pCloud: A Distributed System for Practical PIR

Stavros Papadopoulos, Spiridon Bakiras, and Dimitris Papadias

Abstract—*Computational* Private Information Retrieval (cPIR) protocols allow a client to retrieve one bit from a database, without the server inferring any information about the queried bit. These protocols are too costly in practice because they invoke complex arithmetic operations for every bit of the database. In this paper we present *pCloud*, a distributed system that constitutes the first attempt towards practical cPIR. Our approach assumes a disk-based architecture that retrieves one *page* with a single query. Using a *striping* technique, we distribute the database to a number of cooperative peers, and leverage their computational resources to process cPIR queries in parallel. We implemented pCloud on the PlanetLab network, and experimented extensively with several system parameters. Our results indicate that pCloud reduces considerably the query response time compared to the traditional client/server model, and has a very low communication overhead. Additionally, it scales well with an increasing number of peers, achieving a linear speed-up.

Index Terms—Privacy, Private Information Retrieval, Databases, Distributed Systems, Implementation.

1 Introduction

Consider an n-bit database $\mathcal{DB} = \{x_1, x_2, \ldots, x_n\}$. A Private Information Retrieval (PIR) protocol allows a client to retrieve bit x_i, while keeping the value of the index i secret from the server. PIR can also retrieve blocks of data (e.g., an ℓ-bit record) by viewing the database as n/ℓ elements, each with size ℓ bits. The ability to hide the information that a user is interested in (from both the server and other clients) has some very appealing applications. For instance, an investor in the stock market may be unwilling to disclose a trading strategy to other parties. Additionally, browsing privately through a patent or pharmaceutical database may conceal critical information about a company's new product. In the context of location-based services, PIR may be utilized to hide the location of a user that is otherwise revealed through a spatial query [15]. Finally, PIR has been considered as a building block in the Pynchon Gate [24], a system for receiving pseudonymously addressed email.

Given the vast number of applications that may benefit from private queries, PIR has received a lot of attention in the cryptography community. *Single-server* PIR is a family of protocols that assume a non-replicated database stored at a single site. These protocols, also known as *computational* PIR (cPIR), utilize certain cryptographic assumptions to ensure privacy. Figure 1 illustrates the operation of a generic cPIR scheme. Initially, using a randomized query generation algorithm $Q(\cdot)$, the client constructs a query Q_i that is based on the index of the bit to be retrieved. Next, the query is transmitted to the server, which executes a reply generation algorithm $R(\cdot)$ on $\mathcal{DB}$ and returns an answer R_i to the client. Note that the server cannot infer any information (in polynomial time) about the index i through Q_i. Finally, using R_i, the client extracts the value of x_i through a re-construction algorithm $A(\cdot)$.

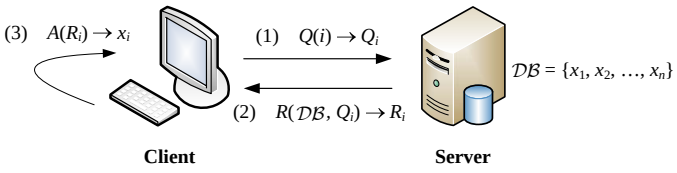


Fig. 1. cPIR framework

In the database community there has been very little work on cPIR because of its prohibitive cost for datasets of practical size. Indeed, the computational complexity at the server is $\Omega(n)$, since the reply generation algorithm has to process every bit in the database; even if a single bit is omitted, query privacy is violated because the server can deduce that the client is not interested in that information. Furthermore, most schemes require at least one modular multiplication for each processed bit, which is a very expensive arithmetic operation. Consequently, Sion and Carbunar [25] argue that it may actually be preferable, in terms of query response time, to transmit the entire database to the client.

However, transmitting the entire database to the client is not a viable solution, for several reasons. First, this approach is not scalable for large databases, because the available bandwidth at the client side is the performance bottleneck that dictates the query response time. In other words, it is impossible for the server to improve the quality-of-service by allocating more resources, such as additional servers, more bandwidth, etc. Second, bandwidth is not always abundant or free, and the cost of

- *S. Papadopoulos is with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong. E-mail: stavros@cse.ust.hk.*
- *S. Bakiras is with the Department of Mathematics and Computer Science, John Jay College, City University of New York, New York, NY 10019. Email: sbakiras@jjay.cuny.edu.*
- *D. Papadias is with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong. E-mail: dimitris@cse.ust.hk.*

downloading a database may be prohibitive in certain cases. For instance, if the client is a wireless device (e.g., PDA, cell phone), the aforementioned method faces several challenges: (i) the wireless channel is typically very slow and error-prone, (ii) the user is charged by the amount of transferred data and may be unwilling to download a large database, and (iii) the device may not have the necessary resources (e.g., memory, battery power) to complete the download.

Motivated by this observation, we introduce *pCloud* (for *Private Cloud*), a distributed system that leverages the computational resources inside a peer-to-peer (P2P) cloud, in order to speed-up the processing of cPIR queries. PIR offers some unique advantages that are appealing to a large user population. Consequently, interested parties may agree to contribute their computational and communication resources and, in exchange, benefit from a PIR system that answers queries in "real-time". A similar example of client cooperation is *The Onion Router* (*Tor*) [11], [5], an overlay (P2P) architecture that provides client anonymity for Internet applications[1]. The Tor project has been very successful and it is widely used by the military, law enforcement officers, journalists, etc.

We envision our system as the fundamental building block for implementing private query processing techniques on indexed data. Therefore, pCloud assumes a disk-based architecture that retrieves one *page* with a single query. In particular, we utilize a *striping* technique to disseminate the database to a number of cooperative peers, which allows the parallel processing of cPIR queries. Moreover, since the query result is produced at numerous, possibly untrustworthy peers, we incorporate an *authentication mechanism* that enables the client to verify that the retrieved page originated at the server. We implemented pCloud on the PlanetLab network [3], and experimented extensively with several system parameters. Our results indicate that pCloud reduces considerably the query response time compared to the traditional client/server model, and has a very low communication overhead. Additionally, it scales well with an increasing number of peers, achieving a linear speed-up. In summary, our contributions are the following.

- We implement a well known cPIR protocol [14], evaluate its cost in terms of computational time and bandwidth consumption, and identify its performance challenges when retrieving large data blocks. Based on our study, we propose a striping technique, which enables the protocol to retrieve arbitrarily large information without compromising computational cost at the server.
- We introduce pCloud that adapts the above technique in a distributed setting. We design an efficient data placement policy, a fast result retrieval method, and a query authentication mechanism. Furthermore, we show how pCloud deals with data updates as well as random node failures.
- We experiment with pCloud on the PlanetLab network and present detailed results from actual queries over a real network. We show that, despite previous concerns, cPIR can be useful in practice.

The remainder of this paper is organized as follows. Section 2 reviews the related work on private information retrieval. Section 3 presents the cPIR protocol utilized in our system, and optimizes its performance through an actual implementation. Section 4 describes in detail the pCloud architecture, and Section 5 presents the results of our PlanetLab experiments. Finally, Section 6 concludes the paper.

2 Related Work

Section 2.1 surveys the literature on PIR and clarifies our contribution, while Section 2.2 describes the use of PIR in database applications.

2.1 PIR protocols

The existing PIR protocols can be categorized into *information-theoretic*, *computational*, and *secure hardware*. Note that we do not delve into the cryptographic primitives that form the basis of several protocols presented here. The interested reader is referred to [12], [22] for more detailed surveys on PIR.

Information-theoretic PIR. The methods of this category ensure that the query discloses no information about the retrieved bit, even if the server has unbounded computational power. Chor et al. [9] prove that, for a single server, all n bits of the database need to be transmitted to the client. They also show that in order to achieve information-theoretic PIR with sublinear communication cost, the database must be replicated into k *non-colluding* servers, and the client must query independently each of these servers. Information-theoretic protocols have been studied extensively in the literature, resulting in several communication-efficient methods [9], [6], [7], [29]. However, in all these protocols, if the servers cooperate with each other, query privacy is violated.

Computational PIR. This class (denoted by cPIR) does not rely on assumptions about non-colluding servers. Instead, it is based on a *single-server* architecture, and employs well-known cryptographic primitives that guarantee query privacy in the presence of a computationally bounded server. The first cPIR protocol [18] relies on the *quadratic residuosity* assumption, which states that it is computationally hard to distinguish the quadratic residues in modulo arithmetic of a large (typically 1024-bit) composite modulus. Based on the above assumption, one may construct a cPIR protocol with a communication complexity $O(n^{\epsilon})$, where ϵ is an arbitrarily small positive constant. Nevertheless, their basic scheme incurs a very large communication cost, and requires the transmission of $O(\sqrt{n})$ 1024-bit integers between the server and the client.

1. Note that PIR is orthogonal to anonymity. Anonymity hides the identity of the client, while PIR hides the content of the query.

Cachin et al. [8] introduce the first single-server protocol with polylogarithmic communication complexity. The scheme builds upon the *ϕ-hiding* assumption: it is hard to distinguish which of two primes divides $\phi(m)$ for a hard-to-factor composite modulus m. The communication complexity of the protocol is $O(\log^a m)$, where a depends on the desired security (a typical value is $a = 8$). The above asymptotic complexity is improved by Lipmaa [19] who introduces a $O(\log^2 n)$ protocol that takes advantage of length-flexible additively homomorphic public-key cryptosystems. An additional benefit of Lipmaa's work is that it allows the client to retrieve an ℓ-bit block with a single answer, instead of a single bit that is common in most cPIR protocols. Another protocol that retrieves ℓ-bit blocks is by Gentry and Ramzan [14], which is also based on the ϕ-hiding assumption. However, for a particular instantiation, this scheme incurs only $O(\ell)$ communication cost, which is independent of the database size.

The main limitation of the aforementioned cPIR protocols is their high computational cost because they require (for a single query) $\Omega(n)$ modular multiplications over a large modulus. Consequently, researchers investigated different cryptographic primitives that utilize cheaper arithmetic operations. Gasarch and Yerukhimovich [13] propose two protocols with $O(n^\epsilon)$ communication complexity, based on the worst-case hardness of certain lattice problems. The improvement is due to the fact that they perform modular additions instead of multiplications. However, there is a non-zero probability of error in retrieving the queried bit. Melchor and Gaborit [20] introduce another lattice-based scheme with $O(\sqrt{n})$ communication complexity, which improves on the computational cost of [13] by performing additions on much smaller integers. Nevertheless, its large communication cost makes it impractical for a distributed setting.

Secure hardware PIR. PIR has also been addressed from a hardware perspective [16], [27], [28]. In hardware-based PIR schemes, the server utilizes a *secure coprocessor* (SC), which acts as a proxy between the client and the server. In this environment, PIR queries are processed as follows: (i) the client sends to the SC an encrypted version of the index of the desired bit (using public-key cryptography), (ii) the SC processes the query on the raw data and extracts the result, without revealing what is actually being retrieved, and (iii) the SC sends back to the client an encrypted version of the result, which is unreadable by the server. These methods have an optimal communication cost, since the client and the server only transmit encrypted versions of the query and the result, respectively. Furthermore, they are typically much faster than cPIR protocols. Nevertheless, their performance superiority comes at a certain cost: the requirement for a trusted third-party (i.e., the manufacturer of the SC). In other words, the client has to rely on the SC that it will not leak any sensitive information to the server.

Our contribution. We target at the scalability problem of cPIR. We specifically focus on cPIR because (i) information-theoretic protocols impose a stringent constraint of non-collusion among the participating servers, which renders them impractical for real-world applications, and (ii) in secure hardware PIR the querier must trust the manufacturer of the SC. On the other hand, cPIR does not entail the above limitations. Our goal is not to introduce a new protocol, but rather develop a distributed implementation of an existing scheme, in order to reduce the response time of private block retrieval in databases of practical sizes. Among the existing cPIR methods, we use the protocol of [14] (henceforth referred to as GR-PIR) as the building-block of our system because it is the most communication efficient method to date and, thus, an excellent candidate for a distributed implementation.

2.2 PIR in Databases

In the database literature, there exist two schemes ([15], [17]) that employ PIR techniques in order to privately answer spatial database queries, such as nearest neighbor (NN) and range queries. Ghinita et al. [15] apply the original cPIR scheme of Kushilevitz and Ostrovsky [18] to support NN queries. They assume a centralized server, and devise several optimizations (e.g., data compression, multiple CPUs, etc.), in order to reduce the communication and computational cost of the protocol. Despite these optimizations, their methods still inherit the limitations of [18], resulting in high query response times and excessive bandwidth consumption.

SPIRAL [17] is a hardware-based spatial PIR protocol that leverages a SC at the server to provide query privacy. In particular, the SC uses a random permutation to shuffle the database, and stores an encrypted version of the permuted database back at the server. Clients resolve their queries through the SC, which only accesses the encrypted version of the data. Query privacy is assured because the server is oblivious to the permutation, and does not have access to the decryption key. Similar to all hardware-based schemes, SPIRAL requires a trusted third-party and is, therefore, orthogonal to our work.

3 GR-PIR for Large Block Retrieval

In Section 3.1 we describe the GR-PIR protocol [14], providing the necessary mathematical background and security considerations. In Section 3.2 we identify a serious implementation challenge, concerning the retrieval of blocks larger than 32 bytes. To tackle this challenge, we introduce a *striping technique* that allows GR-PIR to efficiently retrieve a database page of arbitrary length, while maintaining the computational cost at the server constant. Although general, the resulting solution is ideal for parallelization and, thus, suitable for pCloud. Table 1 provides the most important terminology used throughout the paper.

TABLE 1
Summary of symbols

Symbol	Description
$\mathcal{DB}$	Database stored at the server
n	Size of $\mathcal{DB}$ (bytes)
$\mathcal{B}_j$	Block j of $\mathcal{DB}$
ℓ	Size of each block (bytes)
t	Number of $\mathcal{DB}$ blocks
p_j ($\pi_j = p_j^{c_j}$)	The prime (prime power) associated with $\mathcal{B}_j$
$m = \mathcal{P}_1 \cdot \mathcal{P}_2$	A composite modulus of primes $\mathcal{P}_1$, $\mathcal{P}_2$
$\langle g \rangle$	Cyclic group generated by $g \in \mathbb{Z}_m^*$
D	Page size (bytes)
k	Number of $\mathcal{DB}$ partitions
t'	Number of blocks in each $\mathcal{DB}$ partition

3.1 Protocol

Henceforth, for ease of presentation, we deviate from our earlier notation and refer to all units of information in terms of bytes (instead of bits). Let $\mathcal{DB}$ denote the database stored at the server, and n its size in bytes. GR-PIR consists of four phases, outlined in Figure 2. During the first phase (Figure 2, line 1), the server segments $\mathcal{DB}$ into t blocks, each of size ℓ bytes. Every block $\mathcal{B}_j$ ($j = 1, \ldots, t$) is associated with a prime power $\pi_j = p_j^{c_j}$, where $p_j \leq 2^{8 \cdot \ell}$ is a *small* prime, and $c_j = \lceil 8 \cdot \ell / \log p_j \rceil$. Observe that π_j has at least the same size (ℓ) as $\mathcal{B}_j$. Both p_j and π_j constitute public knowledge. The server expresses each $\mathcal{B}_j$ as a number in $[0, 2^{8 \cdot \ell} - 1]$, after appending zeros as needed in $\mathcal{B}_t$. Subsequently, it calculates e as the smallest positive integer that satisfies $e \equiv \mathcal{B}_j \pmod{\pi_j}$, for all $j = 1, \ldots, t$, using the Chinese Remainder Theorem (CRT). Note that e is a solution of the CRT modulo $\prod_{j=1}^{t} \pi_j$ and, thus, its size is approximately $t \cdot \ell = n$. The server *pre-computes* e prior to receiving any queries for the current instance of $\mathcal{DB}$ (e is query-independent), and stores it locally.

GR-PIR

Setup (server)
1. e = *CRT*($\mathcal{DB}$)

Query Generation (client)
2. ($\mathcal{P}_1$, $\mathcal{P}_2$) = *getPrimes*(π_i), $m = \mathcal{P}_1 \cdot \mathcal{P}_2$
3. (g, $|\langle g \rangle|$) = *getGenerator*(m), store $h = g^{|\langle g \rangle|/\pi_i}$
4. Send (m, g) to the server

Reply Generation (server)
5. Compute $g_e = g^e$ and return it to the client

Answer Extraction (client)
6. $\mathcal{B}_i$=*Pohlig-Hellman*(h, $g_e^{|\langle g \rangle|/\pi_i}$)

Fig. 2. Outline of GR-PIR

In the second phase of the protocol, the client generates and transmits its query to the server. Suppose that the client wishes to privately retrieve data residing in block $\mathcal{B}_i$. It first computes two equal-length prime numbers $\mathcal{P}_1$ and $\mathcal{P}_2$ (line 2), such that $\mathcal{P}_1 = 2q_1\pi_i + 1$ and $\mathcal{P}_2 = 2q_2 d + 1$, where q_1 and q_2 are random primes, and d is a random number. Subsequently, it computes $m = \mathcal{P}_1 \cdot \mathcal{P}_2$, and instantiates group $\mathbb{Z}_m^* = \{x \in \mathbb{Z}_m | gcd(m, x) = 1\}$. Next (line 3), it draws a random element $g \in \mathbb{Z}_m^*$, such that it generates a cyclic group $\langle g \rangle$ with order $|\langle g \rangle| = q \cdot \pi_i$, where q is an integer (i.e., g's order is divisible by π_i). It also keeps $q = |\langle g \rangle|/\pi_i$ secret, and stores $h = g^q$ for future use. Henceforth, it is implied that all the exponentiations of g are performed within the group $\langle g \rangle$ (i.e., modulo m). The client sends query $Q = (m, g)$ to the server (line 4), and the query generation phase concludes.

During the third phase (line 5), the server evaluates $g_e = g^e$, and sends the result back to the client. According to the CRT, e can be written as $e = \mathcal{B}_i + \pi_i \cdot E$, for some $E \in \mathbb{Z}$. Observe that

$$g_e^q = g^{e|\langle g \rangle|/\pi_i} = g^{\mathcal{B}_i|\langle g \rangle|/\pi_i} g^{E|\langle g \rangle|} = g^{\mathcal{B}_i|\langle g \rangle|/\pi_i} = h^{\mathcal{B}_i}$$

Therefore, in the last phase of the protocol (line 6), the client can retrieve $\mathcal{B}_i$ by computing $\log_h g_e^q$, using the Pohlig-Hellman algorithm [21]. The latter efficiently calculates the discrete logarithm within the subgroup $H \subset \langle g \rangle$ of order $\pi_i = p_i^{c_i}$, when p_i is small. Figure 3 sums up the interaction between the client and the server in GR-PIR.

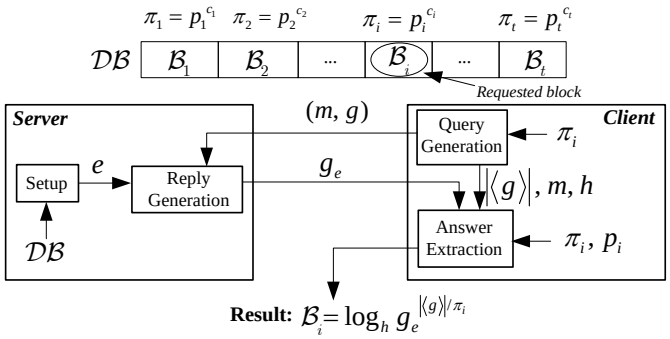


Fig. 3. Client/server interaction

Given $\langle g \rangle$, two prime powers π_1 and π_2, and a promise that one of π_1 or π_2, denoted as π_b, divides $|\langle g \rangle|$, determining π_b (and, thus, revealing the client's query) is as difficult as factoring m. With the above assumptions, an adversary can utilize two well-known attacks in order to factor m: (i) *number field sieve* [21], which is the most efficient algorithm for factoring a large integer, and (ii) Coppersmith's lattice-based attack [10]. To safeguard against the first attack, and following the current security standards, we must set the minimum length of m to 128 bytes. Concerning the second attack, we must adjust the length of m (i.e., $\log m$) to more than *four times* the block size ℓ. Given that both requirements are satisfied, it is *computationally intractable* to breach GR-PIR. In addition, according to [14], the smallest prime p_1 should be at least $2t$.

Finally, note that if the client substitutes d with a value $\pi_j \neq \pi_i$ in the computation of $\mathcal{P}_2$, i.e., such that $\mathcal{P}_2 = 2q_2\pi_j + 1$, then it can easily generate a group $\langle g \rangle$ whose order is divisible by *both* π_i and π_j.

Consequently, according to [14], the client can extract both blocks $\mathcal{B}_i$ and $\mathcal{B}_j$ from the single reply g_e of the server, by running the Pohlig-Hellman algorithm twice. In this way, the protocol can extract two blocks with a single query, achieving the same communication cost and computational cost at the server as a single block retrieval.

3.2 Striping Technique

Our objective is to utilize GR-PIR in order to build a simple interface for efficiently retrieving *pages* of arbitrary size[2]. We intend our solution as a *black-box* for developing private query processing mechanisms on indexed data. In particular, we aim at providing a simple primitive $pGet(\mathcal{DB}, i)$ that privately retrieves the i-th page from a database $\mathcal{DB}$. However, as we show next, the direct adaptation of GR-PIR to database applications is computationally prohibitive. The reason is that the block size ℓ gravely impacts the performance of the protocol, even for values as small as 64 bytes. Subsequently, we propose a solution to this problem.

We developed GR-PIR in C++ using the GMP [4] and LiDIA [1] libraries, which support efficient number theoretic computations on very large integers. We deployed our code on a machine with Intel Core Duo 2.53GHz CPU and 4GB of RAM, running Linux Fedora Core 9. We experimented by varying ℓ between 32 and 128 bytes. Due to Coppersmith's attack we set $\log m = 4 \cdot \ell$. Figure 4 illustrates the effect of ℓ on the performance of the scheme, for a $\mathcal{DB}$ with size 128KB. We display the experimental results averaged over 30 runs. Note that the communication cost between the client and the server is always $12 \cdot \ell$ because each of the m, g, and g_e (exchanged during the query) has length equal to $4 \cdot \ell$. Also, due to the number field sieve attack, the length of the modulus m cannot be smaller than 128 bytes and, thus, the communication and computational costs are equivalent for all values $\ell \leq 32$ (i.e., it is impractical to set $\ell < 32$).

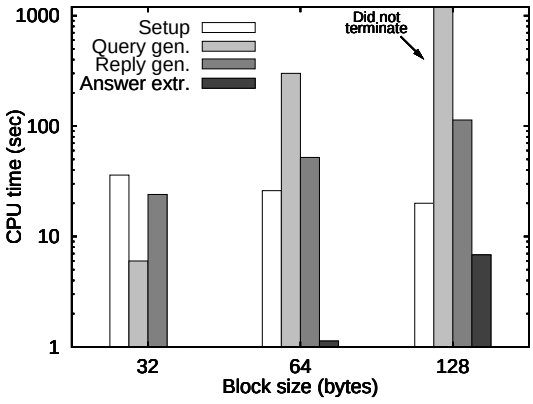


Fig. 4. Effect of ℓ on the performance of GR-PIR

2. Note that the page size is fixed for a particular database.

As ℓ changes from 32 to 64 bytes, the query generation cost increases 50 times (300 vs. 6 seconds), whereas the algorithm does not terminate (within reasonable time) when $\ell = 128$ bytes. This is due to the expensive computations of random primes that are required in function *getPrimes* (Figure 2, line 2). Specifically, the primes to be found ($\mathcal{P}_1$, $\mathcal{P}_2$) are of length $\frac{\log m}{2}$. Due to the prime number theorem [21], the probability that a random number of length $\frac{\log m}{2}$ is prime is approximately

$$\frac{1}{\ln 2^{\frac{\log m}{2}}} \approx \frac{1}{\log 2^{\frac{\log m}{2}}} = \frac{1}{\frac{\log m}{2}} = \frac{1}{2 \cdot \ell}$$

In addition, the Miller-Rabin primality test algorithm [21] (used in our implementation) involves $O(\ell)$ modular multiplications. Therefore, the complexity of generating a prime of length $\frac{\log m}{2}$ is quadratic in ℓ, which justifies our experimental results.

Another observation in Figure 4 is that the modulus size has a negative impact on the computational cost of several number theoretic operations. In particular, our experiments show that when we double ℓ, the reply generation time at the server approximately doubles as well. This indicates that a large modulus imposes a considerable overhead on the entailed modular exponentiation. The performance degradation is even larger for the answer extraction algorithm. Specifically, the CPU time increases almost tenfold when ℓ changes from 32 to 64 bytes (1.13 vs. 0.15 seconds), and 46 times when ℓ quadruples (6.82 vs. 0.15 seconds). The only advantage of having a larger block size is that the database setup phase is expedited, due to the fact that the CRT algorithm processes fewer blocks. Recall, however, that the CRT computation is an offline task and does not affect query processing.

The above remarks suggest that, in order for the client to retrieve a page of D bytes, it is always more beneficial to issue D/ℓ independent queries, with ℓ fixed to 32 bytes (instead of posing a single query with $\ell = D$). Consequently, in the remainder of this paper we assume that ℓ is *fixed* to 32 bytes. Additionally, as a further optimization, we can avoid the cost of the query generation phase through *query materialization*. That is, for each π_j value the client can compute *offline* a pair (m, g) and store it locally prior to query processing. The π_j's are database-independent, so the same queries may be used in multiple databases. Nevertheless, to prevent access pattern attacks, the client should generate a new random query (offline) whenever it issues a pre-computed one. Note that, even with the above optimizations, GR-PIR still results in an excessive computational cost for all parties involved, when retrieving any practical page size D. Re-visiting our example, in order to privately retrieve a 2048-byte page, the client has to issue 64 queries (since this page consists of 64 32-byte blocks), which translates to a response time of 1610 seconds.

To tackle this challenge, we apply a *striping* technique on the database (similar to RAID disks [23]), illustrated in Figure 5. Assume that the page size is D. Initially,

$\mathcal{DB}$ is divided into t' successive *stripes*, each consisting of exactly $k = D/\ell$ blocks of size ℓ bytes. In this way, every stripe is exactly the size of one page, and $t' = t/k$. Next, the blocks inside each stripe are assigned to the k partitions in a round robin manner. Stripe j is associated with a prime power π_j, defined similarly to Section 3.1. All π_j values (as well as their corresponding base p_j primes) constitute public knowledge. Following the striping process, each partition is considered as a separate database, consisting of t' ℓ-byte blocks. The server computes a value e_j for every partition j, using the CRT, and stores it locally. Note that the same set of π values are used for all partitions.

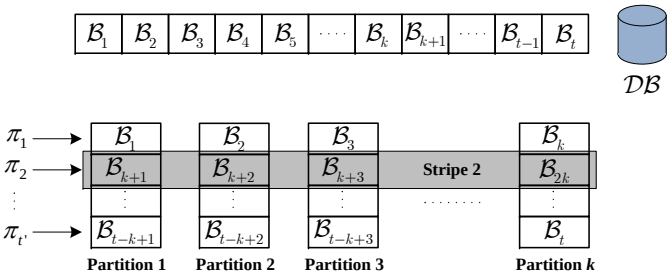


Fig. 5. The striping technique

Suppose that the client wishes to retrieve the i-th page of $\mathcal{DB}$. Initially, it forms a *single* query based on π_i (as explained in Section 3.1), and transmits it to the server. The latter processes the query on *every* partition j (i.e., the corresponding e_j's), and returns k replies to the client. Upon receiving the replies, the client invokes the Pohlig-Hellman algorithm for each one of them, in order to re-construct the underlying block. Eventually, it combines the extracted answers and retrieves the desired page.

As verified by our experiments, using the striping technique, the computational cost of retrieving a page of arbitrary length D is identical to the cost of returning a 32-byte answer in the original GR-PIR scheme. In particular, the cost of the modular exponentiation in the reply generation is linear in the length of the exponent e_j (based on the sliding window exponentiation algorithm utilized in the GMP library) and independent of the stripe size (which in this case is equal to D). Consequently, the query response time in our previous example plummets from 1610 to 35.6 seconds, when striping is used. Moreover, the database preparation at the server (setup) is faster with our approach (16.64 vs. 36 seconds in the original scheme). Finally, the communication cost between the client and the server is 256 bytes for the query (two 128-byte numbers), plus $4 \cdot D$ bytes for the replies (k replies, each consisting of $4 \cdot 32$ bytes). Observe that our technique incurs an identical communication overhead, compared to the case where D bytes are retrieved from GR-PIR without striping.

Recall that in the end of Section 3.1 we remarked that GR-PIR can extract two blocks with one query by making the order of $\langle g \rangle$ divisible by two π values. This modification can also be applied in combination with our striping technique, in order to allow the retrieval of two stripes (i.e., database pages) with a single query. In the sequel, for simplicity, we assume that the client is interested in a single database page.

4 System Architecture

In this section we present in detail the pCloud architecture. The goal of our system is to leverage the computational resources of cooperative peers, in order to expedite the processing of cPIR queries. Towards this end, pCloud organizes the peers in an overlay network, partitions the database into disjoint data segments, and disseminates the individual segments to the peers, in order to allow efficient query execution in parallel.

Before embarking on the description of our system, we first explain the *threat model*. We aim at query privacy, i.e., we seek to guarantee that no party other than the querier can infer the requested information. pCloud is built-upon GR-PIR and, thus, it inherits its threat model. Specifically, we assume that the adversary is computationally bounded. Moreover, our scheme is secure against collusion among any number of peers and the server. Finally, pCloud integrates a simple authentication mechanism to protect against malicious peers, i.e., the querier can verify the integrity of the extracted data.

In Section 4.1 we describe the network topology and data placement policy of pCloud, while in Section 4.2 we illustrate the query processing mechanism. Finally, Section 4.3 describes the operation of pCloud in dynamic environments.

4.1 Topology and Database Partitioning

We implement a *two-tier* architecture that slightly differs from existing P2P systems. The first tier consists of the participating peers, which form an unstructured Gnutella-like network overlay [2], where each node is connected to a number of random neighbors (typically 5). The motivation behind this choice will become clear in Section 4.2. The peers hold disjoint partitions of the database $\mathcal{DB}$ and are ready to answer queries upon request. The second tier is the database server, which holds the most current view of the entire $\mathcal{DB}$. The inclusion of the server is necessary in our setting for the following reasons: (i) the PIR query needs to process every bit of the database. Suppose that the client does not receive replies for all the different database partitions. This suggests that at least one partition has not been processed, because either it did not exist in the network, or the node that accommodated it failed. If an adversary has access to complete ISP logs (i.e., it can monitor all the packets that are transmitted inside the network), it can determine the non-processed partitions and, thus, infer information about the query. (ii) The server is the only entity that receives the data updates and, therefore, holds the most up-to-date $\mathcal{DB}$ instance at all times. Note that, although a centralized entity exists, query processing

inside the P2P network is completely independent (e.g., the server does not maintain an index to guide query propagation).

The actual challenge lies in how to partition and distribute $\mathcal{DB}$ to the peers, so that query processing experiences a linear performance speed-up with respect to the number of partitions. Towards this end, pCloud exploits the features of the striping technique presented in Section 3.2, namely the ability to privately extract an arbitrarily large (yet a priori fixed) amount of information, while retaining the query processing cost linear to the partition size. In other words, if we simply subdivide $\mathcal{DB}$ into k partitions, as demonstrated in Section 3.2, and disseminate them among the peers, the query execution cost will be k times lower, compared to the traditional client/server model. The above statement is true, provided that every partition is accommodated by at least one peer, and is processed during the query execution. The number of partitions k constitutes public knowledge.

Clearly, the size of the stripe (i.e., k) adjusts a trade-off between the computational cost at the peers and the communication cost at the client. On one hand, a large stripe size implies smaller partitions, i.e., better computational cost at the peers. On the other hand, if the stripe is large, the client receives many redundant results, since it is only interested in a single page within the stripe. We expect that, in a real system, the size of the stripe will be determined explicitly by the database size (assuming there are enough peers to handle the load). Ideally, we would like to impose a low computational cost at the individual peers, and at the same time maintain a reasonable communication overhead at the client.

To avoid the situation where a page is split between two consecutive stripes (which would require two expensive PIR queries for that page), we may adjust the number of partitions, so that the stripe size is always a multiple of the page size. In this setting, PIR queries are constructed as follows: Suppose that a client wants to retrieve a database page starting at block $\mathcal{B}_i$. The initial step is for the client to determine the id of the stripe (let *sid*) that holds the required page; this is equal to *sid* = $\lceil i/k \rceil$. Next, the client constructs a query corresponding to π_{sid}, according to the methodology described in Section 3 (Figure 2). In the following section we discuss in detail how the query is resolved inside the P2P network.

Observe that the aforementioned partitioning process resembles the database outsourcing model, where each peer can be thought of as a (possibly untrustworthy) service provider. Consequently, it is essential to incorporate an *authentication mechanism* so that the client can verify the *correctness* of the result (i.e., that the result originated at the server and is not falsified). We choose a solution based on public-key digital signatures, due to its simplicity and good performance. In this setting, the server obtains a *private* and a *public* key through a trusted key distribution center. The private key is known only to the server, while the public key is accessible by all the clients. Using its private key, the server digitally signs the data by generating one signature (typically 128 bytes) *for every page* of the database. Then, the signature corresponding to a page is appended immediately after the end of that page in the database (i.e., signatures are interleaved with pages within a stripe). The authentication mechanism increases in the total size of partitions. Nevertheless, this overhead is not significant; for instance, authenticating 2KB pages increases the database size by only 6.25%, which is amortized over a large number of peers.

Primitive *pGet*($\mathcal{DB}$, i) is modified in order to extract the signature following the requested page in the stripe, i.e., if D is the page size, *pGet* must retrieve $(D + 128)/32$ blocks from the appropriate stripe. Subsequently, the client verifies the correctness of the page using the signature and the public key of the server. Note that in the presence of updates, it is possible for peers to collude and present to the client an outdated version of a page. To safeguard against this situation, each signature incorporates the timestamp of the last update occurred in the database. As soon as the client extracts the signature, it contacts the server and requests the last update timestamp. Only if it matches the one in the signature does the client proceed with the answer extraction.

4.2 Query Processing

Similar to most existing P2P systems, query processing in pCloud involves two distinct phases: query and result propagation. Regarding the first phase, our choice is a *flooding protocol*, since the client needs to retrieve a reply *for every unique partition* that resides inside the network (to achieve optimal parallelization). The most efficient topology for query flooding is a tree structure, but these topologies are very difficult to maintain in a dynamic environment because they are not resilient to node[3] failures. Additionally, structured topologies, such as Distributed Hash Tables (e.g., Chord [26]) are also not applicable in our setting because (i) they mostly support queries for specific items, and (ii) their performance degrades drastically in the presence of frequent node failures. The above reasons led us to use a simple Gnutella-like network overlay, as described in the previous subsection. This topology is easy to maintain and is robust against random node failures, because it offers multiple paths between any pair of nodes.

During query propagation, every node forwards the query to all its neighbors, until the query's lifetime expires (Figure 6, step 1). Specifically, every query has a Time-to-Live (TTL) parameter, which indicates the maximum number of hops that it is allowed to travel. In our case, the TTL value should be large enough so that the query reaches a number of nodes that is larger than the number of partitions k. The query Q is a tuple $\langle Q.id, Q.TTL, Q.IP, Q.\mathcal{DB}, Q.m, Q.g \rangle$, where $Q.id$ is a randomly generated id, $Q.TTL$ is the TTL

3. We use the terms node and peer interchangeably.

value, $Q.IP$ is the IP address of the client, $Q.\mathcal{DB}$ is a description of the queried database, and $Q.m$ and $Q.g$ are the modulus and group generator, respectively, that are constructed based on the index of the required page.

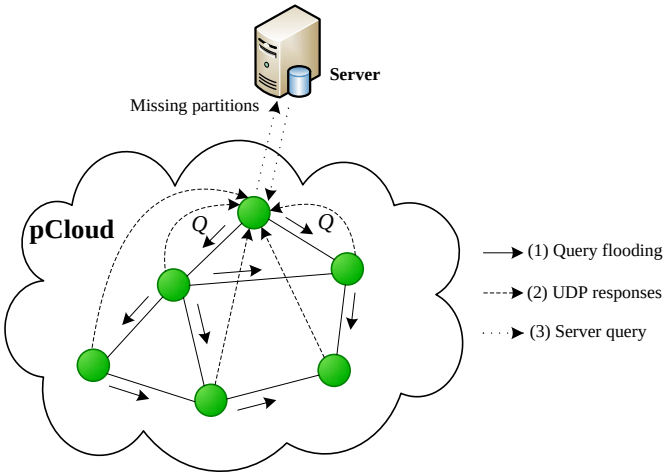


Fig. 6. Query processing

Whenever a peer receives a query message, it runs the reply generation algorithm of GR-PIR (Figure 2, line 5) on its own partition. Once the result is computed, the peer transmits it back to the client along with the id of the partition. In the original Gnutella protocol, query hits are transmitted on the reverse of the path that is created during the flooding process. However, this is not a practical approach in our setting because (i) a single node failure may potentially cause multiple result losses, and (ii) it increases considerably the communication cost. Consequently, we follow the mechanism of the new Gnutella specification, which states that query hits are transmitted directly from the peer to the client, via the UDP protocol. UDP is an unreliable, connectionless service that is well-suited for transmitting a single packet between two nodes. Unlike TCP, which requires an expensive three-way handsake to establish a connection, UDP involves only the direct transmission of data packets. Although UDP is not reliable for transferring large amounts of data (because packet losses are not detected), in our method a query result is a 128-byte integer that can fit in a single packet. Therefore, UDP is suitable for transmitting the results back to the client.

Note that, in some cases (e.g., due to node failures), the client may not be able to retrieve the results from all k partitions for the specified $Q.TTL$. Therefore, after the client forwards the query to its neighbors, it starts a timer T_{net} that indicates the maximum amount of time that it is willing to wait, in order to retrieve all existing results from the network. The value of the timer should reflect (i) the average CPU time required at each peer for generating the result (T_{cpu}), and (ii) the worst-case round-trip latency inside the network (T_{lat}). Thus, a good estimate for the timer would be $T_{net} = T_{cpu} + T_{lat}$.

As soon as the client receives a new result from the network that is part of the queried page, it immediately extracts the corresponding block, using Pohlig-Hellman (Figure 2, line 6). If the timer T_{net} expires and there are still some missing results, the client creates a list with the corresponding ids, and sends a query to the server in order to retrieve them (Figure 6, step 3). This is a very expensive procedure because the replies are generated sequentially at the server. However, if the missing results do not concern the client (i.e., they are not part of the queried page), the response time is not affected. In any case, the client has to query the server for all missing results, in order to guarantee privacy against an adversary with traffic monitoring capabilities. Note that the client/server communication utilizes a reliable TCP connection.

A last remark concerns the *bootstrapping* mechanism of pCloud. When a peer joins the network, it contacts the database server and receives a list of recently seen IP addresses. Next, it contacts some of these peers (using the ping/pong messages of Gnutella), in order to discover other nodes in the network that can establish a new TCP connection. Eventually, it joins the overlay topology and starts participating in query propagation. The server also transmits to the new peer a database partition and, after the peer completes the setup (Figure 2, line 1), it can start evaluating received queries. To improve the performance, we utilize an informed method for allocating the partitions to the incoming peers. Specifically, the server maintains a list of the recently requested partitions, which is a good estimate of the partitions that are missing inside the network. From that list, it transmits the most recent partition to the new peer. A further optimization is to also store the IP addresses of the nodes that requested these partitions, so that the new peer may try to connect around their neighborhood. The intuition is that, even in the case the partition exists somewhere in the network, it may be far away (in terms of the TTL value) from those nodes.

4.3 Data Updates and Node Failures

In this section we discuss two scenarios that may potentially affect the query response time in pCloud: data updates and node failures. Regarding data updates, one observation is that pCloud resembles a cooperative caching environment, i.e., an overlay network where peers do not share their own data but rather store information from other servers. Consequently, we may employ similar techniques (e.g., cache invalidation protocols) to deal with stale results. In particular, we assume the following data update process. When the server receives a number of updates, it modifies the corresponding pages, and produces a new set of signatures that incorporate the current timestamp. Subsequently, it broadcasts (using the flooding mechanism of query propagation) a list with the outdated (i.e., affected) partitions to all the peers.

If a peer receiving the server's message is the owner of an obsolete partition, it (i) suspends query processing

(i.e., it does not return any results), and (ii) contacts the server to receive the updates. Note that, to save communication resources, the server does not transmit the entire partition, but only sends the blocks that are affected by the updates. After the peer receives the updates, it invokes the setup algorithm that is required by GR-PIR, and eventually resumes query processing. On the other hand, if the peer is a client that is currently involved in query processing, it has to contact the server in order to retrieve the updated partitions. Note that, if updates are very frequent the client has to request many partitions from the server, which greatly increases the query response time. Therefore, pCloud is not recommended for applications involving frequent updates (e.g., data streaming).

The next issue regards random node failures, which is an inherent characteristic of most P2P networks. In pCloud, the query and result propagation mechanisms are very robust against node failures because (i) queries are flooded inside the network through multiple paths, and (ii) results are transmitted directly from each peer to the client without any intermediate hubs. However, the main limitation of pCloud is that if a node that holds the only copy of a certain partition fails, it affects the query response time because the missing partition has to be processed at the server. In the worst case that all nodes fail, pCloud reduces to a conventional client/server architecture.

To minimize the effect of node failures, we slightly modify pCloud to include *data replication*. Specifically, if the expected number of peers in the network is N, the server adjusts the total number of partitions k, in order to inject a certain percentage of replication $r = (N-k)/k$ inside the network (note that $N \geq k$). Consequently, depending on the level of replication, any partition may be stored at multiple peers. However, every node still maintains a single partition. Therefore, for a given N, increasing replication creates larger partitions, and the reply generation time at the peers may become considerable. Nevertheless, as we show in our experiments, this strategy actually pays off in terms of query response time, even for moderate failure rates. The reason is that replication increases the chances of finding all the partitions inside the network, thus avoiding the server entirely. In addition, although the individual computational time increases at each peer, query processing is still performed in parallel, leading to better response times.

5 Experiments

We developed GR-PIR enhanced with our striping technique in C++, utilizing the GMP [4] and LiDIA [1] libraries. Furthermore, we implemented the network components of pCloud in Python, and deployed our code on PlanetLab [3]. We installed our server program at an underutilized node with powerful hardware, in order to simulate the superior computational and bandwidth capabilities of the server. We generated a random connected graph as our network topology, with average node degree equal to 5. We compared pCloud against the traditional client/server model, which is hereafter referred to as CS. For fairness, the server in CS also employs our striping technique. Recall from Section 3.2 that the partitioning method does not have an impact on the computational time. Therefore, in CS we decompose the database so that the stripe size is equal to the page size; this yields optimal communication cost.

We investigate the following performance metrics: (i) the *query response time* at the querier[4], i.e., the time that elapses from the instance the query is posed, until the actual answer is extracted. From this cost we exclude the time to generate g and m (Figure 2, lines 2-3) because we assume that the client materializes all queries offline (following the strategy explained in Section 3.2). (ii) The *computational time* at the server and a participant peer (other than the querier), which is consumed by the reply generation algorithm. Note that we do not include the overhead of the database setup (phase 1 of GR-PIR), as this task is conducted offline. (iii) The *communication cost* at all the above parties, which takes into account both incoming and outgoing packets.

Table 2 illustrates our system parameters, with their default values appearing in bold face. In every experiment we test over the values of one parameter, while fixing the remaining ones to their default values. We perform 10 queries per experiment, and plot the average values for each metric. Each query privately retrieves a random database page. Section 5.1 presents the results for a static network/database, whereas Section 5.2 demonstrates our findings in a dynamic environment.

TABLE 2
System Parameters

Parameter	Range
Static Case	
Number of peers (N)	100, 150, **200**, 250, 300
Database size (n)	1, 5, **10**, 15, 20 (MB)
Page size (D)	512, 1024 **2048**, 4096 (bytes)
Dynamic Case	
Replication (r)	0, 25, **50**, 75, 100 (%)
Node failures (f)	0, 5, **10**, 15, 20 (%)

5.1 Static Case

In the absence of node failures, we assume that the server segments the database to *as many partitions as the number of peers*. We also suppose that each partition is located at exactly one peer. Finally, we set the TTL to a large enough value to ensure that the query reaches the entire network. Note that the above setting achieves the maximum possible parallelization for the specified number of peers. This is due to the fact that we did not experience any UDP packet losses and, thus, the querier never contacted the server. Therefore, in the remainder of

4. We henceforth use terms querier and client interchangeably.

this subsection, the computational/communication cost at the server always refers to the CS approach.

The first set of experiments evaluates the performance of pCloud against CS, when varying the number of peers N and assigning the default values to the database size n and page size D (i.e., 10MB and 2048 bytes, respectively). Our results are grouped in Figure 7. Observe that the maximum number of peers is 300, although there are approximately 1000 peers registered with PlanetLab. The reason is that PlanetLab is rather unstable, and a large fraction of peers are either down, or exhibit frequent problems (such as ssh failures, time drifting, excessive workload, etc.).

Figure 7(a) depicts the average response time at the client, whereas Figure 7(b) illustrates the computational cost at a participant peer and the server. In pCloud the response time is up to more than two orders of magnitude smaller than in CS. The reason is that in CS the server processes all partitions sequentially, leading to an excessive computational time. On the other hand, in pCloud the participating peers perform the reply generation algorithm on a very small portion of the database in parallel. This significantly reduces the computational requirements at each peer and, thus, the result collection at the querier.

Concerning the effect of N, larger N values cause the response time in pCloud to drop linearly because each peer performs the PIR query on a smaller partition (ranging from $\sim$ 109KB when $N = 100$, to $\sim$ 37KB when $N = 300$). Observe, however, that the response time does not decrease as fast as the computational cost at a peer. Note that the answer extraction algorithm requires 10.2 seconds in both CS and pCloud, as it is independent of the total number of peers (it is only affected by the page size). Although in CS this cost is negligible compared to the CPU time consumed at the server, in pCloud it emerges as a significant portion of the response time when the computational effort at a peer decreases (reaching 72% of the overhead when $N = 300$). Therefore, in Figure 7(a), the curve for pCloud would smoothly continue its decline for $N > 300$, eventually converging to 10.2 seconds (an unavoidable cost).

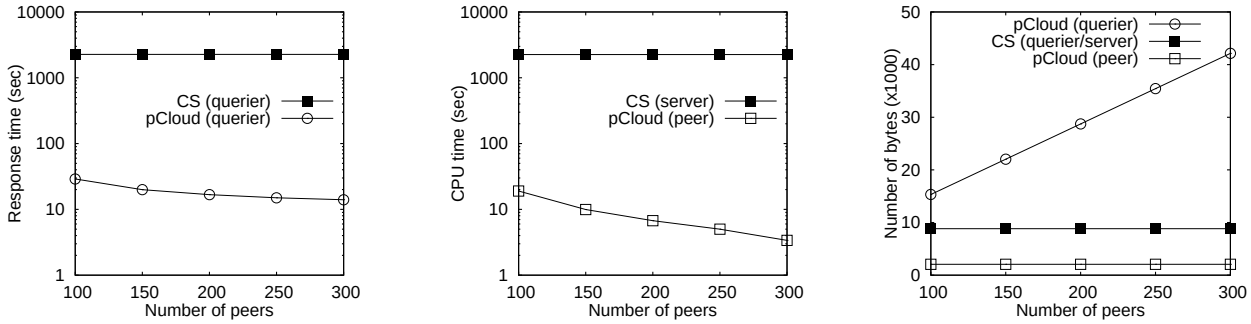
Figure 7(c) draws the average communication cost for all parties involved. As previously mentioned, in CS the bandwidth consumption is optimal, which is about four times the page size. This is because each reply has size fixed to (approximately) 128 bytes, but conveys only 32 bytes (i.e., one block) of useful information. On the other hand, in pCloud this cost is determined by the total number of replies, which increases with the number of peers. However, observe that even in the worst case the client receives less than 42KB of data, which constitutes a negligible fraction of the database (0.4%). The communication cost per query per peer entails the propagated queries and the UDP replies sent back to the client. Since we maintain the average number of neighbors constant in all our topologies, the bandwidth consumption is almost independent of N (with small, unobservable in the figure, fluctuations due to randomness). Moreover, this cost is only $\sim$2KB, i.e., 4% of the partition size.

The second set of experiments assesses the effect of the database size n on the performance of the schemes ($N = 200$, $D = 2048$). Figure 8(a) provides the response time at the server, and Figure 8(b) illustrates the reply generation time. pCloud achieves more than two orders of magnitude improvement in response time compared to CS (29 vs. 7306 seconds when $n = 20$MB). As expected, the overhead increases with the database size for both pCloud and CS, because more bits need to be processed. However, the response time in pCloud (Figure 8(a)) raises more slowly than in CS because a substantial fraction of the response time is consumed by the answer reconstruction algorithm. Note that the response times shown here are not representative of a real world deployment of our system, since we utilized a rather limited number of peers. Given that pCloud scales linearly with the number of peers, we expect the response time to be very low in typical overlay networks (containing tens of thousands of nodes), even for very large databases. Finally, the communication overhead in both schemes does not depend on the database size and is, thus, omitted.

We conclude the experimental evaluation of the static case by studying the behavior of pCloud and CS for variable page size ($N = 200$, $n = 10$MB). Figure 9(a) draws the response time at the querier, whereas Figure 9(b) shows the computational time at the involved parties. As we raise the page size, the average computational time at a peer slightly decreases. The reason is that, when the page size is larger, the authentication overhead (i.e., signatures) becomes smaller. Therefore, the database/partition size decreases, which results in a lower CPU time. In CS, the computational overhead at the server is constant because, as previously mentioned, the partitioning method (determined by the page size) does not influence the CPU consumption at the server.

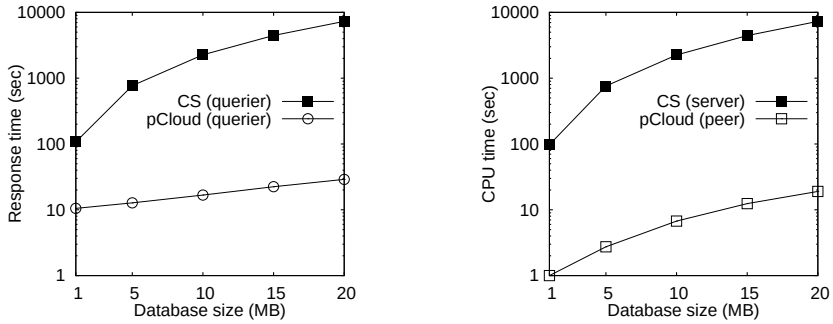
On the other hand, the query response time increases with the page size in both CS and pCloud. This is because a larger page size implies that a larger number of 32-byte blocks comprise the page. Consequently, the answer reconstruction algorithm must be executed on more replies to extract all the page blocks. In pCloud, this cost phases out the minor computational savings at the peers and, therefore, the response time curve gradually raises. The respective overhead in CS increases very slightly so that it cannot be noticed in the figure. The reason is that, even in the worst case when $D = 4096$ bytes, the answer extraction cost accounts for a negligible fraction of the response time (0.8%), which is mostly dominated by the query processing at the server.

Finally, Figure 9(c) depicts the bandwidth consumption. The cost at both the querier and a participant peer in pCloud are independent of the page size. However, the communication overhead in CS increases quickly for larger values of D, since the server needs to transmit a larger page back to the querier.



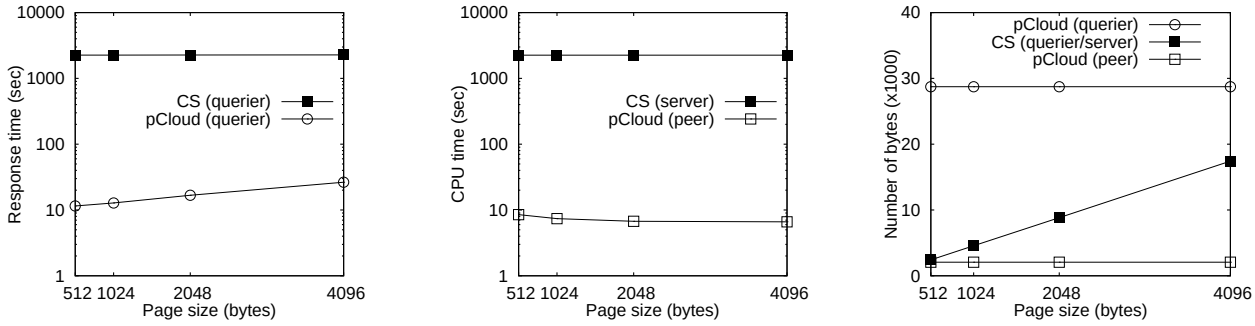
(a) Response time (b) Computational cost (c) Communication cost

Fig. 7. Varying the number of peers N



(a) Response time (b) Computational cost

Fig. 8. Varying the database size n



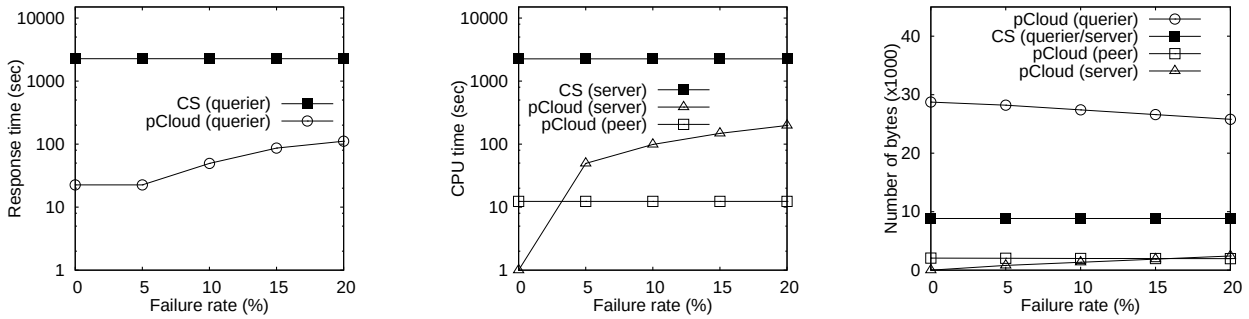
(a) Response time (b) Computational cost (c) Communication cost

Fig. 9. Varying the page size D

To sum up, pCloud benefits from the distributed setting significantly, improving response time by more than two orders of magnitude compared to the client/server model. The system scales linearly with the number of peers and, thus, enables efficient PIR even in the presence of large databases. This comes at a small communication overhead that constitutes a negligible percentage of the database. Additionally, the participating peers contribute low CPU and bandwidth resources per query, which strengthens the motivation for joining pCloud.

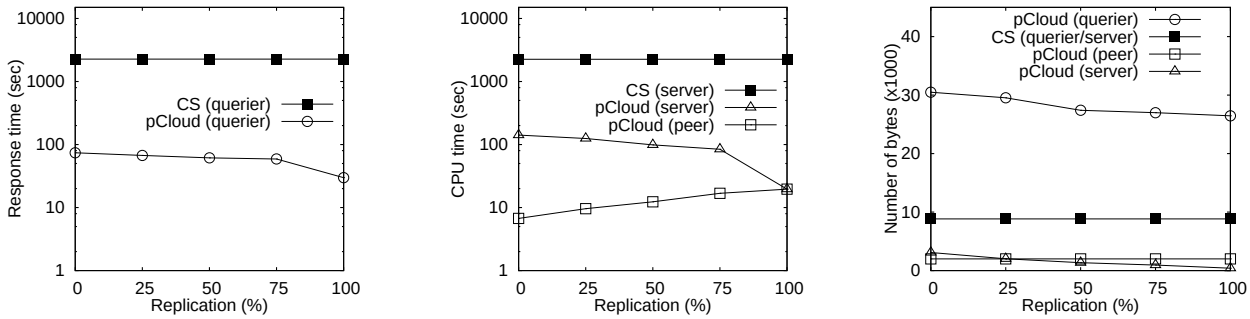
5.2 Dynamic Case

The dynamic case captures situations where peers in pCloud fail during query processing, or they store an obsolete partition instance (due to updates). Therefore, these peers cannot contribute to query execution. Since it is very difficult to simulate scenarios that involve updates, we experimented only with node failures. Note that this is the worst-case scenario because, contrary to an outdated peer, a failed node does not propagate the query. This may prevent other peers from receiving the query and, thus, take part in query processing. We performed two sets of experiments, varying two param-



(a) Response time (b) Computational cost (c) Communication cost

Fig. 10. Varying the node failure rate f



(a) Response time (b) Computational cost (c) Communication cost

Fig. 11. Varying the amount of replication r

eters: (i) the percentage of node failures f, and (ii) the amount of replication r, measured as the percentage of the partitions that are duplicated in the network. We also fixed the rest of the parameters to their default values ($N = 200$, $n = 10MB$, and $D = 2048$) and investigated the same metrics as in the static case. Although the performance of CS is not affected by these parameters, we include its corresponding costs to facilitate the comparison.

In the first set of experiments we vary the node failure rate f, while setting r to 50% (meaning that half of the partitions exist in the network twice). Figure 10(a) depicts the response time, and Figure 10(b) illustrates the computational cost at the parties involved in query processing. The computational cost at each peer is not affected by f because the partition size remains constant. On the other hand, as f increases, the respective cost at the server raises as well. This is because the probability that a partition is not located in the network becomes larger and, thus, the client must attend to the server to collect a reply for more partitions. Consequently, the client experiences a linearly larger response time.

However, observe in Figure 10(a) that the response time is smaller than the CPU time consumed at the server. This is due to the fact that not all of the replies received from the server are useful for the querier to extract its complete answer. For example, in the worst case where the failure rate is $f = 20\%$ the querier requests 16 blocks from the server, whereas only 3 are contained in the desired page. As soon as the client receives a reply concerning its query it immediately applies the answer extraction algorithm on it. This leads to concluding the query processing before retrieving the last reply from the server. Furthermore, note that for $f \leq 5\%$ (and with 50% replication) the querier does not need *any* of the replies returned by the server in order to finish its query, since there is at least one peer in the network possessing a block contained in the queried page. Therefore, the response time remains constant for the above values of f. Finally, note that even in the case the client requires a reply relevant to its query from the server, it executes the answer extraction algorithm for the already received replies from the network while waiting for the server's answer. This saves a considerable fraction of the 10.2 seconds consumed in total for all page blocks by this algorithm, since the largest part of the query is still answered by the network.

Figure 10(c) demonstrates the communication cost as a function of f. Since more partitions are likely to be retrieved from the server, the server bandwidth consumption increases with larger f. On the other hand, the communication cost at the querier is smaller for a larger number of node failures, since the probability of receiving duplicate replies decreases. Finally, the overhead at

a participant peer also slightly decreases (although this is not easily discernible in the figure), because it may propagate the query to fewer neighbors (i.e., if a failed node was its neighbor).

The second set of experiments in the dynamic case assesses the effect of replication on the performance of pCloud, when the node failures are fixed to 10%. Figure 11(a) plots the response time, and Figure 11(b) presents the CPU time at the entities that process the query. As expected, both the computational time at the server, as well as the response time, decrease with a larger r because the probability a partition is not located in the network diminishes. Nevertheless, observe that the curve of the response time in Figure 11(a) has a slightly smaller slope than that of the CPU time at the server in Figure 11(b). The reason is that, as we increase r, the partition size becomes larger. Consequently, the server and each participant peer must consume more time to process the query on a partition, a fact that balances out some of the computational savings at the server. This also indicates that there is a trade-off between the aforementioned costs, which should be carefully adjusted in order to achieve the optimal response time. Note that when $r = 100\%$, the response time in pCloud reaches 29.79 seconds, which is 76 times smaller than in CS. Comparing this value to the respective cost in the static case (16.73 seconds), we conclude that replication achieves a relatively similar performance to the static case.

Finally, Figure 11(c) plots the communication cost for the above experiment. The overhead at the server drops gradually for larger values of r, since the client requests fewer replies. Also, the communication cost at the querier becomes slightly lower for the same reasons explained for Figure 10(c). Finally, the bandwidth consumption at a participant peer is unaffected by r because it only depends on the number of neighbors (which is fixed, since the topology for a particular N and f is stable).

To conclude, node failures adversely affect the performance of pCloud. However, replication is a plausible solution that can tackle this challenge. In particular, we showed that with 100% replication, we achieve a response time close to the static case, while maintaining the communication cost almost identical. In other words, replication can render pCloud very resilient in the presence of node failures.

6 Conclusions

Private information retrieval (PIR) is an important field with several practical applications. However, despite the extensive cryptography literature, there is very limited work on databases due to the prohibitive cost of PIR on datasets with realistic sizes. In order to tackle this cost, we leverage the computational resources of a peer-to-peer cloud. Specifically, the proposed pCloud solution embeds a state-of-the-art PIR protocol in a distributed environment, by utilizing a novel striping technique. pCloud can retrieve arbitrarily large blocks of information with a single query. We present a comprehensive solution that includes a data placement policy, result retrieval and authentication mechanisms. We implement our system in PlanetLab and perform an extensive set of experiments, which confirm the effectiveness and practicality of our scheme. Specifically, compared to the traditional client/server architecture, pCloud drops the query response time by orders of magnitude, and its performance improves linearly with the number of peers. Finally, it is resilient to node failures and can handle updates. We hope that this work motivates further research on distributed PIR.

Acknowledgments

This work was supported by grant HKUST 618108 from Hong Kong RGC, and the NSF CAREER Award IIS-0845262.

References

[1] LiDIA - A C++ Library For Computational Number Theory. http://www.cdc.informatik.tu-darmstadt.de/TI/LiDIA/.
[2] Limewire. http://www.limewire.com/.
[3] PlanetLab. http://www.planet-lab.org/.
[4] The GNU MP Bignum Library. http://gmplib.org/.
[5] Tor: anonymity online. http://www.torproject.org/.
[6] A. Ambainis. Upper bound on communication complexity of private information retrieval. In *ICALP*, 1997.
[7] A. Beimel, Y. Ishai, E. Kushilevitz, and J.-F. Raymond. Breaking the $O(n^{\frac{1}{2k-1}})$ barrier for information-theoretic private information retrieval. In *FOCS*, 2002.
[8] C. Cachin, S. Micali, and M. Stadler. Computationally private information retrieval with polylogarithmic communication. In *EUROCRYPT*, 1999.
[9] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *FOCS*, 1995.
[10] D. Coppersmith. Finding a small root of a univariate modular exponentiation. In *EUROCRYPT*, 1996.
[11] R. Dingledine, N. Mathewson, and P. F. Syverson. Tor: The second-generation onion router. In *USENIX Security Symposium*, 2004.
[12] W. Gasarch. A survey on private information retrieval (column: Computational complexity). *Bulletin of the EATCS*, 82:72–107, 2004.
[13] W. Gasarch and A. Yerukhimovich. Computationally inexpensive cPIR. Unpublished Draft, 2006.
[14] C. Gentry and Z. Ramzan. Single-database private information retrieval with constant communication rate. In *ICALP*, 2005.
[15] G. Ghinita, P. Kalnis, A. Khoshgozaran, C. Shahabi, and K.-L. Tan. Private queries in location based services: Anonymizers are not necessary. In *SIGMOD*, 2008.
[16] A. Iliev and S. W. Smith. Private information storage with logarithm-space secure hardware. In *Proc. International Information Security Workshops*, 2004.
[17] A. Khoshgozaran, H. Shirani-Mehr, and C. Shahabi. SPIRAL: A scalable private information retrieval approach to location privacy. In *PALMS*, 2008.
[18] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *FOCS*, 1997.
[19] H. Lipmaa. An oblivious transfer protocol with log-squared communication. In *ISC*, 2005.
[20] C. A. Melchor and P. Gaborit. A fast private information retrieval protocol. In *ISIT*, 2008.
[21] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*. CRC Press, 1997.

[22] R. Ostrovsky and W. E. S. III. A survey of single-database private information retrieval: Techniques and applications. In *PKC*, 2007.
[23] D. A. Patterson, G. A. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *SIGMOD*, 1988.
[24] L. Sassaman, B. Cohen, and N. Mathewson. The pynchon gate: a secure method of pseudonymous mail retrieval. In *WPES*, 2005.
[25] R. Sion and B. Carbunar. On the computational practicality of private information retrieval. In *NDSS*, 2007.
[26] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, 2001.
[27] S. Wang, X. Ding, R. H. Deng, and F. Bao. Private information retrieval using trusted hardware. In *ESORICS*, 2006.
[28] P. Williams and R. Sion. Usable PIR. In *NDSS*, 2008.
[29] S. Yekhanin. Towards 3-query locally decodable codes of subexponential length. *Journal of the ACM*, 55(1), 2008.

Stavros Papadopoulos received the BS degree in Informatics from the Aristotle University of Thessaloniki, Greece, in 2000, and the PhD degree in Computer Science and Engineering from the Hong Kong University of Science and Technology in 2005. Currently, he is a postdoctoral researcher at the Department of Computer Science and Engineering of the Chinese University of Hong Kong. His research interests include authenticated query processing, private information retrieval, spatial databases and cloud computing.

Spiridon Bakiras received the BS degree in Electrical and Computer Engineering from the National Technical University of Athens in 1993, the MS degree in Telematics from the University of Surrey in 1994, and the PhD degree in Electrical Engineering from the University of Southern California in 2000. Currently, he is an assistant professor in the Department of Mathematics and Computer Science at John Jay College, City University of New York. Before that, he held teaching and research positions at the University of Hong Kong and the Hong Kong University of Science and Technology. His current research interests include database security and privacy, mobile computing, and spatiotemporal databases. He is a member of the ACM and the IEEE, and a recipient of the US National Science Foundation (NSF) CAREER award.

Dimitris Papadias is a Professor of Computer Science and Engineering, Hong Kong University of Science and Technology. Before joining HKUST in 1997, he worked and studied at the German National Research Center for Information Technology (GMD), the National Center for Geographic Information and Analysis (NCGIA, Maine), the University of California at San Diego, the Technical University of Vienna, the National Technical University of Athens, Queen's University (Canada), and University of Patras (Greece). He serves or has served in the editorial boards of the VLDB Journal, IEEE Transactions on Knowledge and Data Engineering, and Information Systems.