Single Pass Client Preprocessing Private Information Retrieval

Arthur Lazzaretti
Yale Univeristy

Charalampos Papamanthou
Yale University

Abstract

Recently, many works have considered Private Information Retrieval (PIR) with client-preprocessing: In this model a client and a server jointly run a preprocessing phase, after which client queries can run in time sublinear in the size of the database. In addition, such approaches store no additional bits per client at the server, allowing us to scale PIR to a large number of clients.

In this work, we propose the first client-preprocessing PIR scheme with "single pass" client-preprocessing. In particular, our scheme is concretely optimal with respect to preprocessing, in the sense that it requires exactly one linear pass over the database. This is in stark contrast with existing works, whose preprocessing is proportional to $\lambda \cdot N$, where λ is the security parameter (e.g., $\lambda = 128$). Our approach yields a preprocessing speedup of 45-100$\times$ and a query speedup of up to 20$\times$ when compared to previous state-of-the-art schemes (e.g., Checklist, USENIX 2021), making preprocessing PIR more attractive for a myriad of use cases that are "session-based".

In addition to fast preprocessing, our scheme features extremely fast updates (additions and edits)—in constant time. Previously, the best known approach for handling updates in client-preprocessing PIR had time complexity $O(\log N)$, while also adding a $\log N$ factor to the bandwidth. We implement our update algorithm and show concrete speedups of about 20$\times$ in update time when compared to the previous state-of-the-art updatable scheme (e.g., Checklist, USENIX 2021).

1 Introduction

Private Information Retrieval (PIR), as defined by Chor et al. [7] is a protocol between a client and a server where the server holds a public database DB of N bits and the client holds an index $i \in \{0, \ldots, N-1\}$. The goal of the protocol is for the client to learn $\mathsf{DB}[i]$ without revealing any information about i to the server. PIR has found many applications, such as in private contact tracing, oblivious ad serving, private movie streaming, among others [2, 3, 14, 16, 21].

A PIR protocol is non-trivial if its bandwidth is sublinear in N. For large N, it is desirable to also have $o(N)$ server computation. This was shown to be impossible by Beimel et al. [4] for a single query, posing a significant limitation for practical adoption. Due to the above limitation, Beimel et al. [4] defined a *server-preprocessing* PIR scheme, where the server runs a query-independent and client-independent preprocessing in the beginning of the protocol, after which it is possible to achieve $o(N)$ amortized server computation per query. However, all known protocols in this model are of theoretical nature and are far from practical due to large hidden constants (e.g., [4, 24]) or the use of trusted setup [6, 19].

Sublinear PIR through *client preprocessing*. In search for more practical schemes with sublinear server time, Corrigan-Gibbs and Kogan [8] recently introduced a slight modification of the above server-preprocessing PIR model: Here, the client and server *jointly* run a preprocessing phase, after which the client's queries run in $o(N)$ time. The distinguishing factor of this model is that the server does not store any additional information other than the database. Instead, the additional information used for future queries is stored client-side, allowing more efficient scaling to a large number of clients. We call this model the client-preprocessing PIR model (also known as offline-online PIR). Client-preprocessing PIR has definitely brought PIR closer to practice, due to particularly fast online server times. For example, one of the fastest schemes, Checklist [21] (published in USENIX 2021), can answer a PIR query on a database of 3 million entries in less than a millisecond!

At a high level, Checklist (and all prior client-preprocessing PIR schemes [8, 21, 23, 30, 35]) works as follows. We assume here the two-server model where the database is replicated in two, non-colluding servers, Server 0 and Server 1. During the *offline* phase, the client picks $\lambda \cdot \sqrt{N}$ independent random sets $S_1, S_2 \ldots,$. Each S_i is a subset of $\{0, \ldots, N-1\}$ and has

Scheme	Preprocessing Time	Query Time	Query BW	Client Storage	Update Time
Checklist [21]	$O(\lambda \cdot N)$	$O(\sqrt{N})$	$O((\log N)(\lambda \log N + w))$	$O(N \log N + \lambda\sqrt{N} \cdot w)$	$O(\log N)$
SinglePass	$O(N)$	$O(Q)$	$O(Q \cdot w)$	$O(N \log N + (N/Q)w)$	$O(1)$

Table 1: Table measuring asymptotics of updatable single pass scheme against Checklist for databases of N elements of size w. We only include word size in the asymptotics for storage and bandwidth. $Q \in [N]$ is a parameter that determines query time and size, along with storage.

size $\sqrt{N}$. These sets are sent to Server 0. Server 0 computes the parities/hints $p_1, p_2 \ldots$, of these sets by setting

$$p_i = \bigoplus_{k \in S_i} \mathsf{DB}[k],$$

where DB is the public database. The hints are then sent to the client to be stored together with the sets S_i. During the *online* phase, on a query to some index x, the client finds a preprocesssed set S_i (and its corresponding parity p_i) such that S_i contains x, and sends $S_i \setminus \{x\}$ to Server 1.[1] The server computes the parity p of $S_i \setminus \{x\}$ and sends p to the client. Then the client can retrieve $\mathsf{DB}[x]$ as $p \bigoplus p_i$. With many additional careful considerations to this outline, this provides privacy for unlimited queries. The query phase also requires sending a fresh set to Server 0, to replace the set used and maintain the client state's distribution.

Limitation of client-preprocessing PIR schemes. The informal description above shows that all existing client-preprocessing PIR schemes share a significant limitation: Their client-preprecessing phase requires at least $\lambda\sqrt{N} \cdot \sqrt{N} = \lambda \cdot N$ database accesses, for security parameter λ. If we set $\lambda = 128$, the entire database will have to be accessed roughly 128 times during preprocessing before a single query can be issued! (The reason $\lambda\sqrt{N}$ sets are required is technical—to ensure that one of these subsets contains any index the client wishes to query with overwhelming probability.) For use cases where the database is dynamic, or the number of queries is small, such a slow preprocessing can be difficult to justify and lead to particularly slow implementations.

Our contribution: PIR preprocessing in a single pass. In this work, we overcome this limitation by proposing the first client-preprocessing PIR scheme whose preprocessing is (what we informally define as) "single pass", meaning that the preprocessing performs one linear pass over the database, operating on each element exactly once. This is asymptotically optimal [4], and the preprocessing roughly equals the cost of a single PIR query for a non-preprocessing PIR scheme, meaning that preprocessing becomes much more accessible for the many applications that require a small amount of queries. We implement our single pass client-preprocessing PIR scheme and show that it performs extremely well in comparison to previous known schemes. Crucially, our scheme does not require the client to perform offline computation for extended periods of time or to maintain a permanent storage to have fast queries. Using our scheme, the first query runs at approximately the cost of a non-preprocessing PIR query, after which subsequent queries run extremely fast, for the duration of a "session", after which we can delete the state. In other words, the faster preprocessing opens the avenue to allow for preprocessing to happen "online", and consequently for us not to have to worry about the added complexity of dealing with updates, since preprocessing happens often.

Furthermore, as a second contribution, we also show that our scheme can natively support updates to the preprocessed client state in constant time. Previously, client-preprocessing schemes could not support edits and additions natively (except if using $O(\lambda\sqrt{N})$ time per update). Ma et al. [25] (USENIX 2022) and Kogan et al. [21] (USENIX 2021) studied different techniques to overcome this challenge, however both approaches incurred non-constant overhead per update. For example, [21] first maps the PIR scheme to keyword PIR and then uses a "waterfall technique" from hierarchical ORAM [13] to support edits and additions in $O(\log N)$ amortized time per operation, at the cost of also an additional $\log N$ overhead in query bandwidth. With our new scheme, the independence of λ in our preprocessing algorithm and client state allows us to define Edit and Add algorithms for our preprocessed client state both of which run in $O(1)$ time. This means that our scheme can not only improve PIR in the static 'session-based' setting, but also greatly reduce server and client time for updates in the case where clients keep the preprocessed state in storage and wishes to update it sporadically. In Table 1, we provide a table with the comparison between the asymptotics of our new scheme and Checklist. In the table (and henceforth), we denote our new scheme as SinglePass. Just like Checklist, our scheme suffers from a client storage with linear dependency on N. However, because our client storage no longer has a dependency on λ, it actually performs much better in many cases. For example, for a word size of 1024 bytes, $Q = \sqrt{N}$, $\lambda = 128$ our storage is only worse than the client storage for previous schemes [8, 21–23, 30] for N greater than one billion elements. In practice, databases with a size on the order of a million elements encompass a large array of PIR

[1] Since Checklist sets do not have a fast membership testing mechanism, they instead keep a hashmap which maps indices to sets, which is processed offline, for faster queries. This incurs $O(N \log N)$ client storage.

use cases, including private blocklists [21], metadata-hiding communication [1, 2], private movie streaming [14], private wikipedia [27], among others. This is the size of databases we focus on mainly for this work, and where our single pass PIR scheme excels, since we can also greatly reduce Q.

1.1 Notation

Throughout the paper, we will use $\alpha \xleftarrow{\$} \mathbb{S}$ as shorthand for meaning α is an element sampled uniformly at random from the set $\mathbb{S}$. For a natural number N, we will denote $[N]$ as the set of elements $\{0, \ldots, N-1\}$. Let $\mathcal{P}_N$ denote the set of all permutations of $[N]$.

1.2 Intuition

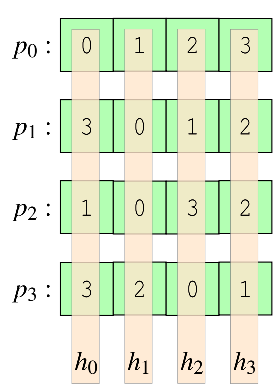
Below, we provide some intuition on our main technical contribution, a novel "single pass" client-preprocessing PIR scheme. We go through a short example of a preprocessing and query step with visuals.

Paramaters and additional notation:

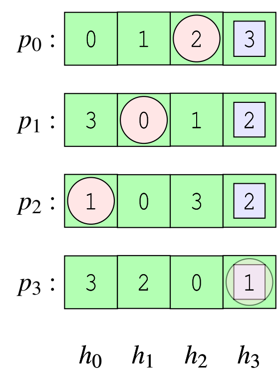
Let $N = 16, Q = 4$, where Q is a tunable parameter for the set size. Again, we work with two, non-colluding servers which both hold the database. First, we organize the database DB as Q arrays of $N/Q = m$ elements. Let DB_i be the array containing $\mathsf{DB}[i * m : (i+1) * m]$.

Example preprocessing and query:

Offline, Server 0 first samples a permutation p_i for each DB_i, $i \in [Q]$, where each p_i is a permutation of $[m]$. Then, for each $j \in [m]$, Server 0 computes $h_j = \bigoplus_{i \in [Q]} \mathsf{DB}_i[p_i(j)]$ and sends it to the client (along with the seed used to generate the permutations). After the offline phase, the client state can be pictured as follows:



This picture shows what elements are contained in each hint given the permutations sampled by Server 0 were $p_0, \ldots, p_3$. For example, $\mathsf{DB}_3[2]$ is contained in h_1 because $p_3(1) = 2$. Now, let us examine how we use this state to perform queries. Consider a query to $x = (1, 2)$. First, we find position ind in p_1 such that $p_1(ind) = 2$. In this case, $ind = 3$. Notice that the hint h_3 contains $\mathsf{DB}[x] = \mathsf{DB}_1[2]$. After we find ind, we send the column that ind belongs to to Server 1, replacing $p_1(2)$ with a random element from $[N/Q]$. We also sample a random position $r_i \in [N/Q]$ for each p_i and send the array of $[p_i(r_i) : i \in [Q]]$ to Server 0. We name the arrays sent to Server 1 and Server 0 as S_{query} and S_{refresh}, respectively. The servers respond with A_b which is an array with all elements requested by the client (where the indices requested are of the form (i, j), where i is the position of the number if the array, and j is the number itself. This can be pictured as follows.



S_{query} :	3	$\cancel{2} \to r$	2	1
S_{refresh} :	2	0	1	1

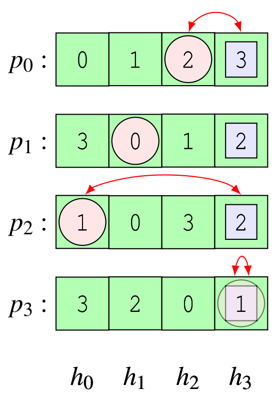
A_1 :	$\mathsf{DB}_0[3]$	$\mathsf{DB}_1[r]$	$\mathsf{DB}_2[2]$	$\mathsf{DB}_3[1]$
A_0 :	$\mathsf{DB}_0[2]$	$\mathsf{DB}_1[0]$	$\mathsf{DB}_2[1]$	$\mathsf{DB}_3[1]$

The red circles represent $p_i(r_i)$ for $i \in [Q]$ and the blue squares represent $p_i(ind)$ for $i \in [Q]$. The client then builds S_{query} and S_{refresh}. S_{query} will be sent to Server 1 as the elements needed to retrieve $\mathsf{DB}[x]$ using its hint (along with an additional random element) and S_{refresh} is sent to Server 0 in order for the client to be able to 'refresh' the client state back to a set of uniform permutations from Server 1's perspective. The client sends S_{query} and S_{refresh} to Server 1 and Server 0, respectively, and gets back A_1 and A_0. The client has now enough information to both:

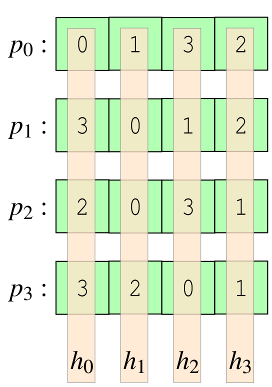
1. Recover $\mathsf{DB}_1[2]$ by simply xoring h_3 with every element in A_1 except $\mathsf{DB}[r]$.
2. Refresh the state by performing *swaps* between the elements sent to Server 0 and the elements sent to Server 1

for each permutation.

The purpose of (1) is clear, to retrieve the element of interest. The purpose of (2) is to *reset* the client state before the next query. For our example, the swapping happens as follows:



Since we have the value for each element on the permutation we wish to swap, we can update the hints accordingly by just xoring the elements in and out. Intuitively, these swaps make it so that each permutation is now completely unknown to the server again. If we can show that the first query is secure, and that after the swaps, each permutation is completely unknown to Server 1, then we can use induction to show that this scheme is secure for any number of queries. Looking ahead, we model and formally prove the statement on the swaps in Lemma 3.1. After the query, this is what our new refreshed client state looks like:



Notice that no swap happened in p_1. This is simply because we don't show any information about p_1 to Server 1. After resetting its state, the client is ready to perform a new query.

We stress here also that although Server 0 sees the preprocessing, it only sees uniform random elements after that (since each r_i is picked independently from each other and from the query itself). Then, Server 0 also learns nothing from any sequence of queries. We formalize this intuition after introducing our full scheme.

1.3 Outline

The work is structured as follows. In Section 2, we define PIR and the client-preprocessing model, as well as other cryptographic primitives to be used in our scheme. In Section 3, we introduce our novel single pass client-preprocessing scheme, prove its security and discuss some efficiency tradeoffs at a high level. In Section 4, we benchmark our PIR scheme against previous state-of-the-art client-preprocessing PIR and show very favorable trade-offs for a myriad of use cases. Finally, in section 5, we discuss an application of our single pass client-preprocessing PIR to a private encyclopedia service, and show that the service has very attractive concrete performance.

2 Model and Definitions

This section is used to expose the model and definitions we will be using throughout our work.

2.1 Private Information Retrieval

In this work, we will consider PIR protocols in the two-server model. The two server model assumes that the database of N bits is replicated in two servers where at least one server behaves honestly. The privacy of the PIR scheme holds for any adversary controlling either server and any number of clients, but does not capture an adversary controlling both servers. This can be satisfiable in practice by having the servers be run by different companies. Throughout this work, we will operate entirely over the two server model. Therefore, we will henceforth denote two server client-preprocessing PIR as simply a client-preprocessing PIR for brevity.

A client-preprocessing PIR scheme for a database DB with N records of length w is tuple of four algorithms[2] defined as:

Definition 2.1 (Client-Preprocessing PIR [21]). *A multi-query, two-server client-preprocessing PIR scheme Π is a tuple of four polynomial-time algorithms:*

- ***Hint***$(\mathsf{DB} \in \{0,1\}^{N\cdot w}) \to (ck, h)$:

  A randomized algorithm that takes in a database $\mathsf{DB} \in (\{0,1\}^w)^N$ and outputs the client keys ck and a client hint h.

- ***Query***$(\mathsf{ck}, x) \to (\mathsf{ck}', q_0, q_1)$:

  A randomized algorithm that takes in the client keys ck and an index $x \in [N]$ and outputs updated client keys ck' and queries q_0, q_1.

- ***Answer***$(\mathsf{DB}, q_b) \to (A_b)$:

  A deterministic algorithm that takes in the database DB a query q_b and outputs an answer A_b.

[2] We use a slightly modified definition from the initial works on Offline/Online PIR [8, 21].

- ***Reconstruct***$(\mathsf{ck}, h, A_0, A_1) \to (h', y_t)$

  A deterministic algorithm that takes as input the client keys ck*, the hint* h *and answer* A *and outputs an updated hint* h' *and* y*, database word at index* x*.*

Furthermore, we require that the multi-query two-server client-preprocessing PIR algorithm $\Pi = (\mathsf{Hint}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Reconstruct})$ *algorithms satisfy the* ***Correctness*** *and* ***Privacy*** *as per Definition 2.2 and Definition 2.3 respectively.*

Concretely, the scheme will run as follows. Whenever a new client connects, Server 0 runs the **Hint** algorithm, and outputs the client's keys and hint, which it returns to the client. After this, for any query x the it desires to query, the client can use the client keys and x to output two queries q_0 and q_1, each directed to Server 0 and Server 1 respectively. Notice that the client at this stage also updates its keys (in our context, the permutations). Each Server returns an answer A_b, which is then used by the client along with the hints to devise its desired information, $\mathsf{DB}[x]$, and update its hints for the next query. Our correctness game will model the fact that under a correct execution, the client should always retrieve the correct word $\{\mathsf{DB}[x_i]\}_{i \in t}$ for any sequence of t queries $x_1, \ldots x_t$ it desires to make. Our privacy game will model the privacy for both servers. Specifically, Server 0 sees the preprocessing and q_0 for each query. Our privacy game models that this should leak no information about each index requested by the client. Server 1 will see only the q_1 for each query and not the preprocessing. Again, we model that seeing the sequence of q_1 for each query performed by a client, no information is leaked about the index the client desires. Our assumption of no collusion comes in here. Although we can model that the information received by each server individually is independent from the queries issued, there are no guarantees about their joint view. We formalize the intuition above on the definitions below.

We now give the definition of correctness for the PIR scheme. Correctness is defined quite naturally. Given a sequence of queries to indices in $x_0, \ldots, x_t \in [N]^T$, the probability that the client outputs $\mathsf{DB}[x_0], \ldots \mathsf{DB}[x_t]$ for each $t \in T$ is 1. Correctness only needs to hold for honest servers, as opposed to privacy for which we will consider malicious servers. We capture this formally in Definition 2.2 below.

Definition 2.2 (Correctness)**.** *A multi-query two-server client-preprocessing PIR scheme* $\Pi = (\mathsf{Hint}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Reconstruct})$ *is* ***correct*** *if, for any* $\lambda, w, N, T \in \mathbb{N}$*, every* $DB \in (\{0,1\}^w)^N$ *and every* $x_0, \ldots, x_{T-1} \in [N]^T$*, the honest execution of the following game always outputs "1":*

- *Compute* $(h, ck) \leftarrow \mathsf{Hint}(DB)$.
- *For* $t = 0, \ldots, T-1$*, compute:*
  - $(\mathsf{ck}, q_0, q_1) \leftarrow \mathsf{Query}(\mathsf{ck}, x_t)$.
  - *For* $b \in \{0,1\}$, $A_b \leftarrow \mathsf{Answer}(q_b)$.
  - $(h, y_t) \leftarrow \mathsf{Reconstruct}(\mathsf{ck}, h, A_0, A_1)$.
- ***If*** $\forall t \in [T]$, $y_t = \mathsf{DB}[x_t]$*, output "1",* ***else*** *output "0".*

We now define privacy for a PIR scheme in our model. Privacy is defined with respect to each server individually. This is standard for the two-server assumption, which assumes non-collusion between both servers. Informally, our privacy definition says that each server, individually, can learn 'no information' about the queries being made by the client, even if an adversary picks the queries adaptively. This implies that each server's view is in fact *independent* from the actual queries being picked. We formalize this intuition below.

Definition 2.3 (Privacy)**.** *A multi-query two-server client-preprocessing PIR scheme* $\Pi = (\mathsf{Hint}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Reconstruct})$ *is* ***private*** *if, for security parameter* λ*, for all polynomially bounded* $N(\lambda), T(\lambda) \in \mathbb{N}$*, for any efficient stateful algorithm* $\mathcal{A}$*, for* $\sigma = \{0,1\}$

$$\Pr\left[\mathsf{PrivGame}^{\sigma}_{\mathcal{A},\lambda,N,T} \to 1\right] \leq 1/2 + \mathsf{neg}(\lambda),$$

where $\mathsf{PrivGame}^0$ *and* $\mathsf{PrivGame}^1$ *are defined as per Figure 1.*

We reiterate that the definitions in this subsection are only slightly modified from the initial works on Offline/Online PIR [8, 21]. The modification is mainly with respect to including the word size w as part of the PIR definition. This helps us better quantify the schemes' efficiency.

The privacy of the scheme is computational, and the correctness is not.

2.2 Pseudorandom Functions and Permutations

2.2.1 Pseudorandom Functions

A pseudorandom function can be used to produce a large number of pseudorandom outputs from a single truly random seed. In our construction, PRFs will be important for concrete efficiency, however, unlike previous schemes, they will not be necessary to argue security.

2.2.2 Sampling Permutations

Our new PIR scheme relies heavily on permutations. Specifically, these permutations will be over 'small' domains of at most a couple million elements. [3] Sampling pseudorandom permutations over small domains (for an adversary that can query the whole permutation) is a well-studied problem with

[3] Small here is comparative to the domain of the AES permutation which has a domain size of 2^{128}. In comparison we require permutations with domain of size on the order of 2^{20} elements.

PrivGame^0 (Privacy for Server 0)

- $b \xleftarrow{\$} \{0,1\}$
- $(\mathsf{ck}, \_) \leftarrow \mathsf{Hint}(\mathsf{DB})$
- $\mathsf{st} \leftarrow \mathcal{A}(1^\lambda, ck)$
- For $t = 0, \ldots, T-1$:
  - $(\mathsf{st}, x_0, x_1) = \mathcal{A}(\mathsf{st})$.
  - $(\mathsf{ck}, q_0, \_) \leftarrow \mathsf{Query}(\mathsf{ck}, x_b)$.
  - $\mathsf{st} \leftarrow \mathcal{A}(\mathsf{st}, q_0)$
- $b' \leftarrow \mathcal{A}(\mathsf{st})$
- Output $b = b'$

PrivGame^1 (Privacy for Server 1)

- $b \xleftarrow{\$} \{0,1\}$
- $(\mathsf{ck}, \_) \leftarrow \mathsf{Hint}(\mathsf{DB})$
- $\mathsf{st} \leftarrow \mathcal{A}(1^\lambda)$
- For $t = 0, \ldots, T-1$:
  - $(\mathsf{st}, x_0, x_1) = \mathcal{A}(\mathsf{st})$.
  - $(\mathsf{ck}, \_, q_1) \leftarrow \mathsf{Query}(\mathsf{ck}, x_b)$.
  - $\mathsf{st} \leftarrow \mathcal{A}(\mathsf{st}, q_1)$
- $b' \leftarrow \mathcal{A}(\mathsf{st})$
- Output $b = b'$

Figure 1: Privacy Games for PIR

a long line of work [18, 26, 28, 29, 31, 32]. To date, current approaches are still not as fast as AES and many have later been found to be insecure [17, 33]. For this work we sample random permutations using a well-known method of sampling truly random permutations, described below, and avoid the problem of sampling pseudorandom permutations over small domains completely.

The Fisher-Yates shuffle, also known as the Fisher-Yates-Durstenfeld-Knuth shuffle [9,10,20], is an algorithm to output an unbiased permutation of the set $[N]$. The algorithm samples a permutation uniformly from the set of all permutations of $[N]$ in $O(N)$ time. In this work, we will use this algorithm to sample permutations uniformly from all the permutations of $[N]$. We will use the following Lemma for our work, proved in previous works. For brevity, we will refer to it as the Fisher-Yates shuffle.

Lemma 2.1 (Fisher-Yates Shuffle [9,10,20])**.** *For any positive integer N, there exists an algorithm $\mathsf{Permute}(N)$ that can output a permutation of the set $[N]$ sampled uniformly from the set of all permutations of $[N]$, $\mathcal{P}_N$, in $O(N)$ time.*

The Fisher-Yates Shuffle and Lemma 2.1 will be crucial in our construction of our single pass client-preprocessing PIR scheme. We use the Permute algorithm above in our main protocol.

We can define a computational version of the Fisher-Yates shuffle by using a PRG and a single random seed to sample all the randomness used in the protocol. Note that for any PPT adversary, a Fisher-Yates shuffle run with true randomness and computational randomness is indistinguishable (except with probability negligibly in the security parameter λ), by security of the PRG. While using a seeded PRG allows for representation of the permutation, this concise representation does not allow for point queries to the permutation in $o(N)$ time. Looking ahead, for our PIR scheme, we store the whole permutation at the client, since this storage turns out to be small compared to the database elements the client needs to store, however, we can optimize offline communication by having the server and client communicate only the seed for the permutations rather than the permutations in their entirety.

3 Single Pass Client-Preprocessing PIR

In this section, we formally present our single pass client-preprocessing PIR scheme. We first give intuition on the scheme's security through a new game which we call the Show and Shuffle game. Afterwards, we present our full scheme (Figure 3), along with our main theorem (Theorem 3.1) and its proof.

3.1 Show and Shuffle

To aid in our proof and abstract out a key concept necessary for our scheme to work, we define a game called Show

Show and Shuffle

Public Parameters: $L, K \in \mathbb{N}$.

Experiment:

1. Sample $(P_1, \ldots, P_L) \xleftarrow{\$} (\mathcal{P}_K)^L$
2. Adversary $\mathcal{A}$ outputs $x = (\ell, k) \in ([L] \times [K])$.
3. Find j such that $P_\ell(j) = k$.
4. Experiment outputs $\mathbf{v} = (v_1, \ldots, v_L)$ where for each $i \in [L]$:
   - $v_i = P_i(j)$ **if** $i \neq \ell$.
   - $v_i \xleftarrow{\$} [K]$ **if** $i = \ell$.
5. Let $b \xleftarrow{\$} \{0,1\}$.
6. **If** $b = 0$: Output $\mathcal{R}_0 = (F_1, \ldots, F_L) \xleftarrow{\$} (\mathcal{P}_K)^L$.
7. **Else:**
   - For each $i \in [L], i \neq \ell$, sample $r_i \xleftarrow{\$} [K]$, and let $P'_i = P_i$ except swapping $P_i(j)$ and $P_i(r_i)$.
   - Let $P'_\ell = P_\ell$
   - Output $\mathcal{R}_1 = (P'_1, \ldots, P'_L)$
8. $\mathcal{A}(\ell, k, \mathbf{v}, \mathcal{R}_b) \to b' \in \{0,1\}$.
9. Game outputs 1 iff $b' = b$

Figure 2: Show and Shuffle experiment.

and Shuffle, formally defined in Figure 2. It is exactly captures single-round security of our scheme. For this game, parametrized by parameters $L, K \in \mathbb{N}$ the experiment uniformly samples L permutations of $[K]$. The adversary outputs a tuple $(\ell, k) \in ([L] \times [K])$. The experiment then defines $j = P_\ell^{-1}(k)$, and outputs $(v_1, \ldots, v_L)$ where $v_i = P_i(j)$ **if** $i \neq \ell$ and $v_j \xleftarrow{\$} [K]$.

Next the experiment flips a bit b, and either outputs a *swapped version* of $(P_1, \ldots, P_L)$ or freshly sampled permutations $(F_1, \ldots, F_L)$ dependent on b. For each of these swapped versions of P_i, denoted P'_i, the experiment samples $r_i \xleftarrow{\$} [K]$ and *swaps* the outputs of $P_i(j)$ and $P_i(r_i)$. We do not perform a swap for P_ℓ. This experiment models exactly one round of our PIR query, and will help us show that after each query, the resulting state of the client's hint is competely uniform. Looking ahead, to prove our scheme secure we will apply the Show and Shuffle Indistinguishability Lemma T times.

Now, we will show that for any adversary playing this game, the adversary cannot guess b with probability greater than $1/2$. This is because the outputs of the experiment in the second round are identically distributed. We prove this in Lemma 3.1.

Lemma 3.1 (Show and Shuffle Perfect Indistinguishability)**.** *For the Show and Shuffle game defined in Figure 2, denoted as* SaS*, for any adversary* $\mathcal{A}$*, for any* $L, K \in \mathbb{N}$*:*

$$\Pr\left[\mathsf{SaS}_{\mathcal{A},L,K} \to 1\right] = 1/2.$$

Proof.

$$\begin{aligned}
&\Pr[\mathsf{SaS}_{\mathcal{A},L,K} \to 1] \\
&= \frac{1}{2}\left(\Pr[\mathsf{SaS}_{\mathcal{A},L,K} \to 1 \mid b=1] + \Pr[\mathsf{SaS}_{\mathcal{A},L,K} \to 1 \mid b=0]\right) \\
&= \frac{1}{2}(\Pr[\mathcal{A}(\ell,k,\mathbf{v},\mathcal{R}_1) \to 1 \mid b=1] \\
&\qquad\qquad + \Pr[\mathcal{A}(\ell,k,\mathbf{v},\mathcal{R}_0) \to 0 \mid b=0]) \\
&= \frac{1}{2}(\Pr[\mathcal{A}(\ell,k,\mathbf{v},\mathcal{R}_1) \to 1 \mid b=1] \\
&\qquad\qquad + (1 - \Pr[\mathcal{A}(\ell,k,\mathbf{v},\mathcal{R}_0) \to 1 \mid b=0])) \\
&= \frac{1}{2} + \frac{1}{2}(\Pr[\mathcal{A}(\ell,k,\mathbf{v},\mathcal{R}_1) \to 1 \mid b=1] \\
&\qquad\qquad - \Pr[\mathcal{A}(\ell,k,\mathbf{v},\mathcal{R}_0) \to 1 \mid b=0])
\end{aligned}$$

Finally, it holds that if we can show that:

$$Pr[\mathcal{A}(\ell,k,\mathbf{v},\mathcal{R}_1) \to 1 \mid b=1] = \Pr[\mathcal{A}(\ell,k,\mathbf{v},\mathcal{R}_0) \to 1 \mid b=0],$$

then we have shown our lemma.

The only difference in the adversary's view, from both inputs, is $\mathcal{R}_b$.

Now, we will show that the distribution of $\mathcal{R}_0$ and $\mathcal{R}_1$. Throughout, we implicitly condition on the other inputs the adversary has access to which are invariant in both schemes: $\ell, k, \mathbf{v}$.

Now, for any $p_1, \ldots, p_L$, where each p_i is a permutation of $[K]$, by construction of our experiment, it follows that:

$$\Pr[(F_1, \ldots, F_L) = (p_1, \ldots, p_L)] = (1/K!)^L.$$

Then, if we can show that the same holds true for our set $\mathcal{R}_1$, we are done. Formally, we have to show that for any $p_1, \ldots, p_L$, where each p_i is a permutation of $[K]$:

$$\Pr[(P'_1, \ldots, P'_L) = (p_1, \ldots, p_L)] = (1/K!)^L.$$

First, notice the important fact that each P'_z for $z \neq \ell$ is only dependent on P'_ℓ and no other permutation. More formally, by construction, we have that:

$$\Pr[P'_z = p_z | \{P'_i = p_i\}_{i \neq z}] = \Pr[P'_z = p_z \mid P'_\ell = p_\ell]. \quad (1)$$

This gives us that for each $P'_z, P'_y, z, y \neq \ell$, P'_z and P'_y are conditionally independent given P'_ℓ. Now, going back to our initial equation, we can break it up as follows:

$$\begin{aligned}
&\Pr[(P'_1,\ldots,P'_L)=(p_1,\ldots,p_L)]\\
&=\Pr[(P'_1,\ldots,P'_{\ell-1},P'_{\ell+1},\ldots,P_L)\\
&\qquad=(p_1,\ldots,p_{\ell-1},p_{\ell+1},\ldots,p_L)|P'_\ell=p_\ell]*\Pr[P'_\ell=p_\ell]\\
&=\left(\prod_{z\in[L],z\neq\ell}\Pr[P'_z=p_z\,|\,P'_\ell=p_\ell]\right)\Pr[P'_\ell=p_\ell]
\end{aligned}$$

Where the first equality follows from a simple chain rule and the second equality follows because the set $\{P'_z\}_{z\neq\ell}$ is conditionally independent given P'_ℓ, from Equation (1).

Notice that the only element that affects P'_z out of P'_ℓ is $j=P'^{-1}_\ell(k)$, so we can condition only on j rather than the whole permutation.

Now, for each $z\in[L], z\neq\ell$, pick any permutation of $[K]$, $p_z=[q_1,\ldots,q_K]$. We need to calculate $\Pr[P'_z=p_z\,|\,j]$. Consider two cases:

Case 1: $q_j=v_z$**:**

$$\begin{aligned}
&\Pr[P'_z=p_z\,|\,j,q_j=v_z]\Pr[q_j=v_z]\\
&=\frac{1}{K}\Pr[P_z=p_z\,|\,j,q_j=v_z]\\
&=\frac{1}{K}\left(\frac{1}{(K-1)!}\right)=1/K!
\end{aligned}$$

Line 1 to 2 holds since $\Pr[q_j=v_z]=\Pr[r_z=j]=1/K$. Line 2 to 3 holds because since $r_z=j$ we did not perform any swaps and the rest of P_z (other than $P_z(j)$ which is fixed) is uniformly distributed by definition.

Case 2: $q_s=v_z$ **for some** $s\neq j$**:**

$$\begin{aligned}
&\Pr[P'_z=p_z\,|\,j,q_s=v_z,P'_z(j)=q_j]\Pr[q_s=v_z,P'_z(j)=q_j]\\
&=\Pr[P'_z=p_z\,|\,j,q_s=v_z,P'_z(j)=q_j]\\
&\qquad\qquad*\Pr[P'_z(j)=q_j|q_s=v_z]\Pr[q_s=v_z]\\
&=\frac{1}{K}\Pr[P'_z=[q_1,\ldots,q_K]\,|\,j,q_s=v_z,P'_z(j)=q_j]\\
&\qquad\qquad*\Pr[P'_z(j)=x|q_s=v_z]\\
&=\frac{1}{K}\Pr[P_z=[q'_1,\ldots,q'_K]|j,q_s=v_z,P_z(s)=q'_s]\\
&\qquad\qquad*\Pr[P_z(s)=q'_s|s\neq j]\\
&=\frac{1}{K}\left(\frac{1}{K-1}\right)\Pr[P_z=[q'_1,\ldots,q'_K]|j,q_z=v_z,P_z(s)=q'_s]\\
&=\frac{1}{K}\left(\frac{1}{K-1}\right)\left(\frac{1}{(K-2)!}\right)=\frac{1}{K!}.
\end{aligned}$$

Where Line 1 to 2 holds by just opening up the conditioning, Line 2 to 3 holds because $\Pr[q_s=v_z]=\Pr[r_z=s]=1/K$. Line 3 to 4 holds by construction, if we redefine $q'_i=q_i$ for every $i\neq s$ and $i\neq j$, and let $q'_s=q_j$ and $q'_j=q_s$ (we are just inverting the swap on P'). Line 4 to 5 holds because $P_z(s)$ cannot equal v_z conditioned on $s\neq j$, but is uniform among the rest of the elements. Finally, Line 5 to 6 holds because when we fix $P_z(s)$ and $P_z(j)$, the rest of the elements are unchanged and uniform by definition of P_z.

So, we have shown that for any $z\neq\ell$, $\Pr[P'_z=p_z\,|\,P'_\ell=p_\ell]=1/K!$.

Plugging this back in to our equation we found before, we have that for any $p_1,\ldots,p_L$:

$$\begin{aligned}
&\Pr[(P'_1,\ldots,P'_L)=(p_1,\ldots,p_L)]\\
&=\left(\prod_{z\in[L],z\neq\ell}\Pr[P'_z=p_z\,|\,P'_\ell=p_\ell]\right)\Pr[P'_\ell=p_\ell]\\
&=\left(\prod_{z\in[L],z\neq\ell}\frac{1}{K!}\right)\Pr[P'_\ell=p_\ell]\\
&=\left(\frac{1}{K!}\right)^{L-1}\Pr[P'_\ell=p_\ell]=\left(\frac{1}{K!}\right)^{L}.
\end{aligned}$$

Note that the last line just follows because $P'_\ell=P_\ell$ was sampled uniformly in the experiment. Then, tying back to the beginning, this means that:

$$\Pr[\mathcal{A}(\ell,k,\mathbf{v},\mathcal{R}_1)\to1\,|\,b=1]=\Pr[\mathcal{A}(\ell,k,\mathbf{v},\mathcal{R}_0)\to1\,|\,b=0].$$

This proves our lemma. ■

3.2 Our Scheme

We now have enough intuition to present our scheme, which uses the intuition and the Show and Shuffle game described above to devise an efficient client-preprocessing PIR scheme. Our scheme can be found in Figure 3.

For clarity, Server 0 will be running Hint(DB), the client runs Query with its index and the keys output by Server 0's Hint and sends q_0 to Server 0 and q_1 to Server 1. Then, each server runs Answer(DB, q_b), and returns the output to the client. Finally, the client runs reconstruct to retrieve the actual word at the desired index and update its hints. The privacy with respect to each server is modeled and explained in Section 2.

For the scheme, we assume that the client implicitly keeps track of x, *ind*, and $\{r_i\}_{i\in[Q]}$ in between Query and Reconstruct. We capture the scheme's security in Theorem 3.1 below.

Theorem 3.1 (Single Pass Client-Preprocessing PIR)**.** *The scheme in Figure 3 is a client-preprocessing Private Information Retrieval scheme as defined in Definition 2.1 and runs with the following complexities:*

- *Hint*(q_h, DB) *runs in* $O(N\cdot w)$ *time and outputs a hint of size* $(N/Q)\cdot w$ *bits.*

Single Pass Client-Preprocessing PIR Scheme

Public Params: Let $Q, N \in \mathbb{N}$ such that $Q|N$. Let $m \in \mathbb{N} = N/Q$. Let DB be an array of N elements of size w. For $i \in [Q]$, let $\mathsf{DB}_i = \mathsf{DB}[i * m : (i+1)m]$.

Hint(DB):

1. For $i \in [Q]$, $p_i \xleftarrow{\$} \mathsf{Permute}(N/Q)$
2. Compute hints $h_0, \ldots h_{m-1}$, where for $j \in [m]$:
$$h_j = \bigoplus_{i=0}^{Q-1} \mathsf{DB}_i \left[p_i(j)\right].$$
3. Output $h = \{h_j\}_{j \in [m]}$, $ck = \{p_i\}_{i \in [Q]}$.

Query($\mathsf{ck}, x = (i^*, j^*) \in [Q] \times [m]$):

1. Parse $x = (i^*, j^*)$. Find $ind \in [m]$ such that $p_{i^*}(ind) = j^*$.
2. Let $S = [p_j(ind) : j \in [Q]]$. Rewrite $S[i^*] = r^* \xleftarrow{\$} [m]$.
3. Sample $r_0, \ldots, r_{Q-1}$ independently and uniformly from $[N/Q]$.
4. Let $S_{\text{refresh}} = [p_i(r_i) : i \in [Q]]$,
5. For $i \in [Q], i \neq i^*$, swap $p_i(ind)$ and $p_i(r_i)$.
6. Output ck, $q_0 = S_{\text{refresh}}$, $q_1 = S_{\text{query}}$.

Answer(DB, q_b):

1. Return $A_b = [\mathsf{DB}_i[q_b[i]] : i \in [Q]]$.

Reconstruct(ck, h, A_0, A_1):

1. Let $DB[x] = \mathsf{DB}_{i^*}[j^*] = \left(\bigoplus_{i \in [Q], i \neq i^*} A_1[i]\right) \oplus h_{ind}$
2. For each $i \in [Q], i \neq i^*$, update:
$$h_{ind} = h_{ind} \oplus A_0[i] \oplus A_1[i]$$
$$h_{r_i} = h_{r_i} \oplus A_0[i] \oplus A_1[i]$$
3. Output $\mathsf{DB}[x], \mathsf{ck}, h$

Figure 3: Our main scheme.

- *Query*(ck,x) *runs in* $O(Q)$ *time.*
- *Answer*(DB,q_b) *runs in* $O(Q\cdot w)$ *time.*
- *Reconstruct*(ck,h,A_0,A_1) *runs in* $O(Q\cdot w)$ *time.*
- *The client stores a state with* $O(N\log N+(N/Q)\cdot w)$ *bits.*
- *The server stores only DB.*

We prove Theorem 3.1 in the Appendix A, however the basic intuition for the security of our scheme is that every time we show any element part of the permutation, we shuffle it back in. Then, we show that an adversary that can break the PIR scheme can win the Show and Shuffle experiment (Figure 2) and contradicts Lemma 3.1.

4 Benchmarking for Static Databases

In this section, we benchmark an implementation of our scheme against previous schemes for a variety of parameters. Our static scheme is implemented in only about 300 lines of Go code and 150 lines of C code, as an extension to the existing PIR framework from Checklist [21]. Benchmarks are all run on an AWS EC2 instance of size t2.2xlarge. All schemes are run on a single thread. [4]

For our comparisons, we compare our scheme against Checklist [21], TreePIR [23] and MIR [30], three novel state-of-the-art client preprocessing PIR schemes.

None of the previous works is able to avoid the dependency on the security parameter for preprocessing and client storage. [5]

Benchmarking the cost of the 'static' version of the scheme serves two purposes. First, it gives a baseline as to what to expect when we also include the logic to handle updates. Secondly and more importantly, the 'static database' scenario is akin to a client that does not want to hold long term storage. Instead, the client would like to preprocess the database on-demand only to perform a small amount of queries quickly, and then leave and delete the state. This is akin to web-browsing and could find applications for websites such as Wikipedia, Youtube, domain name lookups or compromised credential checking where clients usually perform multiple queries in a short span of time, and do not wish to save a permanent client state between sessions. To properly evaluate such scenario, we also include a comparison against the state-of-the-art two-server PIR scheme with no preprocessing from [5, 11, 15], which we will denote as DPF.

[4] An anonymized version of our code for the submission can be found at: https://github.com/SinglePass712/Submission.

[5] In fact, because of this, as aforementioned in Section 1, even for a databases of 1 billion 1024 byte elements, our scheme concretely achieves less client storage than some of these scalable schemes, despite its worst asymptotics, for the same Q [23, 30, 35].

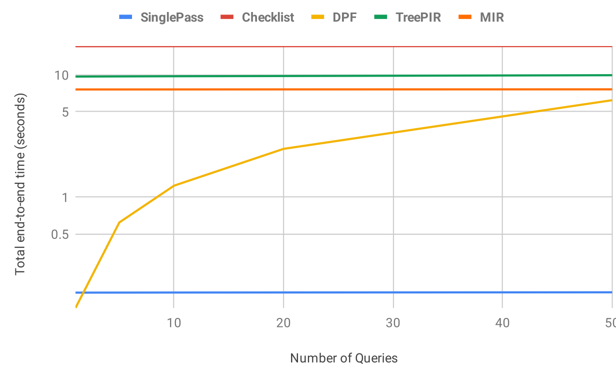


Figure 4: Comparison of total end-to-end time for varying number of queries over a database of one million 512-byte elements.

The choice of parameters picked throughout the section reflect a sample use case of a private encyclopedia service, where a user would browse to a private encyclopedia website, access a couple of articles privately, and afterwards leave (after which we can delete the preprocessing). In this scenario, it matters that the preprocessing is very fast so that it can take place as soon as the user accesses the website with no wait time. Therefore, we split the benchmarks in two different figures. First, we measure end-to-end time of our scheme against previous state of the art client preprocessing schemes and the state-of-the-art two-server PIR scheme with no preprocessing, denoted DPF for a database of one million 512-byte elements. This is shown in Figure 4. Out of the range database sizes and entry sizes we found for encyclopedias online (up to two million entries of size 512 to 2048 bytes) we picked this one arbitrarily to be representative; however the trend seen in Figure 4 holds across any set of parameters. This is fundamentally because of the asymptotic improvement of a factor of λ in preprocessing time. Results show that whereas other schemes start beating DPF after 50+ queries, the total end-to-end time of SinglePass is already better even when performing two queries.

In Figure 5, we provide a more fine-grain comparison of preprocessing time, per-query time, and bandwidth between SinglePass, Checklist,TreePIR and MIR, thorugh the whole range of parameters mentioned in the paragraph above. Here, as aforementioned, the parameter choices reflect our envisioned use case of a private encyclopedia service. For these tests, we *normalize the tests by client storage.* By this, we mean that we run the tests for MIR,TreePIR and Checklist, and subsequently run SinglePass picking the smallest Q such that our client storage does not exceed the storage of the previous schemes. In doing this, we can benchmark the performance of the schemes when given similar client resources, and we see that SinglePass gives us very favorable trade-offs in preprocessing time, and query time, at a modest bandwidth cost with respect to MIR and Checklist. In Appendix C, we include tests normalized by query time. In those tests, all schemes have very close query times, and the other metrics vary. We discuss this further there.

5 Handling Dynamic Databases

In contrast to the scenarios studied in Section 4, there are other cases, such as within user applications (either mobile or desktop) where using some long term client storage is perfectly acceptable. In these cases, it makes sense that the client only run the preprocessing once (or rarely) and re-use the information already preprocessed in the future, downloading only the changes made to the database. In this section we study how to slightly modify our single pass PIR scheme to support updates.

Previous works in preprocessing PIR do not have the ability to handle updates natively in constant time. At a high level, this is because the preprocessing step of previous client-preprocessing PIR schemes involves sampling about $\lambda\sqrt{N}$ independent subsets of $[N]$. Since the hint is a collection of independent subsets, we need to potentially update all of them.

To deal with this, previous works [21, 23] borrowed techniques from hierarchical ORAM [12] to support amortized $O(\log N)$ time updates by incurring overheads in both query time and client storage (this approach is also known as a *waterfall update* approach). The transformation from a static preprocessing PIR scheme into an updatable one in this manner requires two steps. First, a transformation from standard PIR to *keyword* PIR with indices as the keys [7]. Second, a black-box application of this waterfall preprocessing data structure approach to this keyword PIR. We cannot apply the waterfall approach directly to standard PIR since it involves storing log databases of exponentially increasing size (where the smallest holds the newest updates, and we update the $i-th$ smallest only when all the ones smaller than i are full; the largest one holds N elements), and as such we need a way to maintain the original indices.[6] Each of these steps adds some overhead in bandwidth, communication, client and server time. Other approaches have been studies [25] but also incur significant overheads in comparison to the static preprocessing scheme.

To provide a robust set of update operations, we must support edits, additions and deletions. Our new single pass PIR scheme only preprocesses each element exactly once. Since our hint only consists of a single (partitioned) permutation of the database, we can update the hint data structure in $O(1)$ time natively, without any other additional overhead. Below, we give some intuition first on how to handle each of these, and subsequently provide formal algorithms for each.

Edits/Deletions: For an edit to an index $x = (i, j) \in [Q] \times [N/Q]$. We compute $k = p_i^{-1}(j)$ and this tells us which hint contains x, h_k. Then, given the original preprocessed value at index x, denoted $\mathsf{DB}[x]_{old}$ and the new value at x, denoted $\mathsf{DB}[x]_{new}$, the client can update its hint by simply updating $h_k = h_k \oplus \mathsf{DB}[x]_{old} \oplus DB[x]_{new}$, after calculating k. Since all these steps take constant time, editing takes constant time. We also define a deletion to be an edit where $\mathsf{DB}[x]_{new}$ equals 0 (or a special delete value).

Additions: We can support additions at the end of the array in two steps. We first sample a new permutation p_Q, and then let $k = p_Q^{-1}(N)$. We let $h_k = \mathsf{DB}[N]$. Note that we only have to sample the permutation once for every N/Q additions, and sampling the permutation takes $O(N/Q)$ time. Every subsequent addition takes $O(1)$ time, since we just xor it into the appropriate hint by checking the inverse. This gives us constant amortized time for addition. This can be easily deamortized by sampling a constant part of the permutation at each step.

The attentive reader will notice that the Add operation introduces a new permutation, and therefore a new element being sent on each query which may be out of bounds for the current database. The server can just choose to ignore indices out of bounds. Furthermore, every N/Q additions increase the bandwidth by an additive factor of $2w$ (by increasing our value of Q to be $Q+1$). After N additions, this means that our new bandwidth will be $2Q$ rather than our initially selected Q. After N additions, one can re-run the preprocessing step to again tune Q to whatever is desired while maintaining a constant update cost (since this is performed only every N updates). This step too can be de-amortized by running a constant amount of steps at each update.

We give the full procedure for Edit and Add in Figure 6.

Next, we look at how our update capabilities perform in practice.

6 Benchmarking for Dynamic Databases

We implement the updatable logic for our scheme in an additional 200 lines of Go code and 50 lines of C code. We again run all benchmarks on the same AWS EC2 instance of size t2.2xlarge. Schemes are run on a single thread.

After preprocessing, the client now can update its preprocessed hint through an *update* function. In the case of SinglePass, the server streams to the client the changes on the database since the last client update (the client keeps track of the version number for its last update/preprocess call, and sends it along with the update request), and the client updates its hint accordingly, in the fashion described in Section 5.

In addition to benchmarking the preprocessing time, query time, bandwidth and client storage, we now also measure the update time for each of the databases studied before. The update time in the tables and figures reflects the amortized time for a batch of 500 updates.

Unlike in Section 4, we do not include the DPF scheme in our benchmarks for the updatable version, since these use

[6] The full approach is discussed in detail on the Checklist paper [21].

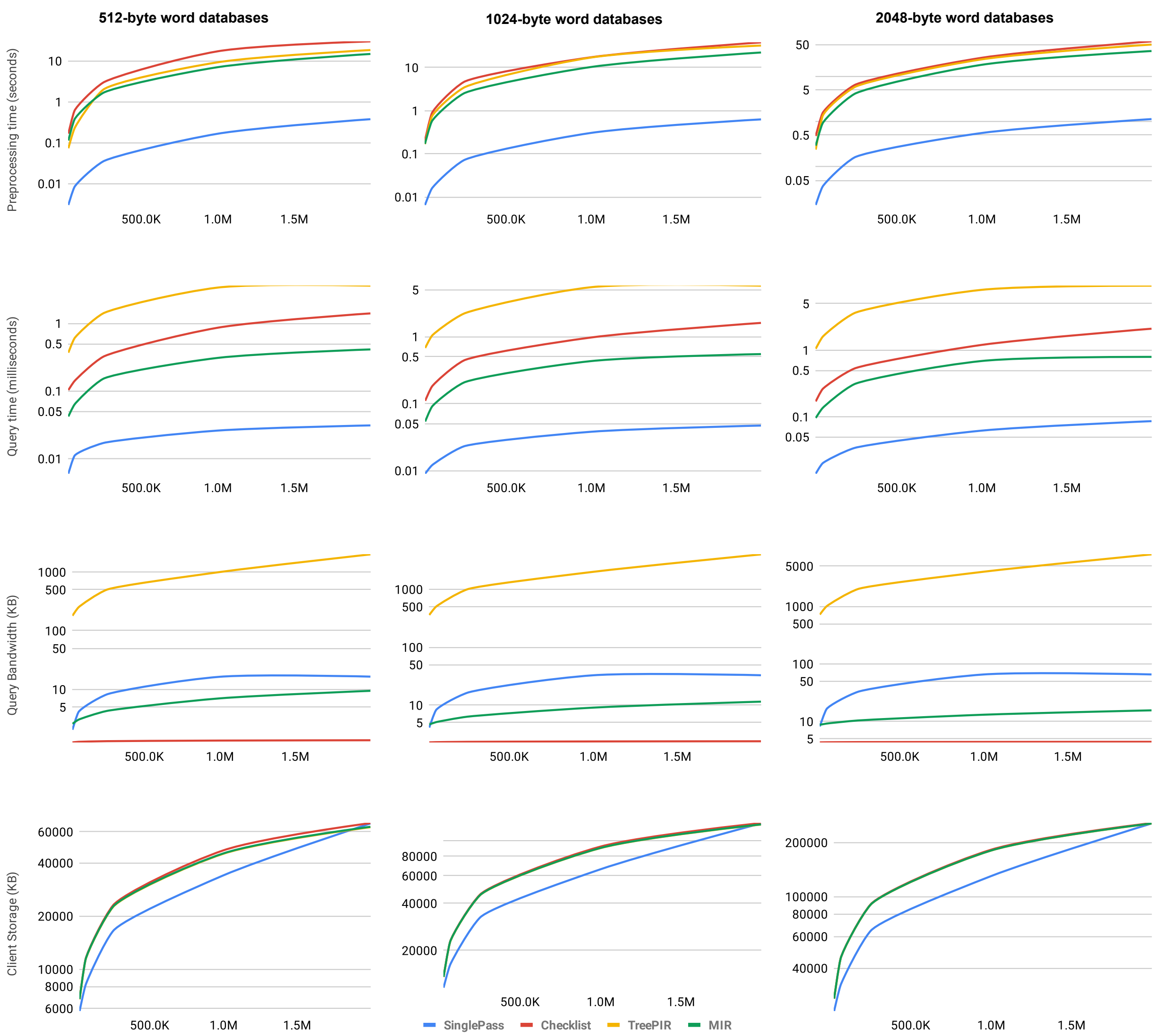


Figure 5: Comparison of benchmarking over preprocessing time, query time, bandwidth and client storageover increasingly sized static databases (x-axis) for different element sizes (on a log scale).

Edit$(ck, h, i, \mathsf{DB}[i]_{old}, \mathsf{DB}[i]_{new})$:

1. Compute i_0, i_1 such that $i = (\lceil N/Q \rceil) i_0 + i_1$.
2. Let $k = p_{i_0}^{-1}(i_1)$.
3. Let $h_k = h_k \oplus \mathsf{DB}[i]_{old} \oplus \mathsf{DB}[i]_{new}$

Add(ck, h, w):

1. Let $j = N \mod (N/Q)$
2. If $j = 0$:
   (a) Sample $p_Q = \mathsf{Permute}([N/Q])$
   (b) Let $Q = Q + 1$
3. Let $k = p_{Q-1}^{-1}(j)$
4. Let $h_k = h_k \oplus w$
5. Let $N = N + 1$

Figure 6: Update Operations for SinglePass

cases assume the client will run the preprocessing only once (or at least extremely sparsely). Also, Checklist is the only scheme out of the previous state of the art PIR schemes which provide an updatable implementation, so in this section, we compare exclusively against Checklist. Note that all previous schemes (Checklist included) do not naively support updates. Instead, one *can* handle updates through employing a similar technique to the one used in Checklist (as also mentioned in [23, 35]). Through this technique, they achieve $O(\log(n))$ time updates and with significant storage overhead at the client.

We run a complete test suite in Figure 7, with the same parameters as we picked in Section 4. This could reflect a mobile app for the private encyclopedia service envisioned in Section 4, where it is okay to use some permanent storage. In this scenario, minimizing update times is crucial to to impose as small as a burden as possible on the servers. On the chart, the update represents the update time for a batch of 500 updates. Notice that while Checklist's update time scales logarithmically with the database size, the update time for SinglePass remains basically the same across all database sizes. The other trends, preprocessing time, query time and query bandwidth, follow mostly the same patterns identified in Section 4, with an improvement in the comparative bandwidth for SinglePass, because of the overhead incurred when mapping Checklist to its updatable version. For databases of up to 1 million elements, we notice that SinglePass has a bandwidth which is at most 1.5x Checklist's bandwidth, while maintaining a query time reduction of roughly 20x on average across alll experiments and a preprocessing time speed up of up to 100x. As in Section 4, we pick Q for our single pass scheme accordingly, so as to benchmark both schemes while using comparable client storage, and include tests that fix query time in Appendix C.

As a second source of comparison, in Table 2, provide a benchmark of SinglePass and Checklist for the blocklist application studied in Checklist, with the parameters picked in the paper: a database of 3 million 32-byte elements that is updated in batches of 500. In comparison to Checklist, our scheme achieves over 100x speed up in preprocessing, over a 47x speed up in query time, an approximate 2x saving in bandwidth and a 19x faster update time. The saving stems primarily from not requiring a dependency on the security parameter for the preprocessing and storage. Because our client storage for using the same set size as Checklist is much smaller, we can tune Q such as to use about the same storage. For the scenario benchmarked in Table 2, this comes out a set size of $Q = 10$. This is how the λ saving in client storage and preprocessing can translate into improved query time. The update time discrepancy of about 20x follows the expected difference between an algorithm that runs in constant time and an algorithm with a $O(\log N)$ overhead. [7]

7 Next Steps

Our new scheme, SinglePass, expands the realm of possibility for Private Information Retrieval by removing the dependency of the scheme's efficiency with the security parameter. As seen throughout Section 4 and Section 6, this allows for great practical savings across the board. Some natural next questions for this line of work are whether we can have a client-preprocessing PIR scheme whose efficiency is independent of λ, and that operates in the single server setting; and whether we can get rid of the linear dependency of our scheme's client state and the database size.

References

[1] Ishtiyaque Ahmad, Yuntian Yang, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. Addra: Metadata-private voice communication over fully untrusted infrastructure. 2021.

[7] One caveat of the comparison is that Checklist supports keyword queries (it is necessary for the updatable version of Checklist that it support keywords in order to achieve the $O(\log N)$ amortized bandwidth). Our single pass scheme is a pure PIR scheme that only supports index queries. However, using cuckoo hashing it could be translated to a keyword PIR scheme with a 2x overhead [34].

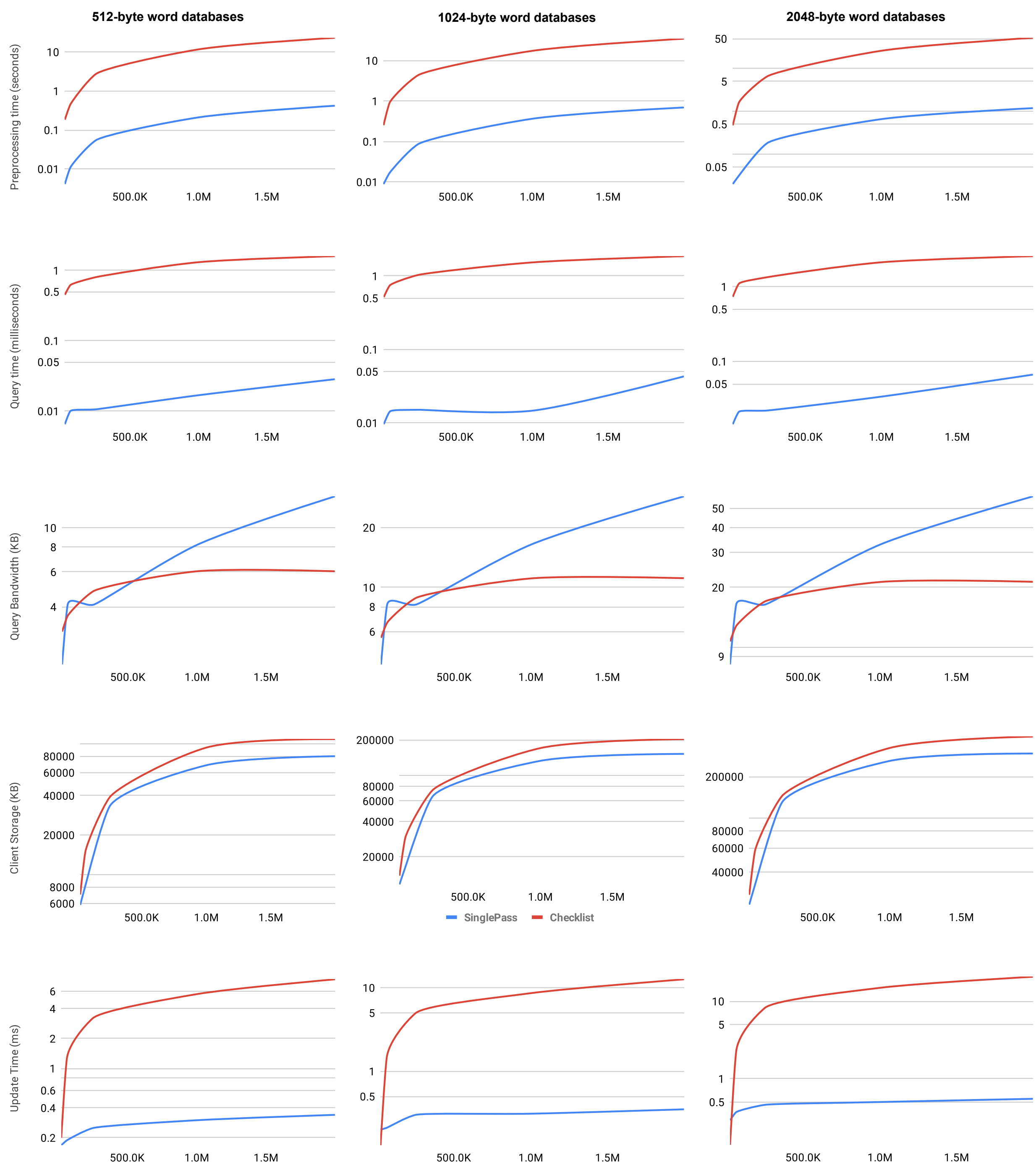


Figure 7: Comparison of benchmarking over preprocessing time, query time, bandwidth, client storage and update time over increasing updatable databases sizes (x-axis) for different element sizes (on a log scale).

Scheme	Preprocessing Time (s)	Query Time (ms)	Query BW (KB)	Client Size (MB)	Update Time (ms)
SinglePass	0.122	0.02ms	0.68KB	23.3MB	0.19ms
Checklist	13.22s	0.95ms	1.48KB	23.6MB	3.78ms

Table 2: Comparison for Updatable Database with 3,000,000 32-byte elements. The update time is for a batch of 500 updates.

[2] Sebastian Angel and Srinath Setty. Unobservable communication over fully untrusted infrastructure. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*, OSDI'16, pages 551–569, USA, November 2016. USENIX Association.

[3] Michael Backes, Aniket Kate, Matteo Maffei, and Kim Pecina. ObliviAd: Provably Secure and Practical Online Behavioral Advertising. In *2012 IEEE Symposium on Security and Privacy*, pages 257–271, May 2012. ISSN: 2375-1207.

[4] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the Servers Computation in Private Information Retrieval: PIR with Preprocessing. In Mihir Bellare, editor, *Advances in Cryptology — CRYPTO 2000*, Lecture Notes in Computer Science, pages 55–73, Berlin, Heidelberg, 2000. Springer.

[5] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function Secret Sharing. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015*, Lecture Notes in Computer Science, pages 337–367, Berlin, Heidelberg, 2015. Springer.

[6] Elette Boyle, Yuval Ishai, Rafael Pass, and Mary Wootters. Can We Access a Database Both Locally and Privately? pages 662–693, November 2017.

[7] Benny Chor, Niv Gilboa, and Moni Naor. Private Information Retrieval by Keywords, 1998. Report Number: 003.

[8] Henry Corrigan-Gibbs and Dmitry Kogan. Private Information Retrieval with Sublinear Online Time. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, Lecture Notes in Computer Science, pages 44–75, Cham, 2020. Springer International Publishing.

[9] Richard Durstenfeld. Algorithm 235: Random permutation. *Communications of the ACM*, 7(7):420, July 1964.

[10] Ronald Aylmer Fisher and Frank Yates. *Statistical tables for biological, agricultural and medical research, edited by R.A. Fisher and F. Yates. 6th ed.* Edinburgh: Oliver and Boyd, 1963. Accepted: 2006-06-27T07:57:52Z.

[11] Niv Gilboa and Yuval Ishai. Distributed Point Functions and Their Applications. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, Lecture Notes in Computer Science, pages 640–658, Berlin, Heidelberg, 2014. Springer.

[12] Oded Goldreich, S. Goldwasser, and S. Micali. How to Construct Random Functions (Extended Abstract). In *FOCS*, 1984.

[13] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, May 1996.

[14] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath Setty, Lorenzo Alvisi, and Michael Walfish. Scalable and private media consumption with Popcorn. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 91–107, USA, March 2016. USENIX Association.

[15] Syed Mahbub Hafiz and Ryan Henry. A Bit More Than a Bit Is More Than a Bit Better: Faster (essentially) optimal-rate many-server PIR. *Proceedings on Privacy Enhancing Technologies*, 2019(4):112–131, October 2019.

[16] Laura Hetz, Thomas Schneider, and Christian Weinert. Scaling Mobile Private Contact Discovery to Billions of Users, 2023. Publication info: Published elsewhere. Minor revision. ESORICS 2023.

[17] Mihir Bellare Hoang, Viet Tung and Stefano Tessaro. Message-recovery attacks on Feistel-based Format Preserving Encryption. Technical Report 794, 2016.

[18] Viet Tung Hoang, Ben Morris, and Phillip Rogaway. An Enciphering Scheme Based on a Card Shuffle. Technical Report arXiv:1208.1176, arXiv, November 2014. arXiv:1208.1176 [cs] type: article.

[19] Justin Holmgren, Ran Canetti, and Silas Richelson. Towards Doubly Efficient Private Information Retrieval. Technical Report 568, 2017.

[20] Donald E Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms*. Addison-Wesley, 1969.

[21] Dmitry Kogan and Henry Corrigan-Gibbs. Private Blocklist Lookups with Checklist. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 875–892. USENIX Association, 2021.

[22] Arthur Lazzaretti and Charalampos Papamanthou. Near-Optimal Private Information Retrieval with Preprocessing. In Guy Rothblum and Hoeteck Wee, editors, *Theory of Cryptography*, Lecture Notes in Computer Science, pages 406–435, Cham, 2023. Springer Nature Switzerland.

[23] Arthur Lazzaretti and Charalampos Papamanthou. TreePIR: Sublinear-Time and Polylog-Bandwidth Private Information Retrieval from DDH. In *Advances in Cryptology – CRYPTO 2023: 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20–24, 2023, Proceedings, Part II*, pages 284–314, Berlin, Heidelberg, August 2023. Springer-Verlag.

[24] Wei-Kai Lin, Ethan Mook, and Daniel Wichs. Doubly Efficient Private Information Retrieval and Fully Homomorphic RAM Computation from Ring LWE, 2022. Report Number: 1703.

[25] Yiping Ma, Zhong Ke, Tal Rabin, and Sebastian Angel. Incremental Offline/Online PIR (extended version). In *USENIX Security 2022*, 2022.

[26] Rashed Mazumder, Atsuko Miyaji, and Chunhua Su. A simple construction of encryption for a tiny domain message. pages 1–6, March 2017.

[27] Samir Menon. SpiralWiki, 2022.

[28] Sarah Miracle and Scott Yilek. Cycle Slicer: An Algorithm for Building Permutations on Special Domains. pages 392–416, November 2017.

[29] Ben Morris and Phillip Rogaway. Sometimes-Recurse Shuffle. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, Lecture Notes in Computer Science, pages 311–326, Berlin, Heidelberg, 2014. Springer.

[30] Muhammad Haris Mughees, Sun I, and Ling Ren. Simple and Practical Amortized Sublinear Private Information Retrieval, 2023. Publication info: Preprint.

[31] Thomas Ristenpart and Scott Yilek. The Mix-and-Cut Shuffle: Small-Domain Encryption Secure against N Queries. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, Lecture Notes in Computer Science, pages 392–409, Berlin, Heidelberg, 2013. Springer.

[32] Emil Stefanov and Elaine Shi. FastPRP: Fast pseudo-random permutations for small domains. Cryptology ePrint Report 2012/254. Technical report, 2012.

[33] Viet Tung Hoang Tessaro, Stefano and Ni Trieu. The Curse of Small Domains: New Attacks on Format-Preserving Encryption. Technical Report 556, 2018.

[34] Kevin Yeo. Cuckoo Hashing in Cryptography: Optimal Parameters, Robustness and Applications. In *Advances in Cryptology – CRYPTO 2023: 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20–24, 2023, Proceedings, Part IV*, pages 197–230, Berlin, Heidelberg, August 2023. Springer-Verlag.

[35] Mingxun Zhou, Andrew Park, Elaine Shi, and Wenting Zheng. Piano: Extremely Simple, Single-Server PIR with Sublinear Server Computation, 2023. Publication info: Published elsewhere. Major revision. IEEE S&P 2024.

A Proof of Main Theorem

In this section, we restate and prove Theorem 3.1

Theorem 3.1 (Single Pass Client-Preprocessing PIR)**.** *The scheme in Figure 3 is a client-preprocessing Private Information Retrieval scheme as defined in Definition 2.1 and runs with the following complexities:*

- $\mathsf{Hint}(q_h, \mathsf{DB})$ *runs in* $O(N \cdot w)$ *time and outputs a hint of size* $(N/Q) \cdot w$ *bits.*
- $\mathsf{Query}(ck, x)$ *runs in* $O(Q)$ *time.*
- $\mathsf{Answer}(DB, q_b)$ *runs in* $O(Q \cdot w)$ *time.*
- $\mathsf{Reconstruct}(ck, h, A_0, A_1)$ *runs in* $O(Q \cdot w)$ *time.*
- *The client stores a state with* $O(N \log N + (N/Q) \cdot w)$ *bits.*
- *The server stores only* DB.

Proof. Complexities: For Hint, by Lemma 2.1, we can sample the permutations needed in $O(N)$ time, and then use random access to compute the hints in $O(N \cdot w)$ time. Query to $x = (i^*, j^*)$ has to find the index *ind* such that $p_{i^*}(ind) = j^*$. Notice that given the expanded format of each permutation and its inverse, this can be done in $O(1)$ time, by simply taking $p_{i^*}^{-1}(j^*)$.[8] After, we just have to index $p_i(ind)$ for $i \in [Q]$

[8] We can store the inverse along with the permutation with constant overhead. In practice, for some scenarios, it might be beneficial to not store the inverse in order to save space. In those cases, the client time would be $O(Q + N/Q)$. This is the only place where the inverse is used for that static scheme. For the updatable scheme, we require the inverse to get $O(1)$ update operations.

which takes $O(Q)$ time, and send that to the server (with a symmetric number of operations to generate the refresh hint). Answer only reads the array of size Q and accesses each element indexed by the array. Assuming random access costs constant time, this also runs in $O(Q)$ time. Finally, Reconstruct does $O(Q)$ operations to update the hint parities of the elements used (from the above complexities, only $O(Q)$ elements are sent/received on each query. The client storage is the hint it receives from Hint (and whatever refresh operations done on it, which don't increase its size) plus the expanded client keys (N indices of $[N]$, therefore, $N \log N$ space). Alternatively, the client can store only the seed used for the permutations and expand them at query time, but by Lemma 2.1 this would then require Query to run in $O(N)$ time.

Correctness: Follows by construction (we reiterate correctness is modeled for honest servers only).

Note that after a correct preprocessing, Server 0 sends back to the client $(ck, h) = (\{p_i\}_{i \in Q}, \{h_j\}_{j \in \lfloor N/Q \rfloor}$ where each p_i is a pseudorandom permutation of $[1, \lfloor N/Q \rfloor]$ and each $h_j = \bigoplus_{i \in Q} \mathsf{DB}[p_i(j)]$.

Then, for a query to $x = (i^*, j^*)$, first define ind to be the element of $[N/Q]$ such that $p_{i^*}(ind) = j^*$. If Server 1 responds to q_1 honestly, then it is clear to see that the client's output for the query is $\left(\bigoplus_{i \in Q, i \neq i^*} \mathsf{DB}_i[p_i(ind)]\right) \oplus h_{ind} = \mathsf{DB}_{i^*}[p_{i^*}(ind)] = \mathsf{DB}_{i^*}[j^*] = \mathsf{DB}[x]$.

For a subsequent query, what is left to show is that for every following query, for every $j \in [N/Q]$, $h_j = \bigoplus_{i \in [Q]} \mathsf{DB}_i[p_i(j)]$ after the swaps. Notice that for each swap between $p_i(k)$ and $p_i(v)$, we let $h_k = h_k \oplus \mathsf{DB}_i[p_i(k)] \oplus \mathsf{DB}_i[p_i(v)]$ therefore effectively removing the old element in this hint's position from the xor and adding the new one (this happens symmetrically on h_v). Then, at the beginning of the next query, each hint h_j is still equal to $\bigoplus_{i \in [Q]} \mathsf{DB}_i[p_i(j)]$. Then, by our argument for the first query, correctness holds as well (and holds for any T).

Privacy: We consider the privacy of each server separately according to the games defined in Figure 1.

Server 0: To show privacy for Server 0 for any $\lambda \in \mathbb{N}$ and any $N(\lambda), T(\lambda)$, for any PPT adversary $\mathcal{A}(\lambda)$,

$$\Pr\left[\mathsf{PrivGame}^0_{\mathcal{A},\lambda,N,T} \to 1\right] \leq 1/2 + \mathsf{neg}(\lambda).$$

Note that in **PrivGame**^{0}, which models the view of Server 0, Server 0 has access to both the client keys, and then for each query $t \in T$, it gets access to the corresponding q_0 for that query, which we will denote here as q_0^t.

Notice that as long as each p_i is bijection from $[N/Q]$ to $[N/Q]$, then each $p_i(r_i)$ is uniform and independent of the query being made, since by definition each r_i is uniform and independent of the query being made. Since for every step, the new swapped p_i is still a bijection, then this holds for any timestep t. So each q_0 is a set of elements in $[N/Q]$ independent of the query being made. Then, since each step q_0 is independent of the query being made, it follows that for any pair x_0, x_1, even conditioned on seeing the preprocessing, an adversary acting as Server 0 cannot distinguish between $b = 0$ and $b = 1$ on the **PrivGame**^{0} experiment. If we use pseudorandomness output by a PRG with security parameter λ rather than true randomness to sample each r_i, we incure a negligible probability of distinguishing, directly from the PRG security definition. Finally, we get that,

$$\Pr\left[\mathsf{PrivGame}^0_{\mathcal{A},\lambda,N,T} \to 1\right] \leq 1/2 + \mathsf{neg}(\lambda).$$

Server 1: To show privacy for Server 1 for any $\lambda \in \mathbb{N}$ and any $N(\lambda), T(\lambda)$, for any PPT adversary $\mathcal{A}(\lambda)$,

$$\Pr\left[\mathsf{PrivGame}^1_{\mathcal{A},\lambda,N,T} \to 1\right] \leq 1/2 + \mathsf{neg}(\lambda).$$

Notice that here, the adversary acting as Server 1 does not get access to the preprocessing (since it is run by Server 0), but it does see q_1 for every timestep $t \in T$.

Here, we use Theorem B.1. Note that Experiment 1 in Theorem B.1 is exactly equivalent to our PIR query at each timestep. Then, by Theorem B.1, we can replace each q_1 shown to Server 1 by uniform random elements of $[N/Q]$. This implies that for any x_0, x_1 picked by the adversary as inputs from **PrivGame**^{1}, the outputs at each timestep will be identically distributed and indistinguishable when using true randomness. Then, we can replace the randomness used by pseudorandomness sampled through a PRG with security parameter λ (which is what we do in our scheme), and it follows by the PRG security that this would be computationally indisinguishable from before. Then, it follows that:

$$\Pr\left[\mathsf{PrivGame}^1_{\mathcal{A},\lambda,N,T} \to 1\right] =\leq 1/2 + \mathsf{neg}(\lambda).$$

∎

B Server 1 Indistinguishability

We include a theorem here to modularize the proof of Theorem 3.1. It is used to prove our scheme satisfies **PrivGame**^{1}.

Theorem B.1 (Query indistinguishability). *For any adaptive adversary $\mathcal{A}$, Experiment 0 and Experiment 1, as defined in Figure 8, are perfectly indistinguishable.*

Proof. We prove this through a series of hybrid experiments, starting from H_0 :

Experiment H_0

Public Parameters: $N, Q \in \mathbb{N}$, assume $Q|N$.

Experiment:

Experiment 0

Public Parameters: $N, Q \in \mathbb{N}$, assume $Q|N$.

Experiment:

1. **For** $t \in \{1, \ldots T\}$:
   (a) Adversary outputs $x^t = (i^t, j^t) \in ([Q] \times [N/Q])$.
   (b) Output $(y_1^t, \ldots, y_Q^t) \xleftarrow{\$} [N/Q]^Q$.

Experiment 1

Public Parameters: $N, Q \in \mathbb{N}$, assume $Q|N$.

Experiment:

1. Sample $P_1^1, \ldots P_Q^1 \xleftarrow{\$} (\mathcal{P}_{N/Q})^Q$.
2. **For** $t \in \{1, \ldots, T\}$:
   (a) Adversary outputs $x^t = (i^t, j^t) \in ([Q] \times [N/Q])$.
   (b) Find *ind* s.t. $P_{i^t}^t(ind) = j^t$
   (c) Output $S = (v_1^t, \ldots, v_Q^t)$ where:
      - $v_i^t = P_i^t(ind)$ **if** $i \neq i^t$.
      - $v_i^t \xleftarrow{\$} [N/Q]$ **if** $i = i^t$.
   (d) Let $\{r_i\}_{i \in Q} \xleftarrow{\$} [N]^Q$.
   (e) For $i \in Q$, let $P_i^{t+1} = P_i^t$ except we swap the values of $P_i^t(r_i)$ and $P_i^t(ind)$ for $i \neq i^t$.

Figure 8: Experiments

Experiment H_0 (cont)

1. **For** $t \in \{1, \ldots T\}$:
   (a) Sample $(P_1^t, \ldots, P_Q^t) \xleftarrow{\$} (\mathcal{P}_N)^Q$.
   (b) Adversary outputs $x^t = (i^t, j^t) \in ([Q] \times [N/Q])$.
   (c) Find *ind* such that $P_{i^t}^t(ind) = j^t$
   (d) Output $(v_1^t, \ldots, v_Q^t)$ where:
      - $v_i^t = P_i^t(ind)$ **if** $i \neq i^t$
      - $v_i^t \xleftarrow{\$} [N/Q]$ **if** $i = i^t$

We claim that Experiment 0 and Experiment H_0 are indistinguishable. For each step, we sample fresh permutations, so consider each step independently. Now, consider the distribution of v_i^t for $i \in Q, t \in T$. Since our permutations are sampled uniformly, and each P_i^t for $i \neq i^t$ is independent from P_{i^t}, every $P_i(ind)$ is uniformly distributed over $[N/Q]$, for $i \in Q, i \neq i^t$. Then, it follows that for $i \in Q, i \neq i^t$, v_i^t is uniformly distributed. Now, also, by definition $v_{i^t}^1$ is uniformly distributed. We have shown that for any step $t \in [T]$, any $i \in [Q]$, v_i^t is distrubuted uniformly and so it since the outputs of both experiments have the same distribution at each step, it follows that Experiment H_0 and Experiment 0 are indistringuishable.

Then, consider the following hybrid:

Experiment H_1

Public Parameters: $N, Q \in \mathbb{N}$, assume $Q|N$.

Experiment:

1. **For** $t \in \{1, \ldots T-1\}$:
   (a) Sample $(P_1^t, \ldots, P_Q^t) \xleftarrow{\$} (\mathcal{P}_N)^Q$.
   (b) Adversary outputs $x^t = (i^t, j^t) \in ([Q] \times [N/Q])$.
   (c) Output $(y_1^t, \ldots, y_Q^t)$ where $y_i^t = P_i^t(x^t)$.
2. Sample $(r_1^{T-1}, \ldots r_Q^{T-1}) \xleftarrow{\$} ([N/Q])^Q$.
3. Let $P_i^T = P_i^{T-1}$ except we swap the values of $P_i^{T-1}(r_i^{T-1})$ and $P_i^{T-1}(x^{T-1})$ for each $i \in [Q], i \neq i^t$.
4. Adversary outputs $x^T = (i^T, j^T)$.
5. Find *ind* s.t. $P_{i^T}^T(ind) = j^T$
6. Output $(v_1^T, \ldots, v_Q^T)$ where:
   - $v_i^T = P_i^T(ind)$ **if** $i \neq i^T$.
   - $v_i^T \xleftarrow{\$} [N/Q]$ **if** $i = i^T$.

.

Notice that for the first $T-1$ steps of the experiment, it runs exactly as H_0, so up to that point they are indistinguishable. The only difference is how we sample each P_i^T. In Experiment H_0, it is sampled uniformly at random, whereas in Experiment H_1, it is sampled by taking each P_i^{T-1}, swapping the only element shown of P_i^{T-1} with a uniform random point and denoting this new permutation as P_T. Notice that, by the indistinguishability of the Show and Shuffle experiment (Lemma 3.1), we can see that each the set of P_i^T in both experiments is identically distributed. Then, it follows directly that Experiment H_0 and Experiment H_1 are indistinguishable.

Now, we more generally define experiment H_k as follows, $k \in \{1,\dots,T-1\}$:

Experiment H_k

Public Parameters: $N,Q \in \mathbb{N}$, assume $Q|N$.

Experiment:

1. **For** $t \in \{1,\dots T-k\}$:
   (a) Sample $(P_1^t,\dots,P_Q^t) \xleftarrow{\$} (\mathcal{P}_N)^Q$.
   (b) Adversary outputs $x^t = (i^t, j^t) \in ([Q]\times[N/Q])$.
   (c) Output $(y_1^t,\dots,y_Q^t)$ where $y_i^t = P_i^t(x^t)$.
2. **For** $t \in \{T-k+1,T\}$
   (a) Sample $(r_1^{t-1},\dots r_Q^{t-1}) \xleftarrow{\$} ([N/Q])^Q$.
   (b) Let $P_i^t = P_i^{t-1}$ except we swap the values of $P_i^{t-1}(r_i^{t-1})$ and $P_i^{t-1}(ind^{t-1})$ for each $i \in [Q]$.
   (c) Adversary outputs $x^t = (i^t, j^t)$.
   (d) Find ind^t s.t. $P_{i^t}^t(ind^t) = j^t$
   (e) Output $(v_1^t,\dots,v_Q^t)$ where:
      - $v_i^t = P_i^t(ind)$ **if** $i \neq i^t$.
      - $v_i^t \xleftarrow{\$} [N/Q]$ **if** $i = i^t$.

.

Notice that for every k, we can show that H_k is indisinguishable from H_{k-1} by the same argument above. The only difference between H_k and H_{k-1} is the $k-$th step, where instead of sampling a fresh random permutation at step $T-k+1$, we use a swapped version of the permutation sampled in the last step. Since distinguishing between H_k and H_{k-1} is exactly equivalent to breaking the Show and Shuffle experiment, we can conclude that this holds for every $k \in \{1,\dots,T-1\}$.

We define H_{T-1} explicity below. After $T-1$ hybrids (where each H_{k-1} and H_k are indistinguishable by the Show and Shuffle lemma), we only sample a permutation once, and swap at each step thereafter (we rearrange slightly for ease of reading):

Experiment H_{T-1}

Public Parameters: $N,Q \in \mathbb{N}$, assume $Q|N$.

Experiment:

1. Sample $(P_1^t,\dots,P_Q^t) \xleftarrow{\$} (\mathcal{P}_N)^Q$.
2. Adversary outputs $x_1 \in [N]$.
3. **For** $t \in \{1,\dots,T\}$
   (a) Adversary outputs $x^t = (i^t, j^t) \in ([Q]\times[N/Q])$.
   (b) Find ind^t s.t. $P_{i^t}^t(ind^t) = j^t$.
   (c) Output $(v_1^t,\dots,v_Q^t)$ where:
      - $v_i^t = P_i^t(ind)$ **if** $i \neq i^t$.
      - $v_i^t \xleftarrow{\$} [N/Q]$ **if** $i = i^t$.

   .

   (d) Sample $(r_1^t,\dots r_Q^t) \xleftarrow{\$} ([N/Q])^Q$.
   (e) Let $P_i^{t+1} = P_i^t$ except we swap the values of $P_i^t(r_i^t)$ and $P_i^t(ind^t)$ for each $i \in [Q]$.

Notice that Experiment H_{T-1} and Experiment 1 are the same, except for the reordering of when each P_i^1 is sampled and therefore they are indistinguishable. We conclude that Experiment 1 and Experiment 0 are perfectly indistinguishable. ■

C More Benchmarks

In this section, we include benchmarks for the same tests as those already performed, however, normalizing by *number of operations performed by the server online*, or in other words, the number of elements the online server has to read. In this case, for both static and dynamic cases, we will see that SinglePass achieves 50-100x better preprocessing time and approximately 80x better storage across the board, with similar query time. The price we pay is that the query bandwidth with comparison to MIR and Checklist is much increaased. However, with query sizes hovering around 150KB-3MB, we find that it still is not an impediment for usage, since 3MB is the size of an average web page. We provide the charts in Figure 9 and Figure 10. As seen in Section 4 and Section 6, we can decrease query bandwidth and query time by using more storage.

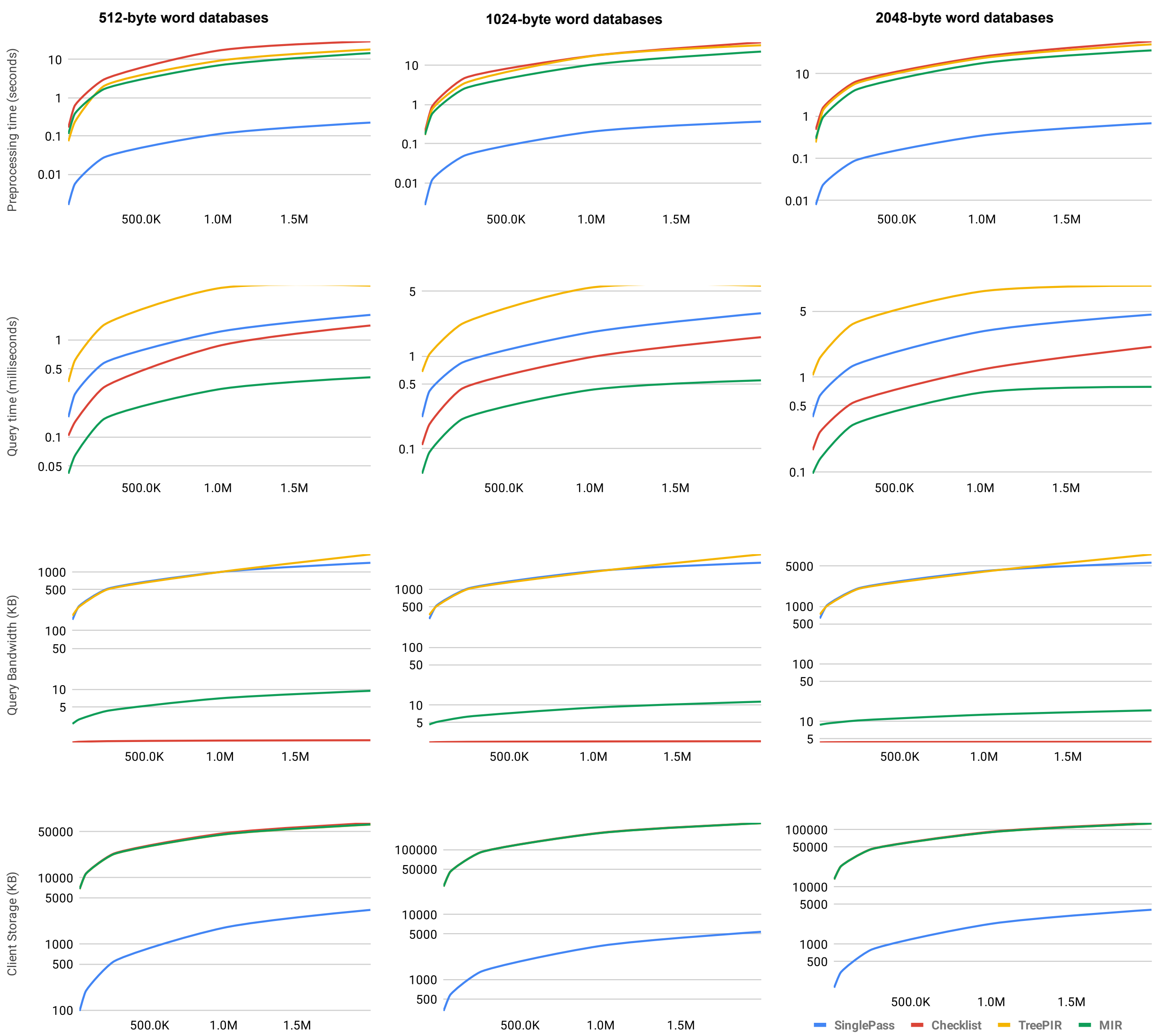


Figure 9: Comparison of benchmarking over preprocessing time, query time, bandwidth and client storageover increasingly sized static databases (x-axis) for different element sizes (on a log scale) when fixing query time.

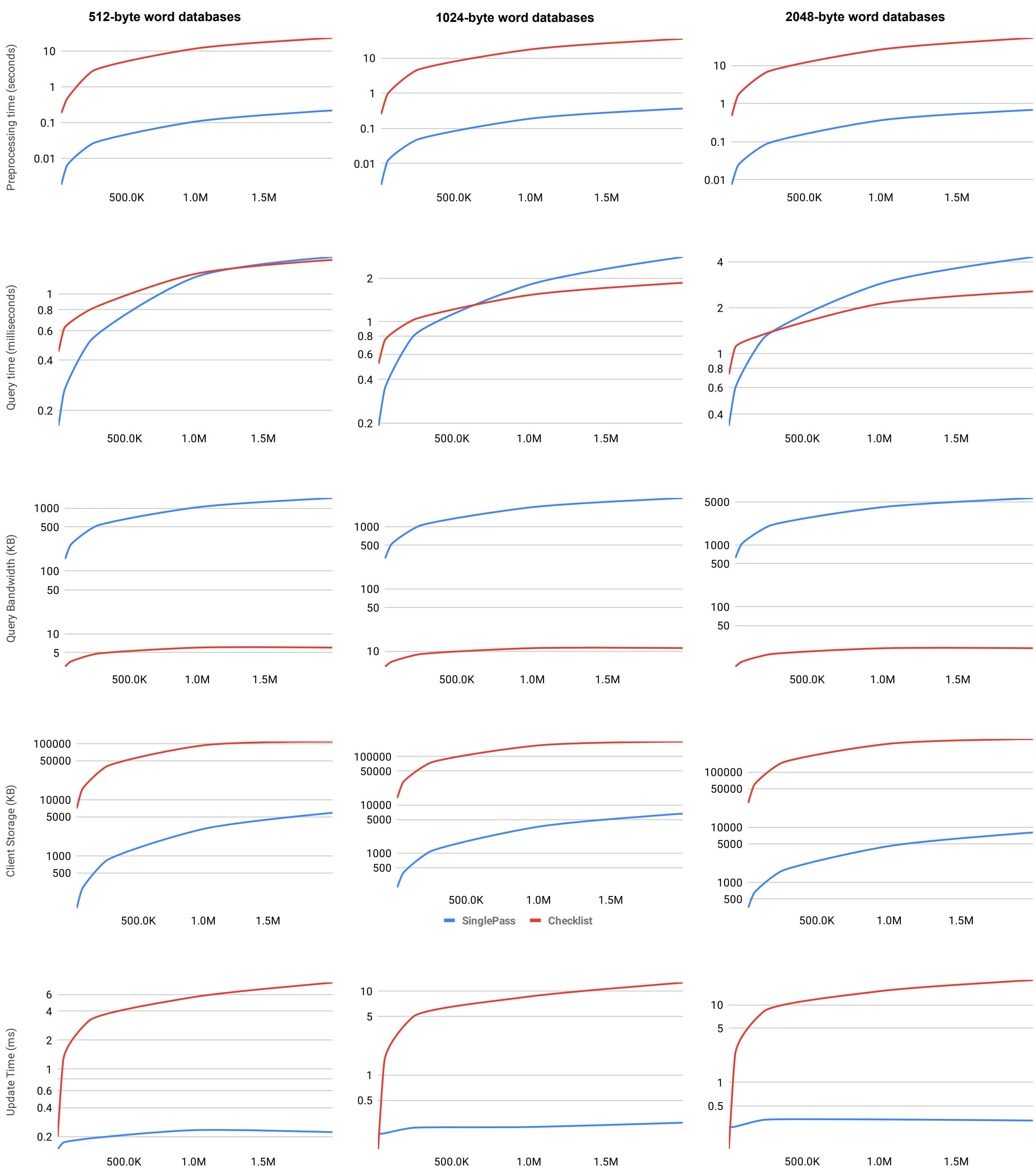


Figure 10: Comparison of benchmarking over preprocessing time, query time, bandwidth, client storage and update time over increasing updatable databases sizes (x-axis) for different element sizes (on a log scale) for fixing query time.