vChain+: Optimizing Verifiable Blockchain Boolean Range Queries (Technical Report)

Haixin Wang[1], Cheng Xu[1,2], Ce Zhang[1], Jianliang Xu[1], Zhe Peng[1], and Jian Pei[2]

[1] *Department of Computer Science, Hong Kong Baptist University, Hong Kong*
[2] *School of Computing Science, Simon Fraser University, Canada*
{hxwang, chengxu, cezhang, xujl, pengzhe}@comp.hkbu.edu.hk, jpei@cs.sfu.ca

***Abstract*—Blockchain has recently gained massive attention thanks to the success of cryptocurrencies and decentralized applications. With immutability and tamper-resistance features, it can be seen as a promising secure database solution. To address the need of searches over blockchain databases, prior work vChain proposed a novel verifiable processing framework that ensures query integrity without maintaining a full copy of the blockchain database. It however suffers from several limitations, including linear-scan search performance in the worst case and impractical public key management. In this paper, we propose a new searchable blockchain system, vChain+, that supports efficient verifiable boolean range queries with additional features. Specifically, we propose a sliding window accumulator index to achieve efficient query processing even for the worst case. We also design an object registration index to enable practical public key management without compromising the security guarantee. To support richer queries, we employ optimal tree-based indexes to index both keywords and numerical attributes of the data objects. Several optimizations are also proposed to further improve the query performance. Security analysis and empirical study validate the robustness and performance improvement of the proposed system. Compared with vChain, vChain+ improves the query performance by up to $913\times$.**

I. Introduction

Blockchain has been receiving tremendous attention in recent years owing to the great success of decentralized applications in various fields such as cryptocurrencies, healthcare, and supply chain management [1]–[3]. It is an append-only ledger built upon the incoming transactions that are agreed upon by a network of untrusted nodes. With the utilization of the hash chains and the distributed consensus protocols, blockchain has the features of immutability and tamper resistance. In a typical blockchain network, there are three types of nodes: *full node*, *miner*, and *light node*, as shown in Fig. 1. A full node maintains a full copy of the blockchain data, including both block headers and complete block states. A miner is also a full node but has an additional responsibility to generate new blocks. A light node, on the other hand, does not maintain the whole blockchain. Only block headers, which include the consensus proofs and the digests of block states, are stored by light nodes. Despite being small in size, the block headers provide sufficient information to verify the integrity of a block.

The unique features of blockchain make it a promising secure database solution, especially in the decentralized environment. As such, there is a growing demand to query the data stored in a blockchain database. For example, in the Bitcoin network, a user may want to find all transactions whose transfer amounts

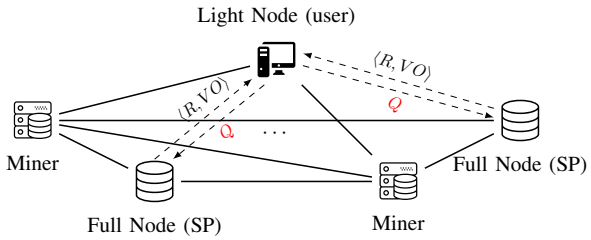


Fig. 1: A Blockchain Network

are between US$1 to US$10 or all transactions associated with some specific sender and receiver addresses in a time interval. Several database companies, such as IBM and Oracle, provide searchable blockchain database solutions by materializing a view of the blockchain data in a traditional centralized database. However, such a design is not desirable for decentralized applications. The query execution integrity is not guaranteed by the centralized party, which could be malicious or compromised. Alternatively, users can maintain a full copy of the entire blockchain database and query the data locally. However, that is impractical to ordinary users as it requires considerable storage, computing, and bandwidth resources.

To tackle the issues mentioned above, Xu *et al.* [4] proposed the vChain framework that supports verifiable boolean range queries over blockchain databases. As shown in Fig. 1, a query user in vChain is only required to act as a light node; the queries are instead outsourced to a full node in the blockchain network, which serves as a *service provider* (SP). Although the SP might be untrusted, users can still verify the integrity of the query results by checking an additional *verification object* (VO). The VO is computed by the SP with the help of a carefully designed *authenticated data structure* (ADS) embedded in the block headers. We will briefly discuss the basic design of the vChain framework and its challenges that limit its practicability.

A. vChain and Its Limitations

In vChain, a specifically designed ADS, AttDigest (as shown in Fig. 2), is added to each block header. AttDigest is computed from a cryptographic set accumulator, which serves as a constant-sized digest to represent a set of data objects. It can also be used to efficiently prove that the data objects in a block mismatch the query condition by using a set disjoint operation. For example, if the object o_i in $block_i$ has two keywords {"A", "B"}, the corresponding AttDigest

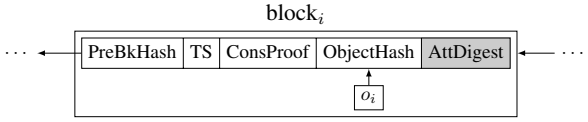


Fig. 2: Block Structure in vChain [4]

is computed as AttDigest $= acc(\{\text{“A”}, \text{“B”}\})$, where $acc(\cdot)$ computes the set accumulative value. When a user asks a query $q =$ “B” $\wedge$ “C”, we can see that $block_i$ mismatches q since “C” $\cap \{\text{“A”}, \text{“B”}\} = \varnothing$. As such, the SP computes a set disjoint proof $\pi_{\varnothing}$ and sends VO $= \{\pi_{\varnothing}, \text{“C”}\}$ to the user. Based on the above information, the user can establish that $block_i$ mismatches the query condition “C” using $\pi_{\varnothing}$ and AttDigest in the block header. In order to efficiently process multiple mismatch blocks in batch for better performance, vChain also proposed the inter-block index, which is a skiplist aggregating data objects across blocks. For each skip, an accumulative value is computed based on the objects in the skipped blocks. If a query mismatches the aggregated blocks due to the same mismatching query condition, we can generate a single mismatch proof to skip these blocks and thus reduce the query cost. Range queries in vChain are achieved by transforming numerical attributes to set-valued attributes with the help of the prefix tree and following a similar query processing procedure.

Although vChain for the first time supports verifiable boolean range queries in blockchain databases, it still has some challenges that limit its practicability. The first one is that in the worst case, the inter-block index cannot help speed up the proving for aggregated mismatching blocks so that the query degenerates to a linear scan process. For example, asume that $q =$ “A” $\wedge$ “B” and three consecutive blocks, containing objects with keywords $o_1 = \{\text{“A”}, \text{“C”}\}$, $o_2 = \{\text{“B”}, \text{“D”}\}$, $o_3 = \{\text{“A”}, \text{“E”}\}$, respectively, are aggregated so that the accumulative value in the inter-block index is computed from $S = \{\text{“A”}, \text{“B”}, \text{“C”}, \text{“D”}, \text{“E”}\}$. In this case, the inter-block index fails to work since S satisfies q. This example shows that in the worst case, vChain has to query each block one by one as the inter-block index cannot aggregate multiple mismatching blocks with different mismatch reasons. With this observation, we evaluate vChain using the 4SQ dataset [5] to measure the utilization of the inter-block indexes for mismatching blocks. Figure 3 shows that in almost 80% cases, the inter-block index fails to work (i.e., skip length is 0), which coincides with the previous analysis. The second limitation of vChain is its practical issue of public key management. Owing to the characteristic of the cryptographic accumulator, its public key size is determined by the largest possible value of the attributes in the system, which would be 2^{256} if a 256-bits hash is used to encode the data attributes. To circumvent this problem, vChain proposed introducing a trusted oracle to generate the public key on the fly. However, such an oracle may not exist in the decentralized environment, which makes vChain challenging to be deployed in real-life applications. Last but not least, since vChain transforms the numerical attributes to set-valued attributes, it can only support integer and fixed-point numbers, which limits its application.

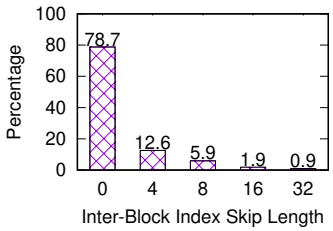


Fig. 3: Statistics of Inter-Block Index Utilization in vChain

B. Our Contributions

To tackle the limitations of vChain, we propose a new searchable blockchain system, vChain+, that supports efficient verifiable boolean range queries with novel designs of ADS to be more efficient, practical, and functional. Instead of using mismatching conditions to process the blocks, we propose a novel sliding window accumulator design for building the ADS in each block. Specifically, for each block, we build a *sliding window accumulator* (SWA) index over the data objects in the most recent k blocks, where k is the sliding window size. With such a design, a time-window historical query $q = [t_s, t_e]$ is first divided into multiple sub-queries, each with a time window size of k. Then, each sub-query can be efficiently processed and verified using the SWA index in a corresponding block.

The major improvement of the SWA index comes from the use of the best index to support different queries (e.g., trie for keyword queries and B+-tree for range queries). For example, considering the aforementioned case with three consecutive blocks, we set $k = 3$ and a trie-based SWA index, including keywords “A”, “B”, “C”, “D”, “E”, can be constructed. During the query processing, we first search the SWA index to get the object sets of keywords “A” and “B”, which are $\{o_1, o_3\}$ and $\{o_2\}$, respectively. Then, a set intersection proof $\pi_{\cap}$ is computed using the accumulating values of the two object sets to prove that the result is $\varnothing$. As such, the SWA index can help speed up the proving for aggregated blocks and mitigates the problem of the inter-block index in vChain.

Apart from the SWA index, we also address the practical issue of public key management by introducing an object registration index. Note that the public key size of a cryptographic set accumulator depends on the universe size of input set elements. As the accumulator in our SWA index is built over data objects (cf. keywords in vChain), we register and index each data object with a small integer ID, so as to bound the universe size and thus limit the public key size. The users can use this index to retrieve the final query results from the corresponding IDs with integrity assurance. Moreover, different from vChain that transforms numeric attributes to set-valued attributes, we use B+-tree to support numerical range queries with floating-point numbers. We also consider arbitrary boolean queries with multiple keywords (cf. limited monotone boolean queries supported in vChain).

Furthermore, we propose several optimizations to further improve the system performance. We propose to build multiple SWA indexes with different sliding window sizes for each block so that the SP can choose the best one according to the query

condition. Meanwhile, as a query may involve a sequence of verifiable set operations, we employ an optimal query plan to reduce the computation overhead of the cryptographic set accumulator. We also propose to prune unnecessary set operations based on empty sets. Security analysis and empirical study are both conducted to validate the proposed methods. Experimental results show that vChain+ improves the query performance by up to $913\times$ and $1098\times$, respectively, against the two constructions of vChain [4].

The rest of this paper is organized as follows. Section II introduces the formal problem formulation, followed by some preliminaries of the cryptographic building blocks in Section III. Section IV presents the processing of verifiable boolean queries, which is then extended to rich query types in Section V. Section VI introduces several optimization techniques and the security analysis is presented in Section VII. Section VIII gives the experiment results. Section IX discusses the related works. Finally, we conclude our paper in Section X.

II. Problem Formulation

As mentioned in Section I, vChain+ follows the same system model as that of vChain [4], but proposes novel ADS designs to provide better query processing efficiency and functionality. The SP is a blockchain full node to provide verifiable query services, while the users are light nodes that maintain only the block headers for verification. On the other hand, the miners, being full nodes, are responsible for appending new blocks to the blockchain and constructing the specifically designed sliding window accumulator (SWA) index in each block to facilitate verifiable queries. With the help of the SWA index, the SP returns both the results and an additional *verification object* (VO) for result integrity verification (as shown in Fig. 1).

The data object in the blockchain is modeled as a tuple in the form of $o_i = \langle t_i, v_i, W_i \rangle$, where t_i is the object's timestamp, v_i denotes the numerical attribute, and W_i is the keyword set of the object. In this paper, we focus on verifiable historical boolean range queries within a certain time window. Specifically, the query is in the form of $Q = \langle [t_s, t_e], [\alpha, \beta], \Upsilon \rangle$, where $[t_s, t_e]$ is the time window predicate,[1] $[\alpha, \beta]$ is the numerical range predicate, and Υ is an arbitrary boolean function on the objects' keyword sets. Different from vChain, Υ is not limited to the monotonic boolean function, but supports $\neg$ (NOT), $\wedge$ (AND), $\vee$ (OR) operators, which is more expressive. Given a query, the SP returns all the data objects that satisfy the query conditions, i.e., $\{o_i = \langle t_i, v_i, W_i \rangle \mid t_i \in [t_s, t_e] \wedge v_i \in [\alpha, \beta] \wedge \Upsilon(W_i) = 1\}$. For instance, in the context of Bitcoin transaction data, a user may ask a query $q = \langle [\text{2021-10}, \text{2021-11}], [10, 20], \mathsf{send{:}2AC0} \wedge \neg \mathsf{receive{:}3E7F} \rangle$ to find all the transactions that happen from October to November of 2021 with a transfer amount between 10 to 20 and associated sender 2AC0 but except with receiver 3E7F.

Threat Model. Similar to vChain [4], we assume that the SP is untrusted and may return tampered or incomplete results due to various reasons such as commercial dishonesty or security breaches. On the other hand, we assume that the blockchain works functionally, i.e., the majority of the miners in the blockchain system are honest and the blockchain network is strongly synchronized. Besides, we assume that the users are trusted and faithfully follow the protocol in the process of query verification. Specifically, with the help of the VO generated by the SP, the users can verify the *soundness* and *completeness* of the results. Soundness means that all the returned results are originated from the blockchain database and satisfy the query conditions. Completeness indicates that no valid result is missing regarding the query conditions.

[1] In the paper, we use timestamp t_i and block height b_i interchangeably.

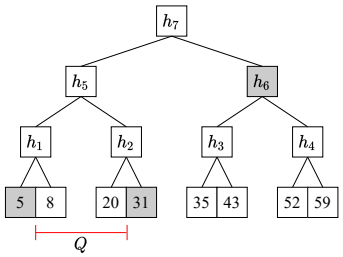


Fig. 4: Merkle Hash Tree

The objective of vChain+ is to design a novel ADS that facilitates the system to achieve much better query performance, more practical public key management, and more flexible query types compared to the existing vChain framework without compromising the security guarantee. We show our designs that fulfill these requirements in the next few sections.

III. Preliminaries

This section gives some preliminaries of cryptographic building blocks which are used in the proposed algorithms.

Cryptographic Hash Function: A cryptographic hash function $H(\cdot)$ is an algorithm which takes an arbitrary-length message m as input and outputs a fixed-length hash digest $H(m)$. It has an important property, collision resistance, indicating that a PPT adversary can find two message $m_1 \neq m_2$ such that $H(m_1) = H(m_2)$ with a negligible probability.

Merkle Hash Tree [6]: A Merkle Hash Tree (MHT) is a tree structure used for efficiently authenticating a set of data objects. Figure 4 shows an example of an MHT with eight objects. In a nutshell, MHT is a binary hash tree constructed in a bottom-up manner. Specifically, each leaf node stores the hash value of the indexed object. Each internal node contains a hash computed using its two child nodes (e.g., $h_6 = H(h_3 || h_4)$, where "||" is the concatenation operation). Thanks to the collision-resistant hash function and the hierarchical structure, the root hash of the MHT (h_7 in Fig. 4) can be used to authenticate the indexed data. For example, for a range query $[6, 25]$, the results are $\{8, 20\}$ with its corresponding proof $\{5, 31, h_6\}$ (shown in shaded nodes in Fig. 4). One can verify these results by reconstructing the root hash using the proof and comparing it with the signed root hash. If they match, it means the results are not being tampered with. Meanwhile, the completeness of the results is ensured by the boundary data $5, 31$ in the proof.

To support other queries, MHT has been extended to

Merkle B+-tree for range queries [7], Merkle R-tree for spatial queries [8], and Merkle Patricia Trie for string search [2].

Cryptographic Set Accumulator [9]: A cryptographic set accumulator is a function that maps a set X to a constant-sized digest $acc(X)$. Similar to a cryptographic hash function, this digest can attest to the corresponding set. Moreover, it supports various verifiable set operations, including intersection (denoted as $\cap$), union (denoted as $\cup$), and difference (denoted as $\setminus$). These set operations can be invoked in a nested fashion and be verified using the accumulative values of the input sets. Specifically, a cryptographic set accumulator scheme consists of the following probabilistic polynomial-time algorithms:

- $\mathsf{ACC.KeyGen}(1^\lambda, U) \rightarrow pk$: On input a security parameter λ and a universe U, it outputs the public key pk.
- $\mathsf{ACC.Setup}(X, pk) \rightarrow acc(X)$: On input a set X and the public key pk, it outputs the accumulative value $acc(X)$ of X.
- $\mathsf{ACC.Update}(acc(X), acc(\Delta), pk) \rightarrow acc(X{+}\Delta)$: On input the set accumulative value $acc(X)$ of a set X, the accumulative value $acc(\Delta)$ of an incremental update Δ (including insertion or deletion of set elements), and the public key pk, it outputs the accumulative value $acc(X{+}\Delta)$ with respect to the new set $X{+}\Delta$.
- $\mathsf{ACC.Prove}(X_1, X_2, opt, pk) \rightarrow \{R, \pi_{opt}\}$: On input two sets X_1, X_2, a set operation $opt \in \{\cap, \cup, \setminus\}$, and the public key pk, it returns the result of the set operation $R = opt(X_1, X_2)$ along with the proof π_{opt}.
- $\mathsf{ACC.Verify}(acc(X_1), acc(X_2), opt, \pi_{opt}, acc(R), pk) \rightarrow \{0, 1\}$: On input the accumulative values $acc(X_1)$, $acc(X_2)$ of sets X_1 and X_2, respectively, a proof π_{opt} with respect to the operation opt, the accumulative value to the answer set R, and the public key pk, it returns 1 if and only if $R = opt(X_1, X_2)$.

In this paper, we use the state-of-the-art cryptographic set accumulator scheme proposed by Zhang *et al.* [9], which supports not only incremental updates but also expressive nested set operations. Another nice property of this scheme is that the proof size for any set operation is constant and the cost of proving a series of nested set operations is linear to the number of set operations. However, it suffers from a relatively high proof generation cost with a complexity of $O(N_1 \cdot N_2)$, where N_1, N_2 are the sizes of the input sets X_1, X_2, respectively. Furthermore, at the expense of expressiveness, its proof size is relatively larger than the one used in vChain [4]. Meanwhile, the public key size of this scheme is $O(|U|^2)$, where $|U|$ is the universe size of the input set elements. To remedy these shortcomings, we propose an object registration index, which assigns each data object with a bounded ID to address the public key size issue in Section IV-A. Furthermore, we propose several techniques to reduce the proof generation overhead in Section VI.

IV. Verifiable Boolean Query Processing

In this section, we consider verifiable boolean queries with multiple keywords. As we explained before, vChain's query processing may degenerate to a linear scan in the worst case. To tackle this issue, we propose a novel sliding window accumulator index design for efficient query processing. The main idea is for each block to build a *sliding window accumulator trie* (SWA-Trie for short) for the data objects in the most recent k blocks, where k is the sliding window size. The root hash of the SWA-Trie is embedded in the blockchain header (see Fig. 5) to support verifiable query processing. In the following, we discuss in detail the issues related to this design: (i) how to manage the accumulator's public key by object registration (Section IV-A), (ii) how to efficiently maintain the SWA-Trie index (Section IV-B), (iii) how to support expressive boolean keyword queries (Section IV-C), and (iv) how to verify the query results (Section IV-D).

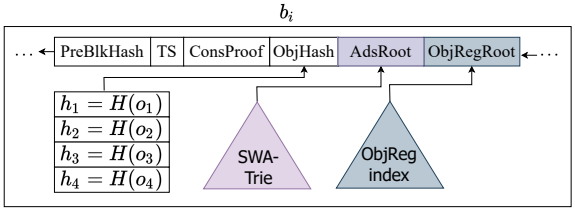


Fig. 5: Extended Block Structure

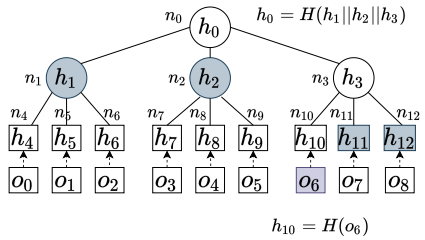
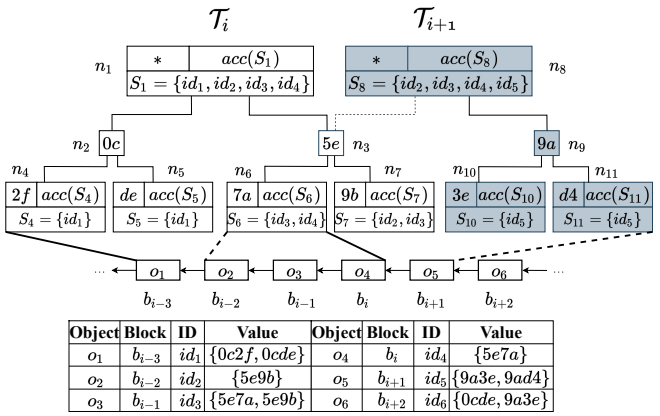


Fig. 6: Object Registration Index

A. Object Registration

As mentioned earlier, we use a cryptographic set accumulator scheme to verify various set operations. However, the public key size of the accumulator scheme used in our design is quadratic to the universe size of input set elements. Recall that the input set elements are the data objects in each sliding window. This poses a challenge in public key management for practical applications. For example, we cannot simply use a cryptographic hash function to encode data objects into 256-bit integer numbers, which would yield a public key with a size of $(2^{256})^2 = 2^{512}$. To tackle a similar issue, vChain proposes to introduce a trusted oracle, which owns a secret key to generate the public keys on the fly [4]. However, such a solution is not very desirable in the context of blockchain applications. It is not easy to find a trusted third party in a decentralized public blockchain environment.

To properly address this issue, we propose to embed an *object registration* (ObjReg) index in each block of the blockchain, as shown in Fig. 5. Instead of storing data objects directly in the set accumulator, we register each data object with an ID and store the IDs in the set accumulator. The ObjReg index is used to track the mapping between the data objects in the recent $2k{-}1$ blocks and their IDs. Here, we enforce a maximum ID, denoted as $MaxID$, which is the maximum possible number of data objects spanned across $2k{-}1$ blocks. Thus, the universe



Object	Block	ID	Value	Object	Block	ID	Value
o_1	b_{i-3}	id_1	$\{0c2f, 0cde\}$	o_4	b_i	id_4	$\{5e7a\}$
o_2	b_{i-2}	id_2	$\{5e9b\}$	o_5	b_{i+1}	id_5	$\{9a3e, 9ad4\}$
o_3	b_{i-1}	id_3	$\{5e7a, 5e9b\}$	o_6	b_{i+2}	id_6	$\{0cde, 9a3e\}$

Fig. 7: Example of SWA-Trie

size of input set elements to the set accumulator is bounded to $MaxID$, thereby limiting the public key size. For example, we set $MaxID$ to 2^{12} for the datasets in our experiments, which limits the public key size to $(2^{12})^2 = 2^{24}$ only. At the same time, this also ensures that the data objects in every consecutive $2k-1$ blocks will always have a distinct ID. As will be shown later, our set operations concern only the data objects within $2k-1$ blocks. Thus, each object in any set operation is guaranteed to have a unique ID.

The ObjReg index is a fully balanced MHT with a fixed fanout. Whenever a new data object arrives, the miner will register the object and assign an ID by incrementing a counter with a modular of $MaxID$. Then, the object is inserted into the ObjReg index according to its ID. Since the ObjReg index is a full tree with a fixed fanout, the location of the object can be easily computed by interpreting the ID as a number using the fanout as the radix. Consider data object o_6 in Fig. 6. As ID 6 can be interpreted as 020 in radix-3, o_6 can be located by following the 1^{st}, 3^{rd}, and 1^{st} node in the respective tree levels. With the ObjReg index and the IDs of the query results, the user can use the ObjReg index to verify the query results as in normal MHT. In the example of Fig. 6, where o_6 is the query result, the SP will return $\{o_6, h_{11}, h_{12}, h_1, h_2\}$ to the user. On the user's side, the root hash of the ObjReg tree is reconstructed and compared against the one stored in the block header. If the verification passes, it can be ensured that data object o_6 indeed corresponds to ID 6.

B. Maintenance of SWA-Trie

Recall that in our design, each SWA-Trie is built over the data objects in the most recent k blocks. Figure 7 shows an example of our designed trie structure with an index sliding window size of 4. For ease of illustration, we assume that each block contains a single data object. In this example, the trie structure $\mathcal{T}_i$ is built over the objects with IDs $\{id_1, id_2, id_3, id_4\}$. Each trie node n contains a hash digest (denoted by h_n) to form a Merkle tree. For the root node and each leaf node, we also store an object ID set (denoted by S_n) and the corresponding set accumulative value (denoted by acc_n). Let $H(\cdot)$ be a cryptographic hash function, $||$ be the string concatenation operator, and $acc(\cdot)$ be the cryptographic set accumulator. We define the fields of each

Algorithm 1: SWA-Trie Maintenance (by the miner)

Function `SWATrieMaintenance`$(b_{i+1},\ b_{i-k+1})$
  Input: current block b_{i+1}, block b_{i-k+1}
  $\mathcal{T}_{i+1} \leftarrow \mathcal{T}_i$;
  Get ObjReg indexes $ObjRegIdx_{i-k+1}$ and $ObjRegIdx_{i+1}$ w.r.t. b_{i-k+1} and b_{i+1}, respectively;
  for *each data object o in b_{i-k+1}* **do**
    $ID \leftarrow ObjRegIdx_{i-k+1}$.lookup$(o)$;
    $\mathcal{T}_{i+1}$.`Update`$(o, ID, false)$;
  for *each data object o in b_{i+1}* **do**
    $ID \leftarrow ObjRegIdx_{i+1}$.insert$(o)$;
    $\mathcal{T}_{i+1}$.`Update`$(o, ID, true)$;
  Write $\mathcal{T}_{i+1}$ to b_{i+1};
Function `Update`$(o,\ ID,\ is\_insert)$
  Input: data o, id ID, insert flag is_insert
  if is_insert **then** $new\_nodes \leftarrow$ insert o to Trie ;
  else $new\_nodes \leftarrow$ delete o from Trie ;
  $l \leftarrow new\_nodes[0]$; `// leaf node`
  $r \leftarrow new\_nodes[-1]$; `// root node`
  if $ID \in S_l$ **then** $acc_\Delta \leftarrow acc(\varnothing)$;
  else $acc_\Delta \leftarrow acc(\{id\})$;
  if is_insert **then**
    $S_l \leftarrow S_l \cup \{ID\}$; $S_r \leftarrow S_r \cup \{ID\}$;
  else
    $S_l \leftarrow S_l \setminus \{ID\}$; $S_r \leftarrow S_r \setminus \{ID\}$; $acc_\Delta \leftarrow -acc_\Delta$;
  $acc_l \leftarrow$ ACC.Update(acc_l, acc_Δ, pk);
  $acc_r \leftarrow$ ACC.Update(acc_r, acc_Δ, pk);
  Update l's hash;
  for *n in new_nodes[1:-1]* **do** `// non-leaf nodes`
    update n's hash;
  Update r's hash;

trie node as follows.

Definition 1 (SWA-Trie Leaf Node)**.** The fields of a leaf node n are defined as:

- $w_n =$ the associated keyword segment of n;
- $S_n =$ the ID set of the objects covered by n;
- $acc_n = acc(S_n)$;
- $h_n = H(H(w_n)||acc_n)$.

Definition 2 (SWA-Trie Non-Leaf Node)**.** Denote the child nodes of a non-leaf node n as $\{c_1, \cdots, c_F\}$. The fields of n are defined as:

- $w_n =$ the associated keyword segment of n;
- $S_n =$ the ID set of the objects covered by n (if n is the root);
- $acc_n = acc(S_n)$ (if n is the root);
- $childHash_n = H(h_{c_1}|| \cdots ||h_{c_F})$;
- $h_n = H(H(w_n)||childHash_n||acc(S_n))$ (if n is root);
- $h_n = H(H(w_n)||childHash_n)$ (if n is non-root).

To incrementally update the SWA-Trie index, we maintain it as a persistent data structure. Algorithm 1 describes the maintenance algorithm. Upon receiving a new block of data objects, the algorithm removes the object IDs in the k-th oldest block (denoted by b_{i-k+1}) and inserts the object IDs in the new block (denoted by b_{i+1}). In the example shown in Fig. 7, to build the SWA-Trie $\mathcal{T}_{i+1}$ for b_{i+1}, the algorithm removes o_1 from $\mathcal{T}_i$ and then inserts o_5 into $\mathcal{T}_{i+1}$. Afterwards, new nodes $\{n_8, n_9, n_{10}, n_{11}\}$ are computed in a bottom-up fashion. It is worth noting that we do not need to recompute the accumulative

Algorithm 2: Boolean Query Processing (by the SP)

Function `BooleanQuery`(Q)
  Input: query condition $Q = \langle [t_s, t_e], \Upsilon \rangle$
  Output: query result R, verification object VO
  $R \leftarrow \varnothing$; $qs \leftarrow$ DivideQuery(Q);
  for *q in qs* **do**
    $\langle [t_{s'}, t_{e'}], \Upsilon \rangle \leftarrow q$; Get block $b_{e'}$ w.r.t. $t_{e'}$;
    $\pi_{trie} \leftarrow \varnothing$; $R_{trie} \leftarrow \varnothing$;
    for *each keyword w in* Υ **do**
      $\langle R_w, \pi_w \rangle \leftarrow$ QuerySWATrie$(w, b_{e'}.root)$;
      Merge π_w to π_{trie}; Add R_w to R_{trie};
    $\langle R_\Upsilon, \pi_\Upsilon \rangle \leftarrow$ Perform ACC.Prove on R_{trie} based on Υ;
    Get ObjReg index $ObjRegIdx_{e'}$ w.r.t. $b_{e'}$;
    $\langle R_{obj}, \pi_{obj} \rangle \leftarrow ObjRegIdx_{e'}$.lookup$(R_\Upsilon)$;
    $R \leftarrow R \cup R_{obj}$;
    Add $\langle \pi_{trie}, R_\Upsilon, \pi_\Upsilon, \pi_{obj} \rangle$ to VO;
  return $\langle R, VO \rangle$;

value of the new root n_8 from scratch. Instead, we can invoke ACC.Update to incrementally update the accumulative value based on the updated object IDs.

C. Verifiable Query Processing

Given a boolean query in the form of $Q = \langle [t_s, t_e], \Upsilon \rangle$, the SP should return all data objects within the time period whose keywords satisfy the boolean expression Υ, i.e., $\{o_i = \langle t_i, W_i \rangle \mid t_i \in [t_s, t_e] \wedge \Upsilon(W_i) = 1\}$. To process the query request, our algorithm consists of three steps. First, the query will be divided into a set of sub-queries, each with a time window length of k. Then, each sub-query will be processed by making use of the SWA index and the ObjReg index. Finally, the results of all sub-queries will be merged to generate the final results. The overall query processing procedure is given in Algorithm 2.

1) Query Dividing: Given a query Q, if the query time window length is no less than k, Q will be divided into multiple k-length sub-queries. If the query window cannot be divided properly, we let the time window of the last sub-query overlap with that of the previous sub-query. For example, assume $k = 4$ and given a query with a time window $[t_1, t_{10}]$, besides the sub-queries with time windows $[t_1, t_4]$ and $[t_5, t_8]$, the last sub-query will be created with a time window $[t_7, t_{10}]$. Note that this may produce redundant results but the correctness of query processing is not affected. On the other hand, if the query time window length is less than k, Q will be treated as a special sub-query, which will be discussed in Section IV-C3.

2) Sub-query Processing: For each sub-query $q = \langle [t_{s'}, t_{e'}], \Upsilon \rangle$ with a time window length of k, we first traverse the SWA-Trie located in block $b_{e'}$ to obtain the intermediate results R_w with corresponding Merkle proof π_w for each keyword w in Υ. To reduce the proof size, we merge π_ws as π_{trie}. Then, verifiable set operations based on Υ will be performed on the intermediate results to obtain the result ID set R_Υ and the set operation proof π_Υ. Finally, the SP will query the ObjReg index located in $b_{e'}$ to find the corresponding data objects with a Merkle proof π_{obj}.

More specifically, first, for each keyword w in Υ, the SP searches the SWA-Trie to find all objects in the trie containing w, which is summarized in Algorithm 3. Starting from the root, the SP traverses the SWA-Trie in a top-down manner. If the keyword segment of a trie node n does not match w, all the data objects under this node do not belong to R_w. In this case, if n is a leaf node, the SP adds w_n and acc_n to π_{trie} as part of the Merkle proof; otherwise, the SP adds w_n and $childHash_n$ (as well as acc_n if n is the root) to π_{trie}. For each node n whose keyword segment matches w, if it is a leaf node, the SP adds S_n to R_w and w_n, acc_n to π_{trie}; otherwise, the subtree will be further explored with w_n (as well as acc_n if n is the root) being added to π_{trie}. Note that the Merkle proofs of different keywords could share some common paths. Hence, the Merkle proofs for all keywords in Υ can be combined to reduce the proof size.

Example. *In the example in Fig. 8, consider sub-queries with time window $[t_{i-2}, t_{i+1}]$ and two keywords 5e7a and 5e9b. We should search the trie $\mathcal{T}_{i+1}$ located in b_{i+1}. We will get the results $R_{5e7a} = S_6 = \{id_3, id_4\}$, $R_{5e9b} = S_7 = \{id_2, id_3\}$ and the Merkle proof $\pi_{trie} = \{\langle *, acc_8 \rangle, \langle 5e \rangle, \langle 9a, childHash_9 \rangle, \langle 7a, acc_6 \rangle, \langle 9b, acc_7 \rangle\}$.*

After getting the intermediate results from trie searches, the SP will conduct verifiable set operations according to Υ using the set accumulator. To support arbitrary boolean queries including the $\neg$ (NOT), $\wedge$ (AND), and $\vee$ (OR) operators, we employ the accumulator proposed in [9]. Specifically, the $\neg$, $\wedge$, and $\vee$ operators in the query boolean expression can be mapped to the corresponding set difference ($\setminus$), set intersection ($\cap$), and set union ($\cup$) operations in the set accumulator scheme.

Example. *In the running example of Fig. 8, for the boolean function $\Upsilon_1 =$ 5e7a $\wedge$ 5e9b, the SP can obtain the result $R_{\Upsilon_1} = R_{5e7a} \cap R_{5e9b} = \{id_3\}$ and the set operation proof π_{Υ_1} by invoking ACC.Prove$(R_{5e7a}, R_{5e9b}, \cap, pk)$. Similarly, for the boolean function $\Upsilon_2 =$ 5e7a $\vee$ 5e9b, the SP can obtain the results $R_{\Upsilon_2} = R_{5e7a} \cup R_{5e9b} = \{id_2, id_3, id_4\}$ and the set operation proof π_{Υ_2} by invoking ACC.Prove$(R_{5e7a}, R_{5e9b}, \cup, pk)$. For the boolean function $\Upsilon_3 = \neg$5e9b, the SP first retrieves all object IDs in $\mathcal{T}_{i+1}$, i.e., $R_* = \{id_2, id_3, id_4, id_5\}$, with its Merkle proof $\pi_* = \{\langle *, childHash_8 \rangle\}$. Then, it can perform a verifiable set difference operation to obtain the results $R_{\Upsilon_3} = R_* \setminus R_{5e9b} = \{id_4, id_5\}$ and the proof π_{Υ_3} by invoking ACC.Prove$(R_*, R_{5e9b}, \setminus, pk)$. For the boolean function $\Upsilon_4 =$ 5e7a $\wedge$ ($\neg$5e9b), a verifiable set difference $R_{5e7a} \setminus R_{5e9b}$ can be performed. For the boolean function $\Upsilon_5 =$ 5e7a $\vee$ ($\neg$5e9b), nested verifiable set operations will be performed. Specifically, the SP will first invoke ACC.Prove on $R_* \setminus R_{5e9b}$ to get $R_{\neg 5e9b}$. Then, a verifiable set union $R_{5e7a} \cup R_{\neg 5e9b}$ will be computed to obtain the set operation proof.*

Next, the SP queries the ObjReg index located in $b_{e'}$ to find the corresponding data objects based on result IDs. It also computes a Merkle proof π_{obj} for the retrieved result objects. Note that the results of the last sub-query may share some common objects with its previous sub-query. As such, when searching the data objects for the last sub-query, the SP will not search those already obtained in the previous sub-query. Finally, the SP packs π_{trie}, R_Υ, π_Υ and π_{obj} together as the

Algorithm 3: Keyword Search (by the SP)

```
Function QuerySWATrie(root, w)
    Input: SWA-Trie root root, keyword w
    Output: query result R, Merkle proof π
    R ← ∅;
    Create an empty queue queue; queue.enqueue(root);
    while queue is not empty do
        n ← queue.dequeue();
        if w_n mismatches w then
            if n.isLeaf() then
                Add ⟨w_n, acc_n⟩ to π;
            else if n.isRoot() then
                Add ⟨w_n, acc_n, childHash_n⟩ to π;
            else
                Add ⟨w_n, childHash_n⟩ to π;
        else if n.isLeaf() then
            R ← R ∪ S_n; Add ⟨w_n, acc_n⟩ to π;
        else
            if n.isRoot() then
                Add ⟨w_n, acc_n⟩ to π;
            else
                Add ⟨w_n⟩ to π;
            for each child c of n do queue.enqueue(c);
    return ⟨R, π⟩;
```

Algorithm 4: Boolean Query Verification (by the user)

```
Function Verify(R, VO)
    Input: result R, verification object VO
    for π in VO do
        ⟨π_trie, R_Υ, π_Υ, π_obj⟩ ← π;
        Verify π_Υ, π_obj w.r.t. the corresponding block header;
        Check R w.r.t. R_Υ and π_obj;
        Perform ACC.Verify based on Υ and π_Υ;
```

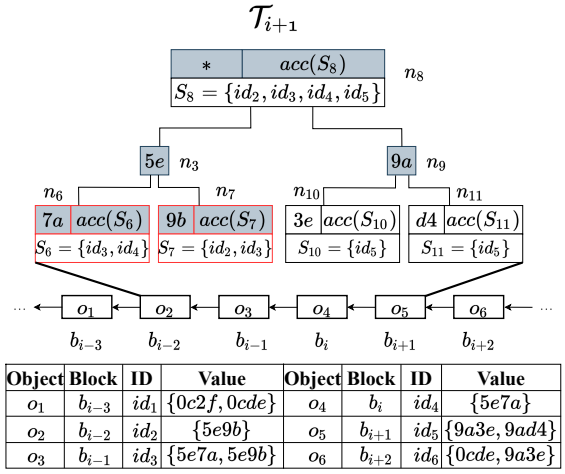
VO for the sub-query.

3) Result Merging: After getting the results of each sub-query, the SP merges them as the final results of the original query.

Note that in a special case where the length of the query time window $[t_s, t_e]$ is less than k, the query Q will be processed as follows. The SP will first visit the block b_e and obtain the result set $R_\Upsilon = \{o_i = \langle t_i, W_i\rangle \mid t_i \in [t_{e-k+1}, t_e] \wedge \Upsilon(W_i) = 1\}$ with the proofs. Next, the SP locates the block b_{s-1}, whose SWA-Trie's root node is used to retrieve the ID set $S_{n_{s-1}}$ of all objects in the sliding window $[t_{s-k}, t_{s-1}]$ and its accumulative value $acc_{n_{s-1}}$. After that, the SP invokes a verifiable set difference operation $\mathsf{ACC.Prove}(R_\Upsilon, S_{n_{s-1}}, \setminus, pk)$ to compute the result set.

D. Query Result Verification

On the user's side, the integrity of the query results can be verified in the following steps. First, the user extracts the proofs from the VO $\langle \pi_{trie}, R_\Upsilon, \pi_\Upsilon, \pi_{obj}\rangle$. Then, the user can verify the integrity of the keyword searches on the SWA-Trie index and the object searches on the ObjReg index by reconstructing their root hashes using π_{trie} and π_{obj}, respectively. If they match the ones stored in the block header, we can establish the soundness and completeness of these searches. After that, the user can perform $\mathsf{ACC.Verify}$ using π_Υ to check the integrity of the set operations for the boolean function Υ. The complete verification procedure is given in Algorithm 4.



Object	Block	ID	Value	Object	Block	ID	Value
o_1	b_{i-3}	id_1	$\{0c2f, 0cde\}$	o_4	b_i	id_4	$\{5e7a\}$
o_2	b_{i-2}	id_2	$\{5e9b\}$	o_5	b_{i+1}	id_5	$\{9a3e, 9ad4\}$
o_3	b_{i-1}	id_3	$\{5e7a, 5e9b\}$	o_6	b_{i+2}	id_6	$\{0cde, 9a3e\}$

Fig. 8: Example of Boolean Query

Example. *In the running example of Fig. 8. On receiving the query results and the VO, the user first reconstructs the trie root hash h'_8 using π_{trie} as follows: $h'_6 = H(H(7a)||acc_6)$, $h'_7 = H(H(9b)||acc_7)$, $h'_3 = H(5e||H(h'_6||h'_7))$, $h'_9 = H(9a||childHash_9)$, and $h'_8 = H(*||H(h'_3||h'_9)||acc_8)$. If h'_8 is identical to h_8 retrieved from the block header, the integrity of the keyword searches is verified. Next, the user verifies the object results with respect to R_Υ using π_{obj}. Finally, the user verifies the integrity of the set operations by invoking $\mathsf{ACC.Verify}$ (e.g., $\mathsf{ACC.Verify}(acc_6, acc_7, \cap, \pi_\Upsilon, acc(R_\Upsilon), pk)$).*

V. Extension to Other Query Types

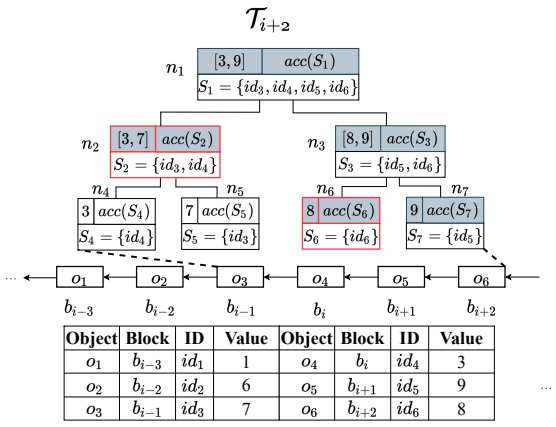
In this section, we discuss how to extend our proposed methods to support other query types such as range queries and boolean range queries.

Single-dimensional Range Queries. Given a range query in the form of $Q = \langle [t_s, t_e], [\alpha, \beta]\rangle$, the SP should return all data objects within the time period whose numerical value falls within $[\alpha, \beta]$, i.e., $\{o_i = \langle t_i, v_i\rangle \mid t_i \in [t_s, t_e] \wedge v_i \in [\alpha, \beta]\}$. We can follow a similar sliding window design for query processing. The miner can build an SWA-B+-Tree to index the numerical values of data objects. Figure 9 shows an example of such an SWA-B+-Tree, where the *index sliding window* size is 4. Each tree node n contains the following fields: a hash digest (denoted by h_n), a numerical value or a numerical range (denoted by v_n or $[l_n, u_n]$), an object ID set (denoted by S_n), and the corresponding set accumulative value (denoted by acc_n). We define them as follows.

Definition 3 (SWA-B+-Tree Leaf Node)**.** The fields of a leaf node n are defined as:

- v_n = the numerical value of n;
- S_n = the ID set of the objects covered by n;
- $acc_n = acc(S_n)$;
- $h_n = H(H(v_n)||acc_n)$.

Definition 4 (SWA-B+-Tree Non-Leaf Node)**.** Denote the child nodes of a non-leaf node n as $\{c_1, \cdots, c_F\}$. The fields of n are defined as:



Object	Block	ID	Value	Object	Block	ID	Value
o_1	b_{i-3}	id_1	1	o_4	b_i	id_4	3
o_2	b_{i-2}	id_2	6	o_5	b_{i+1}	id_5	9
o_3	b_{i-1}	id_3	7	o_6	b_{i+2}	id_6	8

Fig. 9: Example of Range Query

- $[l_n, u_n] = [l_{c_1}, u_{c_F}]$;
- $S_n = S_{c_1} \cup \cdots \cup S_{c_F}$;
- $acc_n = acc(S_n)$;
- $childHash_n = H(h_{c_1}||\cdots||h_{c_F})$;
- $h_n = H(l_n||u_n||childHash_n||acc_n)$.

For example in Fig. 9, $h_4 = H(H(3)||acc(S_4))$ and $h_2 = H(3||7||H(h_4||h_5)||acc(S_2))$. The procedure of maintaining the SWA-B+-Tree is similar to that of the SWA-Trie as discussed in Section IV. Upon receiving a new block b_{i+1}, the miner removes the data objects in b_{i-k+1} and inserts the new data objects in b_{i+1}. For each object update, the miner traverses the updated path in a bottom-up manner to update the corresponding object sets, accumulative values, and hash digests. Similar to the SWA-Trie, the accumulative values can be updated incrementally by invoking ACC.Update. In the case of tree split or merge, the miner can also incrementally update the accumulative values of the affected tree nodes.

The query processing algorithm of single-dimensional range queries is also similar to that of processing boolean keyword queries. The SP will first divide the given query into sub-queries. For simplicity, in the following, we only discuss the range query processing of sub-queries. Given a sub-query $q = \langle [t_s, t_e], [\alpha, \beta] \rangle$, the SWA-B+-Tree in block b_e is traversed to obtain the result ID set R_{range} and its corresponding Merkle proof π_{range}. Starting from the root node, we traverse the SWA-B+-Tree in a top-down manner. For each node n, if its range $[l_n, u_n]$ is entirely covered by $[\alpha, \beta]$, the SP adds S_n to R_{range} and $[l_n, u_n]$, $childHash_n$, and acc_n to π_{range}; if $[\alpha, \beta]$ partially intersects with $[l_n, u_n]$, the child nodes of n will be further explored with $[l_n, u_n]$ and acc_n being added to π_{range}; otherwise if $[\alpha, \beta]$ and $[l_n, u_n]$ have no intersection, the SP simply adds $[l_n, u_n]$, $childHash_n$, and acc_n to π_{range} as part of the Merkle proof. Algorithm 5 describes the detailed query procedure. Then, the SP searches the ObjReg index in b_e to find the data objects corresponding to R_{range} along with a Merkle proof π_{obj}. The result verification on the user's side works similarly to boolean keyword queries. The user uses the proofs returned to reconstruct the root hashes of the SWA-B+-Tree and the ObjReg index, which are then compared with the ones stored in the block header to verify the integrity.

Algorithm 5: Range Query (by the SP)

```
1  Function QuerySWABPlusTree(root, [α, β])
       Input: SWA-B+-Tree root root, query condition [α, β]
       Output: query reuslt R, Merkle proof π
2      Create an empty queue queue; queue.enqueue(root);
3      while queue is not empty do
4          n ← queue.dequeue();
5          if [l_n, u_n] ⊂ [α, β] then
6              R ← R ∪ S_n;
7              if n.isLeaf() then Add ⟨v_n, acc_n⟩ to π;
8              else Add ⟨[l_n, u_n], childHash_n, acc_n⟩ to π;
9          else if [α, β] ∩ [l_n, u_n] ≠ ∅ then
10             Add ⟨[l_n, u_n], acc_n⟩ to π;
11             for each child c of n do queue.enqueue(c);
12         else
13             if n.isLeaf() then Add ⟨v_n, acc_n⟩ to π;
14             else Add ⟨[l_n, u_n], childHash_n, acc_n⟩ to π;
15     return ⟨R, π⟩;
```

Example. *Consider a range query* $Q = \langle [t_{i-1}, t_{i+2}], [2, 8] \rangle$ *in Fig. 9. The SP will traverse* $\mathcal{T}_{i+2}$ *top down and obtain the result ID set* $R_{range} = \{id_3, id_4, id_6\}$ *and a corresponding Merkle proof* $\pi_{range} = \{\langle [3,9], acc_1 \rangle, \langle [3,7], childHash_2, acc_2 \rangle, \langle [8,9], acc_3 \rangle, \langle 8, acc_6 \rangle, \langle 9, acc_7 \rangle\}$ *(shaded in Fig. 9). Then, the SP searches the ObjReg index to get the result object set* $R_{obj} = \{o_3, o_4, o_6\}$ *with a Merkle proof* π_{obj}. *The query results* R_{obj} *along with the* $VO = \langle \pi_{range}, R_{range}, \pi_{obj} \rangle$ *are sent to the user. Upon receiving the query results and the VO, the user reconstructs the SWA-B+-Tree root hash* h'_1 *as follows:* $h'_2 = H(3||7||childHash_2||acc_2)$, $h'_6 = H(H(8)||acc_6)$, $h'_7 = H(H(9)||acc_7)$, $h'_3 = H(8||9||H(h'_6||h'_7)||acc_3)$, $h'_1 = H(3||9||H(h'_2||h'_3)||acc_1)$. *If* h'_1 *is identical to* h_1 *stored in the block header, the integrity of* R_{range} *is attested. Next, the user verifies* R_{obj} *with respect to* R_{range} *by reconstructing the root hash of the ObjReg index using* π_{obj}.

Multi-dimensional Range Queries. It is straightforward to extend our proposed algorithms to support multi-dimensional range queries. For multi-dimensional data, one SWA-B+-Tree can be built for each dimension. The SP can search on each SWA-B+-Tree during query processing to get the query results satisfying the query condition in that dimension. Then, the SP can invoke verifiable set intersections to compute the final results.

Boolean Range Queries. For queries with both a boolean predicate over the keywords and a range predicate over the numerical attributes, the SP can process them as two separate queries to get the intermediate query results satisfying each query condition. Then, the SP invokes a verifiable set intersection to compute the final results. The rest of the algorithm is the same as discussed previously.

To summarize, thanks to the sliding window design, we can choose the most suitable indexes to index the data objects' keywords and numerical attributes, respectively. As such, we can achieve efficient processing for both keyword and range queries. Moreover, due to the expressiveness of the set accumulator adopted in our design, a variety of boolean queries can be supported.

VI. Optimizations

We observe that the bottleneck of the query processing lies in the verifiable set operations, whose overheads are determined by the size of the input sets. In this section, we present three optimization techniques to improve the query performance.

A. Utilizing Multiple Sliding Windows

In our sliding window design, if a query cannot be divided properly, the result set of the last sub-query may share some common objects with the previous sub-query, which affects the overall query performance. It is easy to see that the result set of the last sub-query is influenced by the sliding window size. Therefore, one way to improve the query performance is to build multiple SWA indexes with different sliding window sizes. For example, the miner can build three SWA indexes with sliding window sizes of $2, 4, 8$, respectively. During the query processing, the SP firstly uses the largest sliding window to divide the query into multiple sub-queries. Then, the last sub-query is processed by choosing the minimal sliding window that still covers the required query time window. In particular, assume that the query has a residual time window $[t_{s'}, t_{e'}]$; the window size chosen for the last sub-query should be the minimum value that satisfies $k \geq t_{e'} - t_{s'} + 1$. For example, given three SWA indexes with sliding window sizes of $2, 4, 8$ and a query with time window $[t_1, t_{11}]$, the SP can choose $k_1 = 8$ to process the sub-query with time window $[t_1, t_8]$ and $k_2 = 4$ to process the remaining sub-query with time window $[t_8, t_{11}]$.

B. Optimizing Query Plans

As mentioned in Section III, the verifiable set operations suffer from an expensive proof generation cost with complexity $O(N_1 N_2)$, where N_1 and N_2 are the sizes of the input sets. For example, given two sets $A = \{o_1, o_2, o_3\}$ and $B = \{o_1, o_2, o_3, o_4\}$, we can estimate the cost of $\mathsf{ACC.Prove}(A, B, \cap, pk)$ being $cost(A \cap B) = |A| \times |B| = 12$. It is worth noting that the SP can estimate this cost easily since it can compute the results of all set operations without invoking heavy set accumulator operations. Recall that in our system, the SP searches the relevant SWA indexes and then perform a series of verifiable set operations during the query processing. We observe that some sequences of set operations can be rewritten to other equivalent ones with a smaller proof generation overhead. For instance, given three sets $A = \{o_1, o_2, o_3\}$, $B = \{o_1, o_2, o_3, o_4\}$, and $C = \{o_1\}$, a set operation sequence $(A \cap B) \cap C$ yields a proof generation cost $C_1 = |A| \times |B| + |A \cap B| \times |C| = 15$. However, its equivalent set operation sequence $A \cap (B \cap C)$ has a lower cost $C_2 = |B| \times |C| + |A| \times |B \cap C| = 7$. Therefore, we propose a query plan optimizer to minimize the cost of the verifiable set operations.

Equality saturation is a technique utilizing an e-graph to represent a congruence relation among expressions for program optimizations [10], [11]. Specifically, given a program p with pattern-based rewrites, it first constructs an e-graph that consists of programs equivalent to p based on a set of given rewrite rules.

TABLE I: Rewrite Rules for Set Operations

1	$A \cup B \equiv B \cup A$
2	$A \cap B \equiv B \cap A$
3	$(A \cup B) \cup C \equiv A \cup (B \cup C)$
4	$(A \cap B) \cap C \equiv A \cap (B \cap C)$
5	$(A \cup B) \setminus C \equiv (A \setminus C) \cup (B \setminus C)$
6	$A \setminus (B \cap C) \equiv (A \setminus B) \cup (A \setminus C)$
7	$(A \cap B) \setminus C \equiv (A \setminus C) \cap (B \setminus C)$
8	$A \setminus (B \cup C) \equiv (A \setminus B) \cap (A \setminus C)$
9	$(A \cup B) \cap C \equiv (A \cap C) \cup (B \cap C)$
10	$(A \setminus B) \cap C \equiv (A \cap C) \setminus (B \cap C)$

Then, it extracts the best program from the e-graph. To utilize the equality saturation technique, we define 10 set operation rewrite rules as shown in Table I. The SP can construct an e-graph over the query conditions and find the optimal query plan for the verifiable set operations that yields the lowest proof generation cost.

Furthermore, we observe that we can let the user process the union operations locally, if they are the final operations in the set operation sequence. For example, given a sequence of set operations $((A \cap B) \cup C) \cup D$, the SP can computes only $R_{A \cap B} = (A \cap B)$ using the verifiable set operation and return C, D, as well as $R_{A \cap B}$ with an intersection proof. The user can locally execute the final union operations $(R_{A \cap B} \cup C) \cup D$. This optimization not only reduces the expensive verifiable set operations but also saves space for set operation proofs. Furthermore, the data transmission cost will not increase because all objects in $R_{A \cap B}$, C, and D are results, thus no extra non-result objects are returned.

C. Pruning Empty Sets

During the query processing, some intermediate sets may turn out to be empty. Thus, we may apply an early stop technique to prune unnecessary set operations. For example, given a sequence of set operation $A \setminus (B \cap (C \cup D))$, if $B = \varnothing$, then the result of this sequence is A. In this case, the SP can skip computing C, D, $C \cup D$, and $A \setminus \varnothing$. In general, the SP can traverse the set operation query plan, which is a directed acyclic graph, to find intermediate empty sets and then prune unnecessary set operations. By doing so, we can improve both the query and verification performance and achieve a smaller VO size.

VII. Security Analysis

In this section, we perform a security analysis on our verifiable query processing algorithms. We start by defining the security notion.

Definition 5 (Secure)**.** The verifiable query processing algorithms are secure if the probability for all PPT adversaries to succeed in the following experiment is negligible:

- Run the SWA index construction algorithm and send all data objects O to an adversary;
- The adversary outputs a query Q, a result R, and a VO.

We say the adversary succeeds if the VO passes the verification and one of the following conditions is satisfied:

- There exists an object o_x such that $o_x \in R$ and $o_x \notin O$;

- There exists an object o_x such that $o_x \in R$ and o_x does not satisfy Q;
- There exists an object o_x such that $o_x \in O$ and o_x satisfies Q, but $o_x \notin R$;

The above definition guarantees that the probability, for a malicious SP to convince the user with an unsound or incomplete result, is negligible. We now show that the verifiable query processing algorithms indeed satisfy the desired security requirement.

Theorem 1. *Our proposed verifiable query processing algorithms are secure with respect to Definition 5, if the cryptographic hash function and set accumulator are collision resistant, the set operation proof is secure, and the blockchain integrity as well as availability are ensured.*

Proof We prove this theorem by contradiction.

Case 1: There exists an object o_x such that $o_x \in R$ and $o_x \notin O$. In this case, the adversary forges an object as a query result. However, the user will check the integrity of the query result with respect to the ObjReg index by reconstructing its hash root and verifying with the one stored in the block header. A successfully forged result yields two ObjReg indexes with different data objects but the same hash root. This implies that a collision of the underlying cryptographic hash function exists, which leads to a contradiction.

Case 2: There exists an object o_x such that $o_x \in R$ and o_x does not satisfy Q. This case is impossible since the user will check if the returned result matches the query locally.

Case 3: There exists an object o_x such that $o_x \in O$ and o_x satisfies Q, but $o_x \notin R$. Note that the user (running a light node) syncs the block headers with the blockchain network. Thus, the user always verifies the result with respect to the latest block header. Now suppose there is a missing result o_x. During the verification, the user will check the SWA index (for both boolean keyword queries and range queries) and a series of verifiable set operations (for boolean keyword queries). A missing result means that the adversary is able to forge an incorrect Merkle proof or an incorrect set operation proof. Since the user checks against the hash root of the SWA index stored in the block header, a forged Merkle proof implies a collision to the underlying cryptographic hash function. Similarly, a forged set operation proof contradicts with the security property of the underlying cryptographic set accumulator scheme [9]. Both of them lead to a contradiction.

VIII. Experimental Evaluation

In this section, we evaluate the performance of the proposed vChain+ system. Two datasets are used in the experiments.

- **Foursquare** (4SQ) [5]: The 4SQ dataset includes 1M user check-in records with timestamps. We pack the records within every 30s as a block and each record is represented in the form $\langle timestamp, [longitude, latitude], \{\textit{check-in place's keywords}\}\rangle$.
- **Ethereum** (ETH) [2]: The ETH dataset is extracted from the Ethereum blockchain from Dec 17, 2018 to Dec 26, 2018. It contains around 58,100 blocks with around 3.27M transaction records and the time intervals of the blocks are roughly 15s. Each record can be represented in the form of $\langle timestamp, [amount], \{addresses\}\rangle$, where $amount$ is the transfer amount and $\{addresses\}$ are the addresses of senders and receivers.

TABLE II: Miner's ADS Construction Cost

Dataset	ETH		4SQ	
	T	S	T	S
vChain-acc1	0.26	125.6	0.39	28.1
vChain-acc2	0.04	126.1	0.04	29.3
vChain+	0.11	451.5	0.25	1209.1

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

We run experiments on a machine with dual Intel Xeon E5-2620 v3 2.4GHz CPUs, running CentOS 8. We limit query users to use only 4 threads during the verification, whereas the miner and the SP use all available CPU cores. The vChain+ system is implemented in Rust programming language and the following dependencies are used: Arkworks[2] for bilinear pairing over the BN254 curve to implement the set accumulator, Blake2b[3] for 256-bit hash operations, and Rayon[4] for parallel computation. The source codes are available at https://github.com/hkbudb/vchain-plus. The same programming language and dependencies are also used to implement vChain [4] as the baseline, including two proposed accumulator constructions labeled as vChain-acc1 and vChain-acc2.

We use the following metrics to evaluate the performance of vChain+: (i) the CPU time of the query including SP's query processing and user's result verification, (ii) the size of the VO transmitted from the SP to the user. For each experiment, we randomly generate 10 queries and report the average results. By default, we set the selectivity for numerical range queries as 10%. For boolean query conditions, we use either an $\vee$-connected or $\wedge$-connected boolean function with two keywords.

A. ADS Construction Cost

Table II shows the ADS construction cost on the miner side, including the ADS construction time and the ADS size. In vChain, the maximum size of the inter-block index is set to be 32. For vChain+, we set the sliding window sizes as $\{2, 4, 8, 16, 32\}$ and the fanout for the SWA-B+-tree and the ObjReg index as 4. All optimizations presented in Section VI are employed for vChain+. It can be observed from Table II that the index construction time of vChain+ is longer than vChain-acc2 but shorter than vChain-acc1. Moreover, vChain+ yields a larger ADS compared with vChain. This is expected since the set accumulator used in vChain+ has a size larger than the ones used in vChain to support more expressive set operations. Additionally, as we discussed in Section VI-A, the design of multiple sliding windows introduces multiple SWA indexes, which also increases the ADS size in each block. On the user side, the block header has a constant size of 104 bytes

[2]https://github.com/arkworks-rs/algebra
[3]https://github.com/oconnor663/blake2_simd
[4]https://github.com/rayon-rs/rayon

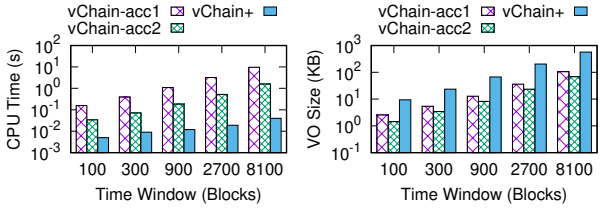


Fig. 10: ∧-Connected Boolean Query Performance (ETH)

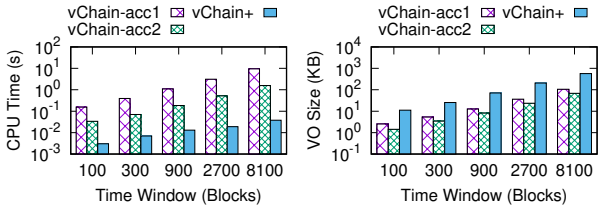


Fig. 11: ∨-Connected Boolean Query Performance (ETH)

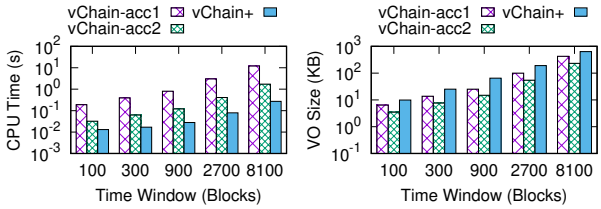


Fig. 12: ∧-Connected Boolean Query Performance (4SQ)

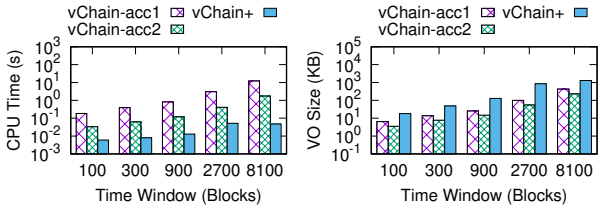


Fig. 13: ∨-Connected Boolean Query Performance (4SQ)

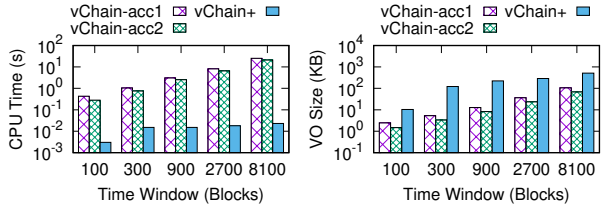


Fig. 14: Range Query Performance (ETH)

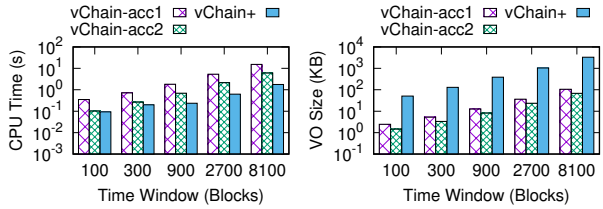


Fig. 15: Range Query Performance (4SQ)

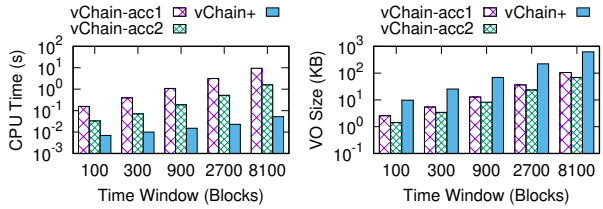


Fig. 16: ∧-connected Boolean Range Query Performance (ETH)

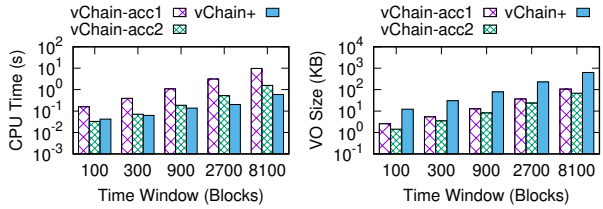


Fig. 17: ∨-connected Boolean Range Query Performance (ETH)

for both vChain and vChain+.

B. Query Performance

Figures 10 to 17 compare the query performance of vChain and vChain+ by varying the query time window from 100 to 8,100 blocks. Five types of query conditions, including ∨- and ∧-connected boolean keyword queries, range-only queries, and ∨- and ∧-connected boolean range queries, are examined. Thanks to the tree-based index searching and accumulator-based sliding window design, vChain+ can handle varied types of queries efficiently. Overall, vChain+ improves the query performance by up to 913× against vChain-acc2 and up to 1098× against vChain-acc1. Note that the VO size of vChain+ is larger than that of vChain in most cases. This is because the size of the set operation proofs generated by vChain+ is relatively larger than that in vChain. However, the total time for VO transmission and query processing of vChain+ is still better than vChain, considering a global-median 29.06 Mbps mobile network speed [12]. For example, as shown in Fig. 16, when the query time window is 8,100, the VO size and the query time of vChain+ are 623KB and 0.05s, respectively, and those of vChain-acc2 are 68KB and 1.59s, respectively. The total time for VO transmission and query processing of vChain+ under the median mobile network speed is 0.221s, which still improves the performance of vChain by 7.3×.

When processing ∨-connected boolean range queries on ETH, we observe that vChain-acc2 slightly outperforms vChain+ when the time window length is 100 blocks (Fig. 17). This is because the set operation proof generation dominates the query time in vChain+. As the ∨-connection boolean condition involves union operations, it results in enlarged input sets for ACC.Prove, which leads to heavier cryptographic operations.

C. Impact of Optimizations and Selectivities

We now evaluate the influence of three different optimization techniques on the query performance and the VO size. We test ∧-connected boolean range queries on the ETH dataset. We enable all optimizations as the baseline (denoted as *all*) then disable each of them to investigate their impact. Specifically, we run the experiment with (i) no multi-sliding windows (*no multi-win*), (ii) not optimizing query plans (*no qp*), and (iii) not pruning empty sets (*no prune*). Figure 18 shows the query performance of different optimizations by varying the query time window from 100 to 8,100 blocks. As can be seen, pruning empty sets and optimizing query plans work for most of the queries and bring the biggest performance improvement. In

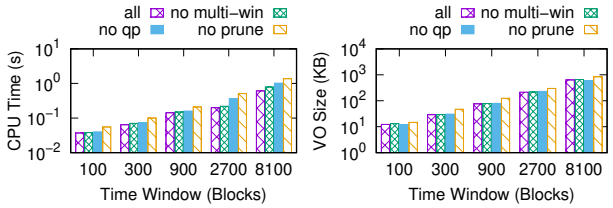


Fig. 18: Query Performance vs. Optimization (ETH)

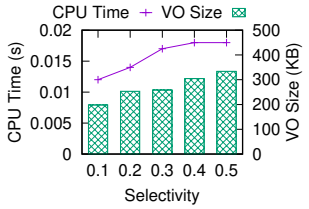


Fig. 19: Impact of Selectivity (ETH)

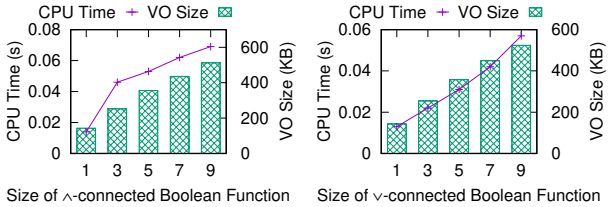


Fig. 20: Impact of Boolean Function Size (ETH)

contrast, the performance improvement of utilizing multiple sliding windows is relatively smaller.

Next, we evaluate the impact of range condition selectivity and boolean function size. Figure 19 shows the range query performance with range selectivity varied from 10% to 50% when the query time window length is fixed at 900 blocks. It can be observed that the CPU time and the VO size rise with the increasing selectivity. This is expected because the SP needs to query more objects from the SWA-B+-Tree and return larger proofs as the selectivity increases. Figure 20 shows the boolean query performance by varying the boolean function size from 1 to 9 when the query time window length is fixed at 2,700 blocks. As expected, the CPU time and the VO size of boolean queries increase with the boolean function size.

We have evaluated the impact of multiple sliding windows on the index construction and query costs. In the interest of space, we include these results in Appendix A.

IX. Related Work

In this section, we briefly review several related studies on blockchain technology and verifiable query processing.

A. Blockchain Technology

Blockchain technology has gained much attention from both academia and industries. Several studies have investigated the blockchain execution model for improving the system scalability [13]–[15]. Sharding techniques have also been studied to scale the blockchain system horizontally [16], [17]. Moreover, some works have focused on reducing the peers' storage overhead by utilizing distributed data storage [18], [19] or moving the on-chain states to off-chain with a stateless design [20], [21]. Along the same direction, Li *et al.* investigated the cost-effective data feeds to blockchains via workload-adaptive data replication [22].

There are also several works studying verifiable query processing over blockchain databases. Hu *et al.* studied applying searchable encryption in blockchain smart contracts to achieve verifiability, which however may incur high execution costs [23]. Xu *et al.* proposed the vChain framework to support verifiable boolean range queries over the blockchain databases [4]. Zhang *et al.* proposed a gas-efficient ADS for range and keyword queries in hybrid-storage blockchains [24], [25]. Moreover, FalconDB also targeted verifiable blockchain databases but proposed to delegate the result verification to a payment contract, which both reduces the user's cost and incentives the SP to perform honestly under an incentive model [26] .

B. Verifiable Query Processing

Verifiable query processing has been extensively studied in the context of outsourced databases to ensure the result integrity even if the SP is untrusted [7], [8], [27]–[34]. Generally, there are two approaches. The first one is to transform a general verifiable query to a Boolean or arithmetic circuit for computing a proof attesting to the execution integrity [27], [28]. However, this approach suffers from high even impractical time for computing the proof. The second category is to design an ADS for specific queries. MHT and its variants [7], [8], [29], [30], mentioned in Section III, are widely used to build ADSs for different queries. It often has efficient proof generation thanks to its hierarchical structure and fast cryptographic hash function. On the other hand, [31], [32] considered using the cryptographic set accumulator for more complex set-valued queries and nested queries. However, the set accumulators used by these works cannot be incrementally updated. Recently, [33], [34] leveraged secure hardware (e.g., Intel SGX) to facilitate verifiable queries. Although the secure hardware provides efficient proof generation, it brings extra deployment costs and may also be prone to side-channel attacks [35], [36].

X. Conclusion

In this paper, we have proposed a new searchable blockchain system, vChain+, that supports verifiable boolean range queries. We have proposed an SWA index design to achieve efficient query performance with integrity assurance. Moreover, an ObjReg index has been designed to enable practical public key management. We have also developed SWA-Trie and SWA-B+Tree to efficiently support a rich variety of queries. Security analysis has been conducted to show the robustness of the proposed methods. Empirical results demonstrate that vChain+ substantially improves the query performance of vChain. As for future work, we plan to extend our SWA index design to support more complex queries such as aggregation queries and join queries.

Appendix

In this appendix, we evaluate the impact of multiple sliding windows. We fix the query as a ∧-connected boolean range

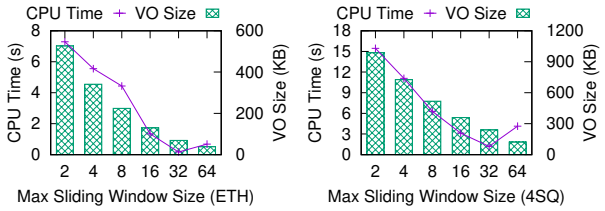


Fig. 21: Impact of Max Sliding Window

TABLE III: Construction Cost of Multiple Sliding Windows

# Sliding Windows	ETH		4SQ	
	T	S	T	S
1	0.03	130.8	0.06	378.3
2	0.05	234.8	0.12	679.5
3	0.07	322.6	0.17	912.6
4	0.09	394.2	0.22	1087.1
5	0.11	451.5	0.25	1209.1

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

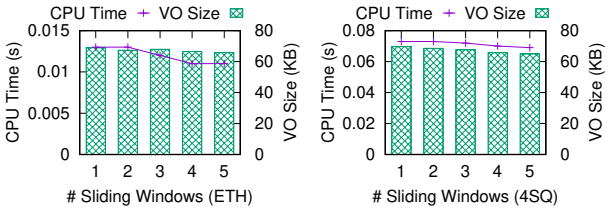


Fig. 22: Impact of Multiple Sliding Windows

query with range selectivity of 10%, query time window length of 900 blocks, and boolean function size of 2.

First, we evaluate the impact of the maximum sliding window size on the query costs. We vary the maximum sliding window size from 2 to 64, i.e., constructing the SWA indexes (i.e., SWA-Trie and SWA-B+-Tree) using the window sizes $\{2\}$, $\{2, 4\}$, $\{2, 4, 8\}$, $\{2, 4, 8, 16\}$, $\{2, 4, 8, 16, 32\}$, and $\{2, 4, 8, 16, 32, 64\}$, respectively. Figure 21 shows the query performance under different sliding window settings. The query time decreases when the maximum sliding window size varies from 2 to 32, but rises when the maximum sliding window size further increases from 32 to 64. The reason is as follows: On the one hand, a larger sliding window size will reduce the number of sub-queries, thereby reducing the number of verifiable set operations; on the other hand, a larger sliding window size may yield larger intermediate result sets, which makes the verifiable set operations heavier. When the maximum sliding window size increases from 32 to 64, the heavy cryptographic operations dominate the query time. The VO size generally decreases by using a larger sliding window size because less sub-queries lead to less set operation proofs.

Next, we evaluate the impact of the number of sliding windows on the index construction and query costs. We vary the number of sliding windows from 1 to 5, i.e., constructing the SWA indexes using the window sizes $\{32\}$, $\{16, 32\}$, $\{8, 16, 32\}$, $\{4, 8, 16, 32\}$, and $\{2, 4, 8, 16, 32\}$, respectively. Table III shows the index construction costs with different sliding windows. As expected, the index construction time and the index size increase with the number of sliding windows since maintaining multiple indexes will consume more time and storage space. Figure 22 shows the query performance under these sliding window settings. The query time and the VO size generally decrease by using more sliding windows. This is because the SP can select the best-fit sliding window for the last sub-query to minimize the intermediate result set, therefore reducing the query costs.

Acknowledgement This work is supported by Hong Kong RGC Projects C2004-21GF, 12201520, 12200819 & 12201018 and CCF-AFSG Research Fund RF20210014. Pei's research is supported in part by the NSERC Discovery Grant program and a grant of the National Research Council Canada (NRC) New Beginnings Initiative. All opinions, findings, conclusions and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] S. Nakamoto. "Bitcoin: A peer-to-peer electronic cash system." (2008), [Online]. Available: https://bitcoin.org/bitcoin.pdf.

[2] G. Wood. "Ethereum: A secure decentralised generalised transaction ledger." (2014), [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf.

[3] J. M. Graglia and C. Mellon, "Blockchain and property in 2018: At the end of the beginning," *Innovations: Technology, Governance, Globalization*, 2018.

[4] C. Xu, C. Zhang, and J. Xu, "vChain: Enabling verifiable boolean range queries over blockchain databases," in *ACM SIGMOD*, 2019, pp. 141–158.

[5] D. Yang, D. Zhang, and B. Qu, "Participatory cultural mapping based on collective behavior data in location-based social networks," *ACM TIST*, pp. 1–23, 2016.

[6] R. C. Merkle, "A certified digital signature," in *Proc. CRYPTO*, 1990, pp. 218–238.

[7] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin, "Dynamic authenticated index structures for outsourced databases," in *ACM SIGMOD*, 2006, pp. 121–132.

[8] Y. Yang, S. Papadopoulos, D. Papadias, and G. Kollios, "Authenticated indexing for outsourced spatial databases," *The VLDB Journal*, pp. 631–648, 2009.

[9] Y. Zhang, J. Katz, and C. Papamanthou, "An expressive (zero-knowledge) set accumulator," in *IEEE EuroS&P*, 2017, pp. 158–173.

[10] R. Joshi, G. Nelson, and K. Randall, "Denali: A goal-directed superoptimizer," *ACM SIGPLAN Notices*, pp. 304–314, 2002.

[11] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "Egg: Fast and extensible equality saturation," *Proceedings of the ACM on Programming Languages*, pp. 1–29, 2021.

[12] "Global median speeds october 2021," Speedtest Global Index. (2021), [Online]. Available: https://www.speedtest.net/global-index.

[13] P. Ruan, T. T. A. Dinh, D. Loghin, M. Zhang, G. Chen, Q. Lin, and B. C. Ooi, "Blockchains vs. distributed databases: Dichotomy and fusion," in *ACM SIGMOD*, 2021, pp. 1504–1517.

[14] P. Ruan, D. Loghin, Q.-T. Ta, M. Zhang, G. Chen, and B. C. Ooi, "A transactional perspective on execute-order-validate blockchains," in *ACM SIGMOD*, 2020, pp. 543–557.

[15] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich, "Blurring the lines between blockchains and database systems: The case of hyperledger fabric," in *ACM SIGMOD*, 2019, pp. 105–122.

[16] M. J. Amiri, D. Agrawal, and A. El Abbadi, "Sharper: Sharding permissioned blockchains over network clusters," in *ACM SIGMOD*, 2021, pp. 76–88.

[17] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi, "Towards scaling blockchain systems via sharding," in *ACM SIGMOD*, 2019, pp. 123–140.

[18] Z. Xu, S. Han, and L. Chen, "Cub, a consensus unit-based storage scheme for blockchain system," in *IEEE ICDE*, 2018.

[19] X. Qi, Z. Zhang, C. Jin, and A. Zhou, "BFT-Store: Storage partition for permissioned blockchain via erasure coding," in *IEEE ICDE*, 2020.

[20] C. Xu, C. Zhang, J. Xu, and J. Pei, "SlimChain: Scaling blockchain transactions through off-chain storage and parallel processing," *Proceedings of the VLDB Endowment*, pp. 2314–2326, 2021.

[21] A. Chepurnoy, C. Papamanthou, S. Srinivasan, and Y. Zhang, "Edrax: A cryptocurrency with stateless transaction validation," *Cryptology ePrint Archive*, 2018.

[22] K. Li, Y. Tang, J. Chen, Z. Yuan, C. Xu, and J. Xu, "Cost-effective data feeds to blockchains via workload-adaptive data replication," in *Proceedings of the 21st International Middleware Conference*, 2020, pp. 371–385.

[23] S. Hu, C. Cai, Q. Wang, C. Wang, X. Luo, and K. Ren, "Searching an encrypted cloud meets blockchain: A decentralized, reliable and fair realization," in *IEEE INFOCOM*, 2018.

[24] C. Zhang, C. Xu, J. Xu, Y. Tang, and B. Choi, "Gemˆ 2-tree: A gas-efficient structure for authenticated range queries in blockchain," in *IEEE ICDE*, 2019, pp. 842–853.

[25] C. Zhang, C. Xu, H. Wang, J. Xu, and B. Choi, "Authenticated keyword search in scalable hybrid-storage blockchains," in *IEEE ICDE*, 2021, pp. 996–1007.

[26] Y. Peng, M. Du, F. Li, R. Cheng, and D. Song, "FalconDB: Blockchain-based collaborative database," in *ACM SIGMOD*, 2020, pp. 637–652.

[27] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *IEEE S&P*, 2013, pp. 238–252.

[28] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou, "vSQL: Verifying arbitrary sql queries over dynamic outsourced databases," in *IEEE S&P*, 2017, pp. 863–880.

[29] Q. Chen, H. Hu, and J. Xu, "Authenticated online data integration services," in *ACM SIGMOD*, 2015, pp. 167–181.

[30] C. Xu, J. Xu, H. Hu, and M. H. Au, "When query authentication meets fine-grained access control: A zero-knowledge approach," in *ACM SIGMOD*, 2018, pp. 147–162.

[31] C. Xu, Q. Chen, H. Hu, J. Xu, and X. Hei, "Authenticating aggregate queries over set-valued data with confidentiality," *IEEE TKDE*, pp. 630–644, 2018.

[32] Y. Zhang, J. Katz, and C. Papamanthou, "IntegriDB: Verifiable sql for outsourced databases," in *ACM CCS*, 2015, pp. 1480–1491.

[33] W. Zhou, Y. Cai, Y. Peng, S. Wang, K. Ma, and F. Li, "VeriDB: An sgx-based verifiable database," in *ACM SIGMOD*, 2021, pp. 2182–2194.

[34] A. Arasu, K. Eguro, R. Kaushik, D. Kossmann, P. Meng, V. Pandey, and R. Ramamurthy, "Concerto: A high concurrency key-value store with integrity," in *ACM SIGMOD*, 2017, pp. 251–266.

[35] P. Kocher, J. Horn, A. Fogh, *et al.*, "Spectre attacks: Exploiting speculative execution," in *IEEE S&P*, 2019.

[36] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *IEEE S&P*, 2019.