HKBU Database Group

Towards Searchable Blockchain

Jianliang Xu

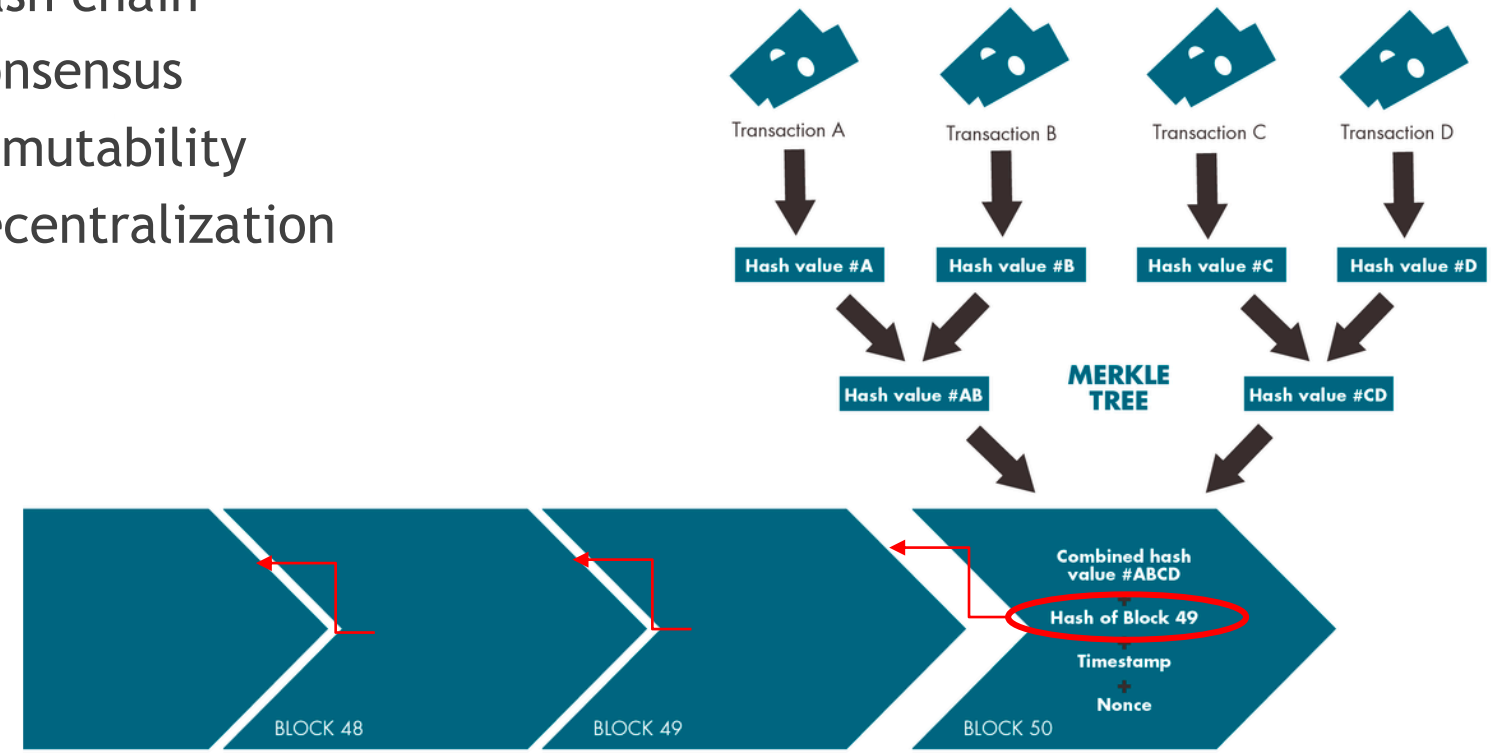
Hong Kong Baptist University

香港浸会大学计算机系 徐建良

http://www.comp.hkbu.edu.hk/~db/

Blockchain Technology

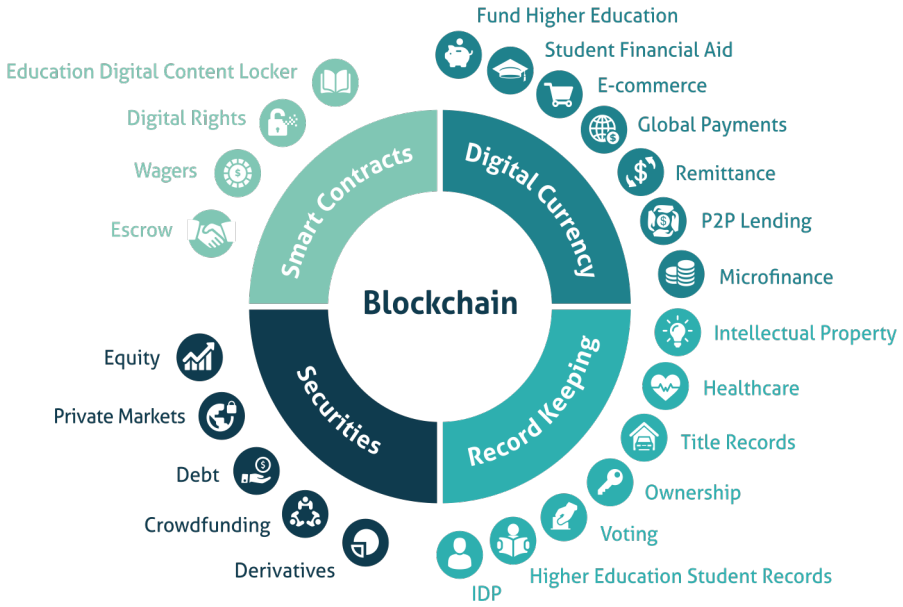
- Blockchain: Append-only data structure collectively maintained by a network of (untrusted) nodes
  - Hash chain
  - Consensus
  - Immutability
  - Decentralization



Source: Wikimedia Commons

Blockchain Technology

- Blockchain: Append-only data structure collectively maintained by a network of (untrusted) nodes
  - Hash chain
  - Consensus
  - Immutability
  - Decentralization
- Applications
  - Digital identities
  - Decentralized notary
  - Distributed storage
  - Smart contracts
  - …



Source: FAHM Technology Partners

Blockchain Database Solutions

- Increasing demand to search the data stored in the blockchain
- Blockchain database solutions to support SQL-like queries

IBM HYPERLEDGER

ORACLE Cloud

SAP Leonardo

BIGCHAIN DB

FlureeDB

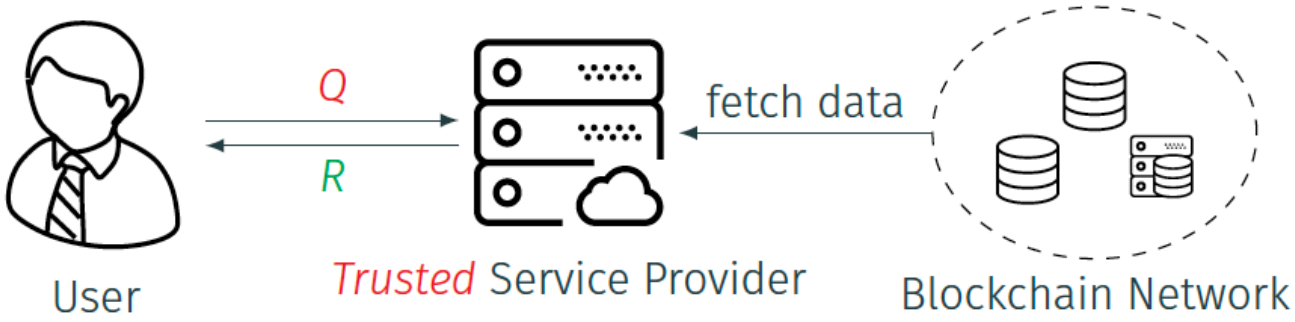
WOLK

bluzelle

SwarmDB

Blockchain Database Solutions

- Increasing demand to search the data stored in the blockchain
- Blockchain database solutions to support SQL-like queries

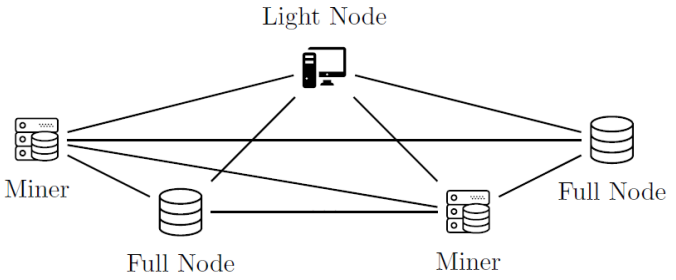


Issue: *relying on a trusted party who can faithfully answer user queries*

HKBU Database Group

Secure Blockchain Search

- The assumption of trusted party may not always hold
- Basic solution to integrity-assured blockchain search
  - Becoming full node
  - High cost
    - Storage: to store a complete replicate (200 GB for Bitcoin as of June 2019)
    - Computation: to verify the consensus proofs
    - Network: to synchronize with the network
- Better solution: becoming light node and outsource query processing to full node
  - Low cost: maintaining block headers only (<50 MB for Bitcoin)
- Challenge: how to maintain query integrity?



Solution #1: Smart Contract

- A *trusted program* to execute user-defined computation upon the blockchain
  - Smart Contract reads and writes blockchain data
  - Execution integrity is ensured by the consensus protocol
- Blockchain offers trusted storage and computation capabilities
  - Function as a *trusted virtual machine*

	Traditional Computer	Blockchain VM
Storage	RAM	Blockchain
Computation	CPU	Smart Contract

Solution #1: Smart Contract

- Leverage Smart Contract for trusted query processing
  - Users submit query parameters to the blockchain
  - Miners execute query processing algorithms and write results into the blockchain
  - Users read results from the blockchain

- Drawbacks
  - Long latency: long time for the consensus protocol to confirm a block
  - Poor scalability: transaction rate of the blockchain is limited
  - Privacy concern: query history is permanently and publicly stored in the blockchain
  - High cost: executing smart contracts in ETH requires paying gas to miners

    (*INFOCOM 2018 requires 4,201,232 gas = 0.18 Ether = 25 USD per query*)

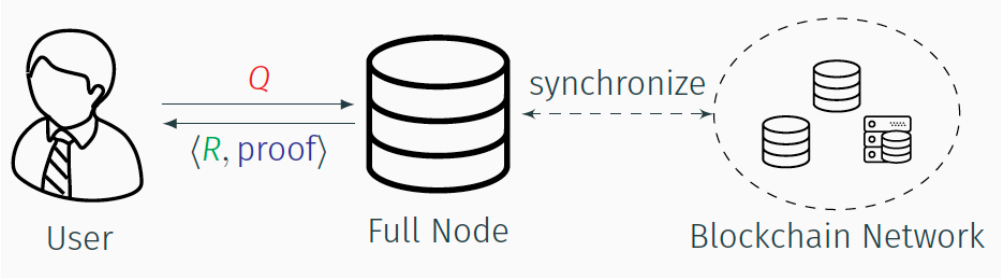
Hu S., et al. "Searching an Encrypted Cloud Meets Blockchain: A Decentralized, Reliable and Fair Realization." IEEE INFOCOM.2018.

Solution #2: Verifiable Computation

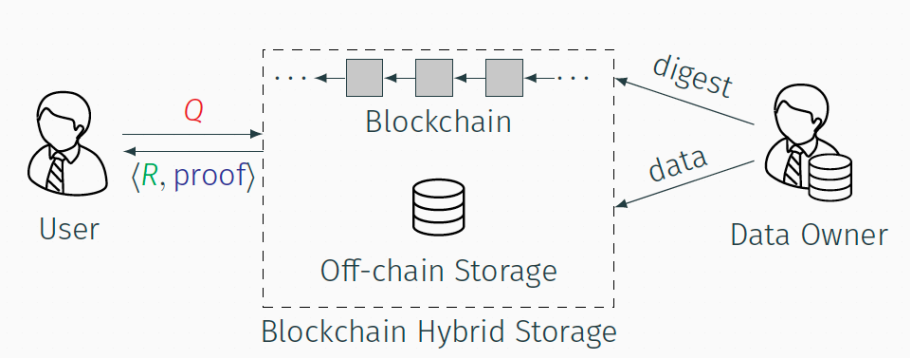
- Verifiable Computation (VC)
  - Computation is outsourced to an untrusted service provider
  - The service provider returns results with a cryptographic proof
  - Users verify the integrity of results using the proof
- Outsource queries to full nodes and verify the results using VC
  - General VC: Expressive but high overhead
  - Authenticated Data Structure (ADS)-based VC: Efficient but customized designs

Our Solutions

- vChain: Enabling Verifiable Boolean Range Queries over Blockchain Databases (**SIGMOD 2019**)



- GEM^2-Tree: Enabling Gas-Efficient Authenticated Range Queries in Hybrid-Storage Blockchain (**ICDE 2019**)



HKBU Database Group

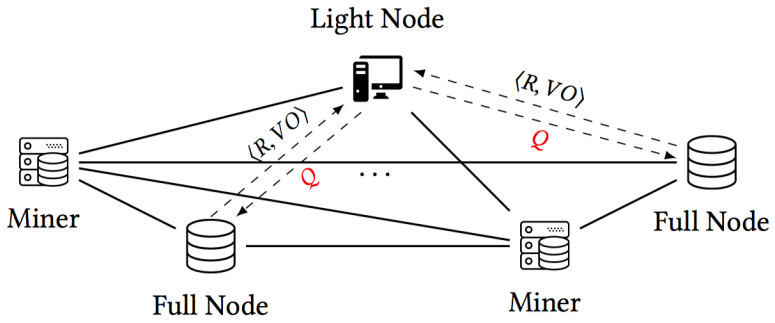
vChain: Enabling Verifiable Boolean Range Queries over Blockchain Databases

Cheng Xu, Ce Zhang, Jianliang Xu

ACM SIGMOD 2019

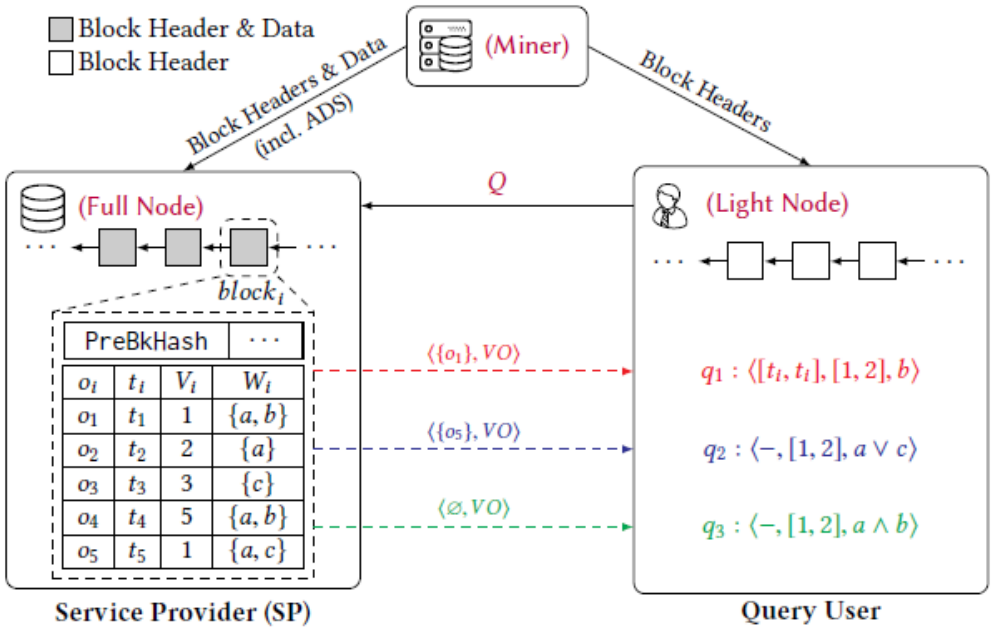
Problem Definition

- Problem: integrity-assured search over blockchain data
- System Model
  - Users become light nodes
  - Queries are outsourced to full nodes
- Full nodes not trusted
  - Program glitches
  - Security vulnerabilities
  - Commercial interest
  - …
- Security requirements:
  - Soundness: none of the objects returned as results have been tampered with and all of them satisfy the query conditions
  - Completeness: no valid result is missing



vChain – System Overview

- Miner: constructs each block with additional ADS to realize VC scheme
- Service Provider: is a full node and computes the results, as well as a verification object (VO)
- Query User: is a light node; uses the VO and block header to verify the results



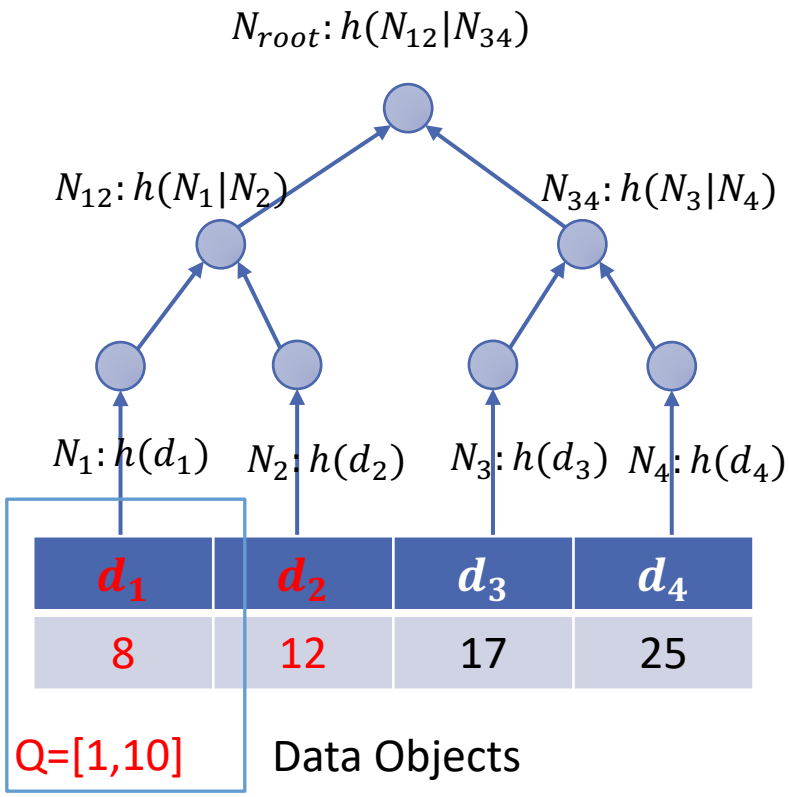
o_i	t_i	V_i	W_i
o_1	t_1	1	$\{a, b\}$
o_2	t_2	2	$\{a\}$
o_3	t_3	3	$\{c\}$
o_4	t_4	5	$\{a, b\}$
o_5	t_5	1	$\{a, c\}$

System Model of vChain

vChain – Data Model & Queries

- Data Model
  - Each block contains several temporal objects $\{o_1, o_2, \ldots, o_n\}$
  - o_i is represented by $\langle t_i, V_i, W_i \rangle$
    (*timestamp*, *multi-dimensional vector*, *set-valued attribute*)
- Boolean Range Queries
  - Bitcoin transaction:
    <timestamp, transfer amount, {"send address", "receive address"}>
    q =⟨[2019-05, 2019-06], [10, +∞], "send:1FFYc"∧"receive:2DAAf"⟩
  - Car rental transaction:
    <timestamp, rental price, {"type", "model"}>
    q = ⟨−, [200, 250], "Sedan"∧("Benz"∨"BMW")⟩

ADS: Merkle Hash Tree (MHT)



- Miners: construct the MHT and embed N_{root} into block header
- Full node -> Client:
  - Result: $\{8, d_1\}$
  - $VO = \{\{12, d_2\}, N_{34}\}$
- Client:
  - Retrieve N_{root} and verify soundness
  - Verify completeness

Limitations:

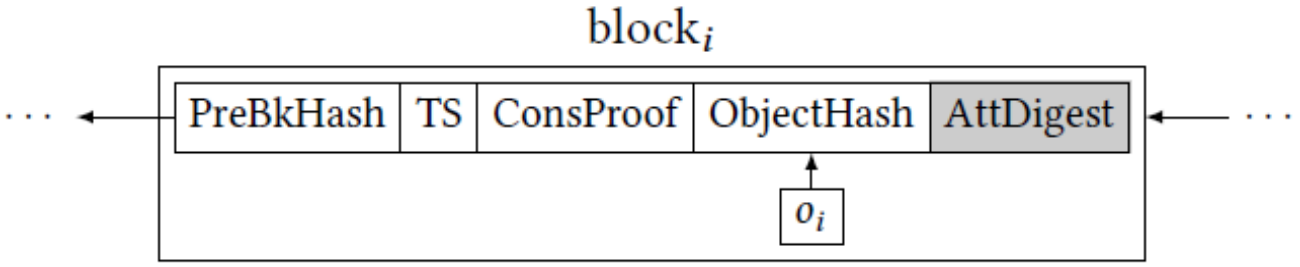
- An MHT supports only the query keys on which the Merkle tree is built
- MHTs do not work with set-valued attributes
- MHTs of different blocks cannot be aggregated

Cryptographic Building Block

- Cryptographic Multiset Accumulator
  - Map a multiset to an element in cyclic multiplicative group in a collision resistant fashion
  - Utility: prove set disjoint
  - Protocols:
    - KeyGen(1^{λ})$\rightarrow (sk, pk)$: generate keys
    - Setup(X, pk)$\rightarrow acc(X)$: return the accumulative value w.r.t. X
    - ProveDisjoint(X_1, X_2, pk)$\rightarrow \pi$: on input two multisets X_1 and X_2, where $X_1 \cap X_2 = \emptyset$, output a proof π
    - VerifyDisjoint($acc(X_1), acc(X_2), \pi, pk$)$\rightarrow \{0,1\}$: on input accumulative values $acc(X_1)$, $acc(X_2)$ and a proof π, output 1 if and only if $X_1 \cap X_2 = \emptyset$

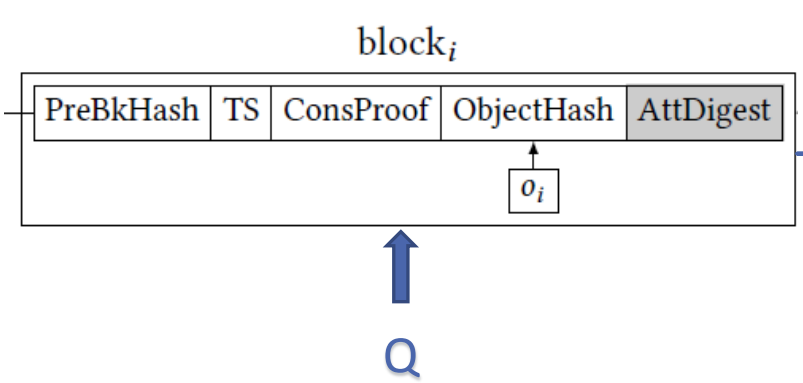
Basic Solution

- Consider *a single object* and *Boolean time-window query*
- Each block stores a single object $o_i = \langle t_i, W_i \rangle$
- ADS generation (Miner)
  - Extend the block header with *AttDigest*
  - $AttDigest = acc(W_i) = \text{Setup}(W_i, pk)$
    - Constant size regardless of number of elements in W_i
    - Support ProveDisjoint(·) & VerifyDisjoint(·)



Basic Solution

Verifiable Query:



Match
- Return o_i as a result
- Integrity is ensured by the *ObjectHash* in the block header

Mismatch
- Use *AttDigest* to prove the mismatch of o_i

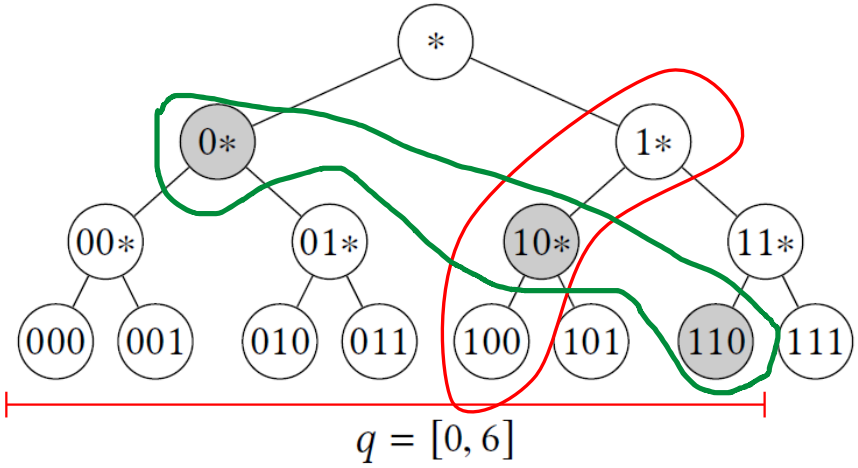
- Example of Mismatch:
  - Transform query condition to a list of sets
    - q="Sedan"∧ ("Benz"∨"BMW") -> {"Sedan"} , {"Benz", "BMW"}
  - Consider o_i: {"Van", "Benz"}
    - {"Sedan"}⋂{"Van", "Benz"} = ∅
  - Apply ProveDisjoint({"Van", "Benz"}, {"Sedan"}, pk) to generate proof π
  - User retrieves *AttDigest* = acc({"Van", "Benz"}) from the block header and use VerifyDisjoint(*AttDigest* , acc({"Sedan"}), π, pk) to verify the mismatch

Basic Solution

- Support time-window queries
  - Find the blocks whose timestamp is within the query window
  - Invoke previous algorithm for each object in theses blocks
- Example
  - Q = “Sedan”∧ (“Benz”∨“BMW”)
  - Objects within the time window
    - o_1: {"Sedan", "Benz"}, o_2: {"Sedan", "Audi"}, o_3: {"Van", "Benz"}
  - Query processing
    - o_1 is returned as a result
    - ProveDisjoint(·) is applied for o_2, o_3
      - Mismatch condition “Benz” ∨ “BMW” for o_2
      - Mismatch condition “Sedan” for o_3

Extension to Range Queries

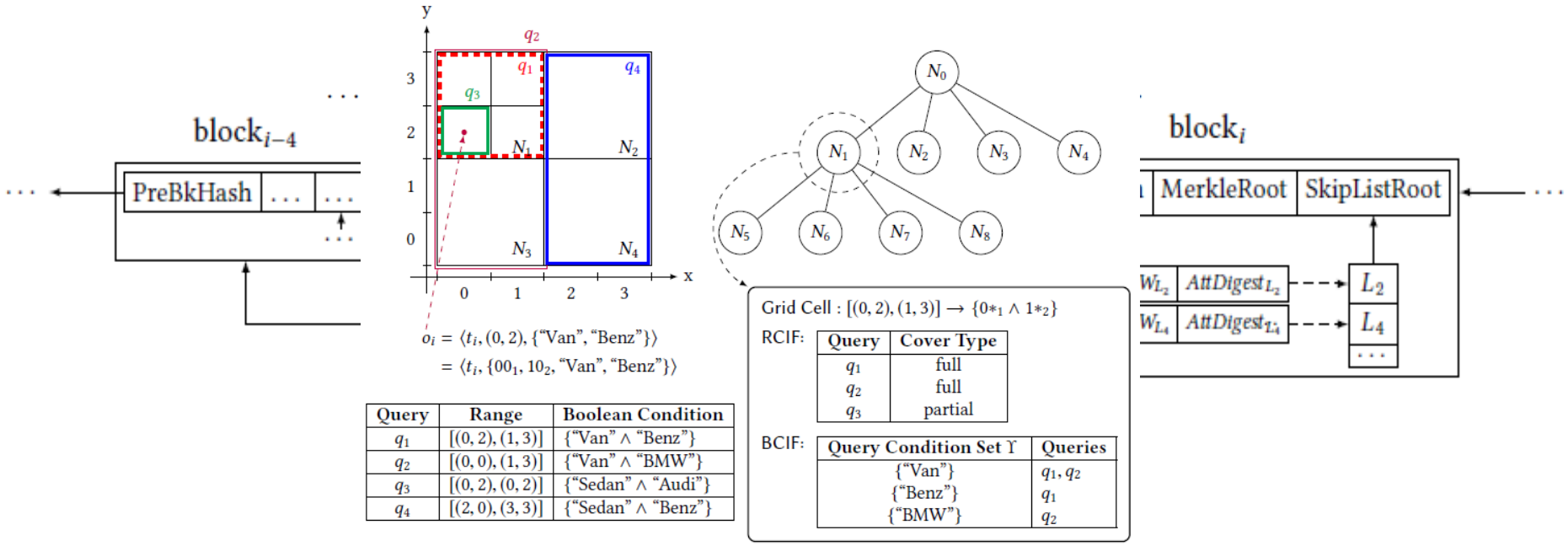
- Idea: transform numerical attributes into set-valued attributes
- Function $\mathrm{trans}(\cdot)$: transform a numerical value into a set of binary prefix elements
  - $\mathrm{trans}(4) = \{1*, 10*, 100\}$, $*$ denotes wildcard matching operator
- Range: the minimum set of tree nodes to cover the range



- [0, 6] -> $\{0*, 10*, 110\}$
- $4 \in [0,6] \rightarrow \{1*, 10*, 100\} \cap \{0*, 10*, 110\} = \{10*\} \neq \emptyset$

Batch Verification & Subscription Queries

- Observation: objects may share common attributes that mismatch the query condition
- Idea: we can aggregate them to speed up query processing
  - Intra-Block Index: aggregate objects inside same block using MHT
  - Inter-Block Index: aggregate objects across blocks using skip list
  - Inverted Prefix Tree: aggregate similar subscription queries from users



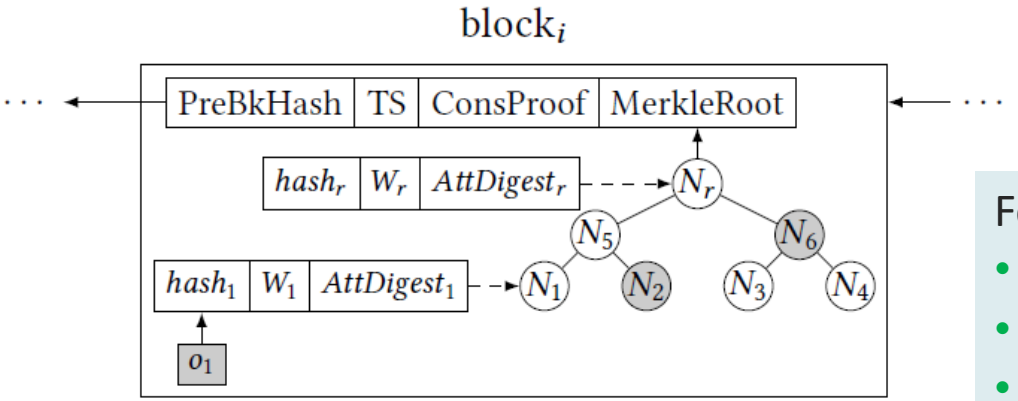
Query	Range	Boolean Condition
q_1	$[(0,2),(1,3)]$	{"Van" ∧ "Benz"}
q_2	$[(0,0),(1,3)]$	{"Van" ∧ "BMW"}
q_3	$[(0,2),(0,2)]$	{"Sedan" ∧ "Audi"}
q_4	$[(2,0),(3,3)]$	{"Sedan" ∧ "Benz"}

Query	Cover Type
q_1	full
q_2	full
q_3	partial

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

Batch Verification: Intra Index

- Each block stores multiple objects
- Two objects in a block may share a common attribute that mismatches the query condition
- Aggregate multiple objects using *intra-block MHT index*



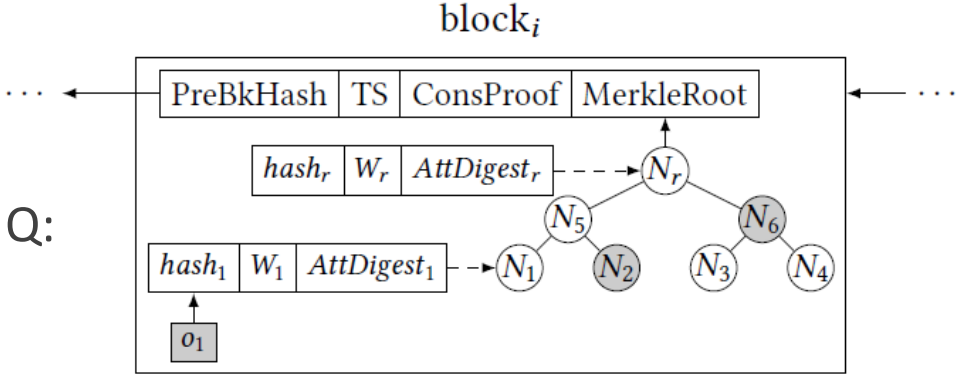
Node	Object	Set Attributes
N_1	o_1	W_1 = {"Sedan", "Benz"}
N_2	o_2	W_2 = {"Sedan", "Audi"}
N_3	o_3	W_3 = {"Van", "Benz"}
N_4	o_4	W_4 = {"Van", "BMW"}

For non-leaf node n:

- $W_n = W_{n_l} \cup W_{n_r}$
- $AttDigest_n = \text{acc}(W_n)$
- $hash_i = \text{hash}\left(\text{hash}\left(hash_{n_l} | hash_{n_r}\right) | AttDigest_n\right)$

Batch Verification: Intra Index

- Query Processing
  - Top-down traversal
  - If node multiset mismatches Q:
    - Compute the mismatch proof
  - Else
    - Continue searching subtrees



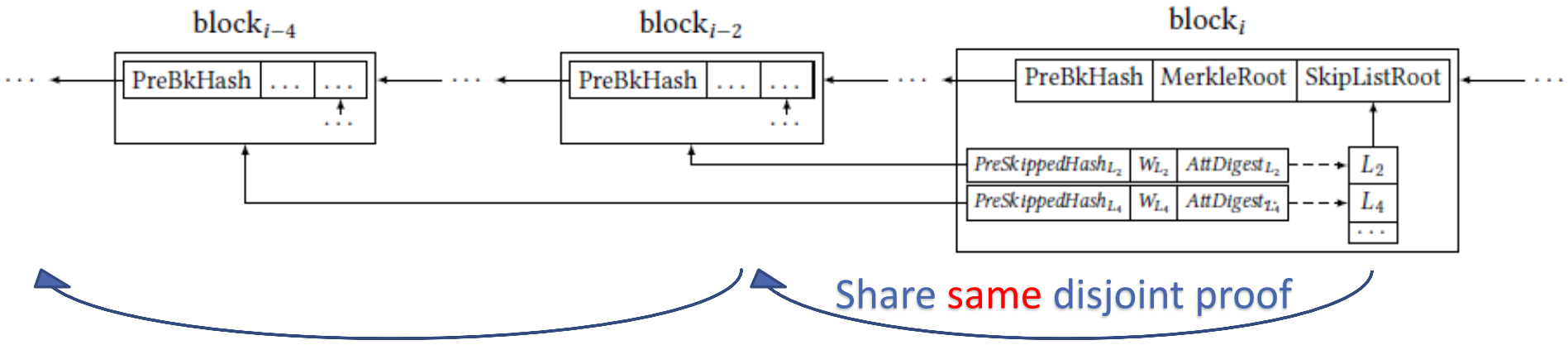
Node	Object	Set Attributes
N_1	o_1	W_1 = {"Sedan", "Benz"}
N_2	o_2	W_2 = {"Sedan", "Audi"}
N_3	o_3	W_3 = {"Van", "Benz"}
N_4	o_4	W_4 = {"Van", "BMW"}

Example

- Query: "Sedan" $\wedge$ ("Benz" $\vee$ "BMW") -> [{"Sedan"}, {"Benz", "BMW"}]
- ProofDisjoint() for N_6 since {"Sedan"} $\cap$ {"Van", "Benz", "BMW"} $= \emptyset$
- ProofDisjoint() for N_2 since {"Benz","BMW"}$\cap${"Sedan", "Audi"} $= \emptyset$
- Object in N_1 is a result
- Client verifies proofs and reconstruct MerkleRoot using VO

Batch Verification: Inter Index

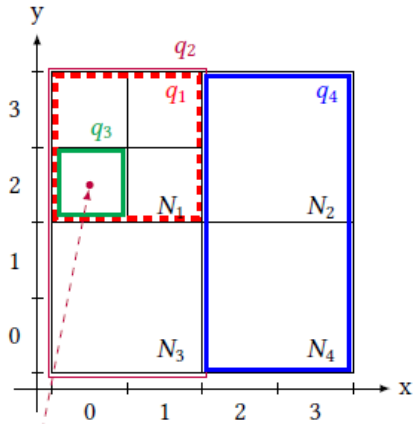
- Objects *across blocks* may share same attributes
- Employ skip list including multiple skip jumps
- Skip multiple blocks that mismatch the query condition



Iterate the skip list from the maximum jump
If W_{L_i} matches q, process next L
Else jump to the L_i-th previous block

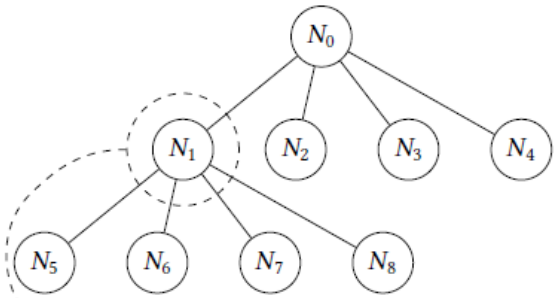
Verifiable Subscription Queries

- Observation: A mismatched object can have the same reason of mismatching for different subscription queries
- Inverted Prefix Tree (IP-Tree)



$o_i = \langle t_i, (0, 2), \{\text{“Van”}, \text{“Benz”}\}\rangle$
$= \langle t_i, \{00_1, 10_2, \text{“Van”}, \text{“Benz”}\}\rangle$

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



Grid Cell : $[(0, 2), (1, 3)] \rightarrow \{0*_1 \wedge 1*_2\}$

RCIF:

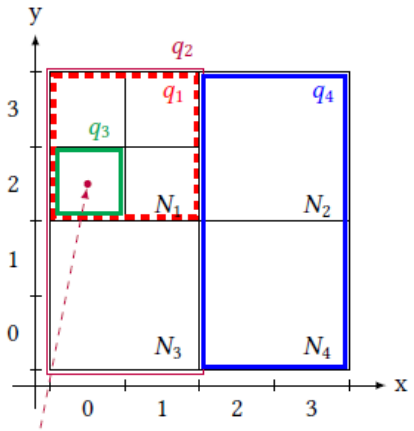
Query	Cover Type
q_1	full
q_2	full
q_3	partial

BCIF:

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

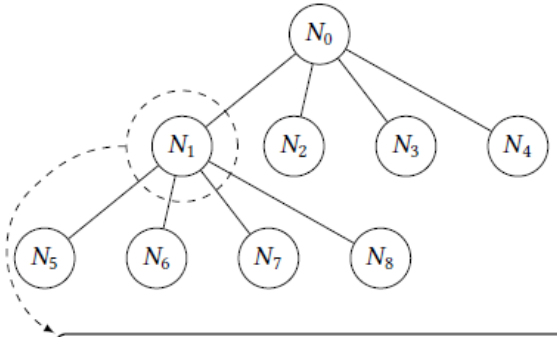
- Grid tree on numerical attributes
- Range Condition Inverted File (RCIF)
  - <query, cover type(full/partial)>
- Boolean Condition Inverted File (BCIF) for ‘full’ queries
  - <query condition set, queries>

Verifiable Subscription Queries



$o_i = \langle t_i, (0,2), \{\text{“Van”}, \text{“Benz”}\}\rangle$
$= \langle t_i, \{00_1, 10_2, \text{“Van”}, \text{“Benz”}\}\rangle$

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



Grid Cell : $[(0,2),(1,3)] \rightarrow \{0*_1 \wedge 1*_2\}$

RCIF:

Query	Cover Type
q_1	full
q_2	full
q_3	partial

BCIF:

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

- Traverse the IP-Tree top-down
  - ProveDisjoint for q_4 (mismatch range condition)
  - q_1 is a result, ProveDisjoint for q_2 (mismatch “BMW”)
  - ProveDisjoint for q_3 (mismatch “Sedan”)

HKBU Database Group

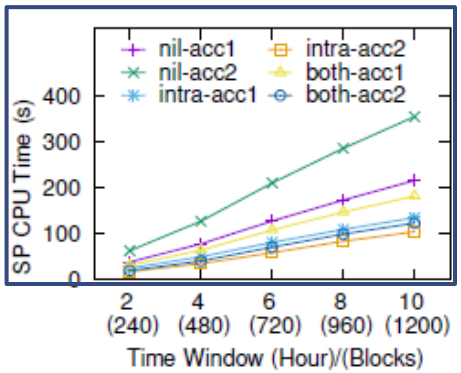
Performance Evaluation

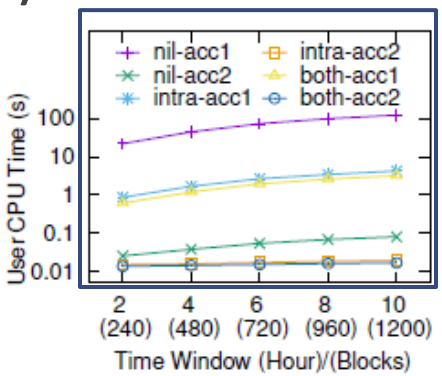
- Evaluation metrics:
  - Query processing cost in terms of SP CPU time
  - Query verification cost in terms of user CPU time
  - Size of the VO transmitted from the SP to the user
- Datasets: 4SQ, WX, ETH
- Numerical range selectivity:
  - 10% for 4SQ and WX
  - 50% for ETH
- Disjunctive Boolean function size:
  - 3 for 4SQ and WX
  - 9 for ETH

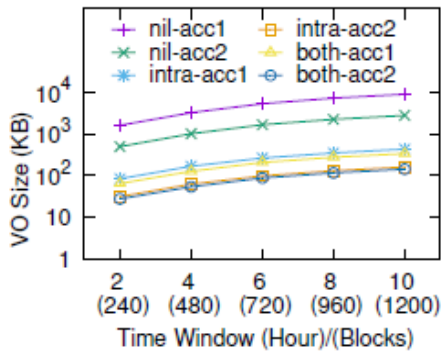
Performance Evaluation

- Time-Window Query Performance

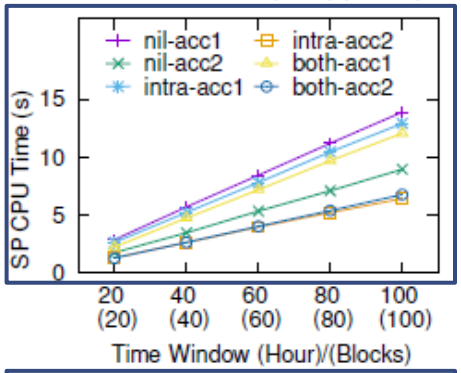
4SQ

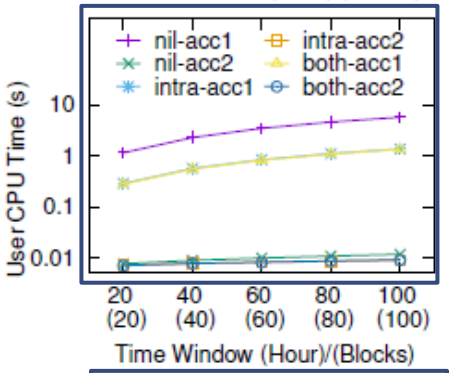


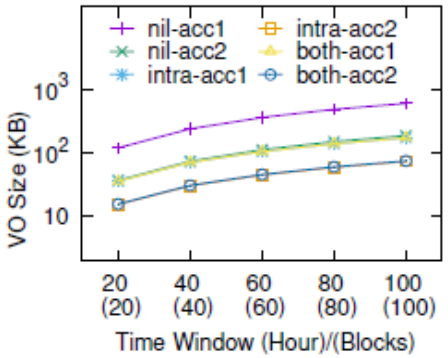




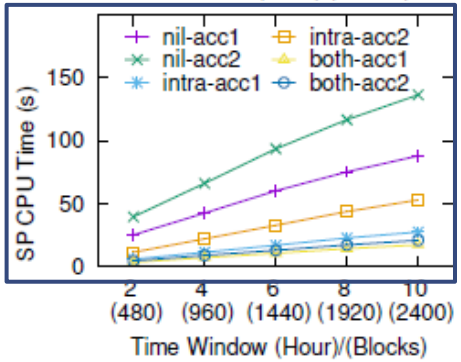
WX

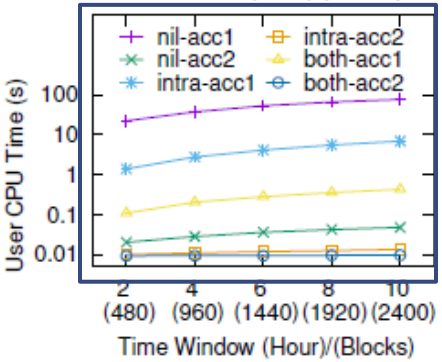


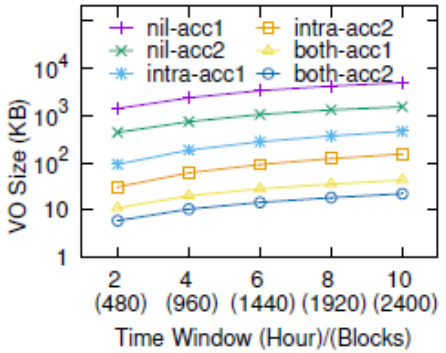




ETH



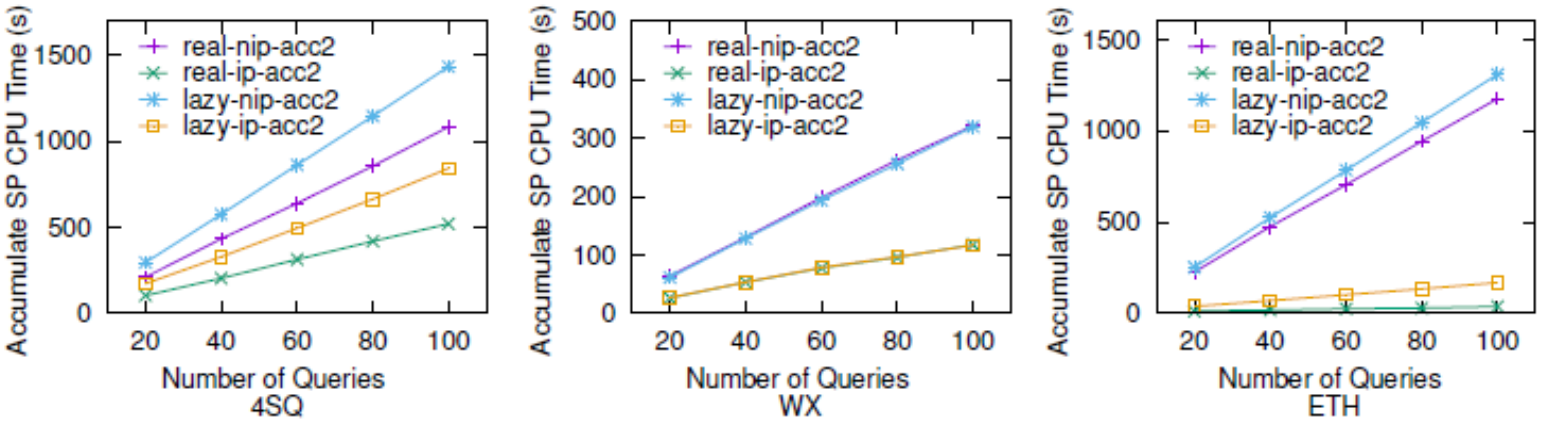




Performance Evaluation

HKBU Database Group

- Subscription Query Performance
  - With or without IP-Tree



The IP-Tree reduces the SP's overhead by at least 50% in all cases tested

HKBU Database Group

GEM^2-Tree: A Gas-Efficient Structure for Authenticated Range Queries in Blockchain

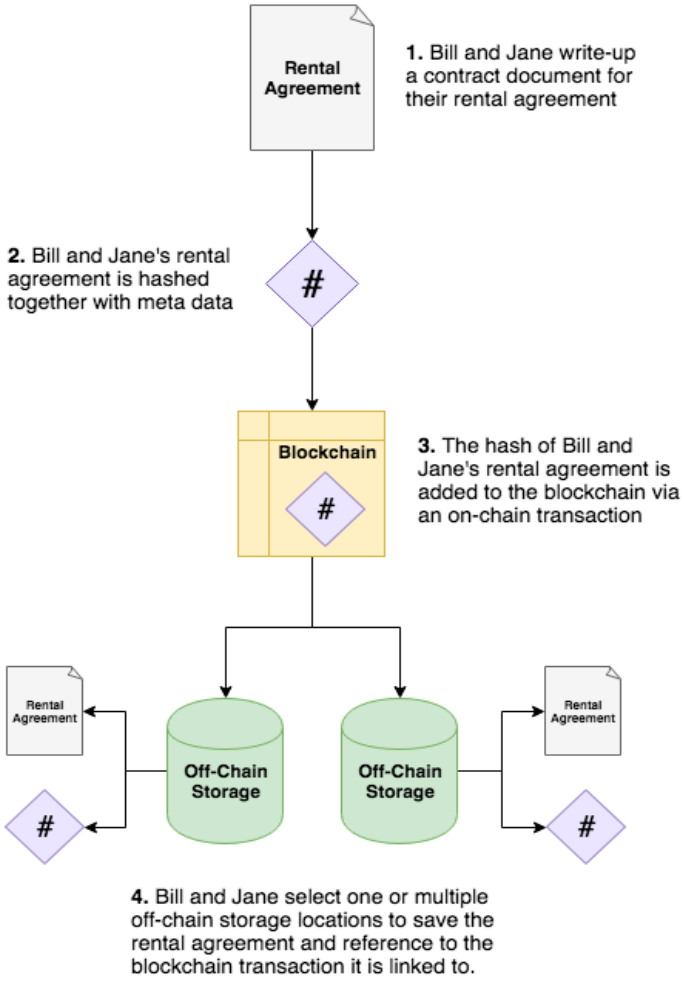
Ce Zhang, Cheng Xu, Jianliang Xu, Yuzhe Tang, Byron Choi

IEEE ICDE 2019

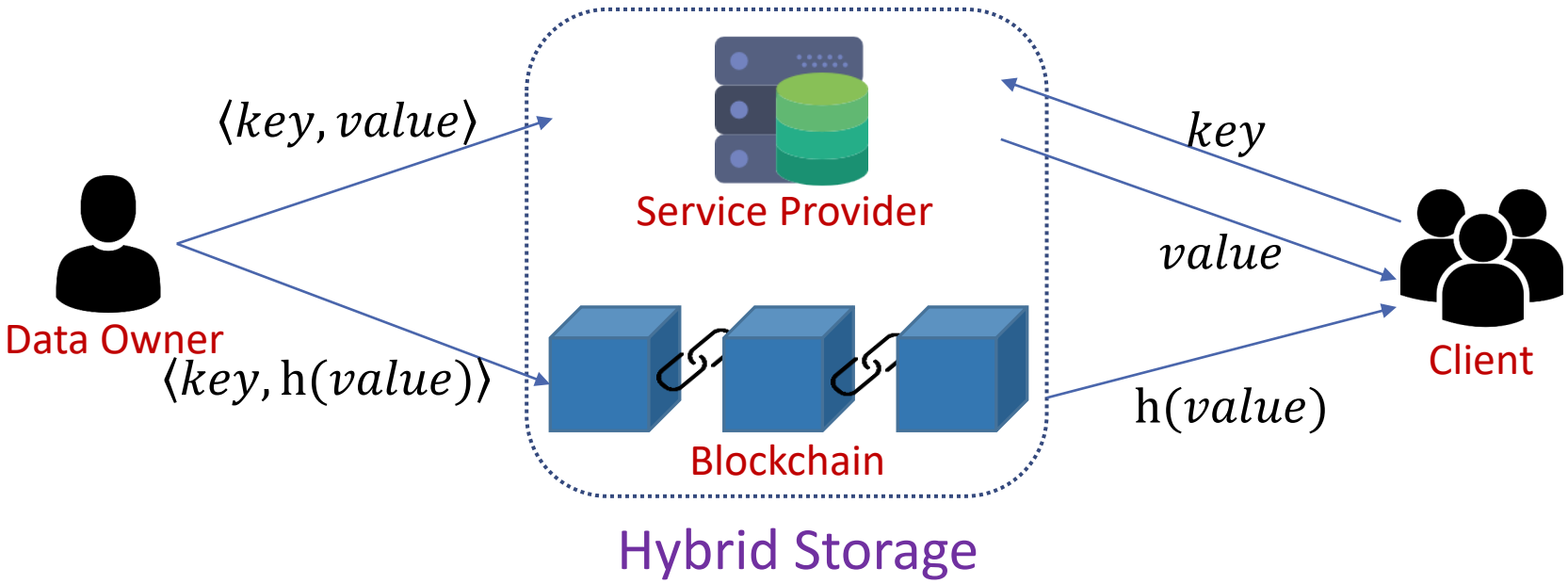
Blockchain Scalability

- Storing *any* information on chain is not scalable
  - Large size: document, image, etc.
  - 500KB per TX x 500 TX per sec => 2 Gb per sec => 8,000 TB annually
- Off-chain storage:
  - Raw data is stored outside of the blockchain
  - A hash of the data is keep on chain to ensure integrity

Example: BACK ALLEY CODER

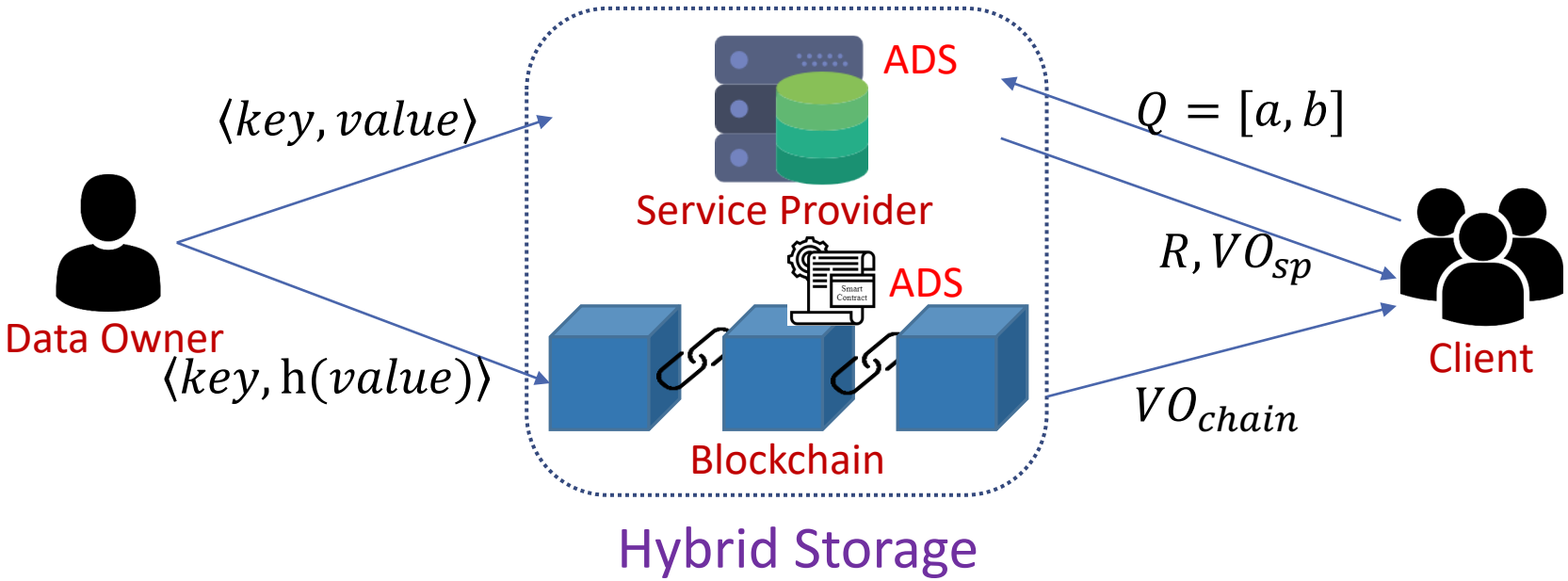


Blockchain Hybrid Storage



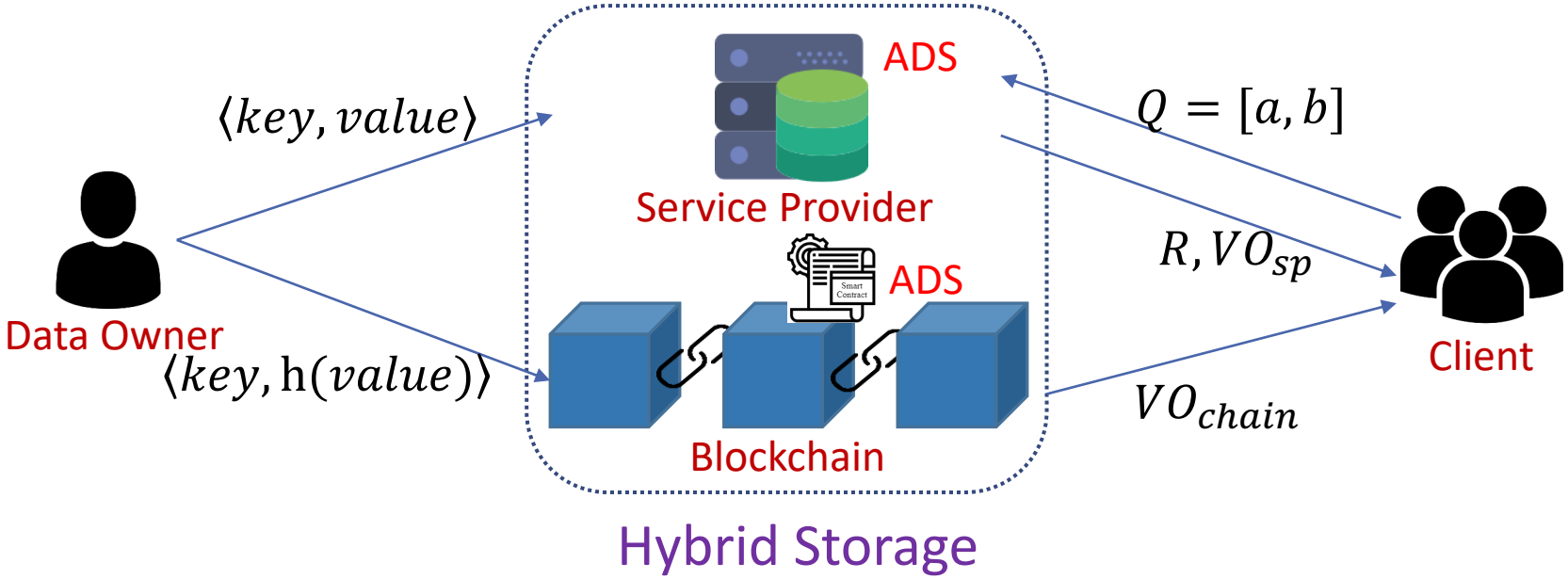
- Pros: high scalability, integrity assured
- Con: only support exact search
- More general queries?

Objective



- Support integrity-assured range queries
- Inspiration: authenticated query processing
  - Use the *authenticated data structure* (ADS) to support queries
  - Leverage both smart contract and the SP to maintain the ADS

System Overview



- Data Owner: send meta-data to blockchain and full data to the SP
- Smart Contract: update on-chain ADS
- Service Provider: maintain the same ADS and process queries
- Client: verify results with respect to the ADS from the blockchain

Challenge

- Each on-chain update requires a smart contract transaction
- Transaction fee for smart contract execution
  - Modeled by *gas* for storage and computation (Ethereum)
- Problem: How to design efficient ADS to be maintained by smart contract under the gas cost model

Ethereum Gas Cost Model

Operation	Gas Used	Explanation
C_{sload}	200	load a word from storage
C_{sstore}	20,000	store a word to storage
$C_{supdate}$	5,000	update a word to storage
C_{mem}	3	access a word in memory
C_{hash}	$30 + 6 \cdot \lvert words \rvert$	hash an arbitrary-length data

HKBU Database Group

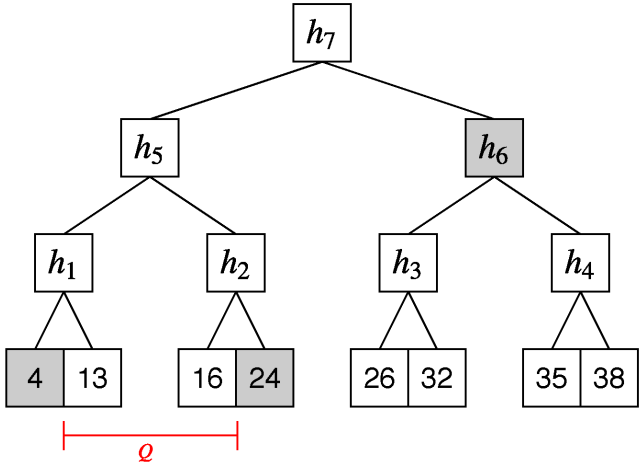
Contributions

- A novel Gas−Efficient Merkle Merge Tree (GEM^2-Tree)
  - Reduce the storage and computation cost of the smart contract
- Optimized version GEM^{2*}-Tree
  - Further reduce the maintenance cost without sacrificing much of the query performance

Preliminaries

- Authenticated Query Processing
  - The DO outsources the authenticated data structure (ADS) to the SP
  - The SP returns results and *verification object* (VO)
  - The client verifies the result using VO

- ADS: Merkle Hash Tree (MHT)
  - Binary tree
  - Hash function combining the child nodes
  - VO: sibling hashes along the search path
  - Verification: reconstructing the root hash



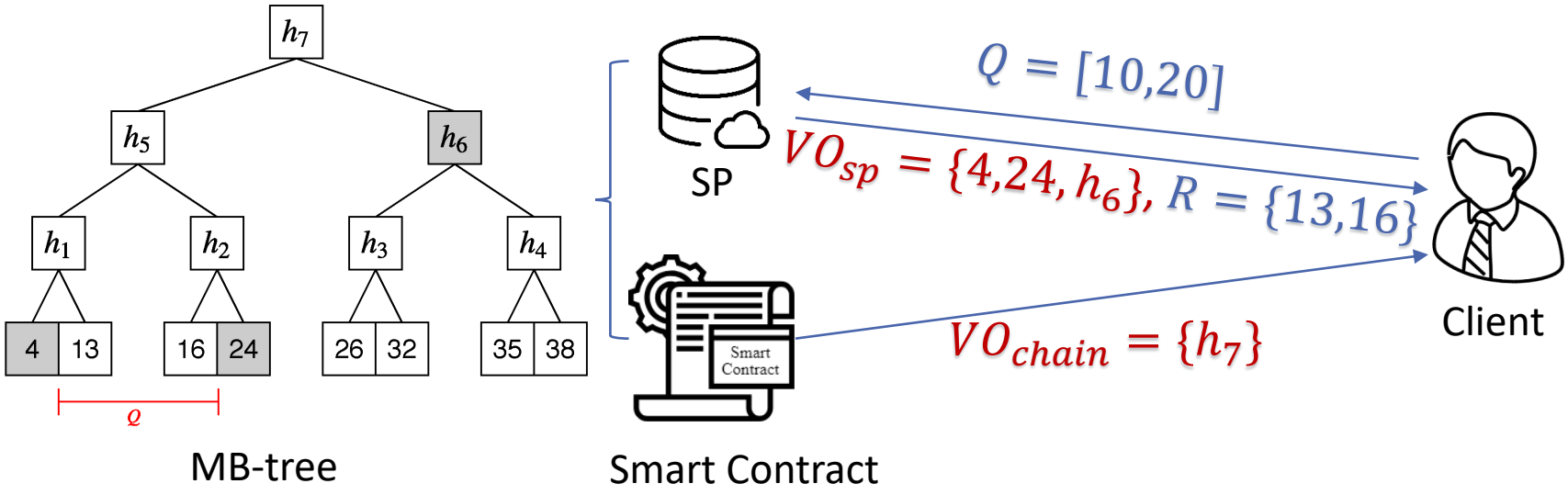
Result: {13,16}

VO: {4, 24, h_6}

- Merkle B-Tree (MB-Tree)
  - Integrate B-tree with MHT

Baseline Solution (1)

- MB-tree
  - Maintained by both the smart contract and the SP
  - Data update requires writes on the entire tree path
  - $C^{\text{insert}}_{\text{MB-tree}} = \log_F N \left(2C_{sstore} + 2C_{supdate} + (2F+1)C_{sload} + C_{hash}\right) + C_{sstore}$



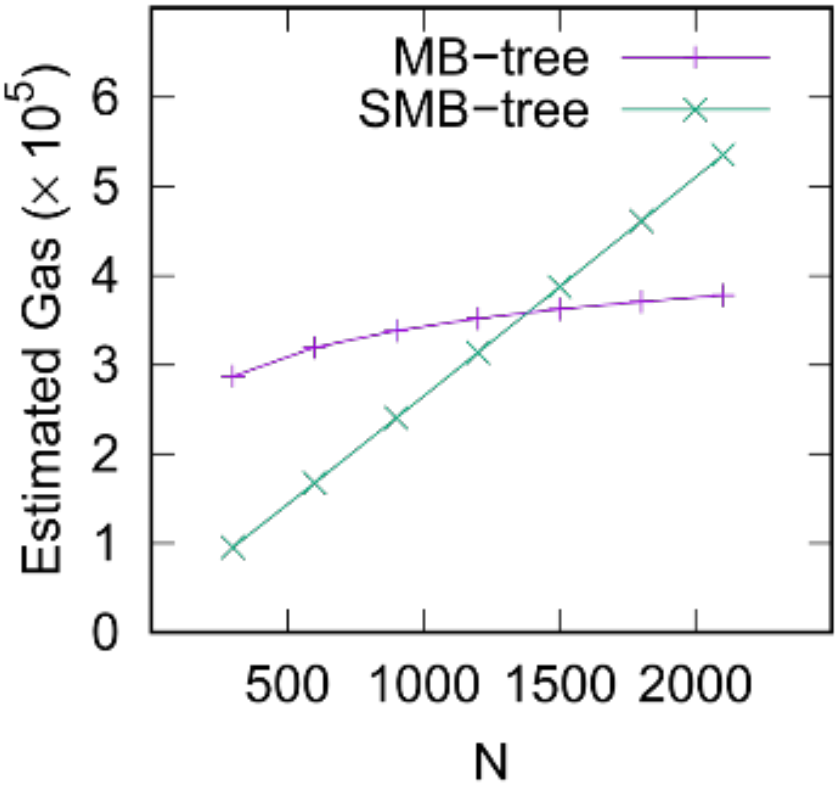
Baseline Solution (2)

- Suppressed Merkle B-tree (SMB-tree)
- Observation of MB-tree: only root hash VO_{chain} is used during query processing

- Idea:
  - Suppress all internal nodes and only materialize the root node in the blockchain
  - The smart contract computes all nodes of the SMB-tree on the fly and updates the root hash to the blockchain storage
  - The SMB-tree in the SP keeps the complete structure (to retain the query performance)

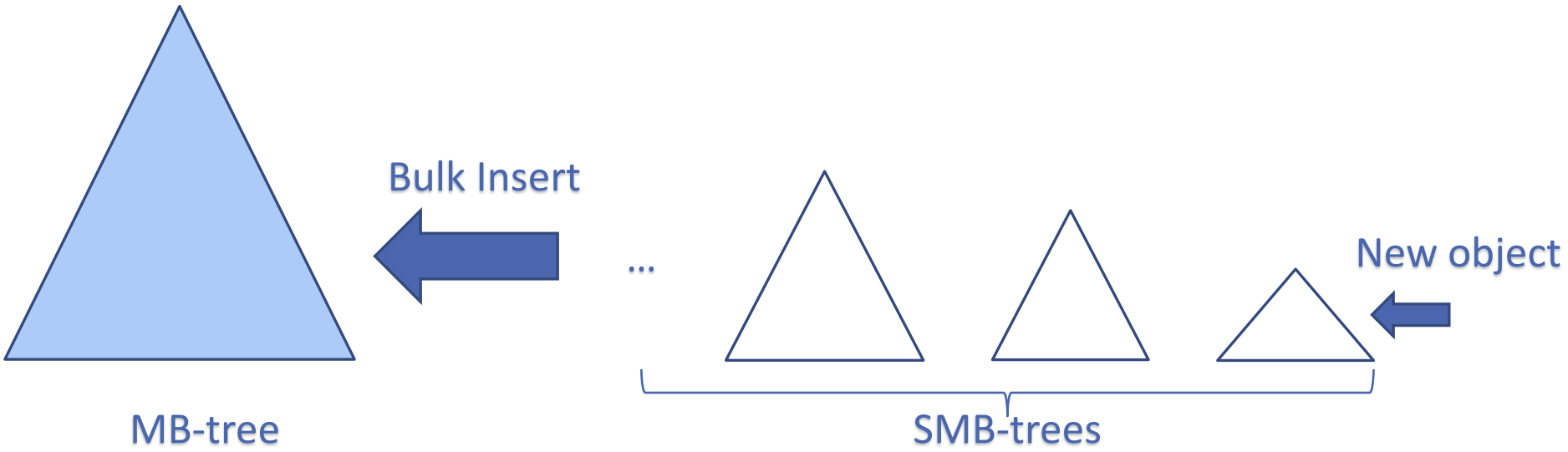
- $C_{\text{SMB-tree}}^{\text{insert}} = N\left(C_{sload} + \log N \cdot C_{mem} + \frac{1}{F}C_{hash}\right) + C_{sstore} + C_{supdate}$

MB-tree vs SMB-tree

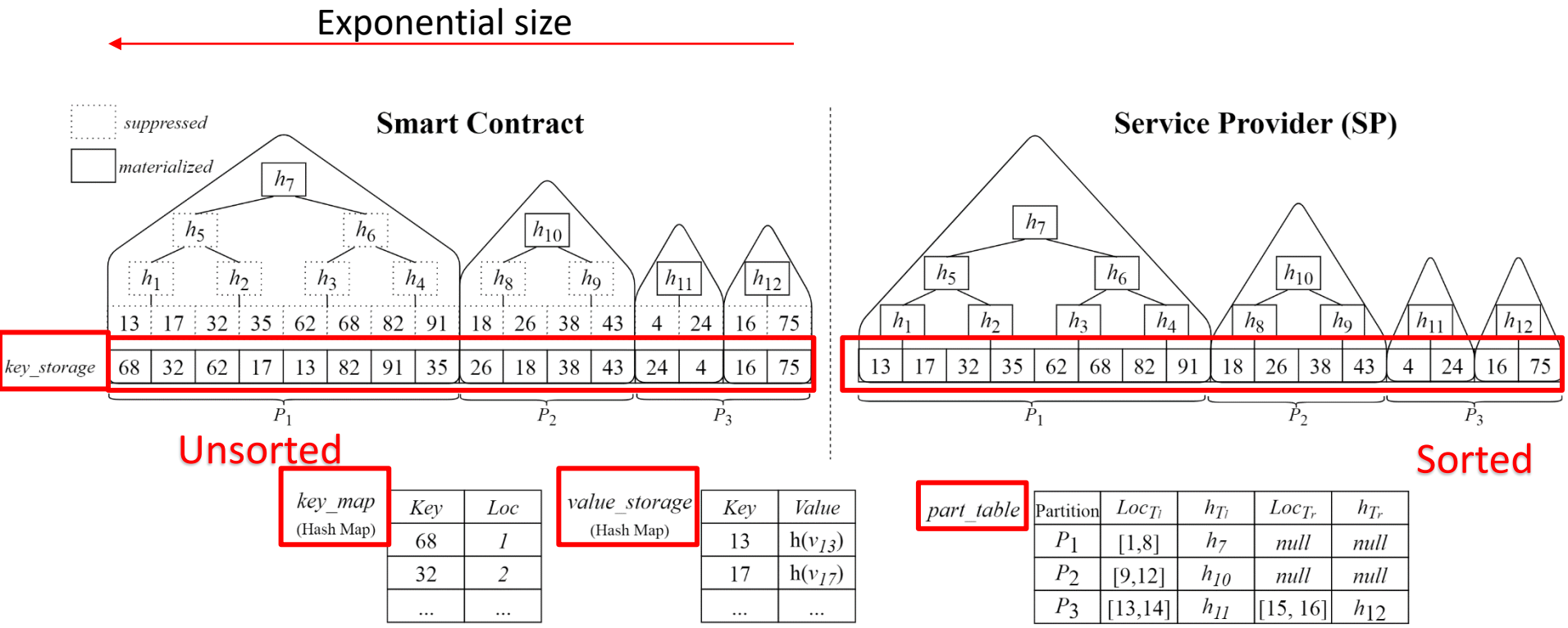


Gas-Efficient Merkle Merge Tree (GEM^2-Tree)

- Maintain multiple separate structures
  - A series of small SMB-trees: index newly inserted objects
  - A full materialized MB-tree: merge the objects of the largest SMB-trees in batch



An Example



key_map (Hash Map)	Key	Loc
	68	*1*
	32	*2*
	...	...

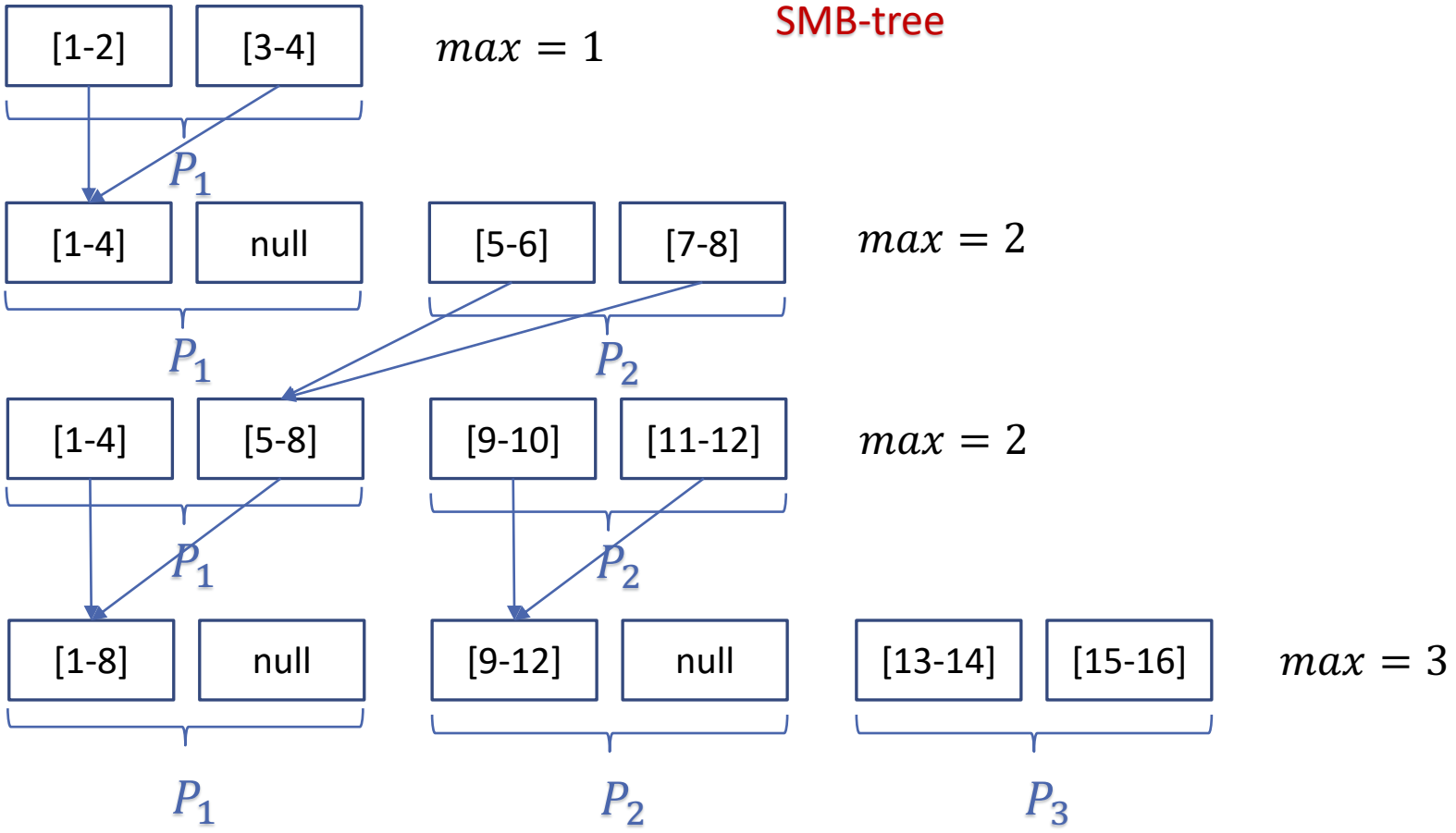
value_storage (Hash Map)	Key	Value
	13	h(v_{13})
	17	h(v_{17})
	...	...

part_table	Partition	Loc_{T_l}	h_{T_l}	Loc_{T_r}	h_{T_r}
	P_1	[1,8]	h_7	*null*	*null*
	P_2	[9,12]	h_{10}	*null*	*null*
	P_3	[13,14]	h_{11}	[15, 16]	h_{12}

- Exponentially-sized partition space: each contains 1 or 2 SMB-trees
  - Partition table stores location range and root hash values
  - Key_map stores the key with the storage location (used in update operation)

Insertion

- Example ($M = 2$)



- If P_{max} is not full, insert object to P_{max};
- Else merge the two SMB-trees to a bigger SMB-tree

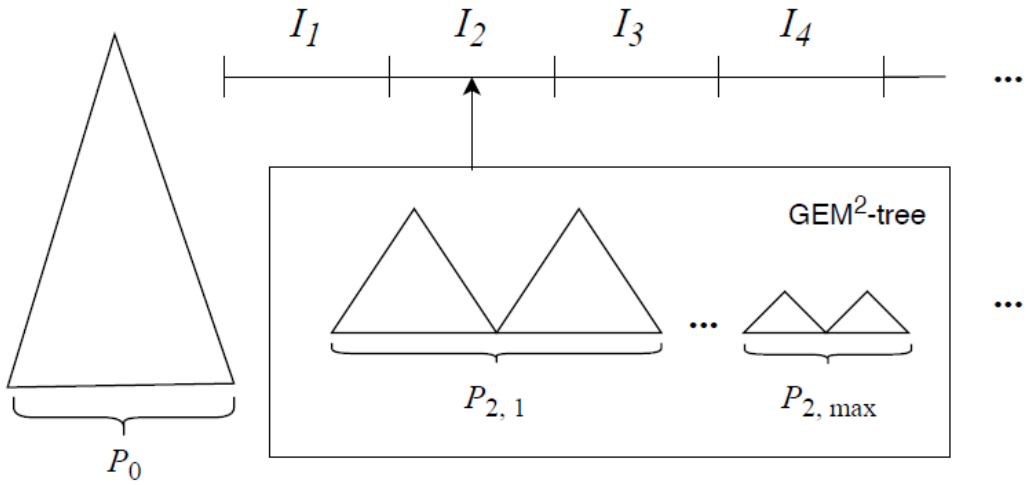
HKBU Database Group

Update and Query Processing

- Updating
  - Observation: storage location of each search key is fixed (*key_map*)
  - The GEM^2-tree structure remains unchanged
  - Update the value of an existing key with a new value
  - Recompute the root hash of the MB-tree or SMB-tree
- Authenticated query processing
  - The SP traverses the MB-tree and multiple SMB-trees
  - Process the range query on them individually
  - Combines the results and VO for each of these trees
  - The client uses the VO and results for each of these trees

Optimized GEM^2–Tree

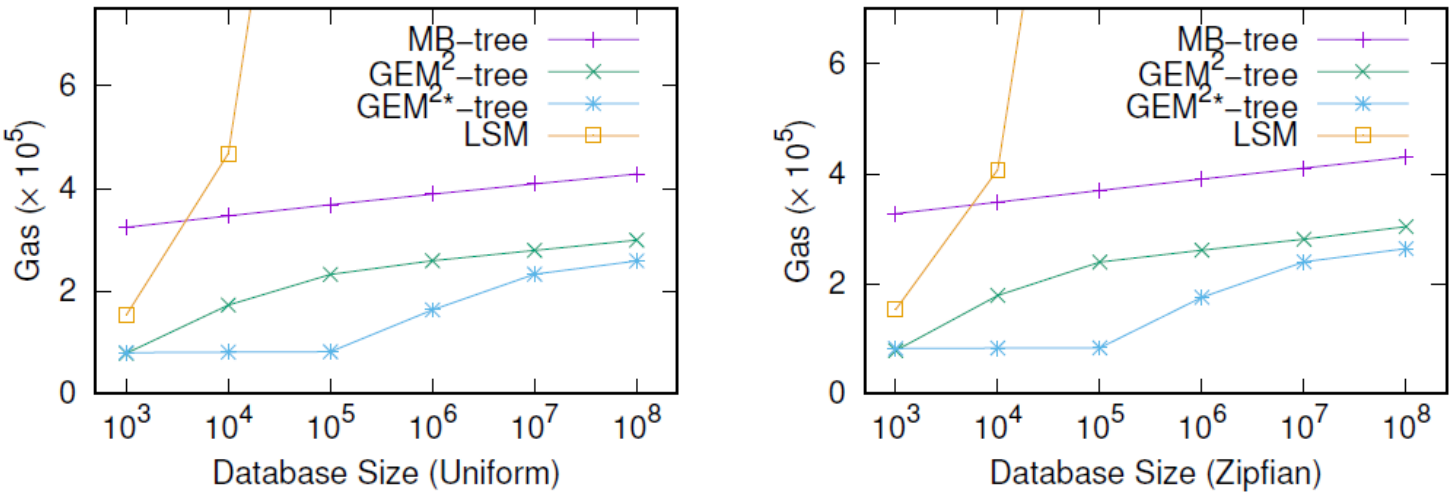
- GEM^{2*}-tree: to further reduce the gas consumption without sacrificing much of the query overhead
- Two-level index structure
  - Upper level: split the search key domain into several regions
  - Lower level: a GEM^2-tree is built for each region I_i
  - Only one single MB-tree for the entire GEM^{2*}-tree



Performance Evaluation

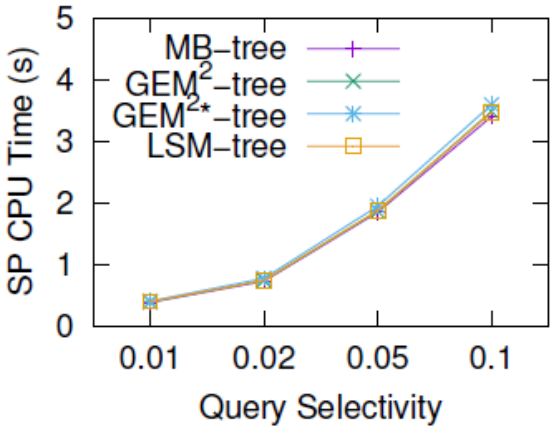
- Dataset:
  - Synthetic data generated by Yahoo Cloud System Benchmark (YCSB)
  - Cardinality: 100M
  - Key size: 4 bytes
  - Key distribution: uniform/zipfian
- Index parameters
  - Maximum size of the smallest SMB-tree, $M = 8$ (word size is 32 bytes and search key 4 bytes)
  - Fan-out of the MB-tree is set to 4 according to the word size 32 bytes
    - $(f-1)l_d + fl_p < 32\text{byte}$
  - $S_{max} = 2{,}048$ based on the cost analysis of MB-tree and SMB-tree
  - Search key domain is split into 100 regions for upper GEM^{2*}-tree

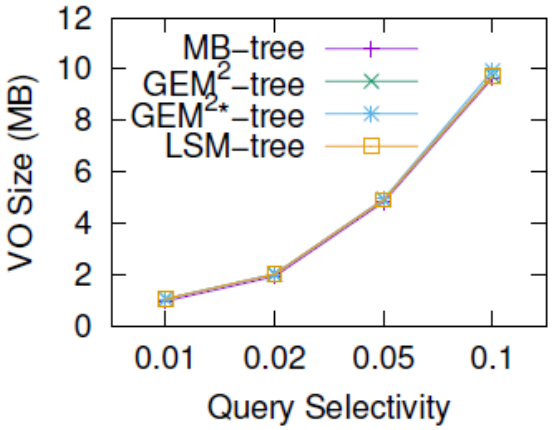
Gas Consumption vs Database Size

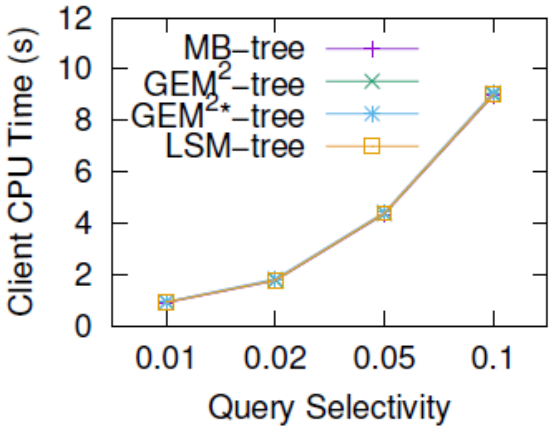


- LSM-tree is able to support the database up to 10,000
  - Merge cost grows exponentially with level increasing
- Gas reduction of the two proposed indexes
- Optimization is better
  - More SMB-trees; efficient bulk insertion thanks to the upper level

Authenticated Query Performance



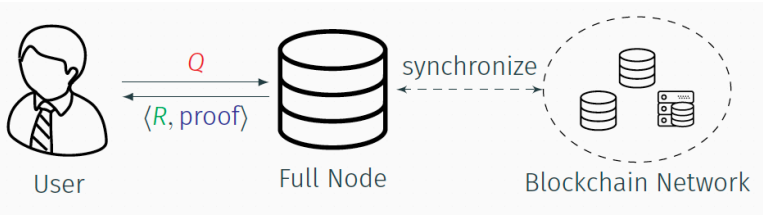




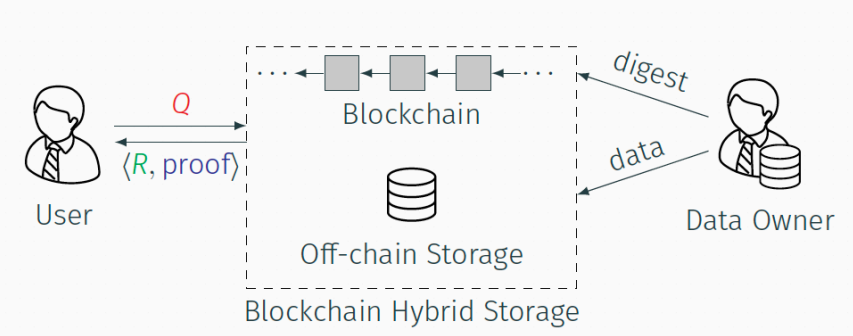
- Compared with the MB-tree, the GEM^2-tree retains the query performance
- GEM^{2*}-tree is slightly worse when the query range is large
- Reduce the gas cost with little penalty on the query performance

Summary

- Searchable blockchain meets the increasing demand of data search
- Two ADS solutions towards searchable blockchain
  - **vChain**: integrity-assured Boolean range search in blockchain databases
  - **$\mathbf{GEM^2}$-tree**: integrity-assured range search in blockchains with hybrid storage



vChain



$\mathbf{GEM^2}$-tree

Future Work

- Extended to more query types
  - Top-k, kNN, skyline, similarity queries
  - Blockchain-based knowledge graphs
- Search on encrypted blockchain data
  - GAS-based performance model
- Privacy-preserving query processing against smart contracts
- Data sharing with fine-grained access control

Thank You !