Enhancing Security for Columnar Storage and Data Lakes

Lotte Felius
felius@cwi.nl
Supervised by Peter Boncz
Centrum Wiskunde & Informatica
Amsterdam, Netherlands

ABSTRACT

In this PhD project, we will investigate enhancements for columnar storage file formats, which play a crucial role in database workloads, but also increasingly in machine-learning workloads, with the overall goal of improving security in next-gen Data Lakes. This is investigated in three lines of proposed work to develop (i) a vector-friendly encryption file layout that allows efficient processing on both CPUs and GPUs in our new FastLanes format, (ii) hybrid encrypted query processing, in which decryption happens client-side, introducing new techniques that allow cloud servers to skip encrypted data in predicate pushdown; and (iii) oblivious data access techniques that exploit the compaction processes already necessary for data life-cyle management in Data Lakes.

VLDB Workshop Reference Format:
Lotte Felius. Enhancing Security for Columnar Storage and Data Lakes. VLDB 2024 Workshop: VLDB Ph.D. Workshop.

1 INTRODUCTION

Data *privacy* and *security* are fundamental rights in our society. This includes securing software and systems that run in cloud services, which are commonly used for both computations and storage of data. To strengthen the security of the *end-to-end query processing pipeline*, we ought to secure *data at rest* (i.e. while being stored on disk), *data in use* (i.e. while being processed), and *data access patterns*, from which a malicious actor could infer information about the underlying data. Security of information processing however has no one-size-fits-all solution and should be defined in terms of a threat model specific to an application. For instance, one could assume that an attacker has the ability to (i) obtain temporary access to the server and obtain a snapshot, (ii) monitor cloud storage access patterns and read or tamper with cloud-stored data, or (iii) gain full access to cloud machines. Data Lakes are a popular way to manage large amounts of data stored in the cloud, using open data formats. Examples are Apache Iceberg and Delta Lake [6], which add a layer of abstraction over tables stored in the cloud as large collections of Parquet data files. This layer contains meta-data that specifies a table schema and lists which files hold its (deleted) rows, enabling e.g. access to multiple versions of the stored data.

Lack of security. At the very least, Data Lakes and columnar formats should support encryption – a feature already supported

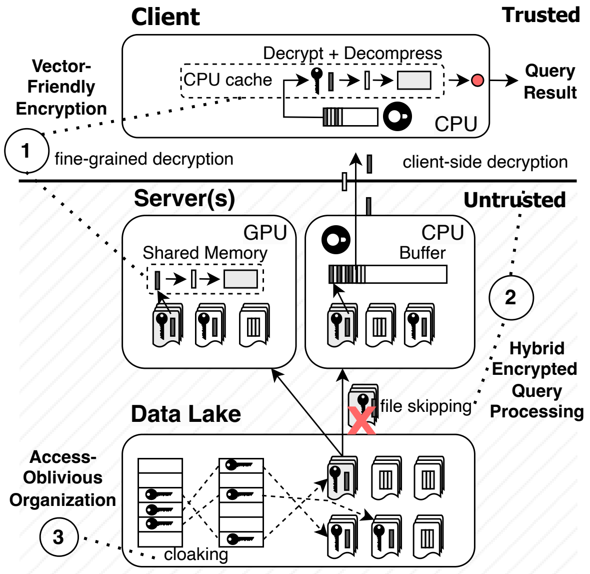

Figure 1: We foresee three lines of work: (i) Vector-Friendly Encryption to make decryption fine-grained to integrate it in vectorized execution and optimize cache usage, (ii) Hybrid Encrypted Query Execution, that can move decryption to the client, while still allowing server-side skipping on encrypted data, (iii) to introduce data cloaking in data re-organization to create Access-Oblivious Data Lakes.

by Parquet and Iceberg but not yet by Delta Lake [6]. FastLanes [1, 2], a new columnar storage file format, is still in the prototype stage and does not include any security features yet. At this point, existing security measures (i.e. simple encryption) in both Data Lakes and columnar file-formats are at most *partly* sufficient in all three aforementioned scenarios. For example, processing encrypted data requires its decryption into plaintext, where it is vulnerable in cases (ii) and (iii). In theory, even in the case of (i), an attacker could read decrypted data present in data caches and use this information to infer other values. In addition, in case of an attacker in scenario (ii) and (iii), plain encryption does not provide any mitigation for *access pattern leakage*. This might reveal sensitive information about the underlying data; in particular when data access patterns are not uniformly distributed.

Security is a trade-off. Solutions that enhance security often greatly reduce the performance or practicality of a system. Encryption on its own is already an expensive process, but we also need to encrypt metadata to avoid leakage. As a consequence, *data skipping techniques* in columnar file-formats and Data Lakes become unusable. Instead, when processing encrypted data, we are

Proceedings of the VLDB Endowment. ISSN 2150-8097.

often forced to decrypt chunks of data that would be skipped otherwise. A straightforward solution for securely processing *data in use* while exploiting efficient data skipping techniques would be using Trusted Execution Environments (TEEs), such as originally Intel SGX [9], and more recently *Confidential Virtual Machines* (CVMs), such as AWS Nitro Enclaves, Intel TDX and AMD-SEV. However, its availability is limited and TEEs always reduce query processing performance due to encryption mechanisms and expensive cache misses [8, 9, 13, 15]. To mitigate the performance over in TEEs, we therefore aim to minimize cache misses by means of e.g. *vector-friendly encryption.*

Alternatives for TEEs. More concerns regarding the usage of TEEs include that (i) e.g. Nitro requires trust in AWS, (ii) the security research community typically quickly succeeds in finding vulnerabilities and (iii) TEEs do not prevent *access pattern leakage.* Therefore, a TEE should be considered as a hardening technology rather than guaranteed security for processing sensitive data. However, despite the doubts regarding their sufficiency for some applications, cloud vendors are increasingly adopting TEEs. This is moslty due to a lack of mature alternatives to secure *data in use.* For example, systems can adopt Homomorphic Encryption (HE) to perform computations on encrypted data, or Property Preserving Encryption (PPE) to evaluate an encrypted predicate. HE in fact has the property to perform additions and multiplications on encrypted data such that $Enc_{k_1}(A) + Enc_{k_2}(B) = Enc_k(A + B)$, where A and B are distinct plaintexts encrypted with key k_1 and key k_2 where $k_1 = k_2 = k$. However, HE is too expensive in terms of performance and storage overhead to be practical yet for an analytical DBMS and PPE suffers from serious security concerns.

A more feasible alternative would be using a *split*-query processing model [17] that performs as much processing as possible on the server, *without* decrypting any sensitive data. Instead, it transports the intermediate and encrypted query result to the client, which then takes care of the decryption and the computation of the final query result. Here, it is vital to minimize the data volume sent to the client for faster query processing. A novel form of *split*-query processing was recently introduced by MotherDuck [7], which hosts a DuckDB cloud service. They leverage the fact that any DuckDB application contains a local DuckDB, since all DuckDB APIs *embed* one, by introducing *hybrid query processing*. In their processing model, query work is split between a cloud DuckDB server and a DuckDB instance embedded in the client application. Hence, it could be useful to introduce encryption in hybrid query processing, according to the split-query processing model mentioned above.

A GPU-friendly design. In this PhD project, we focus on security for columnar storage *and* Data Lakes, since both are intertwined. We however think that the current status quo with Parquet as the dominant data file format for Data Lakes could be disrupted by the increased importance of Machine Learning (ML) workloads. To enable secure computing for ML workloads, the TEE space also provides secure environments for GPUs – e.g. NVIDIA introduced confidential computing hardware features in its Hopper and Blackwell GPUs[1]. However, decryption and other pre-processing tasks of ML are commonly pushed upon CPUs, making sensitive data more vulnerable while being processed. Therefore, we aim to push Data Lake scans, which perform decryption and decompression, into the GPU. This is challenging for Parquet, since it heavily depends on compression of relatively large pages using *general-purpose codecs* like snappy, zstd and lz4. The decompression algorithms for these have an irregular control-flow, which causes control divergence among the threads in a GPU warp. The coarse decompression granularity of Parquet does not fit the GPU caches, leading to spilling of decompressed data to global memory of the GPU, which in turn significantly increases bandwidth usage. Therefore, reading Parquet files on a GPU wastes many of its resources [18]. *FastLanes* is a novel columnar storage file format under design at CWI [1, 2], that achieves an order of magnitude higher decompression performance than Parquet on CPUs, because compilers can auto-vectorize its decompression kernels into efficient SIMD instructions [1]. This is enabled by its novel *data-parallel* column encodings that also naturally map on efficient GPU kernels, and is *fine-grained* so it does not overrun GPU caches. Therefore, we aim to focus on enhancing security for the FastLanes file-format, since it is suitable for both CPUs and GPUs [2].

Research Questions. The main aim of this PhD research is to investigate how we can enhance the security of columnar storage formats, and making the design of FastLanes security-friendly, such that it becomes a building block for (more) secure next-gen Data Lakes. We formulate the following three research questions:

(1) **Vector-Friendly Encryption**: how to enhance the data format such that it allows fine-grained vectorized decryption, to make it performant on both CPUs and GPUs?
(2) **Hybrid Encrypted Query Processing**: How can we leverage encryption techniques such as homomorphic encryption in ZoneMap/MinMax metadata to allow safe server-side *encrypted data skipping* and how can we optimize the column encryption layout to support *encrypted predicate pushdown*?
(3) **Access-Oblivious Data Lake Re-organization**: how can we enhance the re-organization strategies already part of Data Lakes for compacting and re-clustering of large amounts of big data files to *cloak* access patterns to these files – and what consequences could this have for the security design of the FastLanes format?

2 RELATED WORK

Columnar File-Formats such as Parquet and ORC already offer some form of encryption. For instance, Parquet provides modular encryption, where footers, headers, metadata, columns and data pages are encrypted separately. Its page encryption has a default granularity of 1MB. This means that in a scan of multiple columns, the decrypted pages will typically not fit the L1/L2 CPU cache. However, in scenarios where *data in use* must be processed securely, Intel SGX and AMD-SEV are often used, and in such TEEs, cache misses are very expensive [8, 9, 13, 15]. In addition, Parquet and ORC are not optimized for ML workloads. Bullion [12] is a novel columnar storage format that addresses specific characteristics of ML datasets, such as columns holding quantized floating-point types, compressed sparse feature vectors and very wide tables, for which it provides random meta-data access without deserialization.

[1]www.nvidia.com/en-us/data-center/solutions/confidential-computing

However, while Bullion - similar to Data Lake formats - proposes deletion vectors as a solution for deletion-compliance, it does not focus on security.

Secure Databases ensure *privacy* and *integrity* by providing cryptographic methods, e.g. homomorphic encryption, for computing query results on encrypted data [5, 14]. However, existing solutions have practical issues or are not performant. The early prototype CryptDB [14] as well as the currently most mature solution, Microsoft Always Encrypted [5] are both transactional systems rather then analytical. Using per-column encryption in a row-based storage layout, introduces significant storage overhead, because values are encrypted individually. Regarding analytical queries, Monomi [17] introduced split client-server query execution, enabling to execute arbitrarily complex queries over encrypted data between a server and a client. However, Monomi used PostgreSQL as its database engine, which is an OLTP system; hence columnar storage, MinMax statistics and vectorized execution (or decryption) with or without predicate pushdown did not apply there.

Data Lakes can provide tabular or columnar- encryption to enhance security, by encrypting both data files and meta-data files. Currently Apache Iceberg supports this, but Delta does not (yet) [6]. However, there is no security mitigation for passive attackers that can observe access patterns. Opaque [19] hid access patterns by performing *full table scans*, which imposes a significant performance penalty. It used a modified SparkSQL to compute in TEEs, and because it was a distributed system, it also ensured that communication between distributed operators was oblivious, e.g., using its *pad* mode, which came at large additional cost. As an alternative, ObliDB [10] introduced oblivious physical operators inside a hardware enclave and introduced index data structures with oblivious access. However, Data Lakes do not support indexes, and focus on predicate pushdown (data skipping) to reduce table scan cost. Our work seeks to enable this while still providing oblivious access.

3 ENVISIONED APPROACH

This PhD research poses several challenges; we will focus on three lines of research, depicted in Figure 1, which entail (i) *Vector-Friendly Encryption*, (ii) *Hybrid Encrypted Query Processing* and (iii) *Access-Oblivious Data Lake Reorganization.*

3.1 Vector-Friendly Encryption

Efficient I/O in columnar storage is usually performed in blocks and cached in compressed form in a buffer manager. A *vectorized* query processor then takes small compressed vectors, where the decompressing scan is the source operator in a query pipeline. We can leverage this vectorized processing model by encrypting vectors rather then single values or large pages, since *vector-friendly decryption* can speed-up query processing by decrypting and decompressing in one-pass by keeping (intermediate) data in the processor cache. Prior research [4, 8] suggests that the synergy between compression and encryption could improve query processing performance. We therefore argue that it makes sense to first compress and then encrypt data to mitigate encryption overhead. However, before integrating well-performing vector-friendly encryption into FastLanes [1], we will investigate the synergy of first compressing and then encrypting data in DuckDB [16], using its compression API to test the performance on different readily-available lightweight compression methods, such as ALP [3], RLE and DELTA before designing a vector-friendly encryption mechanism, that is suitable for both CPUs and GPUs.

Moreover, when processing encrypted data, we want to be able to skip vectors that are not required to compute the query result. This can be achieved by using AES Counter mode (`AES-CTR`) for [en/de]cryption, from which a fast open-source implementation is available by OpenSSL. A chunk of plaintext data to be encrypted is usually randomized with a nonce or initialization vector (IV) to ensure different ciphertexts. `AES-CTR` requires a nonce to be *unique* but not *random*. We can leverage this characteristic to reduce storage overhead by deterministically constructing the nonce from the position of the vector, and e.g. the column name. Note that we should carefully evaluate the possibilities for constructing the nonce; nonce reusage with the same key can cancel out the encryption. We are aware that this mode of encryption does not offer an integrity check. Hence, we will investigate alternatives to verify integrity, such as comparing a hash or pre-computing a checksum. Lastly, while it makes it harder for the attacker to pinpoint which exact tuples are accessed, compression does give away important characteristics of the underlying data. For example, when data is highly compressible, the attacker could infer the actual data in combination with e.g. the column name. Therefore, we investigate ways to mitigate leaking the compression size, using padding or clustering of thin columns belonging to multiple row-groups.

3.2 Hybrid Encrypted Query Processing

The hybrid query processing model of MotherDuck [7] is beneficial in a threat model were we assume that the server (even if using a TEE) is untrusted; the attacker can e.g. continuously monitor the server's main memory and read the content of caches. In this scenario, we argue that data can remain encrypted on the remote server, but once queried, we could push the decryption of the *most sensitive* data to the local client, similar to [17]. This ensures a higher level of security; encryption of data happens solely on the client with their private key, which is also stored on their own machine. Therefore, we aim to extend the hybrid query processing model into *hybrid encrypted query processing* (HEQP), that allows to securely and efficiently process encrypted data on the client.

Encrypted Data Skipping. An important challenge in HEQP is to avoid transferring unneeded data from server to client. However, analytical file-formats and Data Lakes cannot directly make use of data skipping techniques, since the metadata of sensitive data needs to stay encrypted on the server. Therefore, we propose *encrypted data skipping* (EDS) for remote filtering of data while keeping data encrypted at the server. This can be accomplished through encrypting lightweight indexes, such as MinMax with (Semi) Order Preserving Encryption (OPE) or homomorphic encryption (HE). This will allow to evaluate an encrypted predicate over encrypted data to determine whether the corresponding column chunk, row group, or vector should be skipped. There are some challenges however with these encryption techniques. Specifically, HE is computationally intensive, OPE leaks order, and both have a large data footprint per encrypted value. However, to mitigate the leakage by OPE, CryptDB [14] encrypts tuples in so-called layers of an onion, such that the order is only exposed to the attacker for a short period of time. Note that even when the order of MinMax statistics leaks, the attacker still does not know which exact tuples will be accessed. In this line of research, we will carefully evaluate how to best implement different cryptographic principles for providing EDS while minimizing the performance and storage overhead. Further, we will investigate how we can integrate these principles into a column encryption layout for FastLanes, to support *encrypted predicate pushdown* (EPP). We will evaluate the HEQP model with standard benchmarks such as `TPC-H` and `TPC-DS`, and expect that EPP will speed-up encrypted query processing client-side as it (i) minimizes network transfer and (ii) decrypts only chunks of data that are necessary for computing the final result.

Pushing decryption to the client requires a hybrid query optimizer, such as provided by MotherDuck [7]. The hybrid optimizer splits a query plan into pipeline fragments that are executed locally (on the client) and remotely (on a server). To enhance security, we will need to add optimizer rules to ensure that sensitive columns are *only* processed client-side. An open question here remains how to adapt the hybrid query optimizer, such that it chooses the most performant hybrid query plans when encrypted data is included.

3.3 Access-Oblivious Data Lake Reorganization

A table stored in a Data Lake typically consists of a collection of Parquet files that hold data, plus some meta-data files that describe this collection. To update and manage newly stored data, Data Lakes perform data life-cycle management, in which these collections of files are periodically re-organized, re-clustered and/or re-partitioned. Thus, many small newly written files are re-written into fewer larger files to enable efficient accesses. We think that we can leverage this continuous maintenance by making access patterns more oblivious. There are multiple ways to achieve this; for example, a system that processes a continuous query workload can artificially delay returning results of particular queries, in order to obfuscate the relationship between query processing activities. This is already unintentionally achieved in systems that e.g. use *group commit*, where multiple updates are queued together and written in bulk to reduce disk I/O. In addition, such updates could be written in any random order. When the workload intensity decreases, one could think about running decoy queries, i.e. executing artificial queries that fuzz the system to introduce noise. These artificial queries can be leveraged to do periodical maintenance.

Cloaking partial data access. We outlined our goal of enabling efficient server-side predicate push-down on encrypted data earlier. However, if a query accesses only certain ranges of certain files, this leaks information, also if these files are encrypted. Even if we use query result fuzzing and decoy queries suggested above, the workload as a whole would still leak information, as certain areas would be much hotter than others. This leaks will happen by queries accessing only certain files and column ranges (skipping), and also from accessing only certain columns and even tables (projection). To avoid leakage, one could see the whole Data Lake as one collection of blocks, and scan these continuously in sequential fashion. Even though shared or cooperative scan approaches [11] can reduce the throughput damage of such a repeated-full-scan approach, the damage to query latency is too large to be practical. The nice property of a full scan workload is that it has (roughly) the same access frequency to all blocks. We think this could also be achieved by a system that continuously rewrites data, as Data Lakes do anyway. Such a system could e.g., replicate blocks that have a too high access frequency, reducing that by the replication degree. With our research, we opt for an access-oblivious Data Lake system that achieves good query latency and throughput, at limited overhead, also in terms of replication degree. We will evaluate our algorithms by extensive simulations of common workloads in Data Lakes.

4 CONCLUSIONS

In this PhD research plan, we motivate and outline our goal to enhance the security of columnar storage formats and Data Lakes. We reviewed existing columnar storage formats, secure databases and security in Data Lakes to identify our research questions. Aiming to answer these, we outlined three lines of research from which the goal is to find (i) an efficient *vectorized-friendly encryption* mechanism suitable for both CPUs and GPUs, (ii) a way to integrate *hybrid encrypted query processing* enabling *encrypted data skipping* for client-server architectures in columnar storage layouts, and (iii) to make *data accesses more oblivious* in Data Lakes. Ultimately, we envision to integrate our findings into *FastLanes* [1], and serve as a building block for future file-formats.

REFERENCES

[1] A. Afroozeh and P. Boncz. 2023. The FastLanes compression layout: Decoding> 100 billion integers per second with scalar code. *PVLDB* 16, 9 (2023), 2132–2144.

[2] A. Afroozeh, L. Felius, and P. Boncz. 2024. Accelerating GPU Data Processing using FastLanes Compression. In *DaMoN 2024*.

[3] A. Afroozeh, L. Kuffo, and P. Boncz. 2023. ALP: Adaptive Lossless floating-Point Compression. *Proc. SIGMOD* 1, 4 (2023), 1–26.

[4] S. Ansmink. 2021. *Encrypted Query Processing in DuckDB*. Master's thesis. Vrije Universiteit Amsterdam, The Netherlands.

[5] P. Antonopoulos, A. Arasu, et al. 2020. Azure SQL Database Always Encrypted. In *ACM SIGMOD 2020*.

[6] M. Armbrust, T. Das, L. Sun, B. Yavuz, and et. al. 2020. Delta lake: high-performance ACID table storage over cloud object stores. *PVLDB* 13, 12 (2020), 3411–3424.

[7] RJ. Atwal, Peter A. Boncz, et al. 2024. MotherDuck: DuckDB in the cloud and in the client. In *CIDR 2024*.

[8] I. Battiston, L. Felius, S. Ansmink, L. Kuiper, and P. Boncz. 2024. DuckDB-SGX2: The Good, The Bad and The Ugly within Confidential Analytical Query Processing. In *DaMoN 2024*.

[9] M. El-Hindi, T. Ziegler, M. Heinrich, A. Lutsch, Z. Zhao, and C. Binnig. 2022. Benchmarking the second generation of Intel SGX hardware. In *DaMoN 2022*.

[10] S. Eskandarian and M. Zaharia. 2017. An Oblivious General-Purpose SQL Database for the Cloud. *CoRR* abs/1710.00458 (2017).

[11] G. Giannikis, G. Alonso, and D. Kossmann. 2012. SharedDB: Killing One Thousand Queries With One Stone. *PVLDB* 5, 6 (2012), 526–537.

[12] G. Liao, Y. Liu, J. Chen, and D. Abadi. 2024. Bullion: A Column Store for Machine Learning. *arXiv:2404.08901* (2024).

[13] A. Lutsch, M. El-Hindi, M. Heinrich, D. Ritter, Z. István, and C. Binnig. 2024. Benchmarking Analytical Query Processing in Intel SGXv2. In *DaMoN 2024*.

[14] R. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. 2021. CryptDB: Protecting confidentiality with encrypted query processing. In *USENIX SOSP 2021*.

[15] L. Qiu, R. Taft, A. Shraer, and G. Kollios. 2024. The Price of Privacy: A Performance Study of Confidential Virtual Machines for Database Systems. In *DaMoN 2024*.

[16] M. Raasveldt and H. Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *ACM SIGMOD 2019*.

[17] L. Stephen S. Tu, F. Kaashoek, S. Madden, and N. Zeldovich. 2013. Processing analytical queries over encrypted data. *PVLDB* 6, 5 (2013), 289–300.

[18] X. Zeng, Y. Hui, J. Shen, A. Pavlo, W. McKinney, and H. Zhang. 2023. An Empirical Evaluation of Columnar Storage Formats. *PVLDB* 17, 2 (2023), 148–161.

[19] W. Zheng, A. Dave, J. Beekman, R. Popa, J. Gonzalez, and I. Stoica. 2017. Opaque: An oblivious and encrypted distributed analytics platform. In *USENIX NSDI 2017*.