Single Pass Client-Preprocessing Private Information Retrieval

Arthur Lazzaretti
Yale University

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 run in time sublinear in the database size. However, the preprocessing phase is expensive—proportional to $\lambda \cdot N$, where λ is the security parameter (e.g., $\lambda = 128$).

In this paper we propose SinglePass, the first PIR protocol that is concretely optimal with respect to client-preprocessing, requiring exactly a *single* linear pass over the database. Our approach yields a preprocessing speedup ranging from $45\times$ to $100\times$ and a query speedup of up to $20\times$ when compared to previous state-of-the-art schemes (e.g., Checklist, USENIX SECURITY 2021), making preprocessing PIR more attractive for a myriad of use cases that are "session-based".

In addition to practical preprocessing, SinglePass features constant-time updates (additions/edits). Previously, the best known approach for handling updates in client-preprocessing PIR had 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 $20\times$ over previous state-of-the-art updatable schemes (e.g., Checklist).

1 Introduction

Private Information Retrieval (PIR), as defined by Chor et al. [8] 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 $i \in \{0,\dots,N-1\}$. 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 [3, 4, 15, 17, 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. [5] for a single query, posing a significant limitation for practical adoption.

Sublinear PIR through preprocessing. Due to the above limitation, Beimel et al. [5] 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 due to large hidden constants (e.g., [5, 24]) or the use of trusted setup [7, 19].

In search for more practical schemes with sublinear server time, Corrigan-Gibbs and Kogan [9] 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. Interestingly, in this model 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. 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], can answer a PIR query on 3 million entries in less than a millisecond!

At a high level, Checklist (and all prior client-preprocessing PIR schemes [9, 21, 23, 30, 34]) 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 $S_1, S_2 \dots,$. Each S_i is a subset of $\{0,\dots,N-1\}$ and has size $\sqrt{N}$. These sets are sent to Server 0. Server 0 computes the parities/hints $p_1, p_2 \dots,$ 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

[1] Since Checklist sets do not have a fast membership testing mechanism,

Table 1: **SinglePass** compared to Checklist for PIR on updatable databases of N w-bit elements. $Q \in [N]$ is a parameter of our scheme.

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	$O(N)$	$O(Q)$	$O(Q \cdot w)$	$O(N \log N + (N/Q)w)$	$O(1)$

computes the parity p of $S_i \setminus \{x\}$ and sends p to the client. Then the client can retrieve $\mathsf{DB}[x]$ as $p \bigoplus p_i$. This basic outline can be extended to provide support for unlimited queries.

Limitation of client-preprocessing PIR schemes. The informal description above shows that the preprocessing phase of existing PIR schemes requires $\lambda\sqrt{N} \cdot \sqrt{N} = \lambda \cdot N$ database accesses. For $\lambda = 128$, preprocessing will have to access the entire database roughly 128 times before a single query can be issued! (The reason $\lambda\sqrt{N}$ sets are required is to ensure that one of these subsets contains any index with overwhelming probability.) For use cases where the database is dynamic, or the number of queries is small, such slow preprocessing can be hard to justify and leads to particularly slow implementations.

1.1 SinglePass

We propose SinglePass, the first client-preprocessing PIR scheme whose preprocessing is exactly one linear pass over the database, operating on each element exactly once— this is asymptotically optimal [5]. Using SinglePass, the first query requires a linear pass (namely the preprocessing) over the database. Subsequent queries run extremely fast, for the duration of a "session", after which we can potentially delete the state. In this way preprocessing becomes much more economical for applications that require a small amount of queries. In Table 1, we provide a table with the comparison between the asymptotics of SinglePass and Checklist. In addition, our SinglePass scheme also boasts of perfect correctness, and client storage size *independent* of λ.

Constant-time updates in SinglePass. SinglePass is the first protocol to support updates of the preprocessed client state in constant time. Previously, client-preprocessing schemes could only support edits and additions by using $O(\lambda\sqrt{N})$ time per update. Ma et al. [25] and Kogan et al. [21] 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 [14] to support edits and additions in $O(\log N)$ amortized time per operation, at the cost of an additional $\log N$ overhead in query bandwidth. Our new construction's technique allows us to naturally defineEdit and Add algorithms for our preprocessed client state that 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 wish to update it sporadically.

Evaluation. We implement SinglePass and show that it performs very well in comparison to previous schemes. In specific, we achieve preprocessing speeds ranging from 45-100x compared to all previous schemes, as well as a 10x speed up in query times when using comparable storage to other schemes, for the tests we benchmarked. Our scheme also provides, empirically, a speed up of approximately 20x in update time.

On linear client storage. Just like Checklist, SinglePass suffers from linear client storage. However, because there is no dependency on λ, SinglePass's client storage is only worse than the client storage of previous schemes [9, 21–23, 30] for N greater than one billion elements (This is for a word size of 1024 bytes.) In practice, databases with a size on the order of a million elements encompass a large array of PIR use cases, including private blocklists [21], metadata-hiding communication [2, 3], private movie streaming [15], private wikipedia [27], among others. This is the size of databases we focus on mainly in this work.

SinglePass working example. Below, we go through a short example of a preprocessing and query step of our scheme with visuals for intuition. Throughout the paper, we use

$$\alpha \xleftarrow{\$} \mathbb{S}$$

to indicate α being sampled uniformly at random from the set $\mathbb{S}$. For natural N, we use $[N]$ to denote the set $\{0, \ldots, N-1\}$ and $\mathcal{P}_N$ to denote the set of all permutations of $[N]$.

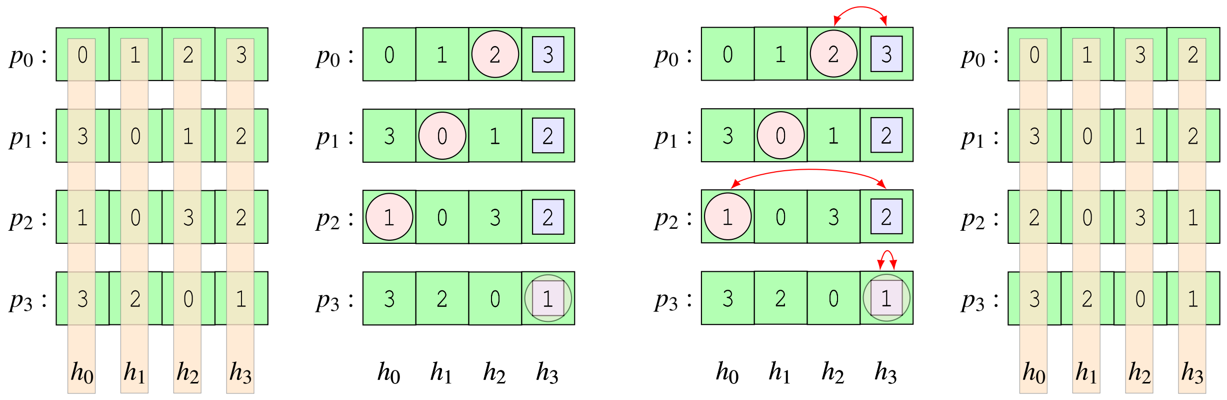
Let now $N = 16$ and $Q = 4$, where Q is a tunable parameter for the set size. Again, we work with two, non-colluding servers both storing DB. First, we organize DB as Q arrays of $N/Q = m$ elements. Let DB_i be the array $\mathsf{DB}[i \cdot m : (i+1) \cdot m]$.

We first describe how preprocessing works in SinglePass: 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 in Figure 1(a).

This picture shows what elements are contained in each hint given the permutations sampled by Server 0 were $p_0, \ldots, p_3$.

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.



(a) Initial state. (b) Chosen elements. (c) Swaps. (d) Final state.

Figure 1: Figures to aid our working example.

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 in Figure 1(b).

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} :	3	$\cancel{2} \to r$	2	1

S_{refresh} :	2	0	1	1

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.

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 client has now enough information to both:

1. Recover $\mathsf{DB}_1[2]$ by simply taking the exclusive or of 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 depicted in Figure 1(c).

Since we have the value for each element on the permutation we wish to swap, we can update the hints accordingly by using exclusive or operations. 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, our new refreshed state can be seen in Figure 1(d).

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.

2 PIR Preliminaries and Definitions

SinglePass works in the two-server model, where a database of N bits is replicated in two servers and at least one server behaves honestly. SinglePass privacy holds for any adversary controlling either server and any number of clients, but does not capture an adversary controlling both servers. This can be realized in practice by having the servers be run by different parties. We now give the definition of client-preprocessing PIR (We use a slightly modified definition from the initial works on offline/online PIR [9, 21].)

Definition 2.1 (Client-Preprocessing PIR [21])**.** *A multi-query, 2-server client-preprocessing PIR scheme* Π *is a tuple of four polynomial-time algorithms:*

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

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

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

  A randomized algorithm that on input ck *and index* $x \in [N]$ *outputs updated client keys* ck′ *and queries* q_0, q_1.

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

  A deterministic algorithm that on input DB *a query* q_b *outputs an answer* A_b.

- $\underline{\mathsf{Reconstruct}(\mathsf{ck}, h, A_0, A_1) \to (h', y):}$

  A deterministic algorithm that on input ck*, h and answers* A_b *outputs an updated hint* h' *and database value y.*

Using the above algorithms we can build a PIR protocol as follows. When a new client connects, Server 0 runs the Hint algorithm, and outputs the client keys and hint, which are returned to the client. After this, the client can use the client keys and query index 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 are then used by the client along with the hints to reconstruct $\mathsf{DB}[x]$, and update the hints for the next query.

Correctness. Correctness 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 $x_1, \dots x_t$ it desires to make. Correctness only needs to hold for honest servers, as opposed to privacy for which we will consider malicious servers. We given the formal definition below.

Definition 2.2 (Correctness)**.** *A multi-query 2-server client-preprocessing PIR* $\Pi = (\mathsf{Hint}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Reconstruct})$ *is **correct** if, for any* $\lambda, w, N, T \in \mathbb{N}$*, every* $\mathsf{DB} \in (\{0,1\}^w)^N$ *and every* $x_0, \dots, x_{T-1} \in [N]^T$*, the honest execution of the following game always outputs "1":*

- *Compute* $(h, \mathsf{ck}) \leftarrow \mathsf{Hint}(\mathsf{DB})$.
- *For* $t = 0, \dots, T-1$*, compute:*
  - $(\mathsf{ck}, q_0, q_1) \leftarrow \mathsf{Query}(\mathsf{ck}, x_t)$.

PrivGame^0
(Privacy for Server 0)

- $b \xleftarrow{\$} \{0,1\}$.
- $(\mathsf{ck}, \_) \leftarrow \mathsf{Hint}(\mathsf{DB})$.
- $\mathsf{st} \leftarrow \mathcal{A}(1^\lambda, \mathsf{ck})$.
- For $t = \{0, \dots, T-1\}$:
  - $(\mathsf{st}, x_0, x_1) \leftarrow \mathcal{A}(\mathsf{st})$.
  - $(\mathsf{ck}, q_0, \_) \leftarrow \mathsf{Query}(\mathsf{ck}, x_b)$.
  - $\mathsf{st} \leftarrow \mathcal{A}(\mathsf{st}, q_0)$.
- $b' \leftarrow \mathcal{A}(\mathsf{st})$.
- 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 $t = \{0, \dots, T-1\}$:
  - $(\mathsf{st}, x_0, x_1) = \mathcal{A}(\mathsf{st})$.
  - $(\mathsf{ck}, \_, q_1) \leftarrow \mathsf{Query}(\mathsf{ck}, x_b)$.
  - $\mathsf{st} \leftarrow \mathcal{A}(\mathsf{st}, q_1)$.
- $b' \leftarrow \mathcal{A}(\mathsf{st})$.
- Output $b = b'$.

Figure 2: Privacy games for PIR.

  - *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".*

Privacy. Recall that Server 0 sees runs the preprocessing and sees q_0, for each queried index x. Also, Server 1 sees only q_1, for each query x, and does not run the preprocessing. Our privacy game models that under this behavior, no information is leaked to either server about x, even for a sequence of adaptively-chosed queries. Note that our assumption of no collusion allows us to prove that the information received by each server individually is independent from the issued queries issued—but there are no guarantees the joint view. We formalize this intuition in the definition below.

Definition 2.3 (Privacy)**.** *A multi-query 2-server client-preprocessing PIR* $\Pi = (\mathsf{Hint}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Reconstruct})$ *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 PrivGame^0 *and* PrivGame^1 *are defined in Figure 2.*

We reiterate that the definitions in this subsection are only slightly modified from the initial works on Offline/Online PIR [9, 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.

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.

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. [2] Sampling pseudorandom permutations over small domains (for an adversary that can query the whole permutation) is a well-studied problem with a long line of work [18,26,28,29,31,32]. To date, we do not have representations of small domain permutations that are fast empirically and also succinct (like we have AES for larger domains). Therefore, in SinglePass we sample random permutations using the Fisher-Yates shuffle, also known as the Fisher-Yates-Durstenfeld-Knuth shuffle [10, 11, 20], which outputs an unbiased permutation over $[N]$ in $O(N)$ time, but requires storing the whole permutation explicitly after generating it to evaluate the permutation in sublinear time. We define this permutation in the following lemma.

Lemma 2.1 (Fisher-Yates Shuffle [10, 11, 20])**.** *For any positive integer N, there exists an algorithm 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.*

We can naturally define a computational version of the Fisher-Yates shuffle by using a PRG and a random seed to sample all the randomness used in the protocol—this is what we use in SinglePass. Using a seeded PRG allows us to concisely represent the permutation, but 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 in practice—however we still use the seed of the PRG to communicate the permutation to the server efficiently, i.e., without sending the whole permutation to the server.

3 Show and Shuffle Experiment

Before we present the SinglePass protocol in detail, we abstract a key concept necessary for SinglePass to work, through an experiment called *Show and Shuffle*—see Figure 3. The Show and Shuffle experiment captures the single-query security of SinglePass.

The experiment first samples L permutations $(P_1, \ldots, P_L)$ over $[K]$ uniformly. Then 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 is chosen uniformly from $[K]$.

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. Set $\mathbf{v} = (v_1, \ldots, v_L)$ such that 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. Outputs 1 iff $b' = b$.
```

Figure 3: Show and Shuffle experiment.

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. To swap P_i into P_i', the experiment samples a uniform r_i from $[K]$ and *swaps* the outputs of $P_i(j)$ and $P_i(r_i)$. We do not perform a swap for P_ℓ. Show and Shuffle models exactly one query of SinglePass, and will help us show that after each query, the resulting client hint is uniform. Looking ahead, to prove the security of SinglePass, we will apply the following Show and Shuffle Indistinguishability lemma (Lemma 3.1) T times.

Lemma 3.1 (Show and Shuffle Perfect Indistinguishability)**.** *For the Show and Shuffle game defined in Figure 3, denoted as SaS, for any adversary $\mathcal{A}$, for any $L, K \in \mathbb{N}$:*

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

[2]With "small", we refer to a domain of size on the order of 2^{20} as opposed to the AES permutation which has a domain size of 2^{128}.

Proof. It is

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

To finish the proof it is enough to show that

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

Note that in the above expression, the only difference in the adversary's view is $\mathcal{R}_b$. Therefore it is enough to show that the distributions of $\mathcal{R}_0$ and $\mathcal{R}_1$ are the same. 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, it is enough to show that the same holds for set $\mathcal{R}_1$. Formally, we have to show that for any $p_1, \ldots, p_L$, where each p_i is a permutation of $[K]$ it is

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

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

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

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 decompose it as follows:

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

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 $\Pr[P'_z = p_z \,|\, j]$. We consider two cases.

For the first case, when $q_j = v_z$ we have

$$\Pr[P'_z = p_z \,|\, j, q_j = v_z] \Pr[q_j = v_z] = \frac{\Pr[P_z = p_z \,|\, j, q_j = v_z]}{K} = \frac{1}{K!}.$$

The first equality holds since $\Pr[q_j = v_z] = \Pr[r_z = j] = 1/K$. The second equaliy 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.

For the second case, when $q_s = v_z$ for some $s \neq j$ we have:

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

where Lines 1 and 2 hold by just opening up the conditioning and Lines 2 and 3 hold because $\Pr[q_s = v_z] = \Pr[r_z = s] = 1/K$. Lines 3 and 4 hold 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'). Lines 4 and 5 hold because $P_z(s)$ cannot equal v_z conditioned on $s \neq j$, but is uniform among the rest of the elements. Finally, Lines 5 and 6 hold 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 the equation we found before, we have that for any $p_1, \ldots, p_L$

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

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

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

This proves our lemma. ■

4 The SinglePass Protocol

In this section, we first present the detailed algorithms of our SinglePass protocol—see Figure 4. Then, we present our main theorem (Theorem 4.1) and its proof. Our proof will be using the result we proved for the Show and Shuffle game—see Lemma 3.1.

We recall that in SinglePass, Server 0 is running Hint(DB), the client runs Query with its index and the keys output by the Hint algorithm and then 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 answer at the queried index and update the 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. The following theorem states our main result.

Theorem 4.1 (Single Pass Client-Preprocessing PIR)**.** *The scheme in Figure 4 is a client-preprocessing Private Information Retrieval scheme as defined in Definition 2.1, with parameters $N,Q,w\in\mathbb{N}$, where $Q|N$, where N is the database size, Q is the query size, and w is the word size, satisifes Correctness and Privacy according to Definitions 2.2 and 2.3 respectively, and has 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.*
- *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 state of $O(N\log N+(N/Q)\cdot w)$ bits.*
- *The server stores only DB.*

We prove Theorem 4.1 in the Appendix A. The crux of our proof is the Show and Shuffle experiment (Figure 3). We define a sequence of hybrids between our experiment and an idealized experiment, and can show each pair of this sequence to be indistinguishable by using Lemma 3.1.

```
SinglePass

Public Parameters: 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 $\mathsf{DB}_i=\mathsf{DB}[i\cdot m:(i+1)m]$.

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

Query(ck, $x=(i^*,j^*)\in[Q]\times[m]$):
  1. Parse $x=(i^*,j^*)$. Find $ind\in[m]$ s.t. $p_{i^*}(ind)=j^*$.
  2. Let $S=[p_j(ind):j\in[Q]]$. Set $S[i^*]=r^*\xleftarrow{\$}[m]$.
  3. Sample $r_0,\dots,r_{Q-1}$ 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 DB$[x]$, ck, $h$.
```

Figure 4: SinglePass algorithms.

5 SinglePass Evaluation

In this section, we present the evaluation of SinglePass. SinglePass is implemented in about 300 lines of Go code and 150 lines of C code, as an extension to the existing PIR frame-

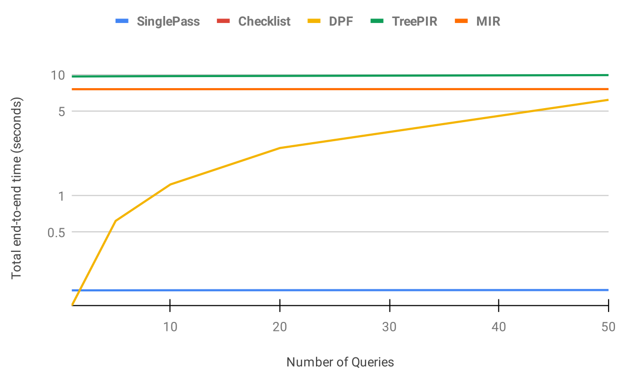


Figure 5: Comparison of end-to-end time for varying number of queries over a database of one million 512-byte elements (log scale).

work from Checklist [21]. Our code can be found in [1]. We compare SinglePass against Checklist [21], TreePIR [23] and MIR [30], three novel state-of-the-art client preprocessing PIR schemes, for a variety of parameters. Benchmarks are all run on an AWS EC2 instance of size t2.2xlarge, on a single thread. We recall that previous works have dependency on the security parameter for preprocessing and client storage.[3]

Here, along with the state-of-the-art client-preprocessing schemes, we also include comparisons against the state-of-the-art two-server PIR scheme with no preprocessing from [6, 12, 16], which we will denote as DPF.

The choice of parameters picked throughout the section reflect a sample use case of a private encyclopedia service, where a user would browse a private encyclopedia website, access a couple of articles privately, and afterwards leave and delete the local state. In this scenario, fast preprocessing is crucial so that the user does not have to wait too long to access the website. In our evaluations we include two sets of benchmarks.

First, we measure end-to-end time of SinglePass compared to end-to-end time of Checklist, TreePIR, MIR and DPF for a database of one million 512-byte elements. This is shown in Figure 5. (The line of TreePIR is hidden underneath the line for Checklist.) From various database sizes and entry sizes we found for encyclopedias online (up to two million entries of size 512 to 2048 bytes) we arbitrarily picked this one to be representative. We however note that the trend seen in Figure 5 holds across any set of parameters. This is 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.

[3]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, 34].

Second, in Figure 6, we provide a more fine-grained comparison of preprocessing time, per-query time, and bandwidth between SinglePass, Checklist,TreePIR and MIR, through the whole range of parameters mentioned above (We do not include DPF since it does not have preprocessing.) 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—therefore we can benchmark the performance of the schemes when given similar client resources. In this scenario we observe that SinglePass provides very favorable trade-offs in preprocessing time and query time, at a modest bandwidth cost with respect to MIR and Checklist. In Appendix B, we also include tests normalized by query time. In those tests, all schemes have very similar query times, and the other metrics vary. We discuss this further in Appendix B.

6 Extending SinglePass to Dynamic Databases

In this section we study how to slightly modify SinglePass to support updates in the database. In particular we wish to support edits, additions and deletions. Recall that SinglePass's hint consists only of a single (partitioned) permutation of the database. Therefore we can direclty update the hint data structure in $O(1)$ time, without any other additional overhead. Below, we give some intuition of how our updates work, and subsequently provide formal algorithms.

Edits/Deletions. To edit the value of index $x = (i, j) \in [Q] \times [N/Q]$, we compute $k = p_i^{-1}(j)$ and this indicates 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 updates the hint by simply setting $h_k = h_k \oplus \mathsf{DB}[x]_{old} \oplus DB[x]_{new}$, after computing 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 in 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 compute the exclusive or of this element and the appropriate hint (which we find by checking the inverse), giving us constant amortized time for addition. This can be deamortized by sampling a constant part of the permutation at each step.

The attentive reader will notice that the addition 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

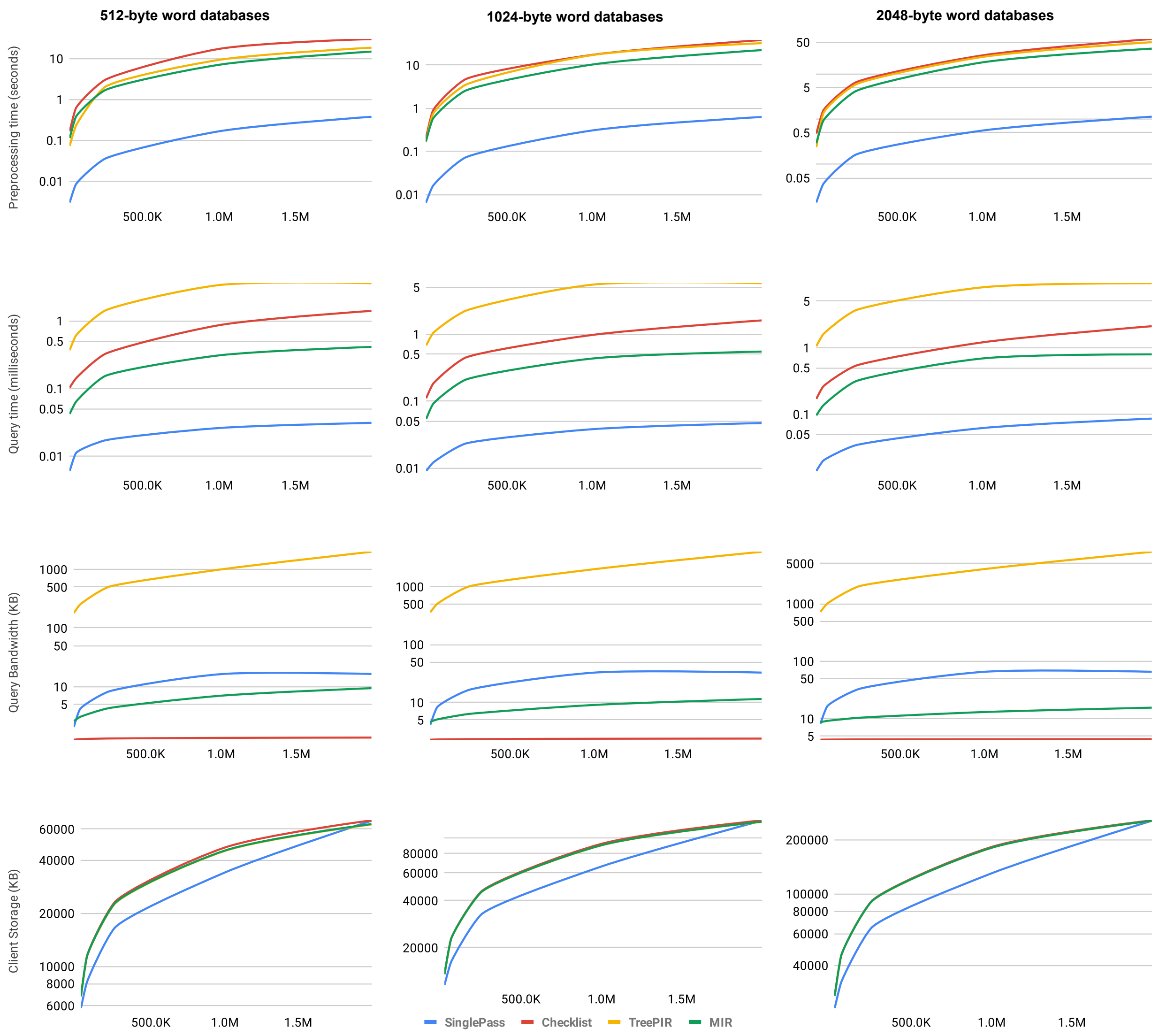


Figure 6: Comparison of benchmarking over preprocessing time, query time, bandwidth and client storageover increasingly sized static databases (x-axis) for different element sizes (on a log scale).

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 a desired value while maintaining a constant update cost (since this is performed only every N updates). This step can also be de-amortized by running a constant amount of steps at each update. We give the full pseudocode for Edit and Add in Figure 7.

Previous work on dynamic PIR. Previous works in preprocessing PIR do not handle updates naturally in constant time. At a high level, this is because their preprocessing step involves sampling around $\lambda\sqrt{N}$ independent subsets of $[N]$. Updates then require checking each subset. To address this, previous works such as Checklist [21] and TreePIR [23] borrowed techniques from hierarchical ORAM [13] that 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, a transformation from standard PIR to *keyword* PIR with indices as the keys [8] and a blackbox 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 a logarithmic number of databases of exponentially-increasing size , and as such we need a way to maintain the original indices.[4] Each of these steps adds some bandwidth overhead, communication, client and server time. Other approaches have been studied [25] but also incur significant overheads in comparison to the static preprocessing scheme.

Evaluation for dynamic databases. We implement the update algorithms for SinglePass in an additional 200 lines of Go code and 50 lines of C code. We run a complete test suite in Figure 8, with the same parameters as we picked in Section 5. All other dymamic schemes that we use in our comparison use the waterfall update approach as we discussed before. Unlike in Section 5, we do not include the DPF scheme in our benchmarks, since these use cases assume the client will run the preprocessing only once, or at least not very often. This could reflect a mobile app for the private encyclopedia service envisioned in Section 5, where it is fine to use some permanent storage. In this scenario, minimizing update times is crucial so as to impose as small as a burden as possible on the servers.

In Figure 8, we show 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. Preprocessing time, query time and query bandwidth, follow mostly the same patterns identified in Section 5, with an

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

```
Edit(ck, $h$, $i$, DB$[i]_{old}$, 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 7: Update Operations for SinglePass.

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 bandwidth which is at most 1.5× Checklist's bandwidth, while maintaining a query time reduction of roughly 20× on average across all experiments and a preprocessing time speed-up of up to 100×. As in Section 5, we pick Q for SinglePass accordingly, in order to benchmark both SinglePass and Checklist using comparable client storage. We include tests that fix query time in Appendix B.

In Table 2, we also provide a benchmark of SinglePass and Checklist for the blocklist application studied in Checklist, with the parameters picked in the original Checklist paper: a database of 3 million 32-byte elements that is updated in batches of 500. Our scheme achieves over 100× speed-up in preprocessing, over 47× speed-up in query time, an approximate 2× saving in bandwidth and a 19× 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 amounts to 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 $20\times$ is due to the $O(\log N)$ additional overhead. [5]

Acknowledgements

This research was funded by Protocol Labs, NetApp and the National Science Foundation.

References

[1] Code repository for SinglePass PIR. Available at: https://github.com/SinglePass712/Submission.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

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

[5] One caveat of the comparison is it is necessary for the updatable version of Checklist to support keyword queries in order to achieve the $O(\log N)$ amortized bandwidth. SinglePass is a pure PIR scheme that only supports index queries. However, using cuckoo hashing we can derive a keyword PIR scheme with a $2\times$ overhead [33].

[9] 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.

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

[11] 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.

[12] 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.

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

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

[15] 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.

[16] 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.

[17] 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.

[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.

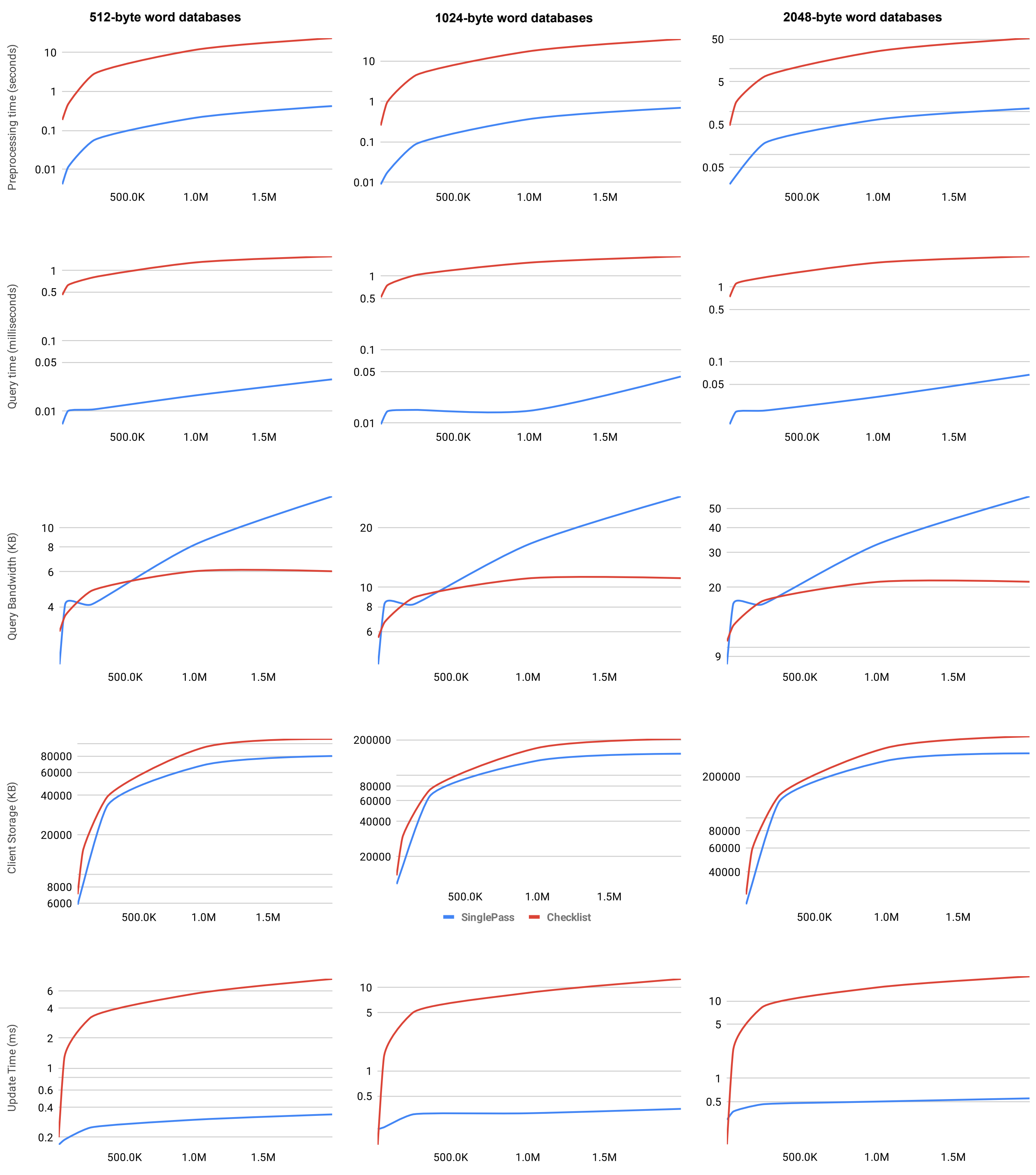


Figure 8: Comparison of benchmarking over preprocessing time, query time, bandwidth, client storage and update time over increasing updatable databases sizes (x-axis) for different element sizes (on a log scale).

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.

[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 pseudorandom permutations for small domains. Cryptology ePrint Report 2012/254. Technical report, 2012.

[33] 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.

[34] 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 prove Theorem 4.1

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 $p_{i^*}^{-1}(j^*)$.[6] After, we just have to index $p_i(ind)$ for $i \in [Q]$ which takes $O(Q)$ time, and send that to the server (with a symmetric number of operations to generate the refresh hint).

[6]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.

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 2.

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 A.1. Note that Experiment 1 in Theorem A.1 is exactly equivalent to our PIR query at each timestep. Then, by Theorem A.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:

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

■

A.1 Server 1 Indistinguishability

Theorem A.1 (Query indistinguishability). *For any adaptive adversary $\mathcal{A}$, Experiment 0 and Experiment 1, as defined in Figure 9, are perfectly indistinguishable.*

Proof. We prove this through a series of hybrid experiments. Note that at each step t, the adversary can pick inputs and then see the outputs for that step before deciding its inputs for the next step. We start from H_0, in which at each step, rather than simply sampling uniformly as in Experiment 0, the experiment samples Q independent permutations of $[N/Q]$ uniformly. and behaves by using the adversary inputs to index these permutations. Since they permutations are uniformly and independently sampled, these are indistinguishable. We expand on this below.

Experiment 0

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

Experiment:

```
1. For $t \in \{0, \dots T-1\}$ :
    (a) Adversary outputs $x^t = (i^t, j^t) \in ([Q] \times [N/Q])$.
    (b) Output $(y_1^t, \dots, y_Q^t) \xleftarrow{\$} [N/Q]^Q$.
```

Experiment 1

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

Experiment:

```
1. Sample $P_1^1, \dots P_Q^1 \xleftarrow{\$} (\mathcal{P}_{N/Q})^Q$.
2. For $t \in \{0, \dots, T-1\}$ :
    (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 $S = (v_1^t, \dots, v_Q^t)$ where:
        • $v_i^t = P_i^t(ind)$ if $i \neq i^t$.
        • $v_i^t \xleftarrow{\$} [N/Q]$ if $i = i^t$.
    (d) 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 9: Experiments

Experiment H_0

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

Experiment:

```
1. For $t \in \{0, \dots T-1\}$ :
    (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) Find $ind$ such that $P_{i^t}^t(ind) = j^t$.
    (d) Output $(v_1^t, \dots, v_Q^t)$ where:
```

Experiment H_0 (cont)

```
        • $v_i^t = P_i^t(ind)$ if $i \neq i^t$.
        • $v_i^t \xleftarrow{\$} [N/Q]$ if $i = i^t$.
```

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. By definition $v_{i^t}^1$ is also uniformly distributed. Then, for any step $t \in [T]$, any $i \in [Q]$, v_i^t is distributed uniformly. Since the outputs of both experiments have the same distribution at each step, Experiment H_0 and Experiment 0 are indistinguishable.

Then, consider the following hybrid:

Experiment H_1

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

Experiment:

```
1. For $t \in \{0, \dots T-2\}$ :
    (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. Sample $(r_1^{T-2}, \dots r_Q^{T-2}) \xleftarrow{\$} ([N/Q])^Q$.
3. Let $P_i^{T-1} = P_i^{T-2}$ except we swap the values of
   $P_i^{T-2}(r_i^{T-2})$ and $P_i^{T-2}(x^{T-2})$ for each $i \in [Q], i \neq i^t$.
4. Adversary outputs $x^{T-1} = (i^{T-1}, j^{T-1})$.
5. Find $ind$ s.t. $P_{i^{T-1}}^T(ind) = j^{T-1}$.
6. Output $(v_1^{T-1}, \dots, v_Q^{T-1})$ where:
    • $v_i^{T-1} = P_i^{T-1}(ind)$ if $i \neq i^{T-1}$.
    • $v_i^{T-1} \xleftarrow{\$} [N/Q]$ if $i = i^{T-1}$.
.
```

Notice that for the first $T-1$ steps of the experiment (iterations 0 through $T-2$), 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-1}. In Experiment H_0, it is sampled uniformly at random, whereas in Experiment H_1, it is sampled by taking each P_i^{T-2}, swapping the only element shown of P_i^{T-2} with a uniform random point and denoting this new permutation as P_i^{T-1}. 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-1} in both experiments is identically distributed, since this is exactly the experiment proven in Lemma 3.1. Then, it follows directly that Experiment H_0 and Experiment H_1 are indistinguishable.

Now, 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 $t \in \{0,\dots T-k-1\}$ :
   (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 $t \in \{T-k, T-1\}$ :
   (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 indistinguishable 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 (rearranged for ease of reading):

Experiment H_{T-1}

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

Experiment:

```
1. Sample $(P_1^0,\dots,P_Q^0) \xleftarrow{\$} (\mathcal{P}_N)^Q$.
2. Adversary outputs $x_1 \in [N]$.
3. For $t \in \{0,\dots,T-1\}$ :
   (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. ■

B 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 10 and Figure 11. As seen in Section 5, we can decrease query bandwidth and query time by using more storage.

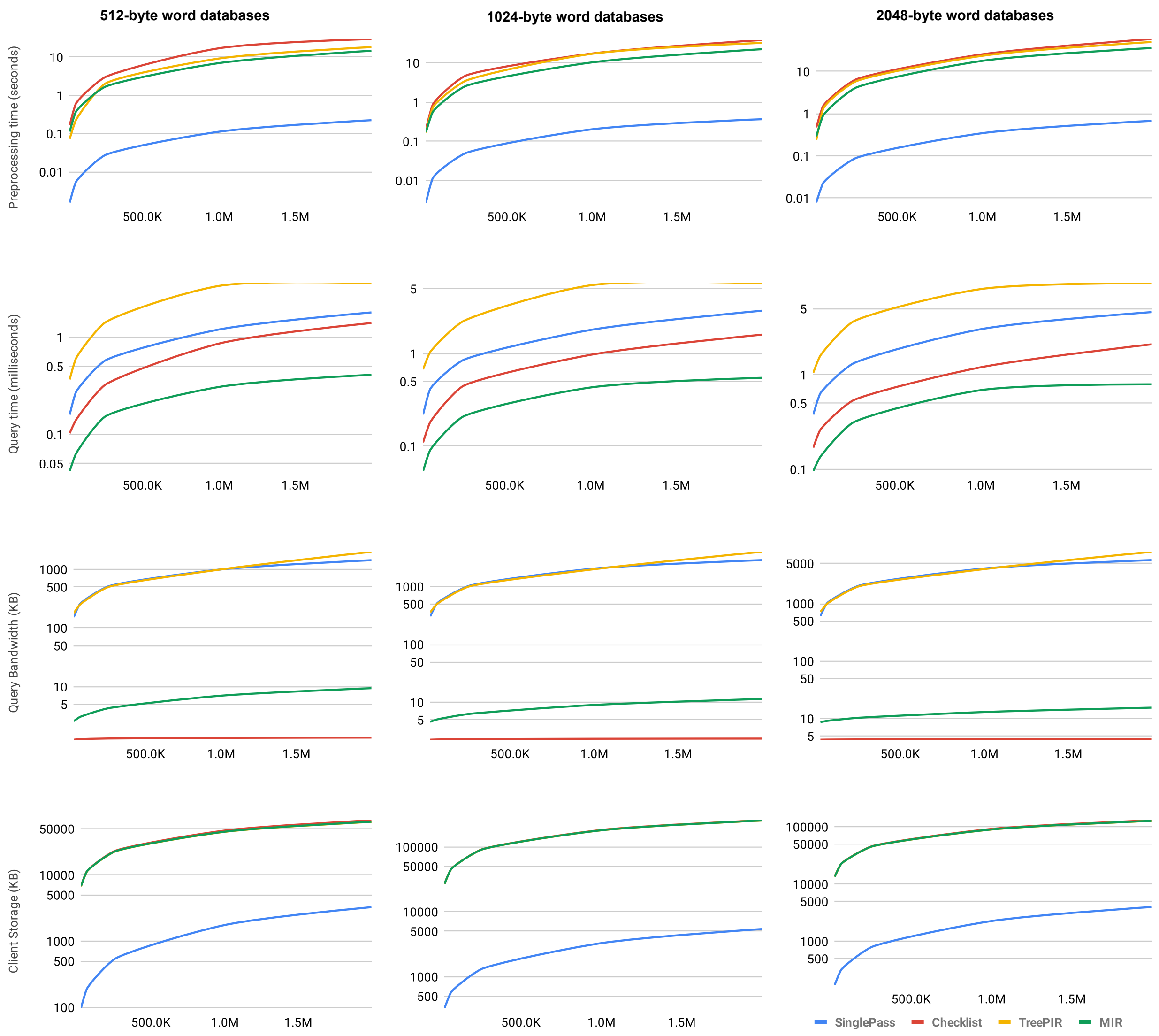


Figure 10: Comparison of benchmarking over preprocessing time, query time, bandwidth and client storageover increasingly sized static databases (x-axis) for different element sizes (on a log scale) when fixing query time.

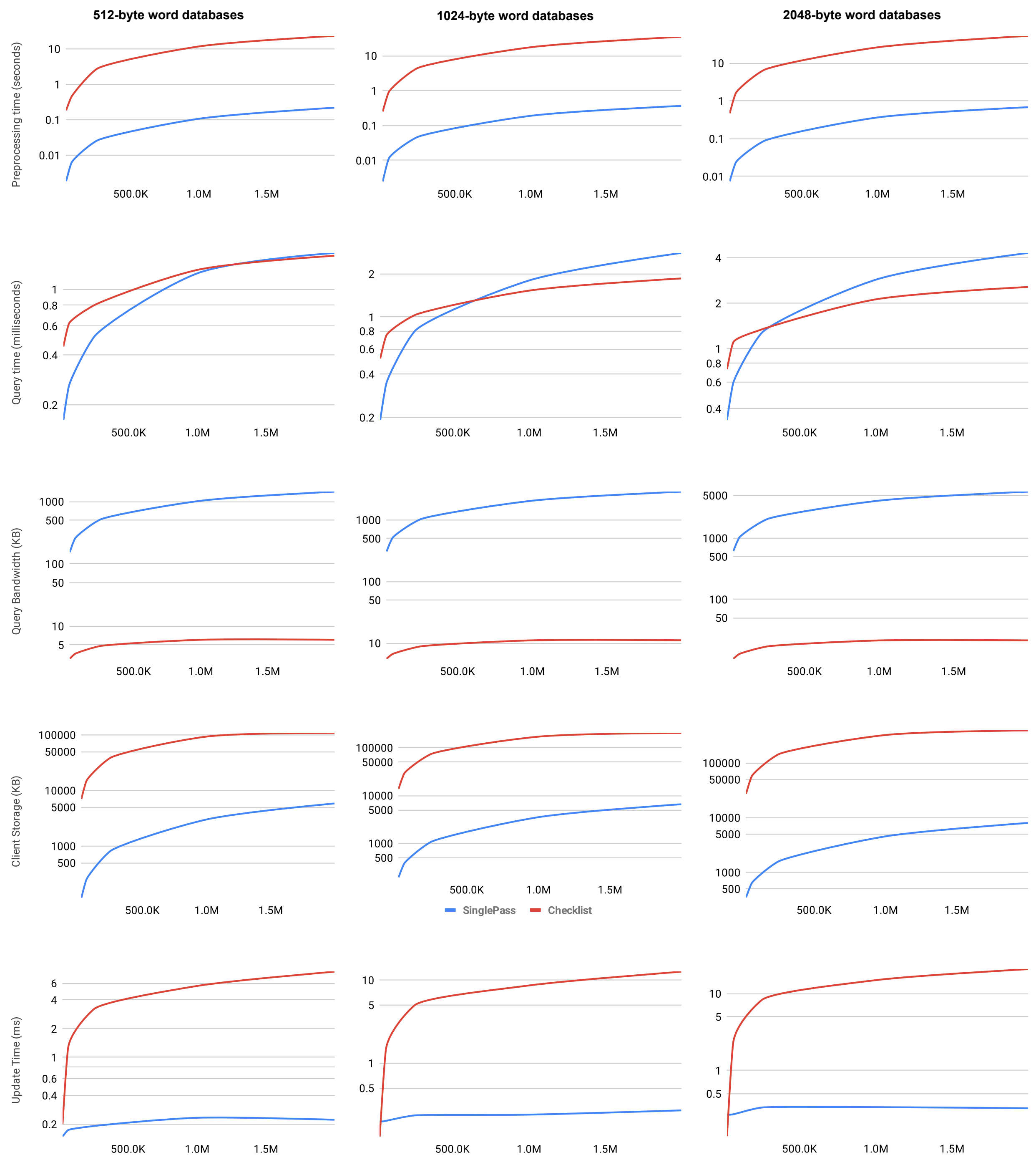


Figure 11: Comparison of benchmarking over preprocessing time, query time, bandwidth, client storage and update time over increasing updatable databases sizes (x-axis) for different element sizes (on a log scale) for fixing query time.