Pai: Private Retrieval with Constant Online Time, Communication, and Client-Side Storage for Data Marketplace

Shuaishuai Li
Zhongguancun Laboratory
Beijing, China
genielss@yeah.net

Weiran Liu
Liqiang Peng
Alibaba Group
weiran.lwr@alibaba-inc.com
plq270998@alibaba-inc.com

Cong Zhang
Institute for Advanced Study, BNRist, Tsinghua University
Beijing, China
zhangcong@mail.tsinghua.edu.cn

Xinwei Gao
Aiping Liang
Lei Zhang
Alibaba Group
xinwei.gao.7@gmail.com
aiping.lap@alibaba-inc.com
zongchao.zl@taobao.com

Dongdai Lin
Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
School of Cyber Security, University of Chinese Academy of Sciences,
Beijing, China
ddlin@iie.ac.cn

Yuan Hong
University of Connecticut
Storrs, CT, USA
yuan.hong@uconn.edu

ABSTRACT

Data marketplace is a critical platform for trading high-quality and private-domain data. A basic functionality in the data marketplace is that a data seller (as a server) owns a private key-value database and provides private query services to data buyers (as clients). This relates to Private Information Retrieval (PIR) by Keyword with symmetric privacy, abbreviated to KSPIR. In the context of PIR, Client-preprocessing PIR supports fast online retrievals by introducing a one-time, query-independent offline phase with linear offline communication, promising for deployment in the data marketplace. However, there are remaining challenges. First, the client-side storage and the online costs are still relatively large. Second, current implementations only consider public array databases (cannot handle private or key-valued databases). Third, existing solutions are somewhat intricate for non-expert PIR developers.

To address these significant deficiencies, we propose a novel client-preprocessing PIR framework Pai, which only requires constant online time, communication, and client-side storage. Building upon Pai, we present its KSPIR variant PaiKSPIR. We also explore an alternative variant of KSPIR named Chargeable KSPIR (CKSPIR) for the data marketplace application where the server seeks payment from the client for retrieval. We have undertaken comprehensive implementations and conducted extensive experiments for Pai. The online query time is only about 1ms with 1KB communication overhead for large key-value databases (e.g., $n = 2^{24}$). Given the superior online time and storage, our protocol is well-suited in the data marketplace for even real-time key-value retrievals.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/alibaba-edu/mpc4j.

1 INTRODUCTION

The demand for data exchange increases for better distributing data value. In today's big data era, different enterprises and organizations may own a lake of data they collect or generate [4, 48]. Meanwhile, various parties (e.g., data science companies and institutions) may seek access to such data for diverse data-driven tasks. This has given rise to the data marketplace that facilitates transactional interactions between data buyers and data sellers [5, 47].

A very basic and simple functionality in the data marketplace is the data retrieval. Specifically, the data seller (as a server) owns a private key-value database and provides query services to its data buyers (as clients). Such functionality is the fundamental query in data marketplace [5, 44]. Implementation becomes straightforward if the server can directly see the queried key and answer its corresponding value. However, in many scenarios, the queried key itself is also sensitive and necessitates the secrecy protection to the data seller. For example, the key may be a disease/medicine name in the healthcare research, a business trend in e-commerce applications, or a threat vulnerability in the network security defense. Revealing such information to the data seller may lead to legal prohibition, trade secret leakage, and potential collusion problems. In short, data retrieval solutions require privacy *both* for the data seller and the data buyers.

1.1 Background and Related Works

The problem of private data retrieval is closely associated with the cryptographic primitive of Private Information Retrieval (PIR) [1]. Standard PIR allows a client to obtain an entry from a *public array* store hosted on a server without revealing which entry is retrieved. PIR has been one of the main research directions in cryptography since its inception by Chor et al. [13, 14]. The naive PIR solution involves the server transmitting the entire database to the client for each query. However, this method incurs significant communication overhead, rendering it impractical for large-scale databases. A long line of works [2, 3, 7–9, 11, 18, 20, 25, 26, 28, 29, 31, 34–37, 39, 40] have been devoted to designing PIRs with low communication costs.

Many variants of PIR have also been studied. Chor et al. [12] considered PIR by Keyword (KPIR), where the database is defined as a set of n key-value pairs and the client uses a search key to retrieve the corresponding value. Ahmad et al. formally introduced KPIR, allowing clients to retrieve the value corresponding to a key from a public key-value store [1]. Gertner et al. [27] introduced PIR with Symmetric privacy (SPIR), which additionally requires that the client can only know its desired entry, thus preserving server privacy. The data marketplace needs a solution supporting keyword queries while protecting server privacy. This PIR variant is called PIR by Keyword with Symmetric privacy (KSPIR), aka., Labeled Private Set Intersection (LPSI) [10, 15].

Although PIR and its variants have been studied for nearly three decades, their efficiency is still not satisfactory due to two bottlenecks. First, there is a fundamental barrier that the amount of server computation will inevitably be linear in the size of the database, formally proved by Beimel et al. [6]. Intuitively, the server must touch every single entry during some query; otherwise, the server learns that the queried entry is not one of the untouched entries. Second, in order to achieve sublinear communication cost (otherwise, the naive solution is sufficient), the client needs to somehow encrypt its query while allowing the server to *obliviously* match the queried entry. This involves Homomorphic Encryption (HE) or similar cryptographic primitives that allow the server to do computations on the encrypted query [46]. HE-like primitives inevitably invoke either algebra operations on large fields [42] or high-degree polynomial operations on lattices [24], both of which are somewhat inefficient, specially for real-time applications.

Several directions have been explored to circumvent the fundamental barrier of linear computation using HE. Beimel et al. [6] proposed to use preprocessing to deal with the lower bound of linear computation. Concretely, the protocol is divided into an *offline phase* and an *online phase*. The offline phase is a one-time and query-independent process, and its one-time nature enables the cost to be amortized over multiple (adaptive) queries. Furthermore, the query-independent property allows the execution of the offline phase to occur before the client decides its queries. The main goal of PIR with preprocessing is to facilitate a fast online phase, with a specific focus on achieving sublinear online computation.

Client-Preprocessing Model. To date, the concretely efficient preprocessing model is the *client-preprocessing model*. In this model, the client downloads and stores "hints" from the server during the offline phase. Then, the client can efficiently execute many online queries with the help of these hints. Although the offline phase can be fairly expensive or may even require downloading the entire database, the amortized cost per query is extremely low, making the client-preprocessing model promising. The client-preprocessing model was initially introduced by Patel, Persiano, and Yeo [43], with a concrete scheme that still needs linear server computation per online query. Corrigan-Gibbs and Kogan [17] proposed the first client-preprocessing PIR scheme with amortized sublinear server computation. Follow-up works continue to make further improvements [16, 30, 32, 33, 45, 49]. Very recently, Zhou et al. [50] proposed an extremely simple efficient client-preprocessing PIR scheme Piano (short for Private Information Access NOw). The simplicity lies in that the construction is completely self-contained and does not invoke any existing PIR scheme as a building block. The efficiency lies in that the scheme only needs lightweight cryptographic primitives such as Pseudo-Random Functions (PRFs) (that can be accelerated with AES-NI instructions), with $\tilde{O}(\sqrt{n})$[1] client storage and $\tilde{O}(\sqrt{n})$ online communication (both the client request and the server response) and server computation per query. Mughees, I and Ren [38] further optimized the online query construction and proposed a simple and practical client-preprocessing PIR (for simplicity of notations, we name their scheme as Spam, short for Simple and Practical AMortized), achieving a server response with constant overhead. Both Piano and Spam offer fast online retrievals and are potential solutions for real-time data retrievals in the marketplace.

1.2 Challenges

However, there are challenges for deploying client-preprocessing PIR like Piano and Spam on the data marketplace.

Efficiency. Their concrete efficiency is still unsatisfactory due to that their asymptotic online complexity is $\tilde{O}(\sqrt{n})$. This affects key-value retrievals for real-time systems, in which the response time is asked to be as short as possible.

Functionality. Their implementations only consider public array databases (cannot handle private or key-valued databases). Although there exist solutions to transform PIR to KSPIR, the additional efficiency costs are unknown.

Incomprehensibility. Their schemes involve relatively complicated procedures such as "hint" generation and compressed encoding that are hard to understand and implement for non-expert PIR developers. In our experience, customers are willing to deploy such schemes if they can understand and implement them on their own.

As a result, we ask the following question:

Can we construct a concretely efficient and easy-to-understand KSPIR protocol with $o(\sqrt{n})$ or even $\tilde{O}(1)$ online time, communication, and client-side storage in the client-preprocessing model?

1.3 Contributions

We present a novel client-preprocessing PIR framework Pai (as the conventional symbol name π for permutation) with asymptotically near-optimal (up to polylogarithmic factors) efficiency features.

PIR with Significantly Reduced Online Costs. Pai circumvents the lower bound by encoding the database with the help of the client in the offline phase, i.e., encoding by encryption and permutation (that is why we name it Pai), at the price of more offline communication cost and more server-side storage cost for each client. Similar to Piano [50] and Spam [38], the offline communication cost of Pai is still $\tilde{O}(n)$. However, this allows Pai to enjoy constant online time, communication, and client-side storage, enabling blazing fast

[1]Throughout this work, we use $\tilde{O}(\cdot)$ to hide polylogarithmic terms, i.e., for any function $f(n)$, we have $\tilde{O}(f(n)) = O(f(n) \cdot \mathrm{poly}(\log n))$.

online query response that is well-suited in the data marketplace for even real-time data retrievals.

The linear offline communication implies that the client must have $O(n)$ transient storage before submitting its query. Similar to Piano and Spam, Pai also uses a streaming algorithm to achieve $\tilde{O}(\sqrt{n})$ bandwidth and storage in the offline phase.

Efficiently Supporting KSPIR. Pai can be extended to KSPIR with high efficiency, some of which were briefly discussed in Piano [50]. Specifically, Ali et al. [2] introduced a method to convert PIR to KPIR, and Freedman et al. [23] presented a method to convert KPIR to KSPIR. Combining these two ideas, we can derive a transformation from PIR to KSPIR. However, to our best knowledge, for the most efficient PIR protocols in the client-preprocessing model (e.g., Piano and Spam), there is no implementation to verify the efficiency of the KSPIR protocol resulting from the conversions.

We apply the transformations of [2, 23] to Pai, Piano, and Spam, respectively. Then, we obtain three new KSPIRs, denoted as PaiKSPIR, PianoKSPIR, and SpamKSPIR. We have implemented all of these three KSPIRs, and compared them with the state-of-the-art (SOTA) KSPIR without preprocessing by Cong et al. [15]. Our results concretely show that KSPIR can also enjoy fast online query response in the client-preprocessing model.

Chargeable KSPIR. We also explore an alternative variant of KSPIR, which arises from the data marketplace application context where the server seeks *payment from the client for retrieval*. Note that in KSPIR, the client may fail to retrieve a value (if its search key is not in the key set of the database). In this case, it is unreasonable to charge the client for the failed retrieval attempt.

To address this issue, we let the server know whether the client successfully obtains a value and charge if and only if the retrieval is successful. We introduce CKSPIR as a novel framework to model and address this particular variant of KSPIR. We note that any KSPIR (e.g., [10, 15]) can be converted to CKSPIR by letting the client send a bit indicating whether it successfully retrieves a value with its key. Based on the idea of Pai, we construct a more efficient CKSPIR protocol PaiCKSPIR by inherently allowing the server to know whether the retrieved key matches some key in the database.

Easy-to-understand for Non-expert Users. A side advantage of Pai and its KSPIR/CKSPIR variants is that they are very easy to understand. Existing solutions either involve advanced cryptographic tools like HE [15], or somewhat complicated "hint" generation and consumption procedures [38, 50]. Instead, Pai only involves encryption, permutation, and operations that are similar to the well-known Diffie-Hellman key exchange [22]. This makes Pai easy to understand and implement even for non-expert PIR developers.

Open-source Implementations and Evaluations. We implemented Pai and its KSPIR variant in Java. The source code is available at https://github.com/alibaba-edu/mpc4j. We conduct experiments with variants of database sizes $n = 2^{20}, 2^{22}, 2^{24}$ and entry sizes 64, 128, 256. Pai enjoys good concrete efficiency, e.g., 8.8 - 91.8× better online communication cost and 2.5 - 8.8× better online time than SOTA client-preprocessing PIRs (Paino and Spam). PaiKSPIR with client-preprocessing enjoys extremely fast online queries, at least three orders of magnitude improvement compared with the SOTA KSPIR without preprocessing [15]. Concretely, the online query time is only about 1ms with 1KB communications for large key-value stores (e.g., $n = 2^{24}$). The online communication and the online time of our PaiCKSPIR is even better than PaiKSPIR by removing unnecessary protection operations for disallowing the server to know whether there is a match in the query.

Trade-offs. Different from the (C)KSPIR where the server only stores one (perhaps encoded) database for all clients, a limitation of our Pai and its variants is the requisite for the server to store an (encoded) database for each of its clients. This limitation is sometimes unacceptable for classic (C)KSPIR applications, e.g., safe browsing [30] and private blocklist lookups [30]. However, since data sellers and data buyers are typically enterprises and companies in the data marketplace, such a limitation can be accepted when performing one-to-one private retrieval services. We can conceptually think that Pai and its variants enable *encrypted caches* in the offline phase to provide extremely fast online queries for real-time requirements. For situations where $\tilde{O}(\sqrt{n})$ online query cost is acceptable, one can deploy (C)KSPIR variants of Piano and Spam, for which we also provide concrete implementations.

2 PRELIMINARIES

Math Notations. Let λ be the computational security parameter and κ be the statistical security parameter. For any integer n, we use $[n]$ to represent the set $\{1, \ldots, n\}$. For any two distributions $\mathsf{D}_0, \mathsf{D}_1$, if no probabilistic polynomial time (PPT) algorithm can distinguish them, then we say that D_0 and D_1 are computationally indistinguishable, denoted $\mathsf{D}_0 \approx_c \mathsf{D}_1$.

Protocol Notations. All of our protocols proceed between a server and a client (the data seller and the data buyer in the data marketplace, respectively). In PIR, we use $\mathsf{DB} = (v_1, \ldots, v_n)$ to denote the database. In KSPIR and CKSPIR, the database DB is a set of n key-value pairs, i.e., $\mathsf{DB} = \{(k_i, v_i)\}_{i \in [n]}$. Let l_k and l_v denote the length of k_i and v_i, respectively. Namely, we assume each k_i belongs to $\mathcal{K} = \{0,1\}^{l_k}$ and each v_i belongs to $\mathcal{V} = \{0,1\}^{l_v}$. Finally, we assume that the keys in the database are distinct, which implies that $l_k \geq \log n$.

2.1 Problem Formulation

We consider the most fundamental query in the data marketplace where a data seller (acts as the server) wants to provide a data retrieval service for its private database of the form $\{(k_i, v_i)\}_{i \in [n]}$. The data buyer (ask as a client) uses a key k to ask for the value v_i corresponding to $k_i \in \{k_1, \ldots, k_n\}$ if $k = k_i$. We require that the server cannot get any information of the key k, while the client can only obtain the value v_i corresponding to $k_i \in \{k_1, \ldots, k_n\}$ if $k = k_i$, or $\perp$ if $k \notin \{k_1, \ldots, k_n\}$, but no other information about the server's database. This relates to Private Information Retrieval (PIR) by Keyword with symmetric privacy, abbreviated to KSPIR.

We also consider the standard PIR as the base of KSPIR, where the database is a public array $(v_1, \ldots, v_n)$, and the client queries with some index $i \in [n]$. The server should not know any information about i at this point, and the goal of the client is to obtain v_i.

We finally consider a special application context where the server seeks payment from the client for successful retrieval. To achieve this goal, we let the server additionally know a Boolean value of whether $k \in \{k_1, \dots, k_n\}$, representing the client queried the information. This captures the situation where it is not reasonable for the client to charge if the value is not successfully queried (i.e., $k \notin \{k_1, \dots, k_n\}$). We name it Chargeable KSPIR (CKSPIR).

2.2 Adversarial Model

We consider a semi-honest (or passive) adversary. That is, the parties will not deviate from the protocol. The adversary attacks the protocol by corrupting one party (either the server or the client) and using its internal state to infer information about the inputs of the other party. Moreover, we assume the adversary cannot perform any side-channel attack. We also assume that the adversary cannot break any standard cryptographic assumption, such as the Decisional Diffie-Hellman (DDH) assumption [22].

2.3 Design Goals

PIR. In PIR, the server takes a database $\mathsf{DB} = (v_1, \dots, v_n)$ as input, and the client takes an index $i \in [n]$ as input. PIR has the following properties.

`Correctness.` At the end, the client outputs v_i.

`Client-Privacy.` The server knows nothing about i. Concretely, let $\mathsf{View}_{\mathsf{ser}}(\mathsf{DB}, i)$ be the view of the server in the protocol. There exists a PPT algorithm Sim taking DB as input such that

$$\mathsf{View}_{\mathsf{ser}}(\mathsf{DB}, i) \approx_c \mathsf{Sim}(\mathsf{DB}).$$

KSPIR. In KSPIR, the server takes a database $\mathsf{DB} = \{(k_i, v_i)\}_{i \in [n]}$ as input, and the client takes a search key k as input. KSPIR has the following properties.

`Correctness.` At the end, the client outputs $z = v_i$ if there exists some $i \in [n]$ such that $k_i = k$; otherwise, the client outputs $z = \perp$.

`Server-Privacy.` The client knows nothing about DB except information contained in the key k it requests and the entry z it obtains. Let $\mathsf{View}_{\mathsf{cli}}(\mathsf{DB}, k)$ be the view of the client in the protocol. There exists a PPT algorithm Sim taking k, z as inputs such that

$$\mathsf{View}_{\mathsf{cli}}(\mathsf{DB}, k) \approx_c \mathsf{Sim}(k, z).$$

`Client-Privacy.` The server knows nothing about k. Concretely, let $\mathsf{View}_{\mathsf{ser}}(\mathsf{DB}, k)$ be the view of the server in the protocol. There exists a PPT algorithm Sim taking DB as input such that

$$\mathsf{View}_{\mathsf{ser}}(\mathsf{DB}, k) \approx_c \mathsf{Sim}(\mathsf{DB}).$$

CKSPIR. CKSPIR additionally outputs a bit to the server, which lets the server know whether the client successfully obtained a value. In CKSPIR, the server takes a database $\mathsf{DB} = \{(k_i, v_i)\}_{i \in [n]}$ as input, and the client takes a search key k as input. CKSPIR has the following properties.

`Correctness.` If there exists some $i \in [n]$ such that $k_i = k$, then the server outputs $b = 1$, and the client outputs $z = v_i$. Otherwise, the server outputs $b = 0$, and the client outputs $z = \perp$.

`Server-Privacy.` The client knows nothing about DB except the information contained in k, z. Let $\mathsf{View}_{\mathsf{cli}}(\mathsf{DB}, k)$ be the view of the client in the protocol. Then there exists a PPT algorithm Sim taking k, z as inputs such that

$$\mathsf{View}_{\mathsf{cli}}(\mathsf{DB}, k) \approx_c \mathsf{Sim}(k, z).$$

`Client-Privacy.` The server knows nothing about k except the information contained in b. Let $\mathsf{View}_{\mathsf{ser}}(\mathsf{DB}, k)$ be the view of the server in the protocol. There exists a PPT algorithm Sim taking DB, b as inputs such that

$$\mathsf{View}_{\mathsf{ser}}(\mathsf{DB}, k) \approx_c \mathsf{Sim}(\mathsf{DB}, b).$$

2.4 Oblivious Pseudo-Random Function

Oblivious Pseudo-Random Function (OPRF) [23] is a two-party protocol, where the server has a master key mk for a PRF F, and the client holds an input x. The protocol requires that the server obtains nothing, and the client obtains the value $\mathsf{F}(mk, x)$ and nothing else. In this work, we will use the OPRF protocol of [19] that defines $\mathsf{F}(mk, x) = \mathsf{H}(x)^{mk}$, where H is a hash function modeled as a random oracle (RO). This protocol is secure under the DDH assumption against semi-honest adversaries.

Public Parameters. Let $\mathbb{G}$ be a DDH-hard cyclic group with prime order p, and $\mathsf{H} : \{0,1\}^* \to \mathbb{G}$ a hash function modeled as a random oracle.

Input. The server has a master key $mk \in \mathbb{Z}_p^*$, and the client has an input $x \in \{0,1\}^*$.

1. The client first samples $\beta \in \mathbb{Z}_p^*$ and sends $\mathsf{H}(x)^\beta$ to the server, who responds with $(\mathsf{H}(x)^\beta)^{mk}$.

2. The client outputs $\mathsf{H}(x)^{mk} = ((\mathsf{H}(x)^\beta)^{mk})^{1/\beta}$.

3 PAI FRAMEWORK

The data marketplace requires (C)KSPIR to support private data queries on key-value database, allowing the client to flexibly express its query request. Although *PIR only allows the client to query a public database for an index*, it is the base for further constructing (C)KSPIR. Our Pai framework also follows such paradigm. Therefore, we start with a brief introduction to Pai in the PIR setting.

Overview of Pai. Figure 1 illustrates the framework of Pai. The core design of Pai focuses on the offline phase (preprocessing), including three key steps as below.

- *Step 1: Encoding and Permuting the Database.* Two random encoding algorithms are used to encode the indices and entries, respectively. The database is simultaneously randomly permuted, making the encoded database pseudorandom from the server's view while allowing the client to retrieve and decode in the online phase.
- *Step 2: Encoding the Database with the Client.* We introduce offline interaction so that the database encoding and permutation are jointly executed by the server and client.
- *Step 3: Reducing the Bandwidth via a Streaming Algorithm.* To achieve a low bandwidth in the offline interaction, the database is represented in two dimensions so that the rows and columns of the database are encoded and permuted respectively. As will be subsequently demonstrated, this approach reduces the offline bandwidth to $\tilde{O}(\sqrt{n})$, same as in Piano and Spam.

Finally, during the online phase, the client sends the encoded index to the server. The server finds and returns the matching

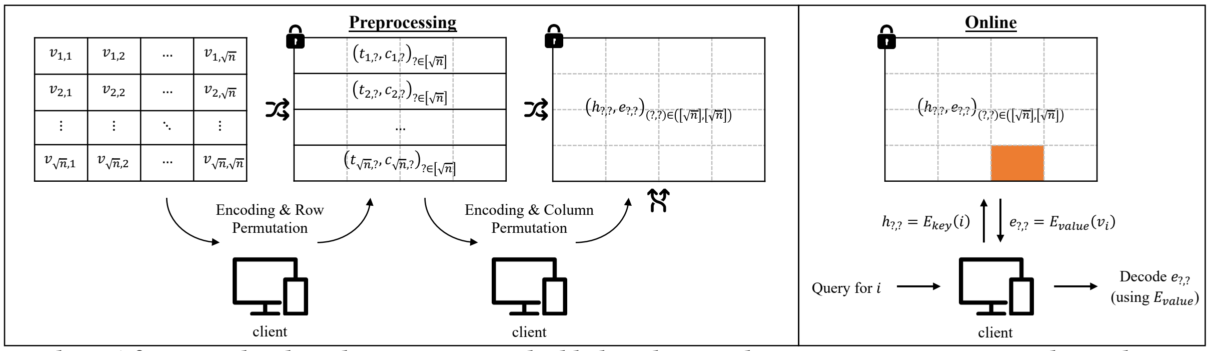

Preprocessing
Online
Encoding & Row Permutation
Encoding & Column Permutation
client
client
client
$h_{?,?} = E_{key}(i)$
$e_{?,?} = E_{value}(v_i)$
Query for i
Decode $e_{?,?}$ (using E_{value})

Figure 1: The Pai framework. Three key steps are embedded in the encoding & row permutation and encoding & column permutation that are jointly performed by the server (data seller) and client (data buyer).

encoded value, which can be efficiently decoded and by the client to derive the final result.

From PIR to KSPIR. We introduce cuckoo hashing table [2] and OPRF [23] to concretely convert our Pai (as well as Piano and Spam) to corresponding KSPIR. More details are deferred to Section 5.

From PIR to CKSPIR. We can upgrade our PIR protocol to obtain a more efficient CKSPIR. More details are deferred to Section 5.3.

4 PAI CONSTRUCTION AND ANALYSIS

4.1 Pai Construction

Now we are ready to describe our Pai construction in the PIR setting. We later transform Pai into (C)KSPIR, supporting private key-value database queries in the data marketplace applications.

Pai is designed in the client-preprocessing model and consists of an offline phase and an online phase. In the offline phase, the client helps the server encode the database to enable a very fast online phase. Here we demonstrate how to design the aforementioned three key steps in the offline phase. The detailed construction is shown in Figure 2.

Step 1: Encoding and Permuting the Database. The foundational concept behind our database encoding approach involves both permutation and encryption of the database. To be precise, given a public array database denoted as v_j for $j \in [n]$, the encoding procedure is articulated as follows.

(1) Choose a random permutation π over $[n]$.
(2) Choose two encoding algorithms $\mathsf{E}_{\mathsf{index}}$ and $\mathsf{E}_{\mathsf{entry}}$, which are used to encode the indices and entries, respectively.
(3) Then, the encoded database is $\{(h_j, e_j)\}_{j\in[n]}$, where $h_j = \mathsf{E}_{\mathsf{index}}(\pi(j)), e_j = \mathsf{E}_{\mathsf{entry}}(v_{\pi(j)})$.

The main goal of the above encoding is to enable "secure" retrieval against the original database by using "insecure" retrieval against the securely encoded (key-value) database. Specifically, we aim to design the following online phase.

(1) When the client decides its search index i, it computes $h = \mathsf{E}_{\mathsf{index}}(i)$ as the query message.
(2) The server finds $j \in [n]$ such that $h = h_j$ and returns $e = e_j$ to the client as the response.
(3) The client decodes e to v and outputs v.

To ensure the correctness of the protocol, it is imperative that the client produces an output $v = v_i$. Two primary conditions must be satisfied: (1) the response message is $e_{\pi^{-1}(i)}$ (which is the encoding of v_i), and (2) extracting v_i from $e_{\pi^{-1}(i)}$ is easy. To meet this, the encoding functions are subjected to the following requirements.

- $\mathsf{E}_{\mathsf{index}}$ is deterministic and collision-resistant.
- $\mathsf{E}_{\mathsf{entry}}$ is invertible. That is, the client can easily inverse v from $e = \mathsf{E}_{\mathsf{entry}}(v)$.

For the client-privacy guarantee, recall that the query message is $h = h_j = \mathsf{E}_{\mathsf{index}}(i)$ with $j = \pi^{-1}(i)$, and the response message is $e_j = e_{\pi^{-1}(i)}$. We need to guarantee that the server obtains nothing about i by seeing j, h_j, and e_j with the knowledge of all v_i. We have the following requirements.

- π^{-1} is a random permutation unknown to the server.
- $\mathsf{E}_{\mathsf{index}}$ is pseudorandom to the server, which guarantees that h_j leaks nothing about i.
- $\mathsf{E}_{\mathsf{entry}}$ is pseudorandom to the server, which guarantees that e_j leaks nothing about v_i.

For the deterministic, collision-resistant, and pseudorandom encoding function $\mathsf{E}_{\mathsf{index}}$, we choose a Pseudo-Random Permutation (PRP) (e.g., AES) with the client holding the PRP key. For the invertible and pseudorandom encoding function $\mathsf{E}_{\mathsf{entry}}$, we instantiate it with a Symmetric Key Encryption (SKE) scheme that is secure against chosen-plaintext attack (CPA-secure) with the client holding the SKE key. A typical CPA-secure SKE is AES with CTR-mode.

Step 2: Encoding the Database with the Client. We now discuss how to encode the database with the client holding the corresponding keys. While it is feasible for the client to initially transmit the encrypted keys and subsequently permit the server to employ HE for computing the encoded database, this would lead to a rather high computational overhead.

To make our offline phase practical, we follow the basic idea of Piano and Spam and let the client download the entire database and encode it. Specifically, the client chooses a PRP key pk and an encryption key ek of a CPA-secure SKE scheme. Moreover, the client chooses a random permutation π over $[n]$. Then, it encrypts the database as $\{(h_j, e_j)\}_{j\in[n]}$, where $h_j = \mathsf{P}(\pi(j)), e_j = \mathsf{SKE.Enc}(ek, v_{\pi(j)})$. Finally, it sends $\{(h_j, e_j)\}_{j\in[n]}$ to the server. Compared with Piano and Spam, the client further needs to upload the encoded database, which incurs extra offline communication.

Figure 2: The construction of Pai.

Protocol Π_{Pai}: PIR with $\tilde{O}(1)$ Client-Side Storage and Online Time

Input: The server $\mathcal{S}$ has a database $\mathsf{DB} = (v_1, \ldots, v_n)$, and the client C has an index $i \in [n]$.
Output: C outputs v_i.

Offline Phase: $\mathcal{S}$ represents the database as a matrix $(v_{j_0,j_1})_{j_0,j_1 \in [\sqrt{n}]}$, where $v_{j_0,j_1} = v_{(j_0-1)\sqrt{n}+j_1}$. $\mathcal{S}$ and C agree on an CPA-secure SKE scheme (SKE.Gen, SKE.Enc, SKE.Dec) and a PRP $\mathsf{P} : \{0,1\}^\lambda \times \{0,1\}^l \to \{0,1\}^l$ with $l \geq \lceil \log n \rceil$. Then, C chooses two random permutations π_0, π_1 over $[\sqrt{n}]$, an SKE key ek, and two PRP keys pk_0, pk_1. We note that π_0 is used for permuting the rows, and π_1 is used for permuting the columns. $\mathcal{S}$ and C interact to encode the database in a streaming way.

(1) Row-Permuting. For each $j_1 \in [\sqrt{n}]$, $\mathcal{S}$ and C perform the following steps.
  (a) $\mathcal{S}$ sends $\{v_{j_0,j_1}\}_{j_0 \in [\sqrt{n}]}$ to C.
  (b) For all $j_0 \in [\sqrt{n}]$, C sets $j = (\pi_0(j_0) - 1)\sqrt{n} + j_1$, and then C computes $t_{j_0,j_1} = \mathsf{P}(pk_0, j)$ and $c_{j_0,j_1} = \mathsf{SKE.Enc}(ek, v_j)$. Then, C sends $\{(t_{j_0,j_1}, c_{j_0,j_1})\}_{j_0 \in [\sqrt{n}]}$ to $\mathcal{S}$.
  (c) Finally, C deletes $\{v_{j_0,j_1}, (t_{j_0,j_1}, c_{j_0,j_1})\}_{j_0 \in [\sqrt{n}]}$ from its local storage.

(2) Column-Permuting. For each $j_0 \in [\sqrt{n}]$, $\mathcal{S}$ and C perform the following steps.
  (a) $\mathcal{S}$ sends $\{(t_{j_0,j_1}, c_{j_0,j_1})\}_{j_1 \in [\sqrt{n}]}$ to C.
  (b) For all $j_1 \in [\sqrt{n}]$, C computes $h_{j_0,j_1} = \mathsf{P}(pk_1, t_{j_0,\pi_1(j_1)})$, and $e_{j_0,j_1} = \mathsf{SKE.Enc}(ek, \mathsf{SKE.Dec}(ek, c_{j_0,\pi_1(j_1)}))$. Then, C sends $\{(h_{j_0,j_1}, e_{j_0,j_1})\}_{j_1 \in [\sqrt{n}]}$ to $\mathcal{S}$.
  (c) Finally, C deletes $\{(t_{j_0,j_1}, c_{j_0,j_1}), (h_{j_0,j_1}, e_{j_0,j_1})\}_{j_0 \in [\sqrt{n}]}$ from its local storage.

Online Phase: To query an index $i \in [n]$, $\mathcal{S}$ and C proceed as follows.

(1) Query. C computes $h = \mathsf{P}(pk_1, \mathsf{P}(pk_0, i))$ and sends h to $\mathcal{S}$.
(2) Response. $\mathcal{S}$ finds h_{j_0,j_1} such that $h_{j_0,j_1} = h$. Then, it sends $e = e_{j_0,j_1}$ to C.
(3) Extract. C decrypts and outputs $z = \mathsf{SKE.Dec}(ek, e)$.

Step 3: Reducing the Bandwidth via a Streaming Algorithm. The basic idea involves bandwidth $\tilde{O}(n)$ during the offline phase. This makes it necessary for the client to have $\tilde{O}(n)$ transient storage (in the offline phase). To solve a similar problem, Piano [50] and Spam [38] leverage a streaming algorithm to reduce the bandwidth to $O(\sqrt{n})$, which significantly reduces the amount of storage required by the client in the offline phase. Their idea is that the server sends $O(\sqrt{n})$ entries each time, and the client processes and then deletes those entries from its storage. However, this idea does not apply to our protocol since we need to permute the entire database. If the client downloads $O(\sqrt{n})$ entries each time and uploads the processed entries, then the server knows that the encoded entries it receives are the permutation of the entries that the client downloaded previously, which leaks information about the permutation. To reduce the bandwidth of Pai, we instead use the folding technique of [31], which represents the database as a 2-dimensional hypercube. Then, the parties permute the rows and columns of the database, respectively. As we will see, this allows us to reduce the offline bandwidth to be $\tilde{O}(\sqrt{n})$. By representing the database as a hypercube with a higher dimension d, we can further reduce the bandwidth to $\tilde{O}(n^{1/d})$. However, the offline communication of our protocol will also increase by a factor of $O(d)$. To make a fair comparison with Piano and Spam, we set $d = 2$ in both the description and implementation of our protocol. In practice, one can obtain the required tradeoff between offline bandwidth and offline communication by using different d.

The above three steps lead to an extremely simple online phase in Pai. Assuming that the client has an index i, the client first computes and sends $h = \mathsf{P}(i)$ to the server. The server finds $j \in [n]$ such that $h = h_j$ and returns $e = e_j$ to the client, where $j = \pi^{-1}(i)$. By decrypting e_j using the SKE key ek, the client obtains the value $v_{\pi(j)} = v_i$.

4.2 Security Analysis for Pai

We show the security of Pai by proving the following theorem.

Theorem 1. *Π_{Pai} is a PIR satisfying correctness and client-privacy.*

Proof. We show the correctness and client-privacy of Π_{Pai}.

Correctness. We need to prove that the client's output is v_i. Assume that $i_0, i_1 \in [\sqrt{n}]$ satisfy that $i = (i_0 - 1)\sqrt{n} + i_1$, then we have

$$\begin{aligned} h =& \mathsf{P}(pk_1, \mathsf{P}(pk_0, i)) = \mathsf{P}(pk_1, \mathsf{P}(pk_0, (i_0 - 1)\sqrt{n} + i_1)) \\ =& \mathsf{P}(pk_1, t_{\pi_0^{-1}(i_0), i_1}) = h_{\pi_0^{-1}(i_0), \pi_1^{-1}(i_1)}. \end{aligned}$$

This means that $e = e_{\pi_0^{-1}(i_0), \pi_1^{-1}(i_1)}$. Following the procedures in the offline phase, we know that for any $j_0, j_1 \in [\sqrt{n}]$, e_{j_0,j_1} is an encryption of $v_{\pi_0(j_0), \pi_1(j_1)}$, which implies that e is an encryption of $v_{i_0,i_1} = v_i$. Note that the output of the client is the decryption of e. Therefore, the output of the client is v_i.

Client-Privacy. Let $\mathsf{View}_{\mathsf{ser}}(\mathsf{DB}, i)$ be the view of the server. We need to construct a PPT simulator Sim taking DB as input such that

$$\mathsf{View}_{\mathsf{ser}}(\mathsf{DB}, i) \approx_c \mathsf{Sim}(\mathsf{DB}).$$

Note that $\mathsf{View}_{\mathsf{ser}}(\mathsf{DB}, i)$ consists of

$$(\mathsf{DB}, \{(t_{j_0,j_1}, c_{j_0,j_1})\}_{j_0,j_1\in[\sqrt{n}]}, \{(h_{j_0,j_1}, e_{j_0,j_1})\}_{j_0,j_1\in[\sqrt{n}]}, h),$$

where $h = \mathsf{P}(pk_1, \mathsf{P}(pk_0, i))$. Sim proceeds as follows.

(1) Sample two sets $\{t^*_{j_0,j_1}\}_{j_0,j_1\in[\sqrt{n}]}, \{h^*_{j_0,j_1}\}_{j_0,j_1\in[\sqrt{n}]}$ of elements from $\{0,1\}^l$, where each set contains n distinct and random elements.
(2) Sample $2n$ ciphertexts $\{c^*_{j_0,j_1}, e^*_{j_0,j_1}\}_{j_0,j_1\in[\sqrt{n}]}$ of zero.
(3) Pick two random elements $j^*_0, j^*_1 \in [\sqrt{n}]$ and sets

$$h^* = h^*_{j^*_0,j^*_1}.$$

(4) Output the simulated view

$$(\mathsf{DB}, \{(t^*_{j_0,j_1}, c^*_{j_0,j_1})\}_{j_0,j_1\in[\sqrt{n}]}, \{(h^*_{j_0,j_1}, e^*_{j_0,j_1})\}_{j_0,j_1\in[\sqrt{n}]}, h^*)$$

Now, we show that the real and simulated views are indistinguishable. First, since P is a PRP over $\{0,1\}^l$, both $\{t_{j_0,j_1}\}_{j_0,j_1\in[\sqrt{n}]}$ and $\{h_{j_0,j_1}\}_{j_0,j_1\in[\sqrt{n}]}$ contain n distinct and random elements in $\{0,1\}^l$. Secondly, for any $j_0, j_1 \in [\sqrt{n}]$, by the CPA security of the underlying SKE scheme, we know that the distribution of both c_{j_0,j_1} and e_{j_0,j_1} is indistinguishable from the distribution of a fresh encryption of zero. Finally, by the correctness of our protocol, we know that $h = h_{\pi_0^{-1}(i_0),\pi_1^{-1}(i_1)}$. Note that π_0, π_1 are random permutations over $[\sqrt{n}]$ unknown to the server, hence h is a random element in $\{h_{j_0,j_1}\}_{j_0,j_1\in[\sqrt{n}]}$. From what has been discussed above, the real and simulated views are indistinguishable. □

We remark that the security of Pai needs the assumption that the client does not make any duplicate queries. The underlying reason is that the client's query is deterministic w.r.t. the client's indices, which allows the server to determine whether two queries correspond to the same index. This would violate the security definition. Fortunately, this assumption can be relaxed without loss of generality by letting the client save the retrieved indices and their answers locally, with additionally $\tilde{O}(\sqrt{n})$ client storage cost.

Complexity Analysis. We analyze the complexity of Π_{Pai} from four aspects: communication, computation, bandwidth, and storage. This analysis encompasses both the offline and online phases.

We first consider the offline phase. During this phase, the parties are required to transmit n plaintexts, $3n$ ciphertexts, and $3n$ strings, each of a length l, where $l \geq \lceil \log n \rceil$. Hence, the total communication cost is $\tilde{O}(n)$. In addition, the client needs to compute P $2n$ times, $2n$ encryptions, and n decryptions. The total computation is also $\tilde{O}(n)$. Moreover, since parties run the offline phase in a streaming way, and each message sent by the parties only contains $\sqrt{n}$ plaintexts or ciphertexts (and $\sqrt{n}$ l-bit long bitstrings), the bandwidth of the protocol is $\tilde{O}(\sqrt{n})$. Finally, it is easy to see that the server-side and client-side storage are $\tilde{O}(n)$ and $\tilde{O}(\sqrt{n})$, respectively.

Next we consider the online phase. For each query in the online phase, the parties transmit an l-bit string h and a ciphertext e. Hence, the communication cost per query is $\tilde{O}(1)$. In addition, the client needs to compute the PRP P one time in addition to executing a single decryption. Since the computational complexity of the permutations is a polynomial of the input length $O(\log n)$, and the computational cost of the decryption is independent of n, the total computational cost is $\tilde{O}(1)$. The bandwidth of the online phase is $\tilde{O}(1)$, for sending h and e. The server-side storage is $\tilde{O}(n)$, and the client-side storage is $\tilde{O}(1)$ for pk_0, pk_1 and ek.

5 KSPIR AND CKSPIR

In this section, we show how to obtain client-preprocessing KSPIR based on Piano, Spam, and our Pai. The construction is non-trivial, and additional privacy problems should be considered. We also provide concrete implementations for KSPIR and compare their efficiency with SOTA KSPIR without preprocessing. The experimental results show that client-preprocessing KSPIR also enjoys extremely low online communication and computation costs, greatly facilitating the real-time query and retrieval needs of the data marketplace.

In addition, we can leverage our idea of Pai to construct a more efficient CKSPIR, allowing the server to know whether it contains the key that the client retrieves. This is normally considered as an "extra leakage" in KSPIR. However, such "extra leakage" is sometimes necessary in the data marketplace. For example, the data sellers can decide how many entries the client is truly retrieved based on that "extra leakage" and ask for data pricing.

5.1 KPIR from PIR

Zhou et al. [50] proposed to use cuckoo hashing table [41] to construct KPIR from Piano. Their construction can be traced back to the idea introduced by Ali et al. [2].

A cuckoo hashing table is specified by v hash functions $\mathsf{H}_1, \ldots, \mathsf{H}_v$ with range $[m]$, where $m = (1+\epsilon)\cdot n$ for some $\epsilon > 0$. The goal of the cuckoo hashing table is to locate n elements into m bins so that each bin contains at most 1 element. The cuckoo hashing table avoids possible collisions by inserting elements using a recursive eviction procedure with the help of these v hash functions: whenever an element is located in an occupied bin, the occupying element is evicted and recursively relocated with a different hash function. By choosing suitable ϵ, we can always find suitable v hash functions with high probability. Like [2], we formalize the procedure of finding suitable v hash functions as $\mathsf{Cuckoo.KeyGen}(\mathsf{EB})$, where EB is a database containing n elements.

The KPIR construction works as follows. Let $\mathsf{EB} = \{(h_i, e_i)\}_{i\in[n]}$ be the key-value database held by the server. First, the server chooses v hash functions by $(\mathsf{H}_1, \ldots, \mathsf{H}_v) \leftarrow \mathsf{Cuckoo.KeyGen}(\mathsf{EB})$. The server then inserts each (h_i, e_i) in the bin $\mathsf{B}_{\mathsf{H}_\tau(h_i)}$ for some $\tau \in [v]$. We remark that whether the insertion succeeds depends on the selected hash functions. This means that the client may be able to learn information about the database based on the selected hash functions that result in a successful insertion. However, this does not violate privacy since the database is not private in KPIR. For empty bins, the server pads with dummy values that are denoted by $\perp$. The server finally sets the PIR database as $\mathsf{DB} = (x_1, \ldots, x_m)$, where each x_j is the element in B_j containing some (h_i, e_i), and sends the descriptions of $(\mathsf{H}_1, \ldots, \mathsf{H}_v)$ to the client.

Whenever the client wants to query the key h, it first computes $i_j = \mathsf{H}_j(h), j \in [v]$. Then, the client runs PIR with the server, where the server takes DB as the database, and the client takes indices $i_1, \ldots, i_v$ as inputs. At the end of PIR, the client obtains $x_{i_1}, \ldots, x_{i_v}$. The client finds x_{i_j} such that the first entry of x_{i_j} is h and outputs the second entry of x_{i_j}. If no such x_{i_j} exists, then the client outputs $\perp$. See Figure 3 for a tiny example with $n = 5$ and $v = 3$.

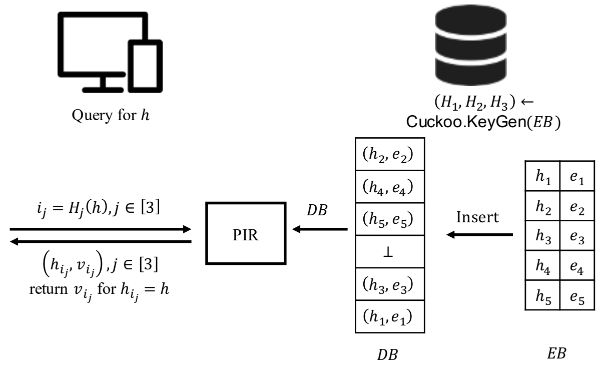
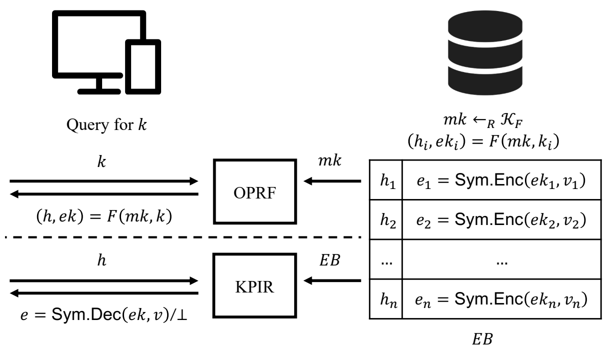


Figure 3: KPIR from PIR.



Query for k
$mk \leftarrow_R \mathcal{K}_F$
$(h_i, ek_i) = F(mk, k_i)$
k
$(h, ek) = F(mk, k)$
OPRF
mk
h
e = Sym.Dec(ek, v)/$\perp$
KPIR
EB
h_1 e_1 = Sym.Enc(ek_1, v_1)
h_2 e_2 = Sym.Enc(ek_2, v_2)
... ...
h_n e_n = Sym.Enc(ek_n, v_n)
EB

Figure 4: KSPIR from KPIR.

5.2 KSPIR from KPIR

Freeman et al. [23] presented a method to construct (K)SPIR from (K)PIR based on OPRF. The main idea is to let the server encrypt each entry in the database by a session key ek_i derived from PRF F with the master key mk and each retrieval key k_i. The server and the client then run (K)PIR, and the client obtains the encrypted entry corresponding to its retrieval key k. To enable the client to decrypt, they invoke OPRF that allows the client to only obtain the corresponding session key ek for the retrieval key k without the server obtaining any information about k.

For the sake of completeness, we review the transformation in detail. See Figure 4 for a tiny example. The security proof of this transformation can be found in [23, Section 4.2].

Let $\{k_i, v_i\}_{i\in[n]}$ be the database in KSPIR. The server first encodes the database as follows.

(1) Let (SKE.Gen, SKE.Enc, SKE.Dec) be an semantically secure SKE scheme.
(2) Let F be a PRF for an OPRF protocol. The server samples a master key mk.
(3) For each $i \in [n]$, the server parses $\mathsf{F}(mk, k_i)$ as (h_i, ek_i), where h_i serves as the key of the encoded database, and ek_i is a SKE key.
(4) For each $i \in [n]$, the server computes $e_i = \mathsf{SKE.Enc}(ek_i, v_i)$.
(5) Let $\mathsf{EB} = \{(h_i, e_i)\}$ be the encoded database.

Now, we can obtain a KSPIR protocol from a KPIR protocol and an OPRF protocol as follows.

(1) To query the key k, the parties run an OPRF, where the server takes mk as input, and the client takes k as input. At the end of OPRF, the client obtains $ek = \mathsf{F}(mk, k)$. The client parses ek as (h, ek).
(2) The parties run a KPIR protocol, where the server takes EB as input and the client takes h as input. Let e be the output of the client.
(3) If $e = \perp$, then the client outputs $\perp$; otherwise, the client runs $v = \mathsf{SKE.Dec}(ek, e)$ to decrypt e and outputs v.

5.3 More Efficient CKSPIR

Since PaiKSPIR is the KSPIR protocol with the best online efficiency, the CKSPIR protocol directly converted from PaiKSPIR by allowing the client to send the matching results to the server is the CKSPIR protocol with the best online efficiency at present. In this section, we present a new CKSPIR protocol that achieves better online efficiency than PaiKSPIR. The full description of PaiCKSPIR is shown in Figure 5.

Our CKSPIR protocol PaiCKSPIR is directly built upon Pai. However, since CKSPIR implements stronger functionality (keyword search) and requires higher privacy (symmetric privacy), PaiCKSPIR has a much more complicated offline phase than Pai. Concretely, the offline phase PaiCKSPIR is mainly different from that of Pai in the following aspects (let $\{(k_i, v_i)\}_{i\in[n]}$ be the database).

Encoding Functions. The two encoding algorithms $\mathsf{E}_{\mathsf{key}}$ and $\mathsf{E}_{\mathsf{value}}$ (which play the roles of $\mathsf{E}_{\mathsf{index}}$ and $\mathsf{E}_{\mathsf{entry}}$ in Pai) are used to encode the keys and values, respectively. Instead of using a PRP, we instantiate $\mathsf{E}_{\mathsf{key}}$ with the function $f_\beta(k) = \mathsf{H}(k)^\beta$, where H is a collision-resistant hash function (CRHF) that maps the keys to elements of a DDH-hard group $\mathbb{G}$ of order p for some large prime p, and $\beta \in \mathbb{Z}_p$ is random key held by the client. In the random oracle model, the function $f_\beta(k)$ is a PRF under the Decisional Diffie–Hellman (DDH) assumption [19]. As in Pai, $\mathsf{E}_{\mathsf{value}}$ is still instantiated with an CPA-secure SKE scheme.

Offline Phase. In Pai, the server directly sends the database to the client. In PaiCKSPIR, we must protect the privacy of the database. Therefore, we let the server send an encryption of the database. Concretely, we let the server sends $(t_i, c_i) = (\mathsf{H}(k_i)^\alpha, v_i \oplus r_i)$ instead of just (k_i, v_i), where $\alpha \in \mathbb{Z}_p$ is random, r_i is a random value, and $\oplus$ is XOR. This prevents the client from obtaining information about (k_i, v_i). Now, we can let the client encrypt the new database $\{(t_i, c_i)\}_{i\in[n]}$ using $\mathsf{E}_{\mathsf{key}}$ and $\mathsf{E}_{\mathsf{value}}$. Concretely, the client chooses a random $\beta \in \mathbb{Z}_p$ and an encryption key ek of a CPA-secure SKE scheme. Moreover, the client chooses a random permutation π over $[n]$. Then, the client encrypts the database as $\{(h_i, e_i)\}_{i\in[n]}$, where $h_i = t^\beta_{\pi(i)}$, $e_i = \mathsf{SKE.Enc}(ek, c_{\pi(i)})$. Finally, the client sends $\{(h_i, e_i)\}_{i\in[n]}$ to the server. Using the same idea as in Pai, we can also reduce the offline bandwidth from $\tilde{O}(n)$ to $\tilde{O}(\sqrt{n})$.

Online Phase. Since the server also participates in encoding the database, PaiCKSPIR will have a different online phase from Pai. Assume that the client has a search key k, the client first computes $h_c = \mathsf{H}(k)^\beta$ and then sends h_c to the server. Then the server computes $h = h_c^\alpha$ and checks whether there exists some h_j such that

$h_j = h$. By the properties of H, we can easily verify the correctness. If the server finds that $h = h_j$, then it sends e_j to the client. By decrypting e_j using the SKE key ek, the client obtains the value $c_{\pi(j)} = v_{\pi(j)} \oplus r_{\pi(j)}$. To let the client obtain the value $v_{\pi(j)}$, we still need to let the client obtain $r_{\pi(j)}$, which seems impossible due to that the server does not know the permutation π. To deal with this, we use OPRF in our protocol. Concretely, instead of choosing r_i at random, we let $r_i = \mathsf{F}(mk, k_i)$, where F is a PRF and mk is a PRF key sampled by the server. Then, the client can obtain the value $r_{\pi(j)} = \mathsf{F}(mk, k_{\pi(j)}) = \mathsf{F}(mk, k)$ by running an OPRF protocol with the server.

Security Proof. We prove the following theorem to state the security of $\Pi_{\mathsf{PaiCKSPIR}}$.

Theorem 2. *$\Pi_{\mathsf{PaiCKSPIR}}$ is a CKSPIR satisfying correctness, server-privacy, and client-privacy.*

Proof. We need to show the correctness, server-privacy, and client-privacy of $\Pi_{\mathsf{PaiCKSPIR}}$.

Correctness. We first consider the case that $k = k'_{i_0,i_1}$ for some $i_0, i_1 \in [\sqrt{n}]$, then $h = h_{\pi_0^{-1}(i_0),\pi_1^{-1}(i_1)}$, and the server will output $b = 1$. Moreover, by the collision-resistance of H, for any $(j_0, j_1) \neq (\pi_0^{-1}(i_0), \pi_1^{-1}(i_1))$, with overwhelming probability it holds that $h \neq h_{j_0,j_1}$. Therefore, with overwhelming probability, the server will return $e_{\pi_0^{-1}(i_0),\pi_1^{-1}(i_1)}$ to the client. By the offline phase, we know that $e_{\pi_0^{-1}(i_0),\pi_1^{-1}(i_1)}$ is an encryption of $m_{i_0,i_1} = v'_{i_0,i_1} \oplus r_{i_0,i_1}$. Therefore, the client will output

$$v = \mathsf{SKE.Dec}(ek, e) \oplus r = v'_{i_0,i_1} \oplus r_{i_0,i_1} \oplus \mathsf{F}(mk, k_{i_0,i_1}) = v'_{i_0,i_1}.$$

If $k \neq k'_{j_0,j_1}$ for any $j_0, j_1 \in [\sqrt{n}]$, then by the collision-resistance of H, the probability that there exists some $i_0, i_1 \in [\sqrt{n}]$ such that $\mathsf{H}(k'_{i_0,i_1})^{\alpha\beta} = \mathsf{H}(k)^{\alpha\beta}$ is negligible (α, β are random). Therefore, with overwhelming probability, the server will output $b = 0$ and the client will output $\perp$.

Server-Privacy. Let $\mathsf{View_{cli}}(\mathsf{DB}, k)$ be the view of the client. We need to construct a PPT simulator Sim taking (k, z) as input such that

$$\mathsf{View_{cli}}(\mathsf{DB}, k) \approx_c \mathsf{Sim}(k, z).$$

Note that $\mathsf{View_{cli}}(\mathsf{DB}, k)$ consists of

$$(k, \{(s_{j_0,j_1}, m_{j_0,j_1})\}_{j_0,j_1 \in [\sqrt{n}]}, \{(t_{j_0,j_1}, c_{j_0,j_1})\}_{j_0,j_1 \in [\sqrt{n}]}, \mathsf{resp}, \mathsf{vc_{oprf}}, z)$$

where resp is the message received from the server in the online phase, and $\mathsf{vc_{oprf}}$ is the view of the client in the OPRF protocol. To simulate this view, Sim performs as follows.

(1) Sample two random permutations π_0^*, π_1^* over the set $[\sqrt{n}]$, two random values $\beta_0^*, \beta_1^* \in \mathbb{Z}_p$ (set $\beta^* = \beta_0^*\beta_1^*$), and one SKE key ek^*.
(2) Choose n random group elements $\{s^*_{j_0,j_1}\}_{j_0,j_1 \in [\sqrt{n}]}$ from $\mathbb{G}$ and n random values $\{m^*_{j_0,j_1}\}_{j_0,j_1 \in [\sqrt{n}]}$ from $\mathcal{V}$.
(3) For all $j_0, j_1 \in [\sqrt{n}]$, compute $t^*_{j_0,j_1} = (s^*_{\pi_0^*(j_0),j_1})^{\beta_0^*}$, $c^*_{j_0,j_1} = \mathsf{SKE.Enc}(ek^*, m_{\pi_0^*(j_0),j_1})$.
(4) If $z \neq \perp$, choose random $i_0^*, i_1^* \in [\sqrt{n}]$ and compute $r^* = m^*_{i_0^*,i_1^*} \oplus z$. Otherwise, choose a random value r^* from $\mathcal{V}$.
(5) If $z \neq \perp$, compute $\mathsf{resp}^* = \mathsf{SKE.Enc}(ek^*, z \oplus r^*)$. Otherwise, compute $\mathsf{resp}^* = \perp$.
(6) Invoke the OPRF simulator on (k, r^*) and let $\mathsf{vc}^*_{\mathsf{oprf}}$ be the output.
(7) Output the simulated view

$$(k, \{(s^*_{j_0,j_1}, m^*_{j_0,j_1})\}_{j_0,j_1 \in [\sqrt{n}]}, \{(t^*_{j_0,j_1}, c^*_{j_0,j_1})\}_{j_0,j_1 \in [\sqrt{n}]}, \mathsf{resp}^*, \mathsf{vc}^*_{\mathsf{oprf}}, z).$$

It remains to show that the simulated view is indistinguishable from the real view. Firstly, note that $f_\alpha(k) = \mathsf{H}(k)^\alpha$ is a PRF in the RO model, hence each s_{j_0,j_1} is indistinguishable from a random value. Moreover, $t_{j_0,j_1} = (s_{\pi_0(j_0),j_1})^{\beta_0}$, where π_0 is a random permutation over $[\sqrt{n}]$ that is sampled by the client. Therefore, we know that $s_{\pi_0(j_0),j_1}$ and $s^*_{\pi_0^*(j_0),j_1}$ are indistinguishable, which implies that t_{j_0,j_1} and $t^*_{j_0,j_1}$ are indistinguishable. On the other hand, by the property of F, we know that every $r_{j_0,j_1} = \mathsf{F}(mk, k'_{j_0,j_1})$ is a pseudorandom value, hence every $m_{j_0,j_1} = v_{j_0,j_1} \oplus \mathsf{F}(mk, k'_{j_0,j_1})$ is indistinguishable from a random value. Moreover, note that $c_{j_0,j_1} = \mathsf{SKE.Enc}(ek, m_{\pi_0(j_0),j_1})$, where ek is a SKE key sampled by the client. Since $m_{\pi_0(j_0),j_1}$ and $m^*_{\pi_0^*(j_0),j_1}$ are indistinguishable, c_{j_0,j_1} and $c^*_{j_0,j_1}$ are indistinguishable. Now, we consider the message $(\mathsf{resp}, \mathsf{vc_{oprf}})$. We first consider the case of $z = \perp$. In this case, we have $\mathsf{resp} = \perp$, and moreover, $k \neq k'_{j_0,j_1}$ for all $j_0, j_1 \in [\sqrt{n}]$, which implies that the OPRF output $r = \mathsf{F}(mk, k)$ has not been used in the offline phase, and we can just use a random value r^* to simulate r. Let $\mathsf{Sim_{oprf}}$ be the OPRF simulator, then we know that $\mathsf{vc_{oprf}} = \mathsf{Sim_{oprf}}(k, r)$ and $\mathsf{vc}^*_{\mathsf{oprf}} = \mathsf{Sim_{oprf}}(k, r^*)$ are indistinguishable. For the case of $z \neq \perp$, by the correctness of our protocol, we know that $k = k'_{i_0,i_1}$ for some $i_0, i_1 \in [\sqrt{n}]$. Note that the database has been permuted by the server, hence i_0, i_1 are in fact two random values. Furthermore, we have $h = h_{\pi_0^{-1}(i_0),\pi_1^{-1}(i_1)}$, which implies that the response message is $e = e_{\pi_0^{-1}(i_0),\pi_1^{-1}(i_1)}$, which is an encryption of m_{i_0,i_1}. Since i_0, i_1 are two random values, we can just simulate e by samples two random values $i_0^*, i_1^* \in [\sqrt{n}]$ and encrypts $m^*_{i_0^*,i_1^*}$ to e^*, which guarantees that e^* and e are indistinguishable. Moreover, since $m^*_{i_0^*,i_1^*}$ and m_{i_0,i_1} are indistinguishable, $r^* = m^*_{i_0^*,i_1^*} \oplus z$ and $r = m_{i_0,i_1} \oplus z$ are indistinguishable. This implies that $\mathsf{vc_{oprf}} = \mathsf{Sim_{oprf}}(k, r)$ and $\mathsf{vc}^*_{\mathsf{oprf}} = \mathsf{Sim_{oprf}}(k, r^*)$ are indistinguishable.

Client-Privacy. Let $\mathsf{View_{ser}}(\mathsf{DB}, k)$ be the view of the server. We need to construct a PPT simulator Sim taking DB as input such that

$$\mathsf{View_{ser}}(\mathsf{DB}, k) \approx_c \mathsf{Sim}(\mathsf{DB}).$$

Note that $\mathsf{View_{ser}}(\mathsf{DB}, k)$ consists of

$$(\mathsf{DB}, \{(t_{j_0,j_1}, c_{j_0,j_1})\}_{j_0,j_1 \in [\sqrt{n}]}, \{(h_{j_0,j_1}, e_{j_0,j_1})\}_{j_0,j_1 \in [\sqrt{n}]}, \mathsf{query}, \mathsf{vc_{oprf}}, b),$$

where $\mathsf{query} = q$ is the query message received from the client in the online phase, and $\mathsf{vc_{oprf}}$ is the view of the server in the OPRF protocol. Since the client offers no inputs in the offline phase, Sim can simulate the messages $(\{(t_{j_0,j_1}, c_{j_0,j_1})\}_{j_0,j_1 \in [\sqrt{n}]}, \{(h_{j_0,j_1}, e_{j_0,j_1})\}_{j_0,j_1 \in [\sqrt{n}]})$ perfectly. Moreover, since the server has no outputs in the OPRF protocol, Sim can simulate $\mathsf{vc_{oprf}}$ by invoking the OPRF simulator on mk and let $\mathsf{vc}^*_{\mathsf{oprf}}$ be the output. The security of the OPRF

Figure 5: The construction of PaiCKSPIR.

Protocol $\Pi_{\mathsf{PaiCKSPIR}}$: CKSPIR with $\tilde{O}(1)$ Client-Side Storage and Online Time

Input: The server $\mathcal{S}$ has a database $\mathsf{DB} = \{(k_j, v_j)\}_{j\in[n]} \in (\mathcal{K}\times\mathcal{V})^n$, and the client C has a search key $k \in \mathcal{K}$.
Output: If there exists some $i \in [n]$ such that $k_i = k$, then $\mathcal{S}$ outputs $b = 1$, and C outputs $z = v_i$. Otherwise, $\mathcal{S}$ outputs $b = 0$, and C outputs $z =\perp$.

Offline Phase: $\mathcal{S}$ first samples a random permutation π over $[n]$ and permutes its database as $\mathsf{DB}' = \{(k'_j, v'_j)\}_{j\in[n]}$, where $(k'_j, v'_j) = (k_{\pi(j)}, v_{\pi(j)})$. Then, $\mathcal{S}$ represents the database as $\mathsf{DB}' = \{(k'_{j_0,j_1}, v'_{j_0,j_1})\}_{j_0,j_1\in[\sqrt{n}]}$. $\mathcal{S}$ and C agree on the following three cryptographic primitives.

- A hash function $\mathsf{H} : \mathcal{K} \to \mathbb{G}$, where $\mathbb{G}$ is a DDH-hard group of order p for some large prime p.
- An CPA-secure SKE scheme $(\mathsf{SKE.Gen}, \mathsf{SKE.Enc}, \mathsf{SKE.Dec})$.
- A PRF $\mathsf{F} : \mathcal{K}_\mathsf{F} \times \mathcal{K} \to \mathcal{V}$ with an OPRF protocol, where $\mathcal{K}_\mathsf{F}$ is the key space of F.

The parties choose the required parameters.

- $\mathcal{S}$ chooses a master key $mk \in \mathcal{K}_\mathsf{F}$ and a random value $\alpha \in \mathbb{Z}_p$.
- C chooses two random permutations π_0, π_1 over the set $[\sqrt{n}]$, two random values $\beta_0, \beta_1 \in \mathbb{Z}_p$ (set $\beta = \beta_0\beta_1$), and one SKE key ek.

Now $\mathcal{S}$ and C interact to encode the database in a streaming way.

(1) Row-Permuting. For each $j_1 \in [\sqrt{n}]$, $\mathcal{S}$ and C perform the following steps.
  (a) For all $j_0 \in [\sqrt{n}]$, $\mathcal{S}$ computes $s_{j_0,j_1} = \mathsf{H}(k'_{j_0,j_1})^\alpha$.
  (b) For all $j_0 \in [\sqrt{n}]$, $\mathcal{S}$ first computes $r_{j_0,j_1} = \mathsf{F}(mk, k'_{j_0,j_1})$ and $m_{j_0,j_1} = v'_{j_0,j_1} \oplus r_{j_0,j_1}$. Then, $\mathcal{S}$ sends $\{(s_{j_0,j_1}, m_{j_0,j_1})\}_{j_0\in[\sqrt{n}]}$ to C.
  (c) For all $j_0 \in [\sqrt{n}]$, C first computes $t_{j_0,j_1} = (s_{\pi_0(j_0),j_1})^{\beta_0}$ and $c_{j_0,j_1} = \mathsf{SKE.Enc}(ek, m_{\pi_0(j_0),j_1})$. Then, C sends $\{(t_{j_0,j_1}, c_{j_0,j_1})\}_{j_0\in[\sqrt{n}]}$ to $\mathcal{S}$.
  (d) C deletes $\{(s_{j_0,j_1}, m_{j_0,j_1}), (t_{j_0,j_1}, c_{j_0,j_1})\}_{j_0\in[\sqrt{n}]}$ from its local storage.

(2) Column-Permuting. For each $j_0 \in [\sqrt{n}]$, $\mathcal{S}$ and C perform the following steps.
  (a) $\mathcal{S}$ sends $\{(t_{j_0,j_1}, c_{j_0,j_1})\}_{j_1\in[\sqrt{n}]}$ to C.
  (b) For all $j_1 \in [\sqrt{n}]$, C computes $h_{j_0,j_1} = t^{\beta_1}_{j_0,\pi_1(j_1)}$ and decrypts $x_{j_0,j_1} = \mathsf{SKE.Dec}(ek, c_{j_0,j_1})$.
  (c) For all $j_1 \in [\sqrt{n}]$, C computes $e_{j_0,j_1} = \mathsf{SKE.Enc}(ek, x_{j_0,\pi_1(j_1)})$. Then, C sends $\{(h_{j_0,j_1}, e_{j_0,j_1})\}_{j_1\in[\sqrt{n}]}$ to $\mathcal{S}$.
  (d) C deletes $\{(t_{j_0,j_1}, c_{j_0,j_1}), (h_{j_0,j_1}, e_{j_0,j_1})\}_{j_1\in[\sqrt{n}]}$ from its local storage.

Online Phase: To query the key k, the parties proceed as follows (and they run the OPRF protocol to let C obtain $r = \mathsf{F}(mk, k)$.

(1) Query. C first computes $q = \mathsf{H}(k)^\beta$ and sends q to $\mathcal{S}$.
(2) Response. $\mathcal{S}$ computes $h = q^\alpha$ and checks whether there exists some h_{j_0,j_1} such that $h_{j_0,j_1} = h$. If the answer is yes, $\mathcal{S}$ outputs $b = 1$ and sends $e = e_{j_0,j_1}$ to C. Otherwise, $\mathcal{S}$ outputs $b = 0$ and sends $\perp$ to C.
(3) Extract. If receiving $\perp$ from $\mathcal{S}$, then C outputs $z =\perp$. Otherwise, C computes $v = \mathsf{SKE.Dec}(ek, e) \oplus r$ and outputs $z = v$.

protocol guarantees that $\mathsf{vc}^*_{\mathsf{oprf}}$ and $\mathsf{vc}_{\mathsf{oprf}}$ are indistinguishable. To simulate the message query, Sim performs as follows.

(1) If $b = 0$, sample a random value $q \in \mathcal{V}$.
(2) Otherwise, choose two random values $i^*_0, i^*_1 \in [\sqrt{n}]$ and compute $q^* = h^{1/\alpha}_{i^*_0,i^*_1}$.

We show that q and q^* are indistinguishable. First, consider the case of $b = 1$, which implies that $k = k'_{i_0,i_1}$ for some $i_0, i_1 \in [\sqrt{n}]$. In this case, the query message in the real view is $q = \mathsf{H}(k)^\beta = \mathsf{H}(k'_{i_0,i_1})^\beta = h^{1/\alpha}_{\pi_0^{-1}(i_0),\pi_1^{-1}(i_1)}$, and the simulated query message is $q^* = h^{1/\alpha}_{i^*_0,i^*_1}$ for random $i^*_0, i^*_1 \in [\sqrt{n}]$. Note that π_0 and π_1 are two random permutations over $[\sqrt{n}]$ that are unknown to the server. Therefore, $\pi_0^{-1}(i_0), \pi_1^{-1}(i_1)$ are two random values of $[\sqrt{n}]$ and q and q^* are indistinguishable.

Now, consider the case of $b = 0$, which implies that $k \neq k_{j_0,j_1}$ for any $j_0, j_1 \in [\sqrt{n}]$. In this case, the real query message q is $\mathsf{H}(k)^\beta$, where β is sampled by the client. And the simulated query message q^* is a random element of $\mathcal{V}$. Since β is unknown to the server, $\mathsf{H}(k)^\beta$ is indistinguishable from a random element. Therefore, q and q^* are indistinguishable.

□

Complexity Analysis. Let us first consider the offline phase. In the offline phase, the parties need to send $4n$ group elements, n plaintexts, and $3n$ ciphertexts. Hence, the communication cost is $\tilde{O}(n)$. In addition, the server needs to compute H and F n times, n Xors and exponentiations. Moreover, the client needs to compute

$2n$ exponentiations, $2n$ encryptions, and n decryptions. Overall, the offline computational cost is $\tilde{O}(n)$. Furthermore, since each message sent by the parties contains $\sqrt{n}$ group elements and $\sqrt{n}$ plaintexts or ciphertexts, the bandwidth of the protocol is $\tilde{O}(\sqrt{n})$. Finally, it is easy to see that the server-side and client-side storage are $\tilde{O}(n)$ and $\tilde{O}(\sqrt{n})$, respectively.

Now, we consider the online phase. Note that the OPRF protocol (described in Section 2.4) requires the parties to send two group elements and compute the hash function one time and three exponentiations, hence both the communication and computational costs are $\tilde{O}(1)$. In addition to executing the OPRF protocol, the parties need to send a group element and a ciphertext (as well as a bit b), which requires $\tilde{O}(1)$ communication. Moreover, the parties need to compute the hash function H one time and two exponentiations, which consumes $\tilde{O}(1)$ amount of computation. Finally, it is easy to see that the bandwidth of the online phase is $\tilde{O}(1)$, and the server-side and client-side storage are $\tilde{O}(n)$ and $\tilde{O}(1)$, respectively.

6 IMPLEMENTATIONS AND EVALUATIONS

6.1 Implementation Details

We implement all our schemes and compare them with several baselines. To eliminate performance gaps caused by programming languages, we carefully study existing open-source codes, and re-implement baselines mainly using Java. Here, we summarize each baseline with implementation details.

Piano by Zhou et al. [50]. The authors provide a full implementation in Go and is open-sourced at https://github.com/pianopir/Piano-PIR. Our re-implementation exactly follows all parameter settings shown in their implementation. In their full implementation, the client generates online queries with *random* indices. To support our KSPIR construction, our re-implementation additionally allows the client to query *specific* indices.

Spam by Mughees et al. [38]. The authors implement their scheme in C++. They set the computational security parameter λ to 80 rather than 128 used in other baselines. To date, we have not found the open-source repository. We fully implement Spam with the parameter setting shown in their work with $\lambda = 128$. We also try our best to implement some optimizations shown in their work, including two subset encoding (Section 3.2), pair backup "hints" generation (Section 3.4), and improved median finding (Section 4.1).

LPSI by Chen et al. [15]. We choose Labeled Private Set Intersection (LPSI) as the baseline for KSPIR. LPSI is a specific type of PSI that allows the client to learn the labels of the elements in the intersection. Chen et al. [10] pointed out that LPSI is equivalent to KSPIR in the batching setting, in the sense that the client can ask multiple keys (elements) for entries (labels) in one query. The SOTA LPSI was proposed by Cong et al. [15], with the open-sourced library named APSI that is available at https://github.com/microsoft/APSI. Their implementation invokes Microsoft SEAL library[2] for Fully Homomorphic Encryption (FHE) and FourQ[3] for Elliptic Curve Cryptography (ECC).

We implement Piano, Spam, and Pai under a unified API. Then, we implement the transformation from PIR to KSPIR by invoking these schemes in a black-box manner with a cuckoo hash. Following the estimates in [21], we choose the cuckoo hash parameters as containing 3 hashes and the number of bins $b = 1.5n$ such that inserting n elements in the cuckoo hash fails with probability at most $2^{-\kappa}$. In this way, we obtain the corresponding KSPIR, namely, PianoKSPIR, SpamKSPIR, and PaiKSPIR. We finally implement our PaiCKSPIR with FourQ ECC as the DDH-hard group.

In our implementation, we set the statistical security parameter κ to 40 and the computational security parameter λ to 128 for all schemes. We utilize AES-NI hardware instruction that is inherently supported by modern Java Virtual Machines for fast PRF and PRP evaluations. The CPA-secure SKE used in Pai and PaiCKSPIR is instantiated as AES with OFB encryption mode. The semantic-secure SKE used in PianoKSPIR, SpamKSPIR, and PaiKSPIR is instantiated as AES with CTR encryption mode.

6.2 Experimental Setup

We evaluate all schemes on a single Intel Core i9-9900K with 3.6GHz and 128GB RAM. Our platform runs Ubuntu 20.04.6 LTS, with Microsoft SEAL 4.1.1, FourQ v3.1, and Java 17.0.1. All query costs are computed as the average over 1000 queries, except the somewhat inefficient APSI, which we average the cost over 100 queries. All experiments are run on a single machine with the 10Gbps bandwidth and 0.05ms RTT latency simulated by the Linux tc command. All computations are performed on 8 threads.

We analyze the performances for databases with $n = 2^{20}, 2^{22}, 2^{24}$ entries. The entry sizes are 64, 128, 256 bytes. Since our experiments consider single-query setting rather than batch-query setting, for APSI, we choose the single-query parameter 1M-1-32[4] for $n = 2^{20}$ entries, and 16M-1-32[5] for $n = 2^{22}$ and $n = 2^{24}$ entries.

6.3 Evaluation Results

We analyze the experimental results for KSPIR and CKSPIR, respectively. The metrics include the one-time pre-processing cost in the offline phase and the costs of the queries in the online phase. We further obtain the online client-side storage cost by using the JOL (Java Object Layout) library[6] to measure the deep sizes (i.e., the size of an object including the size of all referred objects, in addition to the size of the object itself) of Objects packaging the client. Table 1 shows the detailed experiment results for APSI, PianoKSPIR, SpamKSPIR, PaiKSPIR, and PaiCKSPIR.

Evaluations for KSPIR. The online complexity of APSI is much higher than that of the other three protocols, at least three orders of magnitude more expensive. This shows that introducing preprocessing in KSPIR extremely decreases the online costs. The advantage of APSI is that the communication in the offline phase is very low. However, due to the high offline computation complexity of APSI, its overall offline time is similar to that of the other three protocols. We argue that the application scenario of APSI is different from that of the other three protocols. When low offline communication complexity is required, APSI will be a better choice than others.

SpamKSPIR achieves better communication and computation than PianoKSPIR. The online communication cost is 36.1 - 122.4×

[2]https://github.com/microsoft/SEAL

[3]https://github.com/microsoft/FourQlib

[4]https://github.com/microsoft/APSI/blob/main/parameters/1M-1-32.json

[5]https://github.com/microsoft/APSI/blob/main/parameters/16M-1-32.json

[6]http://hg.openjdk.java.net/code-tools/jol

Table 1: Performance of LPSI, PianoKSPIR, SpamKPSIR, PaiKSPIR, and PaiCKSPIR on $n = 2^{20}, 2^{22}, 2^{24}$ database sizes and 64, 128, 256 bytes entry sizes. "Comm." stands for communication cost. Offline costs are the whole preprocessing. Online costs are amortized over 1000 queries except APSI which are amortized over 100 queries. The best results for KSPIR are marked as shallow green . The results are marked as green if PaiCKSPIR is even better than PaiKSPIR. Notice that, this work does not focus on offline costs (all methods require comparable time for pre-processing), thus the results are not highlighted.

		$n = 2^{20}$			$n = 2^{22}$			$n = 2^{24}$		
		64 bytes	128 bytes	256 bytes	64 bytes	128 bytes	256 bytes	64 bytes	128 bytes	256 bytes
Offline Comm. (MB)	APSI	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
	PianoKSPIR	120.069	216.124	408.235	480.085	864.153	1632.289	1920.341	3456.613	6529.158
	SpamKSPIR	120.069	216.124	408.235	480.085	864.153	1632.289	1920.341	3456.613	6529.158
	PaiKSPIR	624.359	1008.581	1777.023	2496.443	4032.715	7105.260	9985.771	16130.862	28421.042
	PaiCKSPIR	432.0	688.0	1200.0	1728.0	2752.0	4800.0	6912.0	11008.0	19200.0
Offline Time (s)	APSI	23.697	35.538	61.234	82.641	128.631	230.835	338.196	542.071	938.441
	PianoKSPIR	17.388	15.517	18.294	70.225	63.999	69.712	288.962	255.728	296.759
	SpamKSPIR	20.544	20.918	24.371	82.174	82.754	104.972	329.514	331.764	417.963
	PaiKSPIR	16.218	17.331	20.157	60.437	67.729	78.297	249.635	275.815	308.757
	PaiCKSPIR	33.311	33.525	35.030	119.743	123.880	128.781	470.224	487.166	516.738
Online Comm. (KB)	APSI	5769.693	9401.396	17270.112	13517.358	22959.823	43418.503	47413.396	82277.844	157817.458
	PianoKSPIR	301.311	536.436	1006.686	602.564	1072.814	2013.314	1205.313	2146.000	4027.375
	SpamKSPIR	8.351	8.726	9.476	16.158	16.533	17.283	31.773	32.148	32.898
	PaiKSPIR	0.391	0.578	0.953	0.391	0.578	0.953	0.391	0.578	0.953
	PaiCKSPIR	0.172	0.234	0.359	0.172	0.234	0.359	0.172	0.234	0.359
Online Time (ms)	APSI	3067.12	4072.03	6249.78	5464.79	8034.68	13886.16	15149.30	25075.80	47133.38
	PianoKSPIR	4.398	5.208	6.754	7.121	9.032	11.323	14.363	17.201	21.827
	SpamKSPIR	3.178	3.235	3.378	4.065	4.497	4.793	7.702	8.021	8.940
	PaiKSPIR	1.391	1.436	1.444	1.045	1.031	1.071	1.055	0.989	1.008
	PaiCKSPIR	0.883	0.906	0.878	0.671	0.673	0.670	0.634	0.649	0.617
Online Client-Side Storage (MB)	APSI	5.260	5.256	5.436	5.274	5.271	5.272	5.271	5.266	5.437
	PianoKSPIR	21.888	28.172	40.740	40.272	53.444	79.789	81.095	109.272	165.626
	SpamKSPIR	21.114	27.396	39.963	37.123	49.682	74.801	69.485	94.598	144.824
	PaiKSPIR	4.931	5.161	4.950	4.760	4.990	4.778	4.758	4.988	4.776
	PaiCKSPIR	5.139	4.921	5.156	4.969	4.749	4.985	4.969	4.753	4.985

better, and the online computation cost is 1.4 - 2.4× better. PaiKSPIR further achieves more efficient online phase than SpamKSPIR. Concretely, PaiKSPIR reduces the online communication cost by 9.9 - 81.3×, and the online time by 2.3 - 8.9×. In addition, PaiKSPIR introduces extremely low storage costs. Under the parameters we tested, the concrete storage cost of PaiKSPIR is only 2.9% – 23.4% than that of PianoKSPIR and SpamKSPIR.

The offline time of PaiKSPIR is similar to that of PianoKSPIR and lower than that of SpamKSPIR. Also, PaiKSPIR has the disadvantage of high offline communication, which is about 4 - 6× larger than that of PianoKSPIR and SpamKSPIR. Given that we have an extremely fast online phase, a relatively high but reasonable offline communication is acceptable.

Evaluations for CKSPIR. For online communication, PaiCKSPIR is 2.3 - 2.7× better than PaiKSPIR, and for online time, PaiCKSPIR is 1.5 - 1.7× better than PaiKSPIR. The main reason is that in the general transformation from PIR to KSPIR (described in Section 5.1), a single keyword query requires the client to make $v = 3$ index queries. For the offline phase, since PaiKSPIR uses our PIR protocol with a database containing $b = 1.5n$ entries, PaiCKSPIR has a lower offline communication, which is 1.4 - 1.5× better than PaiKSPIR. Moreover, since that PaiCKSPIR uses much more public-key operations (i.e., exponentiations), the offline time of PaiCKSPIR is 1.6 - 2.1× higher than PaiKSPIR.

7 CONCLUSION

We propose a novel framework Pai under the client-preprocessing PIR model for the data marketplace. Pai boasts $O(1)$ online time and client-side storage, with the optimality retained up to a polylogarithmic factor. As a comparison, the online time and online client-side storage of the state-of-the-art PIR protocols Piano and Spam under the client-preprocessing model are both $\tilde{O}(\sqrt{n})$.

We then extended PIR to KSPIR and CKSPIR, meeting the fundamental key-value data retrieval requirement for the data marketplace. Our KSPIRs are obtained by applying the ideas from [23] and [2] to Piano, Spam, and Pai. We implement all the KSPIR protocols, and our experimental results show that PaiKSPIR achieves a better efficiency than PianoKSPIR and SpamKSPIR. We also introduce CKSPIR, which is mainly used when the server intends to charge the client for a successful query. The experimental results show that our PaiCKSPIR protocol achieves lower online time and similar client-side storage compared with PaiKSPIR.

REFERENCES

[1] Ishtiyaque Ahmad, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. 2022. Pantheon: Private Retrieval from Public Key-Value Store. *Proc. VLDB Endow.* (2022), 643–656.

[2] Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. 2021. Communication-Computation Trade-offs in PIR. In *USENIX Security 2021*. 1811–1828.

[3] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. 2018. PIR with Compressed Queries and Amortized Query Processing. In *SP 2018*. 962–979.

[4] Michael Armbrust, Tathagata Das, Sameer Paranjpye, Reynold Xin, Shixiong Zhu, Ali Ghodsi, Burak Yavuz, Mukul Murthy, Joseph Torres, Liwen Sun, Peter A. Boncz, Mostafa Mokhtar, Herman Van Hovell, Adrian Ionescu, Alicja Luszczak, Michal Switakowski, Takuya Ueshin, Xiao Li, Michal Szafranski, Pieter Senster, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proc. VLDB Endow.* 13, 12 (2020), 3411–3424.

[5] Abolfazl Asudeh and Fatemeh Nargesian. 2022. Towards Distribution-aware Query Answering in Data Markets. *Proc. VLDB Endow.* 15, 11 (2022), 3137–3144.

[6] Amos Beimel, Yuval Ishai, and Tal Malkin. 2000. Reducing the Servers Computation in Private Information Retrieval: PIR with Preprocessing. In *CRYPTO 2000*. 55–73.

[7] Dan Boneh, Kevin Lewi, and David J. Wu. 2017. Constraining Pseudorandom Functions Privately. In *PKC 2017*. 494–524.

[8] Christian Cachin, Silvio Micali, and Markus Stadler. 1999. Computationally Private Information Retrieval with Polylogarithmic Communication. In *EUROCRYPT 1999*. 402–414.

[9] Yan-Cheng Chang. 2004. Single Database Private Information Retrieval with Logarithmic Communication. In *ACISP 2004*. 50–61.

[10] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. 2018. Labeled PSI from Fully Homomorphic Encryption with Malicious Security. In *CCS 2018*. 1223–1237.

[11] Benny Chor and Niv Gilboa. 1997. Computationally Private Information Retrieval (Extended Abstract). In *STOC 1997*. 304–313.

[12] Benny Chor, Niv Gilboa, and Moni Naor. 1998. Private Information Retrieval by Keywords. *IACR Cryptol. ePrint Arch.* (1998), 3.

[13] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. 1995. Private Information Retrieval. In *FOCS 1995*.

[14] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. 1998. Private Information Retrieval. *J. ACM* (1998).

[15] Kelong Cong, Radames Cruz Moreno, Mariana Botelho da Gama, Wei Dai, Ilia Iliashenko, Kim Laine, and Michael Rosenberg. 2021. Labeled PSI from Homomorphic Encryption with Reduced Computation and Communication. In *CCS 2021*. 1135–1150.

[16] Henry Corrigan-Gibbs, Alexandra Henzinger, and Dmitry Kogan. 2022. Single-Server Private Information Retrieval with Sublinear Amortized Time. In *EUROCRYPT 2022*, Orr Dunkelman and Stefan Dziembowski (Eds.). 3–33.

[17] Henry Corrigan-Gibbs and Dmitry Kogan. 2020. Private Information Retrieval with Sublinear Online Time. In *EUROCRYPT 2020*. 44–75.

[18] Giovanni Di Crescenzo, Yuval Ishai, and Rafail Ostrovsky. 1998. Universal Service-Providers for Database Private Information Retrieval (Extended Abstract). In *PODC 1998*. 91–100.

[19] Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. 2012. Fast and Private Computation of Cardinality of Set Intersection and Union. In *CANS 2012*. 218–231.

[20] Alex Davidson, Gonçalo Pestana, and Sofía Celi. 2023. FrodoPIR: Simple, Scalable, Single-Server Private Information Retrieval. *PoPETs 2023* (2023), 365–383.

[21] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. 2018. PIR-PSI: Scaling Private Contact Discovery. *PoPETs 2018* (2018), 159–178.

[22] Whitfield Diffie and Martin E. Hellman. 1976. New directions in cryptography. *IEEE Trans. Inf. Theory* 22, 6 (1976), 644–654. https://doi.org/10.1109/TIT.1976.1055638

[23] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. 2005. Keyword Search and Oblivious Pseudorandom Functions. In *TCC 2005*. 303–324.

[24] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *STOC 2009*.

[25] Craig Gentry and Shai Halevi. 2019. Compressible FHE with Applications to PIR. In *TCC 2019*. 438–464.

[26] Craig Gentry and Zulfikar Ramzan. 2005. Single-Database Private Information Retrieval with Constant Communication Rate. In *ICALP 2005*. 803–815.

[27] Yael Gertner, Yuval Ishai, Eyal Kushilevitz, and Tal Malkin. 1998. Protecting Data Privacy in Private Information Retrieval Schemes. In *STOC 1998*.

[28] Alexandra Henzinger, Matthew M. Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. 2023. One Server for the Price of Two: Simple and Fast Single-Server Private Information Retrieval. In *USENIX Security 2023*.

[29] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. 2004. Batch codes and their applications. In *STOC 2004*. 262–271.

[30] Dmitry Kogan and Henry Corrigan-Gibbs. 2021. Private Blocklist Lookups with Checklist. In *USENIX Security 2021*. 875–892.

[31] Eyal Kushilevitz and Rafail Ostrovsky. 1997. Replication is NOT Needed: SINGLE Database, Computationally-Private Information Retrieval. In *FOCS 1997*. 364–373.

[32] Arthur Lazzaretti and Charalampos Papamanthou. 2022. Single Server PIR with Sublinear Amortized Time and Polylogarithmic Bandwidth. *IACR Cryptol. ePrint Arch.* (2022), 830.

[33] Arthur Lazzaretti and Charalampos Papamanthou. 2023. TreePIR: Sublinear-Time and Polylog-Bandwidth Private Information Retrieval from DDH. In *CRYPTO 2023*. 284–314.

[34] Helger Lipmaa. 2009. First CPIR Protocol with Data-Dependent Computation. In *ICISC 2009*. 193–210.

[35] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. 2016. XPIR : Private Information Retrieval for Everyone. *PoPETs 2016* (2016), 155–174.

[36] Samir Jordan Menon and David J. Wu. 2022. SPIRAL: Fast, High-Rate Single-Server PIR via FHE Composition. In *SP 2022*. 930–947.

[37] Muhammad Haris Mughees, Hao Chen, and Ling Ren. 2021. OnionPIR: Response Efficient Single-Server PIR. In *CCS 2021*. 2292–2306.

[38] Muhammad Haris Mughees, Sun I, and Ling Ren. 2023. Simple and Practical Amortized Sublinear Private Information Retrieval. Cryptology ePrint Archive, Paper 2023/1072. (2023).

[39] Muhammad Haris Mughees and Ling Ren. 2023. Vectorized Batch Private Information Retrieval. In *SP 2023*. 437–452.

[40] Rafail Ostrovsky and William E. Skeith III. 2007. A Survey of Single-Database Private Information Retrieval: Techniques and Applications. In *PKC 2007*. 393–411.

[41] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *J. Algorithms* (2004), 122–144.

[42] Pascal Paillier. 1999. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *EUROCRYPT 1999 (Lecture Notes in Computer Science)*, Vol. 1592. Springer, 223–238.

[43] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. 2018. Private Stateful Information Retrieval. In *CCS 2018*. 1002–1019.

[44] Rakesh Pimplikar and Sunita Sarawagi. 2012. Answering Table Queries on the Web using Column Keywords. *Proc. VLDB Endow.* 5, 10 (2012), 908–919.

[45] Elaine Shi, Waqar Aqeel, Balakrishnan Chandrasekaran, and Bruce M. Maggs. 2021. Puncturable Pseudorandom Sets and Private Information Retrieval with Near-Optimal Online Bandwidth and Time. In *CRYPTO 2021*. 641–669.

[46] Radu Sion and Bogdan Carbunar. 2007. On the Practicality of Private Information Retrieval. In *NDSS 2007*.

[47] Siyuan Xia, Zhiru Zhu, Chris Zhu, Jinjin Zhao, Kyle Chard, Aaron J. Elmore, Ian T. Foster, Michael J. Franklin, Sanjay Krishnan, and Raul Castro Fernandez. 2022. Data Station: Delegated, Trustworthy, and Auditable Computation to Enable Data-Sharing Consortia with a Data Escrow. *Proc. VLDB Endow.* 15, 11 (2022), 3172–3185.

[48] Matei Zaharia, Ali Ghodsi, Reynold Xin, and Michael Armbrust. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *CIDR*.

[49] Mingxun Zhou, Wei-Kai Lin, Yiannis Tselekounis, and Elaine Shi. 2023. Optimal Single-Server Private Information Retrieval. In *EUROCRYPT 2023*. 395–425.

[50] Mingxun Zhou, Andrew Park, Elaine Shi, and Wenting Zheng. 2024. Piano: Extremely Simple, Single-Server PIR with Sublinear Server Computation. *To appear in SP 2024* (2024).