Bloom Filter Based Encrypted Data Skipping In DuckDB Iceberg

Author: Thijs van der Heijden (2824561)
Vrije Universiteit × Universiteit van Amsterdam

1st supervisor:	Peter Boncz
daily supervisor:	Lotte Felius
2nd reader:	Dandan Yuan

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

August 8, 2025

"Here's to the crazy ones."
Steve Jobs

Abstract

In this thesis, we introduce the notion of Encrypted Data Skipping (EDS), as well as a novel Bloom filter based EDS scheme (BF-EDS). Our novel scheme encrypts just the metadata in such a way that range predicates can still be evaluated, allowing for efficient data skipping while leaking significantly less information compared to existing schemes like Order Preserving Encryption (OPE) and Order Revealing Encryption (ORE). To the best of our knowledge, no research has been done on encrypting just metadata to allow for predicate evaluation, while keeping the data itself encrypted using fast AES encryption schemes. Additionally, we add mapping functions to support signed, unsigned, NULL and string values, increasing the usefulness of BF-EDS.

We integrate BF-EDS in DuckDB Iceberg and perform extensive evaluations using both custom and TPC-H benchmarks, as well as a comparison with an ORE based EDS implementation. Our evaluation shows BF-EDS incurs an overhead between 1.15 and 1.4x when querying compared to plaintext, while significantly outperforming ORE by 1.8x. Additionally, ciphertexts are generated 135x faster compared to ORE, making BF-EDS better suited for write-heavy applications.

Acknowledgements

This thesis was completed at the Centrum Wiskunde & Informatica (CWI) in Amsterdam. I would like to give a special thanks to Peter Boncz for offering me a place in the Database Architectures group and for his feedback and support throughout this thesis. Thank you Dandan Yuan and Lotte Felius for providing me with valuable insights and feedback during our weekly meetings and for your enthusiasm for the work done in this thesis. I am grateful for all the wonderful people I met as part of the DA group, the MotherDuckers who opened their doors to us and the many, many games of table tennis that were played.

Most of all, I want to thank my partner, Charlotte, for her unwavering support and belief in me. Your support means the world to me.

Contents

List of Figures

List of Tables

List of Listings

Glossary

BF-EDS Bloom Filter based Encrypted Data Skipping.

CHF Cryptographic Hash Function.

DBMS Database Management System.

DE Deterministic Encryption.

EBF Encrypted Bloom Filter.

EDBMS Encrypted Database Management System.

EDS Encrypted Data Skipping.

FHE Fully Homomorphic Encryption.

FPGA Field Programmable Gate Array.

FPR False Positive Rate.

HE Homomorphic Encryption.

HF Hash Function.

MCS Minimum Coverage Set.

OLAP Online Analytical Processing.

OLTP Online Transactional Processing.

OPE Order Preserving Encryption.

ORAM Oblivious RAM.

ORE Order Revealing Encryption.

OTF Open Table Format.

PHE Partially Homomorphic Encryption.

PPE Property Preserving Encryption.

PRF Pseudorandom Function.

RE Random Encryption.

SBBF Split Block Bloom Filter.

SIMD Single Instruction Multiple Data.

STE Structured Encryption.

TCB Trusted Computing Base.

TEE Trusted Execution Environment.

TH Trusted Hardware.

UDF User Defined Function.

UDT User Defined Type.

Chapter 1

Introduction

Data privacy and security are fundamental rights, however, upholding these rights remains challenging. In this era of cloud computing, storage of sensitive data is frequently outsourced to third parties like Amazon, Google and Microsoft. Storing our data with these cloud providers requires a level of trust which we may not always be comfortable extending.

Consider, for instance, a hospital which stores its patient data in the cloud. To comply with data privacy regulations like GDPR this data must be encrypted. Ideally, hospitals would like to perform queries and share this data with other institutions, while keeping the data encrypted when at rest and in transit. To efficiently access these large amounts of data, data systems often use metadata containing statistics like row counts and min/max values to skip irrelevant data. When encrypting sensitive data this metadata should also be encrypted to prevent information leakage.

Despite the sensitivity of metadata, encryption support in current data lakehouse systems is limited. Delta Lake, for example, lacks native encryption capabilities and relies entirely on underlying storage and compute systems for encryption. Apache Iceberg supports basic table and metadata encryption using AES-GCM. Most existing solutions require metadata decryption to take place on the untrusted server. This is problematic, as it exposes both decrypted metadata and decryption keys to the untrusted server during query execution. A more secure alternative is to send all encrypted metadata to a trusted client for decryption. This approach, however, introduces significant I/O and decryption overheads, rendering it infeasible at the data lakehouse scale.

We argue that both data and metadata should be encrypted at all times on the server to ensure strong security guarantees. At the same time, we aim to enable efficient access to encrypted data on the server. Specifically, the system

must be able to prune partitions and skip irrelevant data. To achieve this we introduce the notion of Encrypted Data Skipping (EDS). With EDS we encrypt the metadata in such a way that it is still possible to evaluate a predicate over it.

EDS allows for fast access to data while keeping data and metadata encrypted on the server at all times. By skipping irrelevant data and returning only the relevant subset of data, EDS reduces both I/O and decryption overheads. Data files (e.g. in Parquet or ORC files) should be encrypted using fast symmetric encryption schemes such as AES-CTR or AES-GCM. This allows for easier pruning of data, thanks to columnar encryption, as well as efficient compression before encryption, reducing storage overhead.

Existing Property Preserving Encryption (PPE) schemes like Deterministic Encryption (DE), Order Preserving Encryption (OPE), Order Revealing Encryption (ORE) and Homomorphic Encryption (HE) allow various predicates to be evaluated over encrypted data. However, systems like CryptDB[55] and MONOMI[68] show that using these schemes is non-trivial. These schemes are not practical for the encryption of metadata (DE), leak too much information (DE, OPE, ORE) or are simply too slow (HE). To the best of our knowledge, no system has encrypted solely the metadata in such a way to allow for predicate evaluation.

In this thesis we introduce the notion of EDS, along with a novel Bloom filter based EDS scheme called BF-EDS. We extensively evaluate this scheme comparing it with both plaintext and ORE based systems. Implementing our scheme in DuckDB Iceberg allows us to evaluate the practicality of our scheme and its real-world data skipping performance. This leads us to the following research questions:

1. **How can Bloom filters be used to construct a secure and efficient EDS scheme?** This question consists of the following sub-questions.
   (a) How can we minimize the storage overhead while optimizing the performance of the scheme at the algorithmic level?
   (b) How can the performance of the Bloom filters be optimized while maintaining security?
   (c) Can we extend the BF-EDS scheme to support negative, NULL and string values?
   (d) How does the performance of BF-EDS compare to EDS using an existing scheme like ORE?

2. **How can we integrate BF-EDS in an existing data lakehouse system, such as DuckDB Iceberg?**

1.1 Contributions

Our main contributions are as follows:

- **Literature Study:** We perform a literature study analyzing and comparing existing encryption and query techniques used in Encrypted Database Management Systems (EDBMSs) to determine the practicality of our scheme.
- **Bloom Filter Based Encrypted Data Skipping (BF-EDS):** We introduce, implement and evaluate a novel Bloom filter based metadata encryption scheme which allows for EDS using range predicate evaluation.
- **BF-EDS in DuckDB Iceberg:** We integrate BF-EDS in the DuckDB Iceberg extension as well as implementing custom Iceberg manifest file generation code to inject our schemes metadata into Iceberg metadata files.
- **Comprehensive Performance Evaluation:** We perform a comprehensive performance evaluation at both the algorithmic and system levels to construct the best-performing BF-EDS scheme.

1.2 Outline

First, we provide the relevant background information for this thesis in Chapter 2. Section 2.1 introduces a number of cryptographic primitives relevant to the scheme, as well as a number of existing encryption schemes and Hash Functions (HFs). Binary interval trees, which are fundamental to the BF-EDS scheme, are introduced in Section 2.2. We then introduce Bloom filters in Section 2.3. These are used in conjunction with binary interval trees to enable BF-EDS. We conclude the background chapter with Section 2.4, which briefly covers relevant data storage systems, focussing on data lakehouse systems. Apache Iceberg is discussed in more detail, being the system we ultimately use to implement and evaluate BF-EDS.

Following the presentation of relevant background information, Chapter 3 provides an overview of existing techniques and EDBMSs through an extensive literature study. This chapter covers the three main EDBMS categories: software-only, trusted-hardware based and hybrid systems. A comparison is

performed between systems in the three categories based on their functionality (Section 3.4.1), security (Section 3.4.2) and their performance (Section 3.4.3).

Chapter 4 introduces Bloom Filter based Encrypted Data Skipping (BF-EDS), starting with a brief summary of the problem it aims to solve, as well as the threat model it assumes (Section 4.1). The remainder of the chapter introduces a number of novel algorithms which combine binary interval trees and Bloom filters to form the complete BF-EDS scheme. This chapter concludes with a brief security analysis (Section 4.3.3), which discusses the general and worst-case security of our scheme.

In Chapter 5 we discuss the implementation of BF-EDS. The implementation of various existing, as well as novel, binary interval tree algorithms is covered in Section 5.1. A generic and easily extendable HF interface implementation is given in Section 5.2. Three Bloom filter variants are implemented in Section 5.3. The methods to create Iceberg tables for testing and evaluation, inject BF-EDS metadata into existing Iceberg manifest files and the integration of BF-EDS in the DuckDB Iceberg extension, involving conversion of DuckDB queries to our custom range predicates and loading Bloom filters from Iceberg manifest entries, are given in Section 5.5. Section 5.6 concludes the implementation chapter discussing how signed integers and NULL values are mapped to the unsigned integer domain, as well as introducing a novel method to map strings to the numerical domain in a way which maintains their lexicographical ordering (Section 5.6.2).

Chapter 6 contains an extensive evaluation at both the algorithmic and system levels. The performance of our binary interval tree algorithms (Section 6.2), various HFs (Section 6.3) and our three Bloom filter implementations (Section 6.4) is evaluated using comprehensive microbenchmarks. Additionally, we evaluate the hashing output uniformity of a number of HFs to determine their level of security when used in BF-EDS (Section 6.3.2). The accuracy of our novel string mapping approach is evaluated on a real-world dataset in Section 6.5. In Section 6.6 a number of system-level evaluations are performed. The impact of increasing the query range size (Section 6.6.1), Bloom filter bitset size (Section 6.6.2) and manifest file batch size (Section 6.6.3) is evaluated. Additionally, we comparatively evaluate the performance of our BF-EDS DuckDB Iceberg implementation with plaintext DuckDB Iceberg on both our own datasets (Section 6.6.4) as well as TPC-H with various scale factors (Section 6.6.5). We conclude Chapter 6 with a comparison between our BF-EDS DuckDB Iceberg implementation and an ORE based EDS implementation. We compare with ORE as it is a commonly used PPE scheme and this evaluation demonstrates the practicality of our scheme. Both querying performance and ciphertext generation are

evaluated and compared (Section 6.7).

Finally, we list a number of loose ends and potential future work in Chapter 7, before providing a final conclusion to this thesis in Chapter 8.

Chapter 2

Background

This chapter introduces background information required to understand the full BF-EDS scheme. This includes a number of cryptographic primitives, binary interval trees, Bloom filters and data lakehouse systems.

2.1 Cryptographic Primitives

This section introduces a number of foundational cryptographic concepts and primitives relevant to this thesis. A number of encryption schemes, including both PPE and non-PPE schemes, are described. Table 2.1 lists all schemes and their associated leakage. Subsequently, Pseudorandom Function (PRF), HF and Cryptographic Hash Function (CHF) definitions are presented, followed by a number of HFs which can be used in BF-EDS.

2.1.1 Security Parameter

In cryptography the security parameter, typically denoted with λ, is a variable which defines the computational security of a scheme, consequently quantifying the desired level of security. It determines the key length and input sizes. The resources required by the algorithm and the difficulty, as well as probability, for an adversary to break it are considered functions of λ. A higher security parameter typically implies a stronger security at the cost of more computational resources. In this thesis we set the $\lambda = 128$, which is a commonly accepted standard for practical security[69].

2.1.2 Property Preserving Encryption

PPE is a form of encryption where specific plaintext properties are deliberately preserved in the ciphertext. Public tests can be performed to check whether a property predicate holds for a ciphertext. Systems like CryptDB [55] utilize PPE schemes to enable efficient query processing over encrypted data.

Deterministic Encryption

DE[9] is an encryption scheme which deterministically encrypts values. Two equal plaintext values encrypt to the same ciphertext. Equality operations can be performed using DE, and it is commonly used for SQL equality comparisons (`=`), equality-based `JOIN` and `GROUP BY` operations , `DISTINCT` clauses, as well as set operations like `INTERSECT` and `EXCEPT`. DE leaks equality of values through equality of ciphertexts. Commonly DE is achieved through AES encryption without IV's to deterministically encrypt values[55]. In [48] it is shown that through frequency analysis DE can be broken, and an attacker can infer plaintext values from their respective DE ciphertexts.

Order Preserving Encryption

OPE[1][12] is an encryption scheme which allows for ordering of plaintext values through their ciphertexts. Originally proposed in [1], OPE encrypts numerical values in such a way that their ciphertext counterparts maintain plaintext ordering. This allows for efficient numerical comparisons as ciphertexts can be compared as regular numerical values. This simplifies usage of OPE as it requires little change to existing Database Management Systems (DBMSs). While practical, OPE leaks a lot of information. In [13] it is shown that a single OPE ciphertext leaks half of the most significant bits of its underlying plaintext value. The notion of "best-possible" security is introduced in [52]. At this level of security the OPE ciphertexts leak no additional info besides the ordering of data. To achieve this they show that the size of ciphertexts must grow exponentially relative to the length of plaintext values.

Order Revealing Encryption

ORE[61][17][39] is a generalization of OPE. Contrary to OPE, ORE does not place any restrictions on the structure of the ciphertext space. An ORE scheme simply requires there to be a function which compares two ciphertext values to determine the ordering of their plaintext counterparts. This prevents leakage of the total ordering within a column. More recent ORE schemes exist which

increase security and reduce leakage further. Most recently in [39] a new block-based ORE scheme is introduced which is robust against the attacks presented in [48]. A number of SQL operations can be performed using OPE and ORE ciphertexts including range comparisons (`<, >, <=, >=`), `BETWEEN`, `ORDER BY` and `LIMIT`[66].

Although PPE schemes do not explicitly reveal plaintext information, their exposed properties (i.e. equality and ordering) have been shown to be vulnerable to inference attacks[48][22][31]. These attacks can lead to significant leakage and, in some scenarios, even plaintext recovery.

2.1.3 Homomorphic Encryption

Fully Homomorphic Encryption

Fully Homomorphic Encryption (FHE)[27] is an encryption scheme which allows arbitrary operations to be performed directly over ciphertexts. Given ciphertexts $(c_1, c_2 \dots c_n)$ that encrypt messages $(m_1, m_2 \dots m_n)$, an FHE scheme allows the generation of a ciphertext c_t which encrypts $f(m_1, m_2 \dots m_n)$ for any efficiently computable function f. Data confidentiality is preserved throughout computations. The main drawback to using FHE is its performance cost, which prohibits its usage in many systems. It is estimated that FHE runs between 50,000-1,000,000x slower than comparable plaintext operations[64][37].

Partially Homomorphic Encryption

Partially Homomorphic Encryption (PHE) is an alternative to FHE which supports just a single operation (e.g. addition or multiplication) over encrypted data. In some cases this operation can only be performed a fixed number of times. Contrary to FHE schemes, PHE schemes offer a much more manageable overhead. Using multiple different PHE schemes side-by-side offers functionality and security similar to FHE at significantly lower costs. Examples of PHE schemes are Paillier[53] for addition and Elgamal[23] for multiplication. These schemes are asymmetric, requiring them to have large ciphertext spaces leading to large ciphertext size overheads. In [59] more efficient symmetric PHE schemes are presented for addition and multiplication, which outperform existing asymmetric PHE schemes by orders of magnitude.

2.1.4 Pseudorandom Functions

A PRF is a deterministic function $F : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^l$ where the first input is a secret key $k \in \{0,1\}^\lambda$ with length λ, and the second input is a

PPE	Leakage
DE	Equality of values and vulnerability to inference attacks
OPE	Ordering of values and vulnerability to inference attacks
ORE	First ciphertext bit or block that differs
PHE	Operations performed
FHE	Operations performed

Table 2.1: Covered encryption schemes and their corresponding information leakage.

message with an arbitrary length. The function outputs a fixed-length binary string of length l. We write $F_k(x)$ to denote the function F keyed with k on the input x. A function F is considered pseudorandom if, for a randomly chosen key k, the function F_k is indistinguishable from a function chosen uniformly at random from the set of all functions with the same domain and range.

Hash Functions

A HF is a deterministic function which maps an input of arbitrary length to a fixed-length output. In this thesis, keyed HFs are used as approximated PRFs. We assume that each HF acts as an ideal random function, meaning each input hashes to a uniformly distributed and independent output. A keyed HF combines a secret key and input message to generate a fixed-length output. This protects against dictionary attacks in cases where the input domain is small and known to the adversary. During a dictionary attack an adversary hashes all values within the domain using the same HF to obtain all digests. These digests can be inserted into a Bloom filter to compare which bits are set, allowing an adversary to reverse-engineer inserted values. Without knowledge of the secret key it becomes computationally infeasible for an adversary to reverse-engineer hashed values. A CHF is a HF which satisfies additional security properties. A good CHF satisfies four key properties:

1. **Uniformity:** Digests produced by a HF $\mathrm{H}(x)$ should be uniformly distributed and look random. This ensures the hash digest leaks no information about the plaintext input.

2. **Determinism:** For a given input s a HF should always produce the same digest $\mathrm{H}(s) = h$.

3. **Irreversibility:** Given a digest h it should be infeasible to invert the hashing process and obtain the input s.

4. **Approximate Injectivity:** Tiny changes in the input should result in wildly different digests (snowball effect). This prevents leakage of input characteristics.

The security of a CHF is assessed based on a number of properties. Each property in this list implies the property following it:

1. **Collision Resistance:** It should be infeasible for an attacker to find two strings s_1 and s_2 such that, for a CHF $\mathrm{H}(x)$, $\mathrm{H}(s_1) = \mathrm{H}(s_2)$. In other words, it should be infeasible for an attacker to find two different inputs which hash to the same digest.

2. **Second Pre-image Resistance:** Given an arbitrary input s_1 it should be infeasible for an attacker to find a second string s_2 for which $\mathrm{H}(s_1) = \mathrm{H}(s_2)$.

3. **Pre-image Resistance:** Given a digest h of length n produced by CHF $\mathrm{H}(x)$, it should be infeasible for an attacker to find the string s for which $\mathrm{H}(s) = h$. In other words, given a digest it should be infeasible for an attacker to find the input which produced it. A party wishing to conduct a pre-image attack has no other option than brute forcing, which has a time complexity of 2^n.

The security of a CHF is strongly influenced by its digest length n. A longer hash digest exponentially slows down brute force attacks thanks to the brute force time complexity of 2^n[19][69]. In the remainder of this section a number of cryptographic and non-cryptographic HFs are presented. An evaluation of these functions is given in Section 6.3.

SHA256

SHA256[47] (Secure Hash Algorithm 256-bit) is a CHF, part of the SHA-2 family, designed by the NSA and published by NIST in 2001. It produces 256 bit hashes from arbitrary length inputs. SHA256 is widely used, for instance in blockchains, TLS and code signing, thanks to its balance of speed, simplicity and strong security. Due to its widespread use many CPUs include hardware instructions designed to accelerate SHA256 operations. As of writing, SHA256 is considered safe, with no known practical collision or pre-image attacks. SHA256 processes inputs in 512 bit blocks (padding the input if required) using a 64 round compression function. This function uses bitwise operations, modular additions and constants derived from the first eight prime numbers. A fixed set of initial hash values is updated across each block over 64 iterations resulting in a 256 bit digest.

Blake3

Blake3[50] is a CHF which is both faster and, in some aspects, more secure than SHA-2 and SHA-3 algorithms. It uses a Merkle tree structure, splitting inputs into 1KiB individually processable chunks, making it highly parallelizable across any number of threads and Single Instruction Multiple Data (SIMD) lanes. In [54], a performance comparison between SHA256 and Blake3 is performed, showing that Blake3 consistently outperforms SHA256 across various platforms and input sizes. Due to the underlying Merkle tree structure and 1KiB chunk sizes Blake3 struggles with smaller inputs, and in some cases is outperformed by SHA256. Blake3 extensively uses fast SIMD XOR operations to compute its 256 bit digest.

HighwayHash

HighwayHash[2] is a keyed, SIMD-optimized, HF. Unlike SHA256 and Blake3, HighwayHash is not a CHF, providing no formal collision-resistance guarantees. Similar to Blake3, HighwayHash performance takes a hit on small input sizes (< 64 bytes) due to internal initialization and finalization costs. Benchmarks show that for small inputs (8-64 bytes) hashing can take 7-8 CPU cycles per byte, compared to 0.94-0.24 cycles per byte for longer inputs (64-1024 bytes). HighwayHash uses SIMD-accelerated multiply-permute operations (multiplication followed by reordering of elements) on 64 byte blocks, combining these blocks into a digest of either 64 or 256 bits.

Murmur3

Murmur3[6] is a fast, non-cryptographic HF designed for Bloom filters, hash tables and general purpose hashing. It offers no resistance against collision or pre-image attacks, making it unsuitable for cryptographic purposes. Murmur3 offers excellent throughput, high avalanche properties (approximate injectivity) and low collision rates. Murmur3 processes inputs in fixed-size chunks using fast integer multiplication, rotation and XOR operations. A final mixing step is performed to diffuse bits and increase entropy, producing either a 32 or 128 bit digest.

2.2 Binary Interval Trees

A binary interval tree [62][42] is a binary tree which can represent ranges in the domain $[0, T]$, where T is the largest possible range value the tree can represent. Sets of nodes in a binary interval tree can be used to compute whether two

ranges intersect. Figure 2.1 shows an example binary interval tree which can represent ranges in the domain $[0, 7]$.

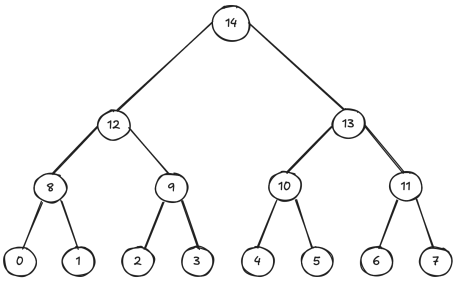
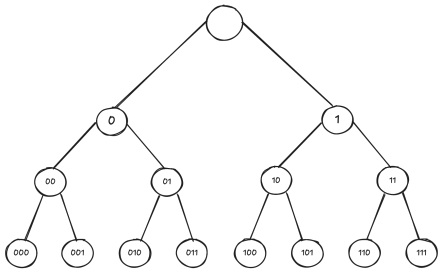
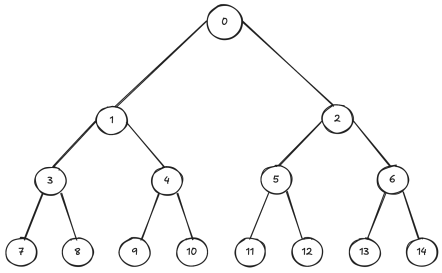


Figure 2.1: Example binary interval tree which can depict ranges in the inclusive domain $[0, 7]$ and has a height of $log_2(7) \approx 3$.

This tree is a simplified example. Constructing a binary interval tree for a domain $[0, T]$ requires the following steps. First, the root node is defined using an empty binary string b. Second, the left and right children of the root node are labeled $b0$ and $b1$ respectively. This process is repeated up to tree depth $log_2(T)$. Figure 2.2a shows the final tree with binary labels. Each leaf node has a binary label which corresponds with its decimal value in Figure 2.1.



(a) Binary interval tree shown in Figure 2.1 annotated with its binary labels. A nodes left and right child inherit the parents binary label appended with a 0 or 1 respectively.



(b) Binary interval tree shown in Figure 2.2a annotated with IDs calculated using the equation in Listing 2.3.

Figure 2.2: Binary interval trees with binary labels and IDs calculated using the equation in Listing 2.3.

Being able to distinguish nodes in a binary interval tree is important as determining range intersections requires performing set intersections. Using binary labels does not work as some nodes have identical binary labels, with the

exception of leading zeroes. To this end every node in the tree has a unique ID. These ID's are computed using Equation 2.3, in which $b[i]$ is the bit at index i - starting from the least significant bit - in the nodes binary label. Equation 2.4 shows ID calculation examples for the nodes with binary labels 110 and 01. Figure 2.2b shows a binary interval tree annotated with IDs based on the binary labels shown in Figure 2.2a.

$$\sum_{i=0}^{|b|-1} (2^i + 2^i \cdot b[i])$$

Figure 2.3: Equation which calculates a unique ID for each node in a binary interval tree. The term $b[i]$ is the bit at index i, from least to most significant bit (right to left), in the nodes binary label[42].

$$\begin{aligned}
&110 \\
&(2^0 + 2^0 \cdot 0) + (2^1 + 2^1 \cdot 1) + (2^2 + 2^2 \cdot 1) \\
&= 1 + 4 + 8 \\
&= 13
\end{aligned}$$

$$\begin{aligned}
&01 \\
&(2^0 + 2^0 \cdot 1) + (2^1 + 2^1 \cdot 0) \\
&= 2 + 2 \\
&= 4
\end{aligned}$$

Figure 2.4: Example calculation of node ID for the nodes with binary labels 110 and 01. Index 0 is the least significant, rightmost bit in the nodes binary label.

2.2.1 Node Coverage And Coverage Set

The *coverage*, cv, of a node is defined as follows:

- If the node is a leaf node, $cv(id) = id$
- If the node is not a leaf node, denote the ID's of its two child nodes as id_1 and id_2, its coverage is $cv(id) = cv(id_1) \cup cv(id_2)$

In other words, the coverage of a node is the set of all nodes whose path to the root flows through this node. Figure 2.5 shows the coverage of node 12 which is equal to the set $\{8, 9, 0, 1, 2, 3\}$. Using this definition of *coverage*, we introduce a set called the *coverage set*. For a leaf node v the *coverage set* $\mathbb{P}(v)$ is defined as the set of all nodes i which cover v (i.e. $v \in cv(i)$). This set can be computed by starting at the leaf node v and traversing upwards toward the root. All nodes on this path, including the root node, are part of the coverage set for v. Figure 2.6 shows the coverage sets $\mathbb{P}(3)$ and $\mathbb{P}(7)$.

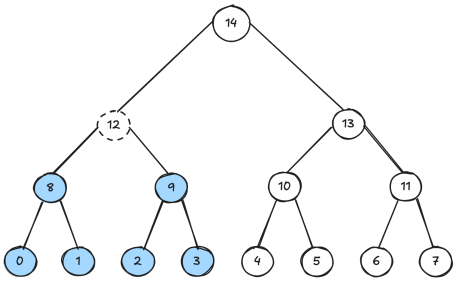


Figure 2.5: *Coverage*, *cv* of node 12 which is equal to $\{8, 9, 0, 1, 2, 3\}$.

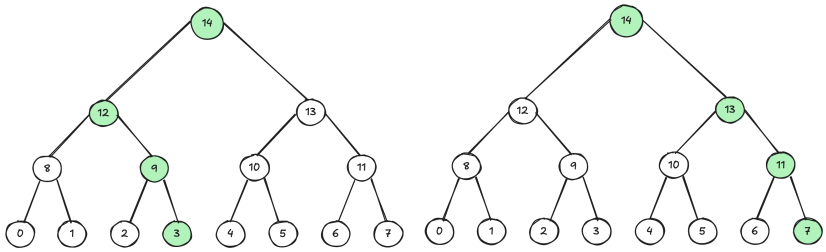


Figure 2.6: The coverage sets $\mathbb{P}(3)$ and $\mathbb{P}(7)$, highlighted in green.

2.2.2 Minimum Coverage Set

For an inclusive range $[s, t]$, $\Lambda(s, t)$ is defined to be the smallest subset of nodes that cover, exclusively, all leaves within the range $[s, t]$. In [42] this set is referred to as the Minimum Coverage Set (MCS). It's size, given $T \geq 4$, is proven to be at most $2 * log_2(T)$. Additionally, it is proven that if a value $v \in [s, t]$ then $\mathbb{P}(v) \cap \Lambda(s, t)$ results in one node[42][62]. The MCS for the range $[3, 7]$ is shown in Figure 2.7.

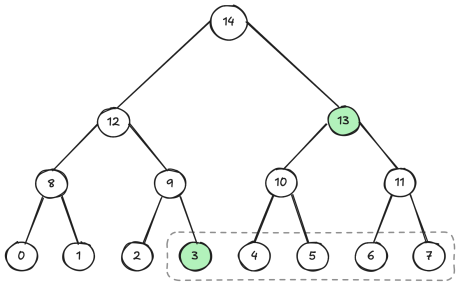


Figure 2.7: MCS $\Lambda(3,7) = \{3, 13\}$. Node 4 is not a member of the MCS as it also covers node 2, which is not part of the MCS range. Node 13 covers the entire range $[4, 7]$, and is preferred over $\{10, 11\}$ as the MCS is defined as the **smallest** subset of nodes.

2.2.3 Determining Range Intersections

When querying for data in the range $[min_q, max_q]$, a block containing data in the range $[min_d, max_d]$ should be returned if and only if $min_q \leq max_d$ and $max_q \geq min_d$. This is equivalent to testing whether $\Lambda(0, max_q) \cap \mathbb{P}(min_d) \neq \emptyset$ and $\Lambda(min_q, T) \cap \mathbb{P}(max_q) \neq \emptyset$. If both intersections intersect in one node, the ranges intersect[69].

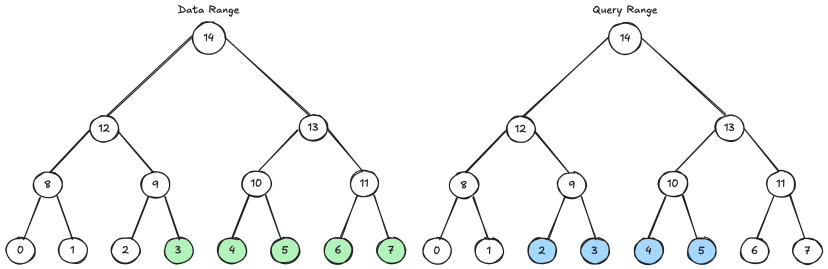


Figure 2.8: Example data and query ranges $[3, 7]$ and $[2, 5]$ respectively. These two ranges intersect.

Figure 2.8 shows an example data as well as query range. Figures 2.9 and 2.10 show the two set intersections. The first intersection results in $\{12\}$ and the second intersection results in $\{13\}$. As both intersections return exactly one node, the two ranges intersect.

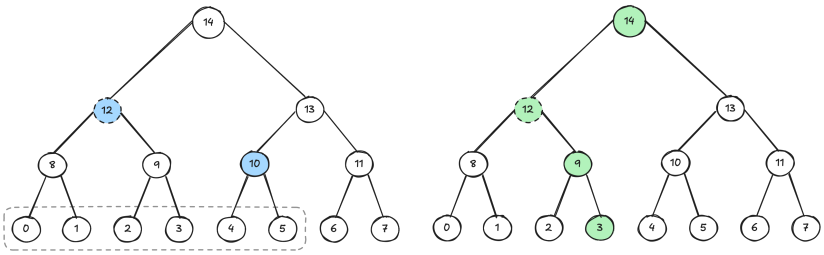


Figure 2.9: Intersection between the sets $\Lambda(0, max_q)$ and $\mathbb{P}(min_d)$ results in $\{12\}$.

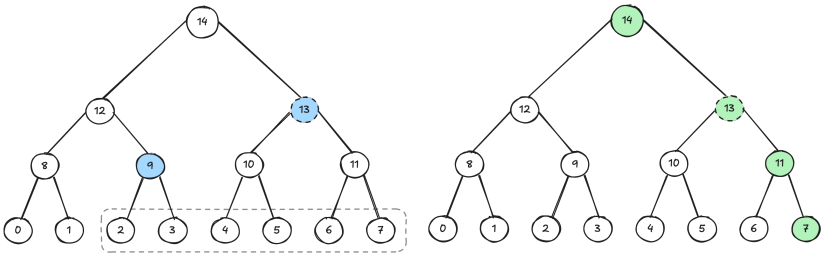


Figure 2.10: Intersection between the sets $\Lambda(min_q, T)$ and $\mathbb{P}(max_d)$ results in $\{13\}$.

2.3 Bloom Filters

A Bloom filter is a space efficient, probabilistic data structure, which can answers set-membership queries. Items can be inserted into a Bloom filter, similar to how items might be added to a set. Consequent set membership queries can return false positives, but never false negatives. In other words, if an item has been inserted, the set membership query will **never** return false. Bloom filters use k HFs (k is not the same as the key in H_k), being different HFs or the same HF using k different seeds, and a bitset of length m. The False Positive Rate (FPR) denotes how often false positives are returned. Adjusting m and k changes the FPR of the Bloom filter.

Inserting an item into a Bloom filter is done in two steps. First, the item is hashed using the filters k HFs, resulting in k hash digests h. Second, the bits at indices $h \,\%\, m$ in the bitset are set. Due to hash collisions the same bit in the bitset might be set multiple times during separate insertions. Figure 2.11 shows a basic Bloom filter as well as insertion of two items.

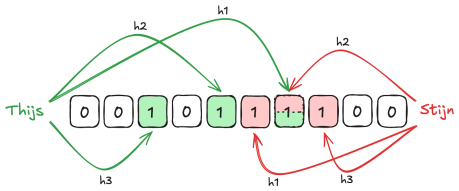
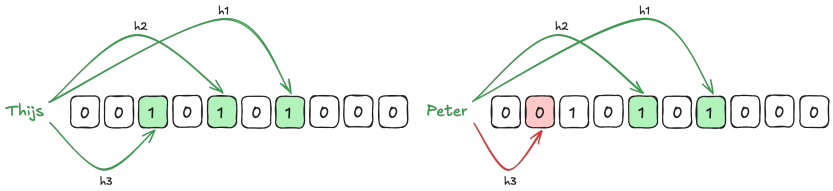


Figure 2.11: Inserting two items into a Bloom filter. Both insertions set the bit at index 6 due to a hash collision.

Checking if an item is in the Bloom filter follows the same steps as an insertion. First, the item is hashed using the same k HFs. Second, if all indices at $h \,\%\, m$ in the bitset are set, the item *might* be in the Bloom filter. If at least one of the indices is not set, the item is definitely not in the Bloom filter[11]. Figure 2.12 shows Bloom filter membership checks with Figure 2.12a showing the check of an item which is in the Bloom filter and Figure 2.12b showing the checking process for an item which is not in the Bloom filter.



(a) Bloom filter check for an item which is in the Bloom filter.

(b) Bloom filter check for an item which is not in the Bloom filter.

Figure 2.12: Checking whether two items are in the Bloom filter. One item is, the other item is not.

2.3.1 Register Blocked Bloom Filters

Insert and check operations on a basic Bloom filter are quite inefficient. Even with a small Bloom filter bitset, which might fit entirely into cache, both operations require at most k random accesses. A blocked Bloom filter offers increased locality by performing all insertions and checks for a single item within a cache-line sized block. This increases locality and reduces the number of potential cache misses down to one. The first of k hash digests is used to select a block. The remaining $k-1$ hash digests set bits within this block. A downside of this approach is a significant increase in FPR. Each block acts as a separate Bloom filter with a very small bitset size m/blocks. Setting $k-1$ bits in such a small bitset, collisions are likely to occur. Increasing the total bitset size compensates

for this as fewer block collisions will occur.

A register blocked Bloom filter is a specialized variant of the general blocked Bloom filter, where each block is machine word sized (e.g. 64 bits on a 64 bit machine). Two main benefits stem from this. First, all operations now happen within a register, increasing throughput. Second, SIMD operations, as well as vectorization, can be used to increase filter throughput even further[44]. Figure 2.13 shows the register Blocked Bloom filter insertion process.

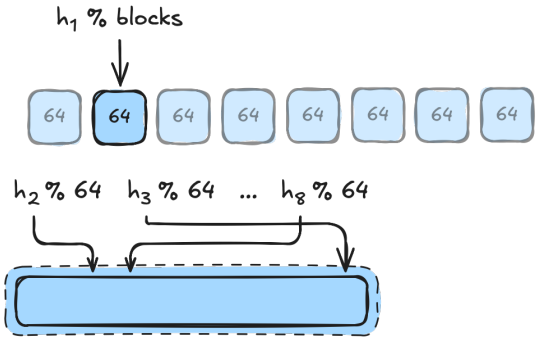


Figure 2.13: Insertion of an item into a register blocked Bloom filter. The first hash is used to select a block, the remaining $k-1$ hashes are used to set bits within this block. Each block is the size of a machine word, in this case 64 bits.

2.3.2 Split Block Bloom Filters

Split Block Bloom Filters (SBBFs) are similar to the abovementioned blocked Bloom filters, differing in the way bits are set within blocks. Instead of setting $k-1$ random bits within a block, the block is split up into eight disjoint contiguous sections, with one bit in each section being set. This approach leads to a significantly lower FPR when compared to blocked Bloom filters[16]. The Parquet file format internally uses the SBBF described in [5], which uses 256 bit blocks and eight HFs. Each block fits nicely within one CPU cache line and the eight HFs fit cleanly into SIMD lanes.

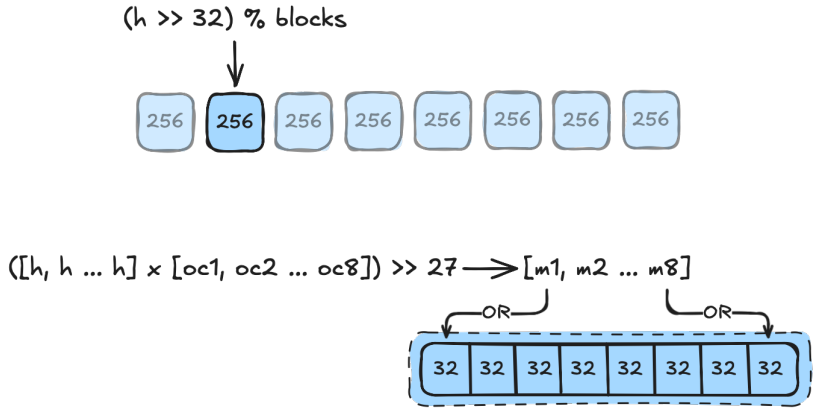


Figure 2.14: Insertion of an item into a SBBF. The most significant 32 bits are used to select a block. Multiply-shift hashing is used to obtain eight distinct indices. Bits at these eight indices are set in the selected block.

During item insertion the to-be-inserted item is hashed once and the 32 most significant bits of this hash are used to select a specific block. the least significant 32 bits are broadcast to eight lanes and subsequently multiplied with eight distinct odd constants. This results in eight distinct hashes. These eight hashes are then shifted to the right by 27 resulting in eight indices within the range $[0, 31]$. This process is called multiply-shift hashing. Each of these indices is used to set a bit in a different 32 bit sub-block. This is done by constructing a mask using these eight indices, which is then inserted into the existing block using a boolean OR operation. All bits set in the mask are copied into the block. All steps in this process can efficiently be performed using SIMD instructions. Figure 2.14 shows the insertion steps described above. Checking for an item only differs from insertion in the last step. Instead of using an OR operation an AND operation is used to check whether all bits in the mask are also set in the block. If the result of this AND operation is equal to the original mask, all bits in the mask were set in the block. This indicates the item is in the Bloom filter[5].

2.4 Data Lakehouse

This section briefly goes over the evolution of data storage systems, followed by an introduction to data lakehouse systems focussing on Apache Iceberg.

2.4.1 Data Lakes

Relational databases optimized for Online Transactional Processing (OLTP) have long been, and still are, the de-facto way businesses store their transactional data. OLTP databases excel in write-heavy applications thanks to their row-based storage. Increasing demand for Online Analytical Processing (OLAP) workloads lead to the development of analytical systems like DuckDB[56] and Clickhouse[18]. These systems often make use of columnar storage and vectorized execution[14]. In OLAP workloads organizations often aggregate all their data into a single system for centralized analytical processing. These collection systems are often called data warehouses[41][41].

Data lakes can store large amounts of structured, semi-structured and raw data. Storing these different kinds of data allows for future processing of data into various different formats. This allows the same data to be used for many different usecases[67]. Columnar file formats like Parquet and ORC are the standard in data lakes as data lake usecases are generally read-heavy, for which columnar storage lends itself well. Columns are split into blocks of rows called row-groups, for each of which metadata is tracked and stored. Data skipping can be used to effectively skip row-groups or entire columns, based on this metadata, if query tuples correlate well with column ordering. If this is not the case, data can be partitioned to allow for effective partition pruning[26][15].

2.4.2 Open Table Formats

Missing features like schema enforcement and lack of ACID transactions force organizations to employ both data warehouses and data lakes to make use of their combined features. Open Table Formats (OTFs) aim to provide a unified solution by adding a distinct metadata layer on top of existing data lake storage. This metadata layer adds traditional data warehouse features like ACID transactions, time travel and schema enforcement to data lakes. These systems are canonically called data lakehouse systems[51].

Delta Lake is Databricks OTF. It uses a transaction log, compacted into Parquet files, to provide ACID transactions, time travel and significantly faster metadata operations using per-file statistics. The transaction log contains object metadata like value ranges (min/max values) and supports Bloom filters and dictionaries for extensive data skipping. Delta Lake is seen as the first system to store object metadata directly in the object store, an approach which Apache Hudi and Iceberg later adopted[20].

Apache Hudi is an open-source OTF designed mainly for fast upserts and deletes, as well as incremental file updates. Hudi mainly optimizes for data

streaming ingestion and workloads where only data ingested over a certain period is required for analysis. By only processing new data Hudi increases query performance. A directory-based data management structure is used, where each table is stored in its own directory and data files belonging to this table are stored in nested sub-directories. Data is stored in Parquet format. Hudi divides tables into partitions based on column values, like for instance timestamps. Metadata files in the root table directory contain partition metadata which is used for partition pruning to optimize query performance[26].

Apache Iceberg is an OTF for large tables and OLAP workloads. Iceberg provides an extended SQL syntax to allow for tasks like merging new data, updating existing rows and performing targeted deletes. Schemas in Iceberg are evolvable, allowing columns to be renamed, reordered, added and deleted. Changing the schema does not require rewriting table data. Partitioning, the grouping of similar rows to improve query performance, is done automatically by Iceberg. Time travel and rollback, as well as data compaction using bin-packing or sorting, are all natively supported. Iceberg and Delta Lake are very similar, with their main differences being in the way they handle transactions (atomic snapshots versus transaction log), handle file updates (merge-on-read versus merge-on-write) and file format compatibility. Both formats are slowly converging, and tools like Databricks' Unity Catalog[28] allow simultaneous reading from both systems at once.

Iceberg is file format agnostic and supports the Parquet, ORC and Avro file formats[43][33]. Being file format agnostic is important for a number of reasons. First, it allows the data from multiple teams in an organization to be combined, even if these teams use different systems with different formats. Second, it provides flexibility for users to choose the format that best fits their needs. Last, it future-proofs Iceberg. Iceberg can add support for future file formats without requiring complete rewrites of existing tables[4].

A hierarchical metadata structure is used in Iceberg, allowing for extensive data skipping from the partition level down to individual files. Metadata is stored exclusively in Avro - a schema-based binary file format - files. Following is an explanation of Icebergs architecture, from bottom to top. Figure 2.15 shows the full Iceberg metadata architecture.

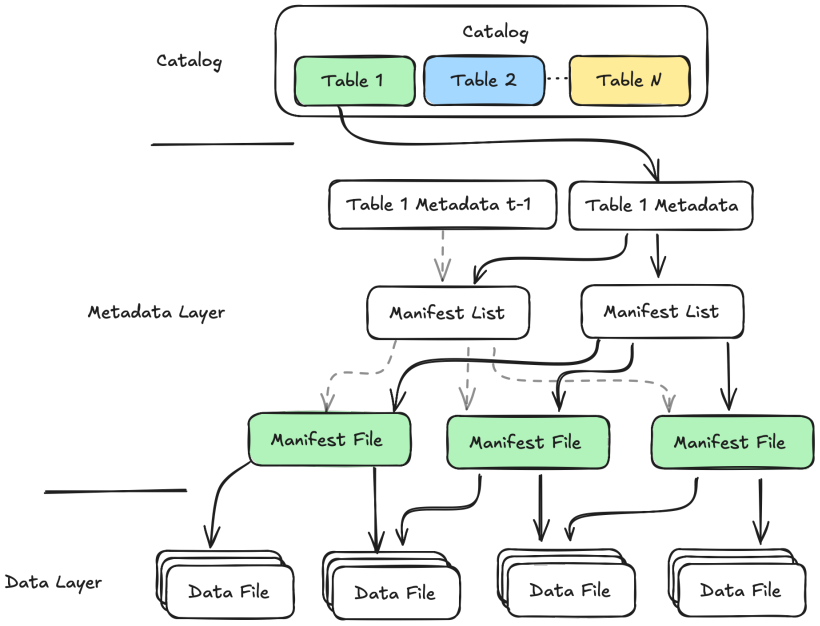


Figure 2.15: The architecture of Apache Iceberg. The solid black arrows indicate active links between files. Three main layers are shown: the data layer, metadata layer and the catalog.

The bottommost layer is the data layer. This layer contains all the actual data files which can be seen as the leaves in the Iceberg metadata tree. Almost every query interacts with a subset of these files (except for pure metadata queries). In real-world usage the data layer is backed by some form of distributed storage. This can be a distributed filesystem like HDFS (Hadoop Distributed File System) or object storage like Amazon S3, Google Cloud Storage or Azure Data Lake Storage. Using object storage enables data lakehouse systems to benefit from extremely scalable and low-cost storage.

The middle layer is the metadata layer. This layer consists of three types of metadata files. Each type tracks a subset of the data in the table. Manifest files are the lowest layer of metadata files. Each manifest file contains many manifest entries, each of which tracks an individual data file, as well as storing relevant file statistics. Statistics include the minimum and maximum values, row counts and partition membership. Some of these statistics are stored in the data files as well. Storing them in the manifest files prevents having to read the footer of the data files, reducing costly I/O operations. Figure 2.16 shows a simplified manifest file containing two manifest entries tracking Parquet files. Iceberg updates these statistics on writes. Other system, like Apache Hive,

compute statistics in long-running read jobs. As a result statistics are more up-to-date and reliable in Iceberg.

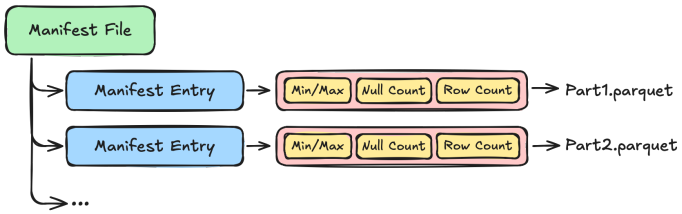


Figure 2.16: Iceberg manifest file containing two example manifest entries. Each entry points to a distinct data file and contains relevant file statistics like column bounds, NULL value counts and row counts.

A manifest list is a snapshot of an Iceberg table at a given point in time. It contains a list of manifest files as well as statistics for each of these files. These statistics can be used for data skipping at this level. Manifest lists are tracked by metadata files. Each metadata file denotes the state of an Iceberg table at a certain point in time. It contains information about the tables schema, partition information, snapshots and which snapshot is the current one. Each time an Iceberg table is changed, a new metadata file is created and registered atomically via the catalog. This atomic registration helps create a linear table history as well as resolve situations in which concurrent writes occur.

At the root of the Iceberg tree is the catalog. The catalog stores information on where to find the latest metadata file for each table. This metadata file pointer must be updated atomically. This prevents situations in which there are two 'most recent' metadata files. There are a number of different catalog backends available, each of which provides the atomicity guarantee but stores the current metadata file pointer differently[63].

2.4.3 Data Skipping In Data Lakehouse Systems

Data skipping in data lakehouse systems is an effective technique for improving query performance. Metadata is used to determine whether data is relevant for a query. Irrelevant data can be safely skipped. Two commonly used data skipping techniques are predicate pushdown and partition pruning. Data skipping can leverage various data structures, including zone maps, dictionaries and Bloom filters.

Predicate pushdown is a query optimization technique where data filtering is performed as early as possible. Filtering operations are pushed down into the data scan operator. By filtering at the scan level the amount of data that

enters the query pipeline is reduced significantly[15]. Partition pruning prunes entire irrelevant data partitions. This is often done in combination with predicate pushdown, and further reduces the amount of data pushed into the query pipeline. Partition pruning is only effective if data is partitioned well. Ordered temporal data lends itself well for partitioning. Non-temporal data can be partitioned as well, for instance by using clustering techniques[46].

Zone maps store the min and max values present in data files, allowing the query optimizer to skip files which do not contain values within the query range. Delta Lake, Apache Hudi and Apache Iceberg store table zone maps which are used during query planning [34][21][74].

Dictionaries can be used for data skipping on columns that store low cardinality data. By storing the values present in the column in a dictionary and replacing column values with dictionary indices, repeating column values take up far less space. When querying for a specific value, predicate pushdown can be used to check a files dictionary first. If the dictionary does not contain the value the file can be skipped (granted all column values are present in the dictionary).

In cases where columns contain high cardinality data, Bloom filters can be used to effectively skip data. All column values can be inserted into a Bloom filter. Consequent queries can query the Bloom filter to determine if a value is present in the column. If the value is not present in the Bloom filter the data can safely be skipped. Parquet natively supports Bloom filters and systems like DuckDB can transparently read and write them[45]. Delta Lake is the only OTF which supports creating Bloom filter indexes for columns, which are stored in the table metadata[10].

Chapter 3

Related Works

This chapter introduces, discusses and reviews related work in the EDBMS and query processing domain. This domain is split into three main categories, and for each of these relevant literature is discussed. The three main EDBMS categories are as follows:

- **Software-Only**: EDBMSs using solely software level cryptography to support queries over encrypted data. These systems make use of PPE schemes like DE, OPE, PHE and FHE to support various operations over encrypted data.
- **Trusted Hardware**: EDBMSs which use Trusted Hardware (TH), being either dedicated TH or a Trusted Execution Environment (TEE), to perform operations over decrypted data in a trusted environment.
- **Hybrid Query Execution**: EDBMSs which split up query processing in some way. Various implementations exist which split queries over various distinct components. In this study we will look at EDBMSs which split up queries over a software and TH component, as well as EDBMSs which use a split client/server execution model.

For each of these categories a number of relevant works are discussed. The structure of this related works section is as follows: Section 3.1 introduces a number of software-only EDBMSs, both using PPE and Structured Encryption (STE), including CryptDB[55] and KafeDB[71]. In Section 3.2 a number of EDBMSs utilizing TH are introduced. Both systems using dedicated TH, like for instance TrustedDB[8], and systems using TEEs, like for instance Azure Always Encrypted[3], are discussed. Hybrid EDBMSs, using either a software-trusted-hardware or client-server split execution model, are discussed in Section

3.3. These include MONOMI[68] and VCrypt[25]. Section 3.4 briefly compares discussed EDBMSs on their functionality, security and performance. There is a large amount of research in the EDBMS domain. However, to the best of our knowledge, no research has specifically focussed on the EDS problem.

3.1 Software-Only EDBMSs

This section discusses relevant research in the area of EDBMSs using a software-only approach.

CryptDB

CryptDB is one of the first software-only EDBMSs which supports queries over encrypted data. It claims to provide provable and practical privacy in face of a compromised database server or curious administrator. A SQL-aware encryption strategy is used. A number of PPE schemes are used to encrypt data in ways supporting various SQL operations. For example: DE is used for equality operations and OPE is used for ordering and range queries.

To prevent leakage of information by less secure schemes like DE and OPE, an onion model is used. Values are encrypted using various encryption schemes, with the least secure scheme on the inside, and more secure schemes going outwards. The outermost layer of the onion is encrypted using a scheme that leaks very little, like for instance Random Encryption (RE) or HE. The PPE schemes used within the onion are less secure but allow for predicates to be evaluated over them. During a query the Onion Key Manager (OKM) determines which layers of the onion need to be 'peeled' off to perform the requested SQL query. Figure 3.1 shows the CryptDB onion model.

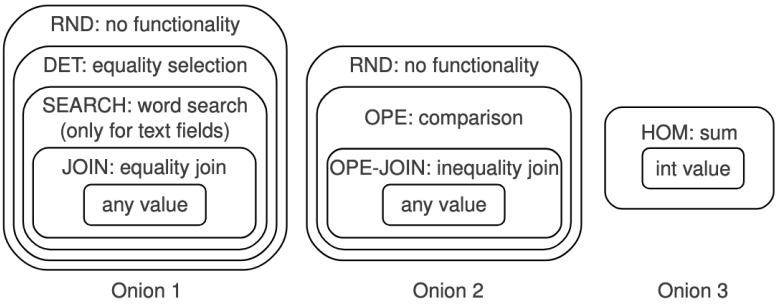


Figure 3.1: CryptDB onion model and the operations which can be performed using each of the onion layers[55].

Performing joins on encrypted columns requires columns to be encrypted using the same keys. To prevent the server from being able to join all columns, without a user requesting this, CryptDB introduces a new cryptographic primitive which allows the server to dynamically adjust the join encryption keys for each column. Initially all columns are encrypted using different join keys, disallowing all joins. When a user requests two columns should be joined, the server computes an onion key which can be used to re-encrypt the two columns to the same join key. Subsequently allowing for the join to happen. This novel cryptographic primitive is based on Elliptic-Curve-Cryptography (ECC).

CryptDB claims a mean throughput overhead of roughly 27% when compared with plaintext PostgreSQL. Evaluation is performed using the TPC-C benchmark focussing on an OLTP workload. Importantly, CryptDB does not change the inner workings of an existing DBMS, but instead relies on client-side query rewriting and User Defined Functions (UDFs). This approach makes CryptDB portable, which is demonstrated in a port to MySQL requiring just 86 lines of code to be changed[55].

Symmetria

Symmetria is a prototype software-only EDBMS using novel symmetric PHE schemes in combination with existing PPE schemes. These schemes are claimed to perform far better than state-of-the-art asymmetric PHE schemes used in earlier EDBMSs, like for instance CryptDB. While Symmetria offers split-execution, performing part of the query on the server and part on the client, it aims to, and can, perform most of queries entirely on the server. Two novel symmetric encryption schemes are introduced: Symmetric Additive Homomorphic Encryption (SAHE) and Symmetric Multiplicative Homomorphic Encryption (SMHE). These schemes are designed specifically to retain the expressiveness of existing PHE schemes, while providing improved query performance compared to state-of-the-art asymmetric PHE schemes.

Previous attempts to develop symmetric PHE schemes did so at the expense of expressivity. Specifically, existing SAHE schemes only support addition of two ciphertexts, whereas their asymmetric counterpart supports addition and subtraction of two ciphertexts, or between a ciphertext and a plaintext value, as well as negation of a ciphertext. Lack of expressiveness forces EDBMSs, like for instance Cuttlefish[60], to use multiple schemes to support complex operations SAHE does not support. As far as the authors know, no SMHE scheme exists. Achieving both good performance and expressivity is traded off against reduced ciphertext compactness (size of ciphertext, as well as query results). Strict compactness guaranteed by asymmetric PHE schemes is traded

for quantitatively good compactness in practice.

SAHE and SMHE ciphertexts are made up of the three elements constituting the vector $\langle v, l_p, l_n \rangle$. The size of v is fixed, however, the elements l_p and l_r are lists of IDs which increase in length as PHE operations are performed. A number of compaction techniques are proposed to keep these lists as small as possible, while maintaining full expressivity. Both lists contain IDs which are used to generate random values either added or subtracted from the message during decryption. List aggregation removes IDs which are present in both lists, as these operations cancel each other out. As IDs can appear multiple times in a list, they are grouped with a count, reducing list size and speeding up decryption thanks to batched operations on grouped IDs. Range folding folds consecutive, regularly spaced, IDs into ranges (e.g. [1, 2, 3, 4] to [1-4]) reducing the list size. During transmission or storage, integer list compression techniques are used to compress l_p and l_r to reduce their size.

Symmetria features a transformation module on the trusted client which rewrites plaintext queries to work on encrypted columns using PPE or PHE operations. A number of query optimization techniques are used to further improve query performance. Expression rewriting simplifies and restructures expressions for better performance. Rewrite rules include constant folding, factoring and replacing nested operations with cheaper equivalents (e.g. `add(c, c)` to `mlp(c, 2)`). Instead of creating intermediate ciphertexts for each homomorphic operation in an expression, operations are batched together and operation pipelining is used to execute homomorphic operations in a single pass. This reduces memory allocations and improves CPU cache locality. Since the encryption process uses pseudorandom numbers generated from IDs, precomputing them when IDs are predictable (e.g. incremental encryption) accelerates both encryption and decryption.

	Paillier	Packed Paillier	SAHE
enc	17285376 ± 0.13%	880921 ± 0.11%	1321 (63) ± 1.43%
dec	16390295 ± 0.01%	781727 ± 0.01%	1202 (153) ± 4.18%
add	34807 ± 1.37%	1666 ± 1.21%	457 ± 3.10%
adp	917141 ± 2.38%	104775 ± 0.95%	71 ± 0.37%
mlp	857943 ± 2.54%	–	406 ± 0.18%
neg	1370859 ± 0.07%	–	397 ± 0.11%
sub	1408870 ± 0.08%	–	819 ± 3.88%

	ElGamal	SMHE
enc	8700278 ± 0.04%	2974 (752) ± 0.29%
dec	4768193 ± 0.02%	3090 (1420) ± 0.23%
mul	25803 ± 0.16%	419 ± 0.92%
mlp	678 ± 1.17%	371 ± 0.11%
pow	505675 ± 2.53%	2856 ± 0.37%
inv	809711 ± 0.09%	3529 ± 0.24%
div	841260 ± 0.14%	4172 ± 0.25%

Figure 3.2: Operation execution time of SAHE and SMHE compared to asymmetric schemes, followed by relative standard error. All execution times given in nanoseconds[59].

Symmetria is evaluated using TPC-H and TPC-DS benchmarks. The incurred ciphertext size overhead is less than naive symmetric alternatives. The authors claim that Symmetria's PHE operations are up to 1000x faster than

Benchmark	System setup	Size	Time
TPC-H	`Plaintext` (text)	106.8 GB	–
	`Plaintext`	34.0 GB	2.4 min
	`Asym`	363.7 GB	84 min
	`Symmetria`	67.8 GB	14 min
TPC-DS	`Plaintext` (text)	38.6 GB	–
	`Plaintext`	15.1 GB	1.5 min
	`Asym`	482.4 GB	228 min
	`Symmetria`	39.7 GB	4 min

Figure 3.3: Storage overhead of SAHE, SMHE and asymmetric schemes compared to plaintext. `Plaintext` (text) indicates uncompressed plaintext data. All other methods use Parquet to store compressed data. Duration indicates the compression time for plaintext data and additionally the encryption time for all other schemes[59].

Paillier and Elgamal equivalents. Figures 3.2 and 3.3 show the operation execution times and storage usage of SAHE and SMHE compared to these asymmetric schemes. Additionally, Symmetria is compared to MONOMI[68], a system which uses similar split execution, however, using asymmetric PHE schemes. Compared to MONOMI Symmetria claims to be, on average, 3.8x faster on TPC-H queries and 7x faster on TPC-DS queries. This demonstrates the practicality of the proposed SAHE and SMHE schemes, despite their lack of strict ciphertext compactness[59].

KafeDB

Most software-only EDBMSs use PPE schemes to perform various SQL operations over encrypted data. PPE schemes like OPE and DE leak a lot of information while HE leaks little but is too computationally demanding. In [65] STE is presented, which is a novel encryption scheme which enables SQL queries over encrypted data without relying on leaky PPE and costly HE schemes.

KafeDB is an end-to-end EDBMS designed to balance security, performance and expressiveness. At the core of KafeDB is OPX, a novel encryption scheme which is built atop STE. OPX supports optimized query execution, allowing integration with existing DBMS. Under the hood OPX uses encrypted multimaps to store encrypted indices for rows, columns and value mappings. This enables SQL queries using filters, joins and projections. A key contribution of KafeDB is the use of structural chaining. This allows a server to perform multiple complex operations while preserving security. Each operation results in a token which can be used by a subsequent operation.

KafeDB includes an emulation layer which reshapes encrypted data structures into relational tables and reformulates encrypted query plans into stan-

dard SQL. This enables OPX-encrypted queries to be executed on unmodified DBMSs. This avoids having to build a custom database from scratch. Similar to other EDBMSs, KafeDB performs encryption and query optimization on a trusted local client, while executing queries over encrypted data on an untrusted server.

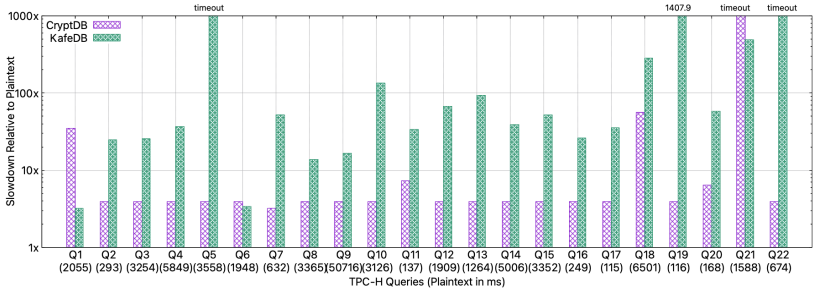


Figure 3.4: KafeDB and CryptDB slowdown relative to plaintext PostgreSQL on TPC-H benchmark using scale factor 1. KafeDB incurs about an order of magnitude more slowdown compared to CryptDB[71].

An evaluation of KafeDB is performed using the TPC-H benchmark with a scale factor of 1. KafeDB claims to offer significantly stronger security compared to existing EDBMSs like CryptDB which rely on PPE schemes. Performance-wise KafeDB claims a median slowdown of 45x compared to plaintext PostgreSQL, with a storage overhead of 13x. Figure 3.4 shows the slowdown of both KafeDB and CryptDB compared to plaintext PostgreSQL on the TPC-H benchmark using scale factor 1. The authors argue that these overheads are justified by the improved security guarantees. Specifically, OPX, and by extension KafeDB, claims to be resilient against known practical leakage-abuse attacks (which DE and OPE are susceptible to)[71].

3.2 Trusted Hardware

Encrypted databases using TH rely on specific hardware to perform queries over encrypted data. Data is often decrypted in this secure environment to allow regular SQL operators to be used without exposing plaintext data to an untrusted server.

3.2.1 Dedicated Trusted Hardware

Dedicated TH often comes in the form of a PCI extension card housing a system-on-a-chip. This system-on-a-chip acts as a secure coprocessor allowing for com-

putations in a tamper-proof environment using dedicated secure memory.

TrustedDB

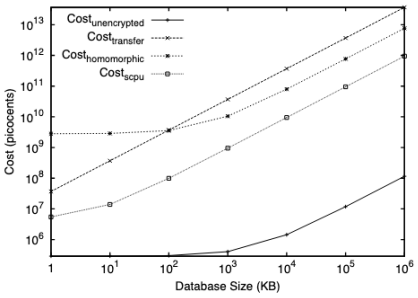
TrustedDB is a modified version of SQLite which performs all SQL operations on dedicated TH. The authors argue that software-only encryption techniques either limit expressiveness of queries or are prohibitively slow. TrustedDB uses cryptographic coprocessors as TH, specifically the IBM 4764, to execute queries on encrypted data. These coprocessors are tamper-resistant and provide a secure place for data to be decrypted and processed, even on an untrusted server.

A custom query parser is used to partition queries into a public and private component. The public component can be executed on the untrusted server, whereas the private component must be executed inside the Secure Coprocessor (SCPU). Performance is maximized by offloading as much computation as possible to the untrusted server. Scalable storage is supported, allowing the SCPUs to page encrypted data from the untrusted host. This allows the SCPUs to handle larger datasets than their memory would otherwise permit.

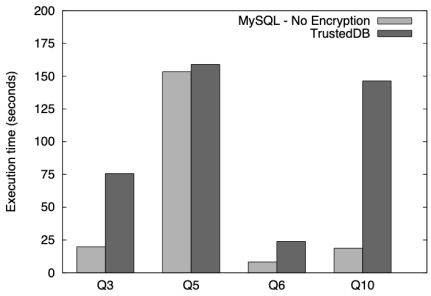
A cost model analysis is performed comparing three approaches of querying EDBMSs:

1. Transferring all encrypted data back to the client for local processing
2. Applying HE for server-side computations
3. Using TH

Via their cost analysis the authors claim that the third option, using TH, is several orders of magnitude more cost-efficient compared to the other two approaches, particularly in large-scale deployments. The large performance difference between approaches 2 and 3 arises from the comparatively low per-operation cost of SCPUs compared to the high computational cost of cryptographic primitives, such as modular multiplication, used in HE. While the cost is orders of magnitude lower for approach 3, the performance is much closer to approach 2. This is due to the much lower CPU speed of the SCPUs (just 233 Mhz). Figure 3.5a shows the results of the performed cost analysis. For each approach the total cost performing a SUM aggregation on all rows is calculated. The TH approach consistently costs orders of magnitude less than both other approaches.



(a) Cost analysis performing a SUM aggregation on all rows in the database with increasing database size.



(b) Comparison of runtime results for a subset of TPC-H queries between TrustedDB and unencrypted MySQL.

Figure 3.5: Cost analysis and runtime results for TrustedDB[8].

System throughput and latency is evaluated using a subset of queries from the TPC-H set (Q3, Q5, Q6 and Q10). Exclusively non-nested queries are used as the TrustedDB query parser does not efficiently support nested queries. The standard TPC-H schema is targeted and a scale factor of 1 is used, resulting in a database size of 1GB. Figure 3.5b shows the runtime results for the above-mentioned queries on the TPC-H dataset. Overall TrustedDB claims to incur a runtime overhead between 1.03x and 7.8x compared to plaintext MySQL. The authors additionally claim that the actual costs are orders of magnitude lower than any software-only EDBMS could achieve[8].

Cipherbase

Cipherbase is an EDBMS that provides strong end-to-end data confidentiality through encryption. It uses a novel architecture combining a conventional DBMS (SQL Server) with lightweight encrypted data processing in TH. It boasts the smallest Trusted Computing Base (TCB) - the code which is run on TH - among similar systems while claiming to provide significantly better security, performance and functionality. Cipherbase uses dedicated Field Programmable Gate Array (FPGA) extension cards as TH.

The Cipherbase architecture has several advantages over existing EDBMSs. While conceptually similar to TrustedDB, Cipherbase aims to perform minimal computations within TH. This reduces the TCB down to a size which can be formally verified to prove absence of bugs and backdoors. Extending an existing DBMS system and only performing low level expression evaluations in TH allows Cipherbase to profit from existing rich DBMS functionalities for 'free'.

Cipherbase supports the full TPC-C benchmark with all data strongly encrypted, as well as supporting indexing and transaction management on fully encrypted data. This division of work between the existing DBMS and the TH results in strong scalability while retaining data protection guarantees. Figure 3.6 shows the Cipherbase architecture.

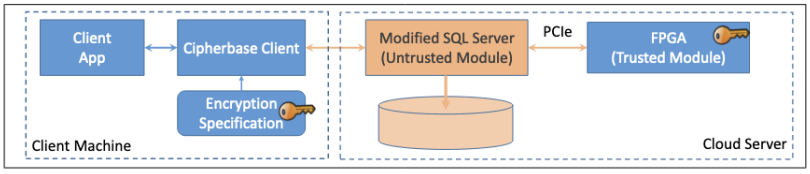


Figure 3.6: Cipherbase architecture using FPGA extension cards as TH[7].

Cipherbase introduces a number of optimizations to increase performance. Each FPGA runs up to n concurrent instances capable of evaluating independent encrypted expressions. This allows independent transactions to use the FPGA in parallel, boosting throughput. If all FPGA instances are busy, additional work is queued up and sent in a single batch when resources become available. This reduces the per-call latency and PCIe overhead. Expression folding folds multiple adjacent expressions, which require the TH, into a single expression. This reduces the number of TH calls, saving PCIe bandwidth and FPGA cycles, and allows for common sub-expression reuse, where a value is decrypted once and used multiple times.

A number of optimizations are specific to indexing. Cipherbase vectorizes the B-tree lookup process, sending the entire tree to the TH instead of sending data for each comparison separately. Binary search is performed on the vector and the match location is returned. This reduces the number of TH calls per lookup significantly. Frequently accessed B-tree vectors are cached, in plaintext, in FPGA memory. Subsequent queries using these B-trees send a reference to the tree. This saves both PCIe bandwidth and decryption costs.

Encrypted results of previously evaluated TH expressions are stored in memory, keyed by encrypted parameters, on the untrusted server. Consecutive evaluations with the same parameters can reuse these cached results. This reduces the number of redundant TH calls, especially in iterative situations where the same TH call might be performed many times.

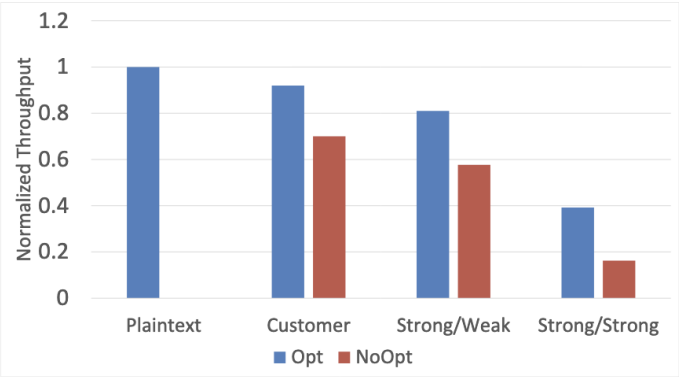


Figure 3.7: Normalized performance of Cipherbase on the TPC-C set compared to plaintext SQL Server. *Customer*: All PII columns in the Customer table are strongly encrypted. *Strong/Weak*: Index and foreign key columns are encrypted using DE, all other columns are strongly encrypted. *Strong/Strong*: All columns are strongly encrypted. *Opt*: With optimizations. *NoOpt*: Without optimizations[7].

Evaluation of Cipherbase is performed using the TPC-C benchmark. Figure 3.7 shows the normalized performance of Cipherbase compared to plaintext SQL Server. Encrypting just the columns containing Personally Identifiable Information (PII), Cipherbase claims to attain around 90% of throughput compared to plaintext SQL Server. Encrypting all columns reduces this to around 40%[7].

3.2.2 Trusted Execution Environments

Many modern processors include a TEE. Common examples of TEEs are Intels SGX and ARMs TrustZone. The TEE is a separate part of the processor which provides a secure and isolated area in which sensitive operations can be performed.

Azure SQL Always Encrypted

Always Encrypted (AE) is a feature of Microsoft SQL Server which adds column-level encryption to keep data confidential. AE claims to guarantee data confidentiality, even if the database server is compromised. Clients manage their own encryption keys, which are never sent to the server. Data stays encrypted at all times, when in rest on disk, when in use (except when inside the TEE) and when in transit. The design of AE inherits many of the design elements from Cipherbase.

AE uses a TEE to run small amounts of trusted code, called an enclave, as part of a larger untrusted process. The TEE hides computations and data from the host process and system. By extension this hides data and computations from the administrator of the system as well. AE uses the TEE to temporarily store decryption keys and perform query operations on decrypted data. Using a TEE introduces technical challenges. These include the necessity to split query processing between the TEE and the untrusted server, and preventing information leakage through data movements to and from the TEE. Figure 3.8 shows the AE architecture, which is similar to the Cipherbase architecture in Figure 3.6. AE uses Intel SGX or Windows VBS enclaves as TEE instead of the FPGAs Cipherbase uses.

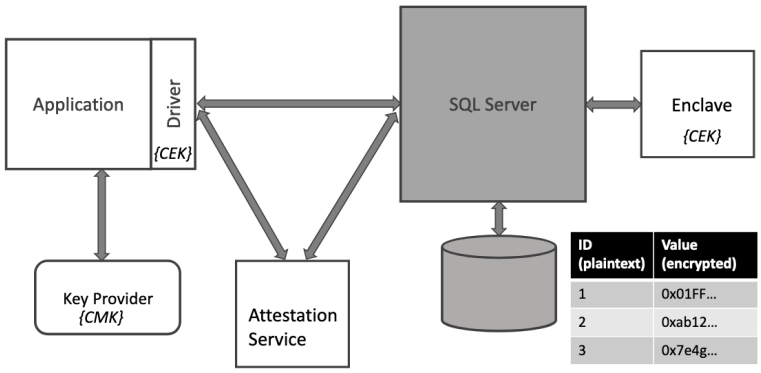


Figure 3.8: Azure Always Encrypted architecture. All parts are trusted except for the grayed out SQL server[3].

A strong adversary threat model is used. This adversary has unbounded power over the SQL server. This allows them to no only read data on disk and in memory, view all incoming and outgoing communications, but also to tamper with this data, by for instance using an attached SQL debugger. This adversary can't access the TEE to see any computations or internal data, as this is specifically what the TEE is designed for. However, an adversary can view memory movements to and from the TEE. AE can use both DE and RE. DE allows for equality operations without use of the TEE whereas RE is more secure but requires TEE usage for all operations, including equality operations.

AE uses a two level key hierarchy to encrypt data. A Column Encryption Key (CEK) is used to encrypt data in the database. This key is then encrypted using the Column Master Key (CMS). The encrypted CEK is stored in the database and the CMS is stored in an external Key Management Service like

Azure Key Vault. Storing the encrypted CEK in the database allows for 'application transparency', a core focus of AE. This allows applications not modified for AE to work with AE without any modifications.

When performing a query which requires TEE usage, the client must send the required CEK's to the enclave. This is done using a remote attestation process which checks the integrity of the host machine and the TEE, after which the Diffie-Hellman key exchange protocol is used to compute a shared secret. The client uses this shared secret to encrypt the required CEK's before sending these to the TEE which decrypts them using the same shared secret. To prevent replay attacks a nonce value is used during the attestation process.

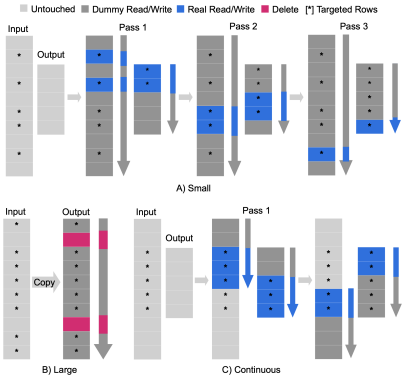
While AE and Cipherbase share the key idea of using TH and minimizing the amount of work performed within this hardware, they differ in a number of key areas. Key management is implicit in Cipherbase, whereas in AE the client is required to send CEKs when a query requires TH. Additionally, AE supports key rotations without exposing plaintext data to the server. Cipherbase focusses mainly on OLTP workloads, whereas AE supports a broader spectrum of workloads thanks to support for JOIN, GROUP BY and LIKE operators.

A slightly modified version of the TPC-C benchmark is used to evaluate the system. Specifically, queries using `ORDER BY` operations are edited as AE does not support these operations within TEEs. Exclusively columns storing personally identifiable information are encrypted. These constitute six of the 22 columns in the `CUSTOMER` table. All other columns remain unencrypted. On this benchmark AE claims to achieve roughly 50% of the throughput compared to plaintext SQL server. AE works well for its intended purpose of encrypting a small subset of sensitive columns, while still allowing simple queries to be performed on this data. Additionally, AE is one of the few reviewed EDBMSs which is not just a prototype system, but can actually be used[3].

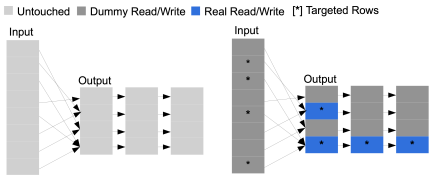
ObliDB

ObliDB is the first oblivious EDBMS capable of efficiently processing general SQL queries over encrypted data while hiding access patterns from adversaries. While TEEs like Intel SGX provide memory encryption and isolated execution, access pattern attacks can be used by adversaries controlling the OS to infer sensitive information through observation of which memory addresses are accessed during query execution. A number of novel oblivious query processing, join and aggregation algorithms are presented. Each of these algorithms is briefly discussed. Besides these algorithms, ObliDB introduces oblivious B+ trees built over Oblivious RAM (ORAM) with a number of optimizations including lazy write-back and removal of parent pointers to minimize the performance over-

head typically associated with usage of ORAM. ORAM, first proposed in [29], is a cryptographic primitive which hides memory access patterns in untrusted memory.



(a) Small, Large and Continuous query processing algorithms. In this example the TEE memory is only large enough to store two rows, requiring *Small Select* to perform three passes.



(b) *Hash Select* query processing algorithm.

Figure 3.9: All four query processing algorithms presented in ObliDB[24].

Four oblivious query processing algorithms are presented, each optimized for different query selectivity and data distributions. *Large Select* is used for queries selecting most of the table. A copy of the full table is used as query output, with a single pass marking unused rows via a dummy write. This avoids the overhead of multiple scan passes if most of the data will be retained. *Small Select* is used for queries with high selectivity, whose result sets fit inside TEE memory. Multiple passes are made over the input table, storing selected rows in a buffer within the TEE. Once the buffer is filled it is written to the output. The number of passes reveals only the result size, not the specific rows selected. *Continuous Select* handles queries where contiguous rows are selected (e.g. range queries on sorted data). For each row at position i writes to i mod |output| if row is selected, else performs a dummy write. Completes in a single pass but leaks that results form contiguous segments. *Hash Select* is a general-purpose query processing algorithm using oblivious hashing. For each row at position i it either writes the row or makes a dummy write to position H(i) in the output. Double hashing with fixed-depth collision chains is used to handle hash collisions obliviously, while maintaining a consistent access pattern regardless of row content (matching query or not). Figure 3.9 shows a visual

representation for each of the four presented query algorithms.

Three oblivious join algorithms are presented. *Hash Join* builds a hash table from the first table. The table is chunked into chunks fitting within TEE memory. Rows from the second table are subsequently probed. For each comparison one real or dummy row is written, depending on the probing result. This maintains a fixed access pattern. *Sort-Merge Join* combines both input tables and uses quicksort on chunks fitting within TEE memory. Chunks are then merged using bitonic sorting networks. A linear scan is performed to eliminate unmatched, and merge matched rows. *Zero-Oblivious-Memory* is a sort-merge join variant which requires no TEE memory.

Additionally, three aggregation algorithms are presented. *Standard Aggregation* sequentially scans the input table while maintaining aggregation statistics within the TEE. Its fixed sequential access pattern leaks only the input table size. *Grouped Aggregation* uses a hash table in TEE memory to track per-group aggregations. All operations access the hash table consistently, regardless of data distribution, to maintain a fixed access pattern. *Fused Select-Aggregate* combines selection and aggregation in a single operator to avoid creation of intermediate tables. Aggregates are instead computed directly over filtered input data.

A key contribution of ObliDB is its novel oblivious query planner. This query planner intelligently chooses between the various oblivious query, join and aggregation algorithms. A choice is made based on input/output table sizes and available TEE memory, claiming to achieve significant speedups between 4.6x and 11x, while maintaining security guarantees. The planners choice is based on information already leaked by the system, avoiding leakage of additional information.

An evaluation is performed comparing ObliDB with existing oblivious EDBMSs, as well as plaintext Spark SQL. On Big Data Benchmark queries, ObliDB claims to perform 1.1-19x better compared to Opaque[72], gaining the most performance on queries which benefit from index usage. A 7x improvement is claimed compared to HIRB[58], another oblivious index system. Compared to plaintext Spark SQL ObliDB claims to incur at most a 2.6x overhead, demonstrating strong security guarantees can be achieved with reasonable performance overhead. ObliDB leaks only table sizes, query result sizes and the structure of the physical query plan. Compared to existing oblivious EDBMSs, ObliDB leaks the same information but with a significantly broader set of supported functionalities and better performance characteristics[24].

3.3 Hybrid Query Execution

3.3.1 Hybrid Software & Trusted Hardware Query Execution

Software-only EDBMSs either have limited throughput due to high computational associated with PHE and FHE operations, or use weaker PPE schemes which leak too much information. Hybrid EDBMSs using a combination of both software-based cryptography and TH aim to offer the simplicity of a software-only solution, with the performance and security of TH implementation.

GaussDB

GaussDB is an EDBMS which utilizes both software-based encryption and hardware-based TEEs such as Intel SGX and ARM TrustZone. It aims to provide robust protection against adversaries, including privileged server administrators. Data remains encrypted throughout the entirety of its lifecycle: at rest, in transit and during computation. Figure 3.10 shows the GaussDB architecture, demonstrating its split query execution over the software-based Rich Execution Environment (REE) and TEE.

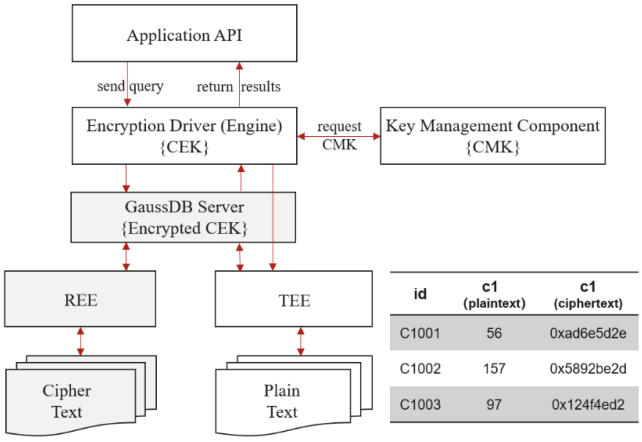


Figure 3.10: GaussDB architecture showing the REE which uses software-based PPE to interact directly with ciphertexts and the TEE which securely decrypts data to perform more complex SQL operations on plaintext data[73].

GaussDB introduces the TEE abstraction layer `secGear` which enables system portability between various hardware platforms. This makes GaussDB one of the first systems to support both Intel SGX and ARM TrustZone in a unified

way. Additionally, this simplifying adding support for potential new TEEs in the future.

Data is encrypted at the column level using either DE or RE via AES. Integrity of encrypted values is ensured via HMAC. A layered key management system using CEKs and CMKs is used, similar to the key management system Azure AE uses. In cases where the TEE is required, the CEKs are transferred to the TEE using a secure SSL channel which is encrypted using a shared key generated using Eliptic-Curve Diffie-Hellman (ECDH). This is similar to the attestation and key transfer used in Azure AE[3], which uses regular Diffie-Hellman. Indexes are encrypted using PPE to allow equality and range queries on ciphertexts. Sensitive and more complex operations, like `JOIN` and `GROUP BY` operations, are offloaded to the TEE. A hybrid query execution model is introduced which intelligently splits operations over the system software and hardware components. The dual mode operations of GaussDB ensures a good balance between security and throughput.

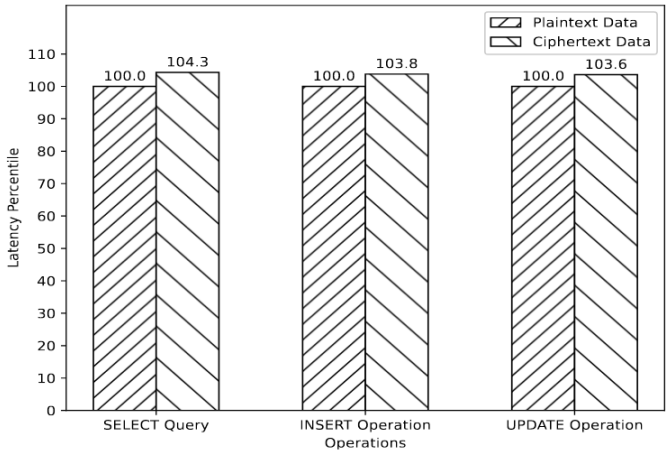


Figure 3.11: GaussDB performance evaluation results for three SQL operations claiming, on average, less than 5% performance overhead with encrypted columns compared to plaintext columns[73].

GaussDB is evaluated using three real-world scenarios. Evaluations are run on production-grade server hardware (Taishan 2280 server with dual-CPUs and 256GB of RAM). Overall GaussDB claims to incur around a 5% overhead with operations on encrypted columns compared to operations on plaintext columns. The evaluation is done using simple queries focussing mostly on OLTP workloads. The overhead for three SQL operations is shown in Figure 3.11. The

evaluation of GaussDB is lackluster and done using a non-generic benchmark. This makes it difficult to compare GaussDB to similar EDBMSs[73].

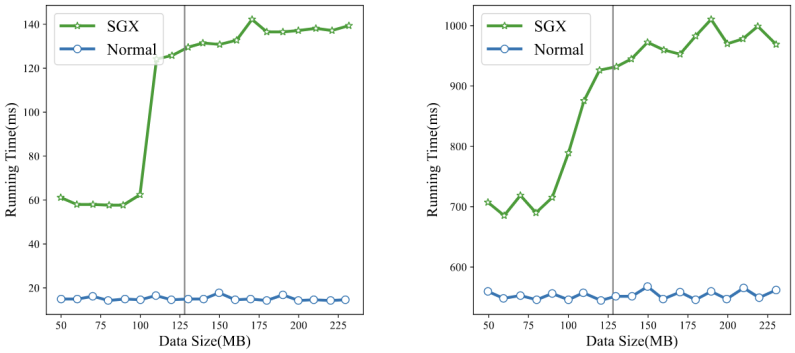
Enc^{2}DB

Enc^{2}DB is a hybrid EDBMS utilizing both software-based cryptographic methods and a TEE (Intel SGX specifically). Similar to GaussDB Enc^{2}DB is built atop openGauss, which is a fork of PosgreSQL extensively modified by Huawei (75% of the kernel code changed, optimized for Kunpeng processors)[32]. Enc^{2}DB aims to maximize security without excessively compromising on query performance. Encrypted query execution is performed transparently to users. Two operating modes are introduced: a software-only mode, EncDB, and a TEE-enabled mode, Enc^{2}DB.

The software-only mode uses a combination of additive and multiplicative HE (addition, group by and multiplication), DE (equality operations) and ORE (range comparisons) to perform SQL operations over encrypted data. This mode however suffers from redundancy and performance inefficiencies, particularly when performing queries with mixed operations which require a number of different PPE schemes. Efficiently supporting multiple PPE schemes requires multiple ciphertext versions per column. Each column can have 3-4 different encrypted forms, leading to storage bloat and increased query processing costs.

In its TEE-enabled mode data is decrypted and processed securely within Intel SGX enclaves. This removes the need for multiple ciphertext columns, reducing storage bloat and query processing costs, as well as supporting more complex queries compared to software-only mode. Adaptively selecting the best performing mode (software-only or TEE) is done using a self-adaptive mode switch based on runtime conditions. Enclave memory usage is a key metric used to decide operating mode, as excessive page faults in Intel SGX enclaves strongly degrade performance. As the SGX SDK does not provide these metrics a micro-benchmark, binary search over quick-sorted data, is presented to estimate the enclave's residual memory capacity. Figure 3.12 shows the runtime of this microbenchmark with increasing data size. Once the residual enclave memory runs out the runtime spikes, enabling estimation of available enclave memory. An extensive cost model taking into account encryption/decryption cost, enclave invocation overhead (ECALL - Enclave Call) and runtime paging overhead (page replacements inside SGX) is used to determine the optimal execution path for a query (software-only or using TEE).

Enc^{2}DB introduces a ciphertext-aware indexing mechanism which is compatible with standard PostgreSQL indexing and query optimizations. This is particularly effective for ORE based queries. Range and equality queries over



(a) Binary Search in SGX (b) Quick Sort in SGX

Figure 3.12: Results running binary search and quicksort within Intel SGX enclave with increasing problem size. Large spike in runtime indicates page faults due to the highly random nature of the algorithms and is used to estimate residual enclave memory.

encrypted data can utilize B(+)-trees via custom User Defined Types (UDTs) and UDFs. Additionally, SGX-local caching, reducing redundant cryptographic operations, and a batched ECALL mechanism, reducing enclave startup overhead, are implemented to further optimize performance.

An extensive evaluation is performed using the TPC-C benchmark, which focusses on OLTP workloads. Enc^{2}DB is compared to the state-of-the-art software-only EDBMS Symmetria[59] and openGauss as a plaintext baseline. Software-only EncDB suffers a 5x-10x performance degradation compared to the baseline due to computationally intensive cryptographic UDFs and storage overhead. Static TEE performs well at low concurrency levels but suffers from extensive page faults at higher levels of concurrency due to exhausted enclave memory. Self-adaptive TEE mode performs best overall, especially under high concurrency as it avoids enclave paging. Additionally, storage overhead is evaluated. Software-only mode incurs a hefty 30x storage overhead, thanks to the large number of ciphertext columns required, compared to TEE-mode which incurs a 7x storage overhead, requiring just one ciphertext column. Overall, Enc^{2}DB claims to consistently outperform Symmetria by around 50-100x. This large performance increase can be attributed to the much smaller number of HE operations Enc^{2}DB has to perform, thanks to its usage of the TEE, compared to Symmetria[40].

3.3.2 Multi-Party Hybrid Query Execution

The database systems above perform the entire query over encrypted data on the database server. Multi-party hybrid query execution moves some parts of the query to the trusted client.

MONOMI

MONOMI is an EDBMS which introduces a new approach based on split client/server execution. This allows MONOMI to perform part of a query on the untrusted server over encrypted data, and compute the final result on the trusted client. Similar to CryptDB, MONOMI uses a number of PPE schemes to perform specific SQL operations. MONOMI stores a each column multiple times, each of which encrypted using a different PPE scheme. This approach differs from CryptDB's onion approach, leaking significantly more information.

MONOMI includes an optimizer designed to determine which encryption schemes should be used to encrypt each of the database columns. Users specify, for each column, which operations are likely to be performed in future queries. MONOMI uses this specification to determine which encryption schemes to use for each column. Additionally, the split client/server query execution plan is optimized using this specification.

MONOMI partitions execution of a query between the client and server by constructing a SQL operator tree consisting of regular SQL operators as well as decryption operations which have to take place on the client. Various optimizations are performed to achieve the best performance. MONOMI can add additional columns containing precomputed encrypted expressions, which would otherwise require downloading a large amount of intermediate data to the client. To avoid sending all data to the client in cases where filter predicates cannot be computed over encrypted data, MONOMI generates a conservative estimate of the predicate and pre-filters intermediate results using this predicate. Exact filtering is subsequently performed on the client.

MONOMI is evaluated using a slightly modified TPC-H bechmark with a scale factor of 10, simulating an OLAP workload. As MONOMI does not support views and more than one string pattern match per query, TPC-H queries 13, 15 and 16 can't be evaluated. Queries 17, 20 and 21 cause trouble due to their correlated subqueries, which the optimizer is unable to handle efficiently. These queries are rewritten to use explicit joins. Query 21 does not have an obvious rewrite so it is left in its original form. Each column, not just those containing personally identifiable information, is encrypted one or more times depending on the optimizer selected settings. Figure 3.13 shows the TPC-H

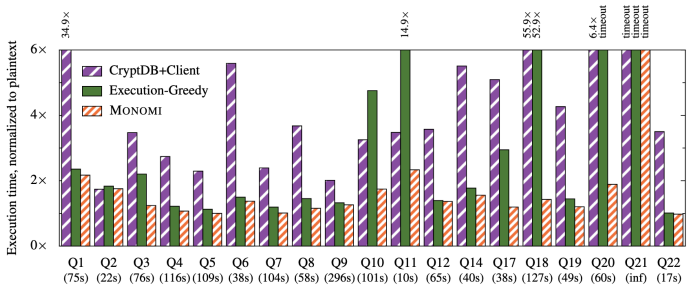


Figure 3.13: MONOMI and modified CryptDB TPC-H execution times, normalized to plaintext PostgreSQL. Overall MONOMI claims to incur a median overhead of 1.24x over plaintext PostgreSQL, with overheads ranging from 1.03x for query 7 to 2.33x for query 11. Query 21 times out due to its subquery complexity and non-obvious rewrite[68].

execution times for MONOMI and a modified version of CryptDB compared to plaintext PostgreSQL. Overall MONOMI claims a median overhead of 1.24x over plaintext PostgreSQL, with overheads ranging from 1.03x for query 7 to 2.33x for query 11[68].

Cuttlefish

Cuttlefish is an Apache Spark-based[70] EDBMS, designed to provide secure and expressive data analytics over encrypted data stored in public clouds. Savvides et al. state that one problem with existing systems, like MONOMI, is the focus on making encrypted computation transparent and hidden from users. This leads to inefficient querying and possible security compromises. Cuttlefish exposes constraints on security and confidentiality through so called Secure Data Types (SDTs). SDTs allow users to annotate data with security information like sensitivity level, data range, decimal accuracy, uniqueness, enumerated type and composite type.

Sensitivity is used by Cuttlefish to determine the level of encryption required for a column. Declaring a column as unique allows Cuttlefish to encrypt the column using DE while maintaining strong security and no leakage. The range and decimal precision of a column can optionally be provided to help Cuttlefish generate more expressive and efficient queries. Enumerated values can be defined which Cuttlefish will optimize using dictionary encoding. This allows Cuttlefish to apply the efficient encryption compilation technique to reduce the encrypted data size overhead which can lead to substantial reductions in query duration. Composite types can be declared to allow Cuttlefish to perform op-

erations on parts of a value. For instance a date value which can be split into a day, month and year part. Figure 3.14 shows a Cuttlefish table definition with SDTs.

```
TABLE Lineitem (
 orderkey,      long,   [+],
 linenumber,    long,   [+, unique],
 tax,           double, [accuracy(2), NONE],
 shipdate,      string, [composite[(4:int[+])-
                  (2:int[range(1-12)])-
                  (2:int[range(1-31)])]],
 shipinstruct,  string, [enum{IN_PERSON, MAIL,
                  RETURN, COLLECT}],
 quantity,      long,   [HIGH])
```

Figure 3.14: Cuttlefish table definition including SDTs[60].

A number of optimization techniques are proposed for the Cuttlefish compiler which make queries more expressive and efficient. Expression rewriting rewrites expressions in semantically equivalent ways which are more suitable for the encryption schemes used. Some encryption schemes only support a limited set of operations. Rewriting can help reduce the number of re-encryptions which have to be performed. Condition expansion uses known data ranges to reduce the number of expensive encrypted arithmetic and re-encryption operations required. A condition which is known to never be true based on known data ranges can be removed from the query. Selective encryption allows non-sensitive fields to remain plaintext allowing for more efficient querying.

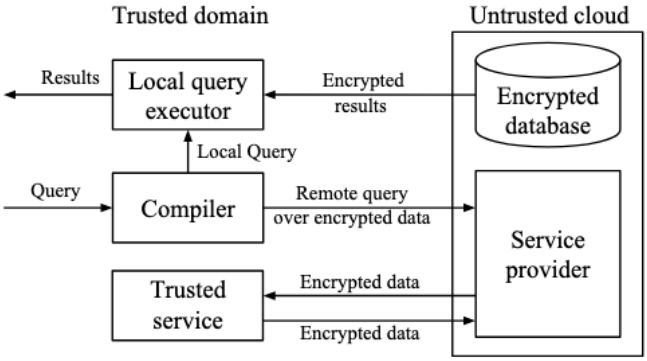


Figure 3.15: Cuttlefish architecture showing the trusted and untrusted domains. The query planner/compiler determines the query operations to perform locally versus on the server[60].

User queries are split into two parts. A remote part which can be executed over encrypted data in the cloud, and a local part which completes any operations not possible in the cloud on either a trusted client or TH (Intel SGX) if available on the server. PHE is used to perform computations on encrypted data. In cases where a required operation is not possible using PHE, re-encryption is performed on the client or TH to transition between encryption formats which do support the required operation. Cuttlefish uses a planner engine to decide where operations should take place. Heuristics are used to make an informed decision, using a number of features including the query structure, selectivity and previously gathered profiler data. The Cuttlefish architecture is shown in Figure 3.15.

Cuttlefish is evaluated using a comprehensive benchmarking suite. TPC-H and TPC-DS benchmarks, both using a scale factor of 100, are used to compare Cuttlefish with MONOMI[68] and Crypsis[35], two similar DBMSs both using a hybrid architecture, as well as plaintext Apache Spark[70]. Two variants of Cuttlefish are evaluated. The first, Cuttlefish-TH, performs re-encryption server side on TH and the second, Cuttlefish-CS, uses client side re-encryption. Cuttlefish supports the full TPC-H and TPC-DS benchmarks. Overall Cuttlefish-TH performs best with a claimed 2.34x overhead on TPC-H and 1.69x overhead on TPC-DS compared to plaintext Apache Spark. Disabling all optimizations increases the overhead to 4.23x on the TPC-DS benchmark, signifying their importance. Overall Cuttlefish claims to have a significantly lower overhead than prior hybrid EDBMSs, while offering better security through minimized use of weaker PPE schemes like DE and OPE[60].

VCrypt

VCrypt is a novel DuckDB extension that introduces fine-grained client-side columnar encryption to DuckDB. Minimal storage and performance overheads are achieved by leveraging DuckDB's columnar compression and vectorized execution capabilities. Traditionally encryption in analytical systems is expensive due to the overhead of storing cryptographic nonce values and the high performance penalty of per-value encryption. VCrypt addresses both issues.

Encrypted values are represented as structs, with metadata specifically formatted to support compression via standard techniques like Run Length Encoding (RLE) and delta compression. Specifically nonce values are structured into high, low and counter components. Batches of encrypted values share the same nonce high and low values, differing only in their counter values. This can be done as nonce values need to be unique, but not random. Sharing nonce high and low values allows these values to be effectively compressed (e.g. using RLE).

The counter values can be compressed using delta compression or bitpacking. The VCrypt nonce splitting structure is shown in Figure 3.16.

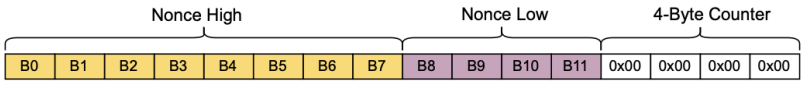


Figure 3.16: VCrypt nonce structure splitting the nonce value into a high, low and counter part. High and low values are RLE compressed, counter value is delta compressed[25].

VCrypt introduces vectorized, batch-based, encryption. Instead of per-value encryption, VCrypt encrypts 128 values into a single encrypted blob, referred to as a "tile". VCrypt tracks the position of each value within the tile using a compact one-byte field called the cipher. For variable-length types like strings it also stores a length and byte offset. These values are XOR-obfuscated and capped to improve compressibility. Tiles are grouped together in 512 byte or larger buffers and encrypted using AES-CTR which is both performant and avoids the per-value authentication tags required by AES-CBC. As AES-CTR does not require fixed input sizes or padding any part of the buffer can be decrypted given the correct offset and nonce. This prevents having to decrypt the entire buffer to retrieve just a single decrypted tile or value. Throughput increases up to 100x when using vectorized encryption over per-value encryption thanks to higher efficiency of AES primitives when operating on larger buffers.

VCrypt introduces a number of UDF which operate on DuckDB vectors (2048 values) performing vectorized encryption and decryption. If multiple rows share the same tile this tile only has to be decrypted once. These UDFs can be used explicitly but are envisioned to be used implicitly in VIEW definitions where the encrypted column is decrypted using the UDF. A key management mechanism is used which utilizes hierarchical encryption and uses the DuckDB secrets manager to store encryption keys securely.

MotherDuck, a system that provides hybrid query processing for DuckDB, is extended to support secure hybrid query processing. The MotherDuck query optimizer, which determines which parts of a query are performed client- and server-side, is extended to only perform sensitive data decryption client-side. This prevents the untrusted server from every seeing plaintext data or decryption keys, while offloading a large part of the computation, such as filters and joins, to the server. A demo based evaluation is performed using the TPC-H benchmark with a scale factor of 1[25].

3.4 Comparison

This section briefly compares the discussed EDBMSs on their functionality, security and performance.

3.4.1 Functionality

Table 3.1 shows the supported SQL operations as well as an overview of categories and used technologies for each discussed EDBMS. As no overview of supported operations is given in [25], VCrypt lists no supported SQL operations. Interestingly, while Azure AE is, in contrast to many other systems, a genuine product, it does not support commonly used order operations. This is due to its exclusive use of DE and RE schemes. These schemes do not support ordering of data. Overall the hybrid EDBMSs support the most functionalities. Operations which can't be performed over encrypted data can be securely performed over decrypted data either on TH or on the secure client. By extension hybrid systems benefit from more expressive queries as well.

EDBMS	**Category**	**Core**	**Supported Functions**						
			Equality Queries	Joins	Group By	Order By	Range Queries	Aggregation	String Matching
CryptDB	Software-Only	PPE	✔	✔	✔	✔	✔	✔	✔
Symmetria		PPE	✔	✔	✔	✔	✔	✔	✔
KafeDB		STE	✔	✔	✔	✔		✔	
TrustedDB	Trusted Hardware	Dedicated Hardware	✔	✔	✔	✔	✔	✔	
Cipherbase		Dedicated Hardware	✔			✔	✔		
Azure AE		TEE	✔	✔	✔			✔	
ObliDB		TEE + ORAM	✔	✔	✔		✔	✔	
GaussDB	Hybrid	PPE + TEE	✔	✔	✔	✔	✔	✔	✔
Enc^2DB		PPE + TEE	✔	✔	✔	✔	✔	✔	
MONOMI		Client + Server	✔	✔	✔	✔	✔	✔	✔
Cuttlefish		Client + Server	✔	✔	✔	✔	✔	✔	✔
VCrypt		Client + Server							

Table 3.1: Supported SQL operations as well as an overview of the category and used technologies for each discussed EDBMS.

3.4.2 Security

Software-only EDBMSs using PPE schemes offer the worst security. Schemes like DE and OPE leak a significant amount of information through their ciphertexts, with DE leaking equality and OPE leaking total ordering of values. Additionally, in [13] it is shown that a single OPE ciphertext leaks half of the most significant bits of its plaintext value, and in [48] it is shown that both DE and OPE are vulnerable to frequency attacks, allowing an attacker to infer plaintext values through frequency analysis. Newer PPE schemes like ORE[61][39] reduce the leakage of information through ciphertexts. These schemes increase security at the cost of significantly worse performance. EDBMSs like Cuttlefish, which allow users to specify uniqueness for a column, can use DE with less leakage as uniqueness guarantees no identical ciphertexts occur. Compared to PPE, STE offers significantly more comprehensive security through encryption of both data storage and search structures, preventing inference through access or search patterns.

TH based EDBMSs offer superior security over software-only EDBMSs. Inherent vulnerabilities in TH present security concerns, as well as side-channel attacks arising from the hardware, which are often overlooked. TH memory access patterns leak some information. Oblivious TH based EDBMSs like ObliDB obfuscate these access patterns, preventing an attacker from inferring queries and returned data while leaking minimal amounts of information.

Hybrid approaches using both software and TH solutions offer the best of both worlds. Performing all operations in TH provides the highest level of security, however, at the cost of performance as TH has limited memory and compute resources. Using software cryptography in combination with TH can offer higher throughput without decreasing security.

3.4.3 Performance

Performance of EDBMSs is difficult to compare. Overall TH and hybrid systems appear to perform best. Schemes like PHE and FHE, while providing excellent security, have significant computational overheads compared to similar plaintext operations run on TH. Specifically FHE, which runs between 50,000-1,000,000x slower than comparable plaintext operations[64][37]. A hybrid system can use both TH to prevent having to use computationally demanding schemes, and software-based cryptography to circumvent hardware limitations. Hybrid systems using a client/server model perform worse than their hybrid software-hardware counterparts. This is due to network latency which constitutes a large part of the overhead of these systems.

Optimizations greatly affect the performance of EDBMSs. Rewrite rules are used extensively in Symmetria to improve query performance by reducing the number of required re-encryptions, favouring more performant schemes and folding constants. Data annotation in Cuttlefish enables selective encryption and optimized encoding based on data format and sensitivity. GaussDB and Enc^{2}DB show that using runtime metrics such as TH memory availability and query complexity, to optimally partition a query across TH and software, significantly improves query throughput. TH transition overhead is reduced in Cipherbase and Enc^{2}DB via batched TH calls, batching multiple encrypted operations into a single call, effectively reducing the per call overhead. Vectorized and batch-based encryption, introduced by VCrypt, significantly boosts throughput by encrypting vectors, instead of individual values, as well as reducing storage overhead via nonce compression.

Chapter 4

Bloom Filter Based Encrypted Data Skipping

This chapter introduces the EDS problem, assumed threat model and the algorithms which implement Bloom filter based EDS.

4.1 Problem Statement

Many queries perform range-based filtering. From a large collection of files, each of which is associated with a specific range, only files whose range intersects with the query range are returned. Files whose range does not intersect with the query range are not relevant and are skipped. This is trivial with plaintext data, however, with encrypted data this becomes difficult. Existing schemes like OPE, ORE and HE can be used to perform EDS, however, they either leak significant amounts of information or are too slow to be used in any practical system. We introduce a Bloom filter based EDS scheme called BF-EDS, which allows for range predicates to be evaluated over encrypted metadata. For each file BF-EDS determines whether the file range intersects with the query range. If this is the case the file is returned to the client, if not, the file is skipped. BF-EDS both leaks significantly less information and is more performant compared to existing PPE schemes.

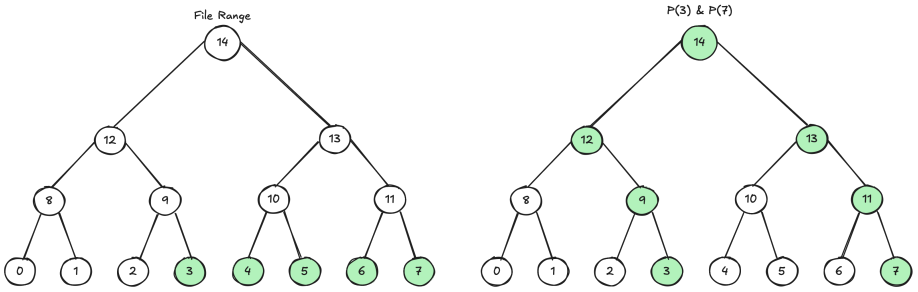
4.2 Threat Model

We assume an honest-but-curious attacker. Someone who works at a cloud provider and has access to users DBMSs. They might be an administrator

who has elevated privileges. This would allow them to attach debuggers to view running processes, as well as view and inspect memory. This is a passive attacker who can see and inspect all data, in and outgoing requests and network usage. We assume the local client is trustworthy.

4.3 The Scheme

Binary interval trees are used to denote ranges. Two set definitions, the coverage set $\mathbb{P}$ and MCS Λ, can be used to test for range intersections using set intersections. Bloom filters can be used to test for set membership, and by extension, to compute set intersections. Adding all nodes in set A to a Bloom filter and consequently testing Bloom filter membership for all nodes in set B results in the intersection set C where $A \cap B = C$. Due to the probabilistic nature of Bloom filters, C might contain elements not present in $A \cap B$. First, a simplified overview of the scheme is given, followed by an in depth explanation.



(a) Binary interval tree denoting the range $[3, 7]$.

(b) Coverage sets $\mathbb{P}(3) = \{3, 9, 12, 14\}$ and $\mathbb{P}(7) = \{7, 11, 13, 14\}$

Figure 4.1: Binary interval tree denoting range $[3, 7]$ with highlighted sets $\mathbb{P}(3)$ and $\mathbb{P}(7)$.

Testing whether two ranges intersect is done using the two set intersections $\Lambda(0, max_q) \cap \mathbb{P}(min_f)$ and $\Lambda(min_q, T) \cap \mathbb{P}(max_f)$. If both intersections result in a non-empty set the ranges intersect. Given a file f containing values in the range $[min_f, max_f]$ we can compute the coverage sets $\mathbb{P}(min_f)$ and $\mathbb{P}(max_f)$, which constitute the right side of both intersections, and insert these sets into a Bloom filter. We choose to insert the coverage sets into the Bloom filter instead of the MCSs due to their constant size. A coverage set will always contain $log_2(T) + 1$ nodes, whereas the number of nodes in a MCS varies. Having a constant number of nodes means each Bloom filter has a similar number of bits

set, reducing information leakage. Figure 4.1a shows the binary interval tree for file range $[3, 7]$ and Figure 4.1b shows the coverage sets $\mathbb{P}(3)$ and $\mathbb{P}(7)$.

Both sets $\mathbb{P}(min_f)$ and $\mathbb{P}(max_f)$ are added to the same Bloom filter. A keyed HF H_k is used to determine the bits to set in the Bloom filter bitset. This HF uses a secret key k_1 known only to the client. Usage of k_1 reduces the susceptibility of BF-EDS to pre-image and dictionary attacks. To prevent intersections between sets which should not intersect, for example between $\Lambda(0, max_q)$ and $\mathbb{P}(max_f)$, the nodes in the two sets are prefixed with distinct symbols before being inserted. All items in $\mathbb{P}(min_f)$ are prefixed with $*$, and items in $\mathbb{P}(max_f)$ are prefixed with $+$. Figure 4.2 shows a simplified Bloom filter insertion of the prefixed sets $\mathbb{P}(3)$ and $\mathbb{P}(7)$.

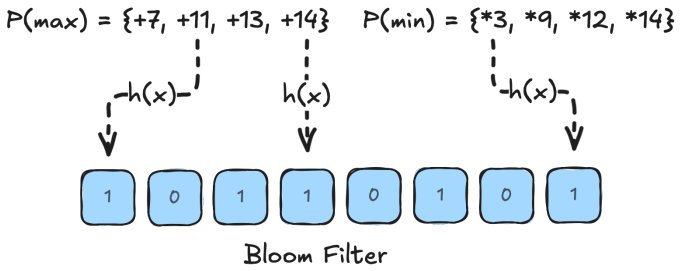
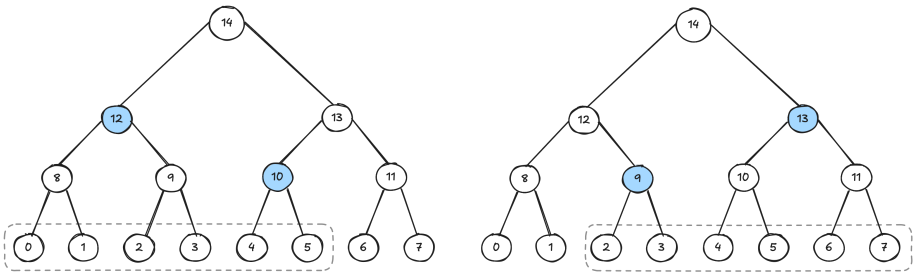


Figure 4.2: Bloom filter insertion of coverage sets $\mathbb{P}(3)$ and $\mathbb{P}(7)$. Both sets are prefixed with distinct symbols to prevent incorrect intersection results.

Testing whether a query range $[min_q, max_q]$ intersects with the file range $[min_f, max_f]$ requires computing the two MCSs, constituting the left side in both intersections, and checking Bloom filter membership for both sets. Given a query range $[2, 5]$ we compute the MCSs $\Lambda(0, 5)$ and $\Lambda(2, 7)$, shown in Figures 4.3a and 4.3b respectively. The items in these sets are prefixed with the same characters as the items in the coverage set we want to intersect with. We prefix the items in $\Lambda(0, 5)$ with $*$ and those in $\Lambda(2, 7)$ with $+$.



(a) MCS $\Lambda(0, 5) = \{*12, *10\}$. (b) MCS $\Lambda(2, 7) = \{+9, +13\}$.

Figure 4.3: MCSs for ranges $[0, max_q]$ and $[min_q, T]$.

Querying the Bloom filter results in two intersections as shown in Figure 4.4. The first intersection is in node $+13$ and the second intersection is in node $*12$. As there are two intersections the ranges intersect. This way we can use Bloom filters in combination with binary interval trees to determine whether two ranges intersect. Due to the probabilistic nature of Bloom filters there will be cases where there are more than two intersections, as well as cases where there are two intersections even with non-intersecting ranges.

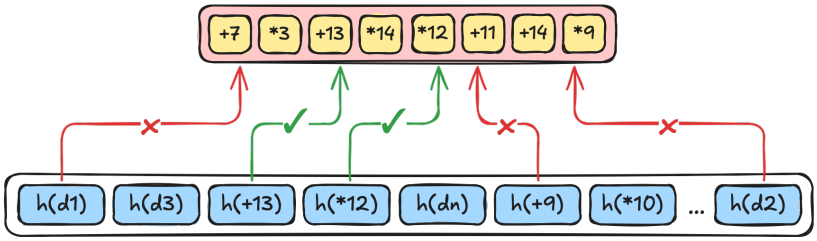


Figure 4.4: Querying the Bloom filter containing elements from $\mathbb{P}(3)$ and $\mathbb{P}(7)$ for intersections with the sets $\Lambda(0, 5)$ and $\Lambda(2, 7)$. Elements in sets checked for intersections are prefixed with the same character.

4.3.1 Bloom Filter Encryption

Plaintext Bloom filters leak little information, however, to further decrease leakage and protect against snapshot attacks we encrypt the Bloom filters using XOR encryption. Each bit in the Bloom filter bitset is encrypted separately using two PRFs F_1 and F_2. Algorithm 1 shows the encryption process. For each bit in the Bloom filter bitset a key k_{pos} is computed using F_1 based on a combination of the secret key k_2 and the current index in the bitset idx. This

key is used by F_2 together with a unique file identifier, f_{uuid}, to compute b. An XOR operation is performed between the plaintext bit value at idx and the most significant bit of b. The resulting value is stored in the encrypted Bloom filter at idx. This process is repeated for each bit in the Bloom filter bitset and results in an Encrypted Bloom Filter (EBF).

Algorithm 1 Bloom filter XOR encryption algorithm for File f_{uuid}.

```
EBF ← new Vec(m)
for idx in m do
    k_pos ← F_1(k_2, idx)
    b ← F_2(k_pos, f_uuid)
    EBF[idx] ← BF[idx] ⊕ b[0]
end for
```

4.3.2 Querying Using Query Tokens

Querying the EBF is done using a query token. This query token contains a list of sub-tokens, each of which contains k pairs, where k is equal to the number of HFs the Bloom filter uses. Algorithm 2 lists the query token generation algorithm. The two MCSs, $\Lambda(0, max_q)$ and $\Lambda(min_q, T)$, are computed and the items in these sets are prefixed with their distinct character. To prevent leaking any information about the query range the combined set of MCS IDs is padded with bogus $node_{id}$'s up to length $2 \cdot log_2(T)$. This ensures all query tokens are the same length, making it more difficult for an adversary observing query tokens to infer query information.

Algorithm 2 EBF query token generation algorithm.

$S_1 \leftarrow \Lambda(0, max_q)$ ▷ Compute minimum coverage sets
$S_2 \leftarrow \Lambda(min_q, T)$
prefix(S_1, $*$) ▷ Prefix elements with same character as $\mathbb{P}(max_f)$ set
prefix(S_2, $+$) ▷ Prefix elements with same character as $\mathbb{P}(min_f)$ set
$S \leftarrow S_1 \cup S_2$

while $|S| < 2 \cdot log_2(T)$ **do** ▷ Add bogus nodes up to $|2 \cdot log_2(T)|$
  $node_{id} \leftarrow \text{rand}(T)$
  $S \leftarrow S \cup node_{id}$
end while

$queryTok \leftarrow$ new Vec($2 \cdot log_2(T)$) ▷ e.g. length 128 with max value 2^{64}
for $node_{id}$ in S **do**
  $subTok \leftarrow$ new Vec(k) ▷ k pairs per sub-token
  for j in k **do**
    $idx \leftarrow H_j(k_1, node_{id})$ ▷ Compute hash to get bitset idx
    $k_{pos} \leftarrow F_1(k_2, idx)$
    Insert (idx, k_{pos}) into $subTok$ ▷ Add (idx, k_{pos}) pair to sub-token
  end for
  insert $subTok$ into $queryTok$
end for
shuffle($queryTok$)

For each element in the combined set we compute the k indices (idx) which would be set in the Bloom filter using the Bloom filters k HFs. We use F_1 to compute k_{pos} for each idx - similar to during the Bloom filter encryption process - adding the pair (idx, k_{pos}) to the sub-token. This sub-token is consequently added to the query token. The resulting query token contains $2 \cdot log_2(T)$ sub-tokens, each of which contains k pairs in the form (idx, k_{pos}). Before sending the query token to the database server, the vector of sub-tokens is shuffled to mix genuine and dummy sub-tokens.

Each file in the collection (e.g. data lake) is associated with a distinct Bloom filter encoding the range of that file, as well as a collection-wide unique file identifier f_{uuid}. During a query each Bloom filter is queried following the steps shown in Algorithm 3. For each sub-token $subTok$ in the query token we check whether for each pair (idx, k_{pos}) the Bloom filter plaintext bit at idx is set. During Bloom filter encryption each plaintext bit is XORed with the k_{pos} generated for its idx. As XOR operations are reversible, performing the same operation on the encrypted bit results in the plaintext bit. If the plaintext bit at idx is

not set, the algorithm can skip the remaining pairs in the sub-token. The bits at all k idx's need to be set for the element to be in the Bloom filter.

Algorithm 3 EBF querying using a query token for File f_{uuid}.

```
intersections ← 0
for subTok in queryTok do
    for (idx, k_pos) in subTok do
        b ← F_2(k_pos, f_uuid)          ▷ Compute F_2 using k_pos and file identifier
        match ← EBF[idx] ⊕ b[0]                          ▷ match is either 0 or 1
        if match == 0 then
            break                  ▷ Early termination, move to next sub-token
        else
            intersections ← intersections + 1
            if intersections == 2 then
                return true
            end if
        end if
    end for
end for
return false              ▷ Less than two intersections, no range intersection
```

If the number of intersections reaches two, the query and file ranges likely intersect. Consequently, the file is returned to the client. Due to the probabilistic nature of Bloom filters, as well as the dummy sub-tokens added to the query token, the two ranges might not actually intersect. Increasing m and reducing the number of dummy sub-tokens reduces this FPR.

4.3.3 Security Analysis

BF-EDS ensures security in a number of ways. First, we use keyed HFs H_k for our Bloom filters. This ensures that even in scenarios where the full plaintext bitset is observed or leaked, an adversary can't infer the actual encoded values. Additionally, using keyed HFs protects BF-EDS against dictionary attacks. Second, we encrypt our Bloom filters at rest to protect against snapshot attackers - attackers who gain control to the server temporarily - making it significantly more difficult for these attackers to recover plaintext bits. In the worst case scenario, where all plaintext bits in the Bloom filter are decrypted (through observation over many queries), our plaintext bitsets leak little information. An adversary may be able to infer the similarity between ranges associated with specific files through similarities in their Bloom filter bitsets. However, unlike schemes like OPE and ORE, BF-EDS does not leak significant ordering informa-

tion. Additionally, each query token contains the same number of sub-tokens, making it harder for an adversary to infer the ranges being queried through query token observation.

Chapter 5

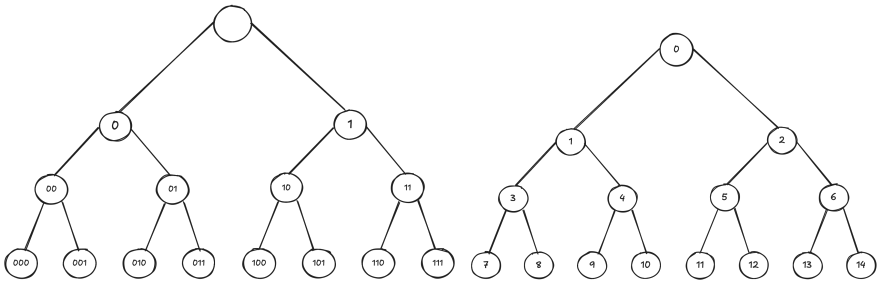
Implementing Bloom Filter Based Encrypted Data Skipping

This chapter covers the implementation of various binary interval tree algorithms, a generic HF interface, three Bloom filter variants, the BF-EDS C++ library and the integration of BF-EDS in DuckDB Iceberg. A small section on range mapping covers how signed integers, NULL values and strings can be mapped to numerical values, allowing BF-EDS over these types.

5.1 Binary Interval Trees

5.1.1 Coverage Set

Calculation of the coverage set is relatively simple. A bottom-up approach can be used to calculate the binary labels of all nodes in the coverage set for a given leaf node v. Starting at this leaf node all nodes in the set can be calculated by, for each layer in the tree, shifting the leaf nodes binary label to the right once. Choosing any leaf node in Figure 5.1a and moving towards the root node shows this pattern. The result is a set of size $log_2(T)+1$ as the root node is also part of this set. As we are interested in the set of ID's and not binary labels we have to convert each label in this set to its ID equivalent. A naive way to calculate the coverage set would be to perform the bottom-up approach and at each depth calculate the node ID.



(a) Binary interval tree with binary labels. (b) Binary interval tree with node ID's.

Figure 5.1: Binary interval trees with both binary labels and ID's.

Looking at Figure 5.1b a pattern can be seen in the ID's of nodes when moving from the root towards a leaf node. Each step the ID of each node is doubled and incremented by one. Additionally, if the rightmost bit is set, the ID is incremented once more. This ensures the ID of a nodes left child is smaller than the ID of its right child. This is propagated down to the leaf nodes and ensures all nodes in the tree have a unique ID. The C++ implementation of this method is shown in Listing 1. Calculating the coverage set using this method is substantially faster than the first method, as we don't have to convert $log_2(T)$ binary labels to ID's. Section 6.2.2 contains a comparative evaluation of both methods.

Listing 1 Implementation of coverage set calculation function in C++ which computes the node ID's in place while moving down the three.

```
std::vector<__uint128_t>
calculateCoverageSetForValue(uint64_t v, int8_t treeHeight) {
    // Create coverage set vector and add root node ID
    // to coverage set (will always be present, no need to compute)
    std::vector<__uint128_t> coverageSet;

    // Reserve space for all nodes in path to root (+1 for root)
    coverageSet.reserve(treeHeight+1);

    coverageSet.push_back(0); // Add root node to set
    __uint128_t currentNodeID = 0; // Set current node to root node

    // Going from root to the leaf node follows the pattern
    // ID = (ID * 2) + v[|v|] (value of rightmost bit at that depth)
    uint64_t mask = 1ull << (treeHeight - 1);
    for (int8_t idx = 0; idx < treeHeight; idx++) {
        // Multiply current node ID by two and add either 1 or 2
        // depending on the bit set at this depth (using mask)
        currentNodeID = (currentNodeID << 1) + (v & mask > 0 ? 2 : 1);
        coverageSet.push_back(currentNodeID);
        mask >>= 1;
    }

    return coverageSet;
}
```

5.1.2 Minimum Coverage Set

The MCS for a range r can be calculated using the algorithm provided in [42]. This algorithm starts at the root of the tree and works its way down using Breadth First Search (BFS). Each iteration the first item is popped from a queue and its coverage is checked. If the coverage of the node is strictly within the range r, the node is a part of the MCS and is added to the result set. If this is not the case the nodes two children are checked. The coverage of both children is calculated. If the child nodes coverage overlaps with r the child node is pushed to the queue. This process continues until the queue is empty.

This top down approach makes sense for large ranges, as the MCS for a large range likely contains nodes closer to the root, compared to a smaller range which is covered by nodes closer to the leaves. Figure 5.2 shows this difference with an exaggerated example. A bottom up approach may be faster with smaller ranges. A compromise between both implementations is a binary search implementation.

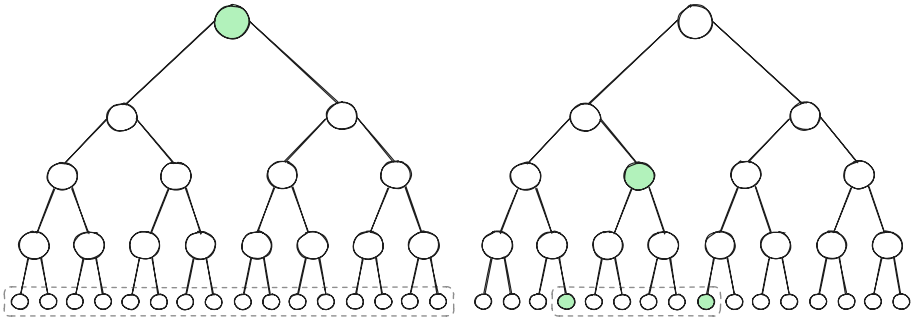


Figure 5.2: Comparison of MCS nodes between small and large ranges.

In this thesis we introduce a binary search variant of the MCS algorithm. Listing 2 shows the code for this binary search variant. The algorithm starts with the leaf node which represents the smallest value in the range r. From this leaf node binary search is used to jump up and down the path from the leaf node to the root. After each jump the coverage set of the node at the current depth is checked. If the coverage set covers exclusively the range r the node is a candidate for the MCS. The search is moved up the path to find a potential node that covers more of r. If the node covers more than r the search is moved down the path. Once the left and right pointers cross, the last checked node is added to the MCS. Figure 5.3 shows this process. The numbers in the nodes show the order in which they are checked.

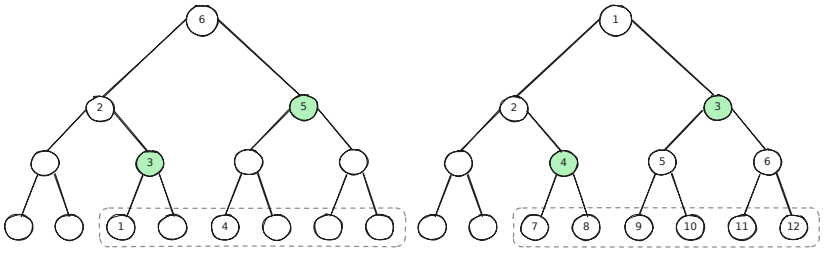


Figure 5.3: Order in which nodes are expanded for the binary search and top down MCS calculation algorithms respectively. In this case the binary search implementation expands less than half the number of nodes compared to the top down algorithm.

Listing 2 MCS calculation algorithm using binary search. Calculates the MCS for the given range r and uses the number of significant bits $sBits$ to determine the height of the tree.

```
std::vector<__uint128_t> calculateMinimumCoverageSetForRange(
  range<uint64_t> r, int8_t sBits) {
  // Result vector to store the MCS
  std::vector<__uint128_t> MCS;

  // Start with the leftmost value of range 'r'
  uint64_t currentNode = r.min;

  while (true) {
    // Binary search initialization, range is from
    // leaf node parent -> root
    int8_t low = 1, high = sBits, level = 0;

    // pRange will store the leaf node coverage of the current
    // candidate node level = 0 initially → leaf node
    range pRange = leafNodesAtEndOfPath(currentNode, sBits, level);

    // Binary search to find the largest subtree rooted at (
    // currentNode >> level) such that its covered leaves lie fully
    // within [r.min, r.max]
    while (low <= high && pRange.min >= r.min && pRange.max < r.max) {
      int8_t mid = low + (high - low) / 2;

      // Compute the range of leaves covered by the subtree rooted at
      // (currentNode >> mid)
      range nextRange =
        leafNodesAtEndOfPath(currentNode >> mid, sBits, mid);

      // If this subtree covers exclusively the range 'r' it's a candidate
      if (nextRange.min >= r.min && nextRange.max <= r.max) {
        // Promote level up to mid and continue searching for
        // a larger subtree
        level = mid;
        pRange = nextRange;
        low = mid + 1;
      } else {
        // Otherwise reduce the range to search for a smaller
        // valid subtree
        high = mid - 1;
      }
    }

    // Right shift the current leaf node by 'level' to move to the
    // parent node at the level where binary search terminated
    currentNode >>= level;
    MCS.push_back(calculateNodeID(currentNode, sBits - level));

    // Advance currentNode past the covered range of the added node
    currentNode = pRange.max + 1;

    // Stop when the target range is fully covered
    if (pRange.max == r.max) break;
  }
```

5.2 Hash Functions

In our scheme we extensively use HFs. Being able to swap the HF being used is important when benchmarking and testing. Therefore a HF function signature is defined. Any function implementing this signature can be used as a HF anywhere this is required. Listing 3 shows this signature, as well as the input struct. The caller provides a uint64 array of size k. At each index the function will return the 64 most significant bits of the k'th hash.

Listing 3 HF universal input struct and HF signature definition.

```
struct hfInput {
  uint8_t* key = nullptr;
  size_t keyLen = 0;
  int k = 0;

  // Only used by AES, Highwayhash and Siphash
  uint8_t* k1 = nullptr;
  size_t k1Len = 0;
};

// Hash function signature type definition
typedef void(*hash_func_signature)(const hfInput&, uint64_t* hashes);
```

Double hashing is used to compute k hashes. Two HFs of the form $h_1(x)$ and $h_2(x)$ are combined to simulate i additional HFs of the form $g(x) = h_1(x) + i \cdot h_2(x)$. This keeps the number of HF calls in situations where $k > 2$ fixed at 2. [36] demonstrates that double hashing does not increase FPR, while reducing the amount of computation significantly. Listing 4 shows the Blake3 HF implementation using double hashing.

Listing 4 Blake3 HF implementation using double hashing.

```
inline void Blake3(const hfInput& in, uint64_t* hashes) {
  // Init hasher
  blake3_hasher blake3Hasher;
  blake3_hasher_init(&blake3Hasher);

  // Initialize hash result array and perform hash
  std::array<uint8_t, BLAKE3_OUT_LEN> res1{};
  blake3_hasher_update(&blake3Hasher, in.key, in.keyLen);
  blake3_hasher_finalize(&blake3Hasher, res1.data(), BLAKE3_OUT_LEN);

  // Copy 64 most significant bits to output
  uint64_t res1Int;
  memcpy(&res1Int, res1.data(), 8);

  if (in.k > 1) {
    std::array<uint8_t, BLAKE3_OUT_LEN> res2{};

    // Re-hash res1 to get res2
    blake3_hasher_reset(&blake3Hasher);
    blake3_hasher_update(&blake3Hasher, res1.data(), BLAKE3_OUT_LEN);
    blake3_hasher_finalize(&blake3Hasher, res2.data(), BLAKE3_OUT_LEN);

    uint64_t res2Int;
    memcpy(&res2Int, res2.data(), 8);

    // Calculate k hashes using double hashing
    for (int i = 0; i < in.k; ++i) {
        hashes[i] = res1Int + (i*res2Int);
    }
  } else {
    hashes[0] = res1Int;
  }
}
```

5.3 Bloom Filters

5.3.1 Basic Bloom Filters

Our basic Bloom filter implementation uses a boolean vector to store bits. During insertion and checking, k hashes are computed using double hashing. The basic Bloom filter is encrypted using XOR encryption. Each bit is encrypted separately. While this is secure, as the server has to observe a lot of queries to decrypt the full Bloom filter, it is also slow. Given $k = 8$ and a tree height of 64,

a query token would contain 128 sub-tokens ($2 \cdot log_2(2^{64})$), each of which contains eight pairs. During a query, theoretically, a maximum of 1024 decryptions are performed. Each decryption requires a computationally demanding hash computation. This high number of decryptions slows down the query significantly, consequently, performing less decryptions should lead to faster queries.

5.3.2 Register Blocked Bloom Filters

Our register blocked Bloom filter implementation is inspired by [44]. A vector of uint64 values is used as the Bloom filter bitset. These values are machine word sized and fit in a single register on a system with a 64 bit architecture. During insertion and checking operations $k + 1$ hashes are computed. The last hash result is used to select a block. A mask is used to insert or check values in a block using a single operation. The code which generates this mask is shown in Listing 5. For each of the k hashes a bit within the mask is set using a logical OR operations.

During insertion a logical OR operation is performed between the block and the mask. This copies over the bits set in the mask to the block, while not changing any bits already set in the block. When performing a check a logical AND is performed between the block and the mask. If the result of this is equal to the mask, all bits in the mask are also set in the block. This indicates the value is present in the Bloom filter.

Listing 5 Mask construction code used in the register blocked Bloom filter implementation. Sets k bits within a 64 bit value. These bits are copied into the actual bitset block using a single logical OR operation.

```
uint64_t constructMask(const std::vector<uint64_t>& hashes) {
  uint64_t mask = 0;
  for (int i = 0; i < k; ++i) {
      mask |= 1ull << (hashes[i] & (blockWidth-1));
  }

  return mask;
}
```

The register blocked Bloom filter is encrypted using XOR encryption at the block level. This makes it slightly easier for an observer to observe all plaintext bit values compared to the basic Bloom filters bit-by-bit encryption. However, for a modest 16384 bit Bloom filter this would still require observing 256 block decryptions.

5.3.3 Split Block Bloom Filters

Our SBBF implementation is inspired by [5] and modified to use ARM NEON instructions. A block in this implementation is represented by the `uint32x4x2_t` datatype, which contains two 128 bit vectors, each of which contain four 32 bit lanes. Listing 6 shows the SBBF insertion code. Highwayhash256 generates a 256 bit hash and returns the first 64 bits of this hash. The most significant 32 bits of this hash are used to select a 256 bit block using the code in Listing 7.

Listing 6 Item insertion code for a SBBF. HighwayHash is used to generate a 256 bit hash of which the most significant 64 are returned. The most significant 32 bits are used to select a block. Multiply-shift hashing is used on the remaining 32 bits to calculate eight distinct hashes. A mask is generated using these eight hashes.

```
void BlockedBloomFilterParquet::insert(uint8_t *key, size_t keyLen,
    uint8_t *k1, size_t k1Len) {
  // Hash the item
  hash_functions::hfInput in = {key, keyLen, 1, k1, k1Len};
  uint64_t hash = hash_functions::Highwayhash256(in);

  // Determine block using most significant 32 bits from hash
  uint32x4x2_t* block = getBlock(hash);

  // Construct mask using multiply-shift hashing
  uint32x4x2_t mask = constructMask(hash);

  /* Perform OR operation between the block and the mask,
    * since block is a pointer this operation directly updates
    * the block data
    */
  block->val[0] = vorrq_u32(block->val[0], mask.val[0]);
  block->val[1] = vorrq_u32(block->val[1], mask.val[1]);
}
```

A `uint32x4x2_t` mask is constructed with one bit set in each of its eight lanes. Listing 8 shows the code used to create this mask. The least significant 32 bits of the hash are used to construct the mask. Multiply-shift hashing is used to generate eight unique hashes using the original hash. Right shifting each of these hashes by 27 results in values within the range $[0, 31]$ set in each of the eight lanes. These values are referred to as the shift values. Another `uint32x4x2_t` value is created with all lanes set to 1. The `vshlq_u32` instruction is used to shift these 1's left by the shift values previously calculated. The result of this is a `uint32x4x2_t` value with one bit set in each lane. A logical OR operation

Listing 7 Block selection code for a SBBF. Right shifting by 32 yields the most significant 32 bits which are used to select a block. Multiplication by the number of blocks followed by right shifting by 32 results in an index in the domain $[0, b-1]$ where b is the number of blocks.

```
uint32x4x2_t* getBlock(uint64_t hash) {
  /* Grab most significant 32 bits and use multiply-shift
    * trick to skip having to perform a modulo operation
    */
  return &this->blocks[((hash >> 32) * this->numBlocks) >> 32];
}
```

is used to set the bits set in the mask in the block.

Checking whether an item is present in a SBBF follows the same steps as inserting an item, with exception of the final step. Listing 9 shows the code for this process. Once the block has been selected and the mask constructed a logical AND operation is performed between the block and mask. If all bits in the mask are also set in the block the result of this operation will be identical to the original mask. The `vceqq_u32` instruction is used to compare the mask and the result, setting all bits in a lane if the two lanes in the given vectors are equal. If the minimum value across all lanes is $2^{64} - 1$, all values are equal.

Listing 8 Mask construction code for a SBBF. The least significant 32 bits of the hash are broadcast to eight distinct lanes. Each of these lanes is multiplied by a predefined odd constant. Right shifting each lane by 27 results in a value within the range $[0, 31]$ set in each lane. These values are called the shift values. The final mask is generated by left shifting eight 1's by the previously computed shift values, resulting in eight lanes with a single bit set at a seemingly random index.

```
uint32x4x2_t constructMask(uint32_t hash) {
  // Constants used in multiply-shift hashing (same as in Parquet)
  const uint32_t rehash_vals[8] =
    { 0x47b6137bU, 0x44974d91U, 0x8824ad5bU, 0xa2b7289dU,
      0x705495c7U, 0x2df1424bU, 0x9efc4947U, 0x5c6bfb31U };

  // Load rehash constants into two 128-bit vectors
  uint32x4x2_t rehash;
  rehash.val[0] = vld1q_u32(&rehash_vals[0]); // first 4 constants
  rehash.val[1] = vld1q_u32(&rehash_vals[4]); // next 4 constants

  /* Broadcast hash across two 128-bit vectors
    * (set all uint32_t values to the 'hash' value)
    */
  uint32x4_t hash_vec = vdupq_n_u32(hash);

  /* Multiply and shift: (rehash * hash) >> 27
    * (shifting right by 27 leaves only 5 bits, being within a range [0, 31])
    */
  uint32x4x2_t shift;
  shift.val[0] = vshrq_n_u32(vmulq_u32(rehash.val[0], hash_vec), 27);
  shift.val[1] = vshrq_n_u32(vmulq_u32(rehash.val[1], hash_vec), 27);

  // Create 128 bit vector with value of 1 in all lanes
  uint32x4_t ones = vdupq_n_u32(1);

  /* Shift left by the per-lane shift values, reinterpret the shift values
    * as signed as the vshl instruction uses the sign to determine direction
    */
  uint32x4x2_t result;
  result.val[0] = vshlq_u32(ones, vreinterpretq_s32_u32(shift.val[0]));
  result.val[1] = vshlq_u32(ones, vreinterpretq_s32_u32(shift.val[1]));

  return result;
}
```

Listing 9 Item checking code for a SBBF. The same block selection and mask construction code is used. Instead of performing a logical OR between the selected block and generated mask, a logical AND operation is performed. If all bits in the mask are set in the block, the result of this operation is identical to the supplied mask.

```
bool BlockedBloomFilterParquet::check(uint8_t *key, size_t keyLen,
    uint8_t *k1, size_t k1Len) {
  // Hash the item
  uint64_t hash = hash_functions::Highwayhash256({key, keyLen, 1, k1, k1Len});

  // Determine block using most significant 32 bits from hash
  uint32x4x2_t* block = getBlock(hash);

  // Construct mask using multiply-shift hashing
  uint32x4x2_t mask = constructMask(hash);

  // Perform AND between the block and mask
  auto res1 = vandq_u32(block->val[0], mask.val[0]);
  auto res2 = vandq_u32(block->val[1], mask.val[1]);

  /* If the AND result and the mask are the same, the element is in the block
    * vceqq_u32 -> Compares two uint32x4_t values, for each lane if the values
    * are equal it sets that lane in the result to 0xFFFFFFFF
    * vminvq_u32 -> Grabs the smallest value from the four lanes
    * If the smallest value is equal to UINT_MAX (0xFFFFFFFF)
    * then all lanes were equal, and therefore the mask and AND
    * result are equal
    */
  return (vminvq_u32(vceqq_u32(res1, mask.val[0])) == UINT_MAX) &&
         (vminvq_u32(vceqq_u32(res2, mask.val[1])) == UINT_MAX);
}
```

The SBBF can be encrypted using XOR or AES encryption. Using XOR encryption the bitset is encrypted on a block level (256 bits), similar to the register blocked Bloom filter. Given a tree height of 64 a query token contains 128 sub-tokens, each containing 8 pairs. These 8 pairs reference indices in the same block, requiring a single decryption per sub-token. Given a large bitset size, containing more than 128 blocks, and perfectly random and uniform hashing, a query would require 128 decryptions. In real-world usage the number of decryptions is far lower due to block collisions and local caching of decrypted blocks.

Encryption using AES encrypts the full Bloom filter bitset at once, requiring a single decryption per query. For our assumed threat model this encryption

method is substantially worse than XOR encryption. An observer can obtain the complete plaintext bitset after just a single query, as opposed to having to observe each bit or 256 bit block be decrypted to piece together the full plaintext bitset. Using AES encryption trades off a higher level of security for better performance. Given the low level of leakage from a plaintext Bloom filter bitset, AES encryption can be a compelling choice if better performance is required. AES-CBC-256 is used as it encrypts in 128 bit blocks. With our 256 bit blocks this method requires no padding. An empty IV is used for two reasons. First, we don't want to store IV's. Second, when using no IV, identical plaintext Bloom filters will encrypt to identical ciphertexts. This is not a concern as this only leaks both files contain the same upper and lower bounds for a specific column, which in many cases could be considered acceptable leakage.

Using The Full 256 Bits

The implementation mentioned above exclusively uses the first 64 bits of a hash, calculating the eight required hashes using multiply-shift hashing. While multiply-shift hashing is cheap and results in uniform and independent hashes, the outputs are neither unpredictable nor collision resistant. This is exacerbated in this implementation as the same 32 bit hash is used as the base for each of the eight resulting hashes. This section describes an extended, more secure implementation, which uses a hash digests full 256 bits.

Overall the insertion and checking methods are the same. Mask calculation is adapted to use the full 256 bits of a digest. Listing 10 shows the multiply-shift step in the modified mask construction method. Instead of using the same 32 bits of the hash for each multiplication, a distinct 32 bit part of the 256 bit hash is used. Simply splitting the 256 bit hash into eight 32 bit hashes does not result in sufficiently independent values, leading to a significantly higher FPR. Multiply-shift hashing these eight hashes with odd constants results in uniform and independent hashes. Section 6.4.1 contains benchmarks comparing performance and FPR between the three SBBF variants.

Listing 10 Difference in mask construction for SBBF when using the digests full 256 bits. Instead of broadcasting the least significant 32 bits of the hash to eight lanes, the eight lanes are filled by the 256 bit digest. This improves security as each lane has a distinct value, as opposed to sharing the same 32 bit value. Multiply-shift hashing is required to generate eight distinct and independent hashes.

```
uint32x4x2_t hashes;
hash_functions::Highwayhash(
  {key, keyLen, 1, k1, k1Len},
  reinterpret_cast<uint64_t*>(&hashes)
);

// Multiply and shift: (rehash * hashes[i]) >> 27
uint32x4x2_t shift;
shift.val[0] = vshrq_n_u32(vmulq_u32(rehash.val[0], hashes.val[0]), 27);
shift.val[1] = vshrq_n_u32(vmulq_u32(rehash.val[1], hashes.val[1]), 27);
```

5.4 BF-EDS Library

This section describes the implementation of the BF-EDS C++ library which contains the algorithms described in Sections 4.3.1 and 4.3.2, as well as all binary interval tree algorithm implementations. The library focusses on ease of use and extendability. Adding a new Bloom filter type, encryption method or HF is trivial. Templating is used extensively to write generic and type-safe code that can reuse existing code, while simultaneously allowing differing implementations when necessary. A `QueryManager` class is introduced to manage keys (client side) and Bloom filter settings, reducing the parameters required for all methods.

Bloom filter XOR encryption is implemented for basic, register blocked and SBBF types. The EBF creation method takes a file range $[min_f, max_f]$. The two coverage sets $\mathbb{P}(min_f)$ and $\mathbb{P}(max_f)$ are calculated and their elements are added to the templated Bloom filter type. The method returns a unique pointer to an instance of the abstract Bloom filter class. By using this abstract Bloom filter class, which exposes a number of generic Bloom filter methods (insert, check, retrieve bitset), each method can use any type of Bloom filter. New Bloom filter types can be added without requiring any code changes in most methods. Once all coverage set elements are inserted into the Bloom filter, its bitset is encrypted using either XOR or AES-CBC-256 encryption. The encryption method is specified using a method parameter.

Creating query tokens uses the algorithm described in Section 4.3.2. The global query token creation method is the same for each Bloom filter type,

however, the sub-token creation methods vary. This difference is due to the difference in block size when encrypting the bitset using XOR encryption (bit-by-bit or block-by-block with varying block sizes). The query token method generates the MCS for the requested query range. For each node ID in the MCS a sub-token is created using a Bloom filter type specific implementation. The query token is padded with bogus sub-tokens up to length $2 * log_2(T)$. Querying is again Bloom filter type specific thanks to the differences in query token sub-tokens. Thanks to the extensive usage of templating, changing the type of Bloom filter and encryption being used solely requires changing a single template parameter. This significantly simplifies both testing and evaluation.

5.5 Iceberg

This section describes the process of creating Iceberg tables for testing and evaluation, updating manifest files with our Bloom filters and integrating BF-EDS in the DuckDB Iceberg extension.

5.5.1 Creating Iceberg Tables

Iceberg tables can be created a number of different ways. Using Apache Spark, `INSERT` operations can be performed to create Parquet files with accompanying Iceberg metadata. This process is slow and does not work for already existing data files. As Iceberg is just a metadata format, metadata can be created for existing files as well using Icebergs `add_files` method. Spark supports using this method, however, when used it compacts and rewrites the original files. Repartitioning, to undo the compaction, is possible but breaks the original ordering of data across files. This makes it unusable for our application, as some evaluations require data to be ordered or distributed across files in a specific way. Additionally, Spark' rewriting of files slows down the process significantly, especially for larger tables.

To tackle the abovementioned issue PyIceberg is used to create Iceberg tables. PyIceberg implements the `add_files` method differently compared to Spark. Specifically, PyIceberg does not compact and rewrite files, unless explicitly asked to do so. Duplicate file checking can be disabled to speed up the Iceberg table creation process significantly.

5.5.2 Adding Bloom Filters To Manifest Files

Iceberg manifest files are stored using the Avro file format. Avro is a data serialization framework which provides a fast, schema-based binary data format.

Manifest files follow a strict schema, a small part of which is shown in Listing 11. The `data_file` field is a record which contains a large number of fields. These fields contain general information and statistics for a specific data file. As data files can contain multiple distinct columns, most statistics are defined as dictionaries (e.g. upper and lower bounds). This allows statistics to be added on a per-column basis. The added Bloom filter extension follows the same pattern as these statistics fields, allowing a Bloom filter to be added for each distinct column in a file. Listing 12 shows the Bloom filter field which is added to the `data_file` record. The `bloom_filters` dictionary maps one or more column IDs to Bloom filter bitsets. Avro implicitly stores the size of the bitset byte arrays, meaning no additional field is required to store m.

Listing 11 Manifest file schema consisting of rows of manifest entry records. A small subset of the fields in the manifest file schema are shown in this Listing.

```
{
  "type": "record",
  "name": "manifest_entry",
  "fields": [
    {
      "field-id": 0,
      "name": "status",
      "type": "int"
    },
    // ...
    {
      "field-id": 2,
      "name": "data_file",
      "type": {
        "type": "record",
        "name": "r2",
        "fields": [
          {
            "field-id": 134,
            "doc": "File format name: avro, orc, or parquet",
            "name": "content",
            "type": "int"
          },
          // ...
        ]
      }
    }
  ]
}
```

Adding Bloom filters to Iceberg manifest files is done using a small C++ helper program. This program takes a number of arguments, including a manifest file path, bitset size m and a number of client-side secret keys used in the BF-EDS PRFs F_1 and F_2, as well as an optional 32 byte key used for AES encryption (if used instead of XOR encryption).

Listing 12 Manifest file schema extension to add support for per-column Bloom filters. Bloom filters are stored in a dictionary which maps column ID to Bloom filter bitset. Avro implicitly stores lengths of byte arrays, meaning no additional field is required to store m.

```
{
"name": "bloom_filters",
"type": [
  "null",
  {
    "type": "array",
    "items": {
      "type": "record",
      "name": "k144_v145",
      "fields": [
        {
          "name": "key",
          "type": "int",
          "field-id": 144
        },
        {
          "name": "value",
          "type": "bytes",
          "field-id": 145,
          "doc": "The Bloom filter bitset"
        }
      ]
    },
    "logicalType": "map"
  }
],
"doc": "Map of column id to Bloom filter",
"default": null,
"field-id": 141
}
```

For each manifest entry a selection of columns is made for which both an upper and lower bound is present in the entry's `upper_bounds` and `lower_bounds` dictionaries. For each of these columns a Bloom filter is generated using the coverage sets $\mathbb{P}(column\_lower)$ and $\mathbb{P}(column\_upper)$ and added to the manifest entry's `bloom_filters` dictionary. Once all Bloom filters for an entry are generated, the `upper_bounds` and `lower_bounds` dictionaries are NULLed to prevent leakage of bounds information. The updated manifest file is then written to the files original location, overwriting the existing file.

5.5.3 Adding BF-EDS To DuckDB Iceberg

A hybrid query execution model makes most sense for the BF-EDS scheme. Accurate data skipping precisely selects files relevant for a query. These files are then transferred to a trusted client which decrypts the files and performs the remaining part of the query locally. Due to the limited amount of time available and the complexities involved in developing a hybrid query execution system (MotherDuck 2.0), the BF-EDS implementation in this thesis is run entirely server side. Adding BF-EDS methods to the DuckDB Iceberg extension is done through linking with the BF-EDS library, introduced in Section 5.4, which implements all required BF-EDS functions.

The initial step in a query consists of converting the set of applied DuckDB table filters to a per-column filter range mapping. Currently this mapping is one-to-one, however, extending this mapping to support multiple filter ranges per column is trivial. For each of these filter ranges a query token is generated. Query tokens are stored in a dictionary which maps column ID's to vectors of query tokens. This allows for multiple query tokens to be added per column. DuckDB's active query ID is stored with this dictionary, ensuring query token creation happens exactly once per query. Additionally, a number of custom settings are added which alter DuckDB Iceberg's querying behavior:

- `use_encrypted_bloom_filters`: Enables usage of EBFs. If not enabled Bloom filters are not loaded and regular upper and lower bounds statistics are used.
- `bloom_filter_encryption_method`: Specifies the encryption method used for the EBF. One of: XOR, AES or NONE.
- `write_to_file`: Whether to write the paths of non-skipped data files to a text file, as a JSON array of strings, instead of reading and querying the actual files. This setting is used during FPR testing as it allows

comparison between the files returned by the regular and BF-EDS Iceberg implementations.

- `skip_reading_parquet`: Whether to skip reading the actual data files once manifest file scans are completed. Used during evaluation to prevent regular Iceberg from reading Parquet files. While similar to the `write_to_file` setting, `write_to_file` incurs a large performance penalty due to its I/O operations.

During a query all manifest entries are scanned. Sequentially, for each of the columns in the query token dictionary, the associated Bloom filter is loaded and queried using the query token. The query predicate can be replaced by the Bloom filter query result. Dependant on the full query, the file the manifest entry references can either be safely skipped, or one or more remaining query tokens have to be applied. Listing 13 shows a query where if `value_1` is larger than 10 the second predicate can be skipped, whereas Listing 14 shows a query where, in this case, the second predicate has to be evaluated. If no Bloom filter exists for the requested column, the file cannot be skipped as it might contain relevant data.

Listing 13 SQL query which can skip evaluating the second predicate if the first predicate is `false`.

```
SELECT COUNT(*) FROM table WHERE value_1 < 10 AND value_2 < 20;
```

Listing 14 SQL query which has to evaluate the second predicate if the first predicate is `false`.

```
SELECT COUNT(*) FROM table WHERE value_1 < 10 OR value_2 < 20;
```

5.6 Range Mapping

Binary interval trees work with unsigned numerical ranges. Using these trees with other data types requires mapping values to the unsigned integer domain. This section describes the process of mapping signed integers, strings and NULL values to the unsigned integer domain.

5.6.1 Signed Integers

Mapping signed integers to the unsigned integer domain is trivial. Listing 15 shows the code responsible for mapping signed integers to unsigned integers.

Listing 15 Code for mapping signed integers to unsigned integers. The signed value is cast to an unsigned 64 bit integer. If the signed value is negative, the most significant bit will be set. An XOR operation turns this bit off. If the signed value is positive, its most significant bit is turned on by the same XOR operation. This effectively maps the negative domain $[-2^{63}, 0]$ to the first 63 bits of the unsigned integer. The positive signed integer domain is mapped to $[2^{63}, 2^{64}]$.

```
return {
  static_cast<uint64_t>(r.min) ^ 0x8000000000000000ULL,
  static_cast<uint64_t>(r.max) ^ 0x8000000000000000ULL,
};
```

The signed integer is first cast to an unsigned 64 bit integer. Signed integers use the most significant bit to store their sign. A negative integer has its most significant bit unset, whereas a positive integer has its most significant bit set. After the cast a bitwise XOR operation is performed with the hexadecimal representation of a 64 bit unsigned integer with just the most significant bit set. This XOR maps negative integer values to lower unsigned integer values than positive integer values (the most significant bit will be unset by the XOR). This maintains the relative ordering of the signed values in the unsigned integer domain. The entire 64 bit signed integer domain fits within a 64 bit unsigned integer.

5.6.2 Strings

Mapping strings to the unsigned integer domain is less trivial. String ordering is often done using lexicographical ordering, which determines the ordering of two strings by comparing them letter-by-letter. When using binary interval trees this approach is not possible. Instead, we convert strings to numerical values using the equation in Figure 5.4.

$$\sum_{i=0}^{m} (s[i] - 97) \cdot b^{m-i}$$

Figure 5.4: String to unsigned integer mapping equation where m denotes the maximum prefix length of the string taken into account, s represents the to-be-converted-string and b is a base which determines the weight of characters in the string.

In this equation m denotes the maximum string prefix length taken into account during mapping, s is the to-be-converted string and b is a base which determines the weight each character contributes to the final numerical value. The C++ character code of the character at index i is reduced by 97, mapping 'a' to 0. This value is subsequently multiplied by b raised to a power which decreases as the character index increases. This way, earlier characters weigh orders of magnitude more compared to later characters. Using this approach a long string like 'azzzzzzzzz' can be mapped to a smaller numerical value than a much shorter string like 'b'. Figure 5.5 shows how the weight of a character drops off exponentially as its index in the string increases, and how increasing b exacerbates this effect.

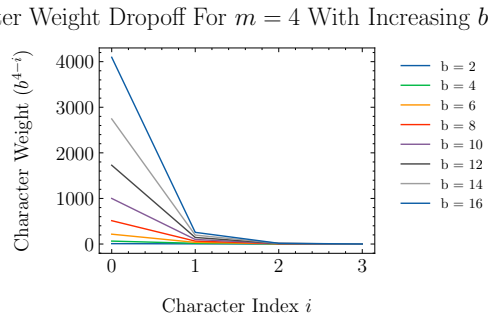


Figure 5.5: Character weight drops off exponentially as letter index in the string increases. Increasing b exacerbates this effect, allowing for longer strings to be mapped with perfect accuracy.

A combination of m and b is said to have perfect accuracy if the absolute lexicographical ordering of strings up to length m is maintained in their mapped numerical counterparts. Perfect accuracy is tested by comparing the strings a $\parallel \mathrm{z}^{m-1}$ and b. If a $\parallel \mathrm{z}^{m-1}$ has a lower mapped numerical value than b, the parameter set has perfect accuracy. Achieving perfect accuracy is challenging. It requires a string composed of a single low-value character, b, to map to a higher numerical value than a longer string which is composed of high-value characters. A careful balance between m and b is required. Increasing both too much overflows 64 bits of space.

Figures 5.6 and 5.7 show how the strings 'azzz' and 'b' are mapped to an integer with parameters $(m = 2, b = 2)$ and $(m = 2, b = 27)$ respectively. The constant b can be lowered to allow for higher values of m without overflowing a 64 bit unsigned integer. Lowering this constant reduces accuracy leading to strings like 'azzzzzzzzz' mapping to a higher value than 'b'. String range mapping, and specifically perfect accuracy pairs, are evaluated in Section 6.5.

$$
\begin{aligned}
&azzz \\
&(97-97)\cdot 2^{4-0} + (122-97)\cdot 2^{4-1} + (122-97)\cdot 2^{4-2}\cdot(122-97)\cdot 2^{4-3} \\
&= 0\cdot 2^4 + 25\cdot 2^3 + 25\cdot 2^2\cdot 25\cdot 2^1 \\
&= 0 + 200 + 100 + 25 \\
&= 325
\end{aligned}
$$

$$
\begin{aligned}
&b \\
&(98-97)\cdot 2^{4-0} \\
&= 1\cdot 2^4 \\
&= 16
\end{aligned}
$$

Figure 5.6: Mapping the strings 'azzz' and 'b' to an integer using parameters $m = 4$ and $b = 2$. Parameter b is too small to assign sufficient weight to earlier characters in the string, leading to 'b' having a lower numerical value than 'azzz'. This leads to numerical ordering which differs from the strings lexicographical ordering.

$$
\begin{aligned}
&azzz \\
&(97-97)\cdot 27^{4-0} + (122-97)\cdot 27^{4-1} + (122-97)\cdot 27^{4-2}\cdot(122-97)\cdot 27^{4-3} \\
&= 0\cdot 27^4 + 25\cdot 27^3 + 25\cdot 27^2\cdot 25\cdot 27^1 \\
&= 0 + 492075 + 18225 + 675 \\
&= 510975
\end{aligned}
$$

$$
\begin{aligned}
&b \\
&(98-97)\cdot 27^{4-0} \\
&= 1\cdot 27^4 \\
&= 531441
\end{aligned}
$$

Figure 5.7: Mapping the strings 'azzz' and 'b' to an integer using parameters $m = 4$ and $b = 27$. This parameter combination correctly maps 'azzz' to a lower numerical value than 'b', preserving the strings lexicographical ordering.

5.6.3 NULL Values

The way NULL values are mapped depends on the semantics associated with them. A NULL value can be seen as being any possible value, or no value at all. Therefore, a data block which contains a NULL value should either always be returned, no matter the query, or only be returned if the range represented by the non-NULL elements intersects with the query range.

If NULL values are seen as being any possible value, a data block which contains a NULL value should always be returned. Inserting the nodes in the sets $\mathbb{P}(0)$ and $\mathbb{P}(T)$, where T is the largest possible 64 bit unsigned integer value, into the Bloom filter will make any query range intersect with the block range. This approach can lead to leakage, as all data blocks containing NULL values will have the same Bloom filter bitset. Adding random elements to the bitset can help reduce this leakage. Seeing NULL values as being no value, requires no changes. BF-EDS inherently ignores NULL values when generating ciphertexts.

Chapter 6

Evaluation

This chapter contains an extensive evaluation of BF-EDS using both system-level and microbenchmarks.

6.1 Preliminaries

All microbenchmarks use Googlebenchmark[30], a widely-used microbenchmarking library. Googlebenchmark uses high resolution timers and tries to mitigate CPU frequency scaling and warmup effects by dynamically adjusting the number of benchmarking iterations. This results in reliable and reproducible benchmark results. Extensive templating support allows the same benchmark to be run with many different parameters, including different Bloom filter types and encryption methods. Benchmarks which generate random ranges use a set list of 100 randomly generated seeds, attached in Appendix 9.1. The keys used in our keyed HFs are attached in Appendix 9.2. All benchmarks are run on an Apple MacBook Pro with an M1 Pro CPU and 32GB of RAM. During benchmark runs no applications beside the benchmarking software are running on the device.

6.2 Binary Interval Trees

As part of research question 1a, various binary interval tree implementations are evaluated. The runtime of these implementations is not crucial, as all of these functions are run on the client side. Additionally, these functions are run orders of magnitudes less frequently compared to server-side functions.

6.2.1 Node ID Calculation

For various tree heights the duration to calculate the ID of the leftmost leaf node is measured. This node is selected as it is present in all trees and calculating the ID of a leaf node takes the longest (longest binary label). Figure 6.1 shows the duration to calculate the node ID with increasing tree heights. As expected, this runtime increases linearly with the tree height. The current complexity of this implementation is $\Theta(h)$. Improving this complexity further is not possible due to the iterative nature of the ID calculation process.

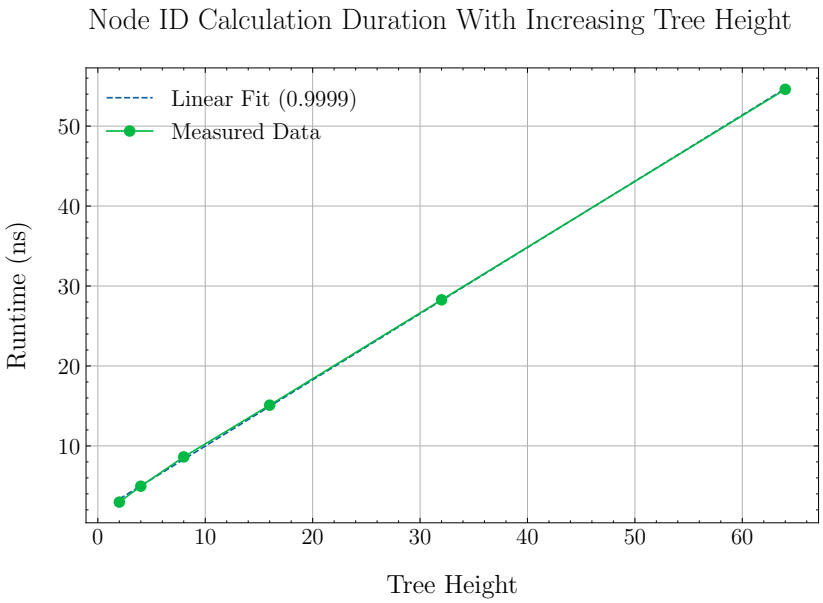


Figure 6.1: Node ID calculation duration for the leftmost leaf node with increasing tree heights. Runtime increases linearly with tree height.

6.2.2 Coverage Set Calculation

For various tree heights the coverage set of the leaf node '0' is calculated. The coverage set size scales linearly with the tree height. A linear relation between the two is expected.

Figure 6.2 shows that this is the case (RMS $3.10 \cdot 10^{-10}$). In some cases the performance scales sublinearly. Given a tree with height h the algorithm always performs h calculations. Therefore, both the runtime upper and lower bounds are fixed, which results in a complexity of $\Theta(h)$. Figure 6.3 compares the runtime duration of calculating the coverage set for the tree leaf '0' between the naive

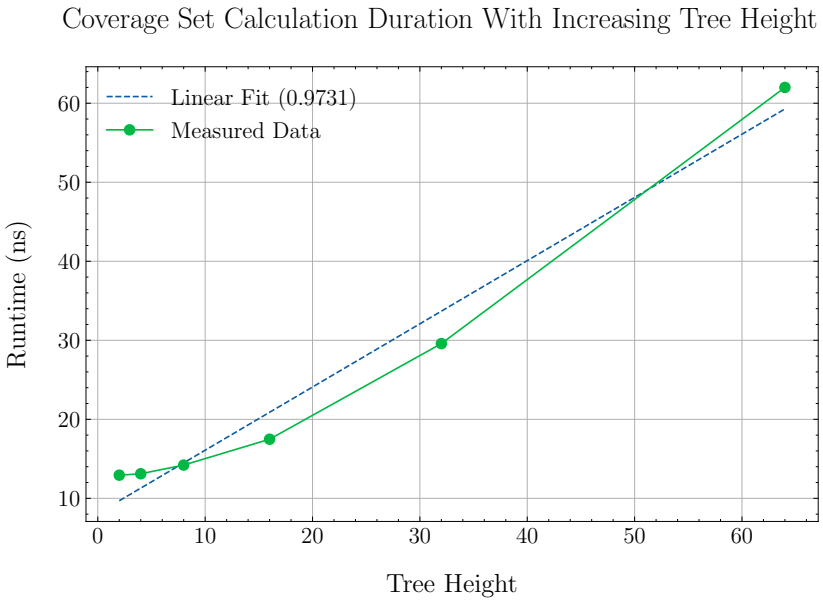


Figure 6.2: Coverage set calculation duration for the leaf node '0' with increasing tree heights.

implementation, which calculates the node ID each iteration, and the optimized implementation, which iteratively updates the node ID while moving down the tree. As expected, the optimized implementation performs significantly better, with the naive implementation scaling quadratically (RMS $3.4 \cdot 10^{-11}$).

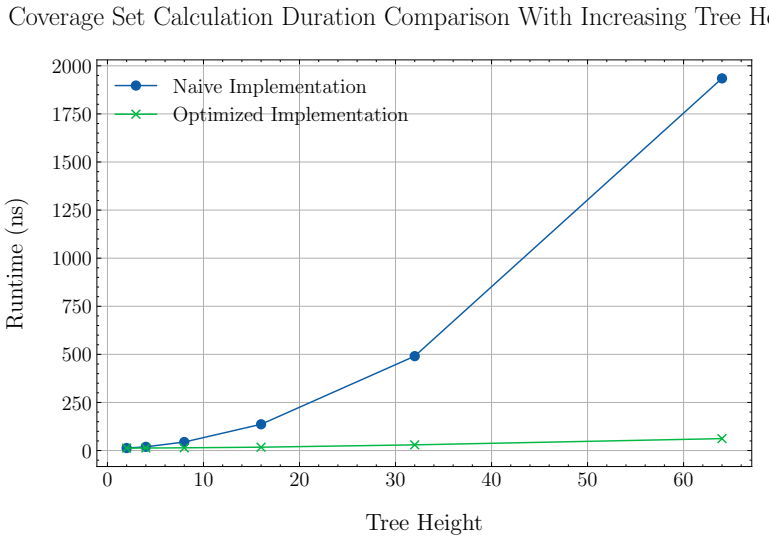


Figure 6.3: Coverage set calculation duration comparison between the naive and optimized implementations for the leaf node '0' with increasing tree heights.

6.2.3 Minimum Coverage Set Calculation

For a number of tree heights, up to 64, the MCS is calculated for 10,000, pre-seeded, randomly generated ranges. Figure 6.4 shows the runtime results for this benchmark. The top down and bottom up approaches are similar in performance, with the bottom up approach being slightly faster in trees with a height less than 32. Figure 6.5 shows the number of expanded nodes for each implementation at various tree heights. The top down approach consistently expands fewest nodes. Due to the ordered nature of the tree the top down approach is performing a form of binary search via BFS.

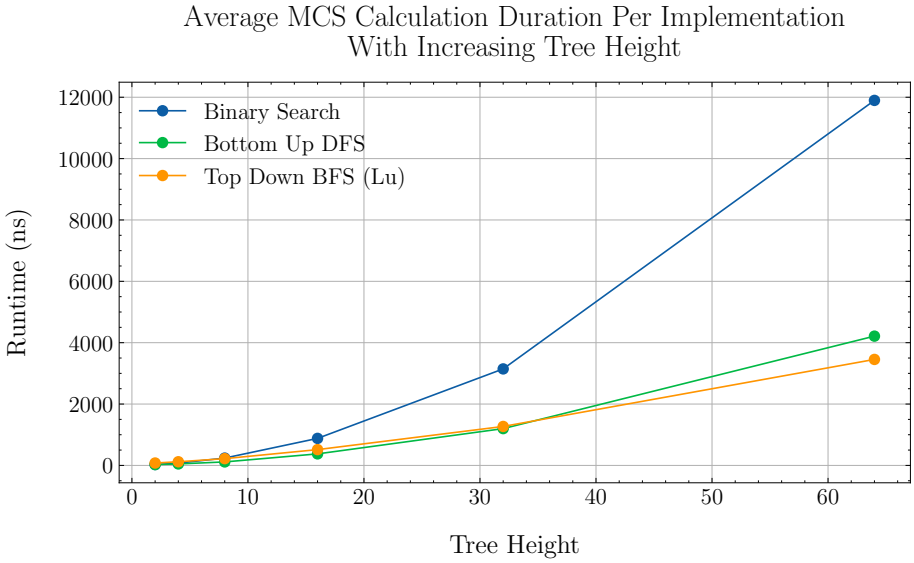


Figure 6.4: Average MCS calculation duration per implementation with increasing tree height. Shown duration is average of 10,000 random range MCS calculations.

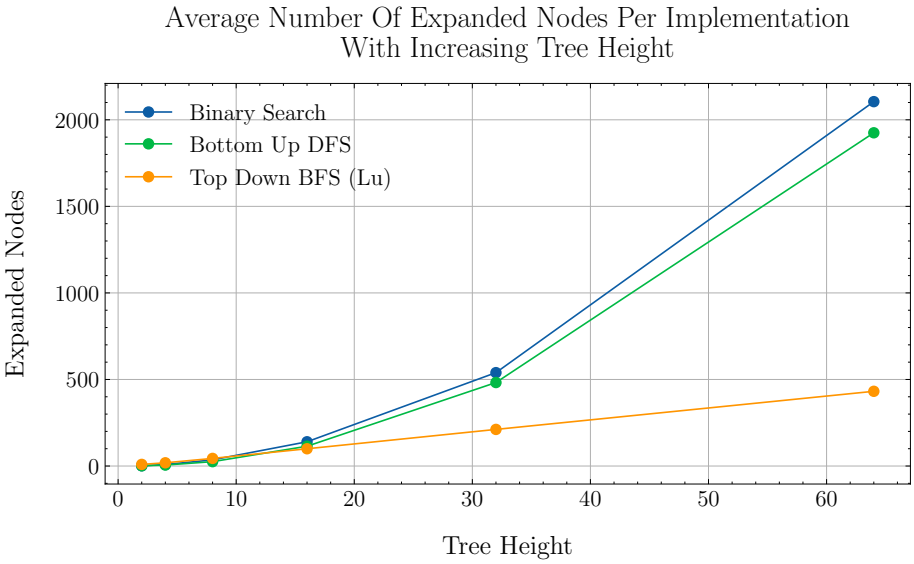


Figure 6.5: Average number of expanded nodes per implementation. Average taken over 10,000 random range MCS calculations. An expanded node is a node for which the coverage is calculated and checked.

The performance difference between the top down and bottom up approaches is small, whereas the difference in expanded nodes is large. At tree height 64 the bottom up approach expands 3.8x more nodes compared to the top down approach, whereas the top down approach performs just 1.2x better when looking at runtime. This stark and unexpected difference can be partly attributed to the queue which the top down approach uses. Profiler results show that the queue expansion and insertions operations occupy around 10% of the total runtime. Additionally, the bottom up approach offers simpler and more linear code, resulting in a smaller than expected performance difference given the large difference in expanded nodes.

The binary search implementation performs very poorly. Figure 6.5 shows this is not due to a much larger number of node expansions. Both the binary search and bottom up implementations expand a similar number of nodes. Most likely, this performance difference stems from the non-linear memory access patterns and high amount of branching this implementation introduces.

6.3 Hash Functions

This section evaluates multiple HFs on both their hashing performance and uniformity.

6.3.1 Performance

The performance of the HF we use plays a significant role in the usability of BF-EDS in real-world systems. Computing hashes quicker leads to lower query durations and a higher system throughput. When using XOR encryption for our Bloom filters the query duration is dominated by hash computations. If AES encryption is used the HF performance plays a much smaller role in query performance.

Figure 6.6 shows the time it takes various HFs to hash keys with increasing sizes. HighwayHash and SipHash perform the best, with SipHash outperforming HighwayHash with keys smaller than 128 bytes. This is unexpected as HighwayHash claims to be consistently faster than SipHash for all input sizes[2]. Do note that SipHash only outputs a 64 bit hash, whereas HighwayHash outputs a 256 bit hash. AES128 and AES256 are not HFs but can be used as one by encrypting the key and then using the first 128 or 256 bits of the ciphertext as a hash.

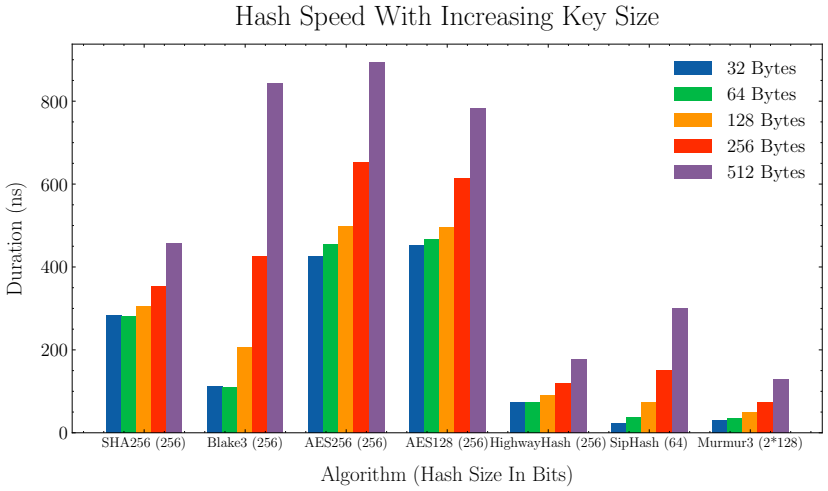


Figure 6.6: Hashing duration for various HFs with increasing key sizes. Overall HighwayHash and SipHash perform best, with SipHash outperforming HighwayHash on inputs smaller than 128 bytes.

6.3.2 Uniformity

A perfectly uniform HF hashes any value v to one of n buckets with probability $1/n$. When used in a Bloom filter, uniform hashing is important as it reduces the number of collisions, reducing the FPR of the filter. In our scheme we use three HFs: H_k for Bloom filter insertions and F_1 and F_2 for Bloom filter XOR encryption. For all three HFs non-uniform hashing leads to lesser security and possibly more leakage. A non-uniform hashing H_k leads to non-uniform bit distribution. This makes it easier for an adversary to infer which inputs correspond to specific bits being set. By observing the distribution of set bits an adversary may be able to infer information about the plaintext key used in H_k, as well as the range associated with the Bloom filter. In case F_1 or F_2 exhibit poor uniformity the resulting encrypted Bloom filter will lack sufficient randomness. This increases the likelihood an adversary could infer the plaintext bits, even before a query decrypts these bits.

To test the uniformity of HFs a simple benchmark is used. This benchmark hashes n random values to b buckets, and compares the distribution of values over these buckets with a perfectly uniform distribution (n/b values per bucket). The significance of the results is evaluated using a χ^2 test. For each HF 50 runs are performed. Each run $100{*}buckets$ randomly generated numbers are inserted, using a different predefined seed each run. The first 50 range generation seeds given in Appendix 9.1 are used. Figure 6.7 shows the resulting mean χ^2 values.

Values which lie closer to $buckets - 1$ indicate more uniform hashing.

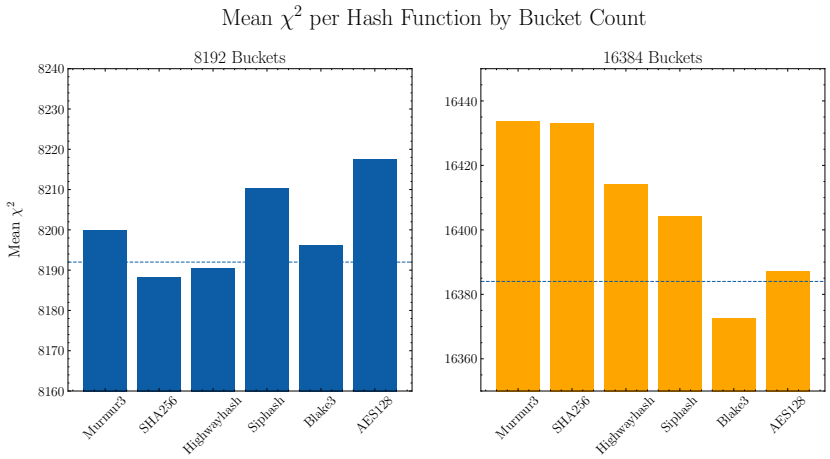


Figure 6.7: Mean χ^2 test results for various HFs hashing to 8192 and 16384 buckets over 50 runs. A mean χ^2 value closer to $buckets - 1$ indicates more uniform hashing. Overall HighwayHash performs best when taking both bucket counts into account.

A statistical p test is used to determine if the resulting distribution in any benchmark run deviates significantly from the expected uniform distribution. In this case, a p value below 0.05 indicates deviation. Figure 6.8 shows the number of runs, for each HF, in which the resulting distribution deviated significantly from the expected uniform distribution. A lower number of deviations indicates more consistent uniform hashing. HighwayHash performs best both in terms of mean χ^2 value and number of deviations.

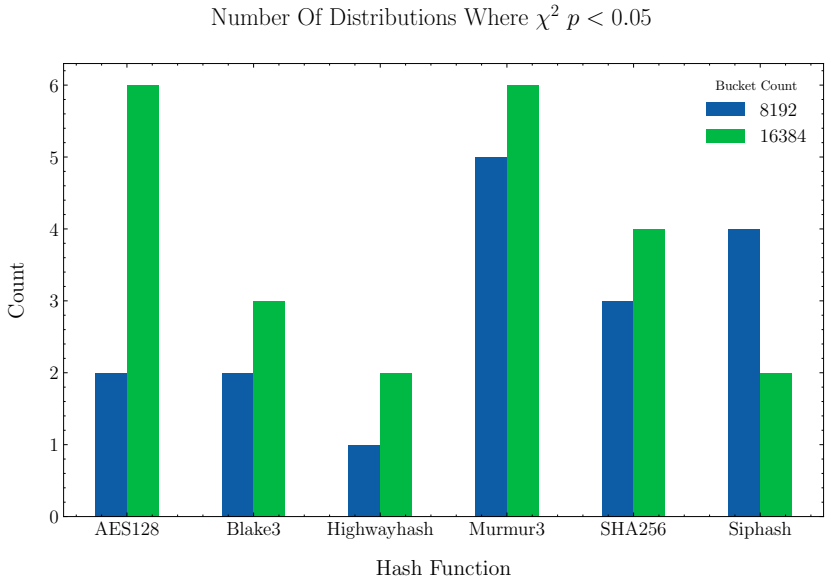


Figure 6.8: Number of runs out of 50 total runs with a p value below 0.05, indicating a significant deviation from the expected uniform distribution. HighwayHash has the lowest number of deviations with both bucket counts, indicating very consistent uniform hashing performance.

6.4 Bloom Filters

6.4.1 Split Block Bloom Filter Comparison

As discussed in Section 5.3.3 two implementations of the SBBF exist. The original implementation uses just the upper 64 digest bits, whereas the adapted implementation uses the full 256 bit digest. For this second implementation two variants are implemented. The first simply splits the 256 bit digest into eight 32 bit values and the second additionally performs multiply-shift hashing on these eight values.

Figure 6.9 shows the spread of false positives these variants returned when querying a table containing 1 million files. The variant using the full 256 bit digest instead of multiply-shift hashing the first 64 bits returns, on average, 6.3x more false positives. Looking at the two spreads it is clear that the full 256 bit variant performs significantly worse in all quartiles as well as in its outliers. This significantly worse false positive is surprising. When using the full 256 bit digest one would expect higher entropy compared to simply multiplying the first 64 bits with eight odd constants. Multiply-shift hashing yields eight distinct and

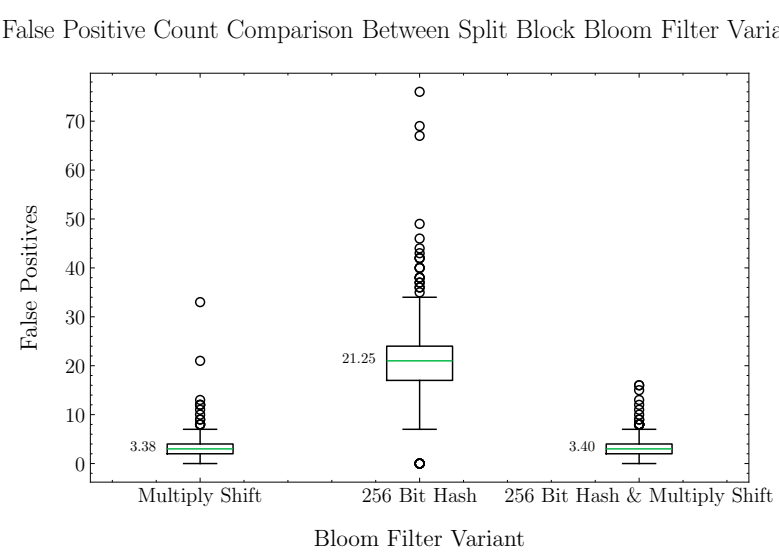


Figure 6.9: Comparison of false positives spread between the regular, 256 bit and 256 bit multiply-shift hashing SBBF variants. The 256 bit variant using no multiply-shift hashing performs significantly worse compared to the other two variants.

independent values, whereas simply splitting the 256 bit digest into eight parts does not yield truly independent values. This results in more collisions within blocks. Using the full 256 bit digest in combination with multiply-shift hashing makes the resulting hashes more unpredictable, resulting in FPRs comparable to the original implementation. Additionally, using a different base in each lane increases security.

Performance wise both variants are very similar. The SBBF variant using the full 256 bit digest is, on average, slightly faster as the 256 bit digest comes spread over eight lanes and as such does does not require an additional broadcast operation. Figure 6.10 shows the runtime duration of performing a single Bloom filter query for both the 64 bit and 256 bit variants.

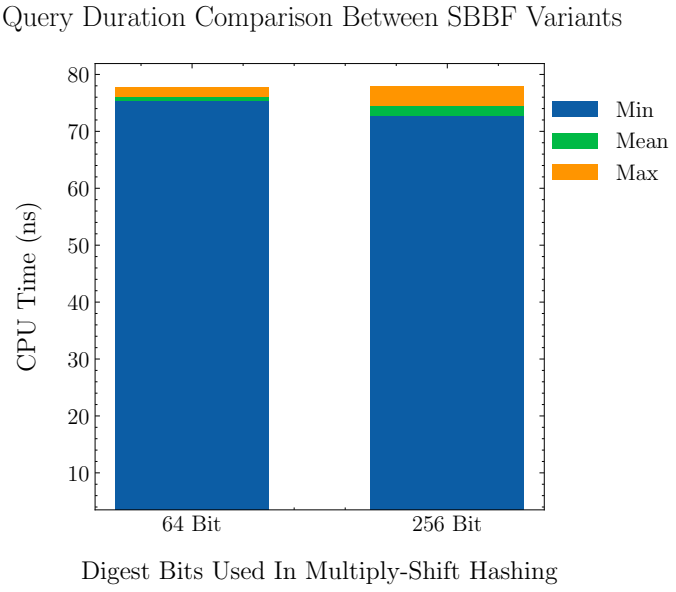


Figure 6.10: Single query duration comparison between the regular SBBF using just the first 64 bits of a digest, and the SBBF variant using the full 256 bits of the digest. Both variants use multiply-shift hashing to generate eight sufficiently independent hashes.

6.4.2 Bloom Filter Comparison

In Sections 2.3 and 5.3 three Bloom filter implementations are discussed. Benchmarks are used to determine which implementation performs best. Query duration is the most important factor in determining the best implementation, as a faster Bloom filter results in faster queries. Figure 6.11 shows the average duration of a single Bloom filter query (using a query token). We benchmark both cases where the query range intersects with the Bloom filter range, as well as cases where the query range *does not* intersect with the Bloom filter range. The bitset size m and HF count k of each implementation are set to values for which the Bloom filter averages less than 1% FPR over 1000 randomly generated test ranges. The register blocked Bloom filter requires a much higher m to achieve this average FPR.

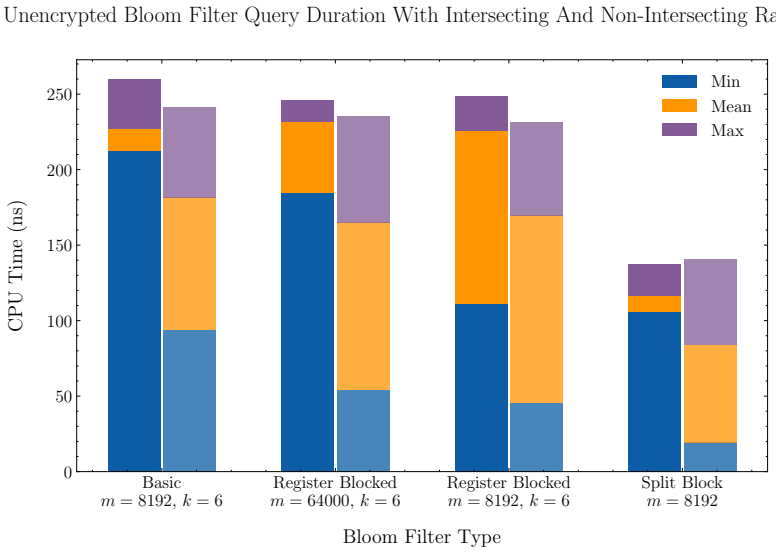


Figure 6.11: Single Bloom filter query duration comparison between three Bloom filter variants. All Bloom filters are unencrypted. The parameters m and k are chosen such that the Bloom filter achieves an average FPR of 1%. We benchmark both cases where the query range intersects with the Bloom filter range, as well as cases where the query range *does not* intersect with the Bloom filter range.

The SBBF performs the best. Overall the minimum and mean query duration with intersecting ranges is a lot lower than with non-intersecting ranges. This is due to early termination as soon as two intersections are found. With non-intersecting ranges all sub-tokens in the query token have to be evaluated to ensure there are not two intersections. The register blocked Bloom filter performs worse than expected. This can be attributed to the HF computation cost being the same for the basic and register blocked Bloom filters. The faster bitset level operations are overshadowed by the hash computation overhead. In comparison, the SBBF only has to compute a single hash, using multiply-shift hashing to compute the required eight hashes. This leads to, on average, 2.1x faster queries compared to the two other implementations.

Figure 6.12 shows the duration of a single query when using XOR encryption (as explained in Section 4.3.1). The SBBF outperforms the other implementations by around 2.5x. Each query processes a single query token, which consists of 128 sub-tokens each containing k hashes. Theoretically a basic Bloom filter, in the worse case, performs 768 decryptions (given all hashed indices are unique and no early termination can be performed). In comparison, the SBBF requires

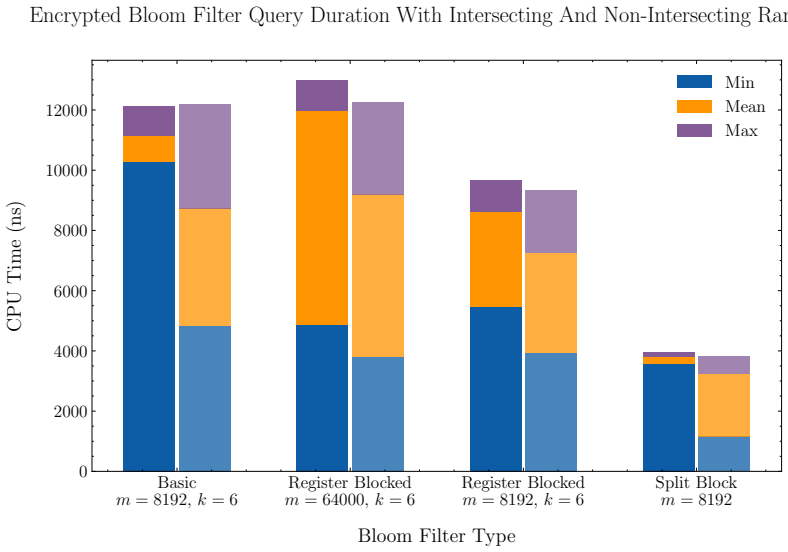


Figure 6.12: Query duration comparison between XOR encrypted Bloom filter variants. We benchmark both cases where the query range intersects with the Bloom filter range, as well as cases where the query range *does not* intersect with the Bloom filter range. The SBBF variant significantly outperforms the other variants.

at most 32 decryptions (32 blocks of 256 bits). If the theoretical maximum number of decryptions are performed the SBBF outperforms the basic Bloom filter by 24x.

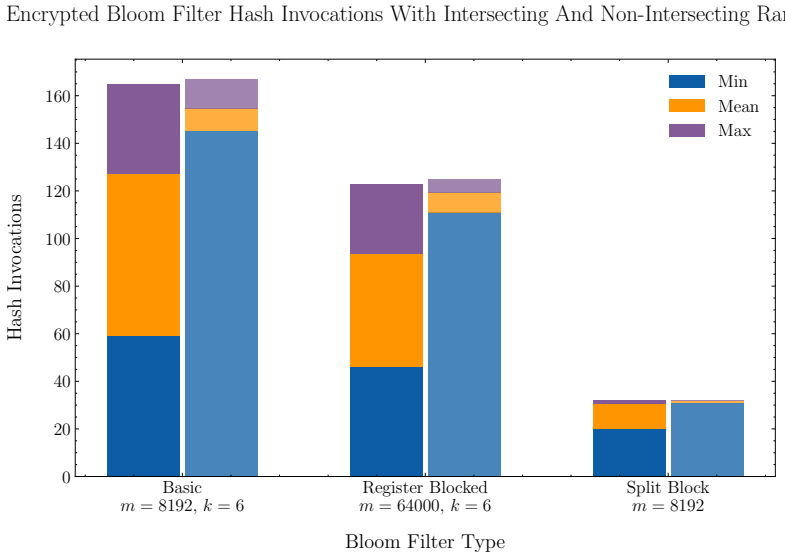


Figure 6.13: Comparison of number of performed XOR decryptions between XOR encrypted Bloom filter variants. The SBBF significantly outperforms the other two variants due to its block-by-block encryption which uses larger blocks compared to the register blocked Bloom filter.

Figure 6.13 shows the number of XOR decryptions each Bloom filter variant performs during a single query. Both with intersecting and non-intersecting queries the basic Bloom filter performs far less decryptions than the theoretical maximum. With intersecting ranges it performs just 16.9% of the 768 possible decryptions, with non-intersecting ranges this percentage lies slightly higher at 18.8%. This low number of computed hashes is the result of early termination and a low FPR.

Figure 6.14 expands on this explanation. Shown is the number of pairs, of which there are k in each sub-token, which intersect with the basic Bloom filter set, both with intersecting and non-intersecting ranges. If one of the pairs is not in the Bloom filter, the rest of the pairs in the sub-token can be skipped. All k pairs in a sub-token need to be present in the Bloom filter for the full sub-token to intersect. The bar at zero shows that for the majority of sub-tokens the first pair is not present in the Bloom filter, meaning the remaining $k - 1$ pairs can be skipped. For 128 sub-tokens this leads to an average of 135 computed

hashes, as shown in Figure 6.13. Reducing the Bloom filter bitset size m leads to more intersections and less early termination, however, this comes at the cost of significantly more incorrect intersections.

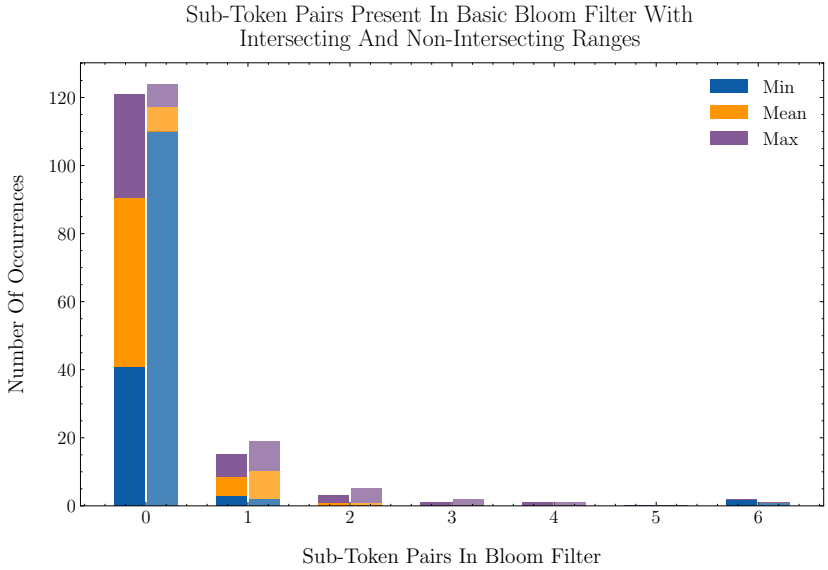


Figure 6.14: Number of matching pairs when querying a basic Bloom filter, k pairs per sub-token, with intersecting and non-intersecting ranges. The first pair in most sub-tokens is not present in the Bloom filter, allowing the remaining $k-1$ pairs to be skipped. This leads to a much smaller number of HF operations than expected.

6.4.3 Encryption Methods Comparison

Bloom filters can be left in plaintext or encrypted using XOR or AES encryption. The SBBF has implementations of both encryption methods. This section contains a comparative evaluation of the performance of these encryption methods using the SBBF. For each range generation seed 100 ranges are generated, for a total of 10,000 ranges. The benchmark is run twice. First with all Bloom filter ranges intersecting with the query range, second with no Bloom filter ranges intersecting. The same benchmark is run for each of the three encryption methods.

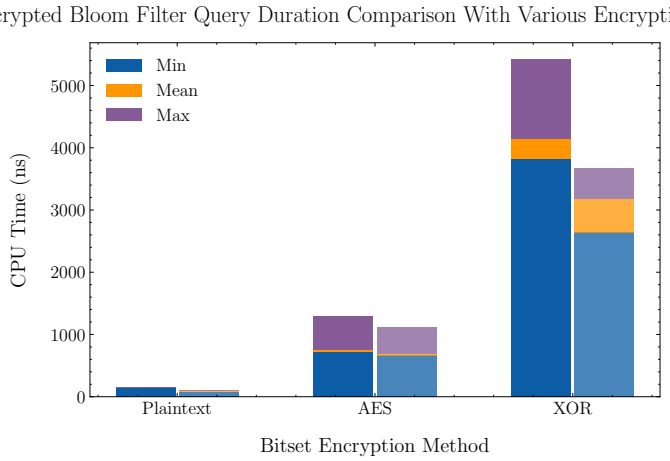


Figure 6.15: Query duration with various encryption methods applied to the SBBF. Per encryption method the benchmark is run twice with 10,000 randomly generated ranges. First with only intersecting ranges, second with only non-intersecting ranges (left and right bars respectively).

Figure 6.15 shows the benchmark results. Plaintext encryption - no encryption - performs the best, as one would expect. On average, queries on the AES encrypted Bloom filter are 5.5x faster compared to the XOR encrypted Bloom filter when querying a non-intersecting range, and 4.6x faster when querying an intersecting range. A larger drop in query duration when querying intersecting ranges can be seen when using XOR encryption, compared to AES encryption. This is due to the fixed decryption overhead of the AES encryption method. The XOR encryption method performs less decryptions when querying intersecting ranges, whereas the AES method still performs one decryption.

6.5 String Range Mapping

In this section the benchmarking process regarding string mapping and the best performing values for the parameters m and b are discussed.

6.5.1 Perfect Accuracy

As discussed in Section 6.5, a combination of m and b is said to have perfect accuracy if the absolute lexicographical ordering of strings up to length m is maintained in their mapped numerical counterparts. Perfect accuracy is tested by comparing the strings $\mathrm{a} \,\|\, \mathrm{z}^{m-1}$ and b. If $\mathrm{a} \,\|\, \mathrm{z}^{m-1}$ has a lower mapped numerical value than b the parameter set has perfect accuracy. Table 6.1 contains all

perfect accuracy parameter combinations, with the best parameter combination, if perfect accuracy is required, being ($m = 12, b = 30$). With this parameter combination, mapped strings up to length 12 are guaranteed to remain lexicographically ordered in the numerical domain.

m	**b**
4	[26, 64]
5	[26, 64]
6	[26, 64]
7	[26, 64]
8	[26, 64]
9	[26, 64]
10	[26, 60]
11	[26, 42]
12	[26, 30]

Table 6.1: String mapping parameter combinations which yield perfect accuracy for strings up to length m. As m increases the max usable b value drops due to integer overflows, clearly seen by the drop in max b from 64 with $m = 4$ down to 30 with $m = 12$. Above $m = 12$ perfect accuracy is no longer possible.

6.5.2 Benchmarks On Real World Data

For this benchmark the NextiaJD Reddit dataset[49] is used. This dataset contains comments posted on various "Subreddits", similar to message boards, along with metadata relating to these comments. Two columns in this dataset are used in our benchmark: the comments and the comment authors. Comments are written in natural language without any specific patterns and are often longer than the maximum possible m value. The comment authors are generally shorter with a mean length smaller than our maximum m value. Before benchmarking data is preprocessed in two steps. First, each string is converted to lowercase. Second, all non-alpha characters are removed from each string as our string ordering solution exclusively supports alpha characters. Extending support to additional characters using their character codes is trivial but limits the largest value of m due to integer overflow.

Listing 16 shows a dataset sample row before and after preprocessing. Preprocessed strings are lexicographically ordered and exported to be used in the benchmarks. Duplicate rows in both the author and body columns are dropped as there is no 'correct' ordering for duplicate strings. Table 6.2 shows relevant statistics for the preprocessed body and author columns.

Listing 16 Sample row from NextiaJD Reddit comments dataset before and after preprocessing.

```
Author    Body
"eczaz05" "I get the impression that he writes his own bars..."
"eczaz"   "igettheimpressionthathewriteshisownbars..."
```

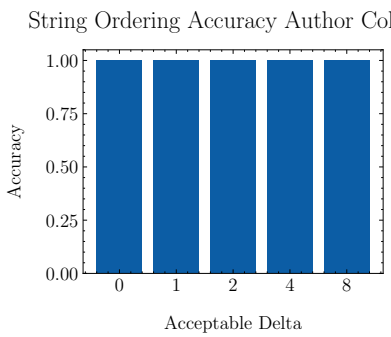
	Row Count	**Min**	**Max**	**Mean**	**Std Dev**
Body	6,886,549	1	9991	123	182.7
Author	633,105	1	20	10	3.7

Table 6.2: Relevant statistics for body and author column string lengths in preprocessed NextiaJD Reddit comments dataset. The mean body column string length is significantly longer compared to the author column. The longest string in the author column can be entirely covered by m, whereas this is not the case for the body column.

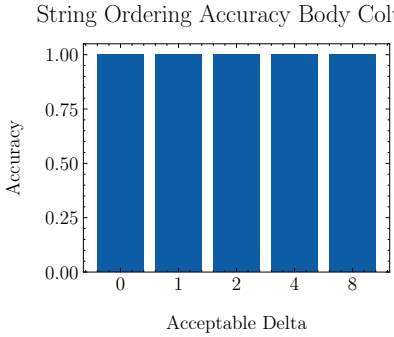
Measuring the accuracy of the mapped strings is done in three steps:

1. All strings in the to-be-benchmarked column are mapped to the numerical domain using the parameters ($m = 12, b = 30$).
2. The list of numerical values is sorted in ascending order.
3. For each string value, the delta between its index in the sorted string list in the sorted numerical list is calculated. If this delta is larger than a set maximum acceptable delta it is counted as an error. Accuracy is calculated using the formula 1 - (error_count / row_count).

The benchmark is run using max acceptable deltas ranging from zero to eight. Figures 6.16a and 6.16b show the ordering accuracy results for these benchmarks. For both columns all five benchmarked delta values resulted in an ordering accuracy of 100%. These results show that the ordered mapped strings are in exactly the same order as their lexicographically ordered counterparts. Additionally, these results show that taking just the first 12 characters into account results in great ordering. Even with a collection of strings where the mean string length is significantly larger than 12.



(a) Accuracy of numerical mapped string ordering on the NextiaJD Reddit dataset author column, with various acceptable deltas. At delta zero accuracy is 100%, indicating all mapped strings remain correctly ordered.



(b) Accuracy of numerical mapped string ordering on the NextiaJD Reddit dataset body column, with various acceptable deltas. At delta zero accuracy is 100%, indicating all mapped strings remain correctly ordered.

Figure 6.16: Accuracy of ordering with mapped strings. All strings mapped to integers using the parameters ($m = 12, b = 30$). Acceptable delta is defined as the maximum delta between the index of a string in the ordered string list compared to the mapped numerical value in the ordered mapped values list. Any delta larger than the acceptable delta is counted as an error.

6.6 DuckDB Iceberg Manifest Querying Performance

This section evaluates the manifest scanning performance using BF-EDS. Determining FPR is done by comparing the data file paths returned by our BF-EDS Iceberg implementation with those returned by regular Iceberg. All files returned by regular Iceberg **must** be present in the files returned by BF-EDS. Any additional files count as false positives. When evaluating performance Parquet file reading is disabled for both implementations, as we only evaluate the manifest querying performance. Signed integer ranges are used during evaluation as they are natively supported by Iceberg, whereas unsigned integer ranges require use of custom types.

6.6.1 Querying Increasing Range Sizes

To obscure information about the range being queried, as well as the number of relevant files, it is important for queries with varying range sizes to take similar amounts of time. An approach which decrypts and queries files on the server

leaks information about the number of relevant files through its query duration. Figure 6.17 shows that queries at all query range sizes take about the same amount of time. A query which matches none of the files on the server takes about the same amount of time as a query which matches every single file. The transfer of relevant files from the server to the client does leak some information. Having a slightly higher FPR obscures information about the actually relevant files, as every query returns some irrelevant files, which could throw an adversary off.

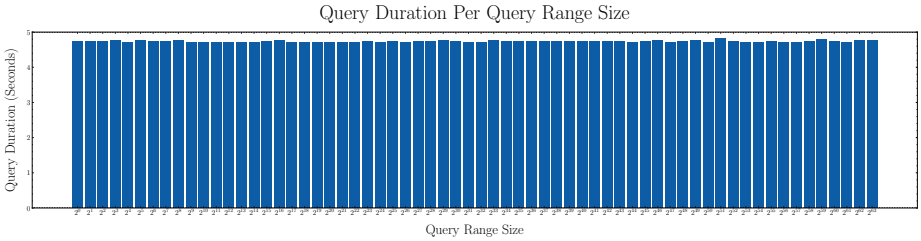


Figure 6.17: Query duration for range sizes from 1 to 2^{63} on a table with 1 million files using AES-encrypted 16384 bit SBBF. While the largest query range size intersects with all files, and the smallest with just a single file, all queries take about the same amount of time. Query duration thus leaks little to an adversary about the queried range size.

6.6.2 Performance With Larger Bloom Filter Bitsets

We are using a SBBF with a fixed k of eight. The only way of influencing the FPR is by adjusting the size of our Bloom filter bitset m. This section evaluates the influence increasing m has on performance.

Figure 6.18 compares the query durations when querying 1 million files using SBBFs with increasing bitset sizes. Increasing the bitset size slows down queries, however, while m doubles each run, the query duration increases far less. Above $m = 8192$ query duration increases faster, however, still far below the linear increase in m. Doubling m from 2^{14} to 2^{15} increases mean query duration by just 17%.

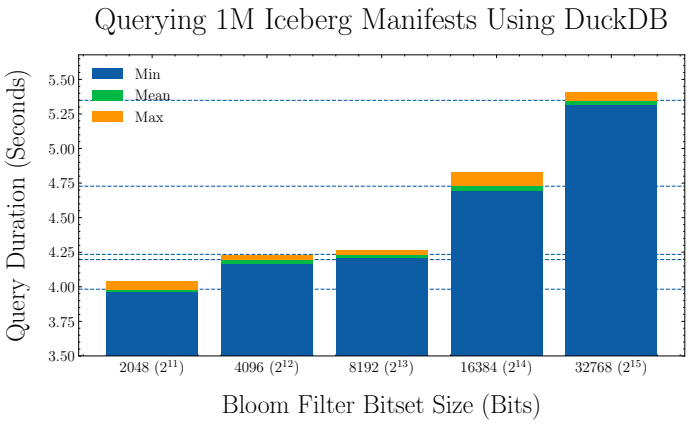


Figure 6.18: Comparison in query duration when querying 1 million files with increasing SBBF bitset size m. Query duration increases sublinearly with linear increasing m. Above $m = 8192$ query duration increases at a higher rate.

Figure 6.19 shows the decrease in FPR as m increases, again using the 1 million file table. The FPR decreases exponentially as m increases. Even with the addition of dummy sub-tokens in the query token the number of false positives can be brought down to near-zero with outliers below 5. This is a significant result as it demonstrates the usability of this scheme for tables with very high file counts, where the number of false positives is much more important than any query duration increase.

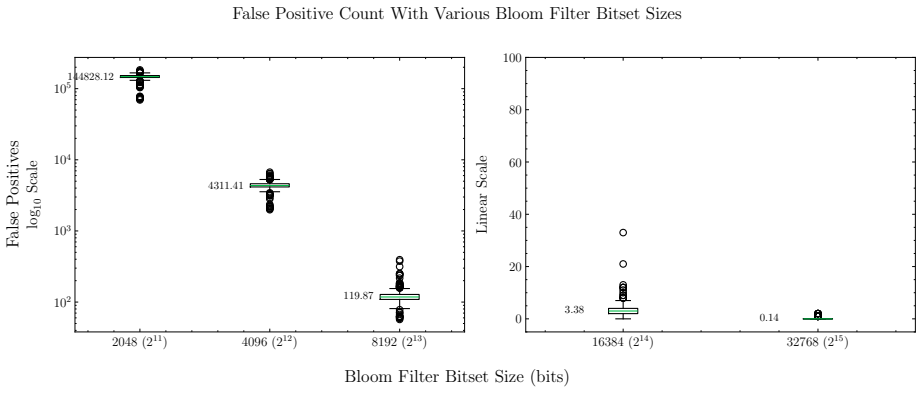


Figure 6.19: FPR when querying 1 million files with increasing SBBF bitset size m.

Increasing m logically increases the size of metadata as well. Figure 6.20 shows manifest file size increases linearly as the Bloom filter bitset size m in-

creases. When adding 2048 bit Bloom filters the manifest file size jumps from 9KB to 470KB, which is more than the expected $9\text{KB} + (1000 \cdot 2048) = 265\text{KB}$. This is due to a combination of Avro's internal block based structure requiring additional sync markers, changes in the schema requiring additional storage and all Bloom filter bitset byte arrays requiring a 1 byte length prefix. Manifest files are limited in size to around 4GB, meaning m can be increased much further to maintain a low FPR, even in tables with orders of magnitude more files.

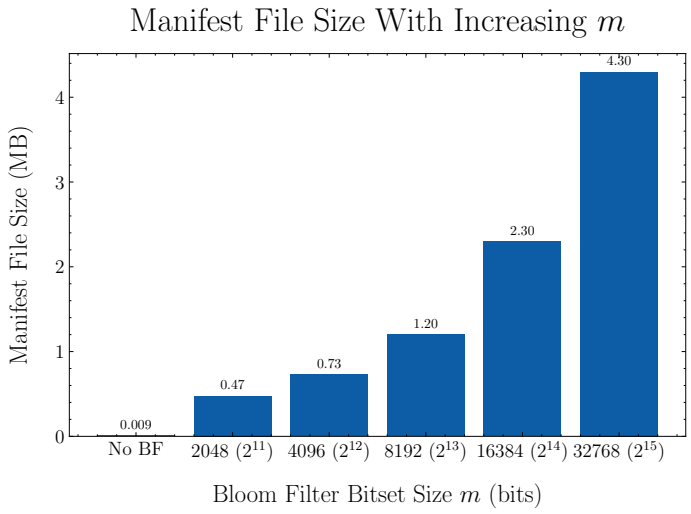


Figure 6.20: Size of individual manifest files containing 1000 manifest entries for various Bloom filter bitset sizes m. Manifest file size increases linearly with m.

6.6.3 Increasing Manifest Batch Size

Each manifest file contains a number of manifest entries. Each entry points to a data file and contains relevant information and statistics about this file. The number of entries per manifest file is determined by the batch size. Increasing the batch size increases the number of entries per manifest file and reduces the number of manifest files to be read. As I/O operations can be slow, increasing the batch size should improve performance.

The same dataset, containing 100,000 files, is converted to an Iceberg table using various batch sizes. For each batch size a separate table is created. With each increase in batch size the resulting table has fewer manifest files. This should reduce the amount of file level I/O operations required when scanning, resulting in faster queries.

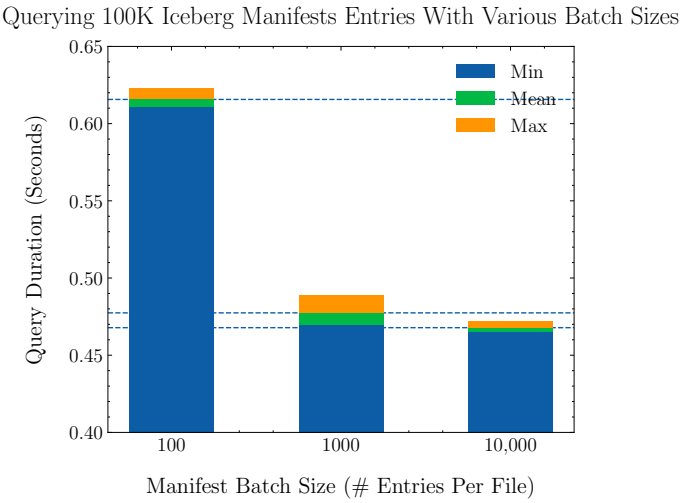


Figure 6.21: Query duration querying 100,000 manifest entries with various batch sizes while using AES-encrypted 16384 bit SBBF. Higher batch sizes result in shorter queries. Diminishing returns with very high batch sizes due to query processing dominating I/O operations and diminishing decrease in manifest file count.

Figure 6.21 shows the mean query duration with various batch sizes. Increasing the batch size improves query performance. Increasing the batch size from 100 to 1000 reduces the number of manifest files by 10x, consequently decreasing query duration by 23%. Repeating the process, using batch size 10,000, decreases the manifest file count to just 10. This yields a decrease in query duration, compared to batch size 1000, of just 3.2%. There are strong diminishing returns. Each 10x batch size increase yields a smaller decrease in manifest file count (100 to 1000 decreases by 900 files, 1000 to 10,000 by just 90). Additionally, higher batch sizes result in larger files. Reading these files requires more memory and potentially leads to increased garbage collection, both of which can cause slowdowns.

6.6.4 Regular Iceberg Performance Comparison

Comparing regular DuckDB Iceberg performance to our BF-EDS implementation is done using the query in Listing 17. This query is run 64 times. Each iteration the query range size is increased to cover all range sizes between 2^0 and 2^{63}. Our BF-EDS implementation uses an AES-encrypted 16384 bit SBBF.

Listing 17 SQL query used to compare regular DuckDB Iceberg and our BF-EDS implementation.

```
SELECT
  COUNT(*)
FROM
  default.data
WHERE
  value BETWEEN query_min AND query_max;
```

Figure 6.22 and Table 6.3 show the benchmark results for three table sizes. Across all tested table sizes the mean overhead is around 40%. With increasing table size the query duration for both the regular and BF-EDS implementations scales linearly. Increasing the table size by 10x increases the query duration around 10x as well.

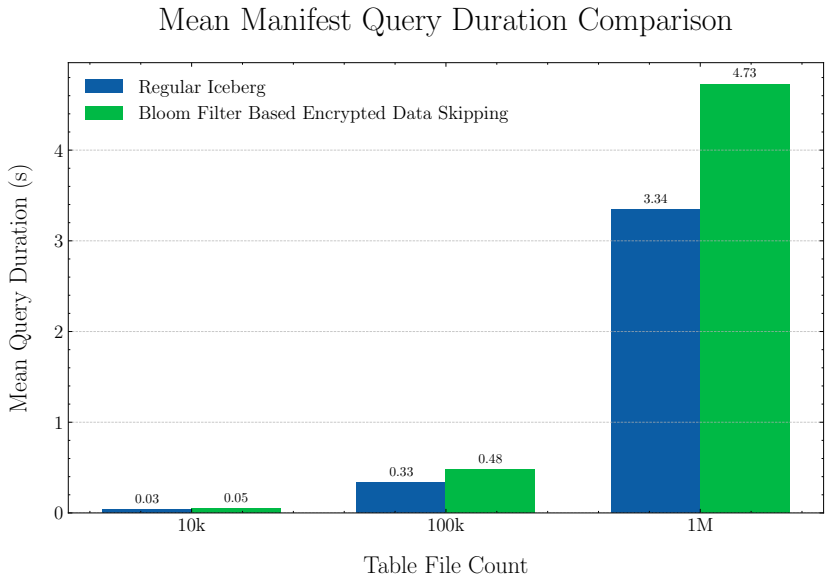


Figure 6.22: Mean query duration comparison between regular DuckDB Iceberg and DuckDB Iceberg using BF-EDS. BF-EDS uses AES-encrypted 16384 bit SBBF. Benchmarked on 64 query ranges covering all range sizes between 2^0 and 2^{63}. Query duration increases linearly with table size for both the regular and BF-EDS implementation.

Table Size	Regular Duration	BF-EDS Duration	Overhead
10k	0,0352	0,0489	39%
100k	0,335	0,477	42%
1M	3,33	4,63	39%

Table 6.3: Mean query duration and overhead when using BF-EDS compared to regular Iceberg for various table sizes. Results obtained by running 64 query ranges covering range sizes from 2^0 up to 2^{63}.

Looking at the overhead per query range size shown in Figure 6.23 we see that the overhead when using BF-EDS decreases as the query range size increases. With a query range size covering the entire domain (2^{63}) the overhead using BF-EDS is just 10%, compared to 40% with smaller query range sizes. The BF-EDS implementation performs the same over all query range sizes, something which is shown in Figure 6.17, whereas the regular implementation performs worse as the query range size increases. This leads to a lower overhead.

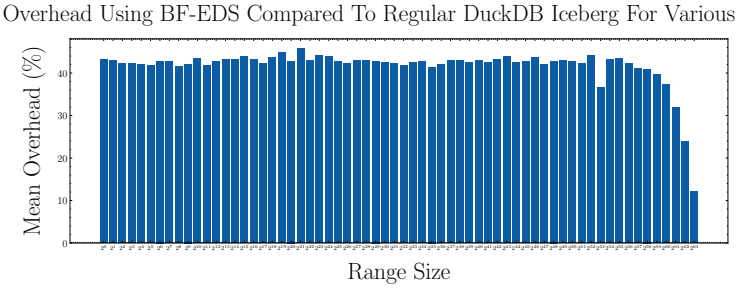


Figure 6.23: BF-EDS overhead compared to regular DuckDB Iceberg as query range size increases from 2^0 to 2^{63}. BF-EDS performance remains stable whereas regular Iceberg performance decreases, leading to a reduced overhead with larger query range sizes.

It is unclear why the regular implementation starts performing worse as the query range size increases, whereas the BF-EDS implementation performance remains consistent. Looking at profiler data, both implementations perform significantly more vector and buffer allocations at higher query range sizes. A possible reason these allocations do not affect the BF-EDS runtime is that the slower decryption operations and Bloom filter lookups mask the allocations. This results in no blocking, whereas The regular implementation might be blocked by these allocations.

6.6.5 TPC-H Performance

TPC-H is a benchmark that evaluates the performance of analytical database systems by simulating a business-oriented query workload. Data is generated using the `tpch` extension in DuckDB. Various scale factors between 1 and 16 are used. Higher scale factors increase the amount of TPC-H data that is generated. The generated data is copied into Parquet files, with each file containing 250 rows. With the largest scale factor of 16 this results in 96 million rows spread over more than 383,000 files.

A modified version of TPC-H query 6 is used to evaluate performance. Our system does not support aggregation operations like `SUM` and `JOIN`, which most TPC-H queries use. Floating point range queries are also not supported. All floating point values in the table are converted to integers using 100x multiplication. This can be done without loss of information as the original column type is `DECIMAL(15, 2)`, indicating only two decimals of precision. Dates are converted to unix timestamps and stored as 64 bit integer values (milliseconds since epoch). The date predicates in the query are converted using the same method. The standard TPC-H query 6 and the modified version used in our evaluation are shown in Listing 6.24.

```
SELECT
  SUM(l_extendedprice * l_discount)
      AS revenue
FROM
  lineitem
WHERE
  l_shipdate >= CAST('1994-01-01' AS
       date)
  AND l_shipdate < CAST('1995-01-01'
       AS date)
  AND l_discount BETWEEN 0.05 AND
      0.07
  AND l_quantity < 24;
```

(a) TPC-H query 6 copied from DuckDB `tpch` extension.

```
SELECT
  COUNT(*)
FROM
  default.lineitem
WHERE
  l_shipdate >= 757382400000
  AND l_shipdate < 788918400000
  AND l_discount BETWEEN 5 AND 7
  AND l_quantity < 2400;
```

(b) TPC-H query 6 modified to work in our system. Decimal and Date type values are converted to integers. `SUM` operation converted to `COUNT` operation.

Figure 6.24: Standard TPC-H query 6 taken from the DuckDB `tpch` extension and the modified query used in our evaluation.

The results of this evaluation are shown in Figure 6.25 and Table 6.4. On average, using BF-EDS incurs a 15% overhead compared to regular DuckDB Ice-

berg. This overhead is significantly lower than the mean overhead of 40% in the comparison between regular DuckDB Iceberg and our BF-EDS implementation in Section 6.6.4.

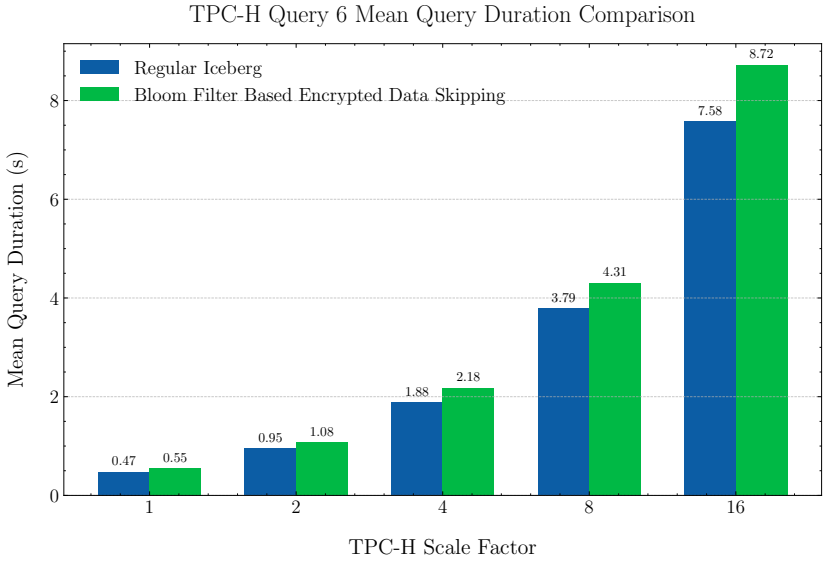


Figure 6.25: TPC-H query 6 mean duration comparison between regular DuckDB Iceberg and DuckDB Iceberg using BF-EDS. Evaluation run with increasing scale factors up to 16. BF-EDS uses AES-encrypted 16384 bit SBBF. On average our BF-EDS implementation incurs a 15% overhead compared to regular DuckDB Iceberg.

Scale Factor	**Regular Duration**	**BF-EDS Duration**	**Overhead**
1	0,473	0,550	16%
2	0,949	1,076	13%
4	1,882	2,174	15%
8	3,788	4,307	13%
16	7,577	8,716	15%

Table 6.4: Mean query duration and BF-EDS overhead when running TPC-H query 6 at scale factors between 1 and 16.

The unmodified TPC-H query 6 performs a summation over all matching rows. Looking at the generated data it appears all rows in the dataset match the predicates in query 6, meaning every row is returned during a query. This can be seen as a query with the largest possible range size (matching all rows/files),

comparable to the range size 2^{63} in Section 6.6.4. Figure 6.23 shows that for large query range sizes, matching 50+% of the tables rows/files, the overhead when using BF-EDS drops significantly. For query range size 2^{63} the overhead is around 10%, comparable to the overhead of 15% seen in the TPC-H evaluation results.

6.7 Order Revealing Encryption Performance Comparison

Both OPE and its generalization ORE can be effectively used for EDS. While both schemes are highly vulnerable to attacks[48][22][31], we provide a comparison with ORE based EDS to demonstrate the practical performance of our solution. This section evaluates and compares the performance of both schemes when used in DuckDB Iceberg. The ORE implementation we use is based on [39] and uses a block-based ciphertext.

6.7.1 Manifest Querying Performance

In this evaluation ORE uses a domain size of 64 bits and a block size of 12 bits. This domain is the same size as our BF-EDS implementation can represent. The block size is taken as the largest block size evaluated in [39], which results in the longest ciphertext and consequently the highest level of security. Using these parameters yields a 3288 bit ciphertext for each numerical value. Ranges are represented using two 3288 bit ciphertexts for a total of 6576 bits. Our BF-EDS implementation uses the same AES-encrypted 16384 bit SBBF used in the comparison between BF-EDS and regular Iceberg in Section 6.6.4. Evaluation is performed by running the query in Listing 17 for ranges with sizes from 2^0 to 2^{63}.

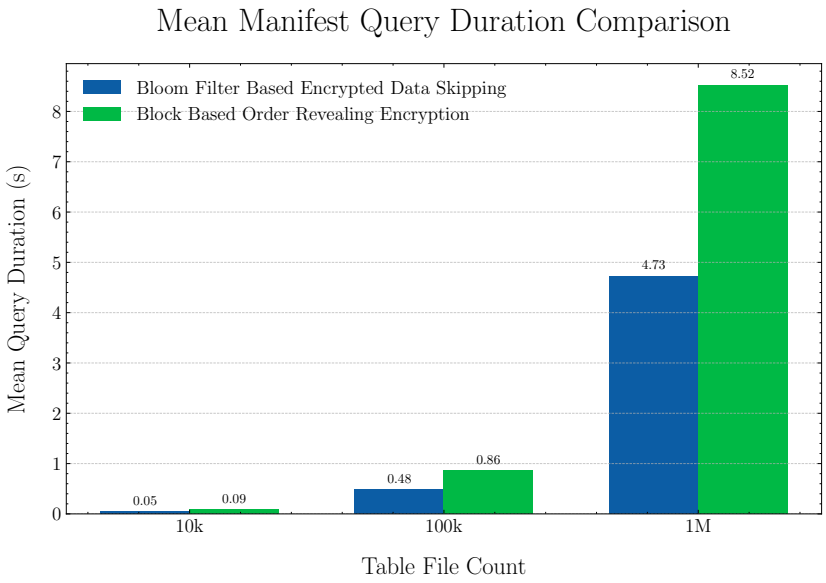


Figure 6.26: Mean DuckDB Iceberg query duration comparison between ORE and BF-EDS using an AES-encrypted 16384 bit SBBF. BF-EDS is 1.8x faster compared to ORE for each of the evaluated table sizes.

Figure 6.26 shows the mean query duration in DuckDB Iceberg for both schemes with various table sizes. BF-EDS is, on average, 1.8x faster compared to ORE for each of the evaluated table sizes.

6.7.2 Ciphertext Generation

This evaluation compares generating ORE and BF-EDS ciphertexts, as well as the duration to update an entire Iceberg manifest file using each scheme. The same ORE and BF-EDS setups used in Section 6.26 are used in this evaluation, with the addition of an ORE scheme using a 32 bit domain, as well as an XOR-encrypted SBBF BF-EDS variant. Figure 6.27 shows the results of the single ciphertext generation benchmark. Both the AES- and XOR-encrypted BF-EDS implementations are orders of magnitude faster than the ORE implementations when generating a single ciphertext. When using AES encryption BF-EDS is between 135 and 67x faster compared to 64 and 32 bit ORE respectively. This is a significant difference, with a real impact in write-heavy applications which often generate ciphertexts.

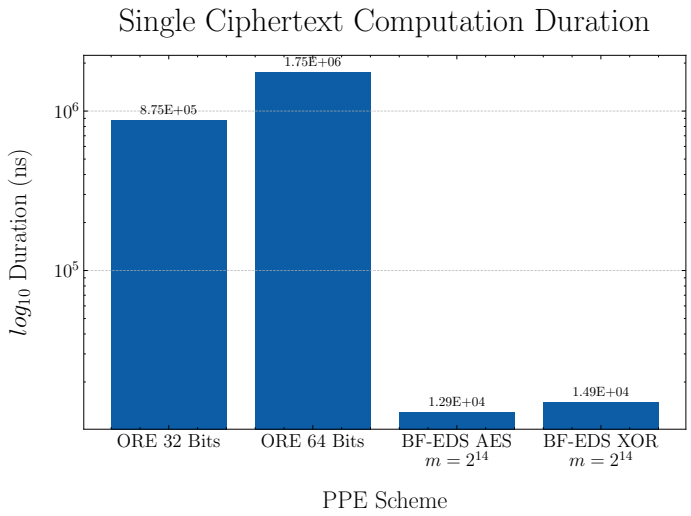


Figure 6.27: Duration to compute a single ciphertext for ORE with two domain sizes, as well as BF-EDS with two encryption methods. BF-EDS is 135x faster than ORE with a 64 bit domain and 67x faster than ORE with a 32 bit domain.

The results of the second evaluation, updating a full Iceberg manifest file, are shown in Figure 6.28. The manifest file used for the evaluation is generated using a batch size of 1000 and thus contains 1000 manifest entries. Both ORE with 64 bit domain as well as BF-EDS using AES-encryption are evaluated. Note that the runtime duration is expressed in seconds as opposed to nanoseconds. Similar to the first evaluation, BF-EDS outperforms ORE by orders of magnitude. Using the largest Bloom filter size 2^{16} (65536 bits) BF-EDS updates the full manifest file 30x faster, taking just 60ms compared to ORE's 1.83 seconds.

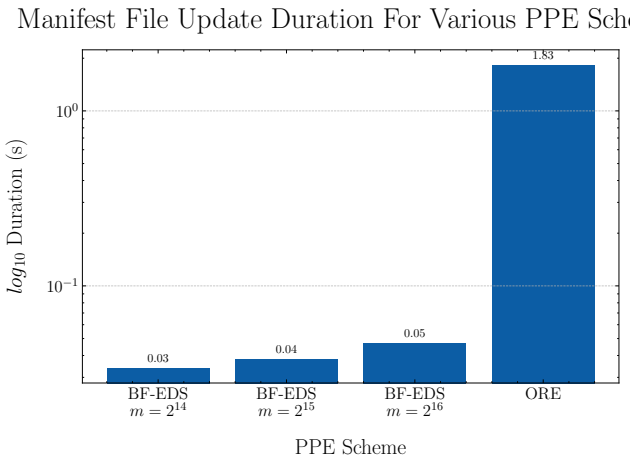


Figure 6.28: Duration to update a full Iceberg manifest file containing 1000 manifest entries using BF-EDS and ORE. BF-EDS significantly outperforms ORE. With the largest Bloom filter bitset size of 2^{16} BF-EDS outperforms ORE by 39x.

Chapter 7

Discussion & Future Work

In this chapter we briefly discuss loose ends as well as potential future work.

In this thesis we have shown that BF-EDS is a performant and secure scheme which can, with high accuracy, perform data skipping for range intersection queries. Throughout this thesis we state that relevant files are returned to a trusted client. Our implementation, however, is purely server-side due to the limited time available. Implementing a hybrid architecture could be done using either a client-server model, or, as seen in the literature study, a software-TH model. Both models would work with BF-EDS. However, as data lakehouse systems focus on OLAP workloads, which oftentimes require many (large) files to be read, a hybrid model using TH makes most sense. With a client-server model network throughput is likely to become a bottleneck. This becomes especially pressing at larger data lake scales (e.g. terabytes, petabytes or more). With a full hybrid system an evaluation of the entire system throughput could be performed. Specifically, a fairer comparison with plaintext DuckDB Iceberg. TH or network throughput overheads will increase the BF-EDS overhead significantly compared to the comparative evaluation performed in this thesis.

Currently our DuckDB Iceberg implementation exclusively supports read operations. Adding support for write operations would significantly increase the usefulness of the scheme. New Bloom filter ciphertexts could be generated on the server in TH, or on a trusted client. Existing Iceberg manifest files could then be updated with these new Bloom filters. Securely communicating the secret keys k_1 and k_2 to TH could be done using attestation and a shared secret via Diffie-Hellman, similar to the attestation processes performed by Azure Always Encrypted[3] and GaussDB[73].

When using SBBFs, during a query only specific blocks of the filters bitset are required. Our DuckDB Iceberg implementation loads the entire Bloom fil-

ter bitset from the manifest entry, instead of loading only the required block. Prefetching could be used to fetch the next required bitset block while concurrently querying the current block. This way latency bound I/O operations could be overlapped with compute operations which increases the overall throughput.

Three Bloom filter variants were implemented and evaluated. Newer Bloom filter variants, for instance the cache-sectorized Bloom filter variant introduced in [38], offer higher throughput with a lower hit to the FPR. The SBBFs we use perform well, however, their performance comes at the cost of a higher FPR compared to basic Bloom filters of the same size. These newer Bloom filters could be added to our BF-EDS library and be evaluated to determine their usefulness in our scheme. Achieving both high throughput and a good FPR reduces the schemes overall storage overhead, as smaller bitsets can be used.

Our current BF-EDS implementation sequentially performs single-key lookups. Given the need to lookup at most $k \cdot (2 \cdot log_2(T))$ keys, vectorized hash primitives could significantly improve query performance. When using a vectorized hash primitive, instead of hashing a single key, an entire vector of keys is hashed at once using SIMD instructions. Given the early termination which oftentimes occurs, we are unsure whether vectorized hashing would actually speed up queries. More hashes than required might be performed, consuming more compute resources which could have been used by other query processes.

We added mapping methods to expand the BF-EDS queryable domain to include signed integers, NULL values and strings. Mapping for additional datatypes could be added to support more complex queries. For instance, booleans and enumerated types could be mapped to specific interval ranges. Via point range intersections more complex SQL operations like `GROUP BY` and `JOIN` should, in theory, be possible. Adding support for this would greatly increase the amount of processing which can be performed on the server, consequently decreasing the amount of data which has to be transferred to a client or TH.

During this thesis DuckLake[57] was released. DuckLake is a simple OTF which re-imagines the traditional data lakehouse architecture. The basic premise of DuckLake is to move all metadata structures into a SQL database, both for catalog and table data. Implementing BF-EDS in DuckLake should be relatively straightforward. DuckDB has native support for Bloom filters, and DuckLake metadata is stored in relational tables, as opposed to Iceberg's Avro files. Additionally, DuckLake can be used with any existing data lake, greatly increasing the number of systems which could benefit from BF-EDS.

Chapter 8

Conclusion

In this thesis we introduced the notion of EDS and presented a novel Bloom filter based EDS scheme (BF-EDS). We implemented and extensively evaluated the BF-EDS scheme. Additionally, we implemented BF-EDS in an existing system, DuckDB Iceberg, to evaluate its practicality and performance. Following are the answers to the research questions we set out to answer:

How can we minimize the storage overhead while optimizing the performance of the BF-EDS scheme at the algorithmic level? Using a Bloom filter which offers the best throughput versus FPR combination minimizes the storage overhead, while offering best-possible performance. The basic Bloom filter provides the lowest FPR with the smallest bitset size. However, the basic Bloom filter suffers in throughput, performing between 2-2.5x worse than a comparable SBBF. Using AES instead of XOR encryption to encrypt the Bloom filter bitset yields significantly higher performance, at the cost of lower security at query time. When using XOR encryption the most significant factor in query performance is the performance of the HFs F_1 and F_2, for which a fast but secure CHF like HighwayHash should be used.

How can the performance of the Bloom filters be optimized while maintaining security? Using a block-based Bloom filter combined with block-by-block XOR encryption offers the best performance versus security tradeoff. The block-size does not influence performance as much as expected, but does increase storage overhead. While using AES encryption offers significantly better performance it reduces security. An adversary has to observe just a single query to obtain the full plaintext bitset. Using a keyed CHF H_k for our Bloom filters protects against dictionary attacks. The uniformity with which H_k hashes

significantly influences the security and leakage of our Bloom filters through non-uniform bit distribution. This makes it easier for an adversary to infer which inputs correspond to specific bits being set. Through observation an adversary may be able to infer information about the plaintext key used in H_k, as well as the range associated with the Bloom filter. A modern CHF like HighwayHash offers both the best performance and the most uniform hashing.

Can we extend the BF-EDS scheme to support negative, NULL and string values? BF-EDS can be extended to support these three types. Any additional type which can in some way be mapped to the numerical domain can be used for BF-EDS. Adding support for negative values was trivial as the full 64 bit signed domain fits inside the 64 bit unsigned domain. Depending on the semantics associated with NULL values they are either ignored (represent no value), or a file containing a NULL value should always be returned (represent any/all values). In the first case, no changes are required as the NULL value won't be represented in the Bloom filter. For the second case the sets $\mathbb{P}(0)$ and $\mathbb{P}(T)$ can be added to the Bloom filter. String values are supported via a novel string mapping algorithm which supports mapping strings with perfect accuracy up to 12 characters. An evaluation on real-world data showed using just 12 characters was sufficient for perfect ordering of longer strings.

How does the performance of BF-EDS compare to EDS using an existing scheme like ORE? BF-EDS is significantly faster both when querying and generating ciphertexts compared to ORE. Additionally, BF-EDS leaks little information, even if the full plaintext bitset is observed. A comparative evaluation was performed comparing querying and ciphertext generation performance between DuckDB Iceberg using BF-EDS and ORE for EDS. While ORE is highly vulnerable to attacks, we performed this comparison to demonstrate the practical performance of our solution, being both faster and more secure compared to ORE. When querying between 10k-1M files, BF-EDS is 1.8x faster on average compared to ORE. BF-EDS generates ciphertexts (EBFs) between 117 and 135x faster, using XOR and AES encryption respectively, compared to ORE. This is a significant difference and is especially relevant for write-heavy applications where many ciphertexts are generated.

How can we integrate BF-EDS into an existing data lakehouse system, such as DuckDB Iceberg? BF-EDS can be integrated into an existing system such as DuckDB Iceberg using a combination of custom settings and checking for Bloom filter presence in manifest files. If Bloom filters are present

these are used and regular bounds information is skipped. Our BF-EDS C++ library contains all required methods to perform BF-EDS, and as such by linking into an existing systems codebase BF-EDS is trivial to add. Updating the Iceberg manifest files to add Bloom filter bitsets proved to be more difficult than expected, due to the rigid binary nature of Avro files. While our implementation works, there is room for improvement as discussed previously.

In this thesis we introduced the concept of EDS and proposed a novel scheme, BF-EDS, which leverages Bloom filters to enable efficient data skipping over encrypted data using range predicates. While existing PPE schemes like OPE and ORE can be used for EDS, BF-EDS significantly reduces information leakage. We implemented BF-EDS in DuckDB Iceberg and evaluated it using custom benchmarks and TPC-H, as well as a comparison with an ORE based EDS implementation. BF-EDS incurred between a 10 to 40% overhead on our own benchmarks, and a 15% overhead compared to plaintext DuckDB Iceberg on the TPC-H benchmark. On average BF-EDS outperformed an ORE-based EDS implementation by 1.8x when querying and 135x when generating ciphertexts. Overall, our findings establish BF-EDS as a practical and secure EDS solution, offering an excellent balance between performance and security.

Chapter 9

Appendix

The appendix contains additional information referenced in this thesis.

9.1 Range Generation Seeds

This list contains the int64 seeds used to generate random ranges during testing and evaluation.

```
-5526699637936106548
-2599495216165156510
 1152894751851419352
  905854664609886134
  836198753323994200
-2453634864968754158
 1239823529904912931
 8799449158572356982
-4728781075803368789
-3371035615212122797
-6806862878324622294
-5103571351601584490
 2894134514233878313
 4344764747414031496
-2216040963088378940
 1699652657573126515
 4062978669480330397
-2927433888129129745
 8068757880956122703
 1132576627787800503
 2265248284352883378
-5739666452106411873
 8585829402665510834
-4518529121956511548
-2097716994910086998
 2363632205110577638
-7971455118372794370
-7171976124329196662
-6384338902479739715
 -139149002439314971
-3554878522296883994
```

6701948383759025869
581683163625277629
7254521096467471263
-7419651579699786674
-4761702335370118884
2050564042523421330
8682352615091205649
5168326528241326321
106968450659912100
-7422480947204228329
1655794280672602103
7360871294234762655
-4995076547733869750
2423844782699678409
1099714007311946935
4051926043588958682
5353893343635388583
7444504291873593822
7176174240641478807
-6078871923352658031
-3011571190410932532
-5230476445706037739
-203249302638236391
-187243911454898925
3154296414642993591
7718163613970455860
-6838567281184370372
3346586808525795136
1609384230784894351
6724113044858062538
2656588490159743405
7168506967979450740
-578549917062904789
-1971118683358498941
7861449626275816724
-1436850705418414870
3718213084596389145
-920663789302100387
5268452343299983855
-1917828076434286675
6348276355767769209
-1416971564537008260
1200247221296323522
-2528819966959134072
976681231239224448
-6893452664840047511
-3671325374383919232
127978847845664228
-8996877824535300611
-2339799268668364837
3257462487445295781
-1321744640853741875
6499297326796480662
-8625726860178189482
2350463590712900358
-2916970912568333488
1508798202809544425
7852535973968707136
850372044584432611
5405077277237659745
-370785963505066635
-3468641342188479676

405172425108258613
-4908015070027393756
-2876958081717649592
9025163892723084554
6689785489388830723
7986081955135897209
-7023371780847147188

9.2 Keyed Hash Function Keys

This list contains the 64 bit keys used during testing and evaluation. These keys are used in the keyed PRFs F1 and F2.

96c199a49e4a416bd5c20c5fcfedc8a380b21eee0ffe33cc252eaa6a36713152a0bd09d2e73f122bc4a2ecad93cd2899bc754fb0f6b2a3a518ed65206b6b4120

a33f192294800b2e422a1d7487b9891633c6ec7db1c84db710a3ada87225f784e46735fabd7fe421e8655f654062a0d4861fe1f8c711d544473ea623259f3ce1

8d2fcf4a47f8907ff23a22e7e809fa0e2a185e9a4826fce5f9f023d71b961f86fee8d9981d3fab1887fcc611199e77c2befe89e7a7baed10a6de68d407318938

d1d4cb0a3dccaaa9a6c70cbc21a0aaed6a340c89cb67faf40748e4d127ca6745093783f878c171e61a2231ec08977f3dcaffb63966fc876f3c37593a8bbb7150

5bb64b1798852752f892cab59267f7f6b0ddba96e780fefc954b186a2ad7db367cdf60f75b059aedb69d54e74059ce54e1c8d3fc5259acb0654ea10f17545220

b9e45903c4ee156047a0c202efedb7adc6d8140d3378d6792414c905865f7eee75d09f5272fa9b3b92420a234589cdb843dd4d483dac0d2a31e504b0502b3e2a

5a11a8e7fed9cdf505e04ff60dd14aee9d03a7416c1b014831c387b0dcb3f80e830c0578f349e786e1352bfe82b51315730dbaec3d118d19a68ea1c4b7fc1481

b64239fb03d50c795673ad1a0a68718f98f003a75673c73917b7a30a911db3dcec33ab2577380e03d93aa516e8a60580250dd34c48df20187c333d29b650f898

2b7c0833b090f764a30f121420f5861f3b3aa781901e79e05309cae29fce6418dc720c85dd9990ee0e55706ef3c2da2187fa7793b85b181f8154f89c1f18a0cf

c8146f3fc77f33f2a978e0e90e5f8bf217607e5f1f760554510bfb869f70c2b2fba8aaecac39bc71f04dfa63a0aa4492195158cf7e8ad4c66c8e2a598a55f421

55545ec9d57d0f93782a786d43f5a7347bac98cc174f4c0b9bbee412884b984373f0948487e5d486e94cd881335a97db226766ae49cd1c1c840fe95b7c27af98

9c9f0e420d26cd294e3da84cfa28634cc9f8ea39bb16e013004738af08f4dd0accae154e181096b9281aa4e929b0b27bf29b59264b94b49e2ff45a1cb4b0e401

adbe30dfa4a837418751361c945b6805f2220756c789f0311652b222a71699066f11c363dad6ad0ec4e54fe3d67a1811514c579873055e1c0520585861c7d717

decc7efbafc59011362c7f81da2fdb7f0afdb605e81ce4d235b0ee5d914a41cb103df418fcc82687d05f236dce9dac30557dc74e1e99fcdb1091ce49852acded

73b9e32af1b69554b4586cc34c5c68062dee4dbc51441626b7b29252d92ff8df2b39133c8f3fda486ee0940008d0f0746a569dedc1e036c56aa6a08051f9cac2

de3b98c42f03c64ce9b01dfdae170f2f76d0628e2789d53b36a64bf732309e6f629361a52abc24809cc2f8dfafc6f06941fbd417bff252c3553be82c538ac803

838b75a161de931cf05200cea366f0a7b4fa66dd9b623beebe9a9378daa5184643d991b928a3e69ca42d1b9b97928ba1250acf26e014bd2e74d772294138d55f

eb898257522edcd3394902ee36be1266935ccec4ba393b464ee4ad8fc484f73acb6f69873deaee25694056713cbc9dc2e478b3fb82ef6a013e404ec692ab5738

6f42defb4de97dfad4f98b79c2621c1e4b5dffbf2a3ce2ea189326f760020ddc85bc69cd2705c5a8ea967a3084efce8d7519bf879755077542d8faa15dec7b23

f9c93872b789427e59a4f6be1ab2b5f67cea5d6e667102fff846de773cceaf58c461d358013885bb814287e007b3295fa108802fa709a5c6e225d4af123c7fa0

4e82235ec5978f44c1719b94605a8fb8a2f1e7a371af88a75a4cd6779fc4e26665142e592d65a0ff97b01b44e638c041d6708a8fa51496c0ef285ac6ddf7ce6a

033986c1ed1d7936f1ab14eeba8b0592fa49388ff6598162a524bc4acbdfb72da697232c819dfb3ac83409c06d0c94114c14a7970bf10357bad6f7a4fbd87093

8f02958a1ea10e503043ab07576b0060c3cf4eb73181dbcd1769ccda1a93b97f38c71b586d9e77b093f47d0104a3217ee2ea440dc497753d49cbd557f55d4585

7371f899e6d586a43f764704eddeddc6e46b54cd4a2c3895417bbf153c74ccdf85928ac82f7b87a1d0b24f524cc0afee60ffd1fa43e2b5a7fe81b1b1b2eef9b2

5b56e46904483e793ee143c20497b5f304f4bbf6685789931e40375cbbadc5f6052145badf89c2c7c0eac217bc096af5af311f69796abde08dd85072999008e6

Bibliography

[1] Rakesh Agrawal et al. "Order Preserving Encryption for Numeric Data". In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. Paris France: ACM, June 2004, pp. 563–574. DOI: 10.1145/1007568.1007632. URL: https://dl.acm.org/doi/10.1145/1007568.1007632 (visited on 07/21/2025).

[2] Jyrki Alakuijala, Bill Cox, and Jan Wassenberg. *Fast Keyed Hash/Pseudo-Random Function Using SIMD Multiply and Permute*. Feb. 2017. DOI: 10.48550/arXiv.1612.06257. arXiv: 1612.06257 [cs]. URL: http://arxiv.org/abs/1612.06257 (visited on 06/24/2025).

[3] Panagiotis Antonopoulos et al. "Azure SQL Database Always Encrypted". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. Portland OR USA: ACM, June 2020, pp. 1511–1525. ISBN: 978-1-4503-6735-6. DOI: 10.1145/3318464.3386141. URL: https://dl.acm.org/doi/10.1145/3318464.3386141 (visited on 03/13/2025).

[4] *Apache Iceberg - Apache Iceberg™*. URL: https://iceberg.apache.org/ (visited on 02/19/2025).

[5] Jim Apple. *Split Block Bloom Filters*. Jan. 2023. DOI: 10.48550/arXiv.2101.01719. arXiv: 2101.01719 [cs]. URL: http://arxiv.org/abs/2101.01719 (visited on 06/18/2025).

[6] Austin Appleby. *MurmurHash3*. 2016. URL: https://github.com/aappleby/smhasher/wiki/MurmurHash3.

[7] Arvind Arasu et al. "Transaction Processing on Confidential Data Using Cipherbase". In: *2015 IEEE 31st International Conference on Data Engineering*. Seoul, South Korea: IEEE, Apr. 2015, pp. 435–446. DOI: 10.1109/icde.2015.7113304. URL: http://ieeexplore.ieee.org/document/7113304/ (visited on 07/09/2025).

[8] Sumeet Bajaj and Radu Sion. "TrustedDB: A Trusted Hardware-Based Database with Privacy and Data Confidentiality". In: *IEEE Transactions on Knowledge and Data Engineering* 26.3 (Mar. 2014), pp. 752–765. ISSN: 1041-4347. DOI: 10.1109/TKDE.2013.38. URL: http://ieeexplore.ieee.org/document/6468039/ (visited on 08/05/2025).

[9] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. "Deterministic and Efficiently Searchable Encryption". In: *Advances in Cryptology - CRYPTO 2007*. Ed. by Alfred Menezes. Vol. 4622. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 535–552. ISBN: 978-3-540-74142-8. DOI: 10.1007/978-3-540-74143-5_30. URL: http://link.springer.com/10.1007/978-3-540-74143-5_30 (visited on 08/04/2025).

[10] *Bloom Filter Indexes*. API Documentation. URL: https://docs.databricks.com/aws/en/optimizations/bloom-filters (visited on 07/30/2025).

[11] *Bloom Filters Explained*. Mar. 2023. URL: https://systemdesign.one/bloom-filters-explained/ (visited on 02/17/2025).

[12] Alexandra Boldyreva et al. "Order-Preserving Symmetric Encryption". In: *Advances in Cryptology - EUROCRYPT 2009*. Ed. by Antoine Joux. Vol. 5479. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 224–241. ISBN: 978-3-642-01000-2 978-3-642-01001-9. DOI: 10.1007/978-3-642-01001-9_13. URL: http://link.springer.com/10.1007/978-3-642-01001-9_13 (visited on 08/04/2025).

[13] Alexandra Boldyreva et al. *Order-Preserving Symmetric Encryption*. 2012. URL: https://eprint.iacr.org/2012/624 (visited on 02/17/2025).

[14] Peter A. Boncz, Marcin Zukowski, and Niels Nes. "MonetDB/X100: Hyper-pipelining Query Execution". In: *Conference on Innovative Data Systems Research*. 2005. URL: https://api.semanticscholar.org/CorpusID:1379707.

[15] Boudewijn Braams. "Predicate Pushdown in Parquet and Apache Spark Author :" in: 2018. URL: https://api.semanticscholar.org/CorpusID:148564722.

[16] Andrei Broder and Michael Mitzenmacher. "Network Applications of Bloom Filters: A Survey". In: *Internet Mathematics* 1.4 (Jan. 2004), pp. 485–509. ISSN: 1542-7951, 1944-9488. DOI: 10.1080/15427951.2004.10129096. URL: http://www.internetmathematicsjournal.com/article/1393 (visited on 07/03/2025).

[17] Nathan Chenette et al. "Practical Order-Revealing Encryption with Limited Leakage". In: *Fast Software Encryption*. Ed. by Thomas Peyrin. Vol. 9783. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 474–493. ISBN: 978-3-662-52992-8 978-3-662-52993-5. DOI: 10.1007/978-3-662-52993-5_24. URL: http://link.springer.com/10.1007/978-3-662-52993-5_24 (visited on 08/04/2025).

[18] *ClickHouse Cloud | Cloud Based DBMS | ClickHouse*. URL: https://clickhouse.com/cloud?utm_source=google.com&utm_medium=paid_search&utm_campaign=21862172336_169330245109&utm_content=719379733837&utm_term=clickhouse_g_c&gad_source=1&gbraid=0AAAAAocOPCYqN-SjfAICf37kfUBvek8xt&gclid=CjwKCAiAlPu9BhAjEiwA5NDSAwMxmV-FuLSuxxbx2s_Zsa9cr-bImWHtKpSs-6zyu4bmnDFolTaLaBoCNskQAvD_BwE (visited on 02/26/2025).

[19] *Cryptographic Hash Functions | IBM Quantum Learning*. URL: https://learning.quantum.ibm.com/course/practical-introduction-to-quantum-safe-cryptography/cryptographic-hash-functions (visited on 05/06/2025).

[20] *Data Skipping for Delta Lake*. URL: https://docs.databricks.com (visited on 02/17/2025).

[21] *Database Data Warehousing Guide - About Zone Maps*. URL: https://docs.oracle.com/database/121/DWHSG/zone_maps.htm#DWHSG8935 (visited on 02/17/2025).

[22] F. Betül Durak, Thomas M. DuBuisson, and David Cash. "What Else Is Revealed by Order-Revealing Encryption?" In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. Vienna Austria: ACM, Oct. 2016, pp. 1155–1166. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978379. URL: https://dl.acm.org/doi/10.1145/2976749.2978379 (visited on 08/04/2025).

[23] T. Elgamal. "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms". In: *IEEE Transactions on Information Theory* 31.4 (July 1985), pp. 469–472. ISSN: 0018-9448, 1557-9654. DOI: 10.1109/tit.1985.1057074. URL: https://ieeexplore.ieee.org/document/1057074/ (visited on 07/21/2025).

[24] Saba Eskandarian and Matei Zaharia. "ObliDB: Oblivious Query Processing for Secure Databases". In: *Proceedings of the VLDB Endowment* 13.2 (Oct. 2019), pp. 169–183. ISSN: 2150-8097. DOI: 10.14778/3364324.3364331. URL: https://dl.acm.org/doi/10.14778/3364324.3364331 (visited on 07/25/2025).

[25] Charlotte Felius and Peter A. Boncz. "VCrypt: Leveraging Vectorized and Compressed Execution for Client-Side Encryption". In: *International Conference on Extending Database Technology*. 2025. URL: https://api.semanticscholar.org/CorpusID:276903952.

[26] Hellman Fredrik. "Study And Comparison Of Data Lakehouse Systems". MA thesis. Sweden: Åbo Akademi University, 2023.

[27] Craig Gentry. "Fully Homomorphic Encryption Using Ideal Lattices". In: *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*. Bethesda MD USA: ACM, May 2009, pp. 169–178. DOI: 10.1145/1536414.1536440. URL: https://dl.acm.org/doi/10.1145/1536414.1536440 (visited on 07/21/2025).

[28] Ali Ghodsi. *Databricks Kicks off Data + AI Summit 2025*. Dec. 2025. URL: https://www.youtube.com/watch?v=PFVJL7_W4dI.

[29] Oded Goldreich and Rafail Ostrovsky. "Software Protection and Simulation on Oblivious RAMs". In: *Journal of the ACM* 43.3 (May 1996), pp. 431–473. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/233551.233553. URL: https://dl.acm.org/doi/10.1145/233551.233553 (visited on 07/25/2025).

[30] *Google Benchmark*. Google. URL: https://github.com/google/benchmark.

[31] Paul Grubbs et al. "Leakage-Abuse Attacks against Order-Revealing Encryption". In: *2017 IEEE Symposium on Security and Privacy (SP)*. San Jose, CA, USA: IEEE, May 2017, pp. 655–672. ISBN: 978-1-5090-5533-3. DOI: 10.1109/SP.2017.44. URL: http://ieeexplore.ieee.org/document/7958603/ (visited on 08/04/2025).

[32] Huawei Cloud Academy. *openGauss Overview*. URL: https://r.huaweistatic.com/s/kunpengstatic/lst/files/pdf/learn/courses/openGauss%20Overview.pdf.

[33] *Iceberg Java API*. API Documentation. URL: https://iceberg.apache.org/docs/1.4.3/api/ (visited on 07/29/2025).

[34] Paras Jain et al. "Analyzing and Comparing Lakehouse Storage Systems". In: *Conference on Innovative Data Systems Research*. 2023. URL: https://api.semanticscholar.org/CorpusID:259267242.

[35] Julian James Stephen et al. "Program Analysis for Secure Big Data Processing". In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. Vasteras Sweden: ACM, Sept. 2014, pp. 277–288. ISBN: 978-1-4503-3013-8. DOI: 10.1145/2642937.2643006. URL: https://dl.acm.org/doi/10.1145/2642937.2643006 (visited on 07/27/2025).

[36] Adam Kirsch and Michael Mitzenmacher. "Less Hashing, Same Performance: Building a Better Bloom Filter". In: *Random Structures & Algorithms* 33 (2006). URL: https://api.semanticscholar.org/CorpusID:2754699.

[37] Jeremy Kun. *A High-Level Technical Overview of Fully Homomorphic Encryption*. May 2024. DOI: 10.59350/a7m2z-wz087. URL: https://www.jeremykun.com/2024/05/04/fhe-overview (visited on 07/21/2025).

[38] Harald Lang et al. "Performance-Optimal Filtering: Bloom Overtakes Cuckoo at High Throughput". In: *Proc. VLDB Endow.* 12.5 (Jan. 2019), pp. 502–515. ISSN: 2150-8097. DOI: 10.14778/3303753.3303757. URL: https://doi.org/10.14778/3303753.3303757 (visited on 03/19/2025).

[39] Kevin Lewi and David J. Wu. "Order-Revealing Encryption: New Constructions, Applications, and Lower Bounds". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. Vienna Austria: ACM, Oct. 2016, pp. 1167–1178. DOI: 10.1145/2976749.2978376. URL: https://dl.acm.org/doi/10.1145/2976749.2978376 (visited on 07/19/2025).

[40] Hui Li et al. *Enc2DB: A Hybrid and Adaptive Encrypted Query Processing Framework*. Apr. 2024. DOI: 10.48550/arXiv.2404.06819. arXiv: 2404.06819 [cs]. URL: http://arxiv.org/abs/2404.06819 (visited on 07/15/2025).

[41] Marilex Rea Llave. "Data Lakes in Business Intelligence: Reporting from the Trenches". In: *Procedia Computer Science* 138 (2018), pp. 516–524. ISSN: 18770509. DOI: 10.1016/j.procs.2018.10.071. URL: https://linkinghub.elsevier.com/retrieve/pii/S1877050918317046 (visited on 02/17/2025).

[42] Yanbin Lu. "Privacy-Preserving Logarithmic-Time Search on Encrypted Data in Cloud". In: *Network and Distributed System Security Symposium*. 2012. URL: https://api.semanticscholar.org/CorpusID:15980765.

[43] Angel Conde Manjon, Diego Colombatto, and Sandeep Adwankar. *Compaction Support for Avro and ORC File Formats in Apache Iceberg Tables in Amazon S3*. July 2025. URL: https://aws.amazon.com/blogs/big-data/compaction-support-for-avro-and-orc-file-formats-in-apache-iceberg-tables-in-amazon-s3/ (visited on 07/29/2025).

[44] *Modern Bloom Filters: 22x Faster!* URL: https://save-buffer.github.io/bloom_filter.html (visited on 02/21/2025).

[45] Hannes Mühleisen. *Parquet Bloom Filters in DuckDB*. Mar. 2025. URL: https://duckdb.org/2025/03/07/parquet-bloom-filters-in-duckdb.html#writing (visited on 07/30/2025).

[46] *MySQL :: MySQL 8.4 Reference Manual :: 26.4 Partition Pruning*. URL: https://dev.mysql.com/doc/refman/8.4/en/partitioning-pruning.html (visited on 02/17/2025).

[47] National Institute of Standards and Technology (US). *Secure Hash Standard*. Tech. rep. Washington, D.C.: National Institute of Standards and Technology, 2015. DOI: 10.6028/nist.fips.180-4. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf (visited on 07/22/2025).

[48] Muhammad Naveed, Seny Kamara, and Charles V. Wright. "Inference Attacks on Property-Preserving Encrypted Databases". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. Denver Colorado USA: ACM, Oct. 2015, pp. 644–655. DOI: 10.1145/2810103.2813651. URL: https://dl.acm.org/doi/10.1145/2810103.2813651 (visited on 07/15/2025).

[49] *NextiaJD Reddit Comments Dataset*. URL: https://event.cwi.nl/da/NextiaJD/.

[50] Jack O'Connor et al. "One Function, Fast Everywhere". URL: https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf.

[51] *Open Table Formats: Which Table Format to Choose*. URL: https://www.starburst.io/data-glossary/open-table-formats/ (visited on 02/17/2025).

[52] "Order-Preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions". In: *Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 578–595. ISBN: 978-3-642-22791-2 978-3-642-22792-9. DOI: 10.1007/978-3-642-22792-9_33. URL: http://link.springer.com/10.1007/978-3-642-22792-9_33 (visited on 07/21/2025).

[53] Pascal Paillier. "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes". In: *Advances in Cryptology — EUROCRYPT '99*. Ed. by Jacques Stern. Vol. 1592. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 223–238. ISBN: 978-3-540-65889-4. DOI: 10.1007/3-540-48910-X_16. URL: http://link.springer.com/10.1007/3-540-48910-X_16 (visited on 02/18/2025).

[54] Marut Pandya. *Performance Evaluation of Hashing Algorithms on Commodity Hardware*. July 2024. DOI: 10.48550/arXiv.2407.08284. arXiv: 2407.08284 [cs]. URL: http://arxiv.org/abs/2407.08284 (visited on 07/22/2025).

[55] Raluca Ada Popa et al. “CryptDB: Protecting Confidentiality with Encrypted Query Processing”. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. Cascais Portugal: ACM, Oct. 2011, pp. 85–100. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043566. URL: https://dl.acm.org/doi/10.1145/2043556.2043566 (visited on 08/05/2025).

[56] Mark Raasveldt and Hannes Mühleisen. “DuckDB: An Embeddable Analytical Database”. In: *Proceedings of the 2019 International Conference on Management of Data*. Amsterdam Netherlands: ACM, June 2019, pp. 1981–1984. ISBN: 978-1-4503-5643-5. DOI: 10.1145/3299869.3320212. URL: https://dl.acm.org/doi/10.1145/3299869.3320212 (visited on 02/26/2025).

[57] Mark Raasveldt and Hannes Mühleisen. *DuckLake: SQL as a Lakehouse Format*. May 2025. URL: https://duckdb.org/2025/05/27/ducklake.html (visited on 05/08/2025).

[58] Daniel S. Roche, Adam Aviv, and Seung Geol Choi. “A Practical Oblivious Map Data Structure with Secure Deletion and History Independence”. In: *2016 IEEE Symposium on Security and Privacy (SP)*. San Jose, CA: IEEE, May 2016, pp. 178–197. DOI: 10.1109/sp.2016.19. URL: http://ieeexplore.ieee.org/document/7546502/ (visited on 07/25/2025).

[59] Savvas Savvides, Darshika Khandelwal, and Patrick Eugster. “Efficient Confidentiality-Preserving Data Analytics over Symmetrically Encrypted Datasets”. In: *Proceedings of the VLDB Endowment* 13.8 (Apr. 2020), pp. 1290–1303. ISSN: 2150-8097. DOI: 10.14778/3389133.3389144. URL: https://dl.acm.org/doi/10.14778/3389133.3389144 (visited on 07/15/2025).

[60] Savvas Savvides et al. “Secure Data Types: A Simple Abstraction for Confidentiality-Preserving Data Analytics”. In: *Proceedings of the 2017 Symposium on Cloud Computing*. Santa Clara California: ACM, Sept. 2017, pp. 479–492. DOI: 10.1145/3127479.3129256. URL: https://dl.acm.org/doi/10.1145/3127479.3129256 (visited on 07/15/2025).

[61] “Semantically Secure Order-Revealing Encryption: Multi-input Functional Encryption Without Obfuscation”. In: *Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 563–594. ISBN: 978-3-662-46802-9 978-3-662-46803-6. DOI: 10.1007/978-3-662-46803-6_19. URL: http://link.springer.com/10.1007/978-3-662-46803-6_19 (visited on 07/21/2025).

[62] Elaine Shi et al. “Multi-Dimensional Range Query over Encrypted Data”. In: *2007 IEEE Symposium on Security and Privacy (SP '07)*. Berkeley, CA: IEEE, May 2007, pp. 350–364. ISBN: 978-0-7695-2848-9. DOI: 10.1109/SP.2007.29. URL: http://ieeexplore.ieee.org/document/4223238/ (visited on 02/24/2025).

[63] Tomer Shiran. *Apache Iceberg: The Definitive Guide*. 1st ed. Sebastopol: O'Reilly Media, Incorporated, 2024. ISBN: 978-1-0981-4862-1 978-1-0981-4859-1.

[64] Vasily Sidorov, Ethan Yi Fan Wei, and Wee Keong Ng. *Comprehensive Performance Analysis of Homomorphic Cryptosystems for Practical Data Processing*. Feb. 2022. DOI: 10.48550/arXiv.2202.02960. arXiv: 2202.02960 [cs]. URL: http://arxiv.org/abs/2202.02960 (visited on 07/21/2025).

[65] “SQL on Structurally-Encrypted Databases”. In: *Lecture Notes in Computer Science*. Cham: Springer International Publishing, 2018, pp. 149–180. ISBN: 978-3-030-03325-5 978-3-030-03326-2. DOI: 10.1007/978-3-030-03326-2_6. URL: https://link.springer.com/10.1007/978-3-030-03326-2_6 (visited on 07/24/2025).

[66] Stanford. *Order-Revealing Encryption*. URL: https://crypto.stanford.edu/ore/ (visited on 07/22/2025).

[67] Ignacio G. Terrizzano et al. “Data Wrangling: The Challenging Yourney from the Wild to the Lake”. In: *Conference on Innovative Data Systems Research*. 2015. URL: https://api.semanticscholar.org/CorpusID:17462093.

[68] Stephen Tu et al. “Processing Analytical Queries over Encrypted Data”. In: *Proceedings of the VLDB Endowment* 6.5 (Mar. 2013), pp. 289–300. ISSN: 2150-8097. DOI: 10.14778/2535573.2488336. URL: https://dl.acm.org/doi/10.14778/2535573.2488336 (visited on 02/17/2025).

[69] Dandan Yuan. “Practical and Secure Searchable Symmetric Encryption Constructions”. PhD thesis. Auckland: University of Auckland, 2023.

[70] Matei Zaharia et al. “Apache Spark: A Unified Engine for Big Data Processing”. In: *Communications of the ACM* 59.11 (Oct. 2016), pp. 56–65. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/2934664. URL: https://dl.acm.org/doi/10.1145/2934664 (visited on 07/25/2025).

[71] Zheguang Zhao and Stanley B. Zdonik. “Encrypted Databases: From Theory to Systems”. In: *Conference on Innovative Data Systems Research*. 2021. URL: https://api.semanticscholar.org/CorpusID:231750594.

[72] Wenting Zheng et al. “Opaque: An Oblivious and Encrypted Distributed Analytics Platform”. In: *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*. NSDI’17. Boston, MA, USA and USA: USENIX Association, 2017, pp. 283–298. ISBN: 978-1-931971-37-9.

[73] Jinwei Zhu et al. “Full Encryption: An End to End Encryption Mechanism in GaussDB”. In: *Proceedings of the VLDB Endowment* 14.12 (July 2021), pp. 2811–2814. ISSN: 2150-8097. DOI: 10.14778/3476311.3476351. URL: https://dl.acm.org/doi/10.14778/3476311.3476351 (visited on 07/15/2025).

[74] *Zone Maps in Data Lake Relational Engine (SAP HANA DB-Managed) | SAP Help Portal*. URL: https://help.sap.com/docs/hana-cloud-data-lake/performance-and-tuning-for-data-lake-relational-engine-basics/zone-maps (visited on 02/17/2025).