vChain+: Optimizing Verifiable Blockchain Boolean Range Queries

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

[1]Hong Kong Baptist University, Hong Kong [2]Simon Fraser University, Canada

{hxwang, chengxu, cezhang, xujl, pengzhe}@comp.hkbu.edu.hk, jpei@cs.sfu.ca

Background

- **Background**: Increasing demand to query blockchain data
- **Blockchain Database Solution**: Relay on a trusted Service Provider for query services

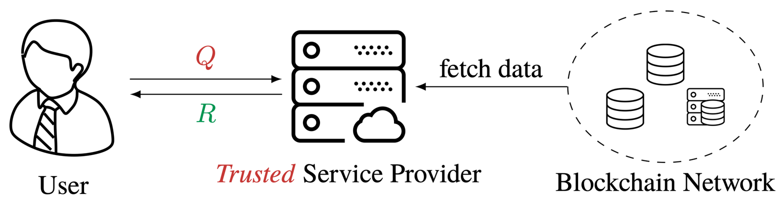

Fig. 1: Workflow of Blockchain Database

- **Issue**: The trusted assumption may not always hold
  - Return partial result to save transmission bandwidth
  - Return tampered data maliciously
- **State-of-the-art: vChain**
  - Let users be light nodes and outsource queries to full nodes (Service Provider)
  - Employ verifiable computation to return result and cryptographic proof
  - System Model
    - Miners construct new block with *authenticated data structure* (ADS) embedded in block header
    - Full nodes compute query results with proof called *Verification Object* (VO)
    - Users verify the results using VO and ADS from block headers
- – Solution: Extend the block header with an *AttDigest* which serves as the ADS
  - Use *AttDigest* to prove mismatching objects
  - Attributes of object $o_i \to S_i$; Query $q \to S_q$
  - $S_i \cap S_q \neq \varnothing$: return o_i as a result
  - $S_i \cap S_q = \varnothing$: generate a set disjoint proof using *AttDigest*

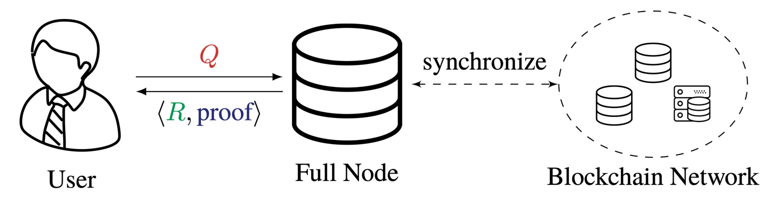

Fig. 2: System Model of vChain

Limitations of vChain

- Query processing may require linear scan
  - Highly depend on data distribution
- Large public key size
  - The pk size of the accumulator used is $O(|U|)$
  - Encoding attributes by 256-bit hash $\to$ pk size $= 2^{256}$
- Limited query type
  - Only support AND ($\wedge$) and OR ($\vee$) operators
  - NOT ($\neg$) operator not supported
  - Only support integer and fixed-point numbers

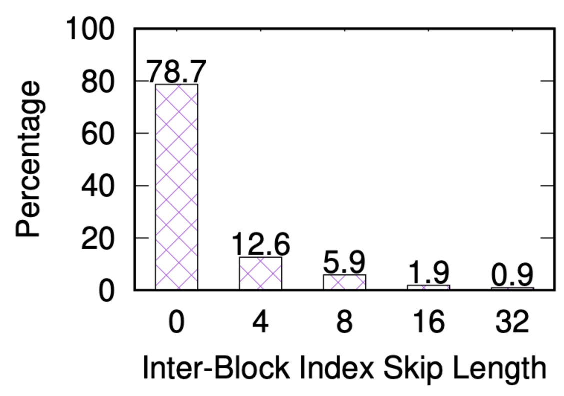

Fig. 3: Statistics of Index Utilization in vChain

Our Solution: vChain+

- A novel design of ADS to be more practical, efficient, and functional
- – A Sliding Window Accumulator (SWA) index for efficient and richer query processing
  - Built over data objects in current block and its previous $k-1$ blocks (totally k blocks)
  - k: sliding window size
  - An object registration (ObjReg) index for practical public key management

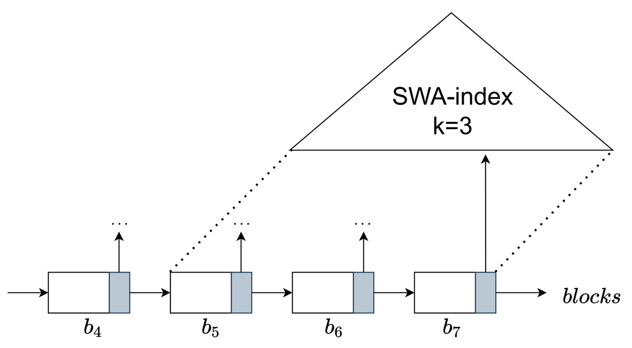

Fig. 4: SWA index Overview

- The SWA index and ObjReg index are designed using *Merkle Hash Tree* and *Cryptographic Set Accumulator*

Cryptographic Building Blocks

- **Merkle Hash Tree**: Enable efficient data verification.
  - A bottom-up constructed multi-way tree.
  - Hash function combining child nodes.
  - Root hash is used to authenticate a set of data objects.
  - Example
    - $Q = [6, 25] \to R = \{8, 20\}, \pi = \{5, 31, h_6\}$

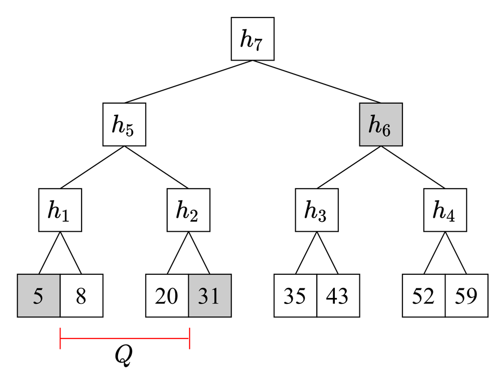

Fig. 5: Merkle Hash Tree

- **Cryptographic Set Accumulator**:
  - Map a set to an element in cyclic multiplicative group in a collision resistant way
  - Utility: prove set operations ($\cap, \cup, \setminus$)
    - $\mathsf{KeyGen}(1^\lambda) \to pk$: generate public key
    - $\mathsf{Setup}(X, pk) \to acc(X)$: compute accumulative value of X
    - $\mathsf{Prove}(X_1, X_2, opt, pk) \to \{R, \pi_{opt}\}$: on input two sets X_1 and X_2, and an operation $opt \in \{\cap, \cup, \setminus\}$, output $R = opt(X_1, X_2)$ and a proof π_{opt}
    - $\mathsf{Verify}(acc(X_1), acc(X_2), opt, \pi_{opt}, acc(R), pk) \to \{0, 1\}$: on input $acc(X_1)$, $acc(X_2)$, opt, π_{opt}, and $acc(R)$, output 1 iff $R = opt(X_1, X_2)$

vChain+: Object Registration

- Issue: the set accumulator requires a pk with size of $O(|U|^2)$
- Observation: the SWA index is built over data objects
- Idea: register each object with an ID and store IDs in set accumulator
  - ID $= counter{+}{+} \mod MaxId$; $MaxId$: max # objects within $2k-1$ blocks
  - ID $\in [0, MaxId - 1] \to |U| = MaxId$
- Build an ObjReg index to track the mapping between data objects and their IDs
  - A Merkle Hash Tree to retrieve authenticated data objects (as query results) with IDs

Boolean Query

- Use an SWA-Trie for efficient Boolean query processing
  - $Q = \langle [t_s, t_e], \Upsilon \rangle$
  - Divided Q into sub-queries with time window size of k
    - $Q = [t_1, t_{10}], k = 4$
    - $q_1 = [t_1, t_4], q_2 = [t_5, t_8], q_3 = [t_7, t_{10}]$
  - Process each sub-query using trie-search and verifiable set operations
- Example
  - Query processing
    - $q = \langle [t_1, t_4], 5e7a \wedge 5e9b \rangle$
    - locate T_4 in b_4 and perform trie-search on $5e7a$ and $5e9b$
    - $R_{5e7a} = \{o_3, o_4\}, R_{5e9b} = \{o_2, o_3\}$
    - Merkle proof $\pi_4 = \{\langle *, acc(S_1) \rangle, \langle 5e \rangle, \langle 9a, h_{child_3} \rangle, \langle 7a, acc(S_4) \rangle, \langle 9b, acc(S_5) \rangle\}$
    - $\mathsf{Prove}(R_{5e7a}, R_{5e9b}, \cap, pk) \to \{R, \pi_\cap\}$
    - $R = \{o_3\}, VO = \{\pi_4, \pi_\cap\}$
  - Result verification
    - Re-construct SWA-Trie root using π_4
    - Compare with $AdsRoot$ in block header
    - $\mathsf{Verify}(acc(S_4), acc(S_5), R, \cap, \pi_\cap)$

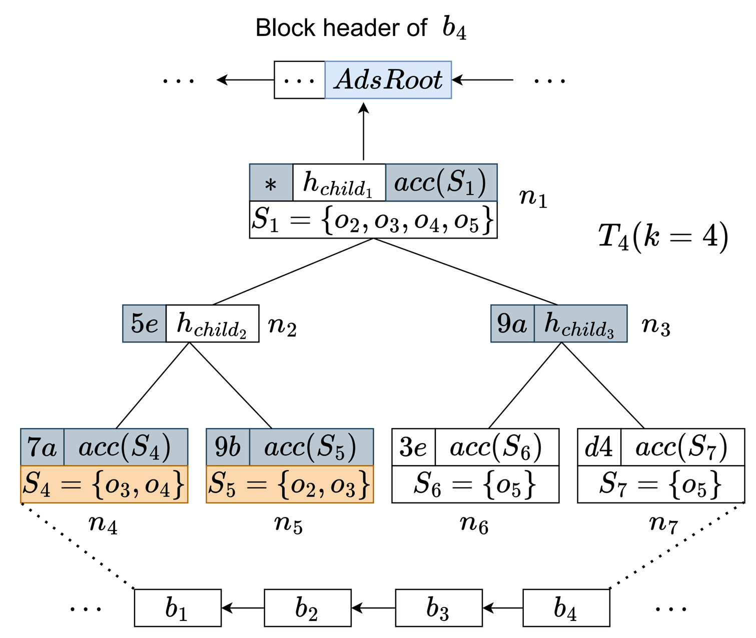

Fig. 6: Boolean query using SWA-Trie

Extension to Other Queries

- Range query
  - SWA-B+-Tree for indexing numerical values for each dimension
  - B+-Tree search on each dimension to obtain intermediate result sets
  - Verifiable set intersections on intermediate results to get the final results
- Boolean range query
  - $\mathsf{BooleanQuery}(Q_W) \to \langle R_W, \pi_W \rangle$
  - $\mathsf{RangeQuery}(Q_V) \to \langle R_V, \pi_V \rangle$
  - $\mathsf{Prove}(R_W, R_V, \cap, pk) \to \langle R, \pi_\cap \rangle$

Optimization

- Multiple sliding windows
  - Build multiple SWA indexes with different k values and choose the best-fit k value when processing queries
- Optimize query plan
  - Find all equivalent plans and pick the one with the smallest cost
- Prune empty set
  - Apply early stop to prune unnecessary set operations

Performance Evaluation

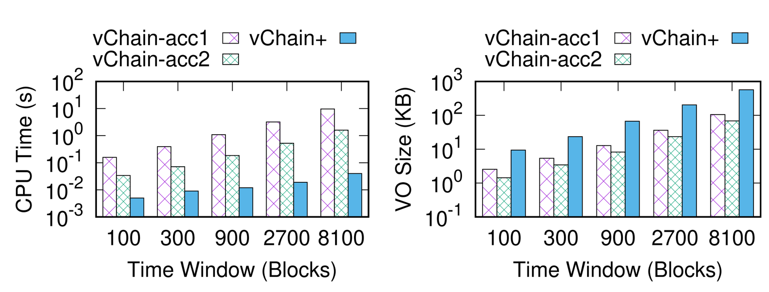

Fig. 7: Boolean Query Performance

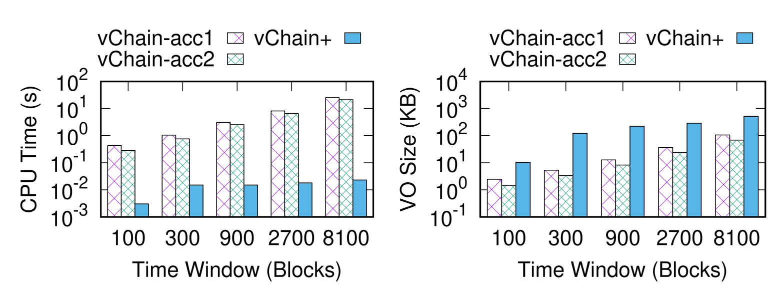

Fig. 8: Range Query Performance