Don't be Dense: Efficient Keyword PIR for Sparse Databases

Sarvar Patel[*] Joon Young Seo[†] Kevin Yeo[‡]

Abstract

In this paper, we introduce SparsePIR, a single-server keyword private information retrieval (PIR) construction that enables querying over sparse databases. At its core, SparsePIR is based on a novel encoding algorithm that encodes sparse database entries as linear combinations while being compatible with important PIR optimizations including recursion. SparsePIR achieves response overhead that is half of state-of-the art keyword PIR schemes without requiring long-term client storage of linear-sized mappings. We also introduce two variants, $\mathsf{SparsePIR}^g$ and $\mathsf{SparsePIR}^c$, that further reduces the size of the serving database at the cost of increased encoding time and small additional client storage, respectively. Our frameworks enable performing keyword PIR with, essentially, the same costs as standard PIR. Finally, we also show that SparsePIR may be used to build batch keyword PIR with halved response overhead without any client mappings.

1 Introduction

Private information retrieval (PIR) [CKGS98] is an important cryptographic primitive that allows a client to retrieve entries from a public database without revealing the client's entry of interest. Due to its strong privacy guarantees, PIR is a critical building block for many practical applications including advertisements [GLM16], anonymous communication [MOT+11, KLDF16, AS16, AYA+21], contact discovery [BDG15, DRRT18], device enrollment [YP21], media consumption [GCM+16], password leak checks [ALP+21] and route navigation [WZPM16].

PIR has been studied in two settings with a single server [KO97, CMS99, Pai99, DJ01, GR05, Lip05, SC07, OI07, GKL10] or multiple, non-colluding servers [CKGS98, Efr09, GI14, DG15, BGI16]. PIR schemes in the multi-server setting are, typically, more efficient. However, they rely on stronger trust assumptions between different organizations that may be difficult to materialize. In our work, we will focus on single-server PIR.

The standard PIR problem considers databases that are n-entry arrays. The client's goal is to retrieve the i-th entry in the array without revealing index i to the server holding the public database. Unfortunately, in most practical applications, databases more closely resemble key-value pairs where users want to retrieve the value associated to a certain key. For example, we can think of data sets like contact lists, videos, websites, etc. To address this, keyword PIR [CGN98] was introduced where databases consist of key-value pairs and a client wishes to retrieve the value associated with a certain key.

[*]Google, `sarvar@google.com`.
[†]Google, `jyseo@google.com`.
[‡]Google and Columbia University, `kwlyeo@google.com`.

	Client Storage	Encoding Size	Response Overhead
Client Mapping	n	n	1x
Cuckoo Hashing [ALP+21]	$O(1)$	$(2+\epsilon)n$	2x
Constant-Weight [MK22]	$O(1)$	n	2-9x
Ours: SparsePIR	$O(1)$	$(1+\epsilon)n$	1x
Ours: SparsePIR^g	$O(1)$	$(1+\epsilon)n$	1x
Ours: SparsePIR^c	$<$ 11.2 KB	$1.03n$	1x

Figure 1: One-round keyword PIR comparison. Response overhead is compared to state-of-the-art standard PIR, Spiral [MW22], that query over dense n-entry arrays.

A naive solution for keyword PIR is to replicate mappings from keys to array indices that need to be stored by all clients. To query, a client uses the mapping to determine the array index storing the entry associated to the query key and uses a standard PIR scheme. Unfortunately, this requires the client to store mappings that are linear in the database size. Chor *et al.* [CGN98] presented a multiple round solution introducing additional overhead. Another solution to keyword PIR utilizes cuckoo hashing [ALP+21] in a single round, but results in 2x response overhead compared to standard PIR. A recent work [MK22] builds one-round keyword PIR using constant-weight equality operators. However, this scheme requires significantly more communication and computation than recent PIR schemes. In the current state of affairs, moving from standard PIR to keyword PIR requires increasing roundtrips, doubling the response size or large long-term client storage. None of these choices are appealing. This paper addresses this inefficiency.

Our Contributions. We present SparsePIR that is a framework for building keyword PIR from a standard PIR. The core technique behind SparsePIR is the usage of encoding algorithms that aim to encode key-value pairs as a function of multiple database entries as opposed to allocating each key-value pair into a single entry as done by prior hashing schemes.

SparsePIR ensures that request and response sizes are identical to the underlying standard PIR schemes without any additional client storage. As a result, SparsePIR halves the response size compared to the cuckoo hashing keyword PIR approach [ALP+21]. The only slight drawback of SparsePIR is a small increase in the server computation costs. For databases with one million 256-byte entries, SparsePIR built on Spiral [MW22] halves response sizes from 42 KB to 21 KB compared to the cuckoo hashing approach [ALP+21] using Spiral. In exchange, SparsePIR uses only 2% more server computation. We also show that SparsePIR uses 2-12x smaller communication and at least 10x less computation than constant-weight keyword PIR [MK22]. We also introduce SparsePIR^g that has identical request and response overhead as SparsePIR. The main benefit of SparsePIR^g is that we can further decrease the encoding size and, thus, computation time by up to 20% in exchange for more time to encode the database. Finally, we introduce SparsePIR^c that achieves near optimal encoding size with small additional client storage of $<$ 11.2 KB. In summary, we show that keyword PIR requires no additional communication and a very slight increase in computation compared to standard PIR. Furthermore, we believe any future improvements to standard lattice-based PIR should also enable faster keyword PIR schemes using our frameworks. Figure 1 presents a comparison of keyword PIRs. Finally, we show that we can build batch keyword PIR schemes using any of our SparsePIR algorithms with small response sizes.

2 Preliminaries

We denote vectors $\mathbf{v}$ as column vectors and $\mathbf{v}^\intercal$ as row vectors. We denote the i-th entry of $\mathbf{v}$ by $\mathbf{v}[i]$. For two vectors $\mathbf{a}$ and $\mathbf{b}$ in $\{0,1\}^n$, we denote the dot product operator as $\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} \mathbf{a}[i] \cdot \mathbf{b}[i]$. For a matrix $\mathbf{M} = [\mathbf{M}_1 \ \ldots \ \mathbf{M}_m]$, we denote the dot product $\mathbf{a} \cdot \mathbf{M} = [\mathbf{a} \cdot \mathbf{M}_1 \ \ldots \ \mathbf{a} \cdot \mathbf{M}_n]$. For any database consisting of key-value pairs $D = \{(k_1, v_1), \ldots, (k_n, v_n)\}$, we denote $D[k]$ to be the standard operation to access the value associated with k in the database D. If $k = k_i$ for some $i \in [n]$, then $D[k] = v_i$. If $k \neq k_i$ for all $i \in [n]$, then $D[k] = \perp$.

Next, we present definitions of PIR and batch PIR. Formal definitions may be found in Appendix A.

PIR. We will restrict ourselves to single-server, one-round schemes. A keyword PIR scheme consists of algorithms $\Pi = (\mathsf{Init}, \mathsf{Encode}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Decrypt})$. Init initializes all crypto keys and Encode enables preparing a database for queries. The client runs Query and $\mathsf{Decrypt}$ to create a request and compute an answer from the server's response. The server runs Answer to create a response from the client's request. For correctness, the client should retrieve the correct value associated with the queried key. In terms of privacy, the server should not learn information about the client's queried key.

Batch PIR. In this setting, clients query a batch of $\ell \geq 1$ query keys to retrieve all ℓ associated values. A batch keyword PIR consists of the same four algorithms $\Pi = (\mathsf{Init}, \mathsf{Encode}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Decrypt})$. For correctness, the scheme should return all ℓ associated values correctly. In terms of privacy, the server should not learn any information about the client's set of queried keys.

3 Keyword PIR over Sparse Databases

3.1 State-of-the-Art PIR Schemes

We start by revisiting the current state-of-the-art PIR schemes [ABFK16, ACLS18, GH19, PT20, ALP$^+$21, MCR21, AYA$^+$21, MW22] that utilize leveled fully homomorphic encryption (FHE) built on the Ring Learning with Errors (RLWE) assumption [LPR10].

Fully Homomorphic Encryption. Most leveled FHE schemes relying on RLWE (such as [Bra12, FV12, BGV14]) share similar mathematical structures. These FHE schemes are based on a cyclotomic ring $R = \mathbb{Z}[x]/(x^N+1)$ where N is the degree of the polynomials. N is also commonly referred to as the ring dimension. Plaintext values are encoded as polynomials in the ring $R_\alpha = R/\alpha R$ where the integer α is referred to as the plaintext modulus. Ciphertexts consist of two polynomials from the ring $R_\beta = R/\beta R$ for some integer β. The secret key s in these FHE schemes will be polynomials in R with very small coefficients. An encryption of a plaintext polynomial pt will look like $(r, r \cdot s + e + \mathsf{pt})$ where r is a uniformly random element from R_β, s is the secret key and e is the noise polynomial where each coefficient is typically small and drawn from a truncated Gaussian distribution. To decrypt a ciphertext $(\mathsf{ct}_1, \mathsf{ct}_2)$, one will compute $\mathsf{ct}_2 - (\mathsf{ct}_1 \cdot s) = e + \mathsf{pt}$. As long as the noise polynomial e remains small, the plaintext pt can be retrieved by rounding to remove e.

FHE schemes allow homomorphic operations over ciphertexts. PIR schemes generally rely upon three core operations: ciphertext-ciphertext addition, ciphertext-plaintext multiplication and ciphertext-ciphertext multiplication. The costs and noise growth of these operations are quite different. Typically, ciphertext-ciphertext addition is the most efficient, ciphertext-plaintext multiplication is a bit more expensive and ciphertext-ciphertext multiplication is the worst of the three.

Choosing FHE Parameters. Each homomorphic operations adds additional noise. Recall that the decryption process succeeds only when the noise polynomial e is small enough that rounding succeeds. The parameters must be chosen to ensure decryption works correctly. One could do this by picking very large parameters, but this increases the ciphertext size and costs of homomorphic operations. So, parameters are typically chosen as small as possible while still guaranteeing proper decryption. The parameters also need to provide a certain level of security. In our work, we will pick parameters with 128 bits of security. The result is a leveled FHE scheme as each homomorphic operation increases noise and a limited number of homomorphic operations are allowed.

Query-Independent PIR Parameters. Following prior works [ACLS18, ALP$^+$21, MCR21, MW22], we will consider the model where the client uploads query-independent parameters to the server. These parameters will be used by the server to process all future requests sent by the client. At a high level, these public parameters enable the server to operate over FHE ciphertexts efficiently. We assume that these parameters are sent by the client to the server in an "offline" phase before any queries are performed.

Recursion. A common technique in standard PIR is to utilize recursion introduced in [KO97] that represents the n-entry array database as a hypercube of dimensions $d_1 \times d_2 \times \ldots \times d_z$ where the product of the dimensions is at least n, $d_1 \cdots d_z \geq n$. To query an entry, the client will generate z indicator vectors $\mathbf{v}_1 \in \{0,1\}^{d_1}, \ldots, \mathbf{v}_z \in \{0,1\}^{d_z}$ where each $\mathbf{v}_i$ has zeroes everywhere except for a one indicating the location of the entry with respect to the dimension d_i. The benefit of this representation is that querying for an entry only requires uploading an encrypted bit vector of length $d_1 + \ldots + d_z$ that can be substantially smaller than n. For example, if we set $d_1 = \sqrt{n}$ and $d_2 = \sqrt{n}$, the total bit length $2\sqrt{n}$. An unfortunate side note is that recursion introduces the need for ciphertext-ciphertext multiplication. The majority of PIR works aim to achieve recursion while obtaining better trade-offs between communication and computation. In our work, we will present keyword PIR schemes whose constructions will utilize recursion to reduce communication. We note there are other important PIR techniques such as compression and oblivious expansion (see Appendix B). However, unlike recursion, they end up being easier to fit into our framework.

Building on a PIR Scheme. Our constructions will be frameworks that can be built upon FHE based PIR schemes, which are the most common ones today. In other words, one can view our constructions as transforming standard PIR schemes into keyword PIRs that can handle sparse databases. We will assume a standard PIR scheme $\Pi_{\mathsf{PIR}} = (\mathsf{Init}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Decrypt})$ with the following properties:

- The underlying PIR scheme will represent any m-element database as a hypercube with dimensions $d_1 \times \ldots \times d_z$. Our framework will be parameterized by the dimensions $(d_1, \ldots, d_z)$ that may depend on n.
- The $\Pi_{\mathsf{PIR}}.\mathsf{Init}$ algorithm produces client key ck and server key sk (i.e., query-independent parameters).
- The $\Pi_{\mathsf{PIR}}.\mathsf{Query}$ algorithm receives z vectors of length $d_1, \ldots, d_z$ that it will homomorphically encrypt and upload to the server. For standard PIR, these z vectors are indicator vectors representing the query index in each dimension. As compression and oblivious expansion can handle arbitrary Hamming weight vectors (see Appendix B), we assume that Π_{PIR} can receive arbitrary vectors for the first d_1-length vector.
- The $\Pi_{\mathsf{PIR}}.\mathsf{Answer}$ algorithm receives an encoding $\mathbf{E}$, a two-dimensional matrix of size $d_1 \times \lceil m/d_1 \rceil$, and homomorphic encryptions of vectors $\mathbf{v}_1, \ldots, \mathbf{v}_z$. The $\Pi_{\mathsf{PIR}}.\mathsf{Answer}$ algorithm will perform the

standard PIR algorithm of applying $\mathbf{v}_1$ to $\mathbf{E}$ to obtain a $\lceil m/d_1 \rceil$ vector, arrange the vector into a $d_2 \times \lceil m/(d_1 d_2) \rceil$ matrix and apply $\mathbf{v}_2$ to obtain a vector of size $\lceil m/(d_1 d_2) \rceil$, and repeat this process for all z dimensions.

- The $\Pi_{\mathsf{PIR}}.\mathsf{Decrypt}$ algorithm receives the server's response and produces a decrypted answer. We will assume that the answer was already decoded from the decrypted plaintext polynomial into a string.

3.2 Warm-Up: Keyword PIR without Recursion

To start, we consider a simplified setting where we will utilize an underlying PIR scheme Π_{PIR} that does not use recursion. In other words, we will represent the encoded database $\mathbf{E}$ as a vector of length $d_1 = m$. For this setting, we do not care if the query vector can be represented succinctly and we will be content with uploading an encrypted version of a linear number of values. In future sections, we will fix this issue to enable recursion. However, we choose to start from this simplified setting as it illuminates some of the ideas that we will use in our more efficient constructions.

Representing Key-Value Pairs. For any pair (k, v) of a key and a value, we will represent them in the following way. For some hash key K, we denote $\mathsf{rep}(\mathsf{K}, k, v)$ as the hash evaluation of k concatenated with v. In more detail, $\mathsf{rep}(\mathsf{K}, k, v) = F(\mathsf{K}, k) \mid\mid v$ where we denote $\mid\mid$ as concatenation. We assume the representation length is fixed for all (k, v). For convenience, we will assume that $\mathsf{rep}(\mathsf{K}, k, v)$ is small enough to be uniquely represented in $\mathbb{Z}_\alpha$ where α is the plaintext modulus and each $\mathsf{rep}(\mathsf{K}, k, v)$ will be stored into one ciphertext. We will discuss later how to handle the setting when larger values and packing multiple values into a ciphertext.

Using our representation, we note that one can distinguish representations of different key-value pairs. For different keys $k_1 \neq k_2$, the resulting representations will be different $\mathsf{rep}(\mathsf{K}, k_1, v) \neq \mathsf{rep}(\mathsf{K}, k_2, v)$ except with negligible probability as the representations are the same only when $F(\mathsf{K}, k_1) = F(\mathsf{K}, k_2)$. If the hash output is sufficiently long, these collisions do not occur in practice.

Encoding the Database. Let K_r be the hash key used to generate representations of key-value pairs. Throughout this section, we will assume that all operations are done in $\mathbb{Z}_\alpha$ where α is the plaintext modulus. We will associate each key $k \in \mathcal{K}$ with a uniformly random subset of $S_k \subseteq [m]$ that will be randomly generated. The goal is to arrange the encoding $\mathbf{E}$ such that for any database $D = \{(k_1, v_1), \ldots, (k_n, v_n)\}$ and for any integer $i \in [n]$,

$$\sum_{x \in S_{k_i}} \mathbf{E}[x] = \mathsf{rep}(\mathsf{K}_r, k_i, v_i)$$

where S_{k_i} is the random subset associated with key k_i. For convenience, we can denote the set S_k using a vector $\mathbf{v}_k$ such that $\mathbf{v}_k[x] = 1$ only when $x \in S_k$. We can re-write the above equation as the dot product:

$$\mathbf{v}_{k_i} \cdot \mathbf{E} = \sum_{x \in S_{k_i}} \mathbf{E}[x] = \mathsf{rep}(\mathsf{K}_r, k_i, v_i).$$

To construct the encoded database $\mathbf{E}$, we note that the above represents n systems of equations that need to be satisfied. We can view each $\mathbf{v}_{k_i}^{\intercal}$ to be the i-th row of a $n \times m$ matrix $\mathbf{M}$. We also denote the vector $\mathbf{y}$ such that the i-th entry is the i-th value, $\mathbf{y}[i] = \mathsf{rep}(\mathsf{K}_r, k_i, v_i)$. Therefore, we must find a vector $\mathbf{E} \in \mathbb{F}^m$ satisfying:

$$\begin{bmatrix} \mathbf{v}_{k_1}^\mathsf{T} \\ \mathbf{v}_{k_2}^\mathsf{T} \\ \cdots \\ \mathbf{v}_{k_n}^\mathsf{T} \end{bmatrix} \cdot \mathbf{E} = \mathbf{M} \cdot \mathbf{E} = \mathbf{y} = \begin{bmatrix} \mathsf{rep}(\mathsf{K}_r, k_1, v_1) \\ \mathsf{rep}(\mathsf{K}_r, k_2, v_2) \\ \cdots \\ \mathsf{rep}(\mathsf{K}_r, k_n, v_n) \end{bmatrix}.$$

In other words, we can generate a correct encoding $\mathbf{E}$ of the database $D = \{(k_1, v_1), \ldots, (k_n, v_n)\}$ as long $\mathbf{M} \cdot \mathbf{E} = \mathbf{y}$ has a solution, i.e. $\mathbf{M}$ has full row rank. Intuitively, increasing the number of columns in $\mathbf{M}$, denoted by m, will also increase the probability that it has full row rank. However, this has the affect of also increasing the size of the encoded database $\mathbf{E} \in \mathbb{F}^m$. Therefore, we would like to pick m in such a way that $\mathbf{M}$ has full row rank with high probability while minimizing the size of the encoding.

Decoding an Entry. From the above encoding scheme, we note that decoding an entry is pretty straightforward. For any query key k, the client will randomly generate the vector $\mathbf{v}_k$. The client also computes the hash evaluation $F(\mathsf{K}_r, k)$. Using our PIR scheme without recursion, the client uploads an encrypted version of $\mathbf{v}_k$. The PIR scheme continues without change and the client will eventually receive the dot product $\mathbf{v}_k \cdot \mathbf{E}$ in plaintext. The client will parse this result as $a \mid\mid b$. If $a = F(\mathsf{K}_r, k)$, then the client returns b as the retrieved value. Otherwise, when $a \neq F(\mathsf{K}_r, k)$, then the client returns $\perp$.

If $k = k_i$ for some $(k_i, v_i) \in D$, then we know that $\mathbf{v}_k \cdot \mathbf{E}$ will be exactly $F(\mathsf{K}_r, k_i) \mid\mid v_i$ enabling retrieval of v_i. On the other hand, suppose that $k \notin D$. Then, we know that $a \neq F(\mathsf{K}_r, k)$ meaning the client returns the right response (except with negligible probability of collisions).

Quick Comparison with Prior Approaches. In the above, we only needed to retrieve a single ciphertext, which was the dot product response. On the other hand, cuckoo hashing required retrieving two entries. Furthermore, the above approach ensures that the number of entries is very close to linear. We have also ensured minimal error growth as no additional homomorphic operations are performed. This shows us that we are heading in the right direction, although there are still several problems that need to be resolved.

Problems with this Approach. While the above approach is very promising, it has two significant problems that we outline below and solve in later sections:

1. As discussed earlier, we considered a simplified setting of PIR that does not utilize recursion. Unfortunately, PIR without recursion ends up requiring linear communication. Most state-of-the-art PIR schemes utilize recursion to represent the database using a hypercube with dimensions $d_1 \times \ldots \times d_z$ such that $d_1 \cdots d_z \geq n$. In our scheme, we chose each row vector $\mathbf{v}_k$ to be uniformly random, which cannot be represented in any succinct manner using recursion.
2. A second, more subtle, issue is that computing a solution to the system of linear equations $\mathbf{M} \cdot \mathbf{E} = \mathbf{y}$ is very expensive. The best possible algorithm for solving a general linear system would already be quadratic in the size of the database n.

3.3 Partition-Based Keyword PIR

At a high level, both problems from the prior construction resulted from the fact that the random matrix $\mathbf{M}$ was very dense with many non-zero entries. The expected number of non-zero entries was $nm/2$, so it is not surprising that finding a solution would require at least quadratic time.

Algorithm 1 $\mathsf{SparsePIR.Init}$ algorithm

```
Input: $1^\lambda$: security parameter.
Output: $(\mathsf{ck}, \mathsf{sk})$: client and server key.
  $(\mathsf{ck}, \mathsf{sk}) \leftarrow \Pi_{\mathsf{PIR}}.\mathsf{Init}(1^\lambda)$
  return $(\mathsf{ck}, \mathsf{sk})$
```

Furthermore, as each row vector has, on average, $m/2$ non-zero entries, it is not surprising that any representation of the row vector must also be large. We present our construction SparsePIR that deals with the issues from the previous section to obtain an efficient keyword PIR.

High-Level Idea of Partitioning. To solve this problem that row vectors have high density, we will use the idea of partitioning where we aim to break down the problem into smaller subproblems. To do this, we will aim to embed the random vector into only the first dimension used in recursion. We will assume that the underlying PIR scheme uses recursion and represents the database as $d_1 \times \ldots \times d_z$ hypercube.

At a high level, we will create $b = (1+\epsilon)n/d_1$ partitions where each part will be size d_1. We will choose the value of ϵ later through experimentation. On input of database $D = \{(k_1, v_1), \ldots, (k_n, v_n)\}$ that we wish to encode, we will assign the n key-value pairs to the b partitions uniformly at random. Therefore, the expected number of key-value pairs in each part is $n/b = d_1/(1+\epsilon) < d_1$. As a result, we have now reduced the problem to efficiently encoding databases of size $O(d_1)$.

Within each part P_i, we will essentially repeat the construction from Section 3.2. Consider the i-th part $P_i = \{(k_1, v_1), \ldots, (k_{|P_i|}, v_{|P_i|})\}$. We will generate $\mathbf{M}_i$ that is a $|P_i| \times d_1$ matrix where each entry is uniformly chosen from $\{0, 1\}$. In particular, the i-th row is generated randomly using K_2 and key k_i. We compute $\mathbf{y}_i^\intercal = [\mathsf{rep}(\mathsf{K}_r, k_1, v_i) \ \ldots \ \mathsf{rep}(\mathsf{K}_r, k_{|P_i|}, v_{|P_i|})]$. Finally, we solve the linear system $\mathbf{M}_i \cdot \mathbf{e}_i = \mathbf{y}_i$ and obtain the encoding $\mathbf{e}_i$ for part P_i.

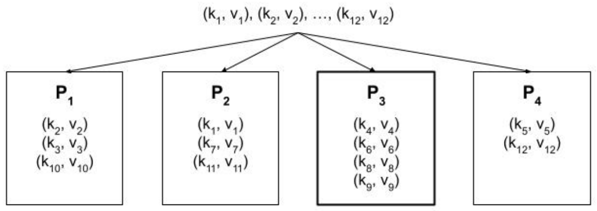
To build the final encoded database $\mathbf{E}$, we put each of $\mathbf{e}_1, \ldots, \mathbf{e}_b$ as column vectors in $\mathbf{E}$ to construct a $d_1 \times b$ matrix. Note, the cost to solve the linear system in each part is $O(d_1^3)$ and the total time across all $b = O(n/d_1)$ parts is $O(n \cdot d_1^2)$. For small values of d_1, this is much more efficient than approach in Section 3.2.

To query for a key k, the client will compute $i = F_1(\mathsf{K}_1, k)$ to determine the partition associated with k. Next, the client randomly generates the associated random vector using hash key K_2 and k that we denote $\mathbf{v}_1$. This will be the vector uploaded for the first dimension of PIR. Suppose that the server applied $\mathbf{v}_1$ to the first dimension of $\mathbf{E}$. Note, the result is $[\mathbf{v}_1 \cdot \mathbf{e}_1 \ \ldots \ \mathbf{v}_1 \cdot \mathbf{e}_b]$. As k is assigned to the i-th partition, the only entry needed to be retrieved is $\mathbf{v}_1 \cdot \mathbf{e}_i$. Therefore, this reduces to performing a standard PIR query over a b-entry array with recursion dimensions $d_2 \times \ldots \times d_z$. We formally present the above ideas in our SparsePIR construction below.

Encoding the Database. The encoding algorithm SparsePIR is formally presented in Algorithm 2 that utilizes the partitioning ideas that were discussed previously. Our algorithms are also portrayed pictorially in Figure 2. Recall that the algorithm receives a database $D = \{(k_1, v_1), \ldots, (k_n, v_n)\}$ as input and must output parameters for decoding and an encoding of the database. The encoding is parameterized by ϵ where $b = (1+\epsilon)n/d_1$ is the number of parts in the partitioning.

First, the algorithm samples three hash keys: K_1, K_2 and K_r. The first key K_1 will be used to generate the random partitioning of the database into b bins such that $F_1(\mathsf{K}_1, \cdot) \in \{0, \ldots, b-1\}$. The second key K_2 will be used to generate the random row vectors within each partition where $F_2(\mathsf{K}_2, \cdot) \in \{0, 1\}$. K_r is used to generate representations $\mathsf{rep}(\mathsf{K}_r, k, v)$ of key-value pairs (k, v).

Partition



Encode P_1, P_2, P_3, P_4 only P_3 shown here

$$\mathbf{e}_3 = \begin{bmatrix} F(\mathsf{K}_2, k_4 \mid\mid 1) & \dots & F(\mathsf{K}_2, k_4 \mid\mid 5) \\ F(\mathsf{K}_2, k_6 \mid\mid 1) & \dots & F(\mathsf{K}_2, k_6 \mid\mid 5) \\ F(\mathsf{K}_2, k_8 \mid\mid 1) & \dots & F(\mathsf{K}_2, k_8 \mid\mid 5) \\ F(\mathsf{K}_2, k_9 \mid\mid 1) & \dots & F(\mathsf{K}_2, k_9 \mid\mid 5) \end{bmatrix}^{-1} \begin{bmatrix} F(\mathsf{K}_r, k_4) \mid\mid v_4 \\ F(\mathsf{K}_r, k_6) \mid\mid v_6 \\ F(\mathsf{K}_r, k_8) \mid\mid v_8 \\ F(\mathsf{K}_r, k_9) \mid\mid v_9 \end{bmatrix}$$

Final Encoding

$$\mathbf{E} = \begin{bmatrix} \mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 & \mathbf{e}_4 \end{bmatrix}$$

Figure 2: Encoding algorithm for SparsePIR with $n = 12$, $b = 4$ and $(d_1 = 5, d_2 = 2, d_3 = 2)$.

For any input database $D = \{(k_1, v_1), \dots, (k_n, v_n)\}$, the next step is to partition the n key-value pairs into the b entries uniformly at random. In particular, the key-value pair (k_i, v_i) is placed into the $F_1(\mathsf{K}_1, k_i)$-th partition. Let $P_1, \dots, P_b$ be the b parts of the partitioning such that each (k_i, v_i) is assigned to exactly one part. Each part P_i will produce an encoding $\mathbf{e}_i$ and the final encoding $\mathbf{E}$ will combine all of $\mathbf{e}_1, \dots, \mathbf{e}_b$ together.

For each part $i \in [b]$, we will use part P_i to construct a matrix $\mathbf{M}_i$, a vector $\mathbf{y}_i$ as well as an encoding $\mathbf{e}_i$. We iterate through the part P_i. For each $(k, v) \in P_i$, we append the bit vector generated by $(F_2(\mathsf{K}_2, k \mid\mid 1), \dots, F_2(\mathsf{K}_2, k \mid\mid d_1))$ as a row vector in $\mathbf{M}_i$ where each $F_2(\mathsf{K}_2, k \mid\mid \cdot) \in \{0, 1\}$. Additionally, we append $\mathsf{rep}(\mathsf{K}_r, k, v)$ into $\mathbf{y}_i$. At the end, we note that $\mathbf{M}_i$ is a vector with $|P_i|$ rows and d_1 columns. Furthermore, $\mathbf{y}_i$ is a vector with $|P_i|$ entries. Next, we compute $\mathbf{e}_i$ that satisfies the following equality $\mathbf{M}_i \mathbf{e}_i = \mathbf{y}_i$. If a solution doesn't exist for any of the part $i \in [b]$, the encoding algorithm outputs $\perp$ and terminates.

After doing this for all parts $i \in [b]$, we obtain the b partial encodings $\mathbf{e}_1, \dots, \mathbf{e}_b$. To obtain the final encoding, we construct the two-dimensional matrix as $\mathbf{E} = [\mathbf{e}_1 \ \dots \ \mathbf{e}_b]$ where each $\mathbf{e}_i$ is a column vector. Note, this matrix has dimension $d_1 \times b$. The output of the encoding is $\mathsf{prms} = (\mathsf{K}_1, \mathsf{K}_2, \mathsf{K}_r)$ and the matrix $\mathbf{E}$.

Decoding an Entry. Our decoding process of Query, Answer and Decrypt are outlined in Algorithms 5, 6 and 7. We also present a diagram of the decoding process in Figure 3. As input, the decoding algorithm will receive parameters $\mathsf{prms} = (\mathsf{K}_1, \mathsf{K}_2, \mathsf{K}_r)$ as well as a query key $k \in \mathcal{K}$. Recall that the goal is to output z vectors $\mathbf{v}_1, \dots, \mathbf{v}_z$ such that applying standard PIR with recursion schemes along with our encoded database $\mathbf{E}$ would enable decoding the value associated with query key k (if it exists).

The first step is to compute $F_1(\mathsf{K}_1, k)$ to compute the partition associated with k. Suppose that k was associated with the i-th part. The next step is to compute $F_2(\mathsf{K}_2, k \mid\mid x)$ for all $x \in [d_1]$ to obtain the random bit vector of length d_1. The above bit vector will be $\mathbf{v}_1$. Next, suppose we applied the first layer of the server's response algorithm in standard PIR schemes that utilize

Algorithm 2 $\mathsf{SparsePIR.Encode}$ algorithm

```
Input: $D = \{(k_1, v_1), \ldots, (k_n, v_n)\}$: database
Output: $(\mathsf{prms}, \mathbf{E})$: parameters and encoding
  Sample hash keys $\mathsf{K}_1, \mathsf{K}_2$ and $\mathsf{K}_r$
  $b \leftarrow (1 + \epsilon)n/d_1$
  $m \leftarrow (1 + \epsilon)n$
  $P_1 \leftarrow \emptyset, \ldots, P_b \leftarrow \emptyset$
  for $i = 1, \ldots, n$ do                                  ▷ Partition database
      $j \leftarrow F_1(\mathsf{K}_1, k_i)$
      $P_{j+1} \leftarrow P_{j+1} \cup \{(k_i, v_i)\}$
  for $j = 1, \ldots, b$ do
      $\mathbf{e}_i \leftarrow \mathsf{GenerateEncode}(\mathsf{K}_2, \mathsf{K}_r, P_j)$
  $\mathsf{prms} \leftarrow (\mathsf{K}_1, \mathsf{K}_2, \mathsf{K}_r)$
  $\mathbf{E} \leftarrow [\mathbf{e}_1, \ldots, \mathbf{e}_b]$
  return $(\mathsf{prms}, \mathbf{E})$
```

Algorithm 3 $\mathsf{RandVector}$ algorithm

```
Input: $(\mathsf{K}_2, k)$: the hash key and database key.
Output: $\mathbf{v}_k$: a randomly generated vector.
  $\mathbf{v}_k \leftarrow [0]^{d_1}$
  for $i = 1, \ldots, d_1$ do
      $\mathbf{v}_k[i] \leftarrow F_2(\mathsf{K}_2, k \mid\mid i)$
  return $\mathbf{v}_k$
```

recursion. Note, that the result is $[\mathbf{v}_1 \cdot \mathbf{e}_1 \ \ldots \ \mathbf{v}_1 \cdot \mathbf{e}_b]$. Furthermore, as we know that k was assigned to the i-th part, we only need to retrieve the i-th column containing the entry $\mathbf{v}_1 \cdot \mathbf{e}_i$. In other words, the rest of the problem becomes a standard PIR query of retrieving the i-th entry from an array with b entries. To do this, we encode the index i using the standard PIR query over a database of dimensions $d_2 \times \ldots \times d_z$.

Packing. Throughout this section, we assumed that $\mathsf{rep}(\mathsf{K}_r, k, v)$ fit into $\mathbb{Z}_\alpha$ where α is the plaintext modulus and each representation was stored in a single ciphertext. We show how to handle arbitrary length values and efficiency improvements using packing. We will represent $\mathsf{rep}(\mathsf{K}, k, v)$ as a base-α string with $L = \lceil \log_\alpha(|\mathsf{rep}(\mathsf{K}, k, v)|) \rceil$ characters in $\mathbb{Z}_\alpha$. We repeat the above encoding algorithm for each part P_i to obtain L encodings $\mathbf{e}_i^1, \ldots, \mathbf{e}_i^L$ using the same matrix $\mathbf{M}_i$. The resulting encoding is: $\mathbf{E} = [\mathbf{e}_1^1 \ \ldots \ \mathbf{e}_1^L \ \ldots \ \mathbf{e}_b^1 \ \ldots \ \mathbf{e}_b^L]$.

If $L > N$, we encode each row of $\mathbf{E}$ using $b \cdot \lceil L/N \rceil$ plaintext polynomials where each tuple $(\mathbf{e}_i^1, \ldots, \mathbf{e}_i^L)$ uses $\lceil L/N \rceil$ polynomials. When $L \leq N$, we can encode $\lfloor N/L \rfloor$ values into a single polynomial meaning each row of $\mathbf{E}$ is encoded using $\lceil b/\lfloor N/L \rfloor \rceil$ polynomials.

Re-Sampling and Optimizations. The encoding algorithm of $\mathsf{SparsePIR}$ fails (outputs $\perp$) when any of the smaller linear systems doesn't have a solution. In practice, rather than terminating, the encoding would simply re-sample new hash keys K_1 and K_2 and try to encode the database again. In our implementation, this will also be the case. However, we show that one may optimize the re-sampling step. The naive approach would be to re-sample both K_1 and K_2, which is unnecessary.

Algorithm 4 GenerateEncode algorithm

```
Input: $(\mathsf{K}_2, \mathsf{K}_r, P)$: hash keys and a part.
Output: $\mathbf{e}$: an encoding of the part.
  $\mathbf{M} \leftarrow []$ as empty array
  $\mathbf{y} \leftarrow []$ as empty array
  for $(k, v) \in P$ do
      Append $\mathsf{RandVector}(\mathsf{K}_2, k)^\intercal$ to $\mathbf{M}$ as row.
      Append $\mathsf{rep}(\mathsf{K}_r, k, v)$ to $\mathbf{y}$.
  $\mathbf{e} \leftarrow \mathsf{SolveLinearSystem}(\mathbf{M}, \mathbf{y})$
  return $\mathbf{e}$
```

Algorithm 5 SparsePIR.Query algorithm

```
Input: $(\mathsf{prms}, \mathsf{ck}, k)$: parameters and the query key.
Output: $(\mathsf{st}, \mathsf{req})$: temporary state and request.
  $b \leftarrow (1+\epsilon)n/d_1$
  Parse $\mathsf{prms} = (\mathsf{K}_1, \mathsf{K}_2, \mathsf{K}_r)$
  $\mathbf{v}_1 \leftarrow \mathsf{RandVector}(\mathsf{K}_2, k)$
  $j \leftarrow F_1(\mathsf{K}_1, k)$
  for $i = 2, \ldots, z$ do                                  ▷ Generate encoding of $j$
      $j_i \leftarrow \lfloor j / \lceil b/d_i \rceil \rfloor$
      $\mathbf{v}_i \leftarrow [0]^{d_i}$
      $\mathbf{v}_i[j_i + 1] \leftarrow 1$
      $b \leftarrow \lceil b/d_i \rceil$
      $j \leftarrow j \mod b$
  $(\mathsf{st}_{\mathsf{PIR}}, \mathsf{req}) \leftarrow \Pi_{\mathsf{PIR}}.\mathsf{Query}(\mathsf{ck}, \mathbf{v}_1, \ldots, \mathbf{v}_z)$
  return $((\mathsf{st}_{\mathsf{PIR}}, k), \mathsf{req})$
```

Recall that K_1 partitions the n key-value pairs into b parts. If any single part P_i receives too many items, the associated linear system $\mathbf{M}_i \cdot \mathbf{e}_i = \mathbf{y}_i$ will not have a solution. In this case, we do not need to even generate $\mathbf{M}_i$. Therefore, we try to sample K_1 until the resulting partition does not over-assign items to any part. Once a good K_1 is found, we fix it. Afterwards, we only keep re-sampling K_2 until all randomly generated matrices have full row rank.

Handing Database Updates. In most applications, the database D will periodically change. Our encoding algorithm does not enable incremental updates to add/remove entries. If the database changes, the encoding must be run again. To handle database updates, new encoded databases may be generated. As we will show in Section 7, the encoding is concretely efficient to enable creating encoded databases every few minutes.

3.4 Efficiently Solvable Matrices

For SparsePIR, we have been generating random matrices such that each entry is chosen to be uniformly random from $\{0, 1\}$. The state-of-the-art algorithm for solving general linear systems for practical sizes of n require at least quadratic (and, typically, near-cubic) time. This is unsurprising as the expected number of non-zero entries in these matrices is $n^2/2$.

Algorithm 6 SparsePIR.Answer algorithm

```
Input: (prms, sk, $\mathbf{E}$, req): parameters, server key, encoded databases and the request.
Output: resp: the response to the request.
  resp $\leftarrow \Pi_{\mathsf{PIR}}$.Answer(sk, $\mathbf{E}$, req)
  return resp
```

Algorithm 7 SparsePIR.Decrypt algorithm

```
Input: (prms, ck, st, resp): parameters, client key, temporary state and response.
Output: $v$: output value
  Parse prms as $(\mathsf{K}_1, \mathsf{K}_2, \mathsf{K}_r)$
  Parse st as $(\mathsf{st}_{\mathsf{PIR}}, k)$
  $x \leftarrow \Pi_{\mathsf{PIR}}$.Decrypt(ck, $\mathsf{st}_{\mathsf{PIR}}$, resp)
  Parse $x$ as (id, $v$)
  if id $= F(\mathsf{K}_r, k)$ then
      return $v$
  else
      return $\perp$
```

To circumvent this issue, SparsePIR utilized partitioning to guarantee that the generated linear systems were very small in size. Recall that each of the $O(n/d_1)$ parts involves solving a linear system of size $O(d_1^2)$, which takes $O(d_1^3)$ time using Gaussian elimination algorithm. As long as we chose d_1 to be small, then the overall encoding algorithm required $O(n \cdot d_1^2)$ time that was relatively efficient. For example, if $d_1 = O(\log n)$, then the overall encoding algorithm would take only $O(n \log^2 n)$ which seems reasonable. However, we critically require that d_1 must be small as the encoding time would grow otherwise.

Taking a closer look, SparsePIR never required that the random matrices were generated such that each entry was chosen uniformly at random from $\{0, 1\}$. In fact, it only required the linear system associated with the generated random matrix to have a solution with high probability. We show that there are ways to generate such random matrices that admit efficiently solvable linear systems.

Sparse Random Matrices. It turns out that core algorithmic problem of generating random matrices such that the associated linear system can be solved efficiently is a well-studied area (see [Wie86, Coo00, Mol05, Por09, PS16] and references therein as some examples). For our work, we will utilize the *random band matrix* constructions of Dietzfelbinger and Walzer [DW19] that satisfy the requirements needed by our keyword PIR schemes.

Random band matrices are constructed such that each row vector will consist of a short band of length w. The short band will be a uniformly random w-bit vector. All entries outside of the band will be zero. The location of the short band is chosen uniformly at random. Dietzfelbinger and Walzer [DW19] showed that for matrices of dimension $(1 - \gamma)n \times n$ for some constant $0 < \gamma < 1$, if the band parameter is chosen as $w = O(\log n/\gamma)$, then the resulting random matrix has full row rank with high probability. To solve the associated linear system, one will first sort the rows by the starting location of the non-zero band of length w. Next, one can employ Gaussian elimination in a very efficient manner. At a high level, Gaussian elimination only needs to consider columns within the w-length band for each row. Therefore, Gaussian elimination requires only $O(n/\gamma^2)$ time. In

Query for k_6

$$\mathbf{v}_{k_6} = \begin{bmatrix} F(\mathsf{K}_2, k_6 \mid\mid 1) \\ \dots \\ F(\mathsf{K}_2, k_6 \mid\mid 5) \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

First Level of Recursion for $d_1 = 5$

$$\mathbf{v}_{k_6} \cdot \mathbf{E} \to \begin{bmatrix} \mathbf{v}_{k_6} \cdot \mathbf{e}_1 & \dots & \mathbf{v}_{k_6} \cdot \mathbf{e}_4 \end{bmatrix}$$

Second Level of Recursion for $d_2 = 2$

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{v}_{k_6} \cdot \mathbf{e}_1 & \mathbf{v}_{k_6} \cdot \mathbf{e}_2 \\ \mathbf{v}_{k_6} \cdot \mathbf{e}_3 & \mathbf{v}_{k_6} \cdot \mathbf{e}_4 \end{bmatrix} \to \begin{bmatrix} \mathbf{v}_{k_6} \cdot \mathbf{e}_3 & \mathbf{v}_{k_6} \cdot \mathbf{e}_4 \end{bmatrix}$$

Third Level of Recursion for $d_3 = 2$

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{v}_{k_6} \cdot \mathbf{e}_3 \\ \mathbf{v}_{k_6} \cdot \mathbf{e}_4 \end{bmatrix} \to \mathbf{v}_{k_6} \cdot \mathbf{e}_3 = F(\mathsf{K}_2, k_6) \mid\mid v_6$$

Figure 3: Decoding algorithm for SparsePIR with $n = 12$, $b = 4$ and dimensions $(d_1 = 5, d_2 = 2, d_3 = 2)$. The server application is presented in plaintext for clarity, but the real construction would perform these operations over FHE ciphertexts homomorphically.

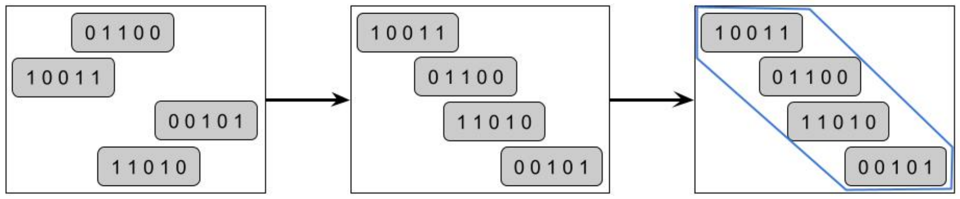


Figure 4: The left matrix is a random band matrix with 4 rows and band width $w = 5$. The middle matrix is the intermediate step of the Gaussian elimination that sorts rows by band location. The final matrix shows that Gaussian elimination only considers a small subset of entries.

Figure 4, we present a diagram depicting random band matrices and the Gaussian elimination algorithm.

Encoding the Database. We modify our construction SparsePIR to utilize random band matrices to improve the efficiency of our encoding algorithm for larger values of d_1. In particular, we only need to modify the RandVector and SolveLinearSystem sub-routines. The new sub-routines may be found in Algorithm 8 and 9. The rest of the encoding algorithm remains unchanged.

Decoding an Entry. We note that the decoding algorithm remains identical except that we utilize the new RandVector algorithm.

Encoding Running Time Analysis. Finally, we show that the usage of random band matrices improves the running time of the encoding algorithm and enables using large values of d_1. Previously, solving a linear system for each of the $O(n/d_1)$ parts took $O(d_1^3)$ using Gaussian elimination. Utilizing the random band matrix constructions, we can reduce this down to $O(d_1)$ (ignoring constant factors of ϵ for now), which makes the total encoding time only $O(n)$. As a result, we can now

Algorithm 8 RandVector algorithm

```
Input: $(\mathsf{K}_2, k)$: the hash key and database key.
Output: $\mathbf{v}_k$: a randomly generated vector.
  /* $F_3$ outputs elements in $\{0, \ldots, d - w\}$. */
  $p \leftarrow F_3(\mathsf{K}_2, k \mid\mid \text{“position”})$
  $\mathbf{v}_k \leftarrow [0]^{d_1}$
  for $i = 1, \ldots, w$ do
      $\mathbf{v}_k[p + i + 1] \leftarrow F_2(\mathsf{K}_2, k \mid\mid i)$
  return $\mathbf{v}_k$
```

Algorithm 9 SolveLinearSystem algorithm

```
Input: $(\mathbf{M}, \mathbf{y})$: band matrix and values to solve for.
Output: $\mathbf{e}$: solution to the linear system $\mathbf{M} \cdot \mathbf{e} = \mathbf{y}$.
  Sort rows of $\mathbf{M}$ and $\mathbf{y}$ according to first non-zero entry of $\mathbf{M}$
  Execute Gaussian elimination to get $\mathbf{e}$
  return $\mathbf{e}$
```

consider arbitrary values of d_1 as the encoding algorithm will not grow significantly as d_1 increases in size.

4 Extending Partition Based Keyword PIR

In the partition based keyword PIR, each of the n keywords was randomly assigned to one of $b = (1 + \epsilon)n/d_1$ partitions of size d_1. This allowed us to treat each partition as a small independent linear system and solve it separately, making the scheme highly efficient.

However, partition based keyword PIR suffers from a major problem: at most d_1 keywords can be assigned to a single partition. This means that ϵ has to be big enough for this to happen with high probability. Section 7 shows that for $n = 2^{20}$ and $d_1 = 128$, ϵ has to be at least 0.38. Throughout the rest of this section and the next one, we will consider the example of $d_1 = 128$ as all prior concretely efficient PIR schemes (including [ACLS18, ALP$^+$21, MCR21, MW22]) use $d_1 \geq 128$. It turns out that smaller values of d_1 are more difficult (see Section 7). Therefore, considering $d_1 = 128$ is effectively the most difficult setting.

Looking back, the two motivations behind the partition based keyword PIR were the inefficiency of solving a large random linear system and *recursion incompatibility* of the generated random vectors. The first issue can be solved by utilizing the *random band matrix* constructions of Dietzfelbinger and Walzer [DW19], which allow us to construct large linear systems that can be solved efficiently. Thus, if we can somehow construct random band matrices that are also *recursion compatible*, we should be able to reduce the ϵ in the partition based keyword PIR while still keeping the scheme efficient. In this section, we present our extended construction $\mathsf{SparsePIR}^g$ that achieves this goal.

Warmup: Two Dimensional Recursion. To motivate our construction, we will start off with constructing *recursion compatible* band matrices for the two dimensional recursion PIR protocol.

Recall that partition based keyword PIR allowed us to view each partition as an independent linear system. Another equivalent way to formulate this is to combine these independent linear

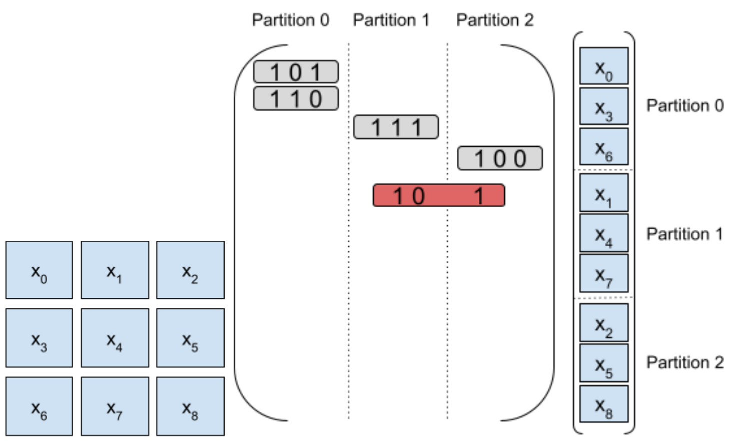


Figure 5: A two dimensional representation of a database of 9 elements and an example LHS of the combined linear system. Each column of the database corresponds to a partition. The gray bands all lie strictly inside a partition and are recursion compatible, while the red band spans partition 1 and partition 2 and is recursion incompatible.

systems and form a single system as in Figure 5. Pictorially, each partition corresponds to a contiguous range of d_1 variables (and columns in the matrix). In Figure 5, notice that the gray bands all lie strictly inside a partition and do not span multiple partitions - this is the crucial property that allowed the row vectors to be recursion compatible in the partition based keyword PIR. On the other hand, the red band spans partition 1 and partition 2 and is not recursion compatible. However, a band spanning multiple partitions is not really fundamentally problematic; it is just that such random band is more likely to be recursion incompatible than not. The main idea behind the new construction is to artificially modify these spanning bands to be recursion compatible, with the hope that the new band matrices remain "random enough" and still has a solution with high probability.

Before proceeding further, we will first permute the variables in the linear system by reversing the order of the variables that correspond to odd indexed partitions as in Figure 6. The motivation behind this change will become apparent later on. Now, instead of choosing a partition and sampling the band within the chosen partition, we instead sample the band across the entire row[1] , allowing the band to span two partitions (we still keep the band width $w < d_1$, which means that a band can span at most two partitions). If this band happens to lie strictly inside a partition, we are done. On the other hand, suppose that the band spans two partitions. Denote $\mathbf{v_1}||\mathbf{v_2}$ as the spanning band where $\mathbf{v_1}$ lies strictly in the first partition and $\mathbf{v_2}$ lies strictly in the second partition. We modify the band by discarding $\mathbf{v_2}$ and appending the reverse of $\mathbf{v_1}$: $\mathbf{v_1}||\mathbf{rev}(\mathbf{v_1})$ where **rev** denotes the reverse function. Observe that with this transformation, the row vector now becomes recursion compatible as illustrated by the green band in Figure 6. To query for the green band, the client can send vector $\begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$ for the first dimension and vector $\begin{bmatrix} 0 & 1 & 1 \end{bmatrix}$ for the second dimension. The width of the new band is upper bounded by $2w$, so the efficiency of solving the linear system remains unchanged asymptotically. Finally, because the new band vector generation algorithm only depends

[1]In our experimental evaluation, we observed that uniform distribution did not result in the best success probability. In particular, the modified bands had to be long enough for the associated linear system to be solvable with high enough probability.

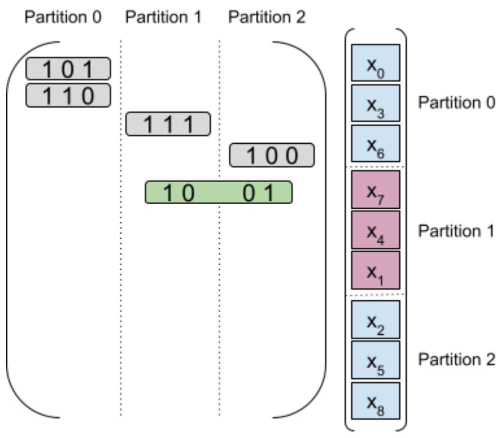


Figure 6: Modified construction from Figure 5. Variables in partition 1 are reversed and are colored purple. The red band in Figure 5 is recursion compatible after transforming to the green band.

on the keyword and the database parameters, the client can replicate this procedure to construct PIR query for an arbitrary keyword.

Extending to Higher Dimensions We will start by focusing on hypercube representation in the form $d_1 \times \underbrace{d \times \cdots \times d}_{z}$, i.e. all subsequent dimensions after the first are equal.

In the context of PIR, one useful way of visualizing a hypercube of this form is as a $d_1 \times d^z$ matrix, where each column corresponds to a first dimension slice (of size d_1) and the column indices correspond to base d representations of the "coordinates" of the first dimension slices in the d^z space. In the context of partition based keyword PIR, each column can be viewed as a partition (numbered from 0 to $d^z - 1$). Selecting partition i is equivalent to transforming i to its base d representation and naturally mapping the jth digit to the corresponding indicator vector for the $j+1$th dimension.

In the two dimensional recursion case, for bands that spanned two partitions, we had to construct the second dimension vector to indicate that we wanted to select those two partitions. This could be done in a straightforward manner by turning on bits at positions corresponding to the partition numbers in the query vector.

However, such naive approach no longer works if we move on to higher dimensions. For example, suppose we embed the database in Figure 5 and Figure 6 on to a hypercube of dimensions $3 \times 2 \times 2$. The partitions (columns) are numbered 00_2, 01_2, 10_2, 11_2 in base two (the last partition will correspond to empty entries in this case). The green band spans partition 01_2 and partition 10_2, but we can no longer construct recursion compatible query vectors (unlike the two dimensional case).

To get around this, we make the following observation: we can construct recursion compatible query vectors for selecting two partitions as long as the Hamming distance between the base d representation of the two partition numbers is 1. Indeed, we can observe that there is no problem of selecting partition 00_2 and 01_2 - simply send vector $\begin{bmatrix} 1 & 0 \end{bmatrix}$ for the second dimension and $\begin{bmatrix} 1 & 1 \end{bmatrix}$ for the third dimension. Similarly, if we want to select partition 10_2 and 11_2, we can send vector $\begin{bmatrix} 0 & 1 \end{bmatrix}$ for the second dimension and $\begin{bmatrix} 1 & 1 \end{bmatrix}$ for the third dimension.

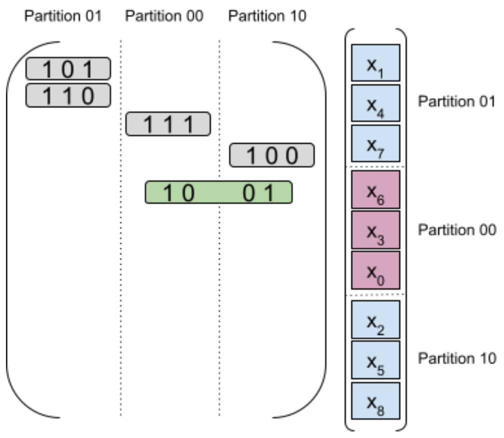


Figure 7: Modified band constructions for a hypercube representation of $3 \times 2 \times 2$ after permuting the partitions according to the Gray code of 01_2, 00_2, 10_2, and 11_2. The green band is now recursion compatible. Note that we must reverse the order of the variables in partition 00_2 (and not in partition 01_2) as this is now the odd-indexed partition.

In fact, we can see that this observation is applicable to an arbitrary hypercube structure of dimensions $d_1 \times d_2 \times \cdots \times d_z$. If we represent the partition numbers in mixed bases $(d_2, \cdots, d_z)$, then we can select the two partitions as long as the Hamming distance between the two partition numbers is 1.

This motivates the following construction: in the linear system, permute the order of the variables in such a way that adjacent partition numbers have Hamming distances 1. This way, all bands that span two adjacent partitions in the linear system will be recursion compatible. In particular, we can use mixed-radix Gray code [Knu05] to construct this permutation. After permuting the variables according to the Gray code, we have to reverse the order of odd indexed partition variables as in the two dimensional case to ensure that the first dimension remains recursion compatible. Figure 7 shows an example construction. Note that the last partition 11_2 can be ignored while solving the linear system, and will correspond to empty entries in the database. We point out that this composition of permutations does not require extra storage on the client side, as computing the nth Gray code can be done efficiently on the fly without requiring extra space.

Our experimental evaluation shows that $\mathsf{SparsePIR}^g$ is able to reduce ϵ and is practically efficient. For example, for $n = 2^{20}$ and $d_1 = 128$, $\mathsf{SparsePIR}^g$ is able to reduce ϵ from 0.38 to 0.1, a significant improvement over naive partition based keyword PIR.

5 Keyword PIR with Client Storage

In SparsePIR, if $d_1 + 1$ items were assigned to any single part then the associated linear system would not have a solution. In practice, we needed slightly less than d_1 items to make sure the solution exists with high probability thus not requiring too much resampling. We would have exact partitioning if all the parts have slightly less than d_1 items. We present $\mathsf{SparsePIR}^c$, an enhancement of SparsePIR with client-side storage, achieving exact partitioning and thus minimal encoded database size at the cost of storing moderate amounts of extra information in the decoding parameter. We provide more detailed algorithms and justification for the choice of client storage in Appendix C.

Explicit Exact Partitioning. One very expensive way of exactly partitioning is for the client to receive and explicitly store the assignment of n items to the n/d_1 parts requiring $n \log(n/d_1)$ bits. When $n = 2^{20}$ and $d_1 = 128$, this would require 1.6 MB, which is quite large.

Exact Partitioning via Boundaries. Our main idea is to first sort the hashed outputs of the n keywords and then evenly split them into n/d_1 parts. The client stores the partition boundary values that separate the n/d_1 parts, $B_0, B_1, \ldots, B_{n/d_1-1}, B_{n/d_1}$. All items in the i-th part are contained in the interval $(B_{i-1}, B_i]$. For convenience, we will assume B_0 and B_{n/d_1} to be the smallest and largest possible hash output respectively. The list B becomes part of prms at setup time. During query time, the client hashes the key k and finds the interval $(B_{i-1}, B_i]$ containing the hash output. Then, the client determines that key k belongs to the i-th part. This only requires $O(n/d_1 \log U)$ bits where U is range of $F(\mathsf{K}_r, k_i)$. For $n = 2^{20}$, $d_1 = 128$, $|U| = 2^{64}$, then this requires 64 KB.

Truncation of Boundaries. For our next improvement, we note that it is not necessary to store the full boundary value. Instead, we can set most of the least significant bits of boundary values to zero and truncate them since we need B_i only to be precise enough to distinguish it from the neighboring keyword hash value to maintain exact partitioning. This can reduce the $\log U$ bits used to represent each boundary. For example, when $n = 2^{20}$ and $d_1 = 128$, we see that approximately 25 bits are necessary through experimentation (see Appendix C for more details). For the concrete example of $n = 2^{20}$ and $d_1 = 128$, this reduces client storage by 60% to 25 KB.

Compression for Persistent Storage. Furthermore, we show that one can further compress the boundary values when storing in persistent storage. In particular, we can store the differences between boundaries as opposed to the boundaries themselves. Additionally, we can apply standard compression techniques for variable length encodings. This can reduce the client storage to less than 11.2 KB for $n = 2^{20}$ and $d_1 = 128$. However, we note that this compression must be decompressed to be able to handle queries (so the client must use 25 KB of temporary storage when performing queries).

Discussion about Client Storage Size. While the client storage of $\mathsf{SparsePIR}^c$ consists of $O(n/d_1)$ boundary values, the practical client storage is very small. In the concrete example above of $n = 2^{20}$ and $d_1 = 128$, the resulting client storage is ≈ 11 KB. If we consider a practical database size used in prior works (such as [AS16, ACLS18, ALP$^+$21]) of $n = 2^{20}$ 256-byte entries, we note that the client storage is equivalent to 0.004% of the total database size (or 43 entries). Furthermore, the client storage is independent of the size of each entry. For larger entries, the client storage is an even smaller percentage of the database. Furthermore, the client storage is smaller for larger values of d_1. For example, $n = 2^{20}$ and $d_1 = 1024$ requires only 1.8 KB of client storage amounting to 0.0006% of the total database (or 7 entries).

Comparison with $\mathsf{SparsePIR}^g$ and $\mathsf{SparsePIR}$. We note that $\mathsf{SparsePIR}^c$ and $\mathsf{SparsePIR}^g$ obtain near-optimal database sizes in different ways. $\mathsf{SparsePIR}^g$ requires large encoding times to create the database. In contrast, $\mathsf{SparsePIR}^c$ requires long-term client storage. If large encoding times are tolerable (i.e., databases may be created in the background before serving), one should choose $\mathsf{SparsePIR}^g$. If long-term client storage is reasonable, then one can choose $\mathsf{SparsePIR}^c$. If neither slow encoding times or long-term client storage are acceptable, one can simply use $\mathsf{SparsePIR}$ instead. See Section 7 for experimental comparisons between the three options.

Comparison with other PIR with Client Storage Schemes. We note that prior works have studied (non-keyword) PIR with client storage schemes such as [PPY18, CK20, MCR21]. In these works, the goal is to utilize client storage to reduce the server computational costs. For example, the schemes in [PPY18] reduce the number of public-key operations to sub-linear while the constructions in [CK20] get total server time to be sub-linear. In contrast, our goal is to utilize client storage to obtain more efficient keyword PIR schemes that effectively reduces the computational time as well. We leave it as future work to combine the different client storage techniques. We point readers to Section 8 for more references and discussion.

6 Batch Keyword PIR

In this section, we show that our techniques can be extended to batch settings where a client wishes to query a batch of $\ell \geq 1$ queries at one time. Clearly, there is a trivial approach of performing ℓ independent PIR queries, but this ends up being very inefficient. The goal in the batch setting is to design techniques that enable performing batch queries more efficiently.

Revisiting State-of-the-Art Batch PIR. To date, the most practically efficient approach to constructing batch PIR schemes arises from the framework introduced by Angel *et al.* [ACLS18] that utilizes cuckoo hashing to encode both the database and queries. We note that the original framework was designed for doing batch PIR over an n-entry array. The framework did not consider batch keyword PIR over sparse databases. However, we will present the framework with respect to the keyword setting and point out where the framework is inefficient when querying sparse databases.

At a high level, the servers utilizes t random hash functions $H_1, \dots, H_t$ to encode a database $D = \{(k_1, v_1), \dots, (k_n, v_n)\}$. We assume that the t hash functions are shared between all clients and the server. The server will replicate each entry in the database t times and assign them into b_{CH} buckets as follows. For each (k_i, v_i), the server assigns the (k_i, v_i) to the t buckets according to the t hash function evaluations $H_1(k_i), \dots, H_t(k_i)$. The resulting encoding consists of a total $t \cdot n$ key-value pairs across b_{CH} buckets.

For the query algorithm, suppose the client receives a batch of $\ell \geq 1$ queries $(q_1, \dots, q_\ell) \in \mathcal{K}^\ell$. For convenience, it is typically assumed that ℓ queries are distinct. If not, the client may deduplicate locally, perform a batch PIR query for a distinct set of keys and then replicate the blocks as needed to produce the correct answer. The client uses the t hash functions, $H_1, \dots, H_t$, to perform cuckoo hashing that allocates each q_i into exactly one of the b_{CH} buckets specified by the t hash functions $H_1(q_i), \dots, H_t(q_i)$. Cuckoo hashing guarantees that each of the b_{CH} buckets is assigned at most one of $\{q_1, \dots, q_\ell\}$. Finally, the client performs b_{CH} keyword PIR queries into each of the b_{CH} buckets to retrieve the assigned query key (or an arbitrary index if no query key was assigned). We further discuss the necessity of keyword PIR later.

The last step is to pick concrete parameters of the cuckoo hashing scheme: t and b_{CH}. This is an important step as bad parameters can cause a large portion of queries to fail. Note, if a set of ℓ query keys $\{q_1, \dots, q_\ell\}$ cannot be allocated according to cuckoo hashing, then the client's query will fail. Angel *et al.* [ACLS18] suggested picking $t = 3$ and $b_{\mathsf{CH}} = 1.5\ell$ using the experimental evaluation of prior works [CLR17, PSZ18] showing that the concrete failure probability is very small.

This approach achieves much better computational efficiency compared to the trivial approach of performing ℓ keyword PIR queries. The trivial approach would require performing $O(n \cdot \ell)$

	Client Storage	Client Time	Resp. Size	Keyword?
3-Way CH [ACLS18]	$O(1)$	$O(n)$	1.5ℓ	$\times$
3-Way CH [ACLS18]	$O(n)$	$O(1)$	1.5ℓ	✓
3-Way+Batch CH	$O(1)$	$O(1)$	3ℓ	✓
SparseBatchPIR	$O(1)$	$O(1)$	1.5ℓ	✓

Figure 8: Comparison of batch (keyword) PIR schemes.

overhead as each keyword PIR query would require linear overhead. On the other hand, this framework requires only $O(n)$ server computation as only one keyword PIR query is performed per bucket and the total bucket size is $3n$.

Prior Approaches to underlying Keyword PIR. The above framework essentially reduces a batch PIR query for ℓ keys into 1.5ℓ keyword PIR queries into 1.5ℓ buckets whose total size is $3n$. We note that the framework requires keyword PIR even if database was an array and the database's keys were $k_1 = 1, \ldots, k_n = n$. Each of the 1.5ℓ buckets is a database whose keys are a subset of $[n]$. To retrieve the value associated with any key $q \in [n]$ from a bucket, one must use keyword PIR.

To perform keyword PIR, Angel *et al.* [ACLS18] proposed several different options. The simplest approach was for the client to download a direct mapping from each array entry to the physical index within each bucket (or a compressed version using Bloom filter or similar data structures). These approaches would require linear storage on the client side that may be expensive. Furthermore, for databases with small values, these maps may be almost as large as the database. Instead, Angel *et al.* [ACLS18] proposed a very clever trick that enables a client to derive the mapping using the original t hash functions that are shared between the server and the client. The client can simply repeat the server's allocation process to compute the mapping. Therefore, the client can re-create the mapping without large long-term storage.

Going back to batch keyword PIR, we note that the above trick of the client repeating the server's allocation process cannot be used. The key observation is that, for standard PIR, the client knows that the database's keys are exactly $k_1 = 1, \ldots, k_n = n$. Therefore, the client can repeat the server's allocation process identically. When the database's keys are from a sparse universe, the client no longer knows the exact keys that appear in the database. Therefore, the client cannot compute the mapping without knowing the keys present in the database.

Replacing the Keyword PIR. To construct batch keyword PIR schemes, it seems like we must use a true keyword PIR as the trick of succinctly generate client maps can no longer be used. Prior to our work, previous keyword PIR schemes incurred significant additional client storage, doubling response sizes or multiple roundtrips compared to standard PIR.

We show that our SparsePIR families of keyword PIR schemes enable significantly more efficient batch keyword PIR schemes. We obtain SparseBatchPIR by plugging our keyword PIR schemes into the framework of Angel *et al.* [ACLS18], whose communication is identical to batch PIR schemes over n-entry arrays. This is a 2x improvement over using any prior keyword PIR scheme that does not require long-term client storage of linear-sized mappings. We present a comparison of batch (keyword) PIR schemes in Figure 8. Formal descriptions of SparseBatchPIR may be found in Appendix D. In Section 7, we experimentally evaluate our new batch keyword PIR schemes showing significant improvements.

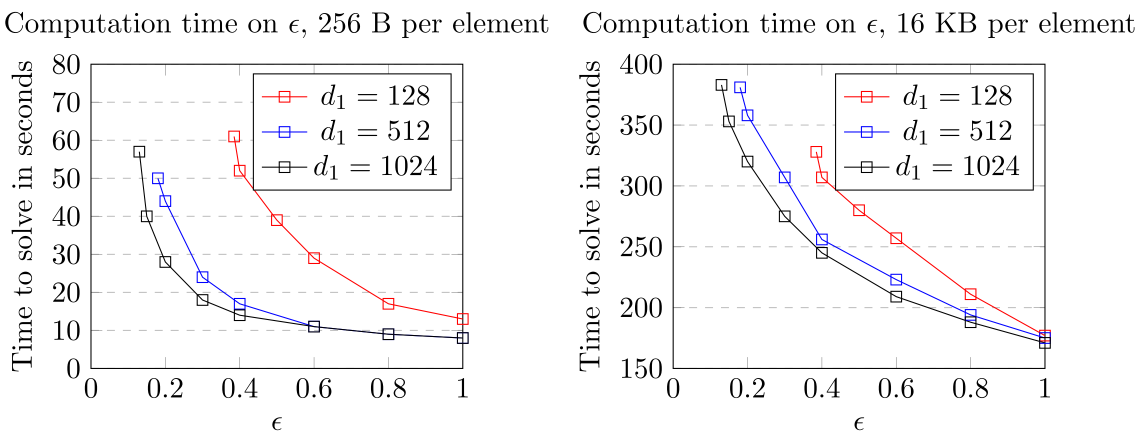


Figure 9: Computation time to solve the SparsePIR linear system vs ϵ on 2^{20} elements for dimension sizes $d_1 = 128, 512$ and 1024. The graphs suggest that the ϵ lower bounds for $d_1 = 128, 512$ and 1024 are roughly 0.38, 0.18, and 0.13 respectively. We chose band parameter $w \in [50, 60]$.

7 Experimental Evaluation

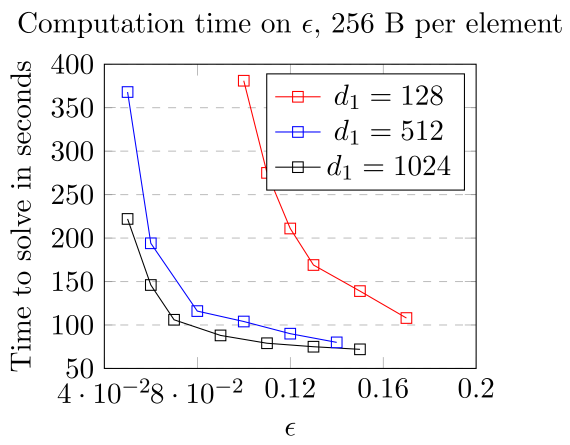
7.1 Implementation

We implemented SparsePIR, SparsePIR^g and SparsePIR^c in C++ on top of several open source PIR schemes: OnionPIR [Oni] and Spiral [Spi]. To compare, we also implement the keyword PIR cuckoo hashing framework of Ali *et al.* [ALP$^+$21] on top of the same PIR schemes. We also implement SparseBatchPIR using the framework of Angel *et al.* [ACLS18]. In total, our implementations required around 2000 lines of codes.

FHE Parameter Selection. Our SparsePIR families of frameworks were constructed carefully to ensure minimal noise growth. As a result, we can directly use the parameters of the underlying standard PIR scheme. The only exception to this is the plaintext modulus, which we round down to the closest prime.

Cuckoo Hashing Optimization. In our cuckoo hashing keyword PIR implementations, we use the optimization that empty entries will be skipped. As empty entries are represented using 0, they can be skipped during server processing. To our knowledge, this optimization was never presented elsewhere. Without this, SparsePIR would actually have better computation as cuckoo hashing keyword PIR needs $(2 + \epsilon)n$ entries while our frameworks use strictly less than $2n$ entries.

Experimental Setup. Our experiments are conducted with machines that are Ubuntu PCs with 12 cores, 3.7 GHz Intel Xeon W-2135 and 64 GB of RAM. We use the AVX2 and AVX-512 instruction sets with SIMD instructions enabled. All our experiments use single-thread execution. Reported results are the average of at least 10 experimental trials with standard deviation less than 10% of the means. Monetary costs are computed using Amazon EC2 pricing of t2.2xlarge instances [AWS] of \$0.09 per GB of outbound traffic and \$0.014 per CPU hour at the time. We do not report client times as they are very small (see [ACLS18, ALP$^+$21, MW22]). Following prior works, we define rate as the ratio of the retrieved record size to the response size.



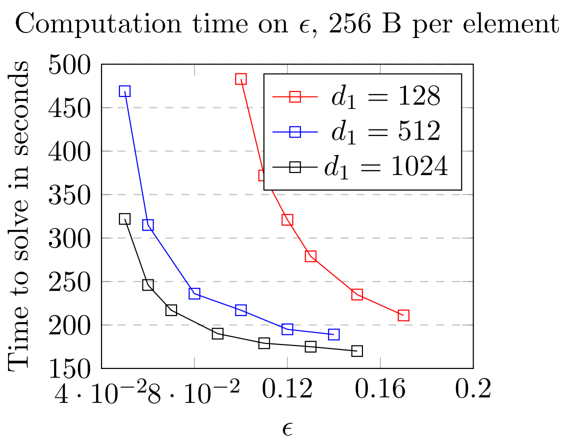


Figure 10: Computation time to solve the SparsePIR^g linear system vs ϵ on 2^{20} elements for dimension sizes $d_1 = 128, 512$ and 1024. We chose band parameter $w \in [85, 120]$.

d_1	SparsePIR	SparsePIR^g	SparsePIR^c
128			
ϵ	0.38	0.1	< 0.03
Client Storage	0 B	0 B	11.1 KB
512			
ϵ	0.18	0.06	< 0.03
Client Storage	0 B	0 B	3.4 KB
1024			
ϵ	0.12	0.05	< 0.03
Client Storage	0 B	0 B	1.8 KB

Figure 11: Comparisons of SparsePIR, SparsePIR^g and SparsePIR^c of achieveable ϵ for $n = 2^{20}$ elements.

7.2 SparsePIR Family Parameters

We choose our parameters to accomodate the recursion dimensions of prior works. Recall Onion-PIR [MCR21] used $d_1 = 128$ dimensions and Spiral [MW22] used $d_1 = 512$. Recall that our frameworks encode a database of n key-value pairs into a serving database of size $(1 + \epsilon)n$.

For SparsePIR and SparsePIR^g, see Figure 9 and Figure 10 for the time to create an encoding of size $(1 + \epsilon)n$. For SparsePIR, we choose band parameter $w \in [50, 60]$ for all our experiments, which seem to provide good trade-offs between the computation time and the solvability of the random band linear systems. Similarly, we choose band parameter $w \in [85, 120]$ for SparsePIR^g. From these empirical results, we can choose appropriate values of ϵ. In Figure 11, we compare SparsePIR, SparsePIR^g and SparsePIR^c in terms of the trade offs between ϵ and the client storage. For SparsePIR^c, we used bzip2 for the compression algorithm.

7.3 Comparisons

Cuckoo Hashing Keyword PIR. Figure 12 compares the cuckoo hashing [ALP+21] and our families of SparsePIR frameworks applied to various PIR protocols. SparsePIR exhibits clear advan-

Database	Onion CH-PIR	Onion SparsePIR	Onion SparsePIR^g	Onion SparsePIR^c	Spiral CH-Pir	Spiral SparsePir	Spiral SparsePIR^g	Spiral SparsePIR^c
$2^{20} \times 256$ B								
Query Size	63 KB	63 KB	63 KB	63 KB	14 KB	14 KB	14 KB	14 KB
Response Size	254 KB	**127 KB**	**127 KB**	**127 KB**	42 KB	**21 KB**	**21 KB**	**21 KB**
Computation	3.03 s	3.04 s	3.10 s	3.05	1.41 s	1.44 s	1.45 s	1.42 s
Rate	0.001	**0.002**	**0.002**	**0.002**	0.006	**0.012**	**0.012**	**0.012**
Server Cost	\$0.000034	**\$0.000027**	\$0.000029	\$0.000028	\$0.0000091	\$0.0000074	\$0.0000074	**\$0.0000073**
$2^{17} \times 30$ KB								
Query Size	63 KB	63 KB	63 KB	63 KB	14 KB	14 KB	14 KB	14 KB
Response Size	254 KB	**127 KB**	**127 KB**	**127 KB**	172 KB	**86 KB**	**86 KB**	**86 KB**
Computation	32.25 s	41.91 s	32.24 s	32.28 s	10.02 s	11.57 s	10.21 s	10.18 s
Rate	0.118	**0.236**	**0.236**	**0.236**	0.174	**0.349**	**0.349**	**0.349**
Server Cost	\$0.00015	\$0.00017	**\$0.00014**	**0.00014**	\$0.000054	\$0.000052	**\$0.000047**	**\$0.000047**
$2^{14} \times 100$ KB								
Query Size	63 KB	63 KB	63 KB	63 KB	14 KB	14 KB	14 KB	14 KB
Response Size	1016 KB	**508 KB**	**508 KB**	**508 KB**	484 KB	**242 KB**	**242 KB**	**242 KB**
Computation	14.43 s	17.32 s	15.14 s	15.10 s	4.93 s	5.91 s	5.11 s	5.17 s
Rate	0.098	**0.197**	**0.197**	**0.197**	0.207	**0.413**	**0.413**	**0.413**
Server Cost	\$0.00014	\$0.00011	**\$0.00010**	**\$0.00010**	\$0.000061	\$0.000044	**\$0.000041**	**\$0.000041**

Figure 12: Comparison of cuckoo hashing (CH) and SparsePIR frameworks for various PIR protocols.

tage over the cuckoo hashing framework on the response size and the rate. This is expected because cuckoo hashing framework requires the server to send two ciphertexts, while SparsePIR sends only one ciphertext. Cuckoo hashing framework has a slight advantage in computation cost with the optimization of skipping empty entries. However, the small additional server computation time is a worthwhile trade-off for the substantial improvement in the response size as evidenced in the reduced server costs. Due to packing, we obtain smaller ϵ for smaller entries (e.g. 256 bytes).

We note that both SparsePIR^g and SparsePIR^c can reduce the server computation compared to SparsePIR because of the small ϵ both encoding schemes can achieve.

Constant-Weight Keyword PIR. In Figure 13, we compare SparsePIR with keyword PIR using constant-weight equality operators [MK22] using their open source implementation [CWR]. We observe that SparsePIR framework instantiated on the Spiral protocol outperforms constant-weight keyword PIR for both communication and computation, demonstrating the benefits of SparsePIR. Our experiments use similar settings as [MK22] with 16-bit keyword lengths and database sizes of $2^{10}, 2^{12}$ and 2^{14}.

Batch Keyword PIR. In Figure 14, we present experiments for batch keyword PIR using the batch PIR framework of Angel *et al.* [ACLS18]. As the underlying keyword PIR, we use the cuckoo hashing framework [ALP+21] and our SparsePIR framework. We use Spiral as the blackbox PIR protocol for both frameworks. We run our experiments on 2^{20} 288-byte entries, which are also used in [ACLS18]. We used $t = 3$ hash functions and the number of buckets to be 1.5ℓ where ℓ is the batch size. We observe that SparsePIR halves response size and reduces server costs in exchange for small increase in computation.

Database	CW Keyword PIR	Spiral SparsePir
$2^{14} \times 256$ B		
Query Size	216 KB	**14 KB**
Response Size	103 KB	**11 KB**
Computation	280.34 s	**0.75 s**
Rate	0.0025	**0.023**
Server Cost	$0.0011	**$0.0000039**
$2^{12} \times 30$ KB		
Query Size	216 KB	**14 KB**
Response Size	206 KB	**82 KB**
Computation	73.68 s	**0.93 s**
Rate	0.146	**0.366**
Server Cost	$0.0003	**$0.000011**
$2^{10} \times 100$ KB		
Query Size	216 KB	**14 KB**
Response Size	515 KB	**236 KB**
Computation	20.98 s	**0.89 s**
Rate	0.194	**0.424**
Server Cost	$0.00013	**$0.000024**

Figure 13: Comparison of SparsePIR and constant-weight (CW) keyword PIR with 16-bit keyword length [MK22].

8 Related Work

Early PIR Schemes. Single-server PIR was introduced by Kushilevitz and Ostrovsky [KO97]. Several follow-ups built PIR using number-theoretic assumptions [Pai99, DJ01, Lip05] and the phi-hiding assumption [CMS99, GR05].

Lattice-based PIR. In recent years, many works studied practically efficient PIR schemes using lattice-based homomorphic encryption as first done in [ABFK16] with many follow-ups including [AS16, ACLS18, GH19, PT20, ALP+21, MCR21, AYA+21, MW22].

Keyword PIR. The notion of keyword PIR was introduced by Chor *et al.* [CGN98] that required multiple rounds. Freedman *et al.* [FIPR05] also considered keyword PIR with database privacy using oblivious PRFs. Ali *et al.* [ALP+21] presented a keyword PIR using cuckoo hashing. Mahdavi and Kerschbaum [MK22] presented a construction from constant-weight equality operators.

Batch PIR. Several works [BIM00, IKOS04, GKL10, LG15, Hen16] have studied batch PIR with the goal of efficiently querying multiple records at once. Recent works [AS16, ACLS18] introduced *probabilistic batch codes* that are the most efficient batch PIR frameworks to date.

Multi-Server PIR. A long line of work has also studied PIR with multiple, non-colluding servers. Early work considered information-theoretic security [CKGS98, Efr09, DG15]. Recent work showed concretely efficient constructions using function secret sharing [GI14, BGI16, HH19].

Other PIR Variants. The notion of PIR with preprocessing was introduced by Beimel *et al.* [BIM00] where the server holds a public hint string. This was also studied as doubly-efficient PIR [BIPW17, CHR17] with sub-linear online time using strong obfuscation assumptions.

Stateful PIR introduced by Patel *et al.* [PPY18] allows clients to compute query-independent hints in an offline phase to help speed up online queries that was also studied in [MCR21]. A

Batch Size (ℓ)	**Spiral CH-BatchPir**	**Spiral SparseBatchPir**
16		
Query Size	24 KB	24 KB
Response Size	48 KB	**24 KB**
Computation	0.56 s	0.64 s
Rate	0.0000051	**0.000010**
Server Cost	\$0.0000063	**\$0.0000045**
64		
Query Size	24 KB	24 KB
Response Size	48 KB	**24 KB**
Computation	0.36 s	0.42 s
Rate	0.0000051	**0.000010**
Server Cost	\$0.0000055	**\$0.0000037**
256		
Query Size	24 KB	24 KB
Response Size	38 KB	**19 KB**
Computation	0.29 s	0.33 s
Rate	0.0000064	**0.000013**
Server Cost	\$0.0000044	**\$0.0000029**

Figure 14: Comparison of cuckoo hashing [ALP+21] and SparsePIR keyword PIR frameworks with Spiral [MW22] for batch PIR queries. The database size is 2^{20} 288-byte entries. The reported numbers are the amortized cost of retrieving a single entry.

recent line of work starting with Corrigan-Gibbs and Kogan [CK20] presented constructions with sub-linear online times. Multiple works have further studied the topic [KC21, SACM21, CHK22, DPC23, Yeo23, HHCG+23].

9 Conclusions

In this paper, we proposed SparsePIR, SparsePIR^g and SparsePIR^c, frameworks that build keyword PIR from lattice-based standard PIR schemes. Our schemes reduce the response size by at least 2x compared to prior keyword PIR constructions. In essence, our frameworks show that keyword PIR may be built with identical communication and computation costs as standard PIR. We also show our keyword PIR frameworks may also be used to also halve the response overhead of batch keyword PIR.

References

[ABFK16] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. XPIR: Private information retrieval for everyone. *PoPETs*, 2016(2):155–174, April 2016.

[ACLS18] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy*, pages 962–979. IEEE Computer Society Press, May 2018.

[ALP+21] Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. Communication-computation trade-offs in PIR. In Michael Bailey and Rachel

Greenstadt, editors, *USENIX Security 2021*, pages 1811–1828. USENIX Association, August 2021.

[AS16] Sebastian Angel and Srinath Setty. Unobservable communication over fully untrusted infrastructure. In *OSDI 16*, pages 551–569, 2016.

[AWS] EC2 On-Demand Pricing. https://aws.amazon.com/ec2/pricing/on-demand/.

[AYA$^+$21] Ishtiyaque Ahmad, Yuntian Yang, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. Addra: Metadata-private voice communication over fully untrusted infrastructure. In *OSDI 21*, 2021.

[BDG15] Nikita Borisov, George Danezis, and Ian Goldberg. DP5: A private presence service. *PoPETs*, 2015(2):4–24, April 2015.

[BGI16] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1292–1303. ACM Press, October 2016.

[BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.

[BIM00] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers computation in private information retrieval: PIR with preprocessing. In Mihir Bellare, editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 55–73. Springer, Heidelberg, August 2000.

[BIPW17] Elette Boyle, Yuval Ishai, Rafael Pass, and Mary Wootters. Can we access a database both locally and privately? In Yael Kalai and Leonid Reyzin, editors, *TCC 2017, Part II*, volume 10678 of *LNCS*, pages 662–693. Springer, Heidelberg, November 2017.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 868–886. Springer, Heidelberg, August 2012.

[CGN98] Benny Chor, Niv Gilboa, and Moni Naor. Private information retrieval by keywords. Cryptology ePrint Archive, Report 1998/003, 1998. https://eprint.iacr.org/1998/003.

[CHK22] Henry Corrigan-Gibbs, Alexandra Henzinger, and Dmitry Kogan. Single-server private information retrieval with sublinear amortized time. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 3–33. Springer, Heidelberg, May / June 2022.

[CHR17] Ran Canetti, Justin Holmgren, and Silas Richelson. Towards doubly efficient private information retrieval. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017, Part II*, volume 10678 of *LNCS*, pages 694–726. Springer, Heidelberg, November 2017.

[CK20] Henry Corrigan-Gibbs and Dmitry Kogan. Private information retrieval with sublinear online time. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 44–75. Springer, Heidelberg, May 2020.

[CKGS98] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *Journal of the ACM (JACM)*, 45(6):965–981, 1998.

[CLR17] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1243–1255. ACM Press, October / November 2017.

[CMS99] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 402–414. Springer, Heidelberg, May 1999.

[Coo00] Colin Cooper. On the rank of random matrices. *Random Structures & Algorithms*, 16(2):209–232, 2000.

[CWR] Constant-weight PIR. https://github.com/rasoulam/constant-weight-pir.

[DG15] Zeev Dvir and Sivakanth Gopi. 2-server PIR with sub-polynomial communication. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th ACM STOC*, pages 577–584. ACM Press, June 2015.

[DJ01] Ivan Damgård and Mats Jurik. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In Kwangjo Kim, editor, *PKC 2001*, volume 1992 of *LNCS*, pages 119–136. Springer, Heidelberg, February 2001.

[DPC23] Alex Davidson, Gonçalo Pestana, and Sofía Celi. FrodoPIR: Simple, scalable, single-server private information retrieval. 2023.

[DRRT18] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. PIR-PSI: Scaling private contact discovery. *Proceedings on Privacy Enhancing Technologies*, 2018(4):159–178, 2018.

[DW19] Martin Dietzfelbinger and Stefan Walzer. Efficient gauss elimination for near-quadratic matrices with one short random block per row, with applications. In *ESA 2019*, 2019.

[Efr09] Klim Efremenko. 3-query locally decodable codes of subexponential length. In Michael Mitzenmacher, editor, *41st ACM STOC*, pages 39–44. ACM Press, May / June 2009.

[FIPR05] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 303–324. Springer, Heidelberg, February 2005.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. https://eprint.iacr.org/2012/144.

[GCM+16] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath Setty, Lorenzo Alvisi, and Michael Walfish. Scalable and private media consumption with popcorn. In *NSDI 16*, pages 91–107, 2016.

[GH19] Craig Gentry and Shai Halevi. Compressible FHE with applications to PIR. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019, Part II*, volume 11892 of *LNCS*, pages 438–464. Springer, Heidelberg, December 2019.

[GI14] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 640–658. Springer, Heidelberg, May 2014.

[GKL10] Jens Groth, Aggelos Kiayias, and Helger Lipmaa. Multi-query computationally-private information retrieval with constant communication rate. In Phong Q. Nguyen and David Pointcheval, editors, *PKC 2010*, volume 6056 of *LNCS*, pages 107–123. Springer, Heidelberg, May 2010.

[GLM16] Matthew Green, Watson Ladd, and Ian Miers. A protocol for privately reporting ad impressions at scale. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1591–1601. ACM Press, October 2016.

[GR05] Craig Gentry and Zulfikar Ramzan. Single-database private information retrieval with constant communication rate. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *ICALP 2005*, volume 3580 of *LNCS*, pages 803–815. Springer, Heidelberg, July 2005.

[Hen16] Ryan Henry. Polynomial batch codes for efficient IT-PIR. *PoPETs*, 2016(4):202–218, October 2016.

[HH19] 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. *PoPETs*, 2019(4):112–131, October 2019.

[HHCG$^+$23] Alexandra Henzinger, Matthew M. Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. One server for the price of two: Simple and fast single-server private information retrieval. In *USENIX Security 23 (to appear)*, 2023.

[IKOS04] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Batch codes and their applications. In László Babai, editor, *36th ACM STOC*, pages 262–271. ACM Press, June 2004.

[KC21] Dmitry Kogan and Henry Corrigan-Gibbs. Private blocklist lookups with checklist. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 875–892. USENIX Association, August 2021.

[KLDF16] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle: An efficient communication system with strong anonymity. *PoPETs*, 2016(2):115–134, April 2016.

[Knu05] Donald Ervin Knuth. *The Art of Computer Programming, volume 4, fasicle 2.* 2005.

[KO97] Eyal Kushilevitz and Rafail Ostrovsky. Replication is NOT needed: SINGLE database, computationally-private information retrieval. In *38th FOCS*, pages 364–373. IEEE Computer Society Press, October 1997.

[LG15] Wouter Lueks and Ian Goldberg. Sublinear scaling for multi-client private information retrieval. In Rainer Böhme and Tatsuaki Okamoto, editors, *FC 2015*, volume 8975 of *LNCS*, pages 168–186. Springer, Heidelberg, January 2015.

[Lip05] Helger Lipmaa. An oblivious transfer protocol with log-squared communication. In *International Conference on Information Security*, pages 314–328. Springer, 2005.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 1–23. Springer, Heidelberg, May / June 2010.

[MCR21] Muhammad Haris Mughees, Hao Chen, and Ling Ren. OnionPIR: Response efficient single-server PIR. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2292–2306. ACM Press, November 2021.

[MK22] Rasoul Akhavan Mahdavi and Florian Kerschbaum. Constant-weight PIR: Single-round keyword PIR via constant-weight equality operators. In *USENIX Security 22*, pages 1723–1740, Boston, MA, 2022.

[Mol05] Michael Molloy. Cores in random hypergraphs and boolean formulas. *Random Structures & Algorithms*, 27(1):124–135, 2005.

[MOT$^+$11] Prateek Mittal, Femi G. Olumofin, Carmela Troncoso, Nikita Borisov, and Ian Goldberg. PIR-tor: Scalable anonymous communication using private information retrieval. In *USENIX Security 2011*. USENIX Association, August 2011.

[MW22] Samir Jordan Menon and David J Wu. Spiral: Fast, high-rate single-server PIR via FHE composition. In *2022 IEEE Symposium on Security and Privacy*, 2022.

[OI07] Rafail Ostrovsky and William E. Skeith III. A survey of single database PIR: Techniques and applications. Cryptology ePrint Archive, Paper 2007/059, 2007. https://eprint.iacr.org/2007/059.

[Oni] OnionPIR. https://github.com/mhmughees/Onion-PIR.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 223–238. Springer, Heidelberg, May 1999.

[Por09] Ely Porat. An optimal bloom filter replacement based on matrix solving. In *International Computer Science Symposium in Russia*, pages 263–273. Springer, 2009.

[PPY18] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Private stateful information retrieval. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1002–1019. ACM Press, October 2018.

[PS16] Boris Pittel and Gregory B Sorkin. The satisfiability threshold for k-XORSAT. *Combinatorics, Probability and Computing*, 25(2):236–268, 2016.

[PSZ18] Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable private set intersection based on OT extension. *ACM Transactions on Privacy and Security (TOPS)*, 21(2):1–35, 2018.

[PT20] Jeongeun Park and Mehdi Tibouchi. SHECS-PIR: Somewhat homomorphic encryption-based compact and scalable private information retrieval. In Liqun Chen, Ninghui Li, Kaitai Liang, and Steve A. Schneider, editors, *ESORICS 2020, Part II*, volume 12309 of *LNCS*, pages 86–106. Springer, Heidelberg, September 2020.

[SACM21] Elaine Shi, Waqar Aqeel, Balakrishnan Chandrasekaran, and Bruce M. Maggs. Puncturable pseudorandom sets and private information retrieval with near-optimal online bandwidth and time. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 641–669, Virtual Event, August 2021. Springer, Heidelberg.

[SC07] Radu Sion and Bogdan Carbunar. On the practicality of private information retrieval. In *NDSS 2007*. The Internet Society, February / March 2007.

[Spi] Spiral. https://github.com/menonsamir/spiral.

[Wie86] Douglas Wiedemann. Solving sparse linear equations over finite fields. *IEEE transactions on information theory*, 32(1):54–62, 1986.

[WZPM16] David J. Wu, Joe Zimmerman, Jérémy Planul, and John C. Mitchell. Privacy-preserving shortest path computation. In *NDSS 2016*. The Internet Society, February 2016.

[Yeo23] Kevin Yeo. Lower bounds for (batch) pir with private preprocessing. In *Eurocrypt 2023 (to appear)*, 2023.

[YP21] Kevin Yeo and Sarvar Patel. Protecting your device information with private set membership. https://security.googleblog.com/2021/10/protecting-your-device-information-with.html, 2021.

A Keyword PIR Definitions

Definition 1 (One-Round Keyword PIR). *A one-round keyword private information retrieval (PIR) scheme over key universe $\mathcal{K}$ and L-bit blocks from $\{0,1\}^L$ consists of the efficient algorithms $\Pi = (\mathsf{Init}, \mathsf{Encode}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Decrypt})$ satisfying:*

- $(\mathsf{ck}, \mathsf{sk}) \leftarrow \mathsf{Init}(1^\lambda)$*: The initialization algorithm receives the security parameter λ, the size of the database n and a database $D = \{(k_1, v_1), \ldots, (k_n, v_n)\}$. The output are the public parameters* prms*, a client key* ck*, a server key* sk *and an encoding of the database E.*
- $(\mathsf{prms}, E) \leftarrow \mathsf{Encode}(D)$*: The encode algorithm receives the database $D = \{(k_1, v_1), \ldots, (k_n, v_n)\}$ and outputs an encoding of the database E as well as a set of parameters for querying using the encoded database. For convenience, we will assume that all n keys are distinct, $k_i \neq k_j$ for all $i \neq j \in [n]$.*

```
$\mathbf{Expt}^{\mathsf{client}}_{b,\mathcal{A}}(\lambda, D)$
$(\mathsf{ck}, \mathsf{sk}) \leftarrow \mathsf{Init}(1^\lambda, D)$
$(\mathsf{prms}, E) \leftarrow \mathsf{Encode}(D)$
$R \leftarrow \emptyset$
For $i = 1, 2, \ldots, \mathsf{poly}(\lambda, |D|)$ :
  - $(q_0, q_1) \leftarrow \mathcal{A}(\mathsf{prms}, \mathsf{sk}, E, R)$
  - $R \leftarrow R \cup \{\mathsf{Query}(\mathsf{prms}, \mathsf{ck}, q_b)\}$
Return $b' \leftarrow \mathcal{A}(\mathsf{prms}, \mathsf{sk}, E, R)$
```

Figure 15: Experiment for privacy of client queries.

- $(\mathsf{st}, \mathsf{req}) \leftarrow \mathsf{Query}(\mathsf{prms}, \mathsf{ck}, q)$*: The query algorithm receives the public parameters* prms*, client key* ck *and a query* $q \in \mathcal{K}$ *and outputs a state* st *and the request* req *to be sent to the server.*
- $\mathsf{resp} \leftarrow \mathsf{Answer}(\mathsf{prms}, \mathsf{sk}, E, \mathsf{req})$*: The answer algorithm receives the public parameters* prms*, server key* sk*, encoding of the database* E *and the request* req *and outputs a response* resp*.*
- $v \leftarrow \mathsf{Decrypt}(\mathsf{prms}, \mathsf{ck}, \mathsf{st}, \mathsf{resp})$*: The process algorithm receives the public parameters* prms*, client key* ck*, state* st *and response* resp *and outputs the value associated to the queried key* q*.*

Additionally, the following guarantees must be satisfied:

Correctness. *For all databases* $D = \{(k_1, v_1), \ldots, (k_n, v_n)\}$ *where* $k_1, \ldots, k_n \in \mathcal{K}$ *are distinct,* $v_1, \ldots, v_n \in \{0,1\}^L$ *and for all query keys* $q \in \mathcal{K}$*, the probability that the client's output when querying* q *is not* $D[q]$ *is at most* $\mathsf{negl}(\lambda)$*.*

Client Query Privacy. *We define client query privacy with respect to the experiment defined in Figure 15 parameterized by a bit* $b \in \{0,1\}$ *and an adversary* $\mathcal{A}$*. For all databases* D *and all stateful, efficient adversaries* $\mathcal{A}$*,* $\Pr[\mathbf{Expt}^{\mathsf{client}}_{b,\mathcal{A}}(\lambda, D) = b] \leq 1/2 + \mathsf{negl}(\lambda)$*.*

Definition 2 (One-Round Batch Keyword PIR)**.** *A one-round batch keyword private information retrieval (PIR) scheme over key universe* $[n]$*, L-bit blocks from* $\{0,1\}^L$ *and batches of* $\ell \geq 1$ *queries consists of the efficient algorithms* $\Pi = (\mathsf{Init}, \mathsf{Encode}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Decrypt})$ *such that* Π *is a one-round keyword PIR with* $\mathcal{K}$ *with the following differences:*

- $(\mathsf{st}, \mathsf{req}) \leftarrow \mathsf{Query}(\mathsf{prms}, \mathsf{ck}, Q)$*: The query algorithm receives a query set* $Q = (q_1, \ldots, q_\ell) \in \mathcal{K}^\ell$ *as opposed to a single query.*
- $(v_1, \ldots, v_\ell) \leftarrow \mathsf{Decrypt}(\mathsf{prms}, \mathsf{ck}, \mathsf{st}, \mathsf{resp})$*: The process algorithm must output* ℓ *b-bit blocks as opposed to a single block.*

Additionally, the following guarantees must be satisfied:

Correctness. *For all databases* $D = \{(k_1, v_1), \ldots, (k_n, v_n)\}$ *where* $k_1, \ldots, k_n \in \mathcal{K}$ *are distinct,* $v_1, \ldots, v_n \in \{0,1\}^L$ *and for all batch queries* $Q \in \mathcal{K}^\ell$*, the probability that the client's output when querying* Q *is not* $\{D[q]\}_{q \in Q}$ *is at most* $\mathsf{negl}(\lambda)$*.*

Client Query Privacy. *We define client query privacy with respect to the experiment defined in Figure 15 parameterized by a bit* $b \in \{0,1\}$ *and an adversary* $\mathcal{A}$ *where queries are now from* $\mathcal{K}^\ell$*. For all databases* D *and all stateful, efficient adversaries* $\mathcal{A}$*,* $\Pr[\mathbf{Expt}^{\mathsf{client}}_{b,\mathcal{A}}(\lambda, D) = b] \leq 1/2 + \mathsf{negl}(\lambda)$*.*

Algorithm 10 PartitionBoundariesSetting algorithm

```
Input: $D = \{(k_1, v_1), \ldots, (k_n, v_n)\}$: database
Output: $B$: List of partition boundary values
  $B_0 = 0, B_{\frac{n}{d_1}} = |U| - 1$ where $U$ is range of $F(\mathsf{K}_r, k_i)$
  for $i = 1, \ldots, n$ do
      $X_i \leftarrow F(\mathsf{K}_r, k_i)$
  $X_{(i)}$ = sorted $X_i$'s
  for $i = 1, \ldots, \frac{n}{d_1} - 1$ do
      $B_i = X_{(i \cdot d_1)}$
```

Algorithm 11 PartitionSelection algorithm

```
Input: $k$: keyword
Output: $i$: part containing keyword $k$
  $X \leftarrow F(\mathsf{K}_r, k)$
  if X is 0 then
      Return 1                                        ▷ Border condition
  Return $i$ such that $B_{i-1} < X \leq B_i$
```

B Compression and Oblivious Expansion

An idea, introduced by Angel *et al.* in SealPIR [ACLS18], was to compress multiple plaintext values into a single ciphertext. Plaintexts are actually degree-N polynomials where one can compress multiple bits into a single plaintext polynomial that is later encrypted into a single ciphertext. Given this single ciphertext, the server may utilize a protocol to obliviously expand back into multiple ciphertexts encrypting a single coefficient of the compressed plaintext. Using this approach, many state-of-the-PIR schemes enable clients to upload only a single ciphertext when querying.

A key insight is that the oblivious expansion may be applied on vectors beyond just indicator vectors (as pointed out by Ali *et al.* [ALP+21]). In particular, one can compress and encrypt sequences of t arbitrary values in a single ciphertext that may later be obliviously expanded by the server into t ciphertexts encrypting the individual values. This has been utilized by previous PIR schemes to fit multiple indicator vectors into a single ciphertext. In our work, we will rely on this property as we will upload encryptions of vectors that have more than one non-zero entry.

C Exact Partition – Compressed List Size

The partition boundary setting and selection algorithm are outlined in Algorithm 10 and 11. The list B become part of prms at setup time and the selection algorithm is used by the client during query time. This only requires $O(n/d_1 \log U)$ bits where U is range of $F(\mathsf{K}_r, \cdot)$. For $n = 2^{20}$, $d_1 = 128$, $|U| = 2^{64}$, then this requires 64 KB.

Compression due to Rounding and Truncation. If we can drop the least significant bits of partition boundary values (B_i) then we could perhaps compress or send less bits. However, we have to be careful that the rounding and truncation includes the current B_i value but does omit the next neighboring value $X_{d_1 \cdot (i+1)}$. That is if we drop too many least significant bits then we will end up including the neighboring point.

We show an example of how we can drop the least significant bits of partition boundaries without meaningfully changing the partitioning. Below, we show the binary representation of an example partition boundary value (B_i) and its next higher neighboring point value (N_i). Then, we present a rounded version with several least significant bits dropped that does not affect the partitioning. We see that the first 19 most significant bits of B_i and N_i are the same. By changing the 20th bit of B_i from 0 to 1 in R_i and zeroing the remaining least significant bits, we include the intended B_i, now on the left of R_i on the number line and exclude the neighboring point, now on the right of R_i.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28			
B$_i$	1	0	0	1	1	0	1	1	1	1	1	1	0	0	0	1	0	0	1	0	1	1	1	1	0	0	1	0	·	·	·
N$_i$	1	0	0	1	1	0	1	1	1	1	1	1	0	0	0	1	0	0	1	1	0	1	0	0	0	0	1	1	·	·	·
R$_i$	1	0	0	1	1	0	1	1	1	1	1	1	0	0	0	1	0	0	1	1	0	0	0	0	0	0	0	0	·	·	·

We want to round and drop just enough least significant bit bits in the B_i value such that a neighboring value does not erroneously get included in the partition. To determine the correct length, we can use some estimations that we confirmed through experimentation. A neighboring value has a mean distance of $|U|/n$ since there are n values uniformly distributed on the entire range. So we would expect $\log(|U|/n) = \log U - \log n$ least significant bits to differ and $\log n$ most significant bits to remain the same. In our concrete example, $\log U$ is 64 and n is 2^{20}, thus we can expect to keep $\log n$ or 20 bits to be the same. With rounding, we will keep approximately 21 bits per boundary value with the additional 1 bit needed per above description to rule out the neighboring value.

We note that the above only depends on the value of n and not the size of the output universe of the function $F(\mathsf{K}_r, \cdot)$. If the size of the output range U of the $F(\mathsf{K}_r, \cdot)$ function was increased to 2^{128} or 2^{256} instead of 2^{64}, we would still only send 21 bits per partition. So 21K bytes is a significant improvement over potentially having to send 64KB to 256KB if we naively encoded the partitioning.

We justify the above bounds by using order statistics. The sorted list of points $X_{(1)}, X_{(2)} \ldots X_{(n)}$ are the order statistics where $X_{(k)}$ is the k-th smallest value and $U_{(1)}, U_{(2)} \ldots U_{(n)}$ are order statistics when underlying is a uniform distribution. The distribution of the k-th order statistics $U_{(k)}$ is described by the beta distribution, $Beta(k, n+1-k)$. The gap between B_i and N_i is distributed as the gap between $U_{d_1 \cdot i}$ and $U_{d_1 \cdot (i+1)}$ which has the same distribution as $U_{(1)}$ or $Beta(1, n)$. This has the mean $1/(n+1)$ assuming a uniform range over $[0, 1]$. Thus we can approximate the mean distance as $2^{\log U}/(n+1)$ apart where U is the range interval (e.g. 2^{64} in our example), and we can approximate that $\log U - \log n$ least significant bits will be different or $\log n$ most significant bits will be the same on average between the boundary value and its neighboring point value. We also get a similar expression if we directly calculate $E[\log X]$ where X is from the beta distribution.

Compression due to Differencing. Instead of sending R_i directly, we can send the difference $R_i - R_{i-1}$ and allow the client to reconstruct R_i from the differences. In practice, we see that this reduces 12-13 bits from each partition boundary description on average. Thus, for the above example, we show in figure below the preceding value. We notice that most of the leading 11 bits are identical and when we take a difference (D_i) between the pair, we see that the leading 13 most significant bits are zeroed.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28			
R$_{i-1}$	1	0	0	1	1	0	1	1	1	1	1	0	1	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	·	·	·
R$_i$	1	0	0	1	1	0	1	1	1	1	1	1	0	0	0	1	0	0	1	1	0	0	0	0	0	0	0	0	·	·	·
D$_i$														1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	·	·	·

There are only n/d_1 boundary values on the range with two neighboring boundary values on average separated by a distance of $(U \cdot d_1)/n$ and thus will differ in $\log U - (\log n - \log d_1)$ least significant bits or will on average not change in the $\log n - \log d_1$ most significant bits. Thus, differencing on average will further reduce the bits needed per boundary value by $(\log n - \log d_1)$. With $d_1 = 128$, and $n = 20$, neighboring boundary values will on average not change in the 13 most significant bits and we can expect the difference to be zero in the 12 to 13 most significant bits. This means typically only bits 13 to 20 from most significant bit have changing values after truncation and differencing. That is, approximately 7-8 bits need to be used to encode each partition boundary (as we have also seen in our experimental evaluation).

As before, we can also justify the above bounds by using order statistics. The distribution of the gap between B_i and B_{i-1} is the same as the distribution of the gap between $U_{d_1 \cdot i}$ and $U_{d_1 \cdot (i-1)}$ which has the same distribution as U_{d_1} or $Beta(d_1, n - d_1 + 1)$. This has the mean distance of $d_1/(n+1)$ assuming a uniform range over $[0, 1]$. Thus, we can approximate the distance as $(U \cdot d_1)/(n+1)$ and the least significant bits that will change as $\log U + \log d_1 - \log n$. Therefore, on average, $\log U - (\log U + \log d_1 - \log n) = \log n - \log d_1$ most significant bits will stay the same. We also get a similar expression by directly calculating for $E[\log X]$ where X is a beta distributed random variable. Thus, differencing will save $\log n - \log d_1$ bits per boundary.

Overall compressed size. Recall, we first needed to keep $\log n$ most significant bits per boundary value, B_i. Using truncation and differencing, we save an additional $\log n - \log d_1$ most significant bits out of the $\log n$ bits needed after truncation. So, in total, we need $\log n - (\log n - \log d_1)$ or $\log d_1$ bits per boundary value. Since there are n/d_1 boundary values, the total cost is around $n/d_1 \cdot \log d_1$ bits. This is a significant improvement over $n \cdot \log(n/d_1)$ required by explicit sending. Asymptotically, this can be better than linear storage, for example, if $d_1 = \sqrt{n}$, we need only $O(\sqrt{n} \cdot \log n)$ storage. Going back to our concrete example of $n = 2^{20}$, $d_1 = 128$, $|U| = 2^{64}$, we show that only 11 KB is required to store the partitioning. The above analysis is only calculating the expected size of the encoding of a partitioning. It is possible that the size is larger, but our experimentation shows that the encoding size is typically not very far from the average.

D Batch Keyword PIR Pseudocode

We denote the batch keyword PIR enabling batch queries of size ℓ built on top of SparsePIR as SparseBatchPIR. Similarly, we can also build batch keyword PIR using SparsePIR^o by replacing SparsePIR usage everywhere. For more details, we point readers back to Section 6.

Algorithm 12 SparseBatchPIR.Init algorithm

```
Input: $1^\lambda$: security parameter.
Output: $(\mathsf{ck}, \mathsf{sk})$: client and server key.
  $(\mathsf{ck}, \mathsf{sk}) \leftarrow \mathsf{SparsePIR.Init}(1^\lambda)$
  return $(\mathsf{ck}, \mathsf{sk})$
```

Algorithm 13 SparseBatchPIR.Encode algorithm

```
Input: $D = \{(k_1, v_1), \ldots, (k_n, v_n)\}$: database
Output: $(\mathsf{prms}, \mathbf{E})$: parameters and encoding
  /* $F_{\mathsf{CH}}$ will output elements in $\{1, \ldots, b_{\mathsf{CH}}\}$. */
  Sample PRF keys $\mathsf{K}_1^{\mathsf{CH}}, \mathsf{K}_2^{\mathsf{CH}}, \mathsf{K}_3^{\mathsf{CH}}$.
  $b_{\mathsf{CH}} \leftarrow 1.5\ell$
  $D_1 \leftarrow \emptyset, \ldots, D_{b_{\mathsf{CH}}} \leftarrow \emptyset$
  for $i = 1, \ldots, n$ do
      for $j = 1, \ldots, 3$ do
          $x \leftarrow F_{\mathsf{CH}}(\mathsf{K}_j^{\mathsf{CH}}, k_i)$
          $D_x \leftarrow D_x \cup \{(k_i, v_i)\}$
  for $i = 1, \ldots, b_{\mathsf{CH}}$ do
      $(\mathsf{prms}_i, \mathbf{E}_i) \leftarrow \mathsf{SparsePIR.Encode}(D_i)$
  $\mathsf{prms} \leftarrow (\mathsf{K}_1^{\mathsf{CH}}, \mathsf{K}_2^{\mathsf{CH}}, \mathsf{K}_3^{\mathsf{CH}}, \mathsf{prms}_1, \ldots, \mathsf{prms}_{b_{\mathsf{CH}}})$
  return $(\mathsf{prms}, (\mathbf{E}_1, \ldots, \mathbf{E}_{b_{\mathsf{CH}}}))$
```

Algorithm 14 SparseBatchPIR.Query algorithm

```
Input: $(\mathsf{prms}, \mathsf{ck}, Q = \{q_1, \ldots, q_\ell\})$: parameters and the batch query.
Output: $(\mathsf{st}, \mathsf{req})$: temporary state and request.
  $b_{\mathsf{CH}} \leftarrow 1.5\ell$
  Parse $\mathsf{prms} = (\mathsf{K}_1^{\mathsf{CH}}, \mathsf{K}_2^{\mathsf{CH}}, \mathsf{K}_3^{\mathsf{CH}}, \mathsf{prms}_1, \ldots, \mathsf{prms}_{b_{\mathsf{CH}}})$.
  /* Each $X_i$ is an element in $Q$ or $\perp$. */
  Execute cuckoo hashing allocation on $Q$ using $F(\mathsf{K}_1^{\mathsf{CH}}, \cdot), \ldots, F(\mathsf{K}_3^{\mathsf{CH}}, \cdot)$ to obtain $X_1, \ldots, X_{b_{\mathsf{CH}}}$.
  for $i = 1, \ldots, b_{\mathsf{CH}}$ do
      if $X_i = \perp$ then
          $(\mathsf{st}_i, \mathsf{req}_i) \leftarrow \mathsf{SparsePIR.Query}(\mathsf{prms}_i, \mathsf{ck}, 1)$
      else
          $(\mathsf{st}_i, \mathsf{req}_i) \leftarrow \mathsf{SparsePIR.Query}(\mathsf{prms}_i, \mathsf{ck}, X_i)$
  $\mathsf{st} \leftarrow (\mathsf{st}_1, \ldots, \mathsf{st}_{b_{\mathsf{CH}}}, X_1, \ldots, X_{b_{\mathsf{CH}}})$
  return $(\mathsf{st}, (\mathsf{req}_1, \ldots, \mathsf{req}_{b_{\mathsf{CH}}}))$
```

Algorithm 15 $\mathsf{SparseBatchPIR.Answer}$ algorithm

Input: $(\mathsf{prms}, \mathsf{sk}, \mathbf{E}, \mathsf{req})$: parameters, server key, encoded databases and the request.

Output: resp: the response to the request.

```
Parse $\mathsf{req} = (\mathsf{req}_1, \ldots, \mathsf{req}_{b_{\mathsf{CH}}})$
Parse $\mathbf{E} = (\mathbf{E}_1, \ldots, \mathbf{E}_{b_{\mathsf{CH}}})$
for $i = 1, \ldots, b_{\mathsf{CH}}$ do
    $\mathsf{resp}_i \leftarrow \mathsf{SparsePIR.Answer}(\mathsf{sk}, \mathbf{E}_i, \mathsf{req}_i)$
return $(\mathsf{resp}_1, \ldots, \mathsf{resp}_{b_{\mathsf{CH}}})$
```

Algorithm 16 $\mathsf{SparseBatchPIR.Decrypt}$ algorithm

Input: $(\mathsf{prms}, \mathsf{ck}, \mathsf{st}, \mathsf{resp})$: parameters, client key, temporary state and response.

Output: A: set of output values

```
Parse $\mathsf{prms} = (\mathsf{st}_1, \ldots, \mathsf{st}_{b_{\mathsf{CH}}}, X_1, \ldots, X_{b_{\mathsf{CH}}})$
Parse $\mathsf{resp} = (\mathsf{resp}_1, \ldots, \mathsf{resp}_{b_{\mathsf{CH}}})$
$A \leftarrow \emptyset$
for $i = 1, \ldots, b_{\mathsf{CH}}$ do
    $y \leftarrow \mathsf{SparsePIR.Decrypt}(\mathsf{ck}, \mathsf{st}_i, \mathsf{resp}_i)$
    Parse $y = (\mathsf{id}, v)$
    $v_i \leftarrow \perp$
    if $X_i \neq \perp \wedge \mathsf{id} = F(\mathsf{K}_r, X_i)$ then
        $v_i \leftarrow v$
    if $X_i \neq \perp$ then
        $A \leftarrow A \cup \{(X_i, v_i)\}$
return $A$
```