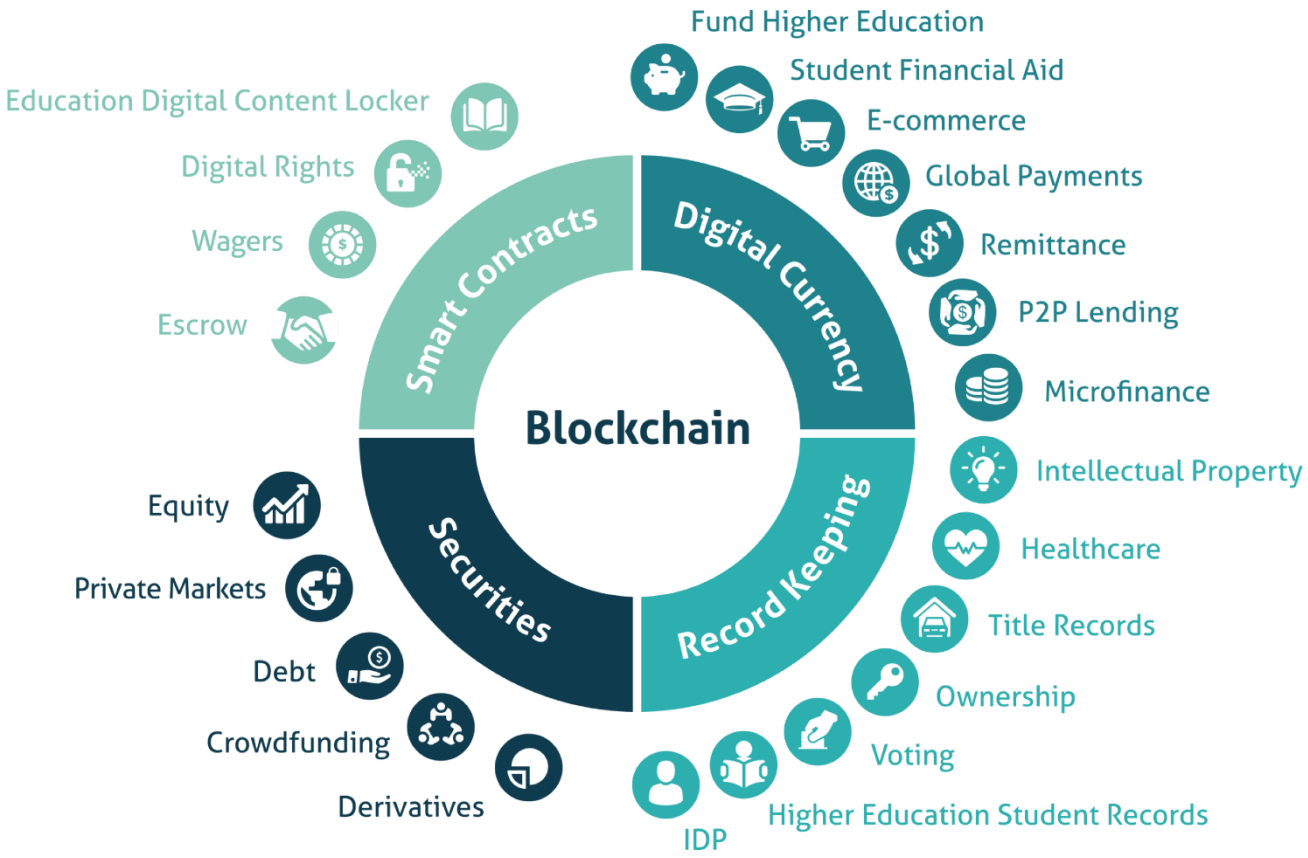
$\mathbf{GEM^2}$-Tree: A Gas-Efficient Structure for Authenticated Range Queries in Blockchain

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

Hong Kong Baptist University, Hong Kong

Syracuse University, NY, USA

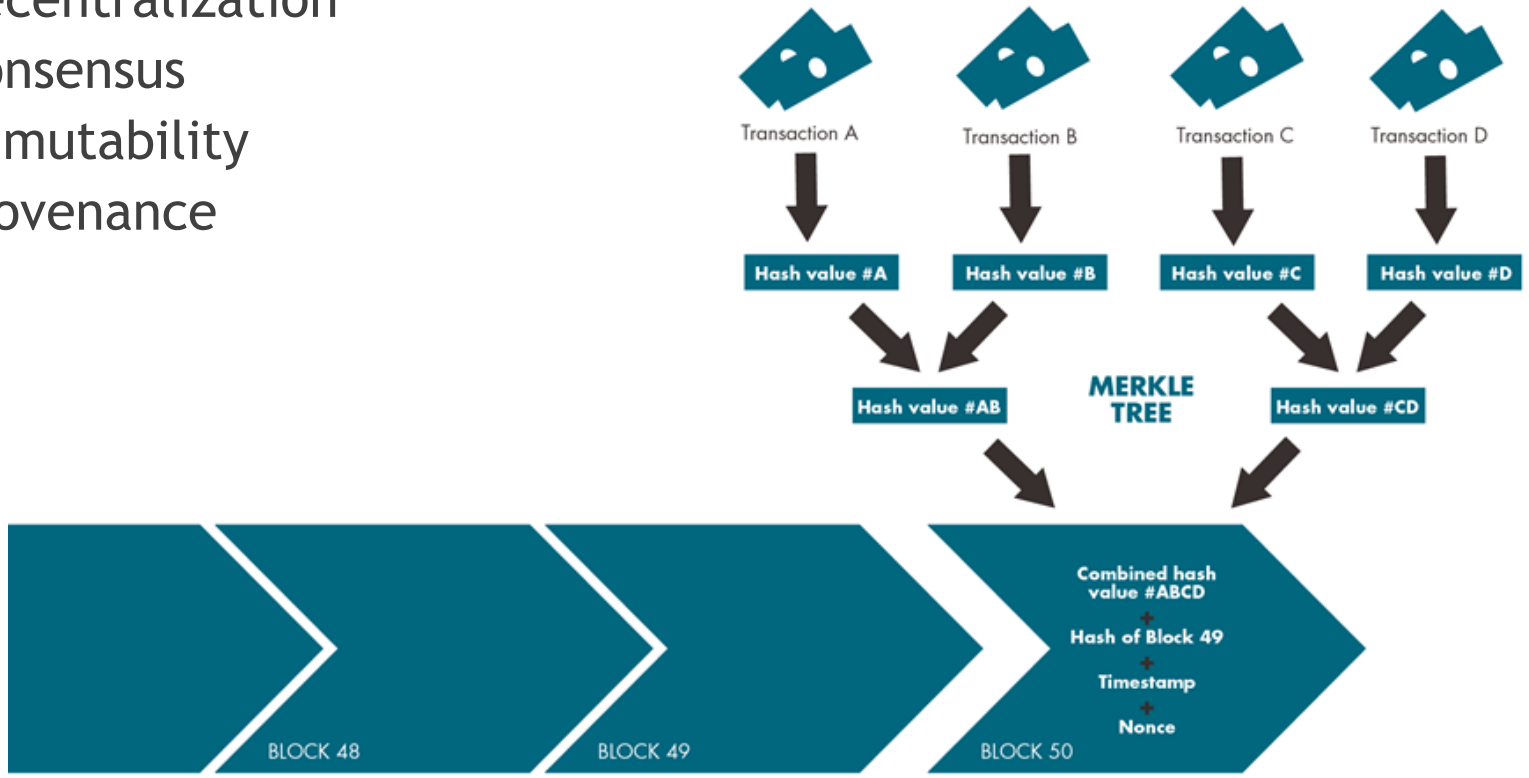
Introduction



Source: [FAHM Technology Partners]

Blockchain Technology

- Distributed Ledger maintained by a community of (untrusted) users
  - Decentralization
  - Consensus
  - Immutability
  - Provenance



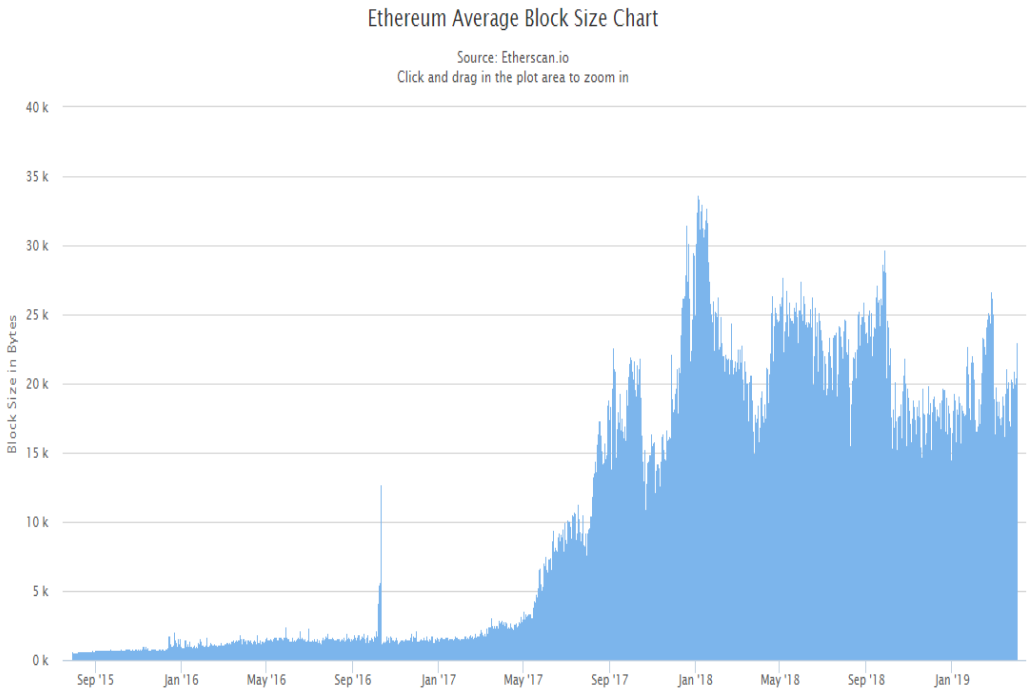
Smart Contract

- A trusted program to execute user-defined computation upon the blockchain
  - Read and write blockchain data
  - Execution integrity is ensured by the consensus protocol
- Offer trusted storage and computation capabilities
- Function as a trusted virtual machine

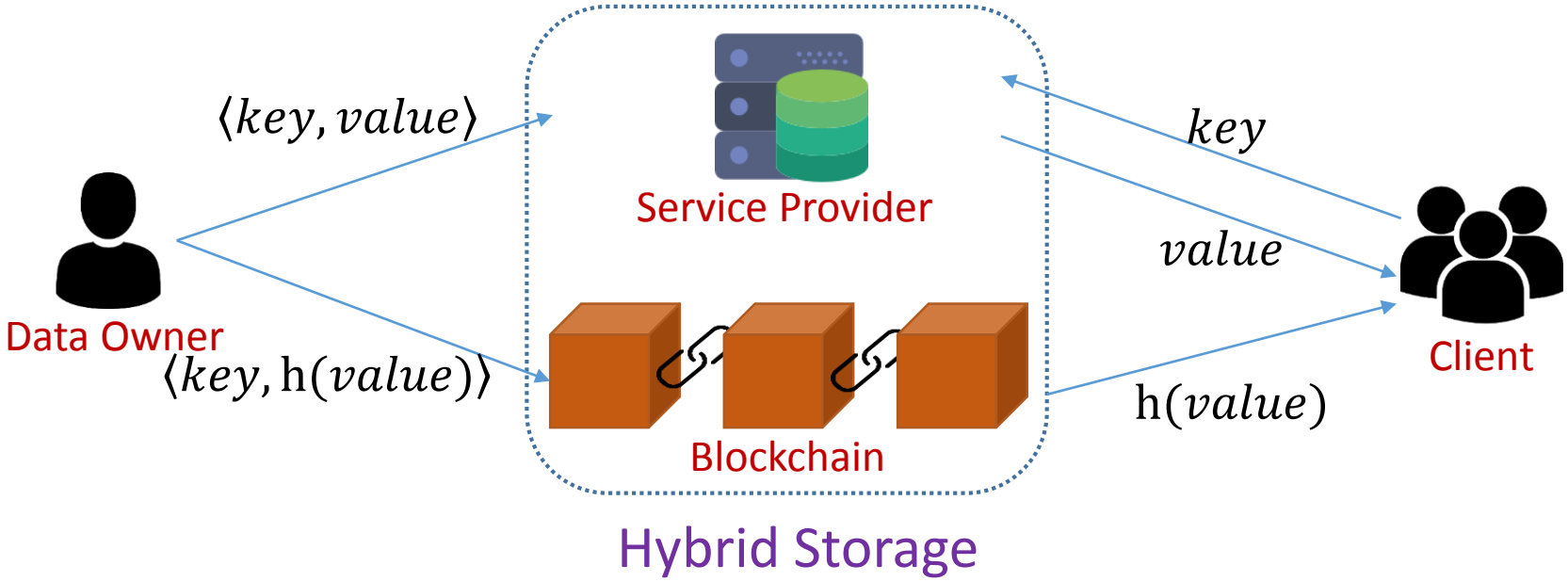
	Traditional Computer	Blockchain VM
Storage	RAM	Blockchain
Computation	CPU	Smart Contract

Blockchain Scalability

- Scalability problem
  - Storing *any* information on chain is not scalable
  - Large size data: document, image, etc.
  - Ethereum: block size 20KB, 15 sec per block
- Off-chain storage
  - Raw data is stored outside of the blockchain
  - A hash of the data is kept on chain to ensure integrity

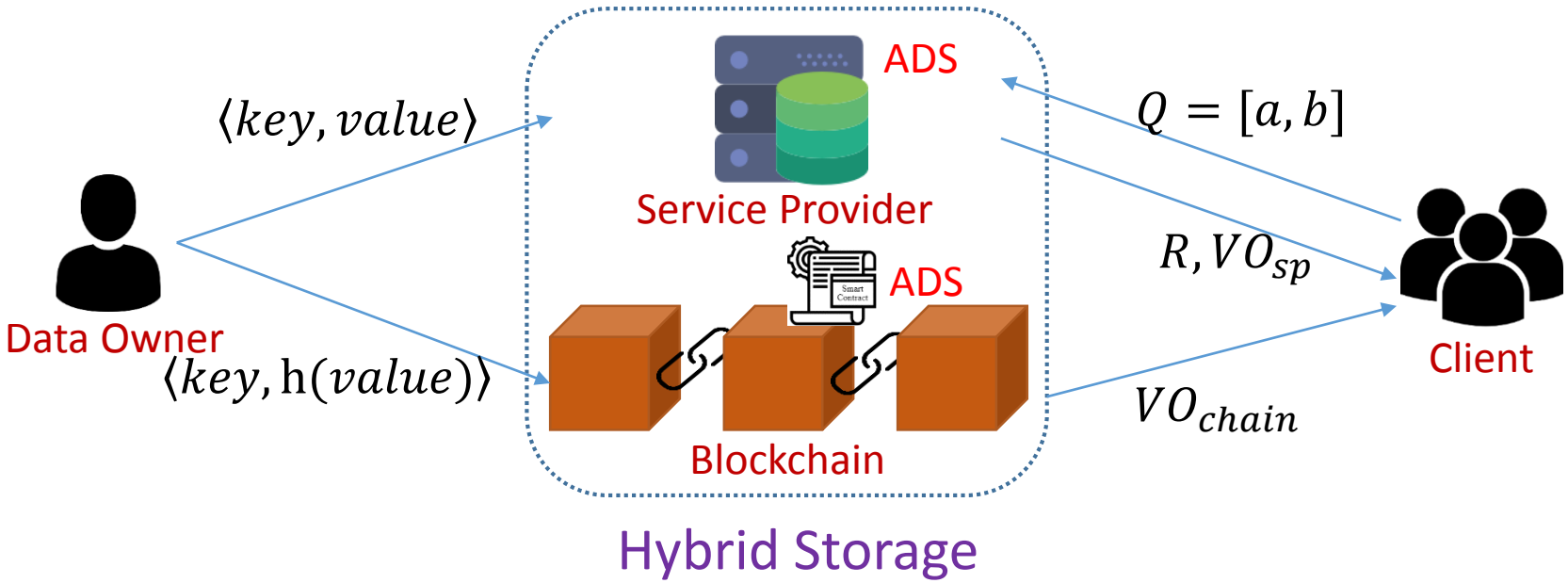


Blockchain Hybrid Storage



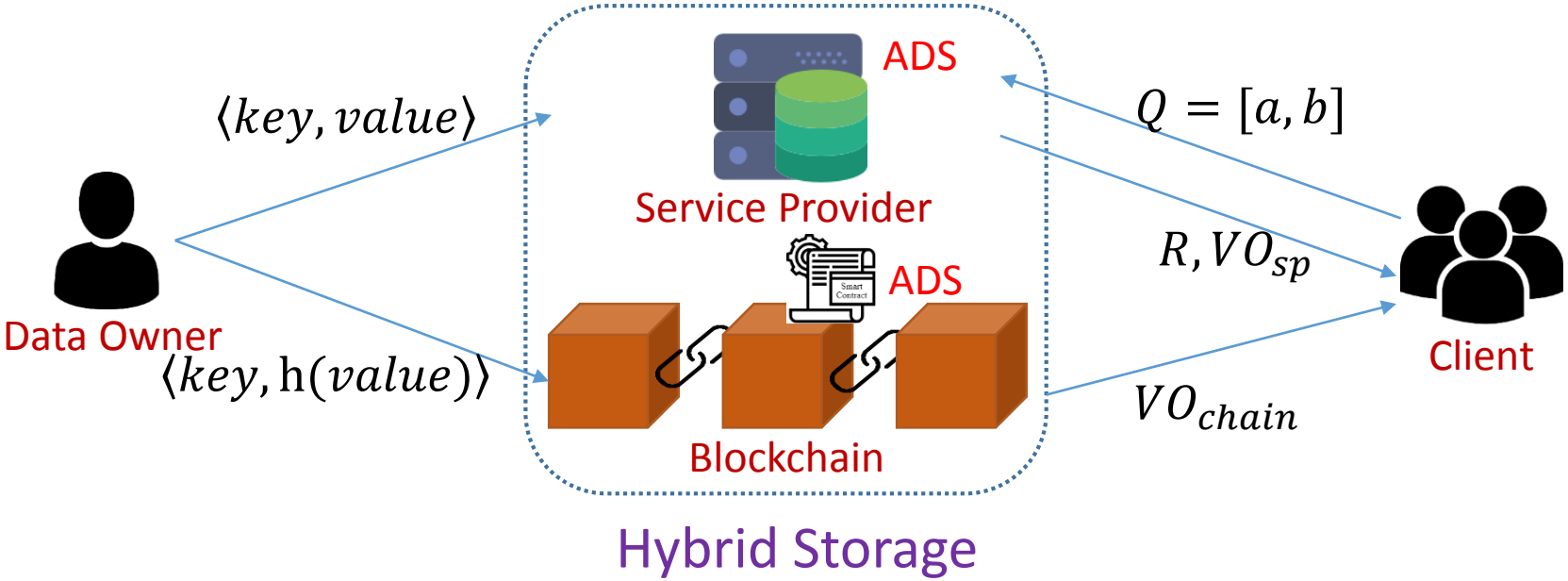
- Pros: high scalability, result integrity assured
- Cons: only support exact search
- Consider other type of queries?

Objective and General Idea



- 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 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 transaction
- Transaction fee for smart contract-enabled blockchain
  - Modeled by *gas* for storage and computation (Ethereum)
- Objective: How to design efficient ADS to be maintained by smart contract under the gas cost model

Ethereum Gas Cost Model

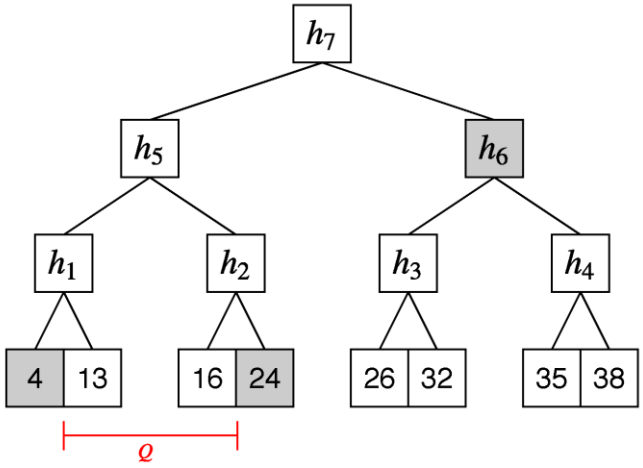
Operation	Gas Used	Explanation
C_{sload}	200	load a word from storage
C_{sstore}	20,000	store a word to storage
$C_{supdate}$	5,000	update a word to storage
C_{mem}	3	access a word in memory
C_{hash}	$30 + 6 \cdot \lvert words \rvert$	hash an arbitrary-length data

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

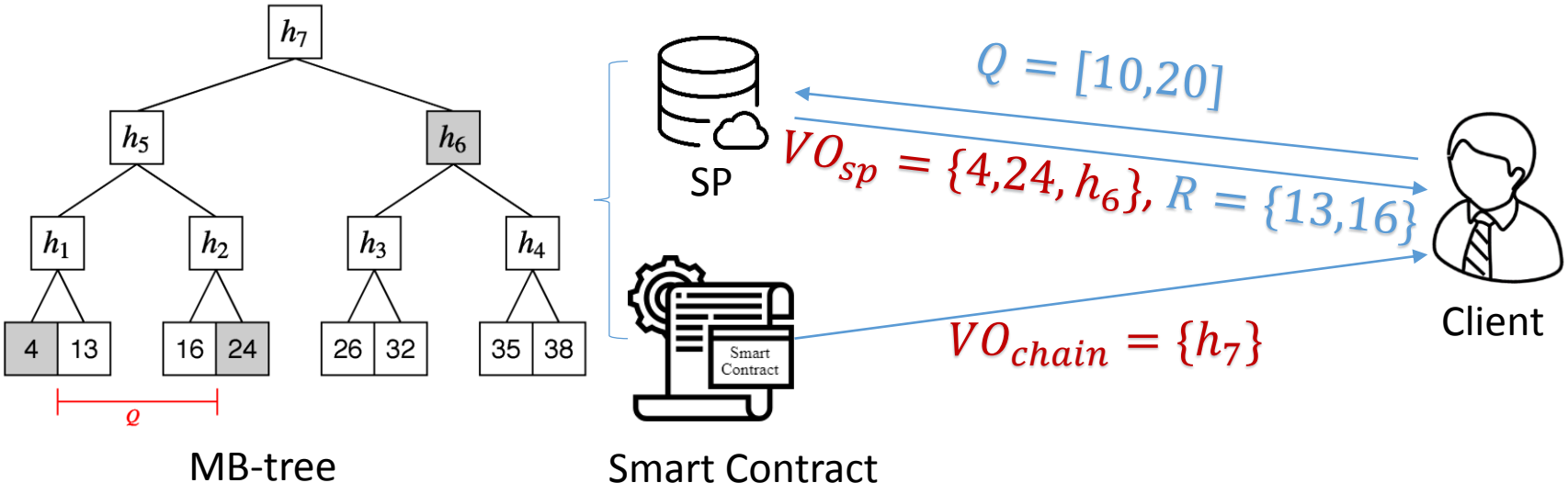


Result: {13,16}

VO: {4, 24, h_6}

Baseline Solution (1)

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



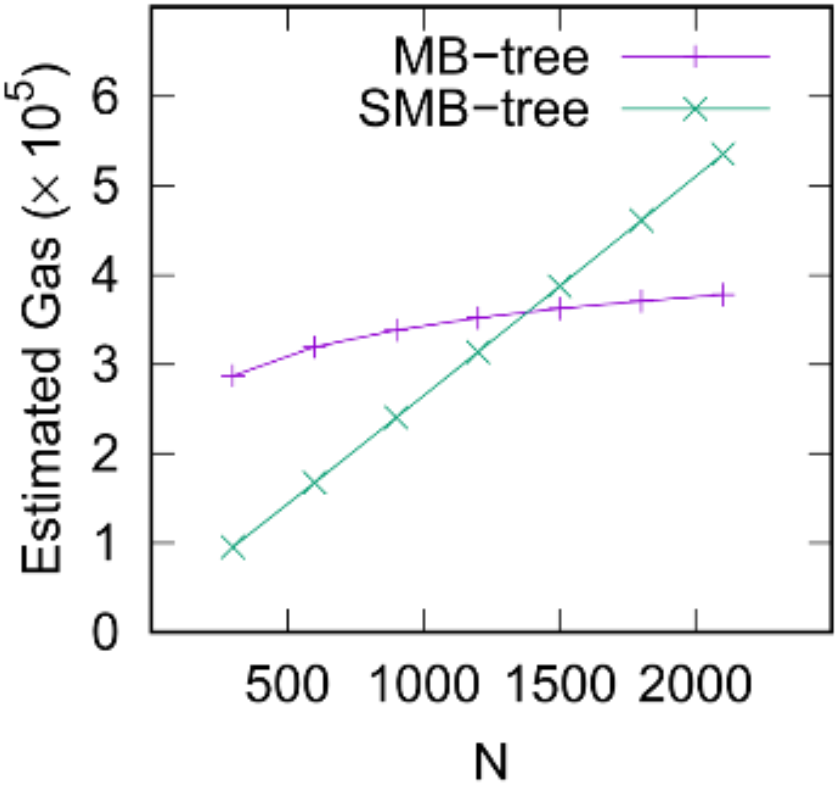
Baseline Solution (2)

- 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)

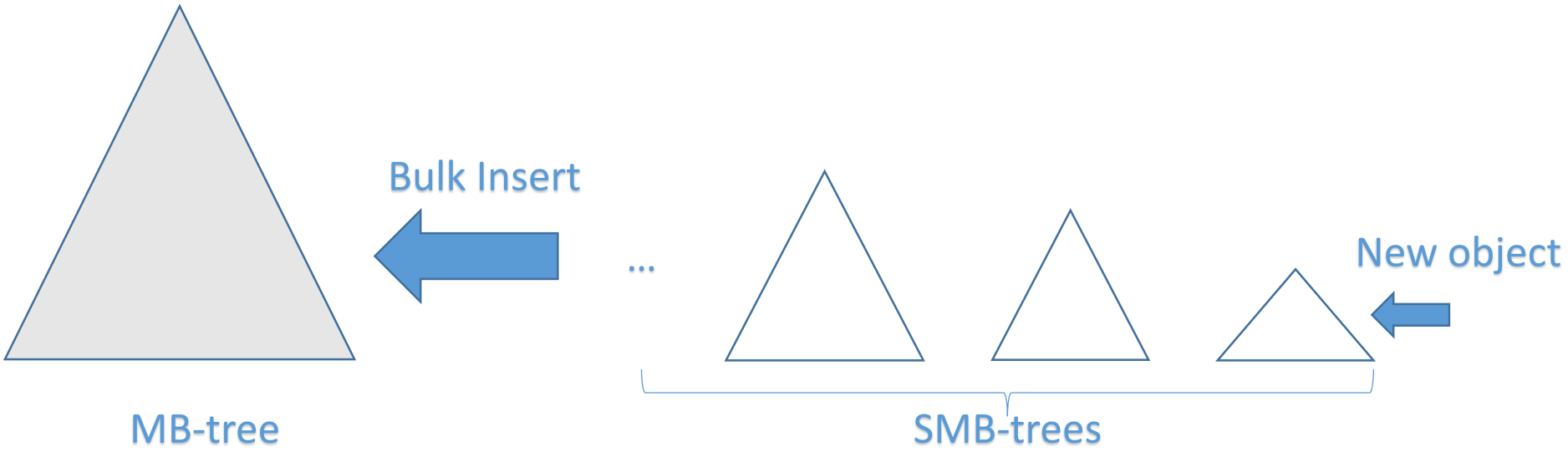
- $C_{\mathrm{SMB-tree}}^{\mathrm{insert}} = N\left(C_{sload} + \log N \cdot C_{mem} + \frac{1}{F} C_{hash}\right) + C_{sstore} + C_{supdate}$

MB-tree vs SMB-tree

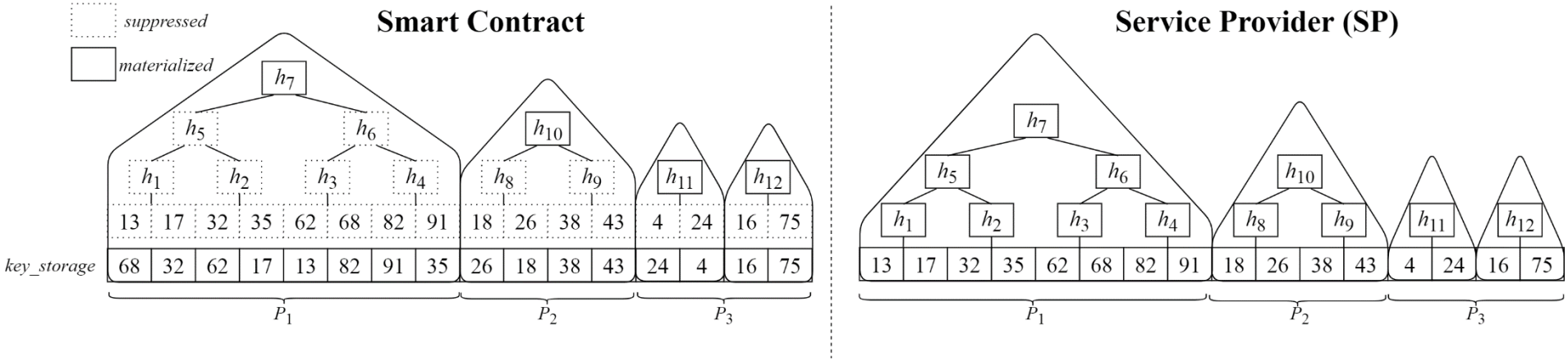


Gas-Efficient Merkle Merge Tree (GEM^2-Tree)

- 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)

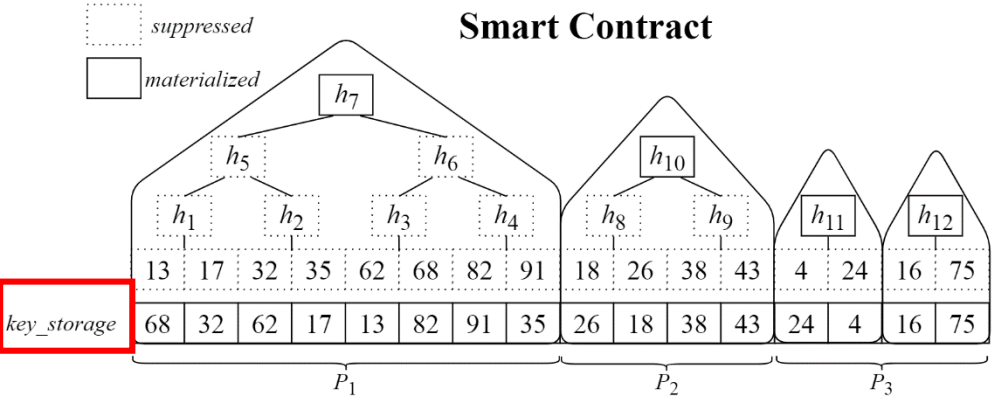
Key	*Value*
13	h(v_{13})
17	h(v_{17})
...	...

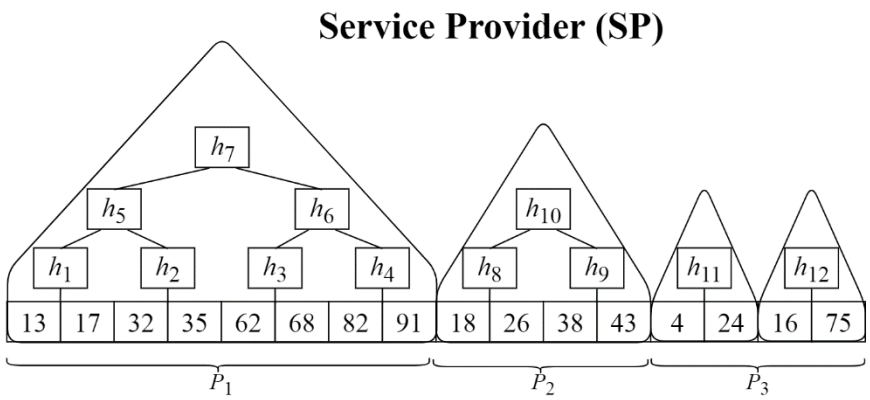
part_table

Partition	Loc_{Tl}	h_{Tl}	Loc_{Tr}	h_{Tr}
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)

An Example





key_map (Hash Map)

Key	*Loc*
68	*1*
32	*2*
...	...

value_storage (Hash Map)

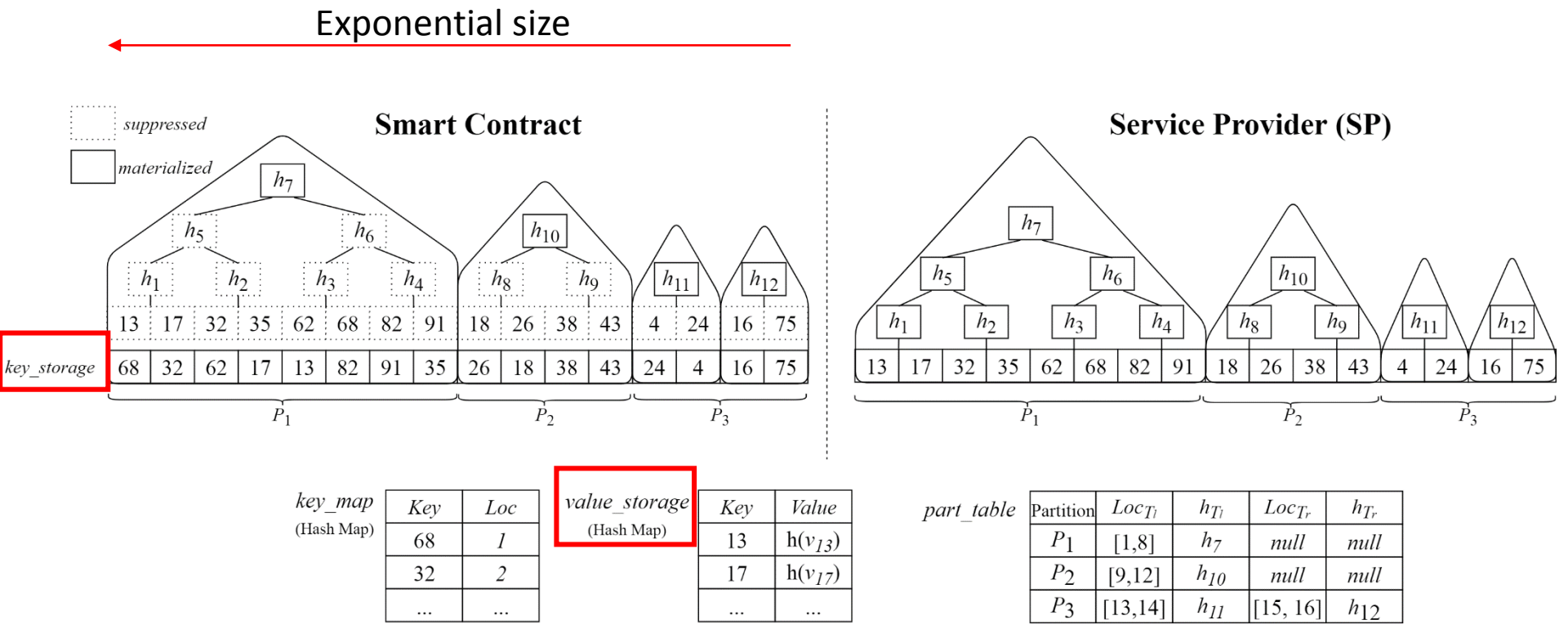
Key	*Value*
13	h(v_{13})
17	h(v_{17})
...	...

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)

An Example



key_map (Hash Map)

Key	*Loc*
68	*1*
32	*2*
...	...

value_storage (Hash Map)

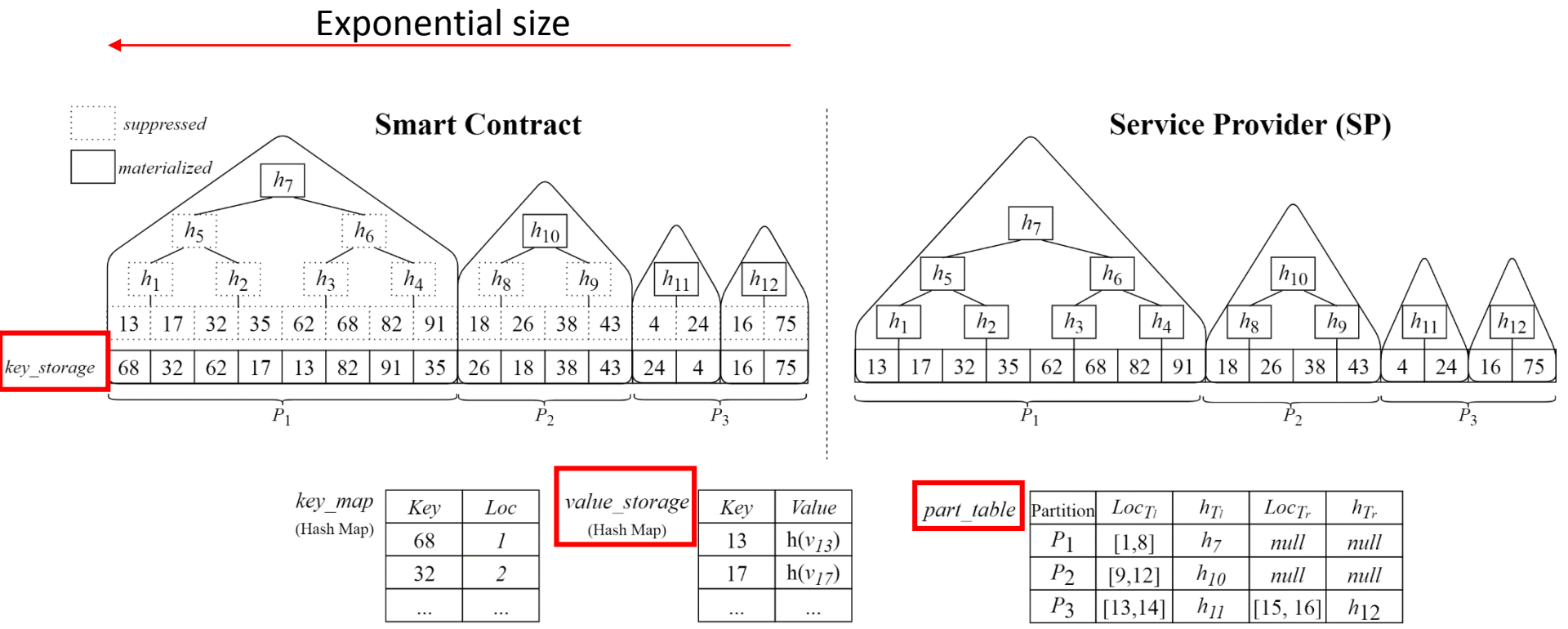
Key	*Value*
13	h(v_{13})
17	h(v_{17})
...	...

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)

An Example



key_map (Hash Map)

Key	Loc
68	*1*
32	*2*
...	...

value_storage (Hash Map)

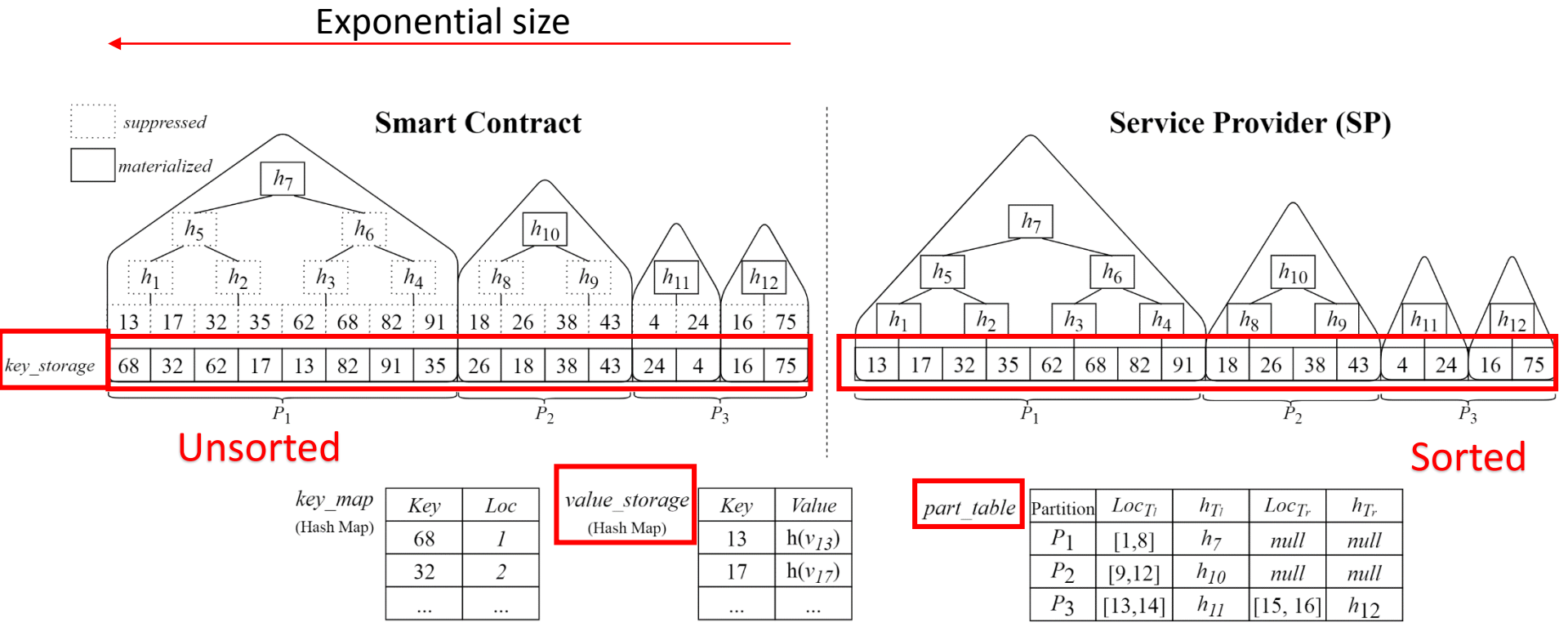
Key	Value
13	h(v_{13})
17	h(v_{17})
...	...

part_table

Partition	Loc_{Tl}	h_{Tl}	Loc_{Tr}	h_{Tr}
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)

An Example



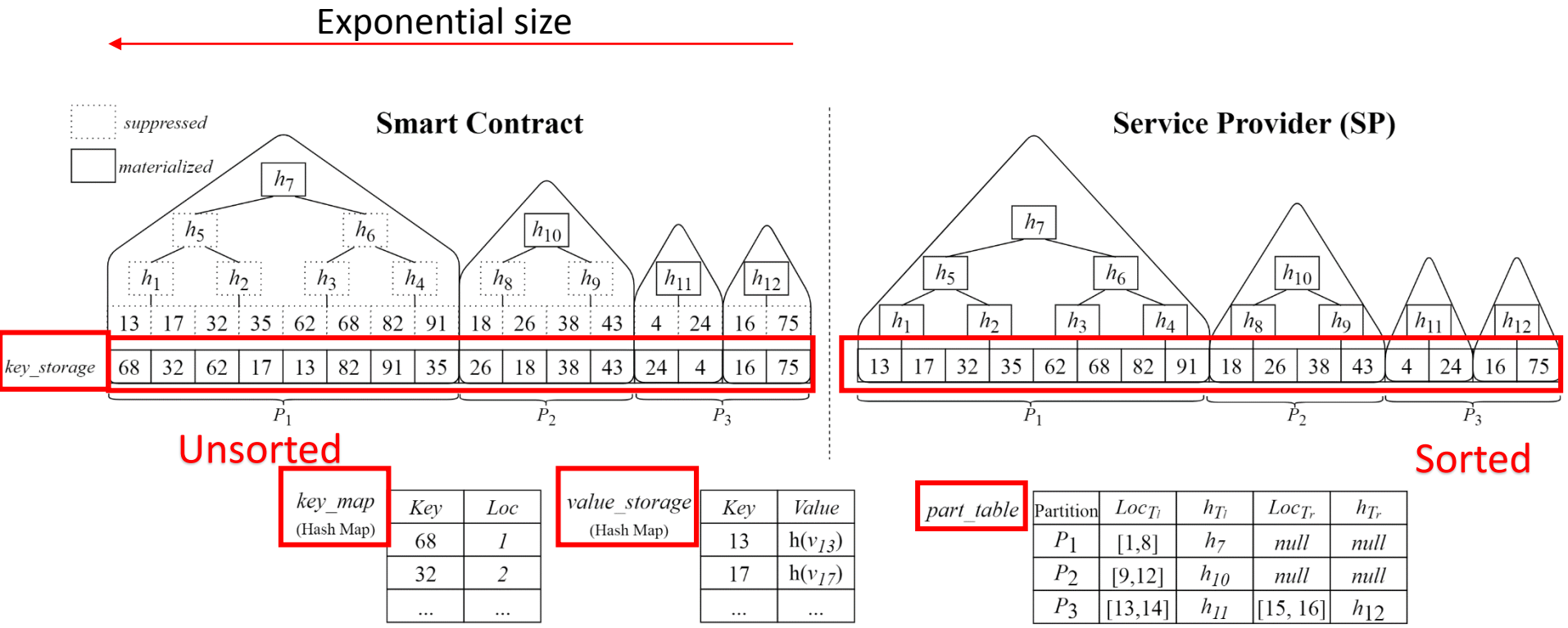
key_map (Hash Map)	Key	Loc
	68	*1*
	32	*2*
	...	...

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)

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)

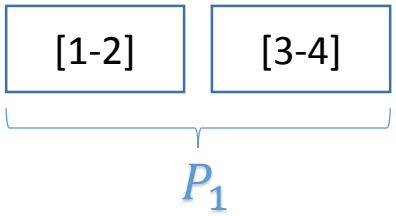
Insertion

- 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

Insertion

- Example ($M = 2$)



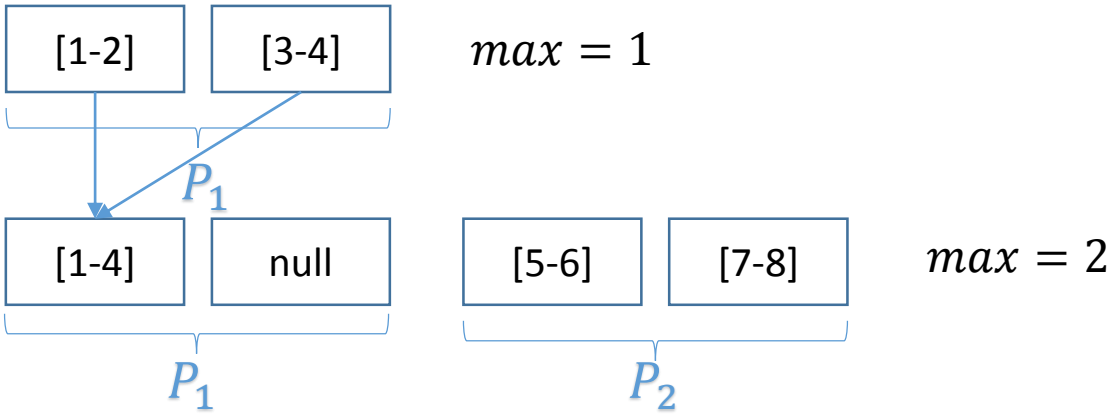
$max = 1$

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

Insertion

- Example ($M = 2$)

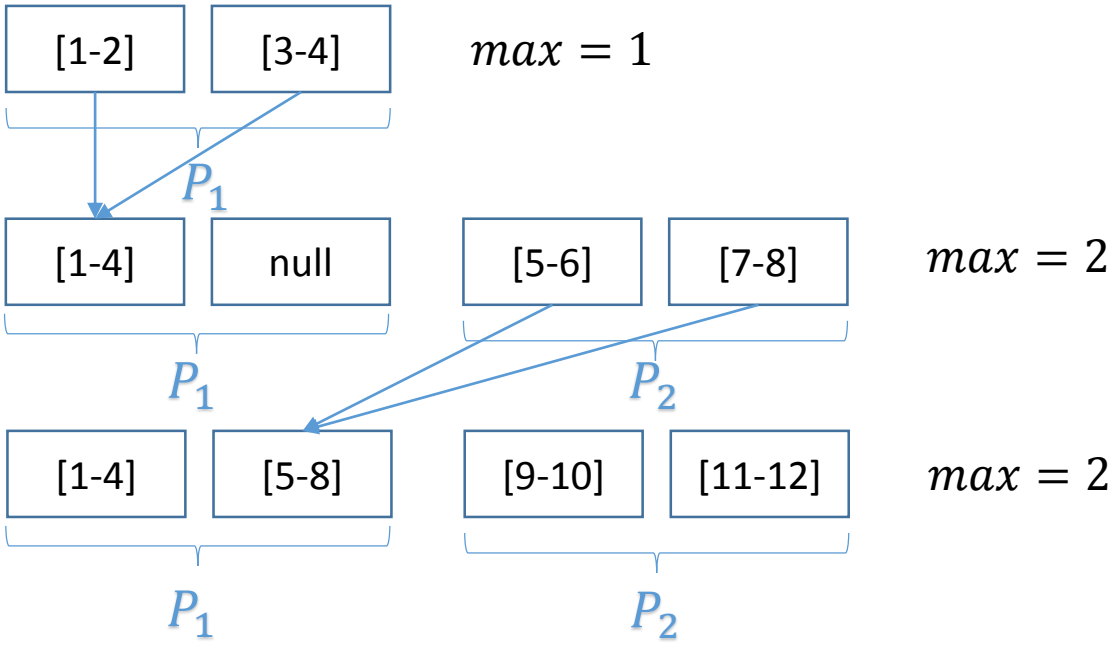
- If P_{max} is not full, insert object to P_{max};
- Else merge the two SMB-trees to a bigger SMB-tree



Insertion

- Example ($M = 2$)

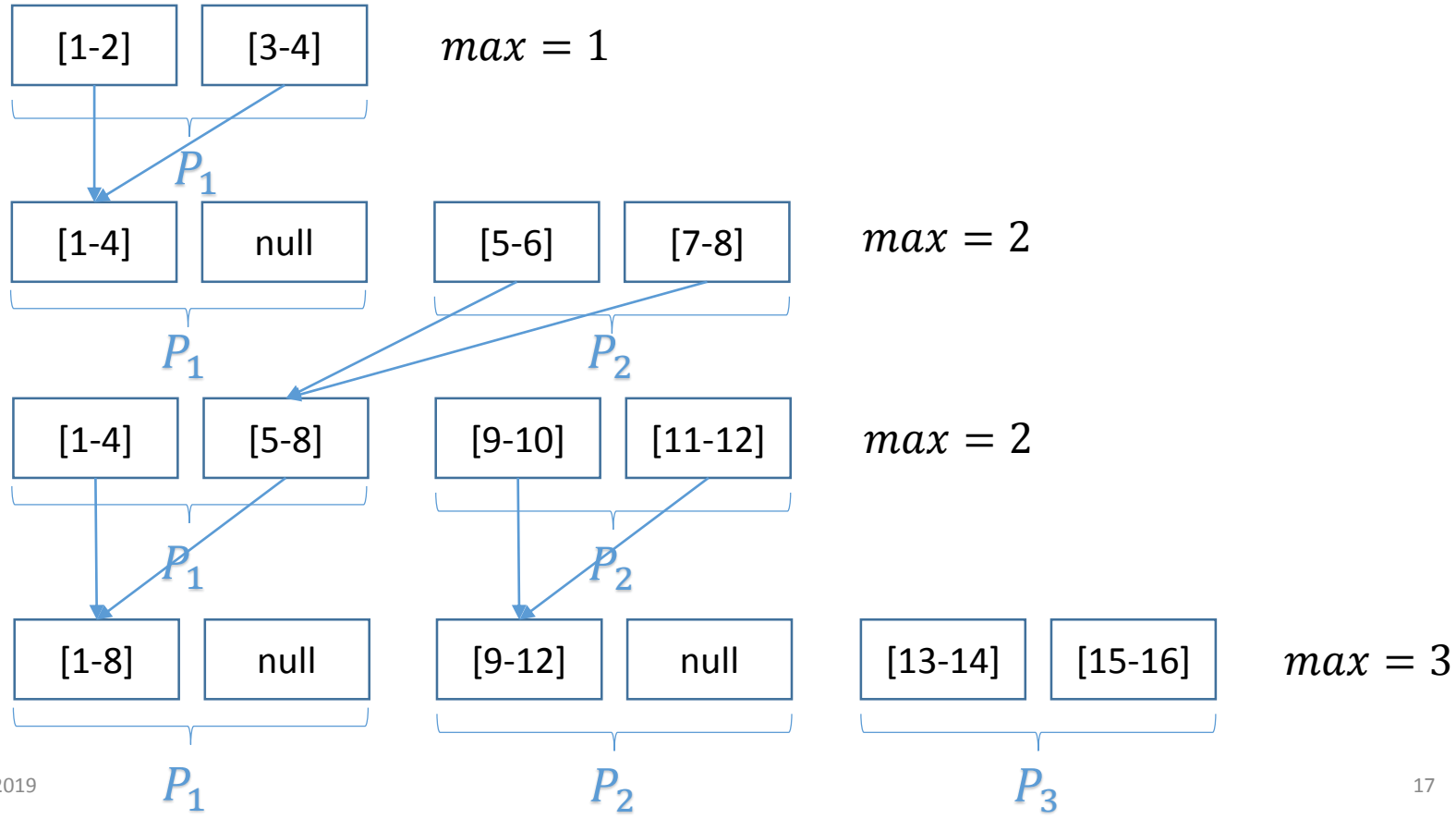
- If P_{max} is not full, insert object to P_{max};
- Else merge the two SMB-trees to a bigger SMB-tree



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



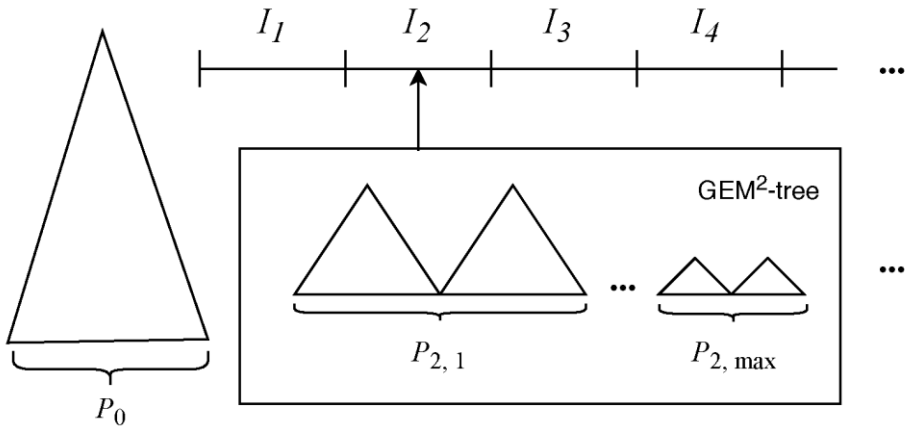
Update and Query Processing

- Update
  - 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

- 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 checks the VO and results against each of these trees

Optimized GEM^{2*}-Tree

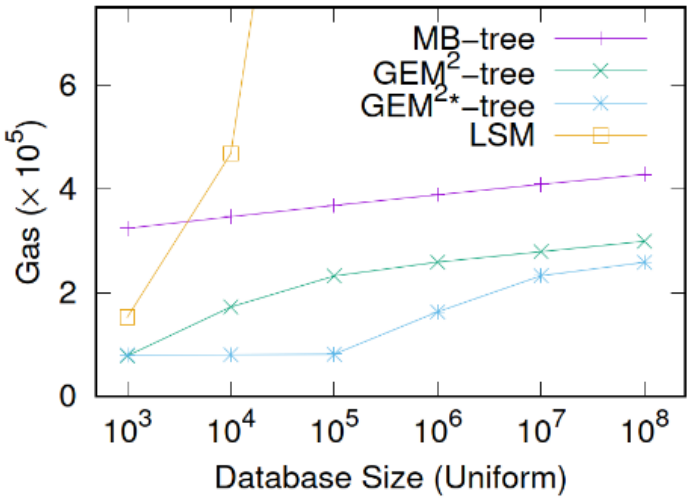
- Objective: to further reduce the gas consumption without sacrificing much of the query overhead
- Design structure
  - Two-level index
    - Upper level: split the search key domain into several regions
    - Lower level: a GEM^2-tree is built for each region I_i
  - Only one single MB-tree for the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

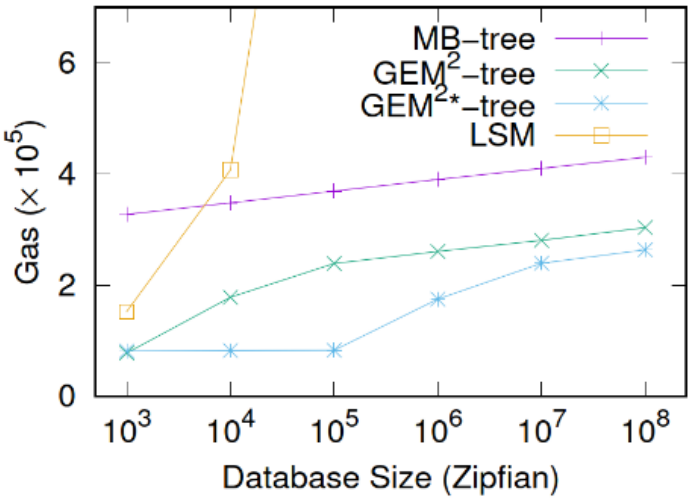


Performance Evaluation

- Dataset
  - Synthetic data generated by Yahoo Cloud System Benchmark (YCSB)
  - Cardinality: 100M
  - Key size: 4 bytes
  - Key distribution: uniform/Zipfian
- Parameters of the index
  - 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 set to 4 according to the word size 32 bytes
    - $(f-1)l_d + f l_p < 32\text{byte}$
  - $S_{max} = 2048$ based on the cost analysis of MB-tree and SMB-tree
  - Search key domain is split into 100 regions for upper-level GEM^{2*}-tree

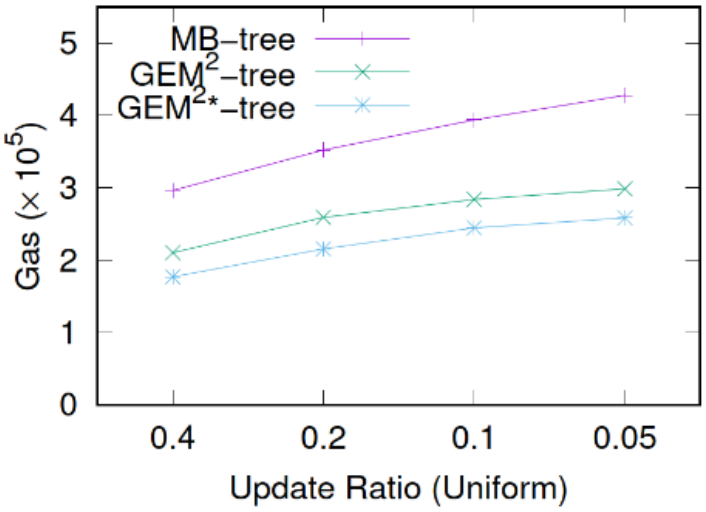
Gas Consumption vs Database Size

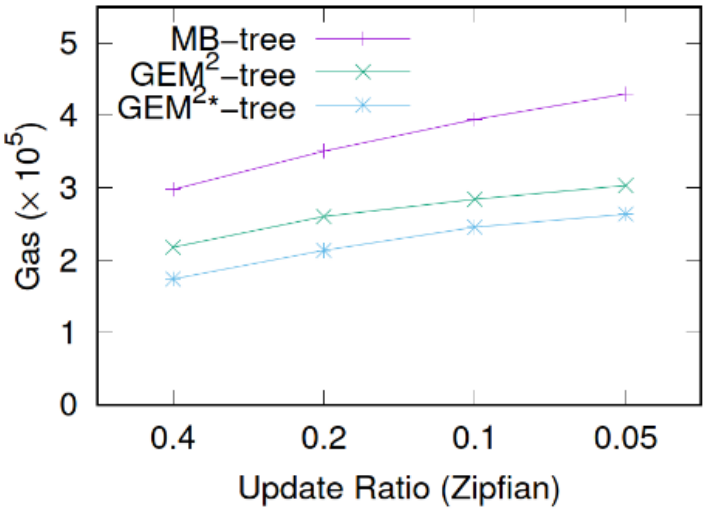




- LSM-tree is able to support the database up to 10,000
  - Merge cost grows exponentially with increasing the level
- Gas reduction of the two proposed indexes
  - Optimized version is the best
  - More SMB-trees, efficient bulk insertion (thanks to the upper level)

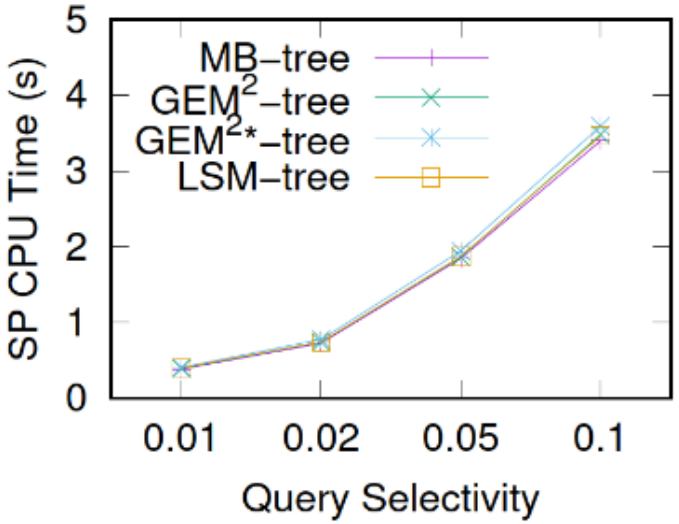
Gas Consumption vs Update Ratio

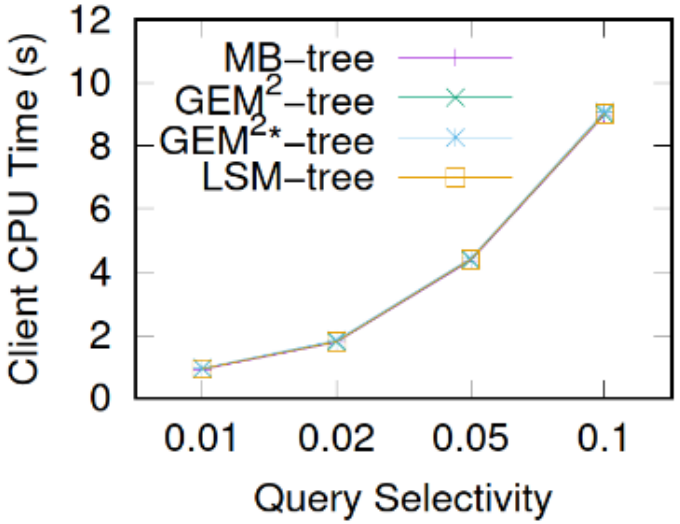




- Update ratio: #update/#total operation
- Update cost is lower than the insertion cost
  - The less the update operations, the more gas consumed

Authenticated Query Performance





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

Summary and Future Work

- Hybrid Storage Blockchain
- Range queries with integrity assurance
- Two proposed index: GEM^2-Tree, GEM^{2*}-Tree
  - Reduce the gas cost with little penalty on the query performance
- Future Work
  - Extended to more query types: join query, keyword search, etc.
  - Search on encrypted blockchain data
  - Data sharing with fine-grained access control

Thanks!
Q&A