Private information retrieval for Open Food Facts

Alexandru Verhovetchi*
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
(Dated: June 29, 2025)

ABSTRACT

Private Information Retrieval (PIR) protocols allow a user to fetch data from a server-hosted database without disclosing which specific item was retrieved. In single-server PIR schemes, privacy is maintained through computational hardness assumptions, with recent works optimizing for diverse use cases while balancing communication and computational efficiency. Although early PIR schemes were prohibitively inefficient, recent work demonstrates that it is possible to make PIR fast enough for practical use. While theoretical advances show notable efficiency improvements, thorough testing of modern methods on large, real-world datasets that are actively used remains crucial to confirm their practical applicability. This paper assesses the efficiency of modern PIR systems, namely SimplePIR, using a large real-world dataset widely adopted in applications, Open Food Facts.

Keywords: Private Information Retrieval (PIR), Privacy, Information Retrieval, Single-Server PIR, SimplePIR.

I. INTRODUCTION

Private Information Retrieval (PIR) protocols enable users to query public databases while keeping their accessed data confidential, a critical feature for privacy-sensitive domains such as healthcare, finance, and open-data platforms [3, 11]. Despite decades of theoretical research, PIR's adoption in real-world applications remains limited due to significant performance challenges, particularly in single-server settings where cryptographic overhead can be prohibitive. In multi-server PIR, strong information-theoretic guarantees can be achieved assuming non-colluding servers [7], but this assumption rarely holds in practice. Single-server PIR was long considered impractical until the introduction of computational PIR, which relies on cryptographic hardness assumptions.

Modern single-server PIR systems (e.g., SimplePIR, FrodoPIR [4, 8]) claim to address these challenges through different algorithmic optimizations and cryptography solutions. These protocols make use of encryption techniques to allow servers to compute over encrypted queries, preserving privacy without revealing the user's intent. SimplePIR uses fast matrix-vector multiplication techniques to minimize latency and communication costs. Other protocols, such as those proposed by Angel et al. [1] and Patel et al. [16], further improve efficiency through compressed or sparse queries, and recent work by Lin et al. [12] introduces preprocessing strategies to reduce online latency. These improvements significantly lower the barrier to practical PIR deployment, but their performance in real-world environments has not been extensively validated.

This work bridges the gap between theory and practice by applying a state-of-the-art PIR protocol to Open Food Facts, the database powering popular nutrition apps like Yuka that are used by millions daily. With approximately 3 million globally sourced products, this real-world dataset enables the first evaluation of PIR systems for food information applications, a privacy-sensitive domain where users currently must reveal their product queries. By quantitatively comparing latency, throughput, and bandwidth costs to conventional retrieval methods (i.e., SQL/HTTP queries), this paper numerically quantifies the practical overhead of adding privacy protection to such applications. The results may offer insights into addressing uncertainties surrounding the practical implementation of a PIR system, while maintaining usability.

II. PROBLEM STATEMENT

Although numerous studies have examined PIR efficiency [3, 4, 8, 17], most rely on synthetic or experimental datasets rather than real-world data with varying record sizes and counts. Recent systems like FrodoPIR [4] and ThorPIR [6] demonstrate improved practicality, but primarily benchmark using uniform record sizes in controlled settings. Even studies modeling realistic applications (e.g., news articles [1] or sparse databases [16]) avoid production datasets. Consequently, it is unclear whether existing PIR systems like SimplePIR are sufficiently optimized to replace conventional non-private retrieval methods in practical applications, such as large food product databases like Open Food Facts.

* a.verhovetchi@student.utwente.nl

A. Research Question

The problem statement leads to the following core research question:

How do modern single-server PIR systems, SimplePIR, perform on real-world datasets like Open Food Facts in terms of latency, throughput, and bandwidth costs?

This question can be further broken down into subquestions:

1. How does efficiency (or associated costs) scale with varying database and record sizes (linear vs. exponential growth)?
2. What impact do different network conditions (high latency, low bandwidth) have on PIR performance?
3. How do the computational and bandwidth costs of private retrieval (using SimplePIR) compare to non-private methods when querying the Open Food Facts database under identical conditions?

Bonus How does server-side PIR latency impact perceived user experience compared to traditional retrieval when querying Open Food Facts?

III. BACKGROUND

Private Information Retrieval (PIR) refers to protocols that enable a client to retrieve a record from a database without revealing which record is being accessed. Early PIR protocols were based on *information-theoretic* guarantees and required multiple non-colluding servers. In this setting, even if some subset of servers colludes, they cannot learn any information about the query [7]. However, this model is impractical in many real-world settings due to the significant overhead of maintaining multiple synchronized and non-colluding datasets.

Single-server PIR was initially considered infeasible, with early approaches such as downloading the entire database being the only way to ensure privacy [3]. A breakthrough came with the work of Kushilevitz and Ostrovsky [11], who introduced *computational PIR* based on standard cryptographic hardness assumptions. In their model, the user encrypts a query index and sends it to the server, which then computes a response over the encrypted input. Upon decryption, the client obtains only the desired database record, while the server gains no knowledge about which record was requested. This approach introduced the use of *homomorphic encryption*, a type of encryption that allows specific algebraic operations to be carried out on ciphertexts such that, when decrypted, the result matches what would have been obtained if operations had been performed on the plaintexts [14]. This property enables the server to operate directly on encrypted queries without learning their contents.

Subsequent research focused on improving the efficiency and practicality of single-server PIR. A notable direction involves the use of *lattice-based cryptography*, particularly schemes based on the *Learning With Errors (LWE)* problem. LWE is a computational hardness assumption involving the problem of solving noisy linear equations over high-dimensional lattices. It is widely believed to be secure even against quantum adversaries and forms the foundation for many modern cryptographic primitives [14]. In the context of PIR, LWE-based constructions allow homomorphic operations that support efficient and scalable protocols.

SimplePIR [8] exemplifies a practical, LWE-based PIR scheme. It achieves sublinear communication complexity and low latency using an efficient matrix-vector multiplication technique, where the client encodes the query as a sparse vector and the server computes an encrypted dot product against the database. The result, once decrypted by the client, reveals the requested item without leaking the query index. The protocol is designed to be computationally efficient on both client and server sides and is suitable for deployment on commodity hardware. In their evaluation, SimplePIR achieved query latencies in the low milliseconds for databases with millions of records, significantly outperforming prior single-server PIR implementations in both speed and communication costs.

The same work also introduces DoublePIR [8], an extension of SimplePIR that applies the protocol recursively to reduce server computation, offering a practical trade-off for scenarios where server efficiency is a bottleneck. DoublePIR improves efficiency by running a second layer of SimplePIR to retrieve just the small portions of data needed (a specific row from the hint matrix and one element from the answer vector), allowing the client to recover the desired database item without downloading the full hint. This significantly reduces communication while preserving privacy.

Similarly, FrodoPIR [4] is built on standard LWE-based encryption and emphasizes simplicity and modularity in its design. Unlike schemes that rely on fully homomorphic encryption (FHE), FrodoPIR avoids complex cryptography in favor of well-understood, conservative constructions that are easier to implement, verify, and maintain. Its architecture separates query generation, encryption, and response evaluation into cleanly defined components, facilitating practical deployment. Despite its simplicity, FrodoPIR achieves competitive performance, demonstrating that strong privacy guarantees can be achieved without sacrificing clarity or efficiency in real-world applications.

Building upon these foundations, several works have introduced targeted optimizations to further improve performance. Angel et al. [1] proposed PIR with compressed queries and amortized batch processing, significantly reducing both communication overhead and computation time in repeated queries. Patel et al. [16] addressed the inefficiencies in sparse query settings, introducing mechanisms that exploit the sparsity of queries to minimize work for both client and server. Lin et al. [12] developed the "Doubly Efficient PIR" protocol, which introduces preprocessing techniques to reduce online query time while retaining strong privacy guarantees.

In parallel, other research has focused on minimizing server computation and enhancing scalability. Zhou et al. [18] introduced Piano, a PIR protocol that achieves sublinear server computation using lightweight cryptographic primitives, making it more suitable for large databases. Fisch et al. [6] developed ThorPIR, which employs Thorp shuffles to reduce preprocessing costs and support scalable deployments. Luo et al. [13] proposed a faster FHE-based PIR protocol that accelerates fully homomorphic evaluation using optimized ciphertext packing. Meanwhile, Kang and Schild [10] introduced Pirouette, a PIR protocol designed to support not only private data retrieval but also private aggregation, making it suitable for applications like distributed data analysis.

Additionally, Devet and Goldberg [5] explored hybrid approaches that combine the benefits of both information-theoretic and computational PIR, offering bandwidth savings while maintaining provable security in mixed trust models.

While these advances mark significant progress, empirical evaluation of PIR schemes on real-world, structured data remains limited. This study aims to address this gap by benchmarking the performance of modern single-server PIR protocols on the Open Food Facts dataset. The dataset includes diverse fields such as barcodes, nutritional values, and allergen tags, enabling tests on both narrow and broad query scopes. Performance metrics are compared against Redis, a widely used in-memory key-value store, to provide a practical baseline for assessing latency and throughput in realistic deployment scenarios.

IV. METHODOLOGY

SimplePIR, in particular, was selected for this study due to its general-purpose robustness, public implementation, and consistent performance across diverse problem settings. While recent optimizations target diverse use cases, including small records [1, 2], amortized server computation [18], and sublinear online time [17], while improving communication-computation trade-offs [5, 12, 13], SimplePIR remains a widely recognized baseline for single-server PIR.

To assess SimplePIR's retrieval correctness and performance under practical conditions, individual product queries were tested by setting the `PRODUCT_ID` environment variable and invoking the `TestQueryProduct` test. This step simulates a typical PIR use case, where a client retrieves a product entry without revealing which record was accessed.

Performance benchmarking was conducted using three Golang test cases implemented within the SimplePIR framework. `https://github.com/Alex-Verh/simplepiroff` The first test evaluated database size scalability across configurations ranging from 10 to 1,000,000 entries. The second test assessed record size variations using records from 8-bit to 512-bit lengths. The third test executed combinatorial tests examining all permutations of database and record sizes. Each test configuration was executed with multiple iterations to ensure statistical reliability of results. Performance metrics were captured through SimplePIR's native results computation logic integrated into the testing pipeline.

Network performance analysis incorporated a model where total network time was calculated as several times the network latency plus the sum of offline download, online download, and online upload sizes divided by the available bandwidth, based on predefined network conditions. The offline download component represented client preprocessing data size, online download reflected server response payload, and online upload constituted the query transmission size. Three network conditions were established for testing: a Good Network with 300 Mbps bandwidth and 10 ms latency, an Average Network with 100 Mbps bandwidth and 50 ms latency, and a Poor Network with 25 Mbps bandwidth and 200 ms latency.

Network	**Bandwidth**	**Latency**
Good Network	300 Mbps	10 ms
Average Network	100 Mbps	50 ms
Bad Network	25 Mbps	200 ms

TABLE I. Network Conditions

Comparative testing against Redis used equivalent Golang test scripts that maintained the same goal as the SimplePIR benchmarks. This included implementing identical database loading procedures to ensure comparable dataset configurations and computing the tests on variable datasets and record sizes. Redis was selected as a baseline because it is a widely adopted, high-performance in-memory key-value store commonly used for real-time data retrieval tasks. Its low-latency, high-throughput characteristics make it a practical reference point for assessing the performance trade-offs introduced by adding cryptographic privacy guarantees via PIR.

The web demonstration system featured a hybrid architecture combining Go and WebAssembly components. Server-side PIR operations were precisely instrumented using time measurements around the core cryptographic functions. The frontend uses HTML/JavaScript with a WebAssembly (WASM) module (compiled from Go) that provides fake PIR client functions, WASM generates random bytes to simulate cryptographic queries, and provides placeholder reconstruction functions, in order to simulate a real PIR process. Time measurement captures the real server-side SimplePIR computational costs but excludes the additional time implications that real client-side cryptography would add, plus network latency that would occur in a true PIR implementation. This way, the application already demonstrates some user experience tradeoffs, with a server-side PIR implementation.

V. MEASUREMENT ENVIRONMENT

A. Experimental Setup

The benchmarking environment differed from the original SimplePIR paper [8]. Whereas the reference implementation utilized AWS EC2 c5n.metal instances (36-core Intel Xeon Platinum 8000 series, 192GB RAM), evaluation was performed on an Apple MacBook Air with M4 chip (10-core CPU, 16GB unified memory) running macOS 14.5. This configuration represents a typical setup used in software development rather than an optimized cloud infrastructure.

Software configurations maintained maximum compatibility Go 1.22, identical SimplePIR repository [9], and equivalent test parameters. The experimental database contained 2^{33} 1-bit entries (≈1GB) as specified. The benchmarks were simulated offline since both configurations measured local computation exclusively.

B. Performance Comparison

Table II demonstrates significant throughput differences between environments despite identical algorithmic parameters:

TABLE II. SimplePIR benchmark comparison

Metric	AWS EC2	M4 Air	Difference
Throughput (GB/s/core)	10.0	25.1	+151%
Offline Download (MB)	121.0	123.6	+2.1%
Online Comm. (KB)	242	240	-0.8%
Execution Time (s)	300	75	-75%

The benchmark results reveal that in raw processing speed, the M4 Air significantly outperforms the AWS EC2 instance, completing tasks 2.5 times faster by achieving 25.1 GB/s per core compared to just 10 GB/s on the cloud server, and finishing executions in 75 seconds versus 300 seconds. Moreover, both systems show nearly identical data transfer requirements, with the M4 Air needing just 2% more initial download data (124 MB vs 121 MB) and maintaining essentially equal per-query communication costs (240 KB vs 242 KB). Also, these results confirm the theoretical consistency of communication costs across different hardware platforms.

C. DoublePIR Contextual Results

To further demonstrate the performance differences, Table III presents DoublePIR [8] benchmarks:

The DoublePIR tests show similar results as SimplePIR. The M4 Air delivers query responses nearly three times faster than the cloud server (20.8 vs 7.4 GB/s per core), while using exactly the same amount of data (16MB initial download) and nearly identical per-query communication (344KB vs 345KB).

TABLE III. DoublePIR benchmark comparison

Metric	AWS EC2	M4 Air	Difference
Throughput (GB/s/core)	7.4	20.8	+181%
Offline Download (MB)	16	16	0%
Online Comm. (KB)	345	344	-0.3%

These results demonstrate that modern laptops can deliver superior performance for private database queries compared to cloud servers. This way, developers working with privacy-preserving systems can also achieve faster processing times using high-performance consumer hardware rather than relying solely on cloud infrastructure.

D. Methodological Considerations

The benchmark results imply important considerations for proper interpretation. First, all throughput measurements must be understood in the context of their specific hardware configurations, as performance characteristics vary significantly between setups. Second, modern hardware can sometimes surpass cloud server capabilities for certain workloads. Third, the algorithm's communication patterns remain remarkably consistent across different platforms, demonstrating excellent portability.

To enable meaningful comparisons with prior published results, all subsequent analyses can be normalized using the following formula for the throughput results:

$$\text{Normalized\_Throughput} = \frac{\text{Measured\_Throughput}}{2.5}$$

This will provide equivalent metrics that account for hardware advancements while maintaining compatibility with the original paper. This way, with fair evaluation, current performance capabilities can be represented accurately.

VI. RESULTS

This section presents experimental results from benchmarking SimplePIR on the Open Food Facts dataset. All tests were executed on an Apple M4 system with 16GB unified memory, with each configuration run for 50 iterations to ensure statistical reliability. Performance metrics include answer time (server processing), setup time (precomputation), query time (client preparation), reconstruct time (client decoding), and communication overhead.

A. RQ1: How does efficiency (or associated costs) scale with varying database and record sizes (linear vs. exponential growth)?

Three controlled experiments were conducted to evaluate how database size and record length impact SimplePIR's operational efficiency.

1. *Database Size Scaling*

Tests with fixed 40-bit records across exponentially growing database sizes revealed several key patterns (Table IV). For small databases (10-1,000 entries), answer times remained below 1ms, demonstrating PIR's viability for lightweight applications. The 10,000-entry mark emerged as a practical threshold where setup time (3.35s) began exceeding query latency (273ms). At the maximum tested size of 1 million entries, answer time reached 429.6ms with 324.8s setup time, showing the protocol's polynomial scaling characteristics.

Answer Time exhibits near linear growth with complexity roughly equal to $\mathcal{O}(n^{1.1})$. A 10× increase in database size (from 10 to 100 entries) results in a 9× increase in answer time (from 0.02 ms to 0.18 ms). Afterwards, growth becomes more pronounced, increasing from 100K to 1M entries leads to a 16.8× rise in time (from 25.52 ms to 429.60 ms).

Setup Time follows clear polynomial growth, scaling approximately as $\mathcal{O}(n^{1.8})$. Increasing the database size from 1K to 100K (a 100× increase) leads to a 9,300× rise in setup time (from 349.52 ms to 32,440.77 ms). Beyond 10K entries, setup becomes the dominant contributor to total runtime, accounting for 92% of the cost at 1M entries. This behavior aligns with theoretical predictions for costly matrix precomputation operations [8].

Query Time scales logarithmically, with complexity $\mathcal{O}(\log n)$. From 10 to 1M entries, the increase in query time remains modest (154.86 ms to 1,185.93 ms, or roughly 2.4× per each 10× rise in entries number).

Reconstruct Time demonstrates sublinear growth, approximated by $\mathcal{O}(n^{0.7})$. When scaling from 100K to 1M entries, time increases from 48.70 ms to 190.00 ms (a 3.9× factor).

A notable threshold appears at 10,000 entries, where setup time (3,353.70 ms) surpasses query latency (273.35 ms). Those results already indicate some recommendations for optimizations, as for smaller databases (<10K), reducing query overhead is essential, whereas for larger databases (>100K), amortizing setup cost becomes critical.

2. *Record Length Impact*

Tests with full database size while varying record lengths from 8-bit to 512-bit revealed distinct computational patterns (Table V). For small records (8-32 bit),

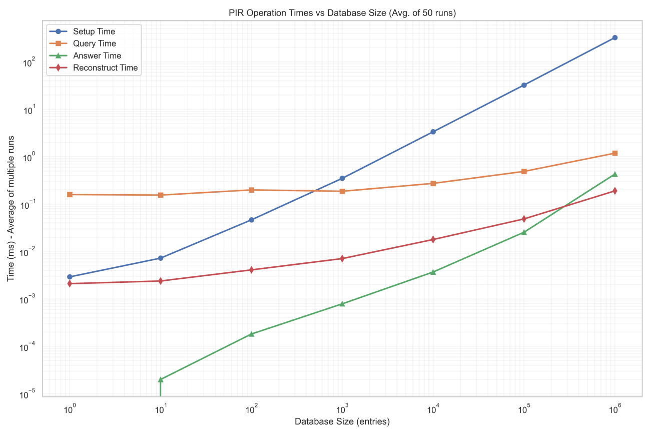


FIG. 1. Database size vs. performance metrics

processing times remained under 1 second, while larger records (256-512 bit) demonstrated quadratic growth characteristics.

Answer Time demonstrates quadratic growth with complexity $\mathcal{O}(d^2)$ for record size d. Doubling record length from 128-bit to 256-bit nearly doubles processing time (2,138.50 ms → 3,959.94 ms), while 512-bit records require 7,793.90 ms. This reflects the matrix-vector product complexity in SimplePIR's core operations.

Setup Time similarly shows quadratic scaling, increasing from 179.53 ms for 8-bit records to 10,621.55 ms for 512-bit records.

Query Time demonstrates linear growth ($\mathcal{O}(d)$), with 8-bit to 512-bit records showing a 6.9× increase (903.76 ms → 6,275.74 ms).

Reconstruct Time also follows linear scaling, with 8-bit to 512-bit records requiring 6.3× more processing (140.73 ms → 884.75 ms). The consistent 1.4ms/bit ratio suggests efficient decoding implementations.

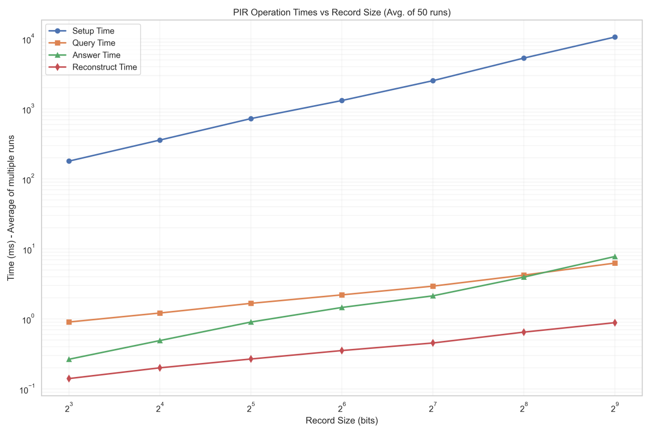


FIG. 2. Record size vs. performance metrics

A critical threshold emerges at 128-bit records, where answer time surpasses 2 seconds and setup time exceeds 2.5 seconds. At this point, the total operational cost

TABLE IV. Database size scaling (fixed 40-bit records)

DB Size (entries)	Answer (ms)	Setup (ms)	Query (ms)	Reconstruction. (ms)
1	0.00	2.92	159.17	2.10
10	0.02	7.25	154.86	2.39
100	0.18	46.51	199.16	4.10
1 000	0.79	349.52	186.46	7.10
10 000	3.70	3,353.70	273.35	17.91
100 000	25.52	32,440.77	489.04	48.70
1 000 000	429.60	324,809.77	1,185.93	190.00

TABLE V. Record size impact (3M entries)

Record Size (bits)	Answer (ms)	Setup (ms)	Query (ms)	Reconstruction (ms)
8	265.52	179.53	903.76	140.73
16	491.19	358.91	1,215.35	200.52
32	903.67	726.68	1,670.11	267.85
64	1,459.55	1,316.57	2,207.54	353.71
128	2,138.50	2,527.73	2,941.00	453.63
256	3,959.94	5,318.05	4,228.23	648.60
512	7,793.90	10,621.55	6,275.74	884.75

becomes significant for interactive applications.

These measurements denote that for small records (less than 32 bits), overheads are negligible, with sub-second processing times. Medium-sized records (32 to 256 bits) remain practical for batch processing scenarios. In contrast, large records (above 256 bits) introduce performance bottlenecks and require further optimization to support interactive use cases effectively.

3. Combined Effects

The full combinatorial tests revealed complex interactions between database size and record length, as summarized in Table VI. See graphical results in the appendix 4, 6, 5, and 7

For **small databases** (up to 1K entries), answer times remained consistently under 2 ms regardless of record size. Setup time varied more widely, ranging from 0.31 ms to 1,015.51 ms. Query time remained relatively stable, between 135.59 ms and 273.63 ms, confirming that fixed PIR overheads dominate at this scale.

In the case of **medium-sized databases** (1K to 100K entries), answer time grew linearly from 1.65 ms to 237.65 ms. Setup time showed quadratic scaling, increasing from 1,015.51 ms to 169,538.02 ms. In particular, the optimal performance configuration was observed at 10K entries with 64-bit records, achieving just 4.24 ms in answer time.

For **large databases** (beyond 100K entries), both database size and record length combined to produce significant polynomial effects. Answer time grew from 237.65 ms to 2,966.15 ms, while setup time increased sharply from 169 seconds to over 3,517 seconds.

Accordingly, the results suggests some optimizations based on the different size boundaries. In the case of small-scale deployments, simplicity should be prioritized; for medium-scale systems, efficient record packing becomes essential, and for large-scale scenarios, setup cost must be taken into account to maintain feasibility.

B. RQ2: What impact do different network conditions (high latency, low bandwidth) have on PIR performance?

Network conditions such as latency and bandwidth significantly influence the practical deployment of Private Information Retrieval (PIR), particularly in latency-sensitive applications. This section evaluates PIR performance under three representative network profiles, with varying database and record sizes, as summarized in Table IV. The total network time is estimated using the formula denoted below. Here, online download refers to the data the client receives during the query phase, online upload is the data sent by the client when making a query, and offline download is any precomputed data the client downloads in advance to reduce online latency.

$$T_{net} = 3 \times \text{latency} + \frac{\text{offline\_dl} + \text{online\_dl} + \text{online\_ul}}{\text{bandwidth}}$$

1. Database Size Scaling

Table VII reports the estimated network time for increasing database sizes, assuming fixed-size 40-bit records. Offline download dominates the communication volume, while online downloads and uploads remain negligible until larger database sizes. For small databases (fewer than 10,000 records), latency is the dominant factor across all network conditions, accounting for most of the network delay. As the database size grows, differences in bandwidth lead to increasingly different performance. Under poor network conditions, the response time exceeds one second once the database

TABLE VI. Combined size performance (selected configurations)

DB Size	Rec. (bits)	Answer (ms)	Setup (ms)	Query (ms)	Recst. (ms)
1	8	0.00	0.31	135.59	0.54
100	32	0.00	26.44	145.49	2.44
1K	128	1.65	1,015.51	273.63	11.00
10K	64	4.24	4,651.23	278.41	20.95
100K	256	237.65	169,538.02	918.33	142.79
1M	512	2,966.15	3,516,970.86	3,496.11	533.10

reaches 100,000 records. At one million entries, poor networks are approximately 13 times slower than good ones.

2. *Record Size Impact*

Table VIII presents results for varying record sizes while using the full size of the database, at approximately three million entries. Larger records increase both offline and online communication costs, with total network time growing roughly linearly with record size. Despite this, total online communication remains relatively modest, under 100 KB even for 512-bit records. However, in constrained environments such as mobile networks with poor conditions, response times surpass 11 seconds for record sizes of 256 bits or more, making such configurations impractical.

3. *Deployment Considerations*

Results indicate that favorable network conditions can support large-scale Private Information Retrieval (PIR) deployments involving over one million records, while maintaining sub-second response times for records up to 256 bits. Under average network conditions, configurations with up to 100,000 records and 128-bit record sizes remain feasible, achieving response times under three seconds. In contrast, poor network environments exhibit a steep performance degradation, requiring either significantly smaller database sizes (fewer than 10,000 records) or the incorporation of caching strategies to maintain acceptable responsiveness. These findings align with user experience research suggesting that web users begin to perceive delays as disruptive beyond the two to three second threshold [15].

C. RQ3: How do the computational and bandwidth costs of private retrieval (using SimplePIR) compare to non-private methods when querying the Open Food Facts database under identical conditions?

The benchmark results reveal fundamental differences between SimplePIR and Redis across all tested configurations. Redis demonstrates constant-time performance regardless of database size, while showing linear scaling only with record length.

Key Observations reveal that Redis maintains an approximately zero millisecond response time across all database sizes, whereas SimplePIR experiences increasing latency ranging from 159 ms to 1,186 ms. This constant-time performance of Redis arises from its hash-based lookup mechanism, which operates in constant time complexity (O(1)). In contrast, SimplePIR's latency grows polynomially due to the homomorphic operations performed on all records.

The data shows that Redis exhibits near-zero latency even for record sizes up to 1 KB. In contrast, SimplePIR latency increases approximately quadratically as record size grows. The shortest Redis response times mainly come from the time it takes for data to travel over the network (which is not a factor in local testing), plus the time needed to access memory and convert data into the Redis format.

1. *Architectural Differences*

The observed performance differences originate from fundamental design decisions. Redis achieves direct lookup through in-memory hash tables with O(1) access time and incurs no cryptographic overhead. However, this approach reveals query patterns and accessed records, thus offering no privacy guarantees. Redis retrieves only the requested record, which contributes to its efficiency.

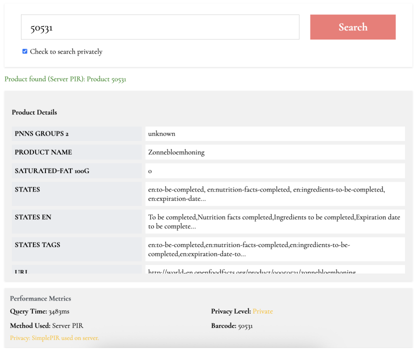
SimplePIR, on the other hand, relies on homomorphic processing that involves computational work proportional to the entire database size. The use of cryptographic operations adds a constant multiplicative overhead. Importantly, SimplePIR preserves privacy by hiding access patterns, but this requires processing every record for each query, resulting in higher latency.

2. *Practical Implications*

These benchmarks quantify the inherent tradeoff between privacy and performance. In scenarios where privacy is not required, Redis outperforms SimplePIR. For applications involving sensitive access patterns, SimplePIR introduces an overhead of one to three seconds for queries on databases containing one million entries. A practical deployment strategy might combine PIR for privacy-sensitive queries with conventional retrieval methods for non-sensitive interactions, thereby balancing privacy guarantees with system responsiveness.

D. Bonus: How does server-side PIR latency impact perceived user experience compared to traditional retrieval when querying Open Food Facts?

As a demonstration, this project implements a partial PIR system that highlights the performance impact of PIR in real-world scenarios, even though it does not provide full privacy guarantees. In the current setup, cryptographic operations are performed only on the server side, while the client merely simulates PIR queries using randomized data. Because the client does not generate actual encrypted queries, the system does not hide the user's access pattern, and thus does not offer true PIR-level privacy. Despite this limitation, the demo allows users to experience the computational overhead (approx. 1.1 seconds vs. milliseconds) introduced by PIR operations compared to direct lookups on the same OpenFoodFacts dataset (approx. 2.8M products) 3. The hybrid architecture combines a Golang-based backend with genuine lattice-based cryptographic computations and a WebAssembly/JavaScript frontend that mimics client behavior. This prototype might serve as a showoff of the trade-offs involved and serve as the groundwork for a fully private PIR system in the future by incorporating real client-side cryptography via WASM-enabled libraries.



Product found (Server PIR): Product 50531

Product Details

PNNS GROUPS 2	unknown
PRODUCT NAME	Zonnebloemhoning
SATURATED-FAT 100G	0
STATES	en:to-be-completed, en:nutrition-facts-completed, en:ingredients-to-be-completed, en:expiration-date...
STATES EN	To be completed,Nutrition facts completed,Ingredients to be completed,Expiration date to be complete...
STATES TAGS	en:to-be-completed,en:nutrition-facts-completed,en:ingredients-to-be-completed,en:expiration-date-to...
URL	http://world-en.openfoodfacts.org/product/00050531/zonnebloemhoning

Performance Metrics

Query Time: 3483ms
Privacy Level: Private
Method Used: Server PIR
Barcode: 50531
Privacy: SimplePIR used on server.

FIG. 3. SimplePIR Demonstration Platform

VII. CONCLUSION

This study demonstrates the performance of using the SimplePIR protocol for privacy-preserving queries on large-scale real-world datasets, exemplified by the Open Food Facts database. Through controlled experimentation, protocol's correctness, scalability, and efficiency were evaluated under a range of database sizes, record lengths, and network conditions.

The conversion of the Open Food Facts CSV dataset into a binary format enabled efficient integration with the SimplePIR framework. The results show that SimplePIR can handle databases with up to one million entries and variable record sizes without compromising functionality. Performance benchmarks revealed that query latency and resource usage grow predictably with database and record sizes, validating the protocol's practical scalability.

Network performance modeling further highlighted the importance of communication efficiency in PIR schemes. The impact of network latency and bandwidth was most pronounced in poor network conditions, where total query time increased significantly due to high upload and download costs. Conversely, under good network conditions, SimplePIR demonstrated near-interactive performance, suggesting its viability in modern web and mobile environments with robust connectivity.

Comparative analysis with a Redis baseline confirmed that, although Redis performs faster in plaintext settings, SimplePIR offers competitive results while preserving client privacy.

In conclusion, SimplePIR presents a practical and reproducible system for deploying private information retrieval protocols at scale. While further optimizations can reduce communication and computation overheads, this work analyses a strong foundation for future developments in privacy-preserving database access and their real-world deployment. However, the current study is limited in scope to a single PIR protocol, a specific dataset, and limited test environments. Real-world deployments may face additional challenges, including integration with dynamic databases, multi-user concurrency, and more complex issues. Future work could explore additional PIR schemes such as DoublePIR or recent compressed variants, incorporate true client-side cryptographic query generation for full privacy guarantees, and evaluate end-to-end user experience to further explore the practical implications of PIR systems.

USE OF AI TOOLS

Some of the sections in this work were refined using an AI language model (OpenAI) to improve clarity, grammar, and phrasing. The use of this tool was limited to paraphrasing and language corrections; all research content, analysis, and conclusions are original and solely the work of the author.

TABLE VII. Network performance by database size (fixed 40-bit records)

DB Size (entries)	Offline DL (KB)	Online DL (KB)	Online UL (KB)	Good (s)	Average (s)	Poor (s)
1	20	0	0	0.031	0.152	0.606
10	40	0	0	0.031	0.153	0.613
100	100	0	0	0.033	0.158	0.632
1,000	280	0	0	0.037	0.172	0.690
10,000	900	0	0	0.054	0.222	0.888
100,000	2,840	2	2	0.106	0.378	1.510
1,000,000	8,960	8	8	0.269	0.868	3.472

TABLE VIII. Network performance by record size (3M entries)

Record (bits)	Offline DL (KB)	Online DL (KB)	Online UL (KB)	Good (s)	Average (s)	Poor (s)
8	6,676	6	6	0.208	0.685	2.740
16	9,448	9	9	0.282	0.907	3.629
32	13,360	13	13	0.387	1.221	4.884
64	17,668	17	17	0.502	1.566	6.265
128	24,076	23	23	0.673	2.080	8.319
256	34,776	33	33	0.959	2.937	11.749
512	48,760	47	47	1.333	4.058	16.233

TABLE IX. Redis performance across database sizes (17,808-bit records)

DB Size	Record (bits)	Answer (ms)	Offline DL (KB)	Online DL (KB)	Online UL (KB)	Query (ms)	Recst. (ms)
1	17,808	0.00	0.00	2.17	0.00	0.00	0.00
10	17,808	0.00	0.00	2.17	0.00	0.00	0.00
100	17,808	0.00	0.00	2.17	0.00	0.00	0.00
1,000	17,808	0.00	0.00	2.17	0.00	0.00	0.00
10,000	17,808	0.00	0.00	2.17	0.00	0.00	0.00
100,000	17,808	0.00	0.00	2.17	0.00	0.00	0.00
1,000,000	17,808	0.00	0.00	2.17	0.00	0.00	0.00

TABLE X. Redis performance across record sizes (3M entries)

DB Size	Record (bits)	Answer (ms)	Offline DL (KB)	Online DL (KB)	Online UL (KB)	Query (ms)	Recst. (ms)
3M	8	0.00014	0.00	0.00098	0.00	0.00014	0.00
3M	16	0.00	0.00	0.00195	0.00	0.00	0.00
3M	32	0.00	0.00	0.00391	0.00	0.00	0.00
3M	64	0.00	0.00	0.00781	0.00	0.00	0.00
3M	128	0.00	0.00	0.01562	0.00	0.00	0.00
3M	256	0.00	0.00	0.03125	0.00	0.00	0.00
3M	512	0.00	0.00	0.06250	0.00	0.00	0.00
3M	1,024	0.00	0.00	0.12500	0.00	0.00	0.00

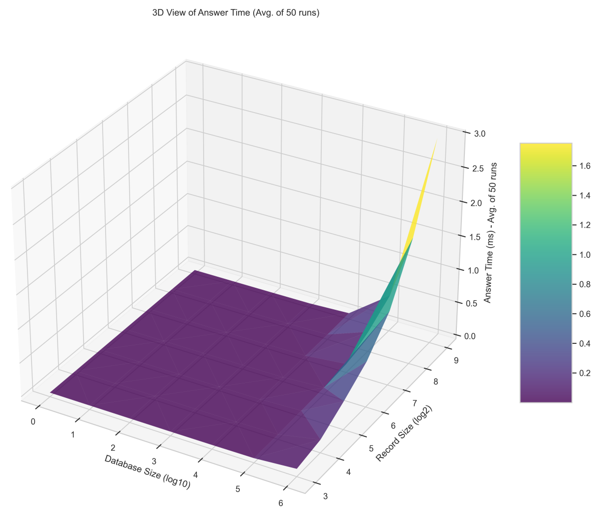


FIG. 4. 3D Average Answer Time

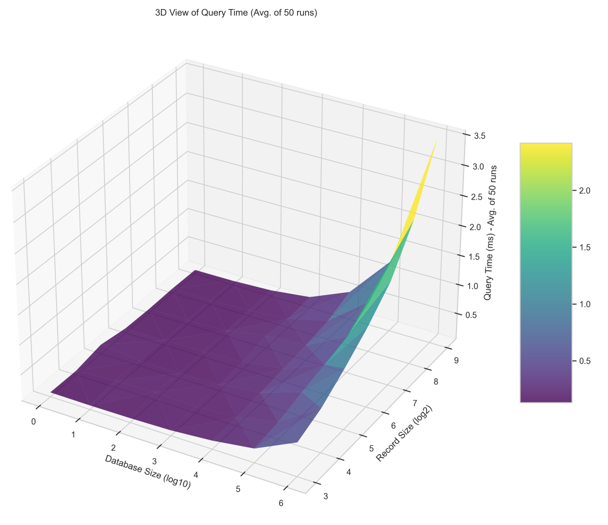


FIG. 5. 3D Average Query Time

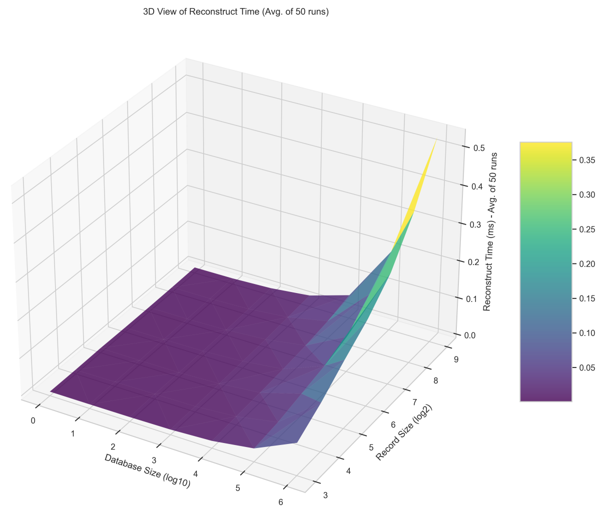


FIG. 6. 3D Average Reconstruct Time

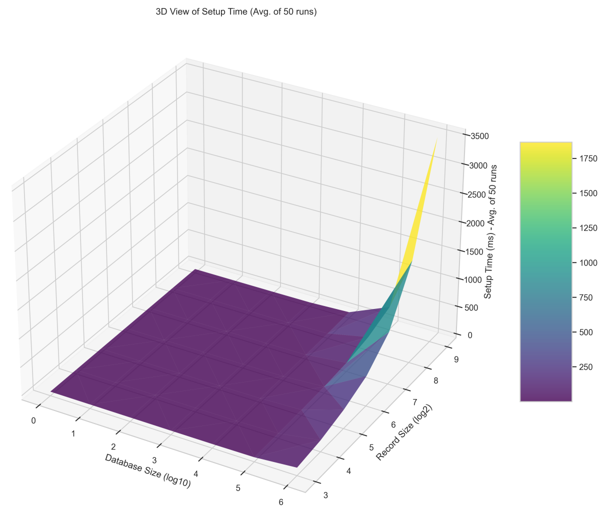


FIG. 7. 3D Average Setup Time

[1] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. Pir with compressed queries and amortized query processing. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2021.

[2] Alexander Burton, Samir Jordan Menon, and David J. Wu. Respire: High-rate PIR for databases with small records. Cryptology ePrint Archive, Paper 2024/1165, 2024.

[3] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 41–50, 1995.

[4] Alan Davidson et al. Frodopir: Simple, scalable, single-server private information retrieval. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

[5] Casey Devet and Ian Goldberg. The best of both worlds: Combining information-theoretic and computational pir for communication efficiency. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2014.

[6] Ben Fisch, Alessandro Lazzaretti, Zeyu Liu, and Charalampos Papamanthou. ThorPIR: Single server PIR via homomorphic thorp shuffles. Cryptology ePrint Archive, Paper 2024/482, 2024.

[7] William Gasarch. A survey on private information retrieval. *Bulletin of the EATCS*, 82:72–107, 2004.

[8] 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 *Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP)*, 2022.

[9] 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 repository. 2022.

[10] Jaewon Kang and Lukas Schild. Pirouette: Query efficient single-server PIR. In *Cryptology ePrint Archive, Paper 2025/680*, 2025.

[11] Eyal Kushilevitz and Rafail Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 364–373, 1997.

[12] Wei-Kai Lin, Erik Mook, and Daniel Wichs. Doubly efficient private information retrieval and fully homomorphic RAM computation from ring LWE. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing (STOC 2023)*, pages 595–608, 2023.

[13] Mingyu Luo, Feng-Hao Liu, and Hongrui Wang. Faster FHE-based single-server private information retrieval. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*, pages 1405–1419, 2024.

[14] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 1–23. Springer, 2010.

[15] Fiona Fui-Hoon Nah. A study on tolerable waiting time: How long are web users willing to wait? *Behaviour Information Technology*, 2000.

[16] Sarvar Patel, Joon Young Seo, and Kevin Yeo. Don't be dense: Efficient keyword PIR for sparse databases. In *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[17] Yinghao Wang, Xuanming Liu, Jiawen Zhang, Jian Liu, and Xiaohu Yang. Crust: Verifiable and efficient private information retrieval with sublinear online time. In *Proceedings of the 2022 ACM Conference on Computer and Communications Security (CCS)*, 2022.

[18] Mingxun Zhou, Alice Park, Elaine Shi, and Wenting Zheng. Piano: Extremely simple, single-server PIR with sublinear server computation. In *2024 IEEE Symposium on Security and Privacy (SP)*, 2024.