent work by Olumofin and Goldberg has shown that this is not case [9].

There are two types of PIR schemes. Computational PIR (CPIR) schemes provide privacy for fetched information given that the computational power of the database operator is polynomially bounded. Alternatively, Information-theoretic PIR (ITPIR) schemes provide information privacy regardless of the computational power of the database operator. To do this, a database query must be split over multiple servers [2], each with a copy of the database, and we must assume that for some threshold t, no more than t servers are colluding. This is a common assumption in many privacy enhancing technologies, including Tor, mix networks and some electronic voting schemes [4].

As shown by Mayberry et al. [8] and Huang [7], we can take advantage of the computational abilities of cloud computing infrastructure to improve the performance of PIR schemes of both flavours. By parallelizing the server computations needed for PIR, we can speed up the query time for private information retrieval to a level necessary in today's fast-paced world.

1.1 Related Work

Our work does not aim to create a new PIR scheme, but to improve current schemes by parallelizing computation-intensive parts of them. Many different PIR schemes, both CPIR and ITPIR, have been developed along with proven computational bounds. In 2007, Sion and Carbunar evaluated a single CPIR scheme and concluded that, at that point, none of the CPIR schemes could perform better than the trivial download solution [10]. However, many of these schemes could be potentially improved by doing some of the involve computations in a cloud computing environment.

Blass et al. introduce PRISM, a scheme for privacy-preserving word search in a cloud computing environment [1]. This work transforms the word search problem into a set of parallel PIR instances on parts of the dataset. In PRISM, a user uploads a set of encrypted data to the cloud and at a later date needs to search the cloud to determine whether or not a word exists. PRISM uses Google's MapReduce framework for sharing the computational load over the cloud. The PIR scheme used for this work is a computational PIR scheme.

More recently, Mayberry, Blass and Chan generalize this approach with PIRMAP [8]. This scheme allows a user to privately retrieve an file from a set of files on a server. In PIRMAP, files are distributed evenly to the worker nodes as well as the encrypted query sent from the user. The work is split over the cloud using MapReduce. Both PIRMAP and PRISM require that a "somewhat homomorphic" encryption scheme be used by the client to hide the contents of the query vector and to decipher the contents of the response from the server. This amount of client-side computation is non-trivial and can be avoided using ITPIR schemes.

In a very similar approach to PIRMAP, Huang [7] introduces MapReducePIR on top of Goldberg's IT-PIR scheme [6]. In Goldberg's scheme the bottleneck in computation is that the server must perform a vector-by-matrix multiplication where the matrix represents the database. Huang uses MapReduce to parallelize this computation. Although the parallelization of this computation seems straightforward, Huang's results do not show an improvement over the non-parallelized version of the same scheme. He suggests the following reasons for these results:

1) Poor choice of cloud testing environment.

2) Efficiency of the Java implementation of Hadoop MapReduce compared to the C++ implementation of Goldberg's scheme.

3) Using poor parameters for PIR.

4) Splitting the database differently may be more efficient.

1.2 Our Contributions

We continue the work started by Huang in parallelizing the server-side computation of Goldberg's PIR scheme. We implement a parallel version of this scheme without using Hadoop MapReduce. This lets us evaluate our work without the possible inefficiencies of Hadoop presented as point (2) above from Huang's work. The main goal of this work is to show that parallelization of Goldberg's PIR scheme improves the time for a server to process a client's query.

Sufficiently Large Databases

From his testing, Huang suggests that his implementation may be faster than the non-parallelized version of Goldberg's scheme if the database is sufficiently large [7]. This is because the query time of his implementation slowly approaches that of the non-parallelized version as the database gets larger. We observe the following through our experiments:

A) *A database needs to be sufficiently large for parallelization to be a benefit.*

Multiple Queries and Locality

The MapReduce framework takes advantage of systems with a distributed file system by selecting a worker to do work on files that are located locally on that worker [3]. This reduces the amount of network traffic and speeds up the computation time. Unlike MapReduce, our implementation does not make use of this concept. However, we do something that provides a similar speed up of computation. Our systems allows users to make several queries of the database at once. By doing this, they are able to take advantage of having the database still located in memory for the queries after the first one. We observe the following:

B) *When multiple queries are made, there is a significant speed up in query time after the first query.*

Database Split

Our implementation parallelizes the server-side computation by splitting the database into smaller subdatabases. As mentioned by Huang, (point (3) above), we may achieve better results by splitting the database in a different way. Through our experiements, we observe the following:

C) *Splitting the database in different ways does not significantly affect the query time.*

2 Implementation

We implement a parallelized version of Goldberg's ITPIR scheme for use in a cloud computing environment. Our implementation does not use MapReduce as done in previous works. This is so that we can be sure that the MapReduce implementation used is not the cause of performance issues. We choose to use MPI (Message Passing Interface) to communicate between nodes. This is because MPI is a very simple interface and is supported on many systems.

2.1 Goldberg's Scheme

Goldberg's ITPIR scheme makes use of Shamir Secret Sharing to ensure the privacy of fetched information. Suppose we have a secret σ in some field $\mathbb{F}$, then we create a random function f of degree at most t that encodes the secret so that $f(0) = \sigma$. We then create a set of shares $\{(\alpha_1, f(\alpha_1)), (\alpha_2, f(\alpha_2)), \dots, (\alpha_\ell, f(\alpha_\ell))\}$ with indices $\alpha_1, \dots, \alpha_\ell \in \mathbb{F}$ with $\alpha_i \neq 0$ and distribute them to a set of ℓ servers. As long as no more than t of the servers are colluding, none of them will be able to determine the secret.

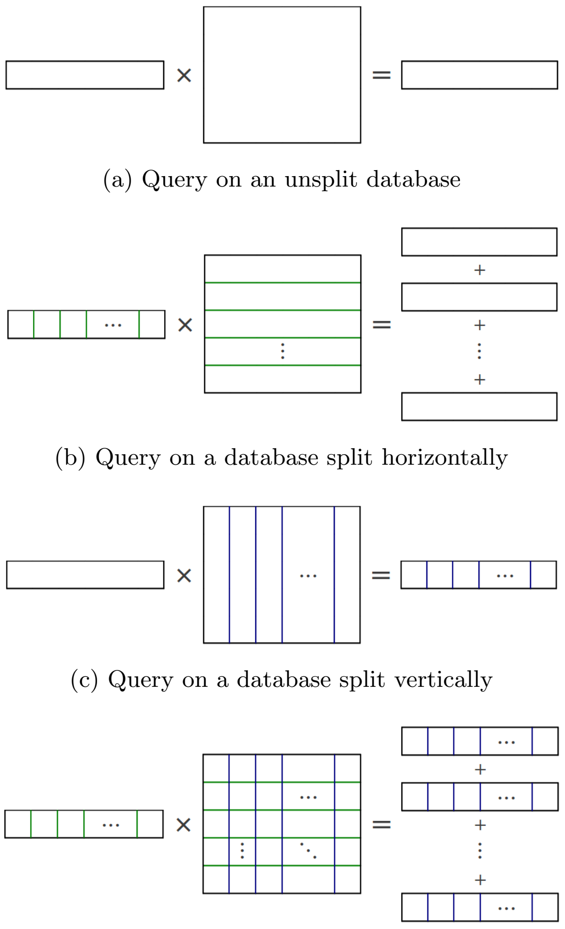
In Goldberg's scheme, we assume that our database is an $r \times s$ matrix D and that each of ℓ servers have a copy of D. Each row of the database represents a record. If we want the record at row β, D_β, we create the elementary vector $\mathbf{e}_\beta = \langle 0, 0, \dots, 1, \dots, 0\rangle \in \mathbb{F}^r$, where the 1 is in the β^{th} position, and multiply by D:

$$D_\beta = \mathbf{e}_\beta \cdot D$$

However, we need to be able to hide β from the database operators, so we use Shamir Secret Sharing to create ℓ shares of $\mathbf{e}_\beta$, $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_\ell \in \mathbb{F}^r$, and send query vector $\mathbf{v}_i$ to server i. Each server computes a response vector $\mathbf{r}_i = \mathbf{v}_i \cdot D$ and returns it to the client. As long as enough servers send their response, the client can recover the database record since $\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_\ell$ are Sharmir Secret Shares for D_β. As long as no more than t servers collude, none of the servers can determine any information about the record that was fetched.

As shown by Devet, Goldberg and Heninger, the client-side computation required for this scheme can be done very quickly [4]. If we compare this with Olumofin and Goldberg's results for the server-side computation [9], we see that, in this scheme, the bottleneck in a database query occurs with this vector-by-matrix multiplication on the server-side. Therefore, by improving the time of these server-side computations, we can improve the performance of the entire scheme.

Goldberg's scheme is implemented in the open source project Percy++ [5]. We implement our work as an extension of this software.



(a) Query on an unsplit database

(b) Query on a database split horizontally

(c) Query on a database split vertically

(d) Query on a database split into submatrices

Figure 1: Ways of splitting the database

2.2 Splitting the Database

Like in Huang's work [7], we will split the database into parts and perform this vector-by-matrix multiplication in parallel computations. That is, we will take the vector-by-matrx multiplication computation for one of the PIR *servers* and split it up over a number of *workers*. So a cluster of workers represents one PIR server.

Both Huang [7] and Mayberry et al.'s [8] implementations split the database horizontally into chunks, each of which span the width of the matrix. If Figure 1a represents an unsplit vector-by-matrix multiplication, then Figure 1b respresents this same operation with the database split horizontally. Notice that if each chunk of the database is given to a worker, that the worker only needs to be given the portion of the query vector that corresponds to their chunk. We have one *master* server that coordinates the workers. When all workers are done the vector-by-matrix multiplication on their portion of the database, the master adds all of the worker responses together and sends the result to the requesting client.

We also can split the database vertically, as in Figure 1c. In this case, each worker receives the entire query, but only produces a portion of the result. The master then concatenates all of the workers' responses and sends the entire vector to the requesting client. We can also combine both methods so that the database is split both vertically and horizontally and each worker does the computation for a submatrix, as shown in Figure 1d.

An advantage of this is that each worker can essentially act as a single PIR server who receives requests from the master as if the master was a client. This also means that this process can be abstracted to provide more layers of database splitting. For example, the master could split the database horizontally and give each chunk to one of a set of intermediate workers. Then these intermediate workers could in turn split their chunks vertically into smaller chunks and send those to their own workers. This sort of abstraction has not been implemented, but could easily be done in the future.

3 Evaluation

The evaluations of our system were done using the Orca cluster on SHARCNET. This system is a cluster of 360 machines of two types. We restricted our testing to the 320 nodes with an AMD Opteron 2.2 GHz processor with 24 cores, 32 GB of local memory and 120 GB of local storage. The nodes are running CentOS 6.x as the operating system.

3.1 Sufficiently Large Databases

To evaulate the performance improvements of our system, we queried a server using the non-parallelized version of Goldberg's scheme and our parallelized version. We tested the parallelized version by splitting our database into 16 and 64 subdatabases, using a split of 4x4 and 8x8, respectively. By "AxB" we mean that we split the database vertically by A and horizontally by B. For each of the above 3 configurations, we ran 100 tests, each of which queried the server 20 times. Figure 2 presents our results.

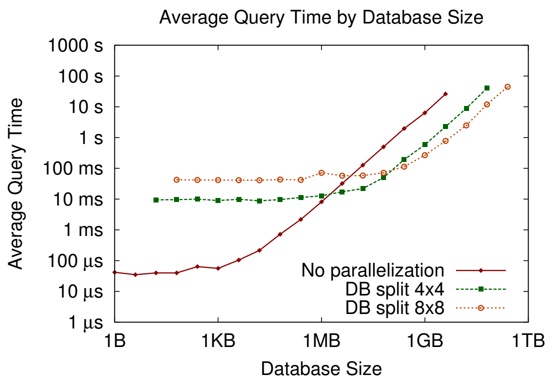


Figure 2: Average query times by database size for each configuration.

Note that, for all configurations, the query times remain approximately constant for small databases. Then when the database size reaches some threshold, the query time grows linearly with respect to database size. This is likely due to the parts of the computation that are the same regardless of database size. This leads to the following observation.

We observe that the database must be sufficiently large for parallelization of the server-side computation to be beneficial. This is an important observation because if database operators are not going to be using a large database, they should not be using this parallelization, but simply using the basic vector-by-matrix multiplication from Goldberg's non-parallelized scheme. This is likely because of the increase in the amount of computation that is constant regardless of database size. Specifically, by splitting the database into more chunks, we require more network communication and the master must do more work. We see that, indeed, the query time for small databases is larger when the database is split into more pieces. We further note that in this case, a sufficiently large database is not very large (only a few MB).

Finally, we note that the parallelized versions improve the non-parallelized verion by approximately a factor of the number of chunks the database is split into. This is evidenced in Figure 2 since for sufficiently large databases, the 4x4 version runs with approximately the same time as a database with 1/16 of the size and, similarly, the 8x8 version runs with approximately the same time as a database with 1/64 of the size. This is the expected result of parallelization.

3.2 Multiple Queries and Locality

As stated in Section 1.2, our system allows a client to make multiple database queries one after another. This takes advantage of having the database still in memory for the queries after the first. Figure 3 shows the average query time based on the query number for each configuration. each line represents the tests for a particular size of database.

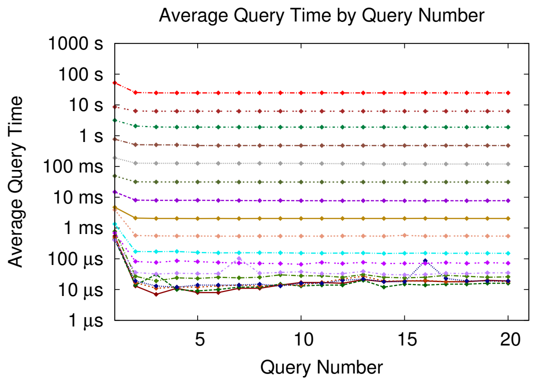
We see that there is an immediate speed up when comparing the first query with the second query. For sufficiently large databases, all queries after the first, take approximately the same time. The query time for queries after the first was generally at most 1/3 of the computation time of the first query. Because of this, if clients need multiple records from the database, they should query for them all at once to take advantage of this property.

The main reason for this observation is that the database chunks needed to be fetched from the network. These results could be improved by first tasking workers to process chunks of the database that are local for the workers. This could make the difference between the first and subsequent queries much smaller.

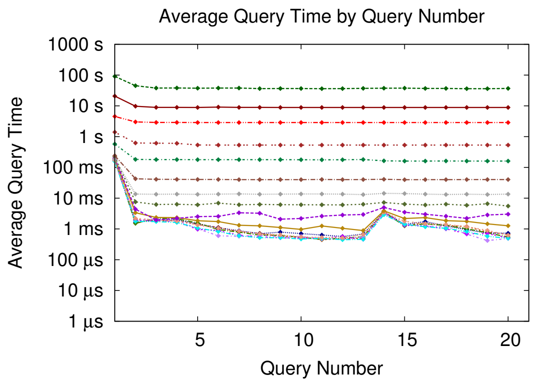
3.3 Splitting the Database

We attempt to evaluate which method of splitting the database, as described in Section 2.2, gives the best computation time for a query. To do this, we tested our implementation on a 4 GB database and seven different ways of splitting the database into 64 chunks. Our results are shown in Figure 4.

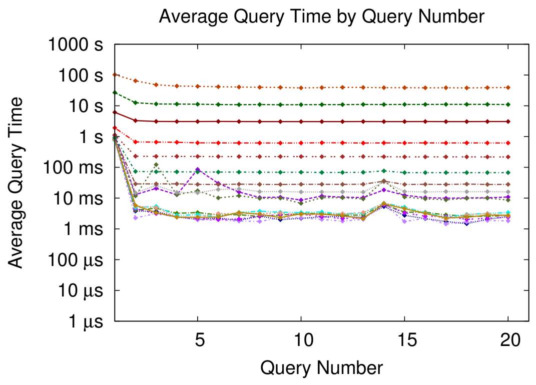
We conclude that the method of splitting the database does not have any significant affect on the query time because, for all splits, the averages are close together and the standard deviations are fairly large, meaning some cases see fairly small or large query times compared to the average. This is significant because it means that a database operator



(a) No parallelization



(b) Database split 4x4



(c) Database split 8x8

Figure 3: The average query time by query number. Each test queried the server 20 times.

Database Split	**Average Query Time**	**Standard Deviation**
1x64	0.773997	0.406644
2x32	0.639325	0.362273
4x16	0.662676	0.394096
8x8	0.777591	0.399065
16x4	0.695891	0.362284
32x2	0.796591	0.453842
64x1	0.907545	0.459233

Figure 4: Average query times for a 4 GB database by how the database is split.

can split the database in any way that is convenient without concern about the affect on query times.

4 Future Work

There are several possible extensions to this paper and our implementation that we feel are worth exploring:

Distributed File Systems

As stated in Section 1.2, MapReduce takes advantage of the locality of files in the distributed file system on a cluster. Our system does not currently do this. Section 3.2 illustrates the increase in query time when a file needs to be fetched from the network. Because of this, we would like to incorporate locality properties into our algorithm.

Relation Between Database Size, Number of Chunks and Usefulness of Parallelization

This work observes that a database must be sufficiently large for there to be any benefit to parallelizing the server-side computation of Goldberg's scheme. We suggest that there is possibly a relation between the number of chunks that a database is split into and the minimum database size for useful parallelization.

CPIR Schemes

This work focused on ITPIR schemes, however some research has been done into parallelizing CPIR schemes [8] [1]. More work could be done to evaluate the possibility of parallelizing different CPIR schemes, as work in this area has generally focused on a couple of CPIR schemes.

5 Conclusions

We expand on the work of Huang [7] by implementing a parallelized version of Goldberg's ITPIR scheme. We tested our implementation on a cluster and found that for sufficiently large databases (approximately 1 MB), parallelizing the server-side computations improves the query time of the scheme. Furthermore, we observe that the improvement is approximately by a factor of the number of pieces the database is split into. We also observe that it is beneficial for a client to perform multiple queries at once to take advantage of the database being in memory. Finally, we conclude that the way that the database is split does not significantly affect query times.

6 Acknowledgements

We would like to thank the Shared Hierarchical Academic Research Computing Network (SHARCNET) and Compute/Calcul Canada for the computing cluster on which the tests for this work were performed. We would also like to thank Ian Goldberg and Bernard Wong for their support and helpful discussions in the process of this work.

References

[1] E.-O. Blass, R. D. Pietro, R. Molva, and M. Önen. Prism - privacy-preserving search in mapreduce. In S. Fischer-Hübner and M. Wright, editors, *Privacy Enhancing Technologies*, volume 7384 of *Lecture Notes in Computer Science*, pages 180–200. Springer, 2012.

[2] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *36th Annual IEEE Symposium on Foundations of Computer Science (FOCS'95)*, pages 41 –50, oct 1995.

[3] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150. USENIX Association, 2004.

[4] C. Devet, I. Goldberg, and N. Heninger. Optimally Robust Private Information Retrieval. In *21st USENIX Security Symposium*, 2012.

[5] I. Goldberg. Percy++ project on sourceforge. `http://percy.sourceforge.net`. Accessed October 2012.

[6] I. Goldberg. Improving the robustness of private information retrieval. In *2007 IEEE Symposium on Security and Privacy*, pages 131–148, 2007.

[7] Y. Huang. MapReducePIR: An implementation and performance analysis. 2011.

[8] T. Mayberry, E.-O. Blass, and A. H. Chan. Pirmap: Efficient private information retrieval for mapreduce. *IACR Cryptology ePrint Archive*, 2012:398, 2012.

[9] F. Olumofin and I. Goldberg. Revisiting the Computational Practicality of Private Information Retrieval. In *15th International Conference on Financial Cryptography and Data Security*, pages 158–172, 2011.

[10] R. Sion and B. Carbunar. On the computational practicality of private information retrieval. In *Proceedings of the Network and Distributed Systems Security Symposium*, 2007.