Authenticating Multi-Chain Queries: Verifiable Virtual Filesystem Is All You Need

Haixin Wang
Hong Kong Baptist University
Hong Kong, China
hxwang@comp.hkbu.edu.hk

Cheng Xu
Hong Kong Baptist University
Hong Kong, China
chengxu@comp.hkbu.edu.hk

Ce Zhang
Hong Kong Baptist University
Hong Kong, China
cezhang@comp.hkbu.edu.hk

Haibo Hu
Hong Kong Polytechnic University
Hong Kong, China
haibo.hu@polyu.edu.hk

Shikun Tian
Digital Technologies, Ant Group
Hangzhou, China
shikun.tiansk@antgroup.com

Shenglong Chen
Digital Technologies, Ant Group
Hangzhou, China
shenglong.chensl@antgroup.com

Ying Yan
Digital Technologies, Ant Group
Beijing, China
fuying.yy@antgroup.com

Jianliang Xu
Hong Kong Baptist University
Hong Kong, China
xujl@comp.hkbu.edu.hk

Abstract

This demonstration showcases a verifiable multi-chain querying system. The proliferation of blockchain-driven applications has created a demand for on-chain data analysis across multiple blockchains. However, existing solutions face challenges in seamlessly integrating with existing blockchains, maintaining compatible with various database engines, and ensuring the integrity of query results. In response, we develop a blockchain indexing system that utilizes a novel verifiable virtual filesystem (V^{2}FS) for query authentication. Our demonstration focuses on highlighting the key features of our system, including blockchain data indexing, verifiable multi-chain query processing, and the usability and performance of our solution. Additionally, we provide an interactive visualization module to enhance attendees' understanding and insights.

CCS Concepts

• **Information systems → Database query processing**.

Keywords

Query processing; Data integrity; Blockchain

ACM Reference Format:
Haixin Wang, Cheng Xu, Ce Zhang, Haibo Hu, Shikun Tian, Shenglong Chen, Ying Yan, and Jianliang Xu. 2025. Authenticating Multi-Chain Queries: Verifiable Virtual Filesystem Is All You Need. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25), June 22–27, 2025, Berlin, Germany.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3722212.3725082

SIGMOD-Companion '25, Berlin, Germany

ACM ISBN 979-8-4007-1564-8/2025/06
https://doi.org/10.1145/3722212.3725082

1 Introduction

The proliferation of blockchain-driven applications has created a growing demand for on-chain data analysis. With the emergence of diverse blockchains, it becomes indispensable to query data from multiple blockchains for tasks like data aggregation and correlation. For example, a financial analyst may need to analyze transaction data from multiple blockchains to gain insight into market trends and make informed investment decisions.

However, several challenges are associated with handling multi-chain queries. First, the querying system should integrate seamlessly with existing blockchains without requiring modifications. Second, the system needs to be adaptable to various database engines and support diverse query types. Third, the system should allow users to verify the integrity of the query results.

Unfortunately, existing solutions fail to meet all these requirements simultaneously. For instance, The Graph [1] is a decentralized protocol for blockchain indexing services. It incentivizes a set of nodes known as *indexers* to aggregate data from multiple blockchains and provide flexible query services. While it features a dispute mechanism that allows users to challenge query responses from indexers, it does not guarantee integrity for all queries and suffers from long dispute resolution delays. To enable strong integrity assurance and support diverse query types, one might explore general verifiable computation (VC) schemes that can generate cryptographic proofs for arbitrary computing tasks. Nevertheless, general VC schemes have drawbacks in terms of high time complexity and significant engineering challenges, which make them impractical for real-world query applications.

To fully address the challenges of blockchain compatibility, database compatibility, and strong integrity assurance, we develop a novel blockchain indexing system that employs a *verifiable virtual filesystem* (V^{2}FS) proposed in our prior work [5]. Unlike traditional approaches that focus on *verifying computation*, V^{2}FS employs *data verification* for query authentication. In the system, the client leverages an off-the-shelf database engine to evaluate queries using data fetched on-demand from a remote indexing service provider (ISP). A

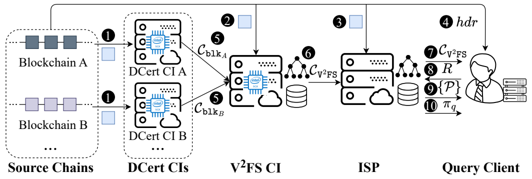

Blockchain A
Blockchain B
...
DCert CI A
DCert CI B
...
❶
❶
❷
❸
❹ hdr
❺ $\mathcal{C}_{\texttt{blk}_A}$
❺ $\mathcal{C}_{\texttt{blk}_B}$
❻ $\mathcal{C}_{\texttt{V}^2\texttt{FS}}$
❼ $\mathcal{C}_{\texttt{V}^2\texttt{FS}}$
❽ R
❾ $\{\mathcal{P}\}$
❿ π_q
Source Chains
DCert CIs
V^{2}FS CI
ISP
Query Client

Figure 1: System Model

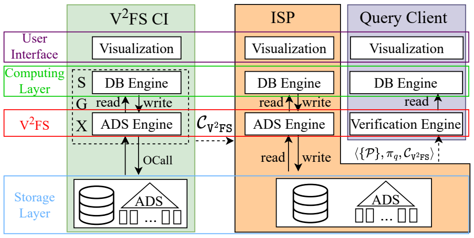


Figure 2: System Architecture

Merkle-based authenticated data structure (ADS) is used to authenticate the data, which offers both efficiency and a strong integrity guarantee. V^{2}FS acts as a pluggable module that seamlessly bridges the client's query evaluation layer and the ISP's storage layer. This enables a smooth integration with existing database engines. To ensure compatibility with various blockchains, the system leverages the DCert framework [3], a decentralized certification framework compatible with any existing blockchains, to certify blocks from different blockchains. To our best knowledge, our system is the first to simultaneously achieve blockchain compatibility, database compatibility, and strong integrity assurance. In this demonstration, we aim to showcase the usability and performance of our system, including a visualization module that provides attendees with an interactive experience of verifiable multi-chain query processing.

The rest of the demonstration proposal is organized as follows. Section 2 provides an overview of our system and explains the techniques used. Section 3 describes the system implementation. Section 4 presents the interface and demonstration details.

2 Technical Background

This section outlines our system design, including system architecture, database maintenance, and query processing, followed by optimizations to improve the query efficiency.

2.1 System Design

System Overview. Our system model, depicted in Figure 1, consists of five parties: (i) *Source Chains*, (ii) *DCert Certificate Issuers (DCert CIs)*, (iii) *V^{2}FS Certificate Issuer (V^{2}FS CI)*, (iv) *Indexing Service Provider (ISP)*, and (v) *Query Client*. Source Chains refer to existing blockchains that synchronize data with other parties. A DCert CI is incorporated for each blockchain to certify the latest state of the blockchain. The V^{2}FS CI is responsible for certifying the integrity of our proposed V^{2}FS. The ISP indexes the data from source chains and provides verifiable query services. The query client is a lightweight node which maintains up-to-date block headers from source chains and processes queries using the data from the ISP.

When a new block is added to a source chain, it is sent to the corresponding DCert CI, V^{2}FS CI, and ISP (❶, ❷, and ❸ in Figure 1). Simultaneously, the block header is broadcasted to the client (❹). Then, the DCert CI generates a DCert certificate ($\mathcal{C}_{\texttt{blk}}$) to certify the current state of the blockchain and transmits it to the V^{2}FS CI (❺). After validating the $\mathcal{C}_{\texttt{blk}}$, the V^{2}FS CI updates the data storage and constructs a V^{2}FS certificate $\mathcal{C}_{\texttt{V}^2\texttt{FS}}$ securely inside an SGX enclave [2] (❻). The $\mathcal{C}_{\texttt{V}^2\texttt{FS}}$ serves as the ADS to certify the integrity of V^{2}FS. The ISP manages its own storage, following a procedure similar to the V^{2}FS CI when it receives a new block. The only difference is that no SGX is used since we do not rely on the trustworthiness of the ISP. During query processing, the client firstly requests a certificate $\mathcal{C}_{\texttt{V}^2\texttt{FS}}$ from the ISP and verifies it against the latest block headers observed in the blockchain network (❼). After that, it evaluates the query locally using data retrieved from the ISP (❽ and ❾). At the end of query processing, a *verification object* (VO) consisting of a Merkle proof proof π_q is returned to verify the data integrity (❿).

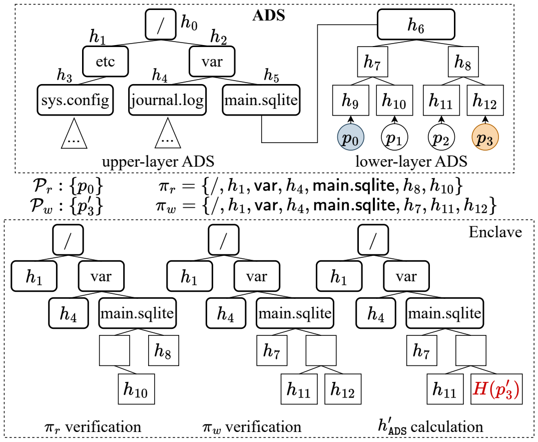


Figure 3: V^{2}FS ADS

System Architecture. Figure 2 illustrates our four-layer architecture. The user interface includes a visualization module that visualizes the workflow of V^{2}FS CI, ISP, and client interactively. To ensure maximum database compatibility, V^{2}FS is designed to decouple storage from computation. It extends the widely-adopted POSIX I/O interface and support off-the-shelf database engines. As shown in Figure 2, V^{2}FS acts as a middleware between the computing and storage layers. The computing layer contains a database engine, responsible for computational tasks in query evaluation and database updates. Meanwhile, the storage layer stores the underlying data synchronized from source chains. The data storage adopts a common filesystem as its primary I/O interface, which stores underlying data as regular *files.* It is responsible for providing *pages* of these *files* to the computing layer as needed. To establish strong integrity guarantee, we integrate a Merkle-based ADS into the storage layer. The V^{2}FS, consisting of a verification engine and ADS engine, offers trustworthy integrity verification and certificate construction during query processing and V^{2}FS maintenance.

To summarize, the adoption of the DCert framework allows our system to be compatible with existing blockchains. The design of V^{2}FS enables seamless integration with various database engines. The incorporation of a Merkle-based ADS provides efficient and strong integrity assurance.

V^{2}FS Maintenance. As mentioned earlier, an ADS is used to verify the integrity of V^{2}FS. To this end, we propose a two-layer

Merkle tree [4] structure as the ADS to authenticate the filesystem. The lower-layer ADS is a complete binary tree built on the pages of each file, while the upper-layer ADS is a trie structure built on the file path. For example, $h_7 = H(h_9||h_{10})$ and $h_2 = H(\text{var}||H(h_4||h_5))$, as shown in Figure 3. The root digests of lower-layer ADSes serve as leaves of the upper-layer ADS. The root digest h_0 of the upper-layer ADS is used to authenticate all the files in the storage layer.

During V^{2}FS maintenance, updates are made to relevant files in the filesystem, and the ADS is accordingly maintained for subsequent verifiable query processing. In a decentralized environment where the trustworthiness of the V^{2}FS CI cannot be assumed, we propose a solution that leverages an SGX enclave to handle database updates. As depicted in Figure 2, the enclave serves as the host for both the database engine and ADS engine, while the storage layer is positioned outside the enclave to optimize enclave memory usage. The database engine is responsible for performing computations based on the new block, while utilizing ADS engine for data access through *read* and *write* operations. These operations are translated into outside calls (OCalls) to interact with the storage layer outside the enclave. Since the outside-enclave storage is inherently untrusted, the ADS engine provides verification for data retrieved from the storage layer. Upon completing the database computation, the enclave program requests the external storage layer to generate Merkle proof π_r for the accessed pages during the computation. Additionally, the storage layer provides the corresponding Merkle paths π_w for updating the ADS. If these Merkle proofs can be successfully verified against the previous ADS root signed by the previous $C_{\mathsf{V^2FS}}$, the enclave program proceeds to compute a new ADS root. Subsequently, a new certificate $C'_{\mathsf{V^2FS}}$ is generated to authenticate the updated database and ADS.

Example. *In the example shown in Figure 2, assume that a new block involves (i) reading p_0; and (ii) updating p_3 to p'_3 in the file /var/main.sqlite. At the end of database updates, two Merkle proofs $\pi_r = \{/, h_1, var, h_4, main.sqlite, h_8, h_{10}\}$ and $\pi_w = \{/, h_1, var, h_4, main.sqlite, h_7, h_{11}, h_{12}\}$ are generated and passed to the enclave. To verify the integrity of the accessed pages, the enclave program uses π_r and π_w to reconstruct the ADS root. If the reconstructed ADS root is the same as h_0, it implies that the page p_0 retrieved to the enclave via OCall and the neighboring nodes related to p_3 in the ADS are correct. The enclave program computes the new ADS root $h'_0 = H(/||H(h_1||H(var||H(h_4||H(main.sqlite||H(h_7||H(h_{11}||H(p'_3))))))))$ and generates a new V^{2}FS certificate $C'_{\mathsf{V^2FS}}$ using h'_{ADS}. Finally, p'_3 is flushed to the external storage and the ADS is updated accordingly.*

Query Processing. As depicted in Figure 2, the client utilizes the same database engine as the V^{2}FS CI and ISP for query processing. The storage layer is located at the ISP. Therefore, the database engine accesses the underlying storage through the V^{2}FS via network communication. This involves retrieving data pages from the ISP as needed. Additionally, the ISP provides the necessary Merkle proofs π_q and the corresponding $C_{\mathsf{V^2FS}}$ as a *verification object* (VO) for integrity validation. To reduce communication costs, the ISP consolidates all Merkle proofs and transmits a single VO at the end of query processing. With this VO, the client can verify that the database engine is utilizing data from the latest blocks in the source chains through $C_{\mathsf{V^2FS}}$, and ensure the correctness of all received pages using the Merkle proofs π_q and h_0 in $C_{\mathsf{V^2FS}}$.

2.2 Query Optimizations

Observing that network transmission bottlenecks query processing, we propose three cache-based strategies to improve performance.

Query with Intra-query Cache. We have noticed that certain pages are visited repeatedly within a single query during query processing. To reduce the network overhead, the client can employ an intra-query cache which keeps recently accessed pages in memory. Once a page is required, the client first checks if it is already in the intra-query cache. A page presenting in the intra-query cache can be directly fetched for subsequent computing. This eliminates the need for network retrieval and boosts query performance.

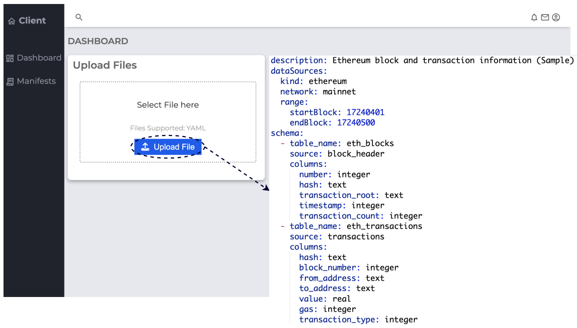
Query with Inter-query Cache. Similarly, caching commonly accessed pages in memory can reduce the impact of repeated network requests. To address the issue of cached pages becoming stale due to blockchain updates between queries, we propose an efficient inter-query cache structure. It allows the client to confirm the freshness of multiple pages with a single request. The cache comprises multiple Merkle subtrees that include cached pages and their ancestor nodes in the ADS. Each node is tagged with a freshness flag indicating *fresh* or *unknown*. If a required page in the cache is marked as *unknown*, the client sends its complete Merkle path to the ISP. The ISP then traverses the path top-down. If a digest in the path matches its counterpart in the current ADS, it indicates that the page, along with other pages covered by the matching node, are *fresh*. The ISP confirms this by returning the matching node and generates a Merkle proof. The client verifies the Merkle proof to ensure the integrity of the ISP's response and forwards the cached page to the database engine for query processing. Conversely, if none of the digests in the path align with their corresponding nodes in the ADS, it signifies that the requested page has been updated, and the ISP returns the updated page. The client adds the updated page to the cache and returns it to the database engine.

Bloom Filter Integrated Freshness Checking. To further reduce network communication costs, we design a versioned bloom filter (VBF) that summarizes the historical update information of all pages. The VBF is managed by the V^{2}FS CI inside the SGX enclave. When query processing reveals a needed page in the cache marked as *unknown*, the VBF steps in first to check its freshness. If the VBF indicates that the page has not been updated since last access, it is ensured to be *fresh*. Otherwise, the VBF cannot guarantee the freshness of the page due to potential false positives of the bloom filter. In this case, the client reverts to freshness validation through the inter-query cache.

3 System Implementation

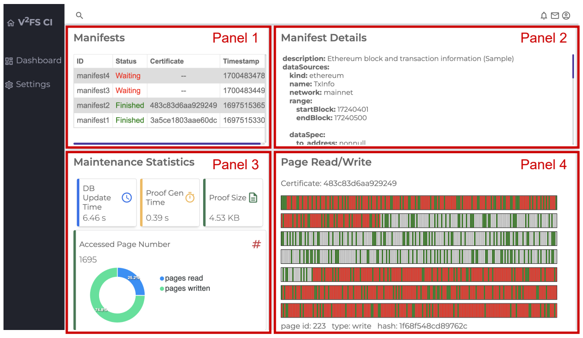
We implement a prototype system using Rust programming language. The system architecture, as depicted in Figure 2, comprises four layers. At the base, the storage layer manages the underlying data and a Merkle-based ADS using 256-bit BLAKE2b[1] as the hash function and RocksDB[2] as the underlying storage. The V^{2}FS middleware extends POSIX I/O interface with integrity checks, bridging the storage and computing layers. For the V^{2}FS CI and ISP, an ADS engine handles storage layer updates and proof verification during V^{2}FS maintenance. On the client side, a verification engine enables

[1] https://github.com/oconnor663/blake2_simd
[2] https://rocksdb.org



```
description: Ethereum block and transaction information (Sample)
dataSources:
  kind: ethereum
  network: mainnet
  range:
    startBlock: 17240401
    endBlock: 17240500
schema:
  - table_name: eth_blocks
    source: block_header
    columns:
      number: integer
      hash: text
      transaction_root: text
      timestamp: integer
      transaction_count: integer
  - table_name: eth_transactions
    source: transactions
    columns:
      hash: text
      block_number: integer
      from_address: text
      to_address: text
      value: real
      gas: integer
      transaction_type: integer
```

Figure 4: Manifest Uploading



ID	Status	Certificate	Timestamp
manifest4	Waiting	--	1700483478
manifest3	Waiting	--	1700483449
manifest2	Finished	483c83d6aa929249	1697515365
manifest1	Finished	3a5ce1803aae60dc	1697515330

description: Ethereum block and transaction information (Sample)
dataSources:
kind: ethereum
name: TxInfo
network: mainnet
range:
startBlock: 17240401
endBlock: 17240500

dataSpec:
to_address: nonnull

DB Update Time 6.46 s
Proof Gen Time 0.39 s
Proof Size 4.53 KB
Accessed Page Number 1695

Certificate: 483c83d6aa929249
page id: 223 type: write hash: 1f68f548cd89762c

Figure 5: V^{2}FS CI Dashboard

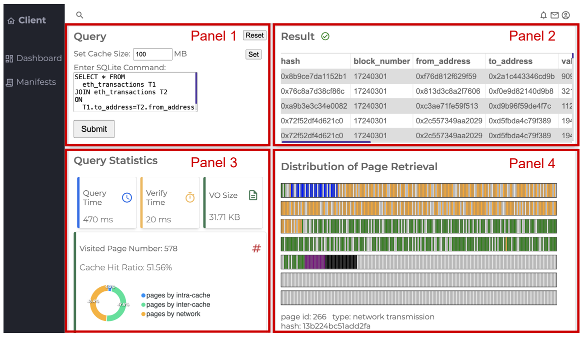
integrity verification for retrieved data during query processing. The computing layer encompasses an off-the-shelf database engine responsible for query evaluation, which is implemented using the Rusqlite library.[3] For the V^{2}FS CI, the database engine and ADS engine run inside an SGX enclave to construct trustworthy certificate. Finally, the user interface layer includes a visualization module, which showcases the system working process and provides an interactive demonstration experience.

4 Demonstration Scenarios

Our demonstration showcase how our system handles blockchain data indexing and verifiable multi-chain queries in four scenarios.

Scenario 1: Subscribing to Multi-chain Indexing Services. Users can subscribe to multi-chain indexing services by uploading a manifest file through the uploading interface (Figure 4). The manifest, represented as a YAML file, provides essential information for the indexing service, including data sources, types, target table schema, and other metadata. The manifests processing information can be viewed in the V^{2}FS CI dashboard (Figure 5). Users can check the status and detailed information of a manifest on Panel 1 and Panel 2, respectively. Once a manifest is processed, users can view the maintenance performance, including database update time, proof generation time, proof size, and the number of accessed pages. Meanwhile, Panel 4 presents the newly constructed $C_{\mathrm{V^2FS}}$ and visualizes the distribution of page access.

Scenario 2: Verifying Multi-chain Queries. Users can issue SQLite queries and verify their results through the client dashboard (Figure 6). Upon submitting a query on Panel 1, the query results are displayed on Panel 2. A ✓ mark indicates successful verification, signifying that the query results are (i) *sound*: correct and untampered with, and (ii) *complete*: including all valid results. On the other hand, a ✖ mark reveals one of the following conditions: (i) the $C_{\mathrm{V^2FS}}$ fails to be verified against the latest blockchain states and SGX public key; or (ii) certain page retrieved from the ISP are tampered or missing. Users can check the performance metrics for query processing such as query processing time and verification time on Panel 3. Additionally, Panel 3 presents the cache hit ratio and the number and types of visited pages. This information allows users to understand the query efficiency based on the effectiveness of the cache structures. Moreover, Panel 4 offers a visualization of page access during query processing for better insights.



Set Cache Size: 100 MB

Enter SQLite Command:

```
SELECT * FROM
  eth_transactions T1
JOIN eth_transactions T2
ON
  T1.to_address=T2.from_address
```

hash	block_number	from_address	to_address	val
0x8b9ce7da1152b1	17240301	0xf76d812f629f59	0x2a1c443346cd9b	90
0x76c8a7d38cf86c	17240301	0x813d3c8a2f7606	0xf0e9d82140d9b8	32
0xa9b3e3c34e0082	17240301	0xc3ae71fe59f513	0xd9b96f59de4f7c	112
0x72f52df4d621c0	17240301	0x2c557349aa2029	0xd5fbda4c79f389	194
0x72f52df4d621c0	17240301	0x2c557349aa2029	0xd5fbda4c79f389	194

Query Time 470 ms
Verify Time 20 ms
VO Size 31.71 KB
Visited Page Number: 578
Cache Hit Ratio: 51.56%

page id: 266 type: network transmission
hash: 13b224bc51add2fa

Figure 6: Client Dashboard

Scenario 3: Understanding the Impact of Queries. Users can explore the impact of different queries on system performance. On the one hand, different data distributions for the same query can lead to varying cache utilization. On the other hand, different query types may have different complexity, which causes various cache hit ratios. By varying query distributions and types, users can observe their influence on the cache hit ratio, consequently affecting the overall query performance.

Scenario 4: Understanding the Impact of Cache Algorithms. Users can also analyze the query performance and page access patterns of cache-based algorithms to understand the effectiveness of optimizations. Specifically, users can experiment with different optimization approaches and adjust the cache size when entering a query on Panel 1 of Figure 6. Then they can compare the query performance in terms of query processing time, verification time, cache hit ratio, and VO size. Additionally, they can analyze the page access distributions resulting from various optimization techniques and cache sizes.

Acknowledgments

This work was supported by Digital Technologies, Ant Group and Hong Kong RGC Grants 12200022, C2004-21GF, and C2003-23Y.

References

[1] [n. d.]. The Graph. https://thegraph.com. Accessed: 2022-12-30.
[2] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *Cryptology ePrint Archive, Paper 2016/086* (2016), 1–118.
[3] Yang Ji, Cheng Xu, Ce Zhang, and Jianliang Xu. 2022. DCert: Towards Secure, Efficient and Versatile Blockchain Light Clients. In *Proc. Middleware.*
[4] Ralph C. Merkle. 1990. A Certified Digital Signature. In *Proc. CRYPTO.* 218–238.
[5] Haixin Wang, Cheng Xu, Xiaojie Chen, Ce Zhang, Haibo Hu, Shikun Tian, Ying Yan, and Jianliang Xu. 2024. V^{2}FS: A Verifiable Virtual Filesystem for Multi-Chain Query Authentication. In *Proc. ICDE.* 1999–2011.

[3] https://github.com/rusqlite/rusqlite