vChain: Enabling Verifiable Boolean Range Queries over Blockchain Databases

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

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

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

ABSTRACT

Blockchains have recently been under the spotlight due to the boom of cryptocurrencies and decentralized applications. There is an increasing demand for querying the data stored in a blockchain database. To ensure query integrity, the user can maintain the entire blockchain database and query the data locally. However, this approach is not economic, if not infeasible, because of the blockchain's huge data size and considerable maintenance costs. In this paper, we take the first step toward investigating the problem of verifiable query processing over blockchain databases. We propose a novel framework, called vChain, that alleviates the storage and computing costs of the user and employs verifiable queries to guarantee the results' integrity. To support verifiable Boolean range queries, we propose an accumulator-based authenticated data structure that enables dynamic aggregation over arbitrary query attributes. Two new indexes are further developed to aggregate intra-block and inter-block data records for efficient query verification. We also propose an inverted prefix tree structure to accelerate the processing of a large number of subscription queries simultaneously. Security analysis and empirical study validate the robustness and practicality of the proposed techniques.

KEYWORDS

Query processing; Data integrity; Blockchain

ACM Reference Format:
Cheng Xu, Ce Zhang, and Jianliang Xu. 2019. vChain: Enabling Verifiable Boolean Range Queries over Blockchain Databases. In *Proceedings of 2019 International Conference on Management of Data (SIGMOD'19).* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

SIGMOD'19, June 30 – July 5, 2019, Amsterdam, NL

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Owing to the success of cryptocurrencies such as Bitcoin [1] and Ethereum [2], blockchain technology has been gaining overwhelming momentum in recent years. A blockchain is an append-only data structure that is distributively stored among peers in the network. Although peers in the network may not trust each other, a blockchain ensures data integrity from two aspects. First, powered by the hash chain technique, data stored on a blockchain are immutable. Second, thanks to its consensus protocol, a blockchain guarantees that all peers maintain identical replicas of the data. These cryptographically guaranteed security mechanisms, together with the decentralization and provenance properties of a blockchain, make blockchains a potential technology to revolutionize database systems [3, 4, 5, 6, 7].

From the database perspective, a blockchain can be seen as a database storing a large collection of timestamped data records. With widespread adoption of blockchains for data-intensive applications such as finance, supply chains, and IP rights management, there is an increasing demand from users to query the data stored in a blockchain database. For example, in the Bitcoin network, users may want to find the transactions that satisfy a variety of *range* selection predicates, such as *"Transaction Fee ≥ $50"* and *"$0.99M ≤ Total Output ≤ $1.01M"* [8]. In a blockchain-based patent management system, users can use *Boolean* operators to search for combinations of keywords, such as *"Blockchain"*∧(*"Query"*∨*"Search"*), in the patents' abstracts [9]. Whereas many companies, including database giants IBM, Oracle, and SAP, as well as startups such as FlureeDB [10], BigchainDB [11], and SwarmDB [12], have devoted their efforts to developing blockchain database solutions to support SQL-like queries, all of them assume the existence of a trusted party who can faithfully execute user queries that are based on a materialized view of the blockchain database. However, such a trusted party may not always exist and the integrity of query results cannot be guaranteed. Query processing with integrity assurance remains an unexplored issue in blockchain research.

In a typical blockchain network [1, 2],[1] there are three types of nodes as shown in Fig. 1: *full node*, *miner*, and *light*

[1]For ease of exposition, in this paper we focus our discussion on public blockchains, but the proposed verifiable query techniques can be easily extended to private blockchains.

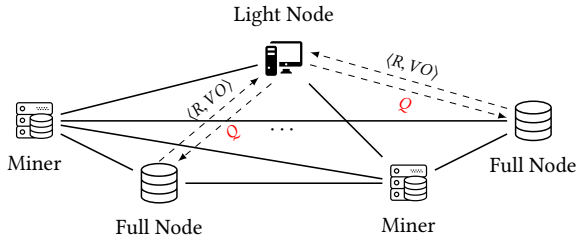


Figure 1: A Blockchain Network

node. A full node stores all the data in the blockchain, including block headers and data records. A miner is a full node with great computing power, responsible for constructing *consensus proofs* (e.g., *nonce* in the Bitcoin blockchain). A light node stores only block headers, which include the consensus proof and the cryptographic hashes of a block. Note that the data records are not stored in light nodes.

To ensure the integrity of queries over a blockchain database, the query user could join the blockchain network as a full node. Then, the user can download and validate the entire database and process queries locally without compromising the query integrity. However, maintaining a full copy of the entire database might be too costly to an ordinary user, as it requires considerable storage, computing, and bandwidth resources. For example, the minimum requirements of running a Bitcoin full node include 200GB of free disk space, an unmetered broadband connection with upload speeds of at least 50KB per second, and a running time of 6 hours a day [13]. To cater to query users with limited resources, especially mobile users, a more appealing alternative is to delegate the storage and query services to a powerful full node, while the query user only acts as a light node to receive results. Nevertheless, how to ensure the integrity of query results remains a challenge because full nodes are untrusted and that is an intrinsic assumption of the blockchain.

To address the aforementioned query integrity issue, in this paper, we propose a novel framework, called vChain, that employs *verifiable query processing* to guarantee the results' integrity. More specifically, we augment each block with some additional *authenticated data structure* (ADS), based on which an (untrusted) full node can construct and return a cryptographic proof, known as *verification object* (VO), for users to verify the results of each query. The communication between a query user (light node) and a full node is illustrated in Fig. 1, where Q denotes a query request and R denotes the result set.

It is worth noting that this vChain framework is inspired by query authentication techniques studied for outsourced databases [14, 15, 16, 17, 18]. However, there are several key differences that render the conventional techniques inapplicable to blockchain databases. First, the conventional techniques rely on a data owner to sign the ADS using a private key. In contrast, in the blockchain network there is no data owner. Only the miners can append new data to the blockchain by constructing consensus proofs according to the consensus protocol. However, they cannot act as the data owner because they cannot hold the private key and sign the ADS. Second, a conventional ADS is built on a fixed dataset, and such an ADS cannot be efficiently adapted to a blockchain database in which the data are unbounded. Third, in traditional outsourced databases, new ADSs can always be generated and appended, as needed, to support more queries involving different sets of attributes. However, that would be difficult due to the immutability of the blockchain, where a one-size-fits-all ADS is more desirable to support dynamic query attributes.

Clearly, the design of the ADS is a key issue of the vChain framework. To address this issue, this paper focuses on *Boolean range queries*, which, as illustrated earlier, are commonly found in blockchain applications [8, 9]. We propose a novel accumulator-based ADS scheme that enables dynamic aggregation over arbitrary query attributes, including both numerical attributes and set-valued attributes. This newly designed ADS is independent of the consensus protocol so that it is compatible with the current blockchain technology. On that basis, efficient verifiable query processing algorithms are developed. We also propose two authenticated indexing structures for intra-block data and inter-block data, respectively, to enable batch verification. To support large-scale subscription queries, we further propose a query indexing scheme that can group similar query requests. To summarize, our contributions made in this paper are as follows:

- To the best of our knowledge, this is the first work on verifiable query processing that leverages built-in ADSs to achieve query integrity for blockchain databases.
- We propose a novel vChain framework, together with a new ADS scheme and two indexing structures that can aggregate intra-block and inter-block data records for efficient query processing and verification.
- We develop a new query index that can handle a large number of subscription queries simultaneously.
- We conduct a security analysis as well as an empirical study to validate the proposed techniques. We also address the practical implementation issues.

The rest of the paper is organized as follows. Section 2 reviews existing studies on blockchains and verifiable query processing. Section 3 introduces the formal problem definition, followed by cryptographic primitives in Section 4. Section 5 presents our basic solution, which is then improved by two indexing structures devised in Section 6. The verifiable subscription query is discussed in Section 7. The security analysis is presented in Section 8. Section 9 presents the experimental results. Finally, we conclude our paper in Section 10.

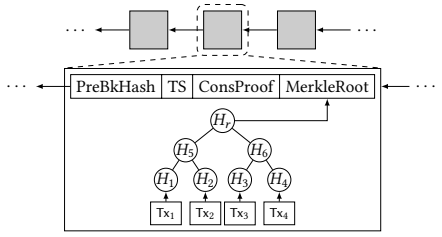


Figure 2: Blockchain Structure

2 RELATED WORK

In this section, we briefly review related studies and discuss relevant techniques.

Blockchain. Since the introduction of Bitcoin cryptocurrency, blockchain technology has received considerable attention from both academia and industry [1, 2, 5]. A blockchain, which is essentially a special form of *Merkle Hash Tree* (MHT) [19], is constructed as a sequence of blocks. As shown in Fig. 2, each block stores a list of transaction records and an MHT built on top of them. The header of each block consists of four components: (i) *PreBkHash*, which is the hash of the previous block; (ii) *TS*, which is the timestamp when the block was created; (iii) *ConsProof*, which is constructed by the miners and guarantees the consensus of the block; and (iv) *MerkleRoot*, which is the root hash of the MHT. The *ConsProof* is usually computed based on the *PreBkHash* and *MerkleRoot*, and varies depending on the consensus protocol. In the widely used *Proof of Work* (PoW) consensus protocol, the *ConsProof* is a *nonce* computed by the miners such that:

$$\mathsf{hash}(PreBkHash \mid TS \mid MerkleRoot \mid nonce) \leq Z$$

where Z corresponds to the mining difficulty. After a miner finds the nonce, it will pack the new block and broadcast it to the entire network. Other miners verify the transaction records and the nonce of the new block and, once verified, append it to the blockchain.

Significant effort has been made to address the various issues of blockchain systems, including system protocols [20, 21], consensus algorithms [22, 23], security [24, 25], storage [7], and performance benchmarking [4]. Recently, major database vendors, including IBM [26], Oracle [27], and SAP [28], all have integrated blockchains with their database management systems, and they allow users to execute queries over blockchains through a database frontend. Besides, many startups such as FlureeDB [10], BigchainDB [11], and SwarmDB [12], have been developing blockchain-based database solutions for decentralized applications. However, they generally separate query processing from the underlying blockchain storage and count on trusted database servers for query integrity assurance. In contrast, our proposed vChain solution builds authenticated data structures into the blockchain structure, so that even untrusted servers can be enabled to offer integrity-assured query services.

Verifiable Query Processing. Verifiable query processing techniques have been extensively studied to ensure result integrity against an untrusted service provider (e.g., [14, 15, 16, 17, 18, 29]). Most of the existing studies focus on outsourced databases and there are two typical approaches: supporting general queries using circuit-based verifiable computation (VC) techniques and supporting specific queries using an authenticated data structure (ADS). The VC-based approach (e.g., SNARKs [30]) can support arbitrary computation tasks but at the expense of a very high and sometimes impractical overhead. Moreover, it entails an expensive preprocessing step as both the data and the query program need to be hard-coded into the proving key and the verification key. To remedy this issue, Ben-Sasson et al. [31] have developed a variant of SNARKs in which the preprocessing step is only dependent on the upper-bound size of the database and query program. More recently, Zhang et al. [29] have proposed a vSQL system, which utilizes an interactive protocol to support verifiable SQL queries. However, it is limited to relational databases with a fixed schema and cannot work with (unbounded) set-valued data.

The ADS-based approach in comparison is generally more efficient as it tailors to specific queries. Our proposed solution belongs to this approach. Two types of structures are commonly used to serve as an ADS: digital signature and MHT. Digital signatures authenticate the content of a digital message based on asymmetric cryptography. To support verifiable queries, it requires every data record to be signed and hence cannot scale up to large datasets [14]. MHT, on the other hand, is built on a hierarchical tree [19]. Each entry in a leaf node is assigned a hash digest of a data record, and each entry in an internal node is assigned a digest derived from the child nodes. The data owner signs the root digest of MHT, which can be used to verify any subset of data records. MHT has been widely adapted to various index structures [15, 16, 17]. More recently, there have been studies of verifiable queries on set-valued data [32, 33, 34, 35, 36].

Another closely related line of research is on verifiable query processing for data streams [37, 38, 39, 40]. However, previous studies [38, 39] focus on *one-time* queries to retrieve the latest version of streamed data. [40] requires the data owner to maintain an MHT for all data records and suffers from long query latency, which is not suitable for real-time streaming services. On the other hand, subscription queries over data streams have been investigated in [41, 42, 43]. So far, no work has considered the integrity issue for subscription queries over blockchain databases.

3 PROBLEM DEFINITION

As was explained in Section 1, this paper proposes a novel vChain framework and studies verifiable query processing

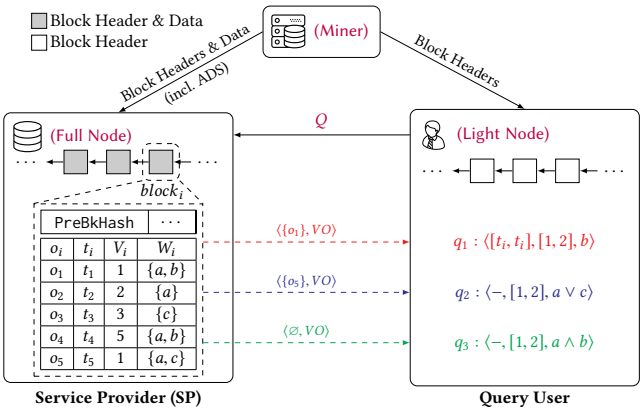


Figure 3: System Model of vChain

over blockchain databases. Fig. 3 shows the system model of vChain, which involves three parties: (i) *miner*, (ii) *service provider* (SP), and (iii) *query user*. Both the miner and the SP are full nodes that maintain the entire blockchain database. The query user is a light node that keeps track of the block headers only. The miner is responsible for constructing the consensus proofs and appending new blocks to the blockchain. The SP provides query services to the lightweight user.

The data stored in the blockchain can be modeled as a sequence of blocks of temporal objects $\{o_1, o_2, \cdots, o_n\}$. Each object o_i is represented by $\langle t_i, V_i, W_i \rangle$, where t_i is the timestamp of the object, V_i is a multi-dimensional vector that represents one or more numerical attributes, and W_i is a set-valued attribute. To enable verifiable query processing, an *authenticated data structure* (ADS) is constructed and embedded into each block by the miners (to be detailed in Sections 5-7). We consider two forms of Boolean range queries: (historical) time-window queries and subscription queries.

Time-Window Queries. Users may wish to search the records appearing in a certain time period. In such a case, a time-window query can be issued. Specifically, a time-window query is in the form of $q = \langle [t_s, t_e], [\alpha, \beta], \Upsilon \rangle$, where $[t_s, t_e]$ is a temporal range selection predicate for the time period, $[\alpha, \beta]$ is a multi-dimensional range selection predicate for the numerical attributes, and Υ is a monotone Boolean function on the set-valued attribute. As a result, the SP returns all objects such that $\{o_i = \langle t_i, V_i, W_i \rangle \mid t_i \in [t_s, t_e] \wedge V_i \in [\alpha, \beta] \wedge \Upsilon(W_i) = 1\}$. For simplicity, we assume that Υ is in a *conjunctive normal form* (CNF).

Example 3.1. In a Bitcoin transaction search service, each object o_i corresponds to a coin transfer transaction. It consists of a transfer amount stored in V_i and a set of sender/receiver addresses stored in W_i. A user may issue a query $q = \langle [2018\text{-}05, 2018\text{-}06], [10, +\infty], \texttt{send:1FFYc} \wedge \texttt{receive:2DAAf} \rangle$ to find all of the transactions happening from May to June of 2018 with a transfer amount larger than 10 and being associated with the addresses "`send:1FFYc`" and "`receive:2DAAf`".

Subscription Queries. In addition to time-window queries, users can register their interests through subscription queries. Specifically, a subscription query is in the form of $q = \langle -, [\alpha, \beta], \Upsilon \rangle$, where $[\alpha, \beta]$ and Υ are identical to the query conditions in time-window queries. In turn, the SP continuously returns all objects such that $\{o_i = \langle t_i, V_i, W_i \rangle \mid V_i \in [\alpha, \beta] \wedge \Upsilon(W_i) = 1\}$ until the query is deregistered.

Example 3.2. In a blockchain-based car rental system, each rental object o_i consists of a rental price stored in V_i and a set of textual keywords stored in W_i. A user may subscribe to a query $q = \langle -, [200, 250],$ "Sedan" $\wedge$ ("Benz" $\vee$ "BMW")$\rangle$ to receive all rental messages that have a price within the range [200, 250] and contain the keywords "Sedan" and "Benz" or "BMW".

Additional examples of time-window queries and subscription queries can be found in Fig. 3.

Threat Model. We consider the SP, as an untrusted peer in the blockchain network, to be a potential adversary. Due to various issues such as program glitches, security vulnerabilities, and commercial interests, the SP may return tampered or incomplete query results, thereby violating the expected security of the blockchain. To address such a threat, we adopt verifiable query processing that enables the SP to prove the integrity of query results. Specifically, during query processing, the SP examines the ADS embedded in the blockchain and constructs a *verification object* (VO) that includes the verification information of the results. The VO is returned to the user along with the results. Using the VO, the user can establish the *soundness* and *completeness* of the query results, under the following criteria:

- **Soundness.** None of the objects returned as results have been tampered with and all of them satisfy the query conditions.
- **Completeness.** No valid result is missing regarding the query window or subscription period.

The above security notions will be formalized when we perform our security analysis in Section 8.

The main challenge in this model is how to design the ADS so that it can be easily accommodated in the blockchain structure while *cost-effective* VOs (incurring small bandwidth overhead and fast verification time) can be efficiently constructed for both time-window queries and subscription queries. We address this challenge in the next few sections.

4 PRELIMINARIES

This section gives some preliminaries on cryptographic constructs that are needed in our algorithm design.

Cryptographic Hash Function. A cryptographic hash

function $\mathsf{hash}(\cdot)$ accepts an arbitrary-length string as its input and returns a fixed-length bit string. It is collision resistant and difficult to find two different messages, m_1 and m_2, such that $\mathsf{hash}(m_1) = \mathsf{hash}(m_2)$. Classic cryptographic hash functions include the SHA-1, SHA-2, and SHA-3 families.

Bilinear Pairing. Let $\mathbb{G}$ and $\mathbb{H}$ be two cyclic multiplicative groups with the same prime order p. Let g be the generator of $\mathbb{G}$. A bilinear mapping is a function $e : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{H}$ with the following properties:

- Bilinearity: If $u, v \in \mathbb{G}$ and $e(u, v) \in \mathbb{H}$, then $e(u^a, v^b) = e(u, v)^{ab}$ for any u, v.
- Non-degeneracy: $e(g, g) \neq 1$.

Bilinear pairing serves as a basic operation for the multiset accumulator as shown later in this paper.

q-Strong Diffie-Hellman (q-SDH) Assumption [44]. Let $pub = (p, \mathbb{G}, \mathbb{H}, e, g)$ be a bilinear pairing as described above. It states that for all polynomials q and for all probabilistic polynomial-time adversaries Adv,

$$\Pr[s \leftarrow \mathbb{Z}_p; \sigma = (pub, g^s, \cdots, g^{s^q}); \\ (c, h) \leftarrow \mathsf{Adv}(\sigma) : h = e(g, g)^{1/(c+s)}] \approx 0$$

q-Diffie-Hellman Exponent (q-DHE) Assumption [45]. Let $pub = (p, \mathbb{G}, g)$ as described above. It states that for all polynomials q and for all probabilistic polynomial-time adversaries Adv,

$$\Pr[s \leftarrow \mathbb{Z}_p; \sigma = (pub, g^s, \cdots, g^{s^{q-1}}, g^{s^{q+1}}, \cdots, g^{s^{2q-2}}); \\ h \leftarrow \mathsf{Adv}(\sigma) : h = g^{s^q}] \approx 0$$

Cryptographic Multiset Accumulator. A multiset is a generalization of a set in which elements are allowed to occur more than once. To represent them in a constant size, a cryptographic mulitset accumulator is a function $\mathrm{acc}(\cdot)$, which maps a multiset to an element in some cyclic multiplicative group in a collision resistant fashion [36].

One useful property of the accumulator is that it can be used to prove set disjoint. It consists of the following probabilistic polynomial-time algorithms:

- $\mathsf{KeyGen}(1^\lambda) \rightarrow (sk, pk)$: On input a security parameter 1^λ, it generates a secret key sk and a public key pk.
- $\mathsf{Setup}(X, pk) \rightarrow \mathrm{acc}(X)$: On input a multiset X and the public key pk, it computes the accumulative value $\mathrm{acc}(X)$.
- $\mathsf{ProveDisjoint}(X_1, X_2, pk) \rightarrow \pi$: On input two multisets X_1, X_2, where $X_1 \cap X_2 = \varnothing$, and the public key pk, it outputs a proof π.
- $\mathsf{VerifyDisjoint}(\mathrm{acc}(X_1), \mathrm{acc}(X_2), \pi, pk) \rightarrow \{0, 1\}$: On input the accumulative values $\mathrm{acc}(X_1)$, $\mathrm{acc}(X_2)$, a proof π, and the public key pk, it outputs 1 if and only if $X_1 \cap X_2 = \varnothing$.

More elaborated constructions of the accumulator and the set disjoint proof will be given in Section 5.2.

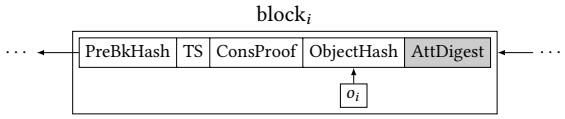


Figure 4: Extended Block Structure

5 BASIC SOLUTION

To enable verifiable queries in our vChain framework, a naive scheme is to construct a traditional MHT as the ADS for each block and apply the conventional MHT-based authentication methods. However, this naive scheme has three major drawbacks. First, an MHT supports only the query keys on which the Merkle tree is built. To support queries involving an arbitrary set of attributes, an exponential number of MHTs need to be constructed for each block. Second, MHTs do not work with set-valued attributes. Third, MHTs of different blocks cannot be aggregated efficiently, making it incapable of leveraging inter-block optimization techniques. To overcome these drawbacks, in this section we propose novel authentication techniques based on a new accumulator-based ADS scheme, which transforms numerical attributes into set-valued attributes and enables dynamic aggregation over arbitrary query attributes.

In the following, we start by considering a single object and focusing on the Boolean time-window query for ease of illustration (Sections 5.1 and 5.2). We then extend it to the range query condition (Section 5.3). We discuss the batch query processing and verification for multiple objects in Section 6. The subscription query is elaborated in Section 7.

5.1 ADS Generation and Query Processing

For simplicity, this section considers the Boolean query condition on the set-valued attribute W_i only. We assume that each block stores a single object $o_i = \langle t_i, W_i \rangle$ and use *ObjectHash* to denote *MerkleRoot* in the original block structure (Fig. 2).

ADS Generation. Recall that in the proposed vChain framework, an ADS is generated for each block during the mining process. It can be used by the SP to construct a verification object (VO) for each query. To this end, we extend the original block structure by adding an extra field, named *AttDigest*, as shown by the shaded part in Fig. 4. Thus, the block header consists of *PreBkHash*, *TS*, *ConsProof*, *ObjectHash*, and *AttDigest*.

To serve as the ADS, *AttDigest* should have three desired properties. First, *AttDigest* should be able to summarize an object's attribute W_i in a way that it can be used to prove whether or not the object matches a query condition. In case of a mismatch, we can just return this digest instead of the whole object. Second, *AttDigest* should be in a constant size regardless of the number of elements in W_i. Third, *AttDigest* should be aggregatable to support batch verification of

multiple objects within a block or even across blocks (Section 6). As such, we propose to use *multiset accumulator* as *AttDigest*:

$$AttDigest_i = \text{acc}(W_i) = \text{Setup}(W_i, pk)$$

While its supported functionalities, including ProveDisjoint(·) and VerifyDisjoint(·), have been described in Section 4, for better readability, we defer detailed constructions to Section 5.2.

Verifiable Query Processing. Given a Boolean query condition and a data object, there are only two possible outcomes: match or mismatch. The soundness of the first case can be easily verified by returning the object as a result, since its integrity can be authenticated by the *ObjectHash* stored in the block header, which is available to the query user on a light node (recall Fig. 3). The challenge lies in how to effectively verify the second case by using *AttDigest*. As CNF is a Boolean function expressed in a list of AND of OR operators, we can view the Boolean function in CNF as a list of sets. For example, a query condition "Sedan" ∧ ("Benz" ∨ "BMW") is equivalent to two sets: {"Sedan"} and {"Benz", "BMW"}. Consider a mismatching object o_i : {"Van", "Benz"}. It is easy to observe that there exists an equivalence set (i.e., {"Sedan"}) such that its intersection with the object's attribute is empty. Thus, we can apply ProveDisjoint({"Van", "Benz"}, {"Sedan"}, pk) to generate a disjoint proof π as the VO for the mismatching object. Accordingly, the user can retrieve $AttDigest_i$ = acc({"Van", "Benz"}) from the block header and use VerifyDisjoint($AttDigest_i$, acc({"Sedan"}), π, pk) to verify the mismatch. The whole process is detailed in Algorithm 1.

It is straightforward to extend the above algorithm to support time-window queries. The process basically finds the corresponding blocks whose timestamp is within the query window and invokes Algorithm 1 repeatedly for each object in these selected blocks. For example, suppose the query Boolean function is "Sedan" ∧ ("Benz" ∨ "BMW"). The list of objects that are within the time window includes o_1 : {"Sedan", "Benz"}, o_2 : {"Sedan", "Audi"}, o_3 : {"Van", "Benz"}, and o_4 : {"Van", "BMW"}. The SP can apply ProveDisjoint(·) for o_2, o_3, and o_4 to prove that they do not match the condition "Benz" ∨ "BMW", "Sedan", and "Sedan", respectively. As for o_1, the SP will return the object directly since it is a match.

5.2 Constructions of Multiset Accumulator

We now discuss two possible constructions of the multiset accumulator used as *AttDigest* in Section 5.1. Each construction has its own advantage and disadvantage, and is suitable to different application scenarios as we will see in Section 9.

5.2.1 Construction 1. We first present a construction proposed in [32], which is based on bilinear pairing and q-SDH assumption. It consists of the following algorithms.

Algorithm 1: Verifiable Query on a Single Object

ADS Generation (by the miner)
- **for** *each object* $o_i = \langle t_i, W_i \rangle$ **do**
  - $AttDigest_i \leftarrow \text{acc}(W_i)$;
  - Write $\langle \text{hash}(o_i), AttDigest_i \rangle$ to the block;

VO Construction (by the SP)
- **Function** `VOConstruction`(o_i, q)
  - **Input:** Object o_i, Query condition $q = \langle \Upsilon \rangle$
  - **if** o_i *matches* q **then** Send o_i to the user;
  - **else**
    - Interpret Υ as a list of sets $\{\Upsilon_1, \cdots, \Upsilon_\ell\}$, s.t. $\Upsilon = \wedge_{\Upsilon_i \in \{\Upsilon_1, \cdots, \Upsilon_\ell\}} (\vee_{x \in \Upsilon_i} x)$;
    - Find Υ_i such that $\Upsilon_i \in \{\Upsilon_1, \cdots, \Upsilon_\ell\} \wedge \Upsilon_i \cap W_i = \varnothing$;
    - $\pi \leftarrow \text{ProveDisjoint}(W_i, \Upsilon_i, pk)$;
    - Send $\langle \pi, \Upsilon_i \rangle$ to the user;

Result Verification (by the user)
- **if** o_i *matches* q **then**
  - Check o_i w.r.t. $\text{hash}(o_i)$ from the block header;
  - Check whether o_i matches q;
- **else**
  - Read $AttDigest_i$ from the block header;
  - Run VerifyDisjoint($AttDigest_i$, acc(Υ_i), π, pk);

KeyGen$(1^\lambda) \rightarrow (sk, pk)$: Let $(p, \mathbb{G}, \mathbb{H}, e, g)$ be a bilinear pairing. Choose a random value $s \leftarrow \mathbb{Z}_p$. The secret key is $sk = (s)$ and the public key is $pk = (g, g^s, g^{s^2}, \cdots, g^{s^q})$.

Setup$(X, pk) \rightarrow$ acc(X): The accumulative value for a multiset $X = \{x_1, \cdots, x_n\}$ is $\text{acc}(X) = g^{P(X)} = g^{\prod_{x_i \in X}(x_i + s)}$. Owing to the property of the polynomial interpolation, it can be computed without knowing the secret key.

ProveDisjoint$(X_1, X_2, pk) \rightarrow \pi$: According to the extended Euclidean algorithm, if $X_1 \cap X_2 = \varnothing$, there exist two polynomials Q_1, Q_2 such that $P(X_1)Q_1 + P(X_2)Q_2 = 1$. As such, if $X_1 \cap X_2 = \varnothing$, the proof can be computed as $\pi = (F_1^*, F_2^*) = (g^{Q_1}, g^{Q_2})$.

VerifyDisjoint$(\text{acc}(X_1), \text{acc}(X_2), \pi, pk) \rightarrow \{0, 1\}$: To verify the proof, the verifier interprets π as (F_1^*, F_2^*). The proof is valid if and only if the following constraint holds:

$$e(\text{acc}(X_1), F_1^*) \cdot e(\text{acc}(X_2), F_2^*) \stackrel{?}{=} e(g, g).$$

5.2.2 Construction 2. Inspired by [35], the second construction is proposed to introduce two additional Sum(·) and ProofSum(·) primitives, which allow the aggregation of multiple accumulative values or set disjoint proofs. It is based on bilinear pairing and q-DHE assumption and consists of the following algorithms.

KeyGen$(1^\lambda) \rightarrow (sk, pk)$: Let $(p, \mathbb{G}, \mathbb{H}, e, g)$ be a bilinear pairing. Choose a random value $s \leftarrow \mathbb{Z}_p$. The secret key is $sk = (s)$ and the public key is $pk = (g, g^s, g^{s^2}, \cdots, g^{s^{q-1}}, g^{s^{q+1}}, \cdots, g^{s^{2q-2}})$.

Setup$(X, pk) \rightarrow \text{acc}(X)$: The accumulative value for a multiset $X = \{x_1, \cdots, x_n\}$ is $\text{acc}(X) = (d_A(X), d_B(X))$, where $d_A(X) = g^{A(X)} = g^{\sum_{x_i \in X} s^{x_i}}$ and $d_B(X) = g^{B(X)} = g^{\sum_{x_i \in X} s^{q-x_i}}$. Similar to the first construction, it can also be computed without knowing the secret key using the polynomial interpolation.

ProveDisjoint$(X_1, X_2, pk) \rightarrow \pi$: It is easy to see that if $X_1 \cap X_2 = \varnothing$, then $Cs^q \notin A(X_1)B(X_2)$, where C is a non-zero constant. As such, if $X_1 \cap X_2 = \varnothing$, the proof can be computed as $\pi = g^{A(X_1)B(X_2)} = g^{(\sum_{x_i \in X_1} s^{x_i})(\sum_{x_j \in X_2} s^{q-x_j})}$.

VerifyDisjoint$(\text{acc}(X_1), \text{acc}(X_2), \pi, pk) \rightarrow \{0, 1\}$: To verify the proof, the verifier interprets $\text{acc}(X_1)$ and $\text{acc}(X_2)$ as $(d_A(X_1), d_B(X_1))$ and $(d_A(X_2), d_B(X_2))$, respectively. The proof is valid if and only if the following constraint holds:

$$e(d_A(X_1), d_B(X_2)) \stackrel{?}{=} e(\pi, g).$$

Sum$(\text{acc}(X_1), \text{acc}(X_2), \cdots, \text{acc}(X_n)) \rightarrow \text{acc}(\sum_i^n X_i)$: On input of multiple accumulative values $\text{acc}(X_1), \cdots, \text{acc}(X_n)$, it outputs the accumulative value for the multiset $\sum_i^n X_i$. $\text{acc}(\sum_i^n X_i) = (d_A(\sum_i^n X_i), d_B(\sum_i^n X_i))$, where $d_A(\sum_i^n X_i) = \prod_i^n d_A(X_i)$ and $d_B(\sum_i^n X_i) = \prod_i^n d_B(X_i)$.

ProofSum$(\langle \pi_1, X_{\pi_1,1}, X_{\pi_1,2} \rangle, \cdots, \langle \pi_n, X_{\pi_n,1}, X_{\pi_n,2} \rangle) \rightarrow \pi'$): On input of multiple set disjoint proofs $\pi_1 = \text{ProveDisjoint}(X_{\pi_1,1}, X_{\pi_1,2}, pk)$, $\cdots$, $\pi_n = \text{ProveDisjoint}(X_{\pi_n,1}, X_{\pi_n,2}, pk)$, it outputs an aggregate proof $\pi' = \sum_{i=1}^n \pi_i$, if and only if $X_{\pi_1,2} = X_{\pi_2,2} = \cdots = X_{\pi_n,2}$.

Compared with Construction 1, Construction 2 supports the aggregation of multiple accumulative values or set disjoint proofs, which can be used by the online batch verification method in Section 6.3. However, it incurs a much larger key size. In particular, the public key size in Construction 1 is linear to the largest multiset size, whereas in Construction 2, the public key size is linear to the largest possible value of the attributes in the system. In real-life applications, the common practice is to use a cryptographic hash function to encode each attribute value into an integer number, which is then accepted by the accumulator. Since the value returned by a typical hash function is in several hundreds of bits, it is costly to generate and publish the public key with such a scale in advance. To remedy this issue, we may introduce a trusted oracle, which owns the secret key and is responsible to answer requests of the public key. Such an oracle can be acted by a trusted third party or be implemented utilizing secure hardware like SGX.

5.3 Extension to Range Queries

The previous sections mainly consider the Boolean queries on the set-valued attribute W_i. In many scenarios, the user may also apply range conditions on the numerical attributes V_i. To tackle this problem, we propose a method that transforms numerical attributes into set-valued attributes. Then, a range query can be mapped to a Boolean query accordingly.

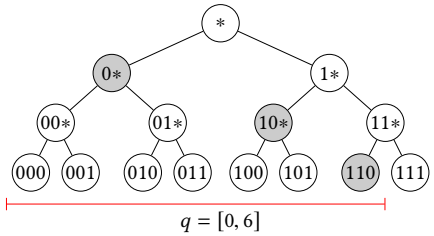


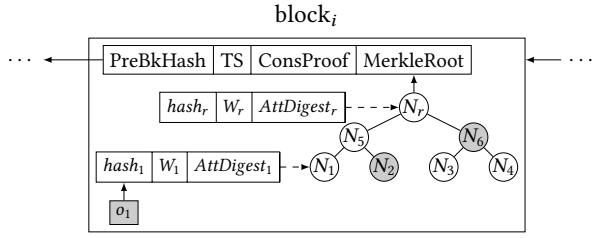
Figure 5: Example of Transformation

The idea goes as follows. First, we represent each numerical value in the binary format. Next, we transform a numerical value into a set of binary prefix elements (denoted as function trans$(\cdot)$). For example, a value 4 can be represented in the binary format 100. Thus, it can be transformed into a prefix set, i.e., trans$(4) = \{1*, 10*, 100\}$, where $*$ denotes the wildcard matching operator. Similarly for a numerical vector, we can apply the above procedure for each dimension. For example, a vector $(4, 2)$ has the binary format $(100, 010)$. Thus, its transformed prefix set is $\{1*_1, 10*_1, 100_1, 0*_2, 01*_2, 010_2\}$. Note that here each element has a subscript notation (i.e., $_1$ and $_2$), which is used to distinguish the binary values in the different dimensions of the vector.

Next, we transform a range query condition into a monotone Boolean function, by using a binary tree built over the entire binary space (e.g., Fig. 5 shows a tree of single-dimension space [0,7]). Specifically, for a single-dimension range $[\alpha, \beta]$, we first represent α and β in its binary format. Next, we view α and β as two leaf nodes in the tree. Finally, we find the minimum set of tree nodes to exactly cover the whole range $[\alpha, \beta]$. The transformed Boolean function is a function concatenating each element in the set using OR $(\vee)$ semantic. For example, for a query range $[0, 6]$, we can find its transformed Boolean function as $0* \vee 10* \vee 110$ (see the gray nodes in Fig. 5). As discussed in Section 5.1, the equivalence set of this Boolean function is $\{0*, 10*, 110\}$. Similarly, in the case of a multi-dimensional range, the transformed Boolean function is the one concatenating the partial Boolean function for each dimension using AND $(\wedge)$ semantic. For example, a query range $[(0, 3), (6, 4)]$ can be transformed to $(0*_1 \vee 10*_1 \vee 110_1) \wedge (011_2 \vee 100_2)$, with equivalence sets of $\{0*_1, 10*_1, 110_1\}$ and $\{011_2, 100_2\}$.

With the above transformations, a query of whether or not a numerical value v_i is in a range $[\alpha, \beta]$ becomes a Boolean query of v_i's transformed prefix set against $[\alpha, \beta]$'s equivalence sets. In the above examples, $4 \in [0, 6]$ since $\{1*, 10*, 100\} \cap \{0*, 10*, 110\} = \{10*\} \neq \varnothing$; $(4, 2) \notin [(0, 3), (6, 4)]$ since there exists some equivalence set $\{011_2, 100_2\}$ such that $\{011_2, 100_2\} \cap \{1*_1, 10*_1, 100_1, 0*_2, 01*_2, 010_2\} = \varnothing$.

Thanks to the data transformation technique, in the sequel we unify the two types of query conditions into a uniform Boolean query condition on the set-valued attribute. More specifically, for each data object $\langle t_i, V_i, W_i \rangle$, it is transformed



Node	Object	Set Attributes
N_1	o_1	W_1 = {"Sedan", "Benz"}
N_2	o_2	W_2 = {"Sedan", "Audi"}
N_3	o_3	W_3 = {"Van", "Benz"}
N_4	o_4	W_4 = {"Van", "BMW"}

Figure 6: Intra-Block Index

into a tuple $\langle t_i, W_i' \rangle$, where $W_i' = \text{trans}(V_i) + W_i$; and a query $q = \langle [t_s, t_e], [\alpha, \beta], \Upsilon \rangle$ is transformed into $\langle [t_s, t_e], \Upsilon' \rangle$, where $\Upsilon' = \text{trans}([\alpha, \beta]) \wedge \Upsilon$. As such, the query results are $\{o_i = \langle t_i, W_i' \rangle \mid t_i \in [t_s, t_e] \wedge \Upsilon'(W_i') = 1\}$.

6 BATCH VERIFICATION

In this section, we discuss how to boost the query performance via batch verification. We first introduce two authenticated indexing structures, namely intra-block index (Section 6.1) and inter-block index (Section 6.2), followed by an online batch verification method (Section 6.3). All these techniques allow the SP to prove mismatching objects in a batch.

6.1 Intra-Block Index

In the previous discussion, we assume that each block stores only one object for simplicity. In general, each block often stores multiple objects. Naively, we can apply the single-object algorithm repeatedly for each object to ensure query integrity, which however incurs a verification complexity linear to the number of objects. Further, it can be observed that if two objects share some common attribute value, they may mismatch some queries due to the same partial query condition. Therefore, to reduce the proofing and verification overhead, we propose an intra-block index which can aggregate multiple objects and improve performance.

Fig. 6 shows a block of the blockchain with the intra-block index. It organizes the *ObjectHash* and *AttDigest* of each object into a binary Merkle tree. The block header consists of the following components: *PreBkHash*, *TS*, *ConsProof*, and *MerkleRoot*, where MerkleRoot is the root hash of the binary Merkle tree. Each tree node has three fields: child hash (denoted by $hash_i$, and is used to form the MHT), attribute multiset (denoted by W_i), and attribute multiset's accumulative value (denoted by $AttDigest_i$). They are computed from the child nodes as follows.

Definition 6.1 (Intra-Block Index Non-Leaf Node). Let $\text{hash}(\cdot)$ be a cryptographic hash function, '|' be the string concatenation operator, $\text{acc}(\cdot)$ be the multiset accumulator, n_l and n_r be the left and right children of node n, respectively. The fields for a non-leaf node n are defined as:

- $W_n = W_{n_l} \cup W_{n_r}$
- $AttDigest_n = \text{acc}(W_n)$
- $hash_n = \text{hash}(\text{hash}(hash_{n_l} \mid hash_{n_r}) \mid AttDigest_n)$

Definition 6.2 (Intra-Block Index Leaf Node). The fields for a leaf node are identical to those for the underlying object.

When building the intra-block index, we want to achieve the maximum proofing efficiency. That is, we aim to maximize the chance of pruning the mismatching objects together during query processing. On the one hand, this means that we should find a clustering strategy such that given a user's query, the chance that a node mismatches the query is maximum. In another words, we strive to maximize the similarity of the objects under each node. On the other hand, a balanced tree is preferred since it can improve the query efficiency. Thus, we propose that the intra-block index is built based on the block's data objects in a bottom-up fashion (by the blockchain miners). First, each data object in the block is assigned to a leaf node. Next, the leaf nodes which yield the maximum Jaccard similarity $\frac{|W_{n_l} \cap W_{n_r}|}{|W_{n_l} \cup W_{n_r}|}$ are iteratively merged. The two merged tree nodes are used to create a new non-leaf node in the upper level. This process is repeated in each level until the root node is created. Finally, the *MerkleRoot* assigned by $hash_r$ is written as one of the components of the block header. Algorithm 2 shows the procedure in detail.

With the above intra-block index, the SP can process a query as a tree search. Starting from the root node, if the attribute multiset of the current node fulfills the query condition, its subtree will be further explored. Also, the corresponding *AttDigest* is added to the VO, which will be used to

Algorithm 2: Intra-Index Construction (by the miners)

```
Function BuildIntraIndex(nodes)
    Input: list of leaf nodes nodes
    while nodes.len > 1 do
        newnodes ← [];
        while nodes.len > 1 do
            n_l ← arg max_n |W_n|; nodes.delete(n_l);
            n_r ← arg max_n |W_{n_l} ∩ W_n| / |W_{n_l} ∪ W_n|; nodes.delete(n_r);
            W_n ← W_{n_l} ∪ W_{n_r};
            AttDigest_n ← acc(node.W);
            h_n ← hash(hash(hash_{n_l} | hash_{n_r}) | AttDigest_n);
            newnodes.add(⟨n_l, n_r, h_n, W_n, AttDigest_n⟩);
        nodes ← newnodes + nodes;
    root ← nodes[0];
    MerkleRoot ← hash_r;
```

Algorithm 3: Query w. Intra-Index (by the SP)

```
Function IntraIndexQuery(root, q)
    Input: Intra-Index root root, Query condition q = ⟨ϒ⟩
    Output: Query Result R, Verification Object VO
    Create an empty queue queue;
    queue.enqueue(root);
    while queue is not empty do
        n ← queue.dequeue();
        if W_n matches q then
            if n is a leaf node then Add o_n to R;
            else
                Add ⟨AttDigest_n⟩ to VO;
                queue.enqueue(n.children);
        else
            Find query condition set ϒ_i w.r.t. W_n (see Alg. 1);
            π ← ProveDisjoint(W_n, ϒ_i, pk);
            add ⟨hash_n, π, ϒ_i, AttDigest_n⟩ to VO;
    return ⟨R, VO⟩;
```

reconstruct the *MerkleRoot* during result verification. On the other hand, if the multiset does not satisfy the query condition, it means that all the underlying objects are mismatches. In this case, the SP will invoke ProveDisjoint$(\cdot)$ with the corresponding *AttDigest* to generate a mismatch proof. Upon reaching a leaf node, the object whose multiset satisfies the query condition is a matching object and will be returned as a query result. Algorithm 3 shows the VO construction using the intra-block index.

For illustration, we use the same set of objects as discussed in Section 5.1. The intra-block index is shown in Fig. 6. The Boolean query from the user is "Sedan" $\wedge$ ("Benz" $\vee$ "BMW"). The query process simply traverses the index from the root node to the leaf nodes. The query result is $\{o_1\}$. The VO returned by the SP includes $\{\langle AttDigest_r\rangle, \langle AttDigest_5\rangle, \langle hash_2, \pi_2, \{\text{"Audi"}\}, AttDigest_2\rangle, \langle hash_6, \pi_6, \{\text{"Van"}\}, AttDigest_6\rangle\}$. Here π_2 and π_6 are two disjoint proofs of the mismatching nodes N_2 and N_6 (shaded in Fig. 6), respectively. Note that $AttDigest_r$ and $AttDigest_5$ will only be used to reconstruct the *MerkleRoot* during result verification. On the user side, the mismatch verification works by invoking VerifyDisjoint$(\cdot)$ using the *AttDigest*, the disjoint set, and the proof π in the VO. Further, in order to verify the result soundness and completeness, the user is required to reconstruct the *MerkleRoot* and compare it with the one read from the block header. In our example, firstly, VerifyDisjoint$(\cdot)$ is invoked using $\langle \pi_2, AttDigest_2, \{\text{"Audi"}\}\rangle$ and $\langle \pi_6, AttDigest_6, \{\text{"Van"}\}\rangle$ to prove that nodes N_2 and N_6 indeed mismatch the query. After that, the user computes hash(o_1) using the returned result, and $hash_5 = \text{hash}(\text{hash}(o_1) \mid hash_2 \mid AttDigest_5)$, $hash_r = \text{hash}(hash_5 \mid hash_6 \mid AttDigest_r)$ based on the VO. Finally, the user checks the newly computed $hash_r$ against the *MerkleRoot* in the block header.

Algorithm 4: Query w. Inter-Index (by the SP)

```
Function InterIndexQuery(block, q)
    Input: Current Block block, Query condition q = ⟨ϒ⟩
    FindJump ← False;
    for L_i from L_max to L_min of block.SkipList do
        if W_{L_i} does not match q and FindJump ≠ True then
            Find query condition set ϒ_i w.r.t. W_n;
            π ← ProveDisjoin(W_{L_i}, ϒ_i, pk);
            add ⟨PreSkippedHash_{L_i}, π, ϒ_i, AttDigest_{L_i}⟩ and
              all hash_{L_j} ∈ SkipList, j ≠ i to VO;
            block ← Skip(i); FindJump ← True;
    if FindJump = False then
        IntraIndexQuery(block.root, q);
        block ← block.prev;
    InterIndexQuery(block, q);
```

6.2 Inter-Block Index

Besides similar objects within the same block, the objects across blocks may also share similarity and mismatch a query due to the same reason. Based on this observation, we build an inter-block index that uses a skip list to further optimize the query performance.

As shown in Fig. 7, the inter-block index consists of multiple skips, each of which skips an exponentially number of previous blocks. For example, the list may skip previous $2, 4, 8, \cdots$ blocks. For each skip, it maintains three components: the hash of all skipped blocks (denoted by $PreSkippedHash_{L_k}$), the sum of the attribute multisets for the skipped blocks (denoted by W_{L_k}), and the corresponding accumulative value w.r.t. W_{L_k} (denoted by $AttDigest_{L_k}$). Note that here we use the summation of attribute multisets to enable online aggregate authentication in Section 7. Finally, the inter-block index is written into the block using an extra field *SkipListRoot*, which is defined as:

- $SkipListRoot = \text{hash}(hash_{L_2} \mid hash_{L_4} \mid hash_{L_8} \mid \cdots)$
- $hash_{L_k} = \text{hash}(PreSkippedHash_{L_k} \mid AttDigest_{L_k})$
- $AttDigest_{L_k} = acc(W_{L_k})$
- $W_{L_k} = \sum_{j=i-k+1}^{i} W_j$

During the query processing, an eligible skip may be used to represent multiple blocks which do not contribute to query results due to the same reason of mismatching. As the user can avoid accessing these skipped blocks, the verification cost can be reduced.

Algorithm 4 shows the query processing procedure with the inter-block index. We start with the latest block in the query time window. We iterate the skip list from the maximum skip to the minimum skip. If the multiset of a skip W_{L_i} does not match the query condition, it means that all the skipped blocks between the current block and the previous i-th block do not contain matching results. Therefore, ProveDisjoint$(\cdot)$ is invoked and output the mismatch proof

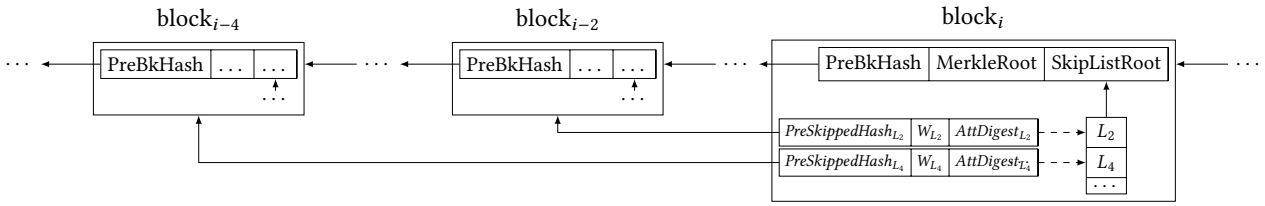


Figure 7: Inter-Block Index

π_i. Then, $\langle PreSkippedHash_{L_i}, \pi_i, \Upsilon_i, AttDigest_{L_i} \rangle$ are added to the VO. The user can use this proof to verify that the skipped blocks indeed mismatch the query. Meanwhile, other hashes except $hash_{L_i}$ are also added to the VO. If we fail to find mismatching blocks during the iteration, the function `IntraIndexQuery(·)` (Algorithm 3) is invoked for the current block and then the previous block is examined next. If we successfully find a mismatch skip, the corresponding preceding block will be examined next. The function `InterIndexQuery(·)` is invoked recursively until we complete checking all the blocks within the query window. Note that we can combine the intra-block index and inter-block index to maximize performance since they are not in conflict.

6.3 Online Batch Verification

Recall that the proposed intra-block index attempts to cluster the objects of the same block in a way to maximize the proofing efficiency of mismatching objects. Nevertheless, some objects/nodes indexed in different blocks or even different subtrees of the same block may also share the same reason of mismatching. Therefore, it would be beneficial to aggregate such objects/nodes online for more efficient proofing. To do so, the Sum(·) primitive introduced by Construction 2 in Section 5.2, which outputs the accumulative value of the aggregated multiset when given multiple accumulative values, can be applied. In our running example shown in Fig. 6, suppose that o_2 and o_4 share the same reason for mismatching a query condition ("Benz"). Then, the SP can return $\pi = \text{ProveDisjoint}(W_2 + W_4, \{\text{"Benz"}\}, pk)$ and $AttDigest_{2,4} = \text{Sum}(\text{acc}(W_2), \text{acc}(W_4))$. And the user can apply $\text{VerifyDisjoint}(AttDigest_{2,4}, \text{acc}(\{\text{"Benz"}\}), \pi, pk)$ to prove that these two objects mismatch in a batch.

7 VERIFIABLE SUBSCRIPTION QUERIES

A subscription query is registered by the query user and continuously processed until it is deregistered. Upon seeing a newly confirmed block, the SP will need to publish the results to registered users, together with VOs. In this section, we first propose a query index to efficiently handle a large number of subscription queries (Section 7.1). After that, we develop a lazy authentication optimization that delays mismatch proofs to reduce the query verification costs (Section 7.2).

7.1 Query Index for Scalable Processing

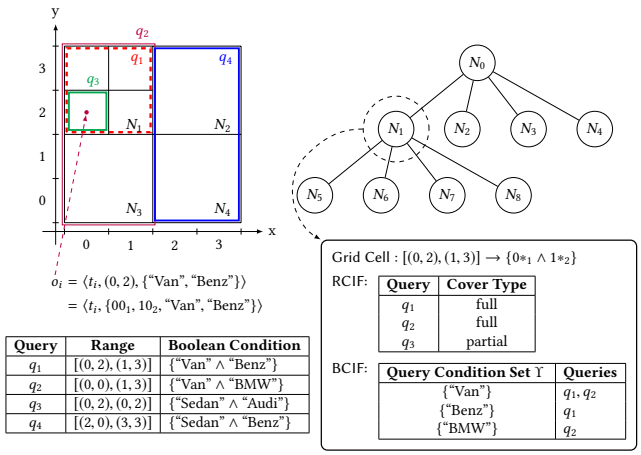
As discussed earlier, the majority of the query processing overhead comes from generating the proofs for mismatching objects at the SP. Fortunately, a mismatching object could have the same reason of mismatching for different subscription queries. Thus, a mismatch proof can be shared by such queries. Inspired by [41], we propose to build an inverted prefix tree, called *IP-Tree*, over subscription queries. It is essentially a prefix tree with reference to inverted files for both the numerical range condition and also the Boolean set condition.

Prefix Tree Component. To index the numerical ranges of all subscription queries, the IP-Tree is built on the basis of a grid tree such that each tree node is represented by a CNF Boolean function (see Section 5.3). For example, the grid node N_1 in Fig. 8, corresponding to the upper-left cell ([0, 2], [1, 3]), is denoted by $\{0*_1 \wedge 1*_2\}$. The root node of the prefix tree covers the entire range space of all subscription queries.

Inverted File Component. Each node of the IP-Tree is associated with an inverted file that is constructed based on the subscription queries indexed under the node. There are two subcomponents for each inverted file:

- **Range Condition Inverted File (RCIF)**. Each entry in the *RCIF* has two attributes: query q_i and its cover type (i.e., full or partial). All the queries in the *RCIF* intersect the numerical space $\mathcal{S}$ of the node. The cover type indicates whether q_i fully covers or partially covers $\mathcal{S}$. The *RCIF* is used to check the mismatch of the numerical range condition.
- **Boolean Condition Inverted File (BCIF)**. The *BCIF* records only the queries that fully cover the node's space. Each entry in the *BCIF* consists of two attributes: query condition set Υ and corresponding queries. The *BCIF* is used to check the mismatch of the Boolean set condition.

We use Fig. 8 as an example to illustrate how to construct the IP-Tree. It is built in a top-down fashion by the SP. We first create the root node and add all queries to its *RCIF* as partial-cover queries. We then split the root node and create four equally-spaced child nodes. For each child node, if a query fully or partially covers the node's space, it will be added to the node's *RCIF*. Also, the equivalence sets of a full-cover query will be added to the node's *BCIF*. Take N_1 as an example. While queries q_1 and q_2 fully cover this node,



Query	Range	Boolean Condition
q_1	[(0, 2), (1, 3)]	{“Van” ∧ “Benz”}
q_2	[(0, 0), (1, 3)]	{“Van” ∧ “BMW”}
q_3	[(0, 2), (0, 2)]	{“Sedan” ∧ “Audi”}
q_4	[(2, 0), (3, 3)]	{“Sedan” ∧ “Benz”}

Query	Cover Type
q_1	full
q_2	full
q_3	partial

Query Condition Set ϒ	Queries
{“Van”}	q_1, q_2
{“Benz”}	q_1
{“BMW”}	q_2

Figure 8: Inverted Prefix Tree

query q_3 only partially covers it. Thus, the *RCIF* contains three intersection queries q_1, q_2, and q_3. The cover types of q_1 and q_2 are *full* and for q_3 the type is *partial*. As for the *BCIF* of N_1, q_1 and q_2 share the equivalence set {“Van”}, and the sets {“Benz”} and {“BMW”} correspond to queries q_1 and q_2, respectively. Next, since q_3 only partially covers N_1, we further split N_1 into four sub-cells. As q_3 fully covers N_7, it is added to the *RCIF* and *BCIF* of N_7. The algorithm terminates when no partial query is found in any leaf node. When a query is registered or deregistered, we update the IP-Tree's nodes corresponding to the numerical range of the query. We may also split or merge the tree nodes if necessary. Note that, to prevent the tree from becoming too deep, we switch back to the case without the IP-Tree when the tree depth reaches some pre-defined threshold.

With the IP-Tree index, the subscription queries can be processed as a tree traversal. We first use an example of single object to illustrate the basic idea. Upon arrival of a new object o, the IP-Tree is traversed along the path from the root to the leaf node that covers o. For any node n_q on the traversal path, the associated queries can be found from n_q's *RCIF*. These queries can be classified into three categories: (1) a full-cover query whose equivalence set(s) in n_q's *BCIF* match o (thus o is added as a result of this query); (2) a full-cover query whose equivalence set(s) in n_q's *BCIF* mismatch o (thus ProveDisjoint(·) is invoked and a disjoint proof is generated for this query); (3) a partial-cover query (no further action is needed). In addition, we identify the queries that appear in n_q's parent's *RCIF* but not in n_q's. Those queries mismatch the numerical range condition for o and thus also a disjoint proof is generated for them. Next, n_q's child node will be processed and this process continues until we reach a leaf node or all queries have been classified as matching or mismatching. Consider a new object $o_i = \langle t_i, (0,2), \{\text{“Van”}, \text{“Benz”}\}\rangle = \langle t_i, \{00_1, 10_2, \text{“Van”}, \text{“Benz”}\}\rangle$ shown in Fig. 8. At N_1, q_1 is classified as a matching query,

Algorithm 5: Subscription Query w. Lazy Authentication (by the SP)

Function `SubscribeInterIndexQuery`($block, q, VO, s$)
- **Input:** Block $block$, Query condition $q = \langle \Upsilon \rangle$, Partial VO, Stack s
- $W_r \leftarrow block.root.W_r$;
- **if** W_r *matches* q **then**
  - $\langle R, \text{VO}_b \rangle \leftarrow$ `IntraIndexQuery`($block.root, q$);
  - $\text{VO} \leftarrow \text{VO} + \text{VO}_b$; Send $\langle R, \text{VO} \rangle$ to user;
  - Empty partial VO and s;
- **else**
  - $\text{VO}_b \leftarrow$ `IntraIndexQuery`($block.root, q$);
  - **if** W_r *has the same mismatch attributes with* s **then**
    - Find the maxiumn skip L_i s.t. it covers m elements on top of s;
    - **if** L_i *is found* **then**
      - $\langle block_i, \text{JumpDistance}_i \rangle \leftarrow s[i], \forall i \in \{m\}$;
      - Pop m elements of s;
      - Rewind partial VO to $block_m$;
      - Add VO_b, $AttDigest_{L_i}$ and other hashes to VO;
      - $s.push(\langle block, \sum_i^m \text{JumpDistance}_i \rangle)$;
    - **else** $s.push(\langle block, 1 \rangle)$;
  - **else**
    - Empty s; $s.push(\langle block, 1 \rangle)$;

q_2 and q_4 mismatch because of the Boolean set condition and the numerical range condition, respectively, whereas q_3 is not confirmed as mismatching until we check N_1's child node N_7.

This idea can be easily extended to a new block of objects that are indexed by an intra-block index. We start from the root of the intra-block index. For any index node n_b, we treat it as a *super object* and apply the above query processing procedure. The only difference is that if a full-cover query is classified as matching, we cannot immediately return the current node n_b as a query result, but to further recursively check its child nodes until reaching the leaf nodes. In the interest of space, the pseudo codes of the detailed algorithms are given in Appendix A.

7.2 Lazy Authentication

Observing that in the previous section, the results and proofs are immediately published to registered users while a new block is confirmed. In particular, even if there is no matching result for a query, the mismatch proofs are still computed and sent. This approach is good for real-time applications. For applications that do not have such real-time requirements, we propose a *lazy authentication* optimization, in which the SP returns the result only when there is a matching object (or the time since the last result has passed a threshold).

In this approach, the VO should prove that the current

object is a match and all other objects since the last result mismatch the query. To achieve this, we may simply wait for the matching result and invoke a time-window query to compute the mismatch proofs on the fly. However, this method can only generate the mismatch proofs for each query separately and is incapable to take advantage of the proofs shared by different subscription queries. Moreover, this method leaves the burden of proofing all to the time when there is a matching result. To address these issues, we propose a new method that makes use of the inter-block index to incrementally generating mismatch proofs.

Using the inter-block index to answer a subscription query is completely different from doing that to a time-window query. The reason is that we can back traverse the blockchain and use the skip list to aggregate proofs in a time-window query. However, we cannot do so for a subscription query because new blocks are not yet available and we do not know whether or not future objects will share the same mismatch conditions. As such, we introduce a *stack* to facilitate tracking the arrived blocks that share the same mismatch conditions. The basic idea is to use the skip list to find the maximum skip distance L_i such that it covers m elements on top of the stack. The *AttDigest*s of these blocks are replaced with $AttDigest_{L_i}$. Thanks to Construction 2 in Section 5.2, the disjoint proofs can be aggregated online by invoking ProofSum(·). For example, we have two mismatching $block_i$ and $block_{i-1}$ in the stack and there is a skip with distance 2. Then, the SP can replace their proofs by an aggregate proof computed from ProofSum(π_i, π_{i-1}). In this way, the SP does not need to compute the set disjoint proofs from scratch when a matching result is found. The detailed procedure is described in Algorithm 5.

8 SECURITY ANALYSIS

This section performs a security analysis on the multiset accumulators and query authentication algorithms.

8.1 Analysis on Multiset Accumulators

We first present a formal definition of the security notion for multiset accumulators and set disjoint proofs.

Definition 8.1 (Unforgeability [32]). We say a multiset accumulator is unforgeable if the success probability of any polynomial-time adversary is negligible in the following experiment:

- Run $(sk, pk) \leftarrow \mathsf{KeyGen}(1^\lambda)$ and give the public key pk to the adversary;
- The adversary outputs two multisets X_1 and X_2, along with a set disjoint proof π.

We say the adversary succeeds if VerifyDisjoint(acc(X_1), acc(X_2), π, pk) outputs 1 and $X_1 \cap X_2 \neq \varnothing$.

This property ensures that the chance for a malicious SP to forge a set disjoint proof is negligible, which serves as a foundation for the security of our proposed query authentication algorithms. We now show that our constructions of the accumulator indeed satisfy the desired security requirement.

Theorem 8.1. *The constructions of the multiset accumulator presented in Section 5.2 satisfy the security property of the unforgeability as defined in Definition 8.1.*

Proof. See Appendix B for a detailed proof. □

8.2 Analysis on Query Authentication

The formal definition of the unforgeability for our query authentication algorithms is given below:

Definition 8.2 (Unforgeability). We say our proposed query authentication algorithms are unforgeable if the success probability of any polynomial-time adversary is negligible in the following experiment:

- Run the ADS generation and give all objects $\{o_i\}$ to the adversary;
- The adversary outputs a query q (either time-window or subscription query), a result R, and a VO;

We say the adversary succeeds if the VO passes the result verification and one of the following results is true:

- R contains an object o^* such that $o^* \notin \{o_i\}$;
- R contains an object o^* such that o^* does not satisfy the query q;
- There exists an object o_x in the query time window or subscription period, which is not in R but satisfies q.

This property ensures that the chance for a malicious SP to forge an incorrect or incomplete result is negligible. We can show that our proposed query authentication algorithms indeed satisfy the desired security requirement.

Theorem 8.2. *Our proposed query authentication algorithms satisfy the security property of the unforgeability as defined in Definition 8.2.*

Proof. See Appendix C for a detailed proof. □

9 PERFORMANCE EVALUATION

In this section, we evaluate the performance of the vChain framework for time-window queries and subscription queries. Three datasets are used in the experiments:

- **Foursquare** (4SQ) [46]: The 4SQ dataset contains 1M data records, which are the user check-in information. We pack the records within a 30s interval as a block and each object has the form of $\langle timestamp, [longitude, latitude], \{check\text{-}in\ place's\ keywords\}\rangle$. On average, each record has 2 keywords.
- **Weather** (WX): The WX dataset contains 1.5M hourly weather records for 36 cities in US, Canada, and Israeli during 2012-2017.[2] For each record, it contains seven

[2] https://www.kaggle.com/selfishgene/historical-hourly-weather-data/

numerical attributes (such as *humidity* and *temperature*) and one *weather description* attribute with 2 keywords on average. The records within the same hour interval are packed as a block.

- **Ethereum** (ETH): The ETH transaction dataset is extracted from the Ethereum blockchain during the period from Jan 15, 2017 to Jan 30, 2017.[3] It contains 90,000 blocks with 1.12M transaction records. Each transaction is in the form of $\langle timestamp, amount, \{addresses\}\rangle$, where *amount* is the amount of Ether transferred and $\{addresses\}$ are the addresses of senders and receivers. Most transactions have two addresses.

Note that the time intervals of the blocks in 4SQ, WX, and ETH are roughly 30s, 1 hour, and 15s, respectively.

The query user is set up on a commodity laptop computer with Intel Core i5 CPU and 8GB RAM, running on CentOS 7 with a single thread. The SP and the miner are set up on a x64 blade server with dual Intel Xeon 2.67GHz, X5650 CPU and 32 GB RAM, running on CentOS 7. The experiments are written in C++ and the following libraries are used: MCL for bilinear pairing computation,[4] Flint for modular arithmetic operations, Crypto++ for 160-bit SHA-1 hash operations, and OpenMP for parallel computation. Also, the SP runs with 24 hyperthreads to accelerate the query processing.

To evaluate the performance of verifiable queries in vChain, we mainly use three metrics: (i) query processing cost in terms of SP CPU time, (ii) result verification cost in terms of user CPU time, and (iii) size of the VO transmitted from the SP to the user. For each experiment, we randomly generate 20 queries and report the average results. By default, we set the selectivity of the numerical range to 10% (for 4SQ and WX) and 50% (for ETH) and employ a disjunctive Boolean function with a size of 3 (for 4SQ and WX) and 9 (for ETH). For WX, two attributes are involved in each range predicate.

9.1 Setup Cost

Table 1 reports the miner's setup cost, including the ADS construction time and the ADS size. Three methods are compared in our experiments: (i) *nil*: no index is used; (ii) *intra*: only intra-block index is used; (iii) *both*: both intra- and inter-block indexes are used, in which the size of *SkipList* in the inter-block index is set to 5. Each method is implemented with two different accumulator constructions (labelled with *acc1* and *acc2*) presented in Section 5.2. Thus, a total of six schemes are evaluated in each experiment. As expected, the ADS construction time of *both* is generally longer than those of *nil* and *intra*, but still within 2s for most cases. Moreover, compared with *acc1*, *acc2* significantly reduces the construction time of *both* because it supports online aggregation and hence can reuse the index of the previous block in constructing the inter-block index. Regarding the ADS size, it is independent of the accumulator used and ranges from 2.6KB to 11.1KB per block for different indexes and datasets.

Table 1: Miner's Setup Cost

Dataset	Acc	nil		intra		both	
		T	S	T	S	T	S
4SQ	acc1	0.17	2.12	0.65	10.7	12.5	11.1
4SQ	acc2	0.06	2.12	0.26	10.7	1.16	11.1
WX	acc1	0.16	1.55	0.52	7.38	1.01	7.68
WX	acc2	0.05	1.55	0.16	7.38	0.20	7.68
ETH	acc1	0.01	0.55	0.07	2.60	0.87	2.93
ETH	acc2	0.14	0.55	0.30	2.60	0.13	2.93

T: ADS construction time (s/block); S: ADS size (KB/block)

We also measure the space required by the user for running a light node to maintain the block headers. For both *nil* and *intra*, the size of each block header is 800 bits, regardless of the dataset or accumulator. Due to the inter-block index, the block header size of *both* is slightly increased to 960 bits.

9.2 Time-Window Query Performance

To evaluate the performance for time-window queries, we vary the query window from 2 to 10 hours for 4SQ and ETH and from 20 to 100 hours for WX. The results for the three datasets are shown in Figs. 9–11, respectively. We make several interesting observations. First, as expected, the indexes substantially improve the performance in almost all metrics. In particular, for the 4SQ and ETH datasets, the performance of using the indexes is at least 2X better than that with the same accumulator but without using any index. This is because the objects in these two datasets share less similarity and hence benefit more from using the indexes for pruning. Second, the costs of the index-based schemes increase only sublinearly with enlarging the query window. This is particularly true in terms of the user CPU time for the index-based schemes using *acc*2, which supports batch verification of mismatches (see Section 6.3). Third, comparing *intra* and *both*, *both* is always no worse than *intra* except concerning the SP CPU time for the 4SQ dataset. On the one hand, this indicates the effectiveness of using the inter-block index. On the other hand, the reason of *both* being worse than *intra* in SP CPU time is mainly because in an inter-block index-based scheme, larger multisets are used as the input of a set disjoint proof, which increases the SP CPU time. More insight on this is provided in Appendix D.3, where we examine the impact of *SkipList* size. The biggest improvement of *both* over *intra* is observed for the ETH dataset. The reason is as follows. Compared with 4SQ, the similarity shared among the objects in ETH is lower; compared with WX, ETH has less objects contained in each block. For both cases, more performance improvement is gained from using the skip list in the inter-block index.

[3] https://www.ethereum.org/

[4] MCL: https://github.com/herumi/mcl/

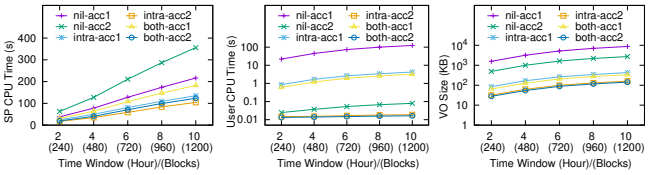


Figure 9: Time-Window Query Performance (4SQ)

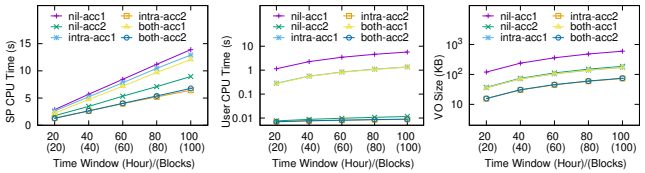


Figure 10: Time-Window Query Performance (WX)

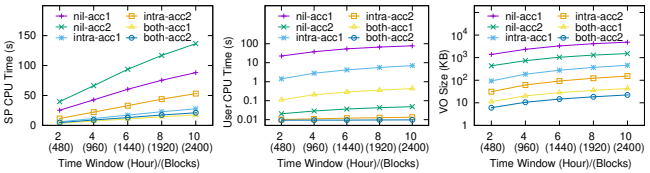


Figure 11: Time-Window Query Performance (ETH)

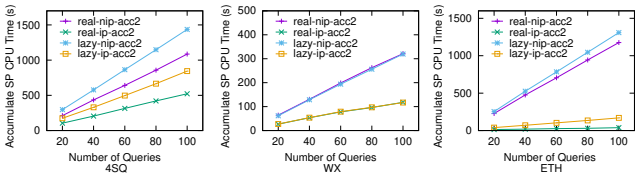


Figure 12: Subscription Query with IP-Tree Index

9.3 Subscription Query Performance

We next evaluate the performance for subscription queries. First, we examine the SP's query processing time with or without using the IP-Tree (denoted as *ip* and *nip*) under the default setting with both intra- and inter-block indexes enabled. We randomly generate different numbers of queries. We set the default subscription period as 2 hours for 4SQ and ETH and 20 hours for WX. As shown in Fig. 12, the IP-Tree reduces the SP's overhead by at least 50% in all cases tested. The performance gain in the ETH dataset (Fig. 12(c)) is more substantial thanks to the sparser distribution of the data.

To compare real-time and lazy authentications, we consider two real-time schemes (with *acc1* and *acc2*) and one lazy scheme (with *acc2* only, as *acc1* does not support the aggregation of accumulative sets and proofs). We vary the subscription period from 2 hours to 10 hours for 4SQ and ETH and 20 hours to 100 hours for WX. Figs. 13–15 show the results of varying the subscription period. Clearly, the lazy scheme performs much better than the real-time schemes in terms of the user CPU time. Furthermore, the CPU time and the VO size in the lazy scheme are increased only sub-linearly with increasing the subscription period. This is because the lazy scheme can aggregate the proofs of mismatching objects

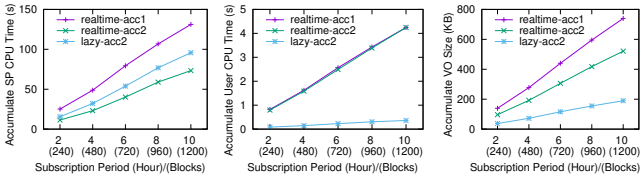


Figure 13: Subscription Query Performance (4SQ)

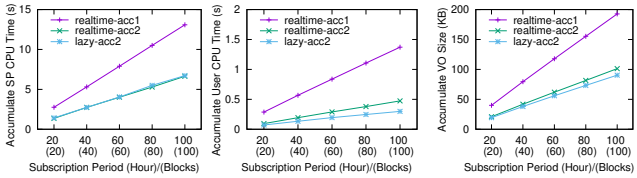


Figure 14: Subscription Query Performance (WX)

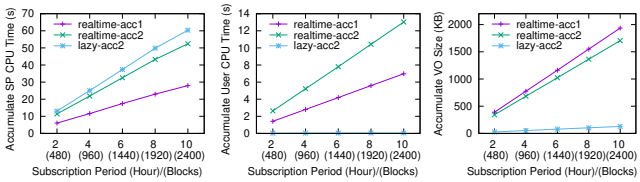


Figure 15: Subscription Query Performance (ETH)

across blocks. In contrast, the real-time schemes compute all the proofs immediately upon arrival of a new block, resulting in a worse performance. In terms of the SP CPU time, as the lazy scheme needs to sacrifice the SP's computation to aggregate mismatch proofs, its performance is generally worse than the real-time schemes when using the same accumulator.

10 CONCLUSION

In this paper, we have studied, for the first time in the literature, the problem of verifiable query processing over blockchain databases. We proposed the vChain framework to ensure the integrity of Boolean range queries for lightweight users. We developed a novel accumulator-based ADS scheme that transforms numerical attributes into set-valued attributes and hence enables dynamic aggregation over arbitrary query attributes. Based on that, two data indexes, namely tree-based intra-block index and skip-list-based inter-block index, and one prefix-tree-based index for subscription queries were designed, along with a series of optimizations. While our proposed framework has been shown to be practically implementable, the robustness of the proposed techniques was substantiated by security analysis and empirical results.

This paper opens up a new direction for blockchain research. There are a number of interesting research problems that deserve further investigation, e.g., how to support more complex analytics queries; how to leverage modern hardware such as multi- and many-cores to scale performance; and how to address privacy concerns in query processing.

REFERENCES

[1] S. Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system.

[2] G. Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151, 1–32.

[3] C. Mohan. 2017. Tutorial on blockchains and databases. *Proc. VLDB Endow.*, 10, 12, 2000–2001.

[4] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan. 2017. Blockbench: A framework for analyzing private blockchains. In *ACM SIGMOD*.

[5] T. T. A. Dinh, R. Liu, M. Zhang, G. Chen, B. C. Ooi, and J. Wang. 2018. Untangling blockchain: a data processing view of blockchain systems. *IEEE Trans. Knowl. Data Eng.*, 30, 7, 1366–1385.

[6] H. T. Vo, A. Kundu, and M. Mohania. 2018. Research directions in blockchain data management and analytics. In *EDBT*.

[7] S. Wang, T. T. A. Dinh, Q. Lin, Z. Xie, M. Zhang, Q. Cai, G. Chen, W. Fu, B. C. Ooi, and P. Ruan. 2018. ForkBase: An efficient storage engine for blockchain and forkable applications. *Proc. VLDB Endow.*, 11, 10, 1137–1150.

[8] Blockchair. 2018. A blockchain search and analytics engine for Bitcoin, Bitcoin Cash and Ethereum. (2018). https://blockchair.com/about.

[9] Vaultitude. 2018. Intellectual property blockchain platform. (2018). https://vaultitude.com/assets/downloads/AVaultitude_WhitePaper.pdf.

[10] B. M. Platz, A. Filipowski, and K. Doubleday. 2017. Flureedb: a practical decentralized database. (2017). https://flur.ee/assets/pdf/flureedb%5C_whitepaper%5C_v1.pdf.

[11] BigchainDB GmbH. 2018. Bigchaindb 2.0: the blockchain database. (2018). https://www.bigchaindb.com/whitepaper/bigchaindb-whitepaper.pdf.

[12] Wolk Inc. 2017. Wolk swarmdb: decentralized database services for web 3. (2017). https://wolk.com/whitepaper/WolkTokenGenerationEvent-20170717.pdf.

[13] Bitcoin. 2018. Running a full node. (2018). https://bitcoin.org/en/full-node.

[14] H. Pang and K.-L. Tan. 2004. Authenticating query results in edge computing. In *IEEE ICDE*.

[15] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. 2006. Dynamic authenticated index structures for outsourced databases. In *ACM SIGMOD*.

[16] Y. Yang, S. Papadopoulos, D. Papadias, and G. Kollios. 2008. Authenticated indexing for outsourced spatial databases. *The VLDB Journal*, 18, 3, 631–648.

[17] Q. Chen, H. Hu, and J. Xu. 2015. Authenticated online data integration services. In *ACM SIGMOD*.

[18] C. Xu, J. Xu, H. Hu, and M. H. Au. 2018. When query authentication meets fine-grained access control: A zero-knowledge approach. In *ACM SIGMOD*.

[19] R. C. Merkle. 1989. A certified digital signature. In *Advances in Cryptology — CRYPTO*, 218–238.

[20] G. Ateniese, A. Faonio, B. Magri, and B. de Medeiros. 2014. Certified bitcoins. In *Applied Cryptography and Network Security*, 80–96.

[21] J. Garay, A. Kiayias, and N. Leonardos. 2015. The bitcoin backbone protocol: Analysis and applications. In *Advances in Cryptology - EUROCRYPT*, 281–310.

[22] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse. 2016. Bitcoin-NG: A scalable blockchain protocol. In *USENIX NSDI*, 45–59.

[23] G. Pirlea and I. Sergey. 2018. Mechanising blockchain consensus. In *ACM SIGPLAN Int'l Conf. Certified Programs and Proofs*, 78–90.

[24] C. Dong, Y. Wang, A. Aldweesh, P. McCorry, and A. van Moorsel. 2017. Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing. In *ACM CCS*, 211–227.

[25] J. Camenisch, M. Drijvers, and M. Dubovitskaya. 2017. Practical uc-secure delegatable credentials with attributes and their application to blockchain. In *ACM CCS*, 683–699.

[26] IBM Blockchain. 2018. Enterprise blockchain solutions and services. (2018). https://www.ibm.com/blockchain.

[27] Oracle. 2018. Transforming the enterprise with oracle blockchain platform. (2018). https://www.oracle.com/cloud/blockchain/.

[28] SAP. 2018. Blockchain applications and services. (2018). https://www.sap.com/products/leonardo/blockchain.html.

[29] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. 2017. vSQL: verifying arbitrary SQL queries over dynamic outsourced databases. In *2017 IEEE Symposium on Security and Privacy*. IEEE, 863–880.

[30] B. Parno, J. Howell, C. Gentry, and M. Raykova. 2013. Pinocchio: nearly practical verifiable computation. In *Security and Privacy (SP), 2013 IEEE Symposium on*, 238–252.

[31] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. 2014. Succinct non-interactive zero knowledge for a von neumann architecture. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, 781–796.

[32] C. Papamanthou, R. Tamassia, and N. Triandopoulos. 2011. Optimal verification of operations on dynamic sets. In *Advances in Cryptology – CRYPTO*, 91–110.

[33] R. Canetti, O. Paneth, D. Papadopoulos, and N. Triandopoulos. 2014. Verifiable set operations over outsourced databases. In *Public-Key Cryptography – PKC*, 113–130.

[34] D. Papadopoulos, S. Papadopoulos, and N. Triandopoulos. 2014. Taking authenticated range queries to arbitrary dimensions. In *ACM CCS*.

[35] Y. Zhang, J. Katz, and C. Papamanthou. 2017. An expressive (zero-knowledge) set accumulator. In *IEEE European Symposium on Security and Privacy (EuroS&P)*.

[36] C. Xu, Q. Chen, H. Hu, J. Xu, and X. Hei. 2018. Authenticating aggregate queries over set-valued data with confidentiality. *IEEE Trans. Knowl. Data Eng.*, 30, 4, 630–644.

[37] D. Schroeder and H. Schroeder. 2012. Verifiable data streaming. In *ACM CCS*.

[38] C. Papamanthou, E. Shi, R. Tamassia, and K. Yi. 2013. Streaming authenticated data structures. In *Advances in Cryptology – EUROCRYPT*, 353–370.

[39] D. Schöder and M. Simkin. 2015. VeriStream – A framework for verifiable data streaming. In *Financial Cryptography and Data Security*, 548–566.

[40] S. Papadopoulos, Y. Yang, and D. Papadias. 2009. Continuous authentication on relational streams. *The VLDB Journal*, 19, 2, 161–180.

[41] L. Chen, G. Cong, and X. Cao. 2013. An efficient query indexing mechanism for filtering geo-textual data. In *ACM SIGMOD*.

[42] C. Thoma, A. J. Lee, and A. Labrinidis. 2016. PolyStream. Cryptographically enforced access controls for outsourced data stream processing. In *ACM Symposium on Access Control Models and Technologies - SACMAT*.

[43] K. Yang, K. Zhang, X. Jia, M. A. Hasan, and X. Shen. 2017. Privacy-preserving attribute-keyword based data publish-subscribe service on cloud platforms. *Information Sciences*, 387, 116–131.

[44] D. Boneh, X. Boyen, and H. Shacham. 2004. Short group signatures. In *Advances in Cryptology – CRYPTO*, 41–55.

Algorithm 6: IP-Tree Construction (by the SP)

Function `BuildIPTree({query})`
  Input: All subscription queries {*query*}
  root.grid ← *FullRange*;
  Create an empty queue *queue*;
  queue.enqueue(⟨*root*, {*query*}⟩);
  while *queue* **is not** *empty* **do**
    ⟨*node*, {$query_n$}⟩ ← *queue.dequeue*();
    for q_i **in** {$query_n$} **do**
      if q_i *full covers node.grid* **then**
        Add ⟨q_i, full⟩ to *node.RCIF*;
      else Add ⟨q_i, partial⟩ to *node.RCIF*;
    Build *node.BCIF* from fully cover queries;
    for *sub_g* **in** *Split(node.grid)* **do**
      sub_n.grid ← *sub_g*;
      sub_q ← {*q* | *node.RCIF*[*q*] = partial ∧ *intersect*(*q*, *sub_g*)};
      queue.enqueue(⟨*sub_n*, {*sub_q*}⟩);
      node.children.add(*sub_n*);

[45] J. Camenisch, M. Kohlweiss, and C. Soriente. 2009. An accumulator based on bilinear maps and efficient revocation for anonymous credentials. In *Public Key Cryptography – PKC*, 481–500.

[46] D. Yang, D. Zhang, and B. Qu. 2015. Participatory cultural mapping based on collective behavior in location based social networks. *ACM Transactions on Intelligent Systems and Technology.*

A PSEUDO CODES OF THE IP-TREE ALGORITHMS

Algorithms 6 and 7 respectively show the construction and query processing algorithms of the IP-tree index introduced in Section 7.

B PROOF OF THEOREM 8.1

Theorem 8.1. *The constructions of the multiset accumulator presented in Section 5.2 satisfy the security property of the unforgeability as defined in Definition 8.1.*

Proof. We omit the proof of this theorem for the first construction of the multiset accumulator in Section 5.2, as it has been shown to hold under the q-SDH assumption in [32].

We prove this theorem holds for the second construction by contradiction. Support that there is an adversary who outputs two multisets X_1 and X_2, where $X_1 \cap X_2 \neq \varnothing$, and a valid set disjoint proof π. This means that

$$e(d_A(X_1), d_B(X_2)) = e(\pi, g) \Rightarrow$$
$$e(g^{A(X_1)}, g^{B(X_2)}) = e(\pi, g) \Rightarrow$$
$$e(g, g)^{A(X_1)B(X_2)} = e(\pi, g) \Rightarrow$$
$$\pi = g^{A(X_1)B(X_2)}$$

On the other hand, because $X_1 \cap X_2 \neq \varnothing$, we can get $Cs^q \in A(X_1)B(X_2)$, where C is a non-zero constant. That is, $A(X_1)B(X_2) = Cs^q + Q(s)$, where $Q(s)$ is some polynomial without the s^q term. Therefore, the adversary can get

$$\pi = g^{A(X_1)B(X_2)} = g^{Cs^q} \cdot g^{Q(s)} \Rightarrow$$
$$g^{s^q} = (\pi / g^{Q(s)})^{C^{-1}}$$

which violates the q-DHE assumption. Therefore, by contradiction, the theorem holds. □

Algorithm 7: Subscription Query w. Intra-Index (by the SP)

Function `SubscriptionIPTree(`r_{IP}, *IntraRoot*`)`
  Input: IP-Tree root r_{IP}, Intra-Index root *IntraRoot*
  Create an empty queue $queue_1$;
  $Q \leftarrow \{\}$;
  $queue_1$.*enqueue*(⟨*IntraRoot*, Q⟩);
  while $queue_1$ **is** *not empty* **do**
    ⟨*node*, Q⟩ ← $queue_1$.*dequeue*();
    $Q \leftarrow$ `QueryIntraNode(`r_{IP}, *node*, Q`)`;
    for *n* **in** *node.children* **do** $queue_1$.*enqueue*(⟨*n*, Q⟩);
Function `QueryIntraNode(`r_{IP}, n_{intra}, Q`)`
  Input: IP-Tree root r_{IP}, Intra-Index node n_{intra}, processed mismatching queries Q
  Output: processed mismatching queries Q
  Create an empty queue $queue_2$ and enqueue r_{IP};
  while $queue_2$ **is** *not empty* **do**
    $n \leftarrow queue_2$.*dequeue*();
    $q_f \leftarrow \{q | n.RCIF[q] = \text{full}\} \backslash Q$;
    for ⟨Υ, qs⟩ **in** *n.BCIF* **do**
      if $n_{intra}.W \cap \Upsilon = \varnothing$ **then**
        $\pi \leftarrow$ ProveDisjoint($n_{\text{intra}}.W, \Upsilon, pk$);
        Add ⟨π, Υ⟩ to q.VO $\forall q \in (q_f \cap qs) \backslash Q$;
        $Q \leftarrow Q \cup (q_f \cap qs)$;
    if n_{intra} **is** *leaf node* **then**
      Add $n_{\text{intra}}.o$ to $q.R$ for $q \in q_f \backslash Q$;
    $q' \leftarrow \{\}$;
    for n' **in** *n.children* **do**
      if n' *intersects* n_{intra} **then** $q' \leftarrow q' \cup n'.queries$;
    $q_p \leftarrow \{q | n.RCIF[q] = \text{partial}\} \backslash q'$;
    for n' **in** *n.children* **do**
      if n' *intersects* n_{intra} **then** $queue_2$.*enqueu*(n') ;
      else
        $qs \leftarrow q_p \cap n'.queries \backslash Q$;
        $W' \leftarrow \text{trans}(n'.grid)$;
        $\pi \leftarrow$ ProveDisjoint($n_{\text{intra}}.W, W', pk$);
        Add ⟨π, W'⟩ to q.VO $\forall q \in qs$;
        $Q \leftarrow Q \cup qs$;
  return Q;

C PROOF OF THEOREM 8.2

Theorem 8.2. *Our proposed query authentication algorithms satisfy the security property of the unforgeability as defined in Definition 8.2.*

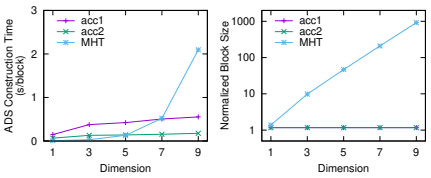


Figure 16: Comparison with MHT

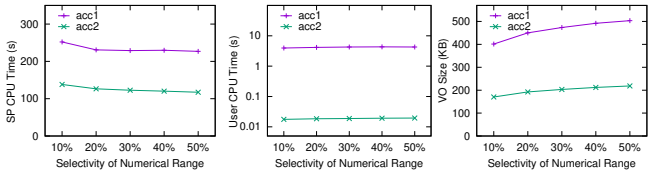


Figure 17: Impact of Selectivity (4SQ)

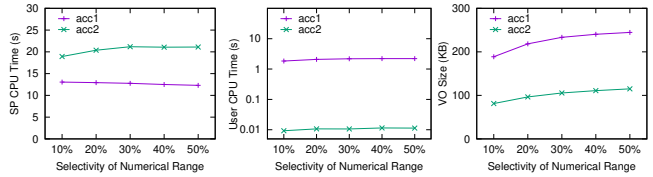


Figure 18: Impact of Selectivity (WX)

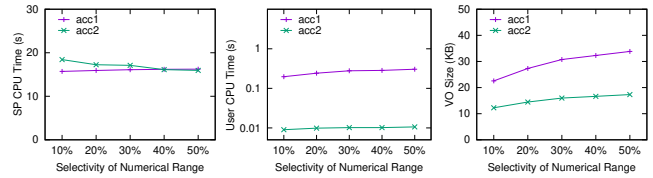


Figure 19: Impact of Selectivity (ETH)

Proof. We prove this theorem by contradiction.

Case 1: The result R contains an object o^* such that $o^* \notin \{o_i\}$.

Recall that in the result verification procedure, the verifier will check the integrity of the object with respect to the *MerkleRoot* stored in the blockchain. Therefore, a successful forge means either a collision of the underlying cryptographic hash function, or a break of the blockchain protocol, which yields a contradiction.

Case 2: The result R contains an object o^* such that o^* does not satisfy the query q.

It is trivial to see that such a case is impossible, as the verifier will check it locally.

Case 3: There exists an object o_x in the query window or subscription period, which is not in R but satisfies the query q.

First, note that the verifier (running a light node) syncs block headers with the blockchain network. Thus, the verifier always verifies the results with respect to the latest block header. Now suppose there is a missing object o_x. During verification, the verifier will examine the multiset accumulative values which cover the whole query window or subscription period. The missing object o_x must fall under one multiset accumulative value in the VO. As discussed in Section 5.1, such a missing object implies that the attribute of this matching object intersects with the equivalent multiset of some part of the query condition. This means that the adversary is able to construct two multisets X_1 and X_2, such that $X_1 \cap X_2 \neq \emptyset$, along with a corresponding set disjoint proof, which contradicts to Theorem 8.1. □

D SUPPLEMENTAL EXPERIMENT RESULTS

This section presents some supplemental experiment results.

D.1 Comparison with MHT

As mentioned in Section 5, one major drawback of the traditional MHT-based solution is its prohibitively high overhead to support queries involving arbitrary attributes for multi-dimensional databases. To demonstrate that, we synthesize several datasets with different dimensionalities using the WX dataset and experimentally compare the MHT solution with our accumulator-based solutions in terms of the setup cost. Note that the original *weather description* attribute is removed from the synthetic datasets, since MTHs cannot work with set-valued attributes. As shown in Fig. 16(a), the construction time of our solutions is only slightly increased with dimensionality. In contrast, the MHT construction time is dramatically increased because it needs to build an MHT for every combination of attributes. Furthermore, to examine the ADS overhead, we show the average size of the ADS-embedded blocks in Fig. 16(b) (normalized by the original block size, plotted in log scale). While our solutions have a negligible fixed-size ADS regardless of data dimensionality, the space overhead incurred by MTH grows exponentially with dimensionality. In particular, when the dimensionality is higher than 3, the MHT's ADS overhead is more than 10X–1,000X the original block size, which is unacceptable for practical use since the ADS would dominate the traffic and storage cost of a blockchain network.

D.2 Impact of Selectivity

Figs. 17–19 show the time-window query performance by varying the selectivity of the range predicate from 10% to 50%. The window size is fixed at 10 hours for 4SQ and ETH and at 100 hours for WX. Both the intra-block and inter-block indexes are enabled. For all datasets, the SP CPU time is generally decreased with increasing selectivity. This is because the SP query processing time is dominated by the proving of mismatching objects. As a result, the more the objects selected, the less is the SP overhead. In contrast, the user CPU time remains largely the same under different settings. As for the VO size, it is slightly increased because of a larger number of hashes introduced by more query results.

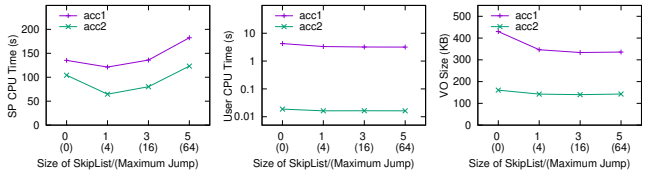


Figure 20: Impact of SkipList Size (4SQ)

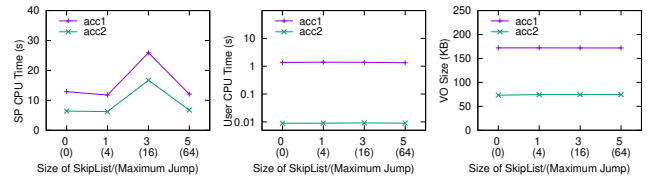


Figure 21: Impact of SkipList Size (WX)

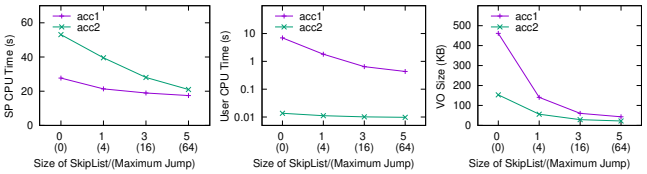


Figure 22: Impact of SkipList Size (ETH)

Overall, we can see that our solution performs efficiently under a wide range of selectivity settings.

D.3 Impact of SkipList

This section investigates the impact of *SkipList* in the inter-block index. We set the query windows size the same as in Appendix D.2. The size of *SkipList* is varied from 0 to 5. Note that a size of zero means that we employ only an intra-block index but no inter-block index. For the other settings, both the intra-block and inter-block indexes are employed. Figs. 20–22 show the results for the three datasets, respectively. It is interesting to observe that the SP CPU time exhibits different trends for different datasets. This can be explained as follows. On the one hand, the *SkipList* helps aggregate mismatch proofs across consecutive blocks. With an increased size of *SkipList*, more mismatch proofs can be aggregated without accessing the actual blocks; hence the SP CPU time is reduced and a smaller VO is resulted. On the other hand, the larger the *SkipList*, the more set elements will be added up as the input of the accumulator, which increases the SP CPU time. Furthermore, the effectiveness of using *SkipList* for aggregating mismatch proofs depends on the distribution of the data. The combination of these factors contributes to the final SP CPU time. As a result, we observe fluctuations in SP CPU time for 4SQ and WX but a steady decrease in ETH when the *SkipList* size increases. Nevertheless, the user CPU time and the VO size are both monotonically reduced thanks to the aggregation of mismatching objects by the inter-block index. Comparing *acc*1 and *acc*2, since *acc2* can support online aggregation, its VO size and also user CPU time are further reduced compared with those of *acc1* in all cases tested.

```
1  pragma solidity ^0.4.0;
2
3  contract vChainContract {
4    struct BlockHeader {
5      bytes32 PreBkHash;
6      bytes32 MerkleRoot;
7      bytes32 SkipListRoot;
8    }
9    struct Block {
10     BlockHeader header;
11     IntraIndex intraindex;
12     InterIndex interindex;
13     Object[] objects;
14   }
15   //stores each block
16   mapping(bytes32=>Block) chainstorage;
17
18   function BuildvChain(Object[] _objects,
19                        bytes32 _PreBkHash) public {
20     Block block;
21     BlockHeader _header;
22     _header.PreBkHash = _PreBkHash;
23     (_header.MerkleRoot, block.intraindex) =
           BuildIntraIndex(_objects);
24     (_header.SkipListRoot, block.interindex) =
           BuildInterIndex(chainstorage);
25     bytes32 _currentBkHash = sha3(_header);
26     block.objects = _objects;
27     block.header = _header;
28     chainstorage[_currentBkHash] = block;
29   }
30 }
```

Listing 1: Implementing vChain in Smart Contract

E PRACTICAL IMPLEMENTATION

This section addresses the practical implementation issues of our proposed vChain framework. Since we need to compute an ADS and embed it into each block, the existing blockchains cannot be used directly. There are two possible solutions. First, we can develop a new chain by extending an open-source blockchain project. Second, we can leverage smart contracts, trusted programs running on blockchains, to build a *logical* chain that constructs and maintains the ADS for each block, on top of an existing blockchain (e.g., Ethereum [2] or Hyperledger [26]). The advantage of the second solution is that we do not need to be bothered by the underlying system implementation, but focus on writing smart contracts to build the logical chain. Listing 1 shows an example of Ethereum smart contract that implements the vChain framework. Specifically, lines 4–14 define the structures of a block header and a block. The mapping structure, *chainstorage*, indexes each block with the block hash (line 16). The function *BuildvChain* first constructs the block header, including the intra-block index and the inter-block index (lines 22–24). Then, the block hash is computed (line 25) and, together with the input objects, assigned to the current block (lines 26–27). Finally, the newly constructed block is added to the *chainstorage* using its block hash (line 28).