This article has been accepted for publication in IEEE Transactions on Knowledge and Data Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2023.3316127



# veffChain: Enabling Freshness Authentication of Rich Queries over Blockchain Databases

Qin Liu, *Member, IEEE,* Yu Peng, *Graduate Student Member, IEEE,* Ziyi Tang, *Graduate Student Member, IEEE,* Hongbo Jiang, *Senior Member, IEEE,* Jie Wu, *Fellow, IEEE,* Tian Wang, *Member, IEEE,* Tao Peng, *Member, IEEE,* and Guojun Wang, *Member, IEEE*

**Abstract**—With the wide adoption of blockchains in data-intensive applications, enabling verifiable queries over a blockchain database is urgently required. Aiming at reducing costs, previous solutions embed a small-sized authenticated data structure (ADS) in each block header, so that a user can verify search results without maintaining a full copy of blockchain databases. However, existing studies focus on exact queries with difficulty to guarantee the freshness of search results. In this paper, we propose two frameworks, called veffChain and veffChain++, to realize freshness authentication of rich queries over blockchain databases. Specifically, veffChain concerns about verifiable latest-$K$ exact queries and employs RSA accumulator to generate constant-size ADSs; veffChain++ integrates RSA accumulator into the Trie tree to further authenticate latest-$K$ fuzzy queries. For improved scalability, an adaptive keyword splitting (AKS) solution is proposed to enable ADSs to be incrementally updated. Compared with the state-of-the-art work, our frameworks have the following merits: (1) *Freshness Guarantee*. The user can efficiently retrieve the freshest data from a blockchain database in a verifiable way. (2) *Flexibility*. The user can specify different query patterns on demand to retrieve data as accurately as possible. The detailed security analysis and extensive experiments validate the practicality of our frameworks.

**Index Terms**—Blockchain, latest-$k$ queries, fuzzy matches, verifiability

## 1 Introduction

DRIVEN by the great success of cryptocurrency systems, blockchain technology has attracted tremendous attention from all circles of society [1]. A blockchain is a public ledger where all the data is stored in a chain of blocks collectively maintained by a network of mutually untrusted nodes [2]. Because of the benefits of persistency and tamper resistance, blockchains have been widely applied to preserve valuable data in decentralized applications, such as healthcare and credit record management. The ever-increasing data volume creates a huge demand for users to retrieve data of interest by querying blockchain databases. In this trend, how to ensure the authenticity of search results offered by untrusted nodes has become a key problem.

A typical blockchain network consists of two types of nodes: light nodes and full nodes. A light node maintains only block headers that include consensus proofs and data digests, while a full node maintains a full copy of the blockchain database, including both block headers and complete data. A naive solution is letting a user join as a full node querying the blockchain database locally. The main insufficiency of this approach is the huge resources consumed on the user side (e.g., a full node in the Bitcoin network needs to have at least 500GB free disk). To address this, previous solutions [3] put forward to embed a small-sized authenticated data structure (ADS) in each block header, so that the user can join as a light node querying full nodes and verifying search results in a light-weighted way. Despite the reduced costs, existing studies mainly focus on exact queries with difficulty to guarantee the freshness of search results.

In many cases, the user wants to retrieve data as accurately as possible when he has only limited knowledge about the underlying data he is searching for. For example, bank staff can enter "$Fin?n?e$" to retrieve the credit records containing the keyword "$Finance$", and a doctor can enter "$Arter*$" to retrieve the medical records containing the keyword "$Arteriosclerosis$", when they are unsure about the exact spellings of search terms. Beyond that, result freshness is essential for time-sensitive applications. For example, a bank is more interested in a client's latest credit records to assess the loan risk; A doctor requires a patient's latest medical examination reports to produce a diagnosis. Therefore, the features of supporting fuzzy matches and freshness authentication are especially important for improving user experience while querying a verifiable blockchain database.

In this paper, we propose two verifiable frameworks with freshness and flexibility assurance, named veffChain and veffChain++, to realize freshness authentication of rich queries over blockchain databases. Specifically, veffChain concerns about verifiable latest-$K$ exact queries, where data is sorted by the ascending order of their timestamps and RSA accumulator [4] is employed to generate a constant-size ADS summarizing the ordering information; while veffChain++ designs a VTrie tree by integrating RSA accumulator into the traditional Trie tree [5] to further authenticate latest-$K$ fuzzy queries. For improved scalability, an adaptive keyword splitting (AKS) solution is proposed

*Qin Liu, Yu Peng, Ziyi Tang, and Hongbo Jiang are with the College of Computer Science and Electronic Engineering, Hunan University, Changsha, Hunan Province, P.R. China, 410082. E-mail: {gracelq628, pengyu411, ziyitang}@hnu.edu.cn; hongbojiang2004@gmail.com*
*Jie Wu is with the Department of Computer and Information Sciences, Temple University, Philadelphia, PA 19122, USA. E-mail: jiewu@temple.edu*
*Tian Wang is with the Institute of Artificial Intelligence and Future Networks, Beijing Normal University & UIC, Zhuhai, Guangdong Province, P. R. China, 519000. E-mail: cs_tianwang@163.com*
*Tao Peng and Guojun Wang are with the School of Computer Science and Cyber Engineering, Guangzhou University, Guangzhou, Guangdong Province, P.R. China, 510006. E-mail: {pengtao, csgjwang}@gzhu.edu.cn*

to enable the ADS embedded in a new block header to be incrementally updated from that in the previous block header. Detailed discussions are also provided to improve query performance and support Boolean range queries. The main contributions of this paper are summarized as follows:

- To the best of our knowledge, this is the first attempt to devise built-in ADSs to realize verifiable latest-$K$ exact and fuzzy queries in blockchain databases.
- We propose two blockchain frameworks to enable freshness authentication of rich queries and propose an AKS solution for improved scalability. Compared with the state-of-the-art work, our frameworks have the following merits: (1) *Freshness Guarantee*. The user can efficiently retrieve the freshest data from a blockchain database in a verifiable way; (2) *Flexibility*. The user can specify different query patterns on demand to retrieve data as accurately as possible.
- We conduct formal security analyses and an empirical study to validate the proposed frameworks.

**Paper Organization.** We introduce the related work in Section 2, before formulating the problem in Section 3. We construct the proposed solutions in Sections 4 and 5 before describing the AKS solution in Section 6 and analyzing the performance and security in Section 7. After discussing the extensions in Section 8, we evaluate the proposed solutions in Section 9. Finally, we conclude this paper in Section 10.

## 2 Related Work

### 2.1 Blockchain Structure

A blockchain consists of a series of blocks, and each block keeps a pointer to the previous block hence forming a chain [1]. Each block consists of two parts: header and body. The block body contains a collection of transactions and a Merkle hash tree (MHT) built based on these transactions. The block header mainly includes four parts: (1) PreHash, the hash value of the previous block. (2) TimeStamp (TS), the time of block generation. (3) ConsProof, the consensus proof data. (4) MerkleRoot, the root hash of MHT. Although the blockchain is maintained by untrusted peers, the ConsProof guarantees that all peers hold identical data replicas, while the MerkleRoot ensures data authenticity. To authenticate a transaction, a user reconstructs the MHT by using a verification object (VO) returned by a full node, and compares its root hash with the MerkleRoot in the block header [3].

### 2.2 Verifiable Query Technologies in Blockchain

To ensure the authenticity of results returned by untrusted full nodes, existing studies usually construct an ADS based on verifiable query techniques, such as accumulator and MHT [4, 6, 7]. Dai et al. [8] integrated Bloom Filter (BF) into MHT to realize verifiable historical transactions in Bitcoin systems. To enrich query expressions, Xu et al. [3] proposed the vChain framework, which implemented verifiable boolean range queries and subscription queries based on the accumulator technology. However, vChain suffered from the limitations of linear-scan search performance in the worst case and a large size of public keys. To overcome these problems, Wang et al. [9] proposed vChain+ by designing a sliding window-based accumulator index and an object registration index. In addition, Peng et al. [10] presented a collaborative blockchain database by utilizing accumulator-based ADSs to provide verifiable keyword and range queries. Another line of work focused on providing verifiable query services in hybrid storage systems combining on-chain and off-chain storages. Zhu et al. [11] designed a blockchain database to provide SQL-like verifiable queries based on Merkle B-tree. To realize multiple complex analytical query primitives, Pei et al. [12] proposed a verifiable query scheme over hybrid storage blockchains by using Merkle semantic Trie-based indexing technique. Wu et al. [13] designed a verifiable query layer deployed in the cloud and utilized Merkle Patricia Tree to provide verifiable query services for blockchain systems. To reduce the GAS cost of smart contracts, Zhang et al. [14] replaced the expensive write operations with light-weighted operations (e.g., read and compute). Their subsequent work [15] proposed a new index structure based on chameleon vector commitment to realize constant GAS costs. In summary, abundant researches have been proposed aiming at improving search efficiency and query expressions in verifiable blockchain systems. However, most of them support only exact queries without considering freshness guarantee.

### 2.3 Verifiable Fuzzy Queries and Freshness Queries

To improve user experience, Li et al. [16] used locality-sensitive hashing (LSH), BF, and homomorphic message authentication code (MAC) to achieve verifiable ranked fuzzy queries. For improved efficiency, Tong et al. [17] constructed an index tree based on the graph-based keyword partition algorithm to achieve adaptive sublinear retrieval. However, the above verifiable fuzzy query schemes are hard to reach completely accurate search due to the inherent false positive and false negative of BF and LSH, respectively. To solve this problem, Shao et al. [18] proposed a wildcard-based verifiable fuzzy query scheme, which integrated keyed-hash MAC into a Trie tree to ensure accuracy. However, this method required all the users to share the key of MACs for verification. That is, the security would be compromised if the key was exposed to untrusted servers [19].

As for verifiable freshness queries, Jin et al. [20] exploited broadcast encryption, key regression, and MHT to achieve instant freshness check for cloud storges. Zhu et al. [21] guaranteed the freshness of cloud data by developing a timestamp-chain. Hu et al. [22] designed a linked key span MHT to provide real-time freshness guarantee for outsourced key-value stores. In summary, existing fuzzy/freshness query schemes were devised in the context of data outsourcing without considering the unique features of blockchain, e.g., the append-only mode and the consistency of data replicas. Besides, the Trie tree supports fast and accurate matches of wildcards (e.g., '?' and '$*$'), and thus could be regarded as the building block of veffChain++.

## 3 Problem Formulation

### 3.1 The System and Threat Model

As shown in Fig. 1, our system model consists of full nodes maintaining the entire blockchain database, and light nodes retaining only the block headers. According to different roles in the verifiable query process, the nodes can be divided into miners, users, and service providers (SPs).

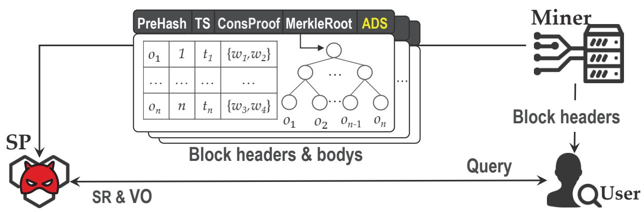



Fig. 1: System and threat model.

• **Miner**. The miner as a full node is responsible for constructing consensus proofs and appending new blocks to the blockchain. To enable verifiable searches, the miner constructs an ADS and embeds it into each regular header. The data in the blockchain database is modeled as a collection of objects. Each object $o_x$ is represented as $(x, t_x, W_x)$, where $x$ is the object identifier, $t_x$ is the timestamp when the object is generated, and $W_x$ is a set of relevant keywords.

• **User**. The user as a light node wishes to retrieve the latest objects of interest, and is allowed to verify search results by using the ADS in a block header and the VO returned by the SP. For improved query experience, this work is committed to supportting two kinds of queries:

- Latest-$K$ exact query. $\mathcal{Q} = (\mathcal{T}, K)$, where $\mathcal{T}$ is an exact search term excluding any wildcards. It retrieves the latest $K$ objects containing keyword $w$ s.t. $w = \mathcal{T}$.
- Latest-$K$ fuzzy query. $\widetilde{\mathcal{Q}} = (\widetilde{\mathcal{T}}, K)$, where $\widetilde{\mathcal{T}}$ as a fuzzy search term contains wildcards. It retrieves the latest $K$ objects for all keywords similar to $\widetilde{\mathcal{T}}$.

• **SP**. The SP as a full node offers verifiable query services to the user. On receiving a query request, the SP searches the blockchain database and constructs a VO that will be returned together with the search results $\mathcal{SR}$. Specifically, for a latest-$K$ exact query $\mathcal{Q} = (\mathcal{T}, K)$, the SP locates the keyword equal to the exact search term $\mathcal{T}$ and puts the $K$ objects with the latest timestamps into $\mathcal{SR}$. For a latest-$K$ fuzzy query $\widetilde{\mathcal{Q}} = (\widetilde{\mathcal{T}}, K)$, the SP first finds all the keywords similar to the fuzzy term $\widetilde{\mathcal{T}}$, and for each similar keyword, it puts the $K$ objects with the latest timestamps into $\mathcal{SR}$.

In our threat model, the SP is assumed to be a potential adversary and may return incorrect search results unintentionally or intentionally. Specifically, the user authenticates the search results $\mathcal{SR}$ from the following aspects:

- **Integrity**. All the objects in $\mathcal{SR}$ satisfy the query criteria, and haven't been tampered with.
- **Freshness**. No object outside $\mathcal{SR}$ satisfies the query criteria and has later timestamp than an object in $\mathcal{SR}$.
- **Completeness**. No keyword similar to the fuzzy search term is overlooked in the search process.

Note that, for latest-$K$ exact queries, the user only needs to verify result integrity and freshness, but for latest-$K$ fuzzy queries, the user needs to verify all these properties.

### 3.2 Notations

Let $\lambda \in \mathbb{N}$ be a security parameter. Notation $[x, y]$ represents the set of integers $\{x, \ldots, y\}$, which can be abbreviated as $[y]$ when $x = 1$. For a finite set $X = \{x_1, \ldots, x_n\}$, notation $|X|$ denotes its cardinality. The set of binary strings of length $x$ is denoted by $\{0,1\}^x$ and the set of finite binary strings is denoted by $\{0,1\}^*$. Given a string $\mathcal{S}$, $|\mathcal{S}|$ refers to the number of characters in $\mathcal{S}$, and $\mathcal{S}[x, y]$ denotes the substring starting from the $x$-th character and ending at the $y$-th character of string $\mathcal{S}$, which can be abbreviated as $\mathcal{S}[y]$ when $x = 1$. Notation $\|$ denotes string concatenation.

TABLE 1: Summary of Notations

| Notations | Descriptions |
|---|---|
| $\mathcal{B}_{[i]}$ | A sequence of blocks $(\mathcal{B}_1, \ldots, \mathcal{B}_i)$ for $i \in [t]$ |
| $(w_j, ln_j)$ | The keyword/latest-number pair of keyword $w_j$ |
| $(w_j, hn_j)$ | The keyword/history-number pair of keyword $w_j$ |
| $(w_j, un_j)$ | The keyword/update-number pair of keyword $w_j$ |
| $(w_j, id_k, k)$ | The $k$-th keyword/object/serial tuple of keyword $w_j$ |
| $\mathsf{W}_i, \mathsf{W}_{[i]}$ | The set of keywords in block $\mathcal{B}_i$ and blocks $\mathcal{B}_{[i]}$ |
| $\mathsf{LN}_{[i]}$ | The latest number set constructed from blocks $\mathcal{B}_{[i]}$ |
| $\mathsf{SO}_{[i]}$ | The sorted object set constructed from blocks $\mathcal{B}_{[i]}$ |
| $\mathsf{SO}_{[i]}.j$ | The sorted object subset associated with keyword $w_j$ |
| $\mathsf{HN}_i, \mathsf{HN}_{[i]}$ | The history number sets for block $\mathcal{B}_i$ and blocks $\mathcal{B}_{[i]}$ |
| $\mathsf{UN}_i, \mathsf{UN}_{[i]}$ | The update number sets for block $\mathcal{B}_i$ and blocks $\mathcal{B}_{[i]}$ |
| $\mathbf{acc}(X)$ | The accumulative value of set $X$ |

The blockchain consists of a serial of blocks $(\mathcal{B}_1, \ldots, \mathcal{B}_t)$. We use $\mathcal{B}_{[i]}$ to denote a sequence of blocks $(\mathcal{B}_1, \ldots, \mathcal{B}_i)$ for $i \in [t]$. The set of keywords updated in block $\mathcal{B}_i$ and blocks $\mathcal{B}_{[i]}$ are denoted by $\mathsf{W}_i$ and $\mathsf{W}_{[i]} = \cup_{k=1}^{i} \mathsf{W}_k$, respectively. Each keyword $w_j$ is associated with a keyword/latest-number pair $(w_j, ln_j)$, a keyword/history-number pair $(w_j, hn_j)$, a keyword/update-number pair $(w_j, un_j)$, and a set of keyword/object/sequence-number tuples $\{(w_j, id_k, k)\}_{k=1}^{ln_j}$. A latest number set and a sorted object set constructed from blocks $\mathcal{B}_{[i]}$ are denoted by $\mathsf{LN}_{[i]} = \{(w_j, ln_j)\}_{w_j \in \mathsf{W}_{[i]}}$ and $\mathsf{SO}_{[i]} = \bigcup_{w_j \in \mathsf{W}_{[i]}} \mathsf{SO}_{[i]}.j$, respectively, where $\mathsf{SO}_{[i]}.j = \{(w_j, id_k, k)\}_{k=1}^{ln_j}$ is the sorted object subset of keyword $w_j$. A history number set and an update number set constructed from block $\mathcal{B}_i$ are denoted by $\mathsf{HN}_i = \{(w_j, hn_j)\}_{w_j \in \mathsf{W}_i}$ and $\mathsf{UN}_i = \{(w_j, un_j)\}_{w_j \in \mathsf{W}_i}$, respectively, with $\mathsf{HN}_{[i]} = \bigcup_{k=1}^{i} \mathsf{HN}_k$ and $\mathsf{UN}_{[i]} = \bigcup_{k=1}^{i} \mathsf{UN}_i$ denoting the corresponding union sets constructed from blocks $\mathcal{B}_{[i]}$. For quick reference, the most relevant notations are shown in Table 1.

### 3.3 Cryptographic Preliminaries

**RSA Accumulator [4].** It provides a constant-size digest for an arbitrarily large set and a constant-size witness to verify the (non-)membership of any elements in this set. Let $\mathbf{N} = p \cdot q$, where $p, q$ are two large primes such that $|p \cdot q| > 3\lambda$, let $g$ be the generator of a cyclic group $\mathrm{QR}_{\mathbf{N}}$, and let $\mathcal{H} : \{0,1\}^* \to \{0,1\}^\lambda$ be a collision-resistant hash function. RSA accumulator takes the public key $pk = \{\mathbf{N}, (g, \mathrm{QR}_{\mathbf{N}})\}$ as the implicit input of all the following algorithms:

• $\mathsf{GenAcc}(X) \to \mathbf{acc}(X)$ : Given a set of elements $X = \{x_1, \ldots, x_n\}$ with $x_i \in \{0,1\}^*$, this algorithm generates the accumulative value $\mathbf{acc}(X) \leftarrow g^{\prod_{i=1}^{n} \mathcal{P}(\mathcal{H}(x_i))} \mod \mathbf{N}$ [1].

• $\mathsf{GenWit}(Y, X) \to \pi$ : It generates the witness $\pi$ for $Y \subseteq X$ as $\mathbf{acc}(X - Y) \leftarrow g^{\prod_{x_i \in (X-Y)} \mathcal{P}(\mathcal{H}(x_i))} \mod \mathbf{N}$.

• $\mathsf{VeriWit}(Y, \pi, \mathbf{acc}(X)) \to \{0,1\}$ : This algorithm checks the witness regarding $Y \subseteq X$, and outputs 1 only when $\pi^{\prod_{x_i \in Y} \mathcal{P}(\mathcal{H}(x_i))} \mod \mathbf{N} = \mathbf{acc}(X)$.

The security of RSA accumulator is based on strong RSA assumption. That is, given the public key and set $X$, the difficulty of finding $x' \notin X$ and $\pi'$ s.t. $\pi'^{\mathcal{P}(\mathcal{H}(x'))} \mod \mathbf{N} = \mathbf{acc}(X)$ equals that of solving the strong RSA problem.

**Trie Tree [5]**. It is an ordered multi-way tree data structure, where each node contains a character and denotes a string of characters in the path from the root to itself. The

1. $\mathcal{P}(x_i) \in \{0,1\}^{3\lambda}$ denotes the prime number corresponding to element $x_i$. It can be implemented by a two-universal hash function [23].

This article has been accepted for publication in IEEE Transactions on Knowledge and Data Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2023.3316127



time complexity for searching a string $\mathcal{S}$ from a Trie tree is $O(|\mathcal{S}|)$. When a Trie tree is used to store keywords, the route from the root to every leaf node results in the generation of a specific keyword. By traversing the Trie tree, it is possible to quickly locate the keyword equal to an exact search term and all the keywords similar to a fuzzy search term. In this paper, we assume that each keyword/search term is appended with a beginning symbol $'\$'$ and an ending symbol $'\#'$. Appendix A illustrates an example of Trie tree.

## 4 THE VEFFCHAIN FRAMEWORK

### 4.1 The Strawman Solution

As a starting point, we describe a strawman solution that realizes verifiable latest-$K$ exact queries over blockchain. For ease of understanding, we introduce the following definitions related to a sequence of blocks $\mathcal{B}_{[i]}$ where $i \in [t]$:

**Definition 1** (Latest Number Set)**.** *Each keyword $w_j \in \mathsf{W}_{[i]}$ is associated with a keyword/latest-number pair $(w_j, ln_j)$, which means that the latest number of objects containing keyword $w_j$ is $ln_j$ when block $\mathcal{B}_i$ is generated. The latest number set constructed from blocks $\mathcal{B}_{[i]}$ is defined as $\mathsf{LN}_{[i]} = \{(w_j, ln_j)\}_{w_j \in \mathsf{W}_{[i]}}$.*

**Definition 2** (Sorted Object Set)**.** *Each keyword $w_j \in \mathsf{W}_{[i]}$ is associated with a set of keyword/object/sequence-number tuples $\mathsf{SO}_{[i]}.j = \{(w_j, id_k, k)\}_{k=1}^{ln_j}$, where $(w_j, id_k, k)$ means that the object with identifier $id_k$ is the k-th latest object containing keyword $w_j$ when block $\mathcal{B}_i$ is generated. The sorted object set constructed from blocks $\mathcal{B}_{[i]}$ is defined as $\mathsf{SO}_{[i]} = \bigcup_{w_j \in \mathsf{W}_{[i]}} \mathsf{SO}_{[i]}.j$.*

When a new block $\mathcal{B}_t$ is appended, the full node sorts the relevant objects by the ascending order of their timestamps for each keyword $w_j \in \mathsf{W}_{[t]}$, so that the fresher object will be assigned with a higher sequence number, and the highest sequence number equals $ln_j$, the latest number of objects containing keyword $w_j$. Therefore, the latest-$K$ objects containing keyword $w_j$ can be denoted by $\{(w_j, id_k, k)\}_{k=ln_j-K+1}^{ln_j}$. To speed up the construction of sorted object sets, the inverted index [24] that records the identifiers and locations of relevant objects for each keyword is adopted. With the inverted index, the complexity of searching keyword $w_j$ is $O(ln_j)$ which is not only sublinear, but also optimal. Compared to the block size, the inverted index consumes a relatively smaller space, and thus can be locally kept by full nodes to quickly locate all the relevant objects instead of traversing the whole blockchain.

The details of the strawman construction are shown in Alg. 1. To stress the main points, the verification of object authenticity is omitted, since it can be easily verified through MerkleRoot in the block header. Our main idea is to let the ADS newly generated summarize the sorted objects and the latest object numbers for all the keywords. Specifically, $\mathcal{B}_t.\mathcal{ADS}$ consists of two accumulative values: $\Phi_I = \mathbf{acc}(\mathsf{SO}_{[t]})$ and $\Phi_F = \mathbf{acc}(\mathsf{LN}_{[t]})$. Given the query $\mathcal{Q} = (\mathcal{T} = w_s, K)$, the SP searches the inverted index to locate all the objects containing keyword $w_s$, and puts the latest $K$ objects into the search results $\mathcal{SR}$. To construct the VO, the SP puts the keyword/latest-number pair $(w_s, ln_s)$ and the latest $K$ keyword/object/sequence-number tuples $\{(w_s, id_k, k)\}_{k=ln_s-K+1}^{ln_s}$ into set $\mathrm{SR}_F$ and set $\mathrm{SR}_I$, respectively, while generating the corresponding witnesses as $\pi_F = \mathbf{acc}(\mathsf{LN}_{[t]} - \mathrm{SR}_F)$ and $\pi_I = \mathbf{acc}(\mathsf{SO}_{[t]} - \mathrm{SR}_I)$.

**Algorithm 1** Strawman Solution in veffChain

ADS Generation (by the miner)
**Input:** Blockchain $\mathcal{B}_{[t]}$
**Output:** The ADS of the new block $\mathcal{B}_t.\mathcal{ADS}$
Construct a latest number set $\mathsf{LN}_{[t]}$ according to Def. 1
Construct a sorted object set $\mathsf{SO}_{[t]}$ according to Def. 2
$\Phi_I \leftarrow \mathsf{GenAcc}(\mathsf{SO}_{[t]})$; $\Phi_F \leftarrow \mathsf{GenAcc}(\mathsf{LN}_{[t]})$
$\mathcal{B}_t.\mathcal{ADS} \leftarrow (\Phi_I, \Phi_F)$

VO Construction (by the SP)
**Input:** Query $\mathcal{Q} = (\mathcal{T} = w_s, K)$, blockchain $\mathcal{B}_{[t]}$
**Output:** Search result $\mathcal{Q}.\mathcal{SR}$, the VO of query $\mathcal{Q}.\mathcal{VO}$
Construct a latest number set $\mathsf{LN}_{[t]}$ according to Def. 1
Construct a sorted object set $\mathsf{SO}_{[t]}$ according to Def. 2
$\mathrm{SR}_F \leftarrow (w_s, ln_s)$; $\pi_F \leftarrow \mathsf{GenWit}(\mathrm{SR}_F, \mathsf{LN}_{[t]})$
$\mathrm{SR}_I \leftarrow \{(w_s, id_k, k)\}_{k=ln_s-K+1}^{ln_s}$; $\pi_I \leftarrow \mathsf{GenWit}(\mathrm{SR}_I, \mathsf{SO}_{[t]})$
$\mathcal{Q}.\mathcal{SR} \leftarrow$ Objects with identifiers in $\mathrm{SR}_I$
$\mathcal{Q}.\mathcal{VO} \leftarrow (\mathrm{SR}_I, \mathrm{SR}_F, \pi_I, \pi_F)$

Verification (by the user)
**Input:** The VO of query $\mathcal{Q}.\mathcal{VO}$, the latest ADS $\mathcal{B}_t.\mathcal{ADS}$
**Output:** Verification report $\mathcal{Q}.\mathcal{VR}$
Parse $\mathcal{B}_t.\mathcal{ADS}$ as $(\Phi_I, \Phi_F)$; $\mathcal{Q}.\mathcal{VR} \leftarrow 0$
**if** $\mathsf{VeriWit}(\mathrm{SR}_I, \pi_I, \Phi_I) \wedge \mathsf{VeriWit}(\mathrm{SR}_F, \pi_F, \Phi_F)$ **then**
    $\mathcal{Q}.\mathcal{VR} \leftarrow 1$ ▷ *1 indicates verification passes*

Once receiving $\mathcal{Q}.\mathcal{VO} = (\mathrm{SR}_I, \mathrm{SR}_F, \pi_I, \pi_F)$, the user checks if the VO meets the following requirement or not: (1) There are $K$ tuples in set $\mathrm{SR}_I$ and their sequence numbers are consecutive. (2) The highest sequence number in set $\mathrm{SR}_I$ equals $ln_s$, the latest object number in set $\mathrm{SR}_F$. If so, the user runs algorithm VeriWit to verify search results. Note that $\mathsf{VeriWit}(\mathrm{SR}_I, \pi_I, \mathcal{B}_t.\Phi_I)$ outputs 1 only when $\mathrm{SR}_I \subseteq \mathsf{SO}_{[t]}$ due to the security of RSA accumulator. This means that the objects in set $\mathrm{SR}_I$ indeed contain keyword $w_s$, validating result integrity. Similarly, $\mathsf{VeriWit}(\mathrm{SR}_F, \pi_F, \mathcal{B}_t.\Phi_F)$ outputs 1 only when $\mathrm{SR}_F \subseteq \mathsf{LN}_{[t]}$. This means that set $\mathrm{SR}_F$ contains the latest object number of keyword $w_s$ and set $\mathrm{SR}_I$ contains the identifiers of latest-$K$ objects, validating result freshness.

### 4.2 The Verifiable Solution for Latest-$K$ Exact Queries

When a new block is appended into the system, the strawman solution calculates the accumulative values of all sets from the beginning, resulting in performance degradation over time. For improved scalability, we utilize the dynamic property of RSA accumulator, enabling the accumulative value of a large set to be rapidly calculated from that of its subset with only the public key, i.e., when $Y \subset X$, we have $\mathbf{acc}(X) = \mathbf{acc}(Y)^{\prod_{x \in X-Y} \mathcal{P}(\mathcal{H}(x))} \mod \mathbf{N}$. Given a sequence of blocks $\mathcal{B}_{[i]}$ for $i \in [t]$, the basic solution works under the following assumption and definitions:

**Assumption 1.** *The timestamps of all objects in a new block are larger than those of the objects in the previous block.*

**Definition 3** (Update Number Set)**.** *Each keyword $w_j \in \mathsf{W}_i$ is associated with a keyword/update-number pair $(w_j, un_j)$, which means that the number of objects containing keyword $w_j$ is updated to $un_j$ when block $\mathcal{B}_i$ is generated. The update number set constructed from block $\mathcal{B}_i$ and blocks $\mathcal{B}_{[i]}$ are defined as $\mathsf{UN}_i = \{(w_j, un_j)\}_{w_j \in \mathsf{W}_i}$ and $\mathsf{UN}_{[i]} = \bigcup_{k=1}^{i} \mathsf{UN}_k$, respectively.*

**Definition 4** (History Number Set)**.** *Each keyword $w_j \in \mathsf{W}_i$ is associated with a keyword/history-number pair $(w_j, hn_j)$, which means that the number of objects containing keyword $w_j$ is $hn_j$ before the generation of block $\mathcal{B}_i$. The history number*

**Algorithm 2** Basic Solution in veffChain

ADS Generation (by the miner)
**Input:** Blockchain $\mathcal{B}_{[t]}$
**Output:** The ADS of the new block $\mathcal{B}_t.\mathcal{ADS}$
$(\Phi'_I, \Phi'_U, \Phi'_H) \leftarrow \mathcal{B}_{t-1}.\mathcal{ADS}$
Construct sorted object sets $\mathsf{SO}_{[t]}$ and $\mathsf{SO}_{[t-1]}$ using Def. 2
Construct an update number set $\mathsf{UN}_t$ according to Def. 3
Construct a history number set $\mathsf{HN}_t$ according to Def. 4
$\Phi_I \leftarrow (\Phi'_I)^{\prod_{x\in\mathsf{SO}_{[t]}-\mathsf{SO}_{[t-1]}} \mathcal{P}(\mathcal{H}(x))}$
$\Phi_U \leftarrow (\Phi'_U)^{\prod_{x\in\mathsf{UN}_t} \mathcal{P}(\mathcal{H}(x))}$; $\Phi_H \leftarrow (\Phi'_H)^{\prod_{x\in\mathsf{HN}_t} \mathcal{P}(\mathcal{H}(x))}$
$\mathcal{B}_t.\mathcal{ADS} \leftarrow (\Phi_I, \Phi_U, \Phi_H)$

VO Construction (by the SP)
$\mathcal{Q}.\mathcal{SR}$ and $\mathcal{Q}.\mathcal{VO} = \{\mathrm{SR}_I, \mathrm{SR}_F, \pi_I, \pi_F\}$ are constructed in the same way as the strawman solution, except that:
$\pi_F \leftarrow \prod_{x\in(\mathsf{LN}_{[t]}-\mathrm{SR}_F)} \mathcal{P}(\mathcal{H}(x))$

Verification (by the user)
**Input:** The VO of query $\mathcal{Q}.\mathcal{VO}$, the latest ADS $\mathcal{B}_t.\mathcal{ADS}$
**Output:** Verification report $\mathcal{Q}.\mathcal{VR}$
Parse $\mathcal{B}_t.\mathcal{ADS}$ as $(\Phi_I, \Phi_U, \Phi_H)$; $\mathcal{Q}.\mathcal{VR} \leftarrow 0$
**if** $\mathsf{VeriWit}(\mathrm{SR}_I, \pi_I, \Phi_I) \wedge (\Phi_U = \Phi_H^{\pi_F\cdot\prod_{x\in\mathrm{SR}_F} \mathcal{P}(\mathcal{H}(x))})$ **then**
  $\mathcal{Q}.\mathcal{VR} \leftarrow 1$ ▷*1 indicates verification passes*

*set constructed from block $\mathcal{B}_i$ and blocks $\mathcal{B}_{[i]}$ are defined as $\mathsf{HN}_i = \{(w_j, hn_j)\}_{w_j\in\mathsf{W}_i}$ and $\mathsf{HN}_{[i]} = \bigcup_{k=1}^{i} \mathsf{HN}_k$, respectively. In the special case, we have $\mathsf{HN}_1 = \mathsf{HN}_{[1]} = \emptyset$.*

A blockchain is an append-only data structure, and thus we have $\mathsf{SO}_{[i-1]} \subset \mathsf{SO}_{[i]}$ under Assumption 1, enabling $\mathcal{B}_i.\Phi_I$ to be incrementally updated from $\mathcal{B}_{i-1}.\Phi_I$. Owing to $\mathsf{LN}_{[i-1]} \not\subset \mathsf{LN}_{[i]}$, we additional define an update number set $\mathsf{UN}_i$ and a history number set $\mathsf{HN}_i$ for each block $\mathcal{B}_i$, and replace $\mathcal{B}_i.\Phi_F$ with $(\mathcal{B}_i.\Phi_U, \mathcal{B}_i.\Phi_H)$, which are the accumulative values of union sets $\mathsf{UN}_{[i]}$ and $\mathsf{HN}_{[i]}$, respectively. Because of $\mathsf{UN}_{[i-1]} \subset \mathsf{UN}_{[i]}$ and $\mathsf{HN}_{[i-1]} \subset \mathsf{HN}_{[i]}$, both $\mathcal{B}_i.\Phi_U$ and $\mathcal{B}_i.\Phi_H$ can be incrementally updated. Alg. 2 shows the main differences from the strawman solution. In subsequent contents, the $\bmod \mathbf{N}$ operation is omitted for simplicity.

**ADS Generation.** The ADS in block $\mathcal{B}_t$ is replaced by $(\Phi_I, \Phi_U, \Phi_H)$, all of which can be dynamically updated from previous accumulative values $(\Phi'_I, \Phi'_U, \Phi'_H)$. If keyword $w_j$ is updated in block $\mathcal{B}_i$, its latest object number will be put into set $\mathsf{UN}_i$, i.e., $ln_j = un_j$ for $w_j \in \mathsf{W}_i$, and its previous object number will be put into set $\mathsf{HN}_i$. That is, for each keyword in $\mathsf{W}_{[i]}$, $\Phi_U$ summarizes the full update history of object number, and $\Phi_H$ summarizes the full update history except for the last update (i.e., excluding the latest object number). Hence, we have $\mathsf{UN}_{[i]} = \mathsf{HN}_{[i]} \bigcup \mathsf{LN}_{[i]}$ and $\Phi_U = \Phi_H^{\prod_{x\in\mathsf{LN}_{[i]}} \mathcal{P}(\mathcal{H}(x))}$. In the special case of $\mathsf{HN}_{[1]} = \mathsf{HN}_1 = \emptyset$, we set $\mathcal{B}_1.\Phi_H$ to $g$.

**VO Construction.** The VO construction algorithm is similar to that of the strawman solution, except that the witness $\pi_F$ is replaced by the exponential value of $\mathbf{acc}(\mathsf{LN}_{[t]} - \mathrm{SR}_F)$. Besides, the SP locally keeps $\varphi_j = \prod_{x\in\mathsf{SO}_{[t]}.j} \mathcal{P}(\mathcal{H}(x))$, the exponential value of $\mathbf{acc}(\mathsf{SO}_{[t]}.j)$ for each keyword $w_j \in \mathsf{W}_{[t]}$ to accelerate the calculation of witness $\pi_I$. Given $\{\varphi_j\}_{w_j\in\mathsf{W}_{[t]}}$, the witness $\pi_I$ can be rapidly calculated as:

$$g^{\prod_{w_j\in\mathsf{W}_{[t]}, j\neq s} \varphi_j \cdot \prod_{x\in\mathsf{SO}[t].s-\mathrm{SR}_I} \mathcal{P}(\mathcal{H}(x))} = g^{\prod_{x\in\mathsf{SO}_{[t]}-\mathrm{SR}_I} \mathcal{P}(\mathcal{H}(x))}.$$

Under Assumption 1, $\varphi_j$ can be incrementally updated as:

$$\varphi_j = \varphi'_j \cdot \prod_{x\in\mathsf{SO}_{[t]}.j-\mathsf{SO}_{[t-1]}.j} \mathcal{P}(\mathcal{H}(x)).$$

where $\varphi'_j = \prod_{x\in\mathsf{SO}_{[t-1]}.j} \mathcal{P}(\mathcal{H}(x))$ is the previous value.

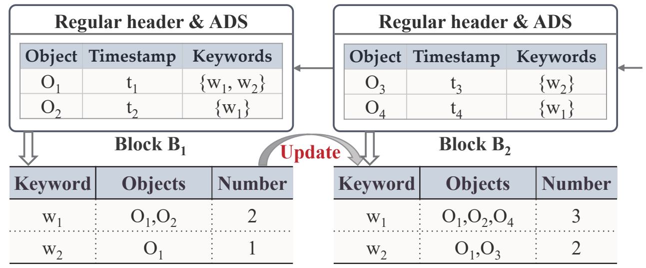



Fig. 2: Illustrative example for veffChain. The timestamp of object $o_i$ is assumed to be smaller than that of object $o_{i+1}$.

**Verification.** If the VO meets the requirements described in the strawman solution, the user verifies result integrity as before and tests if $\Phi_U$ equals $\Phi_H^{\pi_F\cdot\prod_{x\in\mathrm{SR}_F} \mathcal{P}(\mathcal{H}(x))}$ or not for freshness validation. Note that $\Phi_U = \Phi_H^{\prod_{x\in\mathsf{LN}_{[t]}} \mathcal{P}(\mathcal{H}(x))}$ and $\pi_F = \prod_{x\in\mathsf{LN}_{[t]}-\mathrm{SR}_F} \mathcal{P}(\mathcal{H}(x))$. The equation is satisfied only when $\mathrm{SR}_F \subseteq \mathsf{LN}_{[t]}$. This means that set $\mathrm{SR}_F$ contains the latest object number and set $\mathrm{SR}_I$ contains the latest-$K$ results regarding the search term, validating result freshness.

### 4.3 Illustrative Examples

Given a sequence of blocks $\mathcal{B}_{[2]}$, the collection of data objects packed in each block are shown in Fig. 2. Suppose that the user issues an exact query $\mathcal{Q} = (w_1, 2)$ to retrieve the latest 2 objects containing keyword $w_1$. When block $\mathcal{B}_1$ is generated, we have $\mathsf{LN}_{[1]} = \{(w_1, 2), (w_2, 1)\}$, $\mathsf{SO}_{[1]} = \{(w_1, o_1, 1), (w_1, o_2, 2), (w_2, o_1, 1)\}$, $\mathsf{UN}_1 = \mathsf{LN}_{[1]}$ and $\mathsf{HN}_1 = \emptyset$; When block $\mathcal{B}_2$ is generated, we have $\mathsf{LN}_{[2]} = \{(w_1, 3), (w_2, 2)\}$, $\mathsf{SO}_{[2]} = \mathsf{SO}_{[1]} \cup \{(w_1, o_4, 3), (w_2, o_3, 2)\}$, $\mathsf{UN}_2 = \{(w_1, 3), (w_2, 2)\}$ and $\mathsf{HN}_2 = \{(w_1, 2), (w_2, 1)\}$. Therefore, we have $\mathsf{UN}_{[i]} = \mathsf{HN}_{[i]} \cup \mathsf{LN}_{[i]}$ for $i \in [2]$.

**Strawman Solution.** For $i \in [2]$, the miner sets the ADS in the block header as $\mathcal{B}_i.\mathcal{ADS} = (\mathcal{B}_i.\Phi_I, \mathcal{B}_i.\Phi_F)$, where $\mathcal{B}_i.\Phi_I = \mathbf{acc}(\mathsf{SO}_{[i]})$ and $\mathcal{B}_i.\Phi_F = \mathbf{acc}(\mathsf{LN}_{[i]})$. In VO construction, the SP constructs $\mathrm{SR}_I = \{(w_1, o_2, 2), (w_1, o_4, 3)\}$ and $\mathrm{SR}_F = \{(w_1, 3)\}$, and calculates $\pi_I = \mathbf{acc}(\mathsf{SO}_{[2]} - \mathrm{SR}_I)$ and $\pi_F = \mathbf{acc}(\mathsf{LN}_{[2]} - \mathrm{SR}_F)$, so that the user verifies search results by testing the following equations:

$$\pi_I^{\prod_{x\in\mathrm{SR}_I} \mathcal{P}(\mathcal{H}(x))} \stackrel{?}{=} \mathcal{B}_2.\Phi_I, \pi_F^{\prod_{x\in\mathrm{SR}_F} \mathcal{P}(\mathcal{H}(x))} \stackrel{?}{=} \mathcal{B}_2.\Phi_F.$$

The above equations are satisfied only when $\mathrm{SR}_I \subseteq \mathsf{SO}_{[2]}$ and $\mathrm{SR}_F \subseteq \mathsf{LN}_{[2]}$. This means that the latest object number is 3 and the latest-2 objects are $\{o_2, o_4\}$ for keyword $w_1$.

**Basic Solution.** For $i \in [2]$, the ADS in the block header is in the form of $\mathcal{B}_i.\mathcal{ADS} = (\mathcal{B}_i.\Phi_I, \mathcal{B}_i.\Phi_U, \mathcal{B}_i.\Phi_H)$. In the first place, the miner calculates $\mathcal{B}_1.\Phi_I \leftarrow \mathsf{GenAcc}(\mathsf{SO}_{[1]})$, $\mathcal{B}_1.\Phi_U \leftarrow \mathsf{GenAcc}(\mathsf{UN}_{[1]})$, and $\mathcal{B}_1.\Phi_H \leftarrow g$. Next, the ADS can be dynamically generated as follows:

$$\mathcal{B}_2.\Phi_I \leftarrow (\mathcal{B}_1.\Phi_I)^{\prod_{x\in\mathsf{SO}_{[2]}-\mathsf{SO}_{[1]}} \mathcal{P}(\mathcal{H}(x))} = \mathbf{acc}(\mathsf{SO}_{[2]}),$$
$$\mathcal{B}_2.\Phi_U \leftarrow (\mathcal{B}_1.\Phi_U)^{\prod_{x\in\mathsf{UN}_2} \mathcal{P}(\mathcal{H}(x))} = \mathbf{acc}(\mathsf{UN}_{[2]}),$$
$$\mathcal{B}_2.\Phi_H \leftarrow (\mathcal{B}_1.\Phi_H)^{\prod_{x\in\mathsf{HN}_2} \mathcal{P}(\mathcal{H}(x))} = \mathbf{acc}(\mathsf{HN}_{[2]}).$$

In VO construction, the SP constructs sets $\mathrm{SR}_I$ and $\mathrm{SR}_F$, and witness $\pi_I$ as the strawman solution, but calculates witness $\pi_F$ as $\prod_{x\in\mathsf{LN}_{[2]}-\mathrm{SR}_F} \mathcal{P}(\mathcal{H}(x))$. In verification, the users verifies result integrity as before, and tests the following equation for freshness authentication:

$$\mathcal{B}_2.\Phi_U \stackrel{?}{=} (\mathcal{B}_2.\Phi_H)^{\pi_F\cdot\prod_{x\in\mathrm{SR}_F} \mathcal{P}(\mathcal{H}(x))}.$$

This article has been accepted for publication in IEEE Transactions on Knowledge and Data Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2023.3316127



The above equation is satisfied only when $\mathrm{SR}_F \subseteq \mathsf{LN}_{[2]}$. This would imply that, for keyword $w_1$, the latest object number is 3 and the latest-2 objects are $\{o_2, o_4\}$.

# 5 THE veffCHAIN++ FRAMEWORK

## 5.1 Search Terms, Distance, and Similarity

The veffChain framework allows a user to retrieve the freshest objects in a verifiable way, but supports only latest-$K$ exact queries. To improve user query experience, veffChain++ classifies search terms into exact terms excluding any wildcards and fuzzy terms containing wildcards. An exact term is a string of characters chosen from the English alphabet $\mathbb{A}$, and a fuzzy term contains two types of wildcards: $'?'$ denoting an arbitrary character in $\mathbb{A}$, and $'*'$ denoting zero, one, or multiple arbitrary characters in $\mathbb{A}$. For example, a user can enter either "$secur???$" or "$secur*$" to retrieve objects containing the keyword "$security$".

Given a string $\mathcal{S}_1$ excluding any wildcards, and a string $\mathcal{S}_2$ that may contain wildcards $'?'$, their distance, denoted by $\Delta(\mathcal{S}_1, \mathcal{S}_2)$, is calculated according to Def. 5 (which also can be used to quantify the distance between two exact strings). As for the distance between string $\mathcal{S}_1$ and a string $\mathcal{S}_3$ that contains a wildcard $'*'$, we first obtain a transformed string $\mathcal{S}_3'$ by replacing wildcard $'*'$ with $\max\{0, |\mathcal{S}_1| - |\mathcal{S}_3| + 1\}$ wildcards $'?'$, and then set $\Delta(\mathcal{S}_1, \mathcal{S}_3)$ to $\Delta(\mathcal{S}_1, \mathcal{S}_3')$.

**Definition 5** (Distance). *Let $e_1$ be the number of operations required to transform keyword $\mathcal{S}_1$ to $\mathcal{S}_2$, and let $e_2$ be the number of wildcard $'?'$s in $\mathcal{S}_2$. We have $\Delta(\mathcal{S}_1, \mathcal{S}_2) = |e_1 - e_2|$.*

Based on the above definition, a keyword $w$ is regarded as similar to a fuzzy term $\widetilde{\mathcal{T}}$, denoted by $w \approx \widetilde{\mathcal{T}}$, if $\Delta(w, \widetilde{\mathcal{T}}) = 0$. For example, $\Delta(\text{"salt"}, \text{"sa??"}) = 0$ and $\Delta(\text{"salt"}, \text{"se??"}) = 1$. Therefore, we have "$salt$" $\approx$ "$sa??$" and "$salt$" $\not\approx$ "'$se??$". As for fuzzy terms "sa*" and "se*", we first replace the wildcard $'*'$ with two wildcards $'?'$ and obtain $\Delta(\text{"salt"}, \text{"sa*"}) = \Delta(\text{"salt"}, \text{"sa??"}) = 0$ and $\Delta(\text{"salt"}, \text{"se*"}) = \Delta(\text{"salt"}, \text{"se??"}) = 1$. Hence, we have "$salt$" $\approx$ "$sa*$" and "$salt$" $\not\approx$ "'$se*$".

## 5.2 The VTrie Tree

Given a sequence of blocks $\mathcal{B}_{[i]}$, we first build a Trie tree for all keywords in $\mathsf{W}_{[i]}$ as described in Section 3.3. A VTrie tree $\mathcal{VT}$ is constructed from the bottom up by integrating the verification information into Trie tree nodes. As shown in Fig. 3, a leaf node $\mathcal{N}_u \in \mathcal{VT}$ corresponding to a distinct keyword $w_j \in \mathsf{W}_{[i]}$ is defined as follows:

$$\mathcal{N}_u = (\mathcal{C}_u, \mathcal{S}_u, a_u, n_u), \tag{1}$$

where $\mathcal{C}_u =' \#'$ is the character in node $\mathcal{N}_u$ that denotes the end of traverse, $\mathcal{S}_u$ is a string of characters in the path from the root node to its parent node satisfying $\mathcal{S}_u || \mathcal{C}_u = w_j$, $a_u = \mathbf{acc}(\mathsf{SO}_{[i]}.j)$ is the accumulative value of the sorted object subset of keyword $w_j$, and $n_u = ln_j$ is the latest object number of keyword $w_j$. A non-leaf node $\mathcal{N}_v \in \mathcal{VT}$ with $c$ children nodes $\mathcal{N}_{v_1}, \ldots, \mathcal{N}_{v_c}$ is defined as follows:

$$\mathcal{N}_v = (\mathcal{C}_v, \mathcal{S}_v, h_v), \tag{2}$$

where $\mathcal{C}_v$ is the character contained in node $\mathcal{N}_v$, $\mathcal{S}_v$ is a string of characters in the path from the root to its parent (if $v$ is the root node, $\mathcal{C}_v =' \$'$ denoting the start of traverse and $\mathcal{S}_v = \perp$), and $h_v = \mathcal{H}(\mathcal{H}(\mathcal{N}_{v_1}) || \ldots || \mathcal{H}(\mathcal{N}_{v_c}))$ denotes the digest of children nodes' hashes. For the uniqueness of VTrie tree structure, the children nodes are sorted by the lexicographic order of the contained characters.

**Algorithm 3** Match

**Input:** A Trie tree node $\mathcal{N}_x = (\mathcal{C}_x, \mathcal{S}_x, *)$, a search term $\mathcal{ST}$
**Output:** Matching result $\mathcal{MR}$

```
MR ← 0                                          ▷ 0 indicates mismatching
if ST does not contain wildcard '*' and |ST| > |S_x| then
    if N_x is a non-leaf node then
        l ← |S_x| + 1; MR ← 1 − min{1, Δ(S_x||C_x, ST[l])}
    if N_x is a leaf node then
        MR ← 1 − min{1, Δ(S_x||C_x, ST)}
if ST contains wildcard '*' at position L then
    if N_x is a non-leaf node then
        l ← min{L, |S_x| + 1}; ST ← ST[l]
    k ← max{0, |S_x| − |ST| + 2}
    Replace wildcard '*' in ST with k wildcards '?'
    MR ← 1 − min{1, Δ(S_x||C_x, ST)}
```

The search process is a recursive procedure upon the VTrie tree. Given a search term, the SP performs a detection starting from the root node: If a non-leaf node matches the search term, the SP checks all its children nodes; otherwise, the SP stops traversing the subtree rooted at this node. When the traversal reaches a leaf node, the corresponding keyword is considered equal/similar if this node matches the search term. Alg. 3 shows the matching process between a Vtrie tree node and a search term. As searching exact terms is actually a special case of searching a fuzzy term, we focus on verifiable latest-$K$ fuzzy queries in following sections.

## 5.3 The Verifiable Solution for Latest-$K$ Fuzzy Queries

Alg. 4 shows the details of the basic solution that works under Assumption 1. Our main idea is constructing a VTrie tree from the keywords updated so far and uses the root hash as the ADS embedded in the new block header, so that the user can further verify result completeness by validating the VTrie tree reconstructed from the VO. Besides, the full node may locally keep the inverted index as the veffChain framework to speed up the construction of sorted object sets.

**ADS Generation.** As shown in Fig. 3, the ADS embedded in a block header is composed of the root hash $\mathcal{H}(\mathcal{VT}.root)$ of a VTrie tree. Upon arrival of a new block $\mathcal{B}_t$, the miner updates the VTrie tree $\mathcal{VT}$ in the following way: For each keyword $w_j \in \mathsf{W}_t$, it searches the VTrie tree to find corresponding leaf node $\mathcal{N}_u$, s.t. $\mathcal{S}_u || \mathcal{C}_u = w_j$, updates $a_u$ to $\mathbf{acc}(\mathsf{SO}_{[t]}.j)$, and $n_u$ to the latest object number of keyword $w_j$, i.e, $n_u = |\mathsf{SO}_{[t]}.j|$. If there is no matched leaf node, this means that keyword $w_j$ appears for the first time. The miner constructs a new leaf node corresponding to keyword $w_j$ with Eq. 1. and updates the VTrie tree to incorporate this new node. After updating the leaf nodes, the miner re-constructs all the relevant ancestor nodes until reaching the root by using Eq. 2. Under Assumption 1, we have $\mathsf{SO}_{[t-1]}.j \subseteq \mathsf{SO}_{[t]}.j$, and thus $\mathbf{acc}(\mathsf{SO}_{[t]}.j)$ can be calculated by $\mathbf{acc}(\mathsf{SO}_{[t-1]}.j)^{\prod_{x \in \mathsf{SO}_{[t]}.j - \mathsf{SO}_{[t-1]}.j} \mathcal{P}(\mathcal{H}(x))}$, i.e., $a_u$ can be incrementally updated from its previous value.

**VO Construction.** Given a fuzzy query $\widetilde{Q} = (\widetilde{\mathcal{T}}, K)$, the SP traverses the VTrie tree $\mathcal{VT}$ from the top to bottom by running Alg. 5, and outputs a set of matched nodes MN and a set of unmatched nodes UN. Note that each leaf node in set MN corresponds to a keyword similar

This article has been accepted for publication in IEEE Transactions on Knowledge and Data Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2023.3316127



**Algorithm 4** Basic Solution in veffChain++

ADS Generation (by the miner)

**Input:** Blockchain $\mathcal{B}_{[t]}$, a VTrie tree $\mathcal{VT}'$
**Output:** The latest ADS $\mathcal{B}_t.\mathcal{ADS}$, an updated VTrie tree $\mathcal{VT}$

```
for each keyword w_j ∈ W_t do
    Construct a sorted object subset SO_[t].j using Def. 2
    if ∃ a leaf node N_u ∈ VT' s.t. S_u||C_u = w_j then
        Construct a sorted object subset SO_[t-1].j using Def. 2
        a_u ← (a_u)^{∏_{x∈SO_[t].j−SO_[t-1].j} P(H(x))}; n_u ← |SO_[t].j|
    else
        Construct a new leaf node N_u for keyword w_j with Eq. 1 and update tree VT' to incorporate node N_u
Update the ancestor nodes from the bottom up with Eq. 2 to form VT; B_t.ADS ← H(VT.root)
```

VO Construction (by the SP)

**Input:** Query $\widetilde{\mathcal{Q}} = (\widetilde{\mathcal{T}}, K)$, blockchain $\mathcal{B}_{[t]}$, a VTrie tree $\mathcal{VT}$
**Output:** Search result $\widetilde{\mathcal{Q}}.\mathcal{SR}$, the VO of query $\widetilde{\mathcal{Q}}.\mathcal{VO}$

```
Run Search(VT.root, Q~) to generate sets MN and UN
for each leaf node N_u ∈ MN do
    Locate w_j ∈ W_[t] s.t. S_u||C_u = w_j
    Construct a sorted object subset SO_[t].j using Def. 2
    SR_j ← {(w_j, id_k, k)}_{k=ln_j−K+1}^{ln_j}; π_j ← GenWit(SR_j, SO_[t].j)
    Put objects with identifiers in SR_j into Q~.SR
Q~.VO ← {MN, UN, {(SR_j, π_j)}_{w_j≈T~}}
```

Verification (by the user)

**Input:** The VO of query $\widetilde{\mathcal{Q}}.\mathcal{VO}$, the latest ADS $\mathcal{B}_t.\mathcal{ADS}$
**Output:** Verification report $\mathcal{Q}.\mathcal{VR}$

```
Q.VR ← 0                                    ▷ 0 indicates verification fails
Reconstruct the VTrie tree VT' from set MN∪UN with Eq. 2
if H(VT'.root) = B_t.ADS then
    for each leaf node N_u ∈ MN do
        w_j ← S_u||C_u
        Locate SR_j s.t. the keyword in SR_j equals w_j
        ln'_j ← the highest sequence number in SR_j
        if VeriWit(SR_j, π_j, a_u) = 0 ∨ ln'_j ≠ n_u then
            return Q.VR
    Q.VR ← 1
```

**Algorithm 5** Search

**Input:** A VTrie tree $\mathcal{VT}.root$, a fuzzy query $\widetilde{\mathcal{Q}} = (\widetilde{\mathcal{T}}, K)$
**Output:** Matched nodes MN, unmatched nodes UN

```
Q ← empty queue; (MN, UN) ← empty set
Push VT.root into queue Q
while Q is non-empty do
    N_x ← the head of queue Q
    if N_x is a non-leaf node then
        if Match(N_x, T~) then
            Push all the children nodes of N_x into queue Q
        else
            Put N_x into UN
    else
        if Match(N_x, T~) then
            Put N_x into MN
        else
            Put N_x into UN
```

to fuzzy search term $\widetilde{\mathcal{T}}$, and the total number of similar keywords is $|\mathrm{MN}|$. For each similar keyword $w_j$, the SP puts the latest $K$ keyword/object/sequence-number tuples $\{(w_j, id_k, k)\}_{k=ln_j-K+1}^{ln_j}$ into set $\mathrm{SR}_j$ and generates the witness as $\pi_j = \mathbf{acc}(\mathsf{SO}_{[t]}.j - \mathrm{SR}_j)$. The VO returned is in the form of $\{\mathrm{MN}, \mathrm{UN}, \{(\mathrm{SR}_j, \pi_j)\}_{w_j \approx \widetilde{\mathcal{T}}}\}$.

**Verification.** The user first examines if $\widetilde{\mathcal{Q}}.\mathcal{VO}$ meets the following requirements or not: (1) For each similar keyword $w_j$, there are $K$ tuples in set $\mathrm{SR}_j$ and their sequence numbers are consecutive; (2) For each similar keyword $w_j$, there exists a leaf node $\mathcal{N}_u \in \mathrm{MN}$ s.t. $\mathcal{S}_u||\mathcal{C}_u = w_j$; (3) All nodes in set UN (resp. set MN) indeed mismatch (resp. match) the fuzzy term. If so, the user reconstructs the VTrie tree $\mathcal{VT}'$ with nodes in set $\mathrm{MN} \cup \mathrm{UN}$, and tests if $\mathcal{H}(\mathcal{VT}'.root) = \mathcal{B}_t.\mathcal{ADS}$ or not. If it is true, this means that all nodes in set $\mathrm{MN}\cup\mathrm{UN}$ are authentic. That is, all the similar keywords are found and result completeness is confirmed. Next, for the similar keyword $w_j \in \mathrm{SR}_j$ that corresponds to leaf node $\mathcal{N}_u = (\mathcal{C}_u, \mathcal{S}_u, a_u, n_u)$ in set MN, the user verifies result integrity by running $\mathsf{VeriWit}(\mathrm{SR}_j, \pi_j, a_u)$ and verifies result freshness by testing if the highest sequence number $ln'_j$ in set $\mathrm{SR}_j$ equals $n_u$ or not. It is worth noticing that $a_u = \mathbf{acc}(\mathsf{SO}_{[t]}.j)$ and $n_u = ln_j$, which can be authenticated through the root hash. Therefore, algorithm VeriWit outputting 1 means that $\mathrm{SR}_j \subseteq \mathsf{SO}_{[t]}.j$, and $ln'_j = n_u$ means that $ln'_j$ is the latest object number of keyword $w_j$.

**Illustrative Example.** Given a serial of blocks $\mathcal{B}_{[2]}$, the set of objects packed in each block and the updated process of the VTrie tree are as shown in Fig. 3. When block $\mathcal{B}_1$ is generated, we have $\mathsf{SO}_{[1]}.1 = \{(\text{“}big\text{”}, o_1, 1), (\text{“}big\text{”}, o_2, 2)\}$ and $\mathsf{SO}_{[1]}.2 = \{(\text{“}bit\text{”}, o_1, 1)\}$; When block $\mathcal{B}_2$ is generated, we have $\mathsf{SO}_{[2]}.1 = \mathsf{SO}_{[1]}.1 \cup \{(\text{“}big\text{”}, o_4, 3)\}$, $\mathsf{SO}_{[2]}.2 = \mathsf{SO}_{[1]}.2 \cup \{(\text{“}bit\text{”}, o_3, 2)\}$, and $\mathsf{SO}_{[2]}.3 = \{(\text{“}boy\text{”}, o_3, 1), (\text{“}boy\text{”}, o_4, 2)\}$. The ADS in each block header is set as the root hash of the relevant VTrie tree.

Given a query $\mathcal{Q} = (\text{“}bi*\text{”}, 2)$, the search process upon the VTrie tree is marked by the blue thick lines, while the unmatched nodes MN and the matched nodes UN are filled with green and red, respectively. In VO construction, the SP constructs set $\mathrm{SR}_1 = \{(\text{“}big\text{”}, o_2, 2), (\text{“}big\text{”}, o_4, 3)\}$ and witness $\pi_1 = \mathbf{acc}(\mathsf{SO}_{[2]}.1 - \mathrm{SR}_1)$ for similar keyword “$big$” while generating set $\mathrm{SR}_2 = \{(\text{“}bit\text{”}, o_1, 1), (\text{“}bit\text{”}, o_3, 2)\}$ and witness $\pi_2 = \mathbf{acc}(\mathsf{SO}_{[2]}.2 - \mathrm{SR}_2)$ for similar keyword “$bit$”. Given $\widetilde{\mathcal{Q}}.\mathcal{VO} = \{\mathrm{MN}, \mathrm{UN}, \{(\mathrm{SR_j}, \pi_j)\}_{j=1}^{2}\}$, the user reconstructs the VTrie tree with the nodes in set $\mathrm{MN} \cup \mathrm{UN}$. If the root hash equals current ADS, result completeness is verified. The user then verifies result integrity by testing if $\mathsf{VeriWit}(\mathrm{SR}_1, \pi_1, a_8)$ and $\mathsf{VeriWit}(\mathrm{SR}_2, \pi_2, a_9)$ output 1, and verifies result freshness by testing if the highest sequence number in sets $\mathrm{SR}_1$ and $\mathrm{SR}_2$ equal $n_8$ and $n_9$, respectively.

## 6 AKS: Adaptive Keyword Splitting

The basic solutions in veffChain and veffChain++ assume that the timestamps of objects in block $\mathcal{B}_i$ are larger than those of objects in block $\mathcal{B}_{i-1}$. This assumption is reasonable if the speed of block generation is fast enough that all the new objects can be packed into a new block at once. However, when the number of new objects exceeds block capacity, the miner will randomly pack a subset of objects. In this case, the packed objects may be fresher than the unpacked ones, and the assumption is no longer valid. To achieve improved scalability without any assumption, our main idea is to adaptively split a keyword into multiple branches so that Condition 1 is satisfied while constructing sorted object sets from blocks $\mathcal{B}_{[i]}$ for $i \in [t]$ using Def. 6.

**Condition 1.** *For each keyword branch, the timestamps of objects in a new block are larger than those of objects in previous blocks.*

This article has been accepted for publication in IEEE Transactions on Knowledge and Data Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2023.3316127



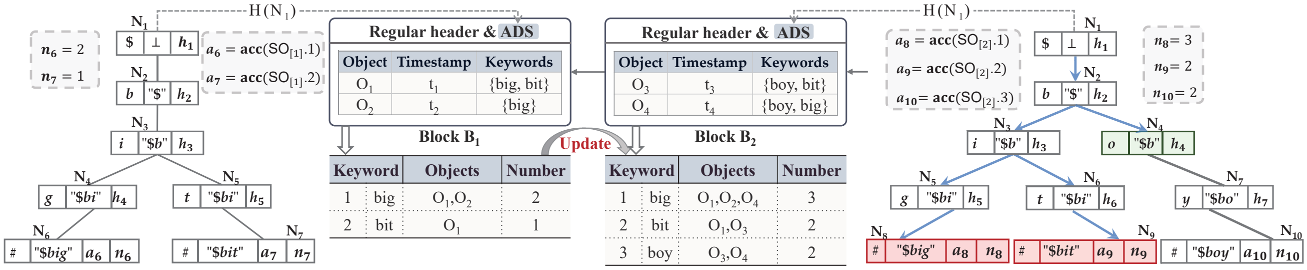



Fig. 3: Illustrative example for veffChain++. The timestamp of object $o_i$ is assumed to be smaller than that of object $o_{i+1}$.

**Definition 6** (Sorted Object Set*). *Let $b_j$ denote the branch amount of keyword $w_j$, let $w_j||v$ denote the $v$-th branch of keyword $w_j$ for $v \in [b_j]$, and let $n_{j,v}$ denote the number of objects in keyword branch $w_j||v$. Each keyword $w_j \in \mathsf{W}_{[i]}$ is associated with a set of keyword-branch/object/sequence-number tuples $\mathsf{SO}_{[i]}.j = \{(w_j||v, id_k, k)\}_{v\in[b_j],k\in[n_{j,v}]}$, where $(w_j||v, id_k, k)$ means that the object with identifier $id_k$ is the $k$-th latest object regarding keyword branch $w_j||v$. The sorted object set constructed from blocks $\mathcal{B}_{[i]}$ is denoted by $\mathsf{SO}_{[i]} = \bigcup_{w_j\in[\mathsf{W}_{[i]}]} \mathsf{SO}_{[i]}.j$.*

Specifically, the branch amount of each keyword is initialized to 1. When a new block is appended, the miner assigns the newly packed objects to appropriate keyword branches according to Condition 1. For a keyword updated in the new block, if multiple branches meet the condition, the branch with a smaller serial number is granted with the higher priority; If there is no suitable branch, the miner splits this keyword and assigns the relevant objects into the new branch. Under Condition 1, the sorted object set constructed according to Def. 6 guarantees that the object newly join in a keyword branch is assigned with a larger sequence number than existing objects. Therefore, we have $\mathsf{SO}_{[i-1]} \subset \mathsf{SO}_{[i]}$ and $\mathcal{B}_i.\Phi_I$ can be incrementally updated.

### 6.1 The Improved Solution in veffChain

Based on the AKS solution, the improved solution achieves verifiable exact-$K$ query processing as follows:

**ADS Construction.** The ADS in block $\mathcal{B}_t$ is in the form of $(\Phi_I, \Phi_U, \Phi_H)$ where $\Phi_U$ and $\Phi_H$ are calculated as the basic solution, but $\Phi_I = \mathbf{acc}(\mathsf{SO}_{[t]})$ with $\mathsf{SO}_{[t]}$ being constructed according to Def. 6. Since $\mathsf{SO}_{[t-1]} \subseteq \mathsf{SO}_{[t]}$, $\mathcal{B}_t.\Phi_I$ can be rapidly calculated as $(\mathcal{B}_{t-1}.\Phi_I)^{\prod_{x\in \mathsf{SO}_{[t]}-\mathsf{SO}_{[t-1]}} \mathcal{P}(\mathcal{H}(x))}$.

**VO Construction.** Given a query $\mathcal{Q} = (\mathcal{T} = w_s, K)$, the SP constructs $\mathcal{Q}.\mathcal{VO} = (\mathrm{SR}_F, \pi_F, \{\mathrm{SR}_v\}_{v\in[b_s]}, \pi_I)$, where $\mathrm{SR}_F$ and $\pi_F$ are calculated as the basic solution, but $\mathrm{SR}_v = \{(w_s||v, id_k, k)\}_{k=n_{s,v}-K+1}^{n_{s,v}}$ contains the latest $K$ keyword-branch/object/sequence-number tuples regarding branch $w_s||v$, and $\pi_I = \mathbf{acc}(\mathsf{SO}_{[t]} - \bigcup_{v\in[b_s]} \mathrm{SR}_v)$ is the witness of $\bigcup_{v\in[b_s]} \mathrm{SR}_v \subseteq \mathsf{SO}_{[t]}$. Note that the SP can locally keep $\varphi_j$ as the basic solution to accelerate the computation of witness $\pi_I$, where $\varphi_j$ can be incrementally calculated from the previous value under Condition 1.

**Verification.** The user first validates the VO by testing: (1) For $v \in [b_s]$, there are $K$ tuples in set $\mathrm{SR}_v$ and their sequence numbers are consecutive. (2) $\sum_{v\in[b_s]} n_{s,v} = ln_s$, where $n_{s,v}$ is the highest sequence number in set $\mathrm{SR}_v$ and $ln_s$ is the latest object number in set $\mathrm{SR}_F$. If so, the user checks if $\mathsf{VeriWit}(\bigcup_{v\in[b_s]} \mathrm{SR}_v, \pi_I, \Phi_I)$ outputs 1 for integrity validation, and verifies result freshness as the basic solution.

### 6.2 The Improved Solution in veffChain++

The main differences from the basic solution are as follows:

**ADS Generation.** Upon the arrival of a new block $\mathcal{B}_t$, the miner updates the VTrie tree as before, except that it sets $a_u$ in leaf node $\mathcal{N}_u$ to the accumulative value of keyword-branch/object/sequence-number tuples of associated keyword. Specifically, for each keyword $w_j \in \mathsf{W}_t$, the miner adaptively splits the keyword under Condition 1, and constructs a sorted object subset $\mathsf{SO}_{[t]}.j$ according to Def. 6. Since $\mathsf{SO}_{[t-1]}.j \subseteq \mathsf{SO}_{[t]}.j$, $a_u = \mathbf{acc}(\mathsf{SO}_{[t]}.j)$ can be rapidly calculated from the previous value as the basic solution.

**VO Construction.** Given a query $\widetilde{\mathcal{Q}} = (\widetilde{\mathcal{T}}, K)$, the SP runs algorithm $\mathsf{Search}$ to obtain sets MN and UN as before. For each similar keyword $w_j$, it constructs a sorted object subset $\mathsf{SO}_{[t]}.j$ according to Def. 6, puts the latest $K$ objects regarding keyword branch $w_j||v$ into $\mathrm{SR}_{j,v}$, and calculates the witness as $\pi_j \leftarrow \mathsf{GenWit}(\bigcup_{v\in[b_j]} \mathrm{SR}_{j,v}, \mathsf{SO}_{[t]}.j)$. The VO is set to $\widetilde{\mathcal{Q}}.\mathcal{VO} = \{\mathrm{MN}, \mathrm{UN}, \{\{\mathrm{SR}_{j,v}\}_{v\in[b_j]}, \pi_j\}_{w_j\approx\widetilde{\mathcal{T}}}\}$.

**Verification.** On receiving the VO, the user first checks if the VO meets the following requirements or not: (1) For each similar keyword $w_j$, there are $K$ tuples in set $\mathrm{SR}_{j,v}$ and their sequence numbers are consecutive, where $v \in [b_j]$. (2) For each similar keyword $w_j$, there exists a leaf node $\mathcal{N}_u \in \mathrm{MN}$ s.t. $\mathcal{S}_u||\mathcal{C}_u = w_j$; (3) All nodes in set UN (resp. set MN) indeed mismatch (resp. match) the fuzzy term. If so, the user verifies result completeness as the basic solution. Next, for the similar keyword $w_j$ that corresponds to the leaf node $\mathcal{N}_u = (\mathcal{C}_u, \mathcal{S}_u, a_u, n_u)$ in set MN, the user verifies result integrity and freshness by testing if $\mathsf{VeriWit}(\bigcup_{v\in[b_j]} \mathrm{SR}_{j,v}, \pi_j, a_u) = 1$ and $\sum_{v\in[b_j]} n_{j,v} = n_u$, where $n_{j,v}$ is the highest sequence number in set $\mathrm{SR}_{j,v}$.

**Merging Branches.** In the AKS solution, there is no limit on the splitting operation, rendering the amount of keyword branches to increase linear with the number of relevant objects in the worst case. As analyzed in Section 7.1, the verification cost grows linearly with the amount of keyword branches. To avoid the continued decline of user-side performance, we set a threshold value $\theta_j$ for each keyword $w_j$, so that a miner can merge the keyword branches on demand. Specifically, when a new block $\mathcal{B}_t$ is appended, the miner splits each keyword $w_j \in \mathsf{W}_j$ according to Condition 1, and merges all branches of keyword $w_j$ into one branch if current amount of branches $b_j$ reaches the predefined threshold $\theta_j$. After merging branches of keyword $w_j$, the orders of existing objects may be changed, resulting in $\mathsf{SO}_{[t-1]}.j \not\subseteq \mathsf{SO}_{[t]}.j$ and $\mathsf{SO}_{[t-1]} \not\subseteq \mathsf{SO}_{[t]}$. Hence, the miner needs to recalculate the accumulative values for set $\mathsf{SO}_{[t]}.j$ and set $\mathsf{SO}_{[t]}$ from the scratch in ADS generation. As for veffChain, the miner may locally keep $\varphi_j$, the exponential value of $\mathbf{acc}(\mathsf{SO}_{[t]}.j)$ for each keyword $w_j$, so that $\mathbf{acc}(\mathsf{SO}_{[t]})$ can be rapidly calculated by $g^{\prod_{w_j\in \mathsf{W}_{[t]}} \varphi_j}$. Let MB denote the set of keywords being merged in current block. For keyword

TABLE 2: Comparison of computation costs

| | Miner | SP | User |
|---|---|---|---|
| Strawman | $O(N+m)$ | $O(N+m-K)$ | $O(K)$ |
| Basic | $O(N_t+2\cdot m_t)$ | $O(N+m-K)$ | $O(K)$ |
| Basic* | $O(N_t+2\cdot m_t)$ | $O(N+m-b\cdot K)$ | $O(b\cdot K)$ |
| Advance | $O(N_t)$ | $O(N_S-S\cdot K)$ | $O(S\cdot K)$ |
| Advance* | $O(N_t)$ | $O(N_S-B\cdot K)$ | $O(B\cdot K)$ |

$N_t = |\mathsf{SO}_{[t]} - \mathsf{SO}_{[t-1]}|$ and $N = |\mathsf{SO}_{[t]}|$ are the numbers of keyword (branch)/object/serial tuples in block $\mathcal{B}_t$ and blocks $\mathcal{B}_{[t]}$, respectively; $m_t = |\mathsf{W}_t|$ and $m = |\mathsf{W}_{[t]}|$ are the number of keywords in block $\mathcal{B}_t$ and blocks $\mathcal{B}_{[t]}$, respectively; $b$ is the number of branches for the search term, $N_S = \sum_{w_j=\widetilde{\mathcal{T}}} |\mathsf{SO}_{[t]}.j|$ is the number of objects for all similar keywords, $B = \sum_{w_j=\widetilde{\mathcal{T}}} b_j$ is the number of branches for all similar keywords, and $S$ is the number of similar keywords.

$w_j \in \mathsf{W}_{[t]} - \mathsf{W}_t$, $\varphi_j$ equals its previous value, for keyword $w_j \in \mathsf{W}_t - \mathrm{MB}$, $\varphi_j$ can be incrementally updated from its previous value, and for keyword $w_j \in \mathrm{MB}$, $\varphi_j$ needs to be recalculated. veffChain++ follows the same rule to update the accumulative value in the corresponding leaf node. As for how to determine the threshold value, we first analyze which factors affect the decision. From the previous analyses, we know that the threshold value is positively affected by the ratio of data generation rate $GR$ to block capacity $BC$. When $GR \le BC$, all new objects can be packed in one block, and the threshold value can be set to 1. Under other circumstances, we can simply set it to $\lceil \alpha * GR/BC \rceil$, where $\alpha$ is a predefined constant. Next, we will analyze the impact of threshold value on the performance of our solutions. Obviously, a large threshold value helps to reduce the frequency of merge operations on the miner side, but a larger number of branches causes higher verification costs on the user side. Therefore, a feasible way is to set an initial threshold value according to the value of $GR/BC$ in the first place, and then dynamically adjust the value to offer a good tradeoff between the user-side and miner-side costs. Appendix B provides examples to illustrate the working process of the AKS solution and the merging operation.

# 7 Analysis

Let Strawman, Basic, and Basic* denote the strawman, basic, and improved solutions in veffChain, and let Advance and Advance* denote the basic and improved solutions in veffChain++, respectively. This section analyzes the performance and security of the proposed solutions.

## 7.1 Performance Analysis

The performance is analyzed in the aspects of computational, communication, and storage complexities. As for computation costs, we only consider the expensive group operations related to RSA accumulator. Given a sequence of blocks $\mathcal{B}_{[t]}$, an exact query $\mathcal{Q} = (\mathcal{T}, K)$, and a fuzzy query $\widetilde{\mathcal{Q}} = (\widetilde{\mathcal{T}}, K)$, the comparison result is shown in Table 2.

As for communication costs, the witnesses in VOs are of size $O(1)$ and $O(S)$ in veffChain and veffChain++, respectively. The size of sorted objects is $O(K)$ in Strawman and Basic, $O(b\cdot K)$ in Basic*, $O(S\cdot K)$ in Advance, and $O(B\cdot K)$ in Advance*. Furthermore, veffChain++ requires the VO to incorporate critical nodes to reconstruct the VTrie tree. In terms of storages costs, the ADS in each block is of constant size for all the above solutions. The main differences lie in the following aspects: (1) veffChain allows the SP to locally maintain $\varphi_j$ for each keyword $w_j$, so that the computational cost of VO construction in Basic and Basic* can be reduced to $O(2\cdot m + C - K)$ and $O(2\cdot m + C - b\cdot K)$, respectively, where $C$ is the number of objects associated with the search term. (2) veffChain++ requires the full node to maintain the VTrie tree in addition to verifying result completeness.

## 7.2 Security Analysis

**Theorem 1.** *The verifiable query solutions in veffChain achieve result integrity and freshness, if the hash function is collision resistant, and the RSA accumulator is secure.*

**Theorem 2.** *The verifiable query solutions in veffChain++ achieve result integrity, freshness, and completeness, if the hash function is collision resistant, and the RSA accumulator is secure.*

The proofs of Theorem 1 and Theorem 2 can be found in Appendix C and Appendix D, respectively.

# 8 Discussion

In this section, we will focus on improving query performance and search functionality of veffChain, while leaving the extensions of veffChain++ to our future work. For ease of illustration, the following discussion are based on the basic solution without keyword splitting. The extensions can be applied to the improved solution with minor modification.

## 8.1 Acceleration by Sliding Time Window

For the veffChain framework, the VO construction time has a worst case complexity in linear with the blockchain length, even when an inverted index is used to speed up queries. Specifically, the SP needs to calculate the accumulative values for sets $\mathsf{SO}_{[t]} - \mathrm{SR}_I$ and $\mathsf{LN}_{[t]} - \mathrm{SR}_F$ to generate the witnesses $\pi_I$ and $\pi_F$, result in the computational costs $O(|\mathsf{SO}_{[t]}| - K)$ and $O(|\mathsf{W}_{[t]}|)$, respectively. Compared with the number of keywords $|\mathsf{W}_{[t]}|$, the block length $t$ has a greater impact on the size of sorted object set $|\mathsf{SO}_{[t]}|$. For example, in dataset 4SQ, when $t$ increases from 20 to 300, $|\mathsf{W}_{[t]}|$ increases from 120 to 236, but $|\mathsf{SO}_{[t]}|$ increases from 1,108 to 16,823. Hence, the key to performance improvement lies in accelerating the computation of witness $\pi_I$.

To this end, our original idea is letting the SP locally maintain $\varphi_j$ for each keyword $w_j$, so that the cost of computing witness $\pi_I$ is reduced to $O(|\mathsf{W}_{[t]}| + |\mathsf{SO}_{[t]}.j| - K)$. However, the latest $K$ objects matching a query normally involve only a small number of blocks, and thus there is no need to construct sorted object sets from the whole blockchain. Inspired by previous work [9], we associate each new block with a sliding time window of size $\tau$, so that the ADS and VO can be quickly generated over a small-sized sorted object set constructed from the most recent $\tau$ blocks. Let $\mathcal{B}_{[x,y]}$ be a sequence of blocks $(\mathcal{B}_x, \ldots, \mathcal{B}_y)$, and let $\mathsf{W}_{[x,y]}$ and $\mathsf{SO}_{[x,y]}$ be the set of keywords and the sorted object set for blocks $\mathcal{B}_{[x,y]}$, respectively. Assume that sorted object sets are constructed by Def. 2 under Assumption 1. We have $\mathsf{SO}_{[i-\tau+1,i]} = \mathsf{SO}_{[i]} - \mathsf{SO}_{[i-\tau]}$ for $i \in [\tau, t]$. The basic solution with window size $\tau$ works as follows:

**ADS Generation.** For a new block $\mathcal{B}_t$, the miner generates the ADS as $(\Phi_I, \Phi_U, \Phi_H)$ where $\Phi_U$ and $\Phi_H$ are calculated with Alg. 2, but $\Phi_I = \mathbf{acc}(\mathsf{SO}_{[t-\tau+1,t]})$.

**VO Construction.** Given a query $\mathcal{Q} = (\mathcal{T} = w_s, K)$, the SP sets $\mathcal{Q}.\mathcal{VO} = (\mathrm{SR}_I, \mathrm{SR}_F, \pi_I, \pi_F)$, where $(\mathrm{SR}_F, \pi_F)$ are constructed by Alg. 2, but $(\mathrm{SR}_I, \pi_I)$ are calculated in the following way: The SP first transforms $\mathcal{Q}$ into a keyword/range query $\mathcal{Q}' = (w_s, [b, t])$, where $\mathcal{B}_t$ is the recent

This article has been accepted for publication in IEEE Transactions on Knowledge and Data Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2023.3316127



block and block $\mathcal{B}_b$ contains the $K$-th latest object of keyword $w_s$. The SP then performs according to the following cases: (1) If $t-b<\tau$, it constructs set $\mathrm{SR}_I$ as before and calculates the witness as $\pi_I \leftarrow \mathsf{GenWit}(\mathrm{SR}_I, \mathsf{SO}_{[t-\tau+1,t]})$. (2) If $t-b+1=\alpha\cdot\tau+\beta$, it divides the range $[b,t]$ into $\alpha$ sub-ranges $\{[b_k,t_k]\}_{k=0}^{\alpha-1}$ of length $\tau$ and a sub-range $[b,b+\beta-1]$ of length $\beta$, where $b_k=t-(k+1)\cdot\tau+1$ and $t_k=t-k\cdot\tau$. For each sub-range $[b_k,t_k]$ of length $\tau$, it traverses blocks $\mathcal{B}_{[b_k,t_k]}$ and puts the keyword/object/serial tuples of keyword $w_s$ into set $\mathrm{SR}_k$, while calculating the witness as $\pi_k \leftarrow \mathsf{GenWit}(\mathrm{SR}_k, \mathsf{SO}_{[b_k,t_k]})$. For the last sub-range $[b,b+\beta-1]$, it constructs set $\mathrm{SR}_\alpha$ from blocks $\mathcal{B}_{[b,b+\beta-1]}$, and calculates the witness as $\pi_\alpha \leftarrow \mathsf{GenWit}(\mathrm{SR}_\alpha, \mathsf{SO}_{[b+\beta-\tau,b+\beta-1]})$. Finally, it sets $\mathrm{SR}_I=\{\mathrm{SR}_k\}_{k=0}^{\alpha}$ and $\pi_I=\{\pi_k\}_{k=0}^{\alpha}$.

**Verification.** The user validates the VO and verifies result freshness as the basic solution. To verify result integrity, the user performs as follows: (1) If $t-b<\tau$, the user runs $\mathsf{VeriWit}(\mathrm{SR}_I,\pi_I,\mathcal{B}_t.\Phi_I)$. (2) If $t-b+1=\alpha\cdot\tau+\beta$, the user runs $\mathsf{VeriWit}(\mathrm{SR}_k,\pi_k,\mathcal{B}_{t_k}.\Phi_I)$ for $k\in[0,\alpha-1]$, and $\mathsf{VeriWit}(\mathrm{SR}_\alpha,\pi_\alpha,\mathcal{B}_{b+\beta-1}.\Phi_I)$. Due to the security of RSA accumulator, algorithm $\mathsf{VeriWit}$ outputs 1 only when $\cup_{k=0}^{\alpha}\mathrm{SR}_k \subseteq \mathsf{SO}_{[b+\beta-\tau,t]}$, validating result integrity.

By using sliding time windows, the VO construction cost is mainly affected by the number of blocks covering the search results. This is a great outcome for latest-$K$ queries, which usually involve only a fraction of blocks. For example, a latest-1 query incurs only costs $O(|\mathsf{SO}_{[i,i]}|)$ when the latest object locates in block $\mathcal{B}_i$ and the window size is set to 1. However, it should be noted that in the above extension, the VO size and user-side verification costs grow linearly with the ratio of the number of covered blocks to the window size. Although the optimization technique of multiple sliding time windows [9] can be applied to alleviate this problem, the latest-$K$ results may cover the whole blockchain in the worst case. Hence, our original solution is more suitable for the case in which users with resource-limited devices wish to retrieve sparsely distributed data.

## 8.2 Extension to Boolean Range Queries

In many cases, the user may want to retrieve the latest objects satisfying a query criteria like (*Blood Pressure* $\geq 120) \wedge$ *Influenza*. In this section, we will discuss how to extend veffChain to support Boolean range queries on numerical attributes and keywords. We express an object $o_x$ as $(x,t_x,W_x,V_x)$, where $V_x$ is a vector of numerical attributes, and the rest are defined in the same way as Section 3.1.

**How to Support Boolean Queries.** A Boolean query $\mathcal{Q}$ that include at least one AND clause and $n$ search terms can be transformed into the form of $\mathcal{Q}'=\mathcal{T}_1\wedge\Omega(\mathcal{T}_2,\ldots,\mathcal{T}_n)$, where $\Omega$ is an arbitrary Boolean formula. For simplicity, we assume $\mathcal{T}_j\in\mathcal{Q}'$ equals keyword $w_j$. After transformation, the search results are a subset of set $\mathsf{SO}_{[t]}.1$. Before going deep into details, we provide the following definitions:

**Definition 7** (Mismatched Object Set)**.** *Each keyword $w_j \in \mathsf{W}_{[i]}$ is associated with a set of mismatched objects $\mathsf{MO}_{[i]}.j=\{(w_j,x)\}_{w_j\notin W_x}$, where $(w_j,x)$ means that object $o_x$ does not contain keyword $w_j$. The mismatched object union constructed from blocks $\mathcal{B}_{[i]}$ is denoted by $\mathsf{MO}_{[i]}=\bigcup_{w_j\in\mathsf{W}_{[i]}}\mathsf{MO}_{[i]}.j$.*

**Definition 8** (Matching)**.** *The matching between an object $o_x$ and a search term $\mathcal{T}$ is denoted by $o_x\bowtie\mathcal{T}$. If $\mathcal{T}\in W_x$, we have $o_x\bowtie\mathcal{T}$. The matching between an object $o_x$ and the Boolean formula $\Omega$ is denoted by $o_x\bowtie\Omega$. We have $o_x\bowtie\Omega$ if $\Omega(\mathcal{T}_2,\ldots,\mathcal{T}_n)$ evaluates to true when each term $\mathcal{T}_j\in\Omega$ is replaced with true or false depending on if $o_x\bowtie\mathcal{T}_j$ or not.*

The algorithm details are described in Appendix E. Our main idea is to associate each keyword with a mismatched object set enabling the user to verify that the unreturned object matching the search term $\mathcal{T}_1$ belongs to mismatched object sets of keywords $\{w_2,\ldots,w_n\}$ and thus mismatches the Boolean formula $\Omega$. Specifically, $\mathcal{B}_t.\mathcal{ADS}$ is in the form of $(\Phi_I,\Phi_U,\Phi_H,\Phi_M)$, where $(\Phi_I,\Phi_U,\Phi_H)$ are calculated by Alg. 2, but $\Phi_M=\mathbf{acc}(\mathsf{MO}_{[t]})$ with $\mathsf{MO}_{[t]}$ being constructed using Def. 7. The blockchain is an append-only structure, and thus $\mathsf{MO}_{[t-1]}\subset\mathsf{MO}_{[t]}$ and $\Phi_M$ can be dynamically generated by $\mathbf{acc}(\mathsf{MO}_{[t-1]})^{\prod_{x\in\mathsf{MO}_{[t]}-\mathsf{MO}_{[t-1]}}\mathcal{P}(\mathcal{H}(x))}$. Given the query $\mathcal{Q}'=\mathcal{T}_1\wedge\Omega(\mathcal{T}_2,\ldots,\mathcal{T}_n)$, the SP constructs $\mathcal{Q}'.\mathcal{VO}=(\mathrm{SR}_I,\widetilde{\mathrm{SR}}_I,\mathrm{SR}_M,\mathrm{SR}_F,\pi_I,\pi_F,\pi_M)$, where $\mathrm{SR}_F$ and $\pi_F$ are calculated by Alg. 2 and the remaining components are calculated as follows: The SP scans the sorted object set $\mathsf{SO}_{[t]}.1$, puts the latest $K$ objects matching the Boolean formula $\Omega$ into set $\mathrm{SR}_I$, and puts the objects mismatching $\Omega$ and having sequence number larger than $x$ into set $\widetilde{\mathrm{SR}}_I$, where $x$ is lowest sequence number in set $\mathrm{SR}_I$. For each element $(w_1,id_k,k)\in\widetilde{\mathrm{SR}}_I$, the SP locates the term $\mathcal{T}_j\in\Omega$, s.t. object $o_{id_k}$ does not contain keyword $w_j$, and puts the element $(w_j,id_k)$ into set $\mathrm{SR}_M$. The witnesses are generated as $\pi_I=\mathbf{acc}(\mathsf{SO}_{[t]}-\mathrm{SR}_I)$ and $\pi_M=\mathbf{acc}(\mathsf{MO}_{[t]}-\mathrm{SR}_M)$.

In verification, the user first checks if the VO abides by the following requirements: (1) There are $K$ tuples in set $\mathrm{SR}_I$, and the sequence numbers in set $\mathrm{SR}_I\cup\widetilde{\mathrm{SR}}_I$ are consecutive; (2) The highest sequence number in set $\mathrm{SR}_I\cup\widetilde{\mathrm{SR}}_I$ equals $ln_1$, the latest object number in set $\mathrm{SR}_F$. If so, it verifies the search results by examining if the following equations hold: (1) $\mathsf{VeriWit}(\mathrm{SR}_I,\pi_I,\Phi_I)=1$; (2) $\Phi_U=\Phi_H^{\pi_F\cdot\prod_{x\in\mathrm{SR}_F}\mathcal{P}(\mathcal{H}(x))}$; (3) $\mathsf{VeriWit}(\mathrm{SR}_M,\pi_M,\Phi_M)=1$. Note that equation (1) is related to result integrity, and equations (2) and (3) are used to verify result freshness. In particular, $\mathsf{VeriWit}(\mathrm{SR}_M,\pi_M,\Phi_M)$ outputs 1 only when $\mathrm{SR}_M\subseteq\mathsf{MO}_{[t]}$ verifying the authenticity of set $\mathrm{SR}_M$. In other words, the objects in set $\mathsf{SO}_{[t]}.1$ that are fresher than the search results indeed mismatch the Boolean formula $\Omega$.

**How to Support Range Queries.** Inspired by previous work [3], a numerical value can be transformed into a set of binary prefix strings by using prefix encoding [25] and represented as a set of distinct keywords by using collision-free hashes. Specifically, we first express the numerical value $v$ of attribute $a$ in the binary format $\hat{v}$, and then construct a prefix set $\mathrm{Prefix}(v)$ by replacing the last $k$ bits of $\hat{v}$ with symbol $'*'$, for $k\in[0,|\hat{v}|-1]$. For each element $x\in\mathrm{Prefix}(v)$, the keyword is calculated as $\mathcal{H}(a||x)$. Given a binary tree built over the entire binary space, a range query is transformed into OR clauses over the keywords corresponding to maximal covering nodes. For example, for attribute $a$ with value range $[0,7]$, the binary format of value 6 is 110 with $\mathrm{Prefix}(6)=\{110,11*,1*\}$, and the keywords are $\{\mathcal{H}(a||110),\mathcal{H}(a||11*),\mathcal{H}(a||1*)\}$. Given a binary tree built cover space $\{000,\ldots,111\}$, the maximal covering nodes for query ranges $[0,3]$ and $[0,2]$ are $\{0*\}$ and $\{00*,010\}$, and the queries are transformed into $\mathcal{H}(a||0*)$

and $\mathcal{H}(a||00*) \vee \mathcal{H}(a||010)$, respectively. An illustrative example and relevant experiment results for Boolean range queries are provided in appendixes F and G, respectively.

## 9 Evaluation

In this section, we will evaluate the performance of the proposed blockchain frameworks, and compare them with the seminal frameworks, vChain [3] and vChain+ [9]. Due to space limitation, we only show the performance of basic solutions without keyword splitting in the evaluation, while implementing the AKS solution in Appendix H.

### 9.1 Parameter Settings

In our evaluations, the miner and the SP are set up on a server with Intel Xeon Gold 2.30GHz CPU and 64GB RAM, running Ubuntu 20.04 LTS. And the hyperledger [26] (Version 2.2) is deployed on the server to simulate the real blockchain environment. The user is set up on a portable laptop with Intel Core i7 2.30GHz CPU and 8GB RAM, running Windows 10 system. The experiments (including chaincode) are implemented in Java language. We choose two real datasets for performance evaluation:

• Foursquare (4SQ) [27]. This dataset contains 1 million data records of user check-in information. Each object is represented as $(id, timestamp, [longitude, latitude], \textit{check-in place})$, where the check-in place contains two keywords on average.

• Weather (WEA)[2]. This dataset contains 1.5 million data records that hold hourly weather data for 36 cities from 2012 to 2017. Each object is represented as $(id, timestamp, city, temperature, \textit{weather description})$, where the weather description contains two keywords on average.

According to the data generation rate, the experiments pack data records in 4SQ and WEA within 30s and 1 hour intervals into a block, respectively, so that each block contains a moderate amount of objects. In the experiment, the dataset size $N$ and the parameter $K$ are set to $[10^4, 10^5]$ and $[20, 450]$, respectively. As for fuzzy queries, the number of similar keywords is set to $S = \{2, 4, 6, 8, 10\}$. Meanwhile, we mainly use the following six metrics to evaluate the solutions: (1) The setup time. (2) The size of ADSs. (3) The query time. (4) The VO construction time. (5) The size of VOs. (6) The verification time. The first 2 metrics are executed on the miner side, the last two are executed on the user side, and the rest are executed on the SP side. To better show the impact of parameters $N$ and $K$ on query performance, we fix the number of similar keywords to 1 in Fig. 5-Fig. 8. To minimize deviation, each simulation is run at least $100$ times to get the average value.

### 9.2 Experimental Results

**Setup**. From Fig. 4, we can see that both the setup time and ADS size of all solutions grow with the increase of $N$. This is because a larger $N$ means a larger number of blocks, resulting in more costs for calculating ADSs embedded in block headers. In terms of the setup time, Basic performs best, and Advance performs worst. The reason is that Advance needs to create a VTrie tree in addition, although both Basic and Advance allows for incremental updates. In terms of the ADS size, Basic and Advance generate the most and the least size, respectively. This is because a single ADS in Strawman, Basic, and Advance holds two accumulative values, three accumulative values, and one hash value, respectively. In addition, the setup time of Advance evaluated on WEA is smaller than 4SQ. The main reason is that compared with 4SQ, WEA has less number of distinct keywords, requiring less time to construct the VTrie tree.

**Query Time**. From Fig. 5-(a),(c), we can see the data size has a minor influence on our solutions. This is because the most time-consuming operation is getting data from the ledger. As an inverted index is kept to speed up the query process, the query time is reduced to $O(K)$. From Fig. 5-(b),(d), we know that the larger $K$, the more query time. The reason is intuitive, i.e., as $K$ increases, more objects need to be accessed from the ledger, resulting in longer query time.

**VO Construction Time**. After getting the search results, the SP will build the VO accordingly. As shown in Fig. 6, under different conditions Basic and Advance consume less execution time compared with Strawman. The main reason is that Strawman requires the recalculation of the accumulative values for all mismatched keywords, without locally saving relevant knowledge. For the same reason, we observe from Fig. 6-(a),(c) that the parameter $N$ has a positive impact on the execution time of Strawman, but has relatively minor impact on both Basic and Advance. From Fig. 6-(b),(d), we can see that the execution time of all solutions is negatively correlated with $K$. This is because as $K$ increases, the number of mismatched objects decreases, rendering the time for calculating the witness decrease.

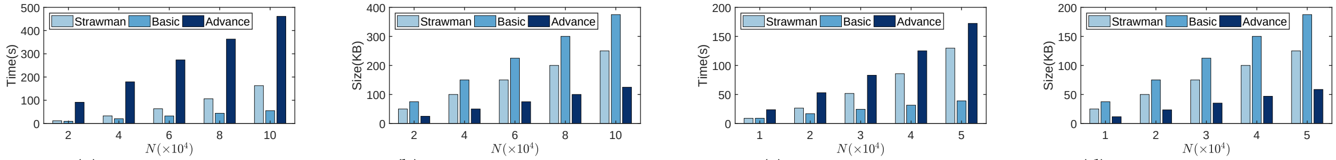
**VO Size**. From Fig. 7, we can see that the VO sizes in Strawman and Basic are less than that in Advance under different parameters. This is because the VOs in Strawman and Basic include only constant-size witnesses, but the VO in Advance contains sufficient nodes to reconstruct the VTrie tree. From Fig. 7-(a),(c) we can see that the VO size in Advance increases, but the VO sizes in Strawman and Basic are constant as $N$ increases. The reason is intuitive, i.e., the larger $N$, the higher the tree, requiring more nodes for tree reconstruction. In terms of the influence of parameter $K$, we can see from Fig. 7-(b),(d) that the VO sizes in all solutions grow with the increase of $K$. This is because a larger $K$ will result in more number of sorted objects to be returned.

**Verification Time**. From Fig. 8-(a),(c), we can observe that the time of both Strawman and Basic is independent of the data size $N$, while the time of Advance has an upward trend as $N$ grows. The reason is that as $N$ increases, Strawman and Basic require only constant group-related operations in verification, but Advance needs more time to reconstruct the VTrie tree in addition. From Fig. 8-(b),(d), we know the time of all solutions grows as $K$ increases. Compared with Strawman and Basic, Advance consumes slightly less time under a moderate $N$. This is because Advance requires fewer group-related operations than Strawman and Basic. When $N$ is not too large, the tree reconstruction time does not take a leading position. From Fig. 4-Fig. 8, we can observe that veffChain++ generates smaller ADSs, but requires full nodes to maintain a VTrie tree (about 3MB and 2.1MB for 4SQ and WEA, respectively), incurs more CPU time on the miner and the SP, and generates larger VOs, compared with veffChain. As for the user-side CPU time, veffChain++ performs better than veffChain only when

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

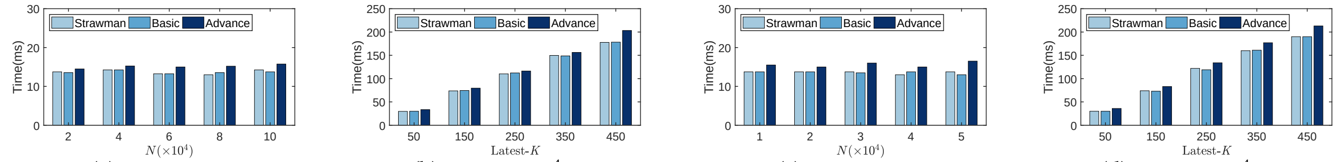
This article has been accepted for publication in IEEE Transactions on Knowledge and Data Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2023.3316127





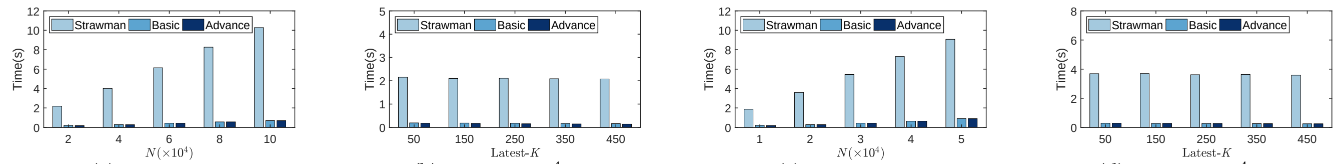

(a) Setup Time (4SQ) (b) The Size of ADSs (4SQ) (c) Setup Time (WEA) (d) The Size of ADSs (WEA)

Fig. 4: The setup costs on the miner side.



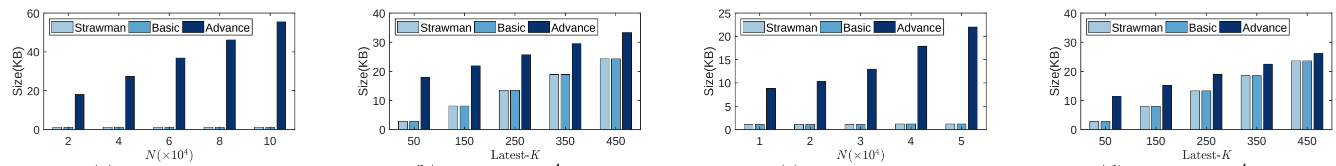

(a) $K = 20$, 4SQ (b) $N = 2 \times 10^4$, 4SQ (c) $K = 20$, WEA (d) $N = 2 \times 10^4$,WEA

Fig. 5: The query processing time on the SP side.




(a) $K = 20$, 4SQ (b) $N = 2 \times 10^4$, 4SQ (c) $K = 20$, WEA (d) $N = 2 \times 10^4$, WEA

Fig. 6: The time of building VOs on the SP side.




(a) $K = 20$, 4SQ (b) $N = 2 \times 10^4$, 4SQ (c) $K = 20$, WEA (d) $N = 2 \times 10^4$, WEA

Fig. 7: The size of VOs transmitted from the SP to the user.

the scale of datasets is moderate. Therefore, veffChain is preferable to veffChain++ if a user wants to quickly retrieve data from a large-scale blockchain database by using exact search terms, and veffChain++ is the optimal choice if a user wants to query the blockchain database with fuzzy terms.

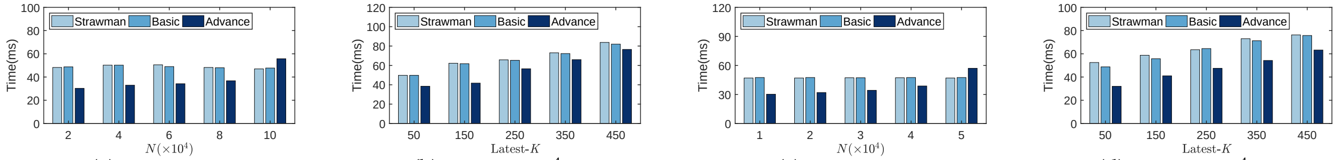
**Fuzzy Queries**. As for veffChain++, we further test the query performance while varying the number of similar keywords $S$. The fuzzy term is generated by replacing either the first few characters or the last few characters of a specific keyword with wildcard $'*'$ so that the number of similar keywords in the dataset is $S$. In reality, the former case that uses wildcard $'*'$ as the prefix of a search term requires accessing a large number of nodes in the VTrie tree, and thus it exhibits the worse performance and can be treated as a baseline. From Fig. 9, we can find that both the computational and communication costs increase as $S$ grows. The main reason is that a larger $S$ means more data satisfies the query criteria, resulting in larger querying time to access data from the ledger, more time to construct a larger VO, and more time to verify the larger VO. Compared with 4SQ, the average length of keywords is longer and the average number of sorted objects associated with a keyword is larger in WEA. Therefore, WEA needs to spend longer query time and VO construction time. In contrast, the VTrie tree in 4SQ contains more leaf nodes, hence generating a bigger VO and requiring longer verification time.

**Comparison with Prior Work.** To validate our frameworks in practice, we conduct comparisons with the seminal frameworks, vChain and vChain+, on dataset 4SQ under varying numbers of blocks. For fair comparisons, the experimental parameters are configured as follows: (1) As veffChain supports only single keyword search, we set the number of search terms in vChain/vChain+ and the number of similar keywords in veffChain++ to 1. (2) We let the SP of vChain and vChain+ return the latest $K$ objects like our frameworks by controlling the temporal ranges of time-window queries. (3) To speed up the query process, we allow the SP to maintain an inverted index in our frameworks, set the size of skiplists in vChain to 5, and set the sliding window size in vChain+ to $\{2, 4, 8, 16, 32\}$. Let vChain1 and vChain2 denote the basic and improved solutions in vChain, respectively. The comparison results are shown in Fig. 10.

Fig. 10-(a) illustrates the setup costs for producing a single block. As for the setup speed, Basic and vChain2 are fastest, while vChain1 is the slowest; As for the ADS size, Advance incurs the minimal cost, while Basic costs a little more than vChain and vChain+ (the ADS sizes of all solutions are less than 0.2KB). From Fig. 10-(b), we can see that as the number of blocks grows, the growth trend of the SP-side CPU time in our solutions is more prominent compared with vChain and vChain+. This is because the more number of blocks means the more number of distinct keywords, which plays a negative impact on the performance of our solutions, but has only a marginal impact on vChain and vChain+. Under the same settings, vChain+ and vChain1 performs best and worst in terms of SP-side CPU time, respectively. Our solutions take more SP-side CPU time than vChain+, but the time difference is within 0.5s. From Fig. 10-(c),(d), we can get that as the number of blocks grows, Advance shows the most obvious increasing tendency in terms of the VO size and user-side CPU time among all solutions. The main reason is that the number of matched/unmatched nodes in Advance gets more as the number of blocks increases thereby generating a larger VO and requiring the user to take more verification time. In terms of the VO size, our Basic performs best, and vChain+

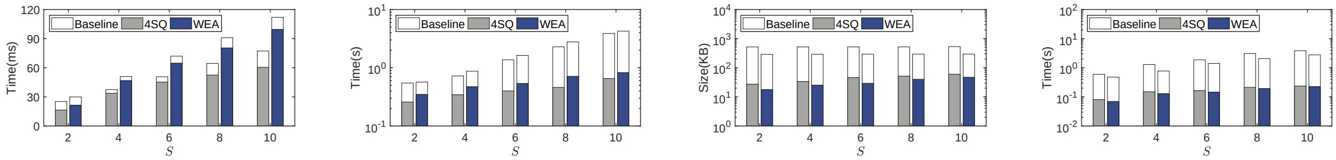
This article has been accepted for publication in IEEE Transactions on Knowledge and Data Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2023.3316127





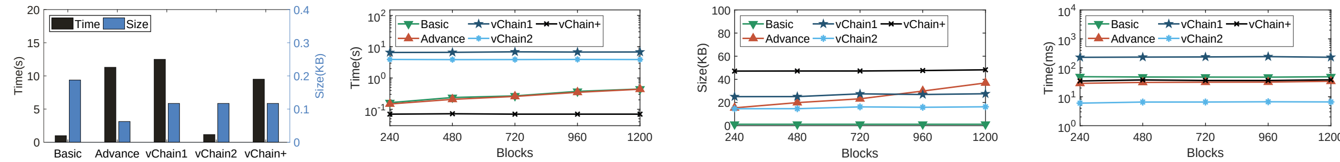

(a) $K = 20$, 4SQ (b) $N = 2 \times 10^4$, 4SQ (c) $K = 20$, WEA (d) $N = 2 \times 10^4$, WEA

Fig. 8: The verification time on the user side.




(a) Query Time (b) VO Construction Time (c) The Size of VOs (d) Verification Time

Fig. 9: The costs of fuzzy query process under different values of $S$ ($N = 2 \times 10^4$, $K = 20$).




(a) Setup Costs (b) SP-side CPU Time (c) The Size of VOs (d) User-side CPU Time

Fig. 10: The comparison results among vChain, vChain+ and our solutions on 4SQ dataset ($K = 20$, $S = 1$).

performs the worst. As for the user-side CPU time, vChain2 performs the best, followed by Advance, vChain+, Basic, and finally vChain1. Although slightly slower than vChain2, our solutions allow the user to verify results within 50ms. To better demonstrate the effectiveness of veffChain++, we further conduct comparisons with vChain and vChain+ while varying the number of similar keywords $S$ in Appendix I.

## 10 Conclusion

In this paper, we propose two frameworks, veffChain and veffChain++, to support latest-$K$ rich queries over a verifiable blockchain database. In veffChain, the accumulator-based ADS embedded in each block header allows a user to effectively verify result integrity and freshness. In veffChain++, the root hash of a VTrie tree as the built-in ADS allows the user to verify result completeness in addition. For improved scalability, we propose the AKS solution to realize the incremental updates of ADSs. The empirical study validates the practical of our frameworks. As part of our future work, we will try to utilize the optimization techniques described in the discussion to further improve the query performance and search functionalities of veffChain++.

## Acknowledgments

This work was supported in part by the National Key Research and Development Program of China under grant 2022YFE0201400; the NSFC grants 62272150, U20A20181, 62272162, 62172159, and 61872133; the Natural Science Foundation of Guangdong Province of China under grant 2023A1515012358; and the Hunan Provincial Natural Science Foundation of China under grants 2021JJ30294, 2023JJ30267, 2020JJ3015.

## References

[1] Z. Du, H. Qian, and X. Pang, "PartitionChain: A scalablse and reliable data storage strategy for permissioned blockchain," *IEEE Transactions on Knowledge and Data Engineering*, 2023.

[2] X. Qi, Z. Zhang, C. Jin, and A. Zhou, "A reliable storage partition for permissioned blockchain," *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[3] C. Xu, C. Zhang, and J. Xu, "vChain: Enabling verifiable boolean range queries over blockchain databases," in *Proc. of SIGMOD*, 2019.

[4] J. Camenisch and A. Lysyanskaya, "Dynamic accumulators and application to efficient revocation of anonymous credentials," in *Proc. of CRYPTO*, 2002.

[5] D. Knuth, *The Art of Computer Programming, Volume 3: (2nd ed.) Sorting and Searching*. Reading, MA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[6] J. Wang, X. Chen, X. Huang, I. You, and Y. Xiang, "Verifiable auditing for outsourced database in cloud computing," *IEEE Transactions on Computers*, 2015.

[7] Q. Liu, Y. Tian, J. Wu, T. Peng, and G. Wang, "Enabling verifiable and dynamic ranked search over outsourced data," *IEEE Transactions on Services Computing*, 2022.

[8] X. Dai, J. Xiao, W. Yang, C. Wang, J. Chang, R. Han, and H. Jin, "LVQ: A lightweight verifiable query approach for transaction history in bitcoin," in *Proc. of ICDCS*, 2020.

[9] H. Wang, C. Xu, C. Zhang, J. Xu, Z. Peng, and J. Pei, "vChain+: Optimizing verifiable blockchain boolean range queries," in *Proc. of ICDE*, 2022.

[10] Y. Peng, M. Du, F. Li, R. Cheng, and D. Song, "Falcondb: Blockchain-based collaborative database," in *Proc. of SIGMOD*, 2020.

[11] Y. Zhu, Z. Zhang, C. Jin, A. Zhou, and Y. Yan, "SEBDB: Semantics empowered blockchain database," in *Proc. of ICDE*, 2019.

[12] Q. Pei, E. Zhou, Y. Xiao, D. Zhang, and D. Zhao, "An efficient query scheme for hybrid storage blockchains based on merkle semantic trie," in *Proc. of SRDS*, 2020.

[13] H. Wu, Z. Peng, S. Guo, Y. Yang, and B. Xiao, "VQL: Efficient and verifiable cloud query services for blockchain systems," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[14] C. Zhang, C. Xu, J. Xu, Y. Tang, and B. Choi, "Gemˆ2-tree: A gas-efficient structure for authenticated range queries in blockchain," in *Proc. of ICDE*, 2019.

[15] C. Zhang, C. Xu, H. Wang, J. Xu, and B. Choi, "Authenticated keyword search in scalable hybrid-storage

blockchains," in *Proc. of ICDE*, 2021.

[16] X. Li, Q. Tong, J. Zhao, Y. Miao, S. Ma, J. Weng, J. Ma, and K.-K. R. Choo, "VRFMS: Verifiable ranked fuzzy multi-keyword search over encrypted data," *IEEE Transactions on Services Computing*, 2023.

[17] Q. Tong, Y. Miao, J. Weng, X. Liu, K. -K. R. Choo, and R. Deng, "Verifiable fuzzy multi-keyword search over encrypted data with adaptive security," *IEEE Transactions on Knowledge and Data Engineering*, 2022.

[18] J. Shao, R. Lu, Y. Guan, and G. Wei, "Achieve efficient and verifiable conjunctive and fuzzy queries over encrypted data in cloud," *IEEE Transactions on Services Computing*, 2022.

[19] Q. Liu, Y. Peng, H. Jiang, J. Wu, T. Wang, T. Peng, and G.Wang, "Authorized keyword search on mobile devices in secure data outsourcing," *IEEE Transactions on Mobile Computing*, 2023.

[20] H. Jin, K. Zhou, H. Jiang, D. Lei, R. Wei, and C. Li, "Full integrity and freshness for cloud data," *Future Generation Computer Systems*, 2018.

[21] J. Zhu, Q. Li, C. Wang, X. Yuan, Q. Wang and K. Ren, "Enabling generic, verifiable, and secure data search in cloud services," *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[22] Y. Hu, R. Zhang, and Y. Zhang, "KV-Fresh: Freshness authentication for outsourced multi-version key-value stores," in *Proc. of INFOCOM*, 2020.

[23] R. Gennaro, S. Halevi, and T. Rabin, "Secure hash-and-sign signatures without the random oracle," in *Proc. of EUROCRYPT*, 1999.

[24] Q. Liu, Y. Peng, H. Jiang, J. Wu, T. Wang, T. Peng, and G.Wang, "SlimBox: Lightweight packet inspection over encrypted traffic," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[25] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and scalable layer four switching", in *Proc. of SIGCOMM*, 1998.

[26] IBM, "Enterprise blockchain solutions and services," 2018, https://www.ibm.com/blockchain.

[27] D. Yang, D. Zhang, and B. Qu, "Participatory cultural mapping based on collective behavior data in location-based social networks," *ACM Transactions on Intelligent Systems and Technology*, 2016.

**Qin Liu** received her B.Sc. in Computer Science in 2004 from Hunan Normal University, China, received her M.Sc. in Computer Science in 2007, and received her Ph.D. in Computer Science in 2012 from Central South University, China. She has been a Visiting Student at Temple University, USA. Her research interests include security and privacy issues in cloud computing. Now, she is an Associate Professor in the College of Computer Science and Electronic Engineering at Hunan University, China.

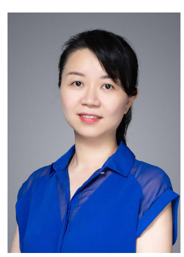
**Yu Peng** is currently working toward the PhD degree with the College of Computer Science and Electronic Engineering, Hunan University, China. His research interests include the security and privacy issues in cloud computing, networked applications, and blockchain.

**Ziyi Tang** received her B.Sc. in Computer Science and Technology in 2020 from Sichuan Agricultural University, China. Currently, She is pursuing the Master degree in the College of Computer Science and Electronic Engineering at Hunan University, China. Her research interests include security issues in BlockChain.

**Hongbo Jiang** received the PhD degree from Case Western Reserve University, in 2008. After that, he joined the faculty of the Huazhong University of Science and Technology as a full professor. Now, he is a full professor with the College of Computer Science and Electronic Engineering, Hunan University. His research concerns computer networking, especially algorithms and protocols for wireless and mobile networks. He is serving as an editor for the IEEE/ACM Transactions on Networking, associate editor for the IEEE Transactions on Mobile Computing, and associate technical editor for the IEEE Communications Magazine.

**Jie Wu** is the Chair and a Laura H. Carnell Professor in the Department of Computer and Information Sciences at Temple University, Philadelphia, PA, USA. Prior to joining Temple University, he was a Program Director at the National Science Foundation and a Distinguished Professor at Florida Atlantic University. His current research interests include mobile computing and wireless networks, network trust and security, and routing protocols. Dr. Wu has regularly published in scholarly journals, conference proceedings, and books. He serves on several editorial boards, including IEEE Transactions on Services Computing, and Journal of Parallel and Distributed Computing. Dr. Wu is a CCF Distinguished Speaker and a Fellow of the IEEE. He is the recipient of the 2011 China Computer Federation (CCF) Overseas Outstanding Achievement Award.

**Tian Wang** received his BSc and MSc degrees in Computer Science from the Central South University in 2004 and 2007, respectively. He received his PhD degree in City University of Hong Kong in 2011. Currently, he is a professor at the Institute of Artificial Intelligence and Future Networks, Beijing Normal University & UIC, China. His research interests include internet of things and edge computing.



**Tao Peng** received the B.Sc. in Computer Science from Xiangtan University, China, in 2004, the M.Sc. in Circuits and Systems from Hunan Normal University, China, in 2007, and the Ph.D. in Computer Science from Central South University, China, in 2017. Now, she is an Associate Professor of School of Computer Science and Cyber Engineering, Guangzhou University, China. Her research interests include network and information security issues.

**Guojun Wang** received his Ph.D. degree in Computer Science, at Central South University, China in 2002. He is a Pearl River Scholarship Distinguished Professor of Higher Education in Guangdong Province, and a Doctoral Supervisor of School of Computer Science and Cyber Engineering, Guangzhou University, China. He has been listed in Chinese Most Cited Researchers (Computer Science) by Elsevier in the past eight consecutive years (2014-2021). His research interests include artificial intelligence, big data, cloud computing, Internet of Things (IoT), and blockchain. He is a Distinguished Member of CCF, a Member of IEEE, ACM and IEICE.