VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
SOFTWARE SYSTEMS DEPARTMENT

Course: Cyber Security Technologies, 2025 Masters PS + KM

Fast lookup of known files hashes database

HashDB

Authors:
Žygimantas Remeika Albertas Grinkevičius

December 16, 2025

Summary

The purpose of this project was to develop an application that would allow fast lookup of the hash database. In digital forensics and cyber security, security engineers to save compute and human resources use collections of already known and approved files (OS, apps) with their respective hashes and metadata to filter them out from scans. Those files come from legitimate verified publishers and known software developers or distributors. Examples include the Linux Foundation, Microsoft, Oracle, and IBM. The National Institute of Standards and Technology (NIST) database is the National Software Reference Library (NSRL) Reference Data Set (RDS). The dataset contains more than 500GB of hashes with metadata. This poses a challenge as keeping 500GB of data in memory is costly, while scanning the full file on every lookup would be slow and inefficient.

Team contacts

1. Žygimantas Remeika - zygimantas.remeika@mif.stud.vu.lt
2. Albertas Grinkevičius - albertas.grinkevicius@mif.stud.vu.lt

Contents

Introduction

In the field of digital forensics, investigators are often faced with the ”needle in a haystack” problem. When analyzing a seized hard drive or a compromised server, the storage media may contain millions of individual files. The vast majority of these files are standard, non-malicious operating system components or common application files (e.g., Microsoft Windows system DLLs, Linux kernel modules, or browser executables). Analyzing every single file manually or scanning them all for malware consumes vast amounts of computing power and human time.

To solve this, the industry uses a technique called ”Known File Filtering” or ”Allowlisting.” This involves computing the cryptographic hash (a digital fingerprint) of every file on the disk and comparing it against a reference database of known good files. If a file’s hash matches a known safe file from a trusted vendor (like Microsoft or Adobe), it can be safely ignored, allowing investigators to focus only on the unknown or suspicious files.

The primary source for this data is the National Software Reference Library (NSRL) Reference Data Set (RDS), maintained by NIST. The modern dataset is massive, containing billions of records which can exceed 1TB of data when fully expanded. Currently, this dataset is distributed as a complex SQLite database. While SQLite is excellent for compatibility, it is not optimized for the high-speed, bulk lookups required in modern forensic scenarios. The core technical challenge of this project involves migrating this dataset of high cardinality from a Relational Database Management System (RDBMS) to a high-performance Key-Value (KV) store to achieve sub-millisecond lookup speeds.

1. Baseline NIST RDS SQLite

The baseline implementation provided by NIST uses SQLite. This is a standard Relational Database Management System (RDBMS) that organizes data using B-Trees (or B+ Trees). While SQLite is reliable, its internal architecture creates significant performance bottlenecks when dealing with a dataset of this magnitude for read-heavy workloads.

1.1. Database Normalization and Joins

The NSRL dataset is highly "normalized" to save storage space. Normalization means that data is split into many separate tables to avoid repetition. For example, the filename `kernel32.dll` appears in millions of different software packages. Instead of writing "`kernel32.dll`" millions of times, SQLite stores it once in a `FILE_NAME` table and other tables reference it using numeric IDs (Foreign Keys).

While this is efficient for storage, it imposes a massive penalty during reading. To reconstruct the complete information for a single file, the database engine must find data in the `FILE` table, then jump to the `PKG` (Package) table, then the `OS` (Operating System) table, and finally the `MFG` (Manufacturer) table. This process is called a `JOIN`.

$$\text{Query Cost} \approx \text{Cost}(FILE) + \text{Cost}(PKG) + \text{Cost}(OS) + \text{Cost}(MFG) \tag{1}$$

In computational terms, the CPU has to work much harder to assemble these pieces every single time a lookup is requested.

1.2. B-Tree Limitations

SQLite uses B-Trees to index data. A B-Tree is a hierarchical structure. To find a specific record, the database must start at the "root" of the tree and traverse down through several layers of "branch" nodes until it reaches the "leaf" node containing the data.

- **Computational Complexity**: A B-Tree lookup has a complexity of $\mathcal{O}(\log n)$. As the number of records (n) grows into the billions, the tree becomes deeper, requiring more steps to find a record.
- **Random I/O**: Each step down the B-Tree often requires fetching a different "page" of data from the hard disk. These pages are often scattered across the disk, leading to "random I/O" operations. Standard hard drives and even SSDs are significantly slower at random reads compared to sequential reads.

For a forensic scan requiring millions of lookups, these B-Tree traversals and JOIN operations accumulate, making the baseline SQLite solution too slow for real-time applications.

2. Target Key-Value store with RocksDB

The RocksDB implementation shifts to a denormalized model. Instead of reconstructing data at read time, data is pre-joined during the migration phase. The `migrate_to_rocksdb.go` code demonstrates this by executing the complex JOIN query once, marshaling the result into a static JSON blob (the `FileData` struct), and storing it contiguously on disk. This trades write-time CPU (during the build) for read-time speed. The retrieval complexity drops to $\mathcal{O}(1)$—a direct hash lookup—eliminating the CPU cycles previously spent on join logic.

2.1. Log-Structured Merge-Trees (LSM-Trees)

The choice of RocksDB introduces a specific storage engine architecture known as the Log-Structured Merge-tree (LSM-Tree), which differs fundamentally from the page-based B-Trees of SQLite.

In a B-Tree, an insert often requires a ”random write” to update a specific page on the disk. In contrast, the LSM-Tree organizes data into two primary structures:

- **MemTable (Memory):** New data is written to an in-memory buffer (typically a Skip List). This makes inserts extremely fast as they do not immediately touch the disk.
- **SSTable (Sorted String Table):** When the MemTable fills, it is flushed to disk as an immutable file. Because the data was sorted in memory, this flush is a ”sequential write,” which is significantly faster than random writes on physical storage media (SSDs/HDDs).

2.2. Compaction

Because data is constantly appended to new files, older versions of data (or deleted keys) may persist in older SSTables. RocksDB employs a background process called Compaction to merge these files, discard obsolete data, and maintain read efficiency. This ensures that the dataset remains optimized for storage without blocking reads.

3. Probabilistic Data Structure: Bloom Filter

The reference code explicitly configures a Bloom Filter with 10 bits per key (`bbto.SetFilterPolicy(grocksdb.NewBloomFilter(10))`). This is the critical theoretical component for optimizing "negative lookups" (checking a hash that is not in the database).

A Bloom Filter is a space-efficient probabilistic data structure representing a set. It consists of a bit array of m bits, initially all set to 0. When a key K is added:

- The system calculates k distinct hash functions $(h_1(K), h_2(K), \ldots, h_k(K))$.
- Each function outputs an index position in the array.
- The bits at these positions are flipped to 1.

3.1. False Positives

If any of the bits at the calculated positions are 0, the element is definitely not in the set. This allows RocksDB to return "Not Found" instantly from RAM, bypassing expensive disk I/O entirely. If all bits are 1, the element might be in the set. The system must then read the SSTable from disk to confirm.

In forensic scanning, a "whitelist" check often involves querying massive numbers of safe files (which exist) and unknown files (which do not). Without a Bloom Filter, every "unknown" file would force the database to scan the disk to confirm its absence. The Bloom Filter reduces the disk I/O for these negative lookups to near zero.

3.2. Key Design

The implementation in `migrate_to_rocksdb.go` utilizes a Composite Key strategy:

$$\texttt{key} := \texttt{sha256} + \texttt{sha1} + \texttt{md5} + \texttt{crc32} \tag{2}$$

Deterministic Access: In Key-Value stores, the access pattern is strictly deterministic. Unlike SQL, where you can query `WHERE file_size > 1000`, a KV store requires the exact key. By concatenating the hashes, the system enforces a strict lookup requirement: the client must possess all cryptographic identifiers to retrieve the metadata.

High Entropy Distribution: Cryptographic hashes (SHA-256, MD5) are designed to output data with maximum entropy (randomness). In an LSM-Tree, this randomness ensures that keys are uniformly distributed across the storage space. This prevents "hotspots" where one specific section of the database receives all the traffic, ensuring that the heavy read load of a forensic scan is evenly balanced across the system resources.

4. Implementation

This section describes the design decisions and components developed for the HashDB system.

4.1. Technology Choices

4.1.1. Programming Language: Go

Go was selected for implementation due to its strong performance characteristics, excellent concurrency support, and straightforward CGO integration for calling RocksDB's C++ library. The `grocksdb` package provides idiomatic Go bindings to RocksDB.

4.1.2. Serialization: JSON

JSON was chosen for value serialization over binary formats (Protocol Buffers, MessagePack) for simplicity and debuggability. While binary formats offer better space efficiency, JSON provides human-readable output and easier debugging during development. For production deployments with the full 1TB dataset, migration to a binary format could reduce storage by approximately 30–40%.

4.2. Denormalization Strategy

The migration process performs a single JOIN query across all four SQLite tables (FILE, PKG, OS, MFG) and stores the complete result as a single JSON document per file hash. This trades increased storage space for eliminated read-time JOIN overhead. Each lookup retrieves all metadata in a single disk operation rather than four separate B-tree traversals.

4.3. Bloom Filter Configuration

A 10-bit-per-key Bloom filter was configured based on the standard space-accuracy tradeoff. This configuration yields a theoretical false positive rate of approximately 0.82%, meaning less than 1% of negative lookups will require unnecessary disk reads. The filter is cached in memory (`SetCacheIndexAndFilterBlocks(true)`), ensuring that negative lookups are resolved entirely from RAM.

4.4. Project Components

The implementation consists of:

- `migrate_to_rocksdb.go` – One-time migration tool that reads SQLite, performs denormalization, and populates RocksDB
- `hashdb.go` – Command-line lookup utility supporting single queries and bulk file processing

- `benchmark.go` – Performance comparison suite testing SQLite vs RocksDB with/without Bloom filters

5. Benchmarking

To validate the performance improvements, a comprehensive benchmarking suite was developed to compare SQLite against RocksDB with and without Bloom filters.

5.1. Benchmark Methodology

The benchmark generates a mixed workload consisting of:

- 70% existing records (positive lookups) – randomly sampled from the database
- 30% non-existing records (negative lookups) – randomly generated hash values

This distribution reflects realistic forensic scanning scenarios where most files on a system are known good files, but a significant portion are unknown and require verification.

5.1.1. Test Configurations

Three configurations were benchmarked:

1. **SQLite (exact match)** – Baseline using the normalized schema with JOIN operations
2. **RocksDB WITH Bloom filter** – LSM-tree with 10-bit Bloom filter enabled
3. **RocksDB WITHOUT Bloom filter** – LSM-tree without probabilistic filtering

5.1.2. Direct I/O Mode

The benchmark supports a `--direct-io` flag that bypasses the operating system's page cache:

```
./run_benchmark.sh 1000 --direct-io
```

This mode provides more accurate measurements of the Bloom filter's benefit by forcing actual disk I/O operations rather than serving data from cached memory.

5.2. Running Benchmarks

Execute the benchmark suite:

```
./run_benchmark.sh [num_queries] [--direct-io]
```

Example with 1000 queries:

```
./run_benchmark.sh 1000
```

5.3. Results

5.3.1. Database Statistics

Metric	Value
Estimated keys	1,911,244
Total SST files on disk	308.69 MB
Bloom filter size	2.28 MB
Bloom filter % of DB size	0.74%

Table 1. RocksDB database statistics

5.3.2. Performance Comparison

Configuration	Total Time	Avg Time	QPS
SQLite (exact match)	1,472 ms	147.2 µs	6,793
RocksDB + Bloom	602 ms	60.2 µs	16,610
RocksDB no Bloom	848 ms	84.8 µs	11,797

Table 2. Benchmark results for 10,000 queries (70% hits, 30% misses) with Direct I/O enabled

5.3.3. Speedup Analysis

Comparison	Speedup	Time Saved
RocksDB + Bloom vs SQLite	2.45x	870 ms
RocksDB no Bloom vs SQLite	1.74x	624 ms
Bloom vs no Bloom	1.41x	246 ms

Table 3. Performance speedup comparison

5.4. Analysis

5.4.1. SQLite Performance Bottlenecks

The SQLite implementation exhibits the highest latency due to:

- **JOIN overhead**: Each query requires traversing four tables (FILE, PKG, OS, MFG)
- **B-Tree traversal**: Multiple $\mathcal{O}(\log n)$ lookups per query
- **Index fragmentation**: Random I/O patterns across scattered index pages

5.4.2. RocksDB Advantages

The RocksDB implementation demonstrates significant improvements:

- **Denormalized storage**: Single $\mathcal{O}(1)$ lookup per query
- **Sequential I/O**: LSM-tree architecture optimizes for sequential reads
- **Compressed storage**: LZ4/Snappy compression reduces I/O volume

5.4.3. Bloom Filter Impact

The Bloom filter provides the most significant benefit for negative lookups:

- **Memory-only rejection**: Non-existing keys are rejected without disk I/O
- **False positive rate**: With 10 bits per key, the theoretical false positive rate is approximately 1%
- **Space efficiency**: The filter requires only $\frac{n \times 10}{8}$ bytes of memory

The theoretical false positive probability for a Bloom filter is:

$$P_{fp} = \left(1 - e^{-kn/m}\right)^k \tag{3}$$

where k is the number of hash functions, n is the number of elements, and m is the number of bits. For 10 bits per key with optimal $k \approx 7$:

$$P_{fp} \approx 0.0082 \approx 0.82\% \tag{4}$$

5.5. Conclusions

The benchmarking results demonstrate that migrating from SQLite to RocksDB with Bloom filters provides substantial performance improvements for hash database lookups. The combination of denormalized storage, LSM-tree architecture, and probabilistic filtering makes this approach well-suited for high-throughput forensic scanning applications.

Key findings:

1. RocksDB with Bloom filter achieves **2.45x speedup** over SQLite, processing 16,610 queries per second compared to SQLite's 6,793
2. Even without Bloom filters, RocksDB outperforms SQLite by **1.74x** due to denormalization eliminating JOIN overhead
3. The Bloom filter provides an additional **1.41x improvement** over RocksDB without filtering, demonstrating its effectiveness for rejecting negative lookups
4. The Bloom filter requires only **2.28 MB** (0.74% of database size) while dramatically reducing disk I/O for non-existing keys
5. Minimum query time dropped from 12 µs (SQLite) to sub-microsecond (833 ns) with RocksDB + Bloom filter

6. Estimates on full NIST RDS production dataset

This section calculates the storage and hardware resources needed to deploy the HashDB application using the full NIST NSRL Reference Data Set (RDS). We used performance data from our initial pilot test (which used a subset of 1.9 million keys) to estimate the requirements for the full production dataset (estimated at 162–280 million unique keys).

6.1. Source Data Storage Requirements

Before the data can be inserted into RocksDB store, we need enough disk space to download and uncompress initial NIST RDS dataset. The NIST RDSv3 is distributed as four separate SQLite databases.

The "Modern" dataset is the largest component because it includes full file paths in its metadata. Table 4 shows the difference between the compressed download size and the actual space needed on disk after extraction.

Dataset Component	Download Size (ZIP)	Uncompressed Size (SQLite)
Modern PC (Full)	119 GB	∼350 GB
Legacy PC (Full)	∼35 GB	∼100 GB
Android (Full)	∼15 GB	∼35 GB
iOS (Full)	∼5 GB	∼15 GB
Total Staging Storage	**∼174 GB**	**∼500 GB**

Table 4. Storage requirements for the raw NIST input artifacts.

Observation: We need to ensure the staging server has at least **500 GB** of free space to hold these uncompressed databases during the migration process.

6.2. Estimating the Final RocksDB Size

We calculated the final database size by looking at the data density from our pilot test. RocksDB uses LSM-Trees (Log-Structured Merge-Trees) and compression algorithms like Snappy or LZ4, which makes it much more space-efficient than the standard SQLite B-Trees used in the source data.

6.2.1. Scaling Factors

- **Pilot Data Density:** In our test, 1,911,244 keys took up 308.69 MB.

$$\text{Average size per key} = \frac{308.69 \text{ MB}}{1{,}911{,}244 \text{ keys}} \approx 161.5 \text{ bytes/key}$$

- **Total Production Keys:** We estimate ∼162,000,000 distinct SHA-256 hashes across all datasets.

6.2.2. Projected Database Size

Using the average size per key, we can estimate the total size:

$$\text{Total Size} = 162{,}000{,}000 \text{ keys} \times 161.5 \text{ bytes} \approx 26.16 \text{ GB}$$

Result: The final application database will likely take up between **26 GB and 30 GB**. This is a significant reduction ($\sim$94%) compared to the 500 GB raw source data.

6.3. Bloom Filter Analysis

To improve lookup speeds, we use a Bloom filter. This allows the system to instantly know if a file hash *does not* exist in the database without reading from the disk.

6.3.1. Size Calculation

- **Configuration:** We allocated 10 bits per key, which gives a False Positive Rate of about 1%.
- **Key Count (n):** 162,000,000

$$\text{Size (bits)} = 162{,}000{,}000 \times 10 = 1{,}620{,}000{,}000 \text{ bits}$$

$$\text{Size (bytes)} = \frac{1{,}620{,}000{,}000}{8} \approx 202.5 \text{ MB}$$

Impact: The Bloom filter adds about 200 MB to the storage size. More importantly, it requires **200 MB of RAM** to be kept in memory at all times.

6.4. Migration Performance (ETL)

The migration process involves reading roughly 1.3 billion rows from SQLite, converting them to JSON, and writing the unique keys to RocksDB. The main bottleneck will be the read speed of the SQLite source files.

6.4.1. Resource Estimates

Metric	Estimate	Basis of Calculation
Time Required	4 – 6 Hours	Assumes reading 50k–80k rows/second.
Processing Load	High CPU	JSON serialization requires significant CPU power (est. 8 cores).
Memory Usage	High RAM	Large buffers are needed to sort keys before bulk insertion (est. 16 GB RAM).

Table 5. Estimated resource consumption for the migration process.

6.5. Hardware Recommendations

To maintain the performance we saw in the pilot (>16,000 queries per second), we recommend the following hardware configurations.

6.5.1. Minimum Setup (Functional)

Good for development or low-traffic testing.

- **CPU:** 2 Cores.
- **Memory:** 4 GB (Enough for the Bloom Filter + OS cache).
- **Storage:** 50 GB SSD.

6.5.2. Ideal Setup (High Performance)

Recommended for the final submission/deployment.

- **CPU:** 8 Cores (To handle parallel JSON processing).
- **Memory: 64 GB**.
  - *Reasoning:* Since the database is only ∼30 GB, having 64 GB of RAM allows us to load the *entire* database into memory (using 'block_cache'). This eliminates disk reads for lookups and could push performance over **100,000 queries per second**.
- **Storage:** 500 GB NVMe SSD.
  - *Reasoning:* We need the space to download and load the RocksDB database.