

Fast RDMA-based Ordered Key-Value Store using Remote Learned Cache

Xingda Wei, Rong Chen, Haibo Chen



XSTORE

KVS: key pillar for distributed systems

Important *building block* for

- ⌚ Databases, GraphStore
- ⌚ Web applications
- ⌚ Cloud infrastructures
- ⌚ Serverless platforms



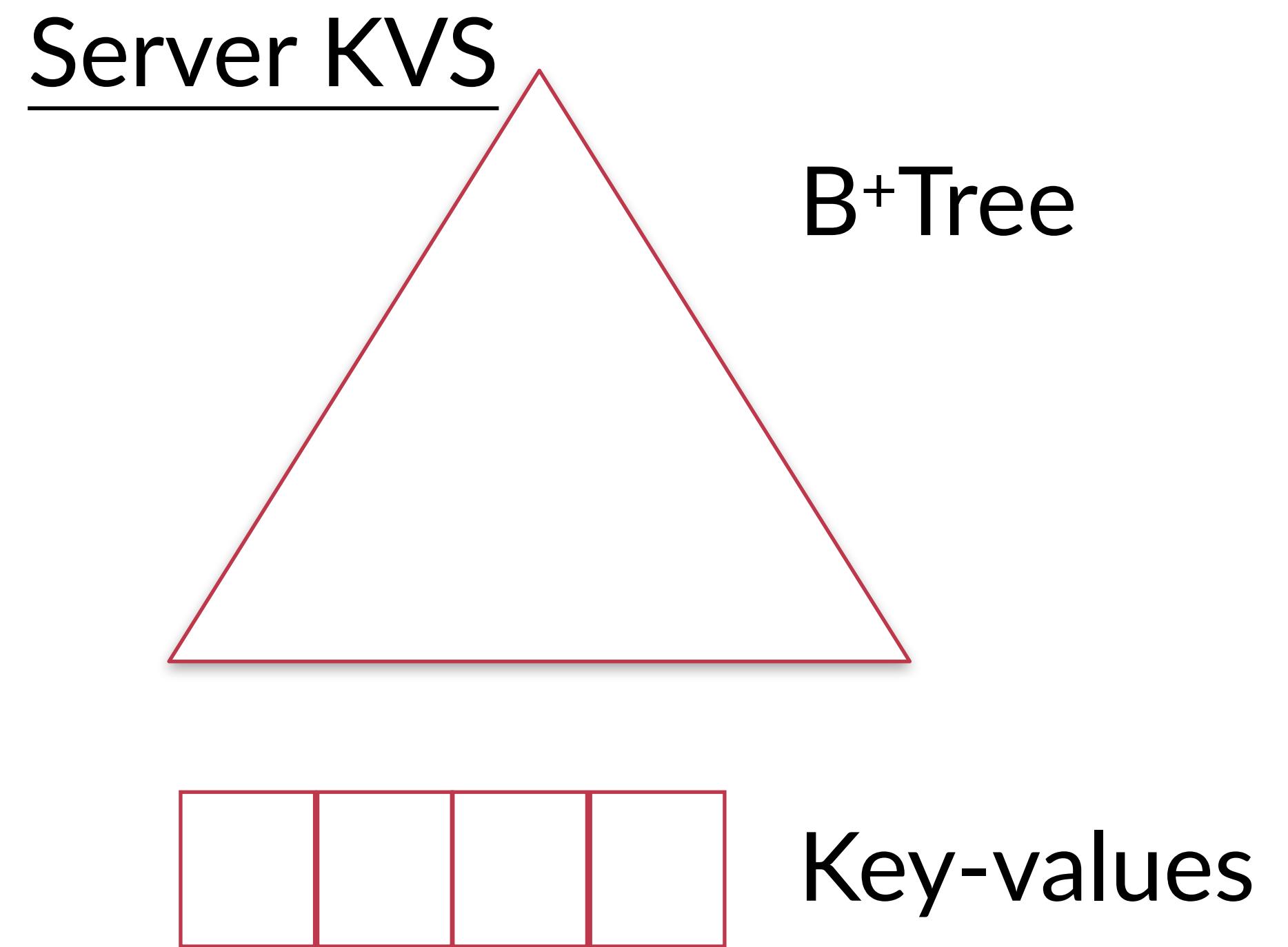
redis



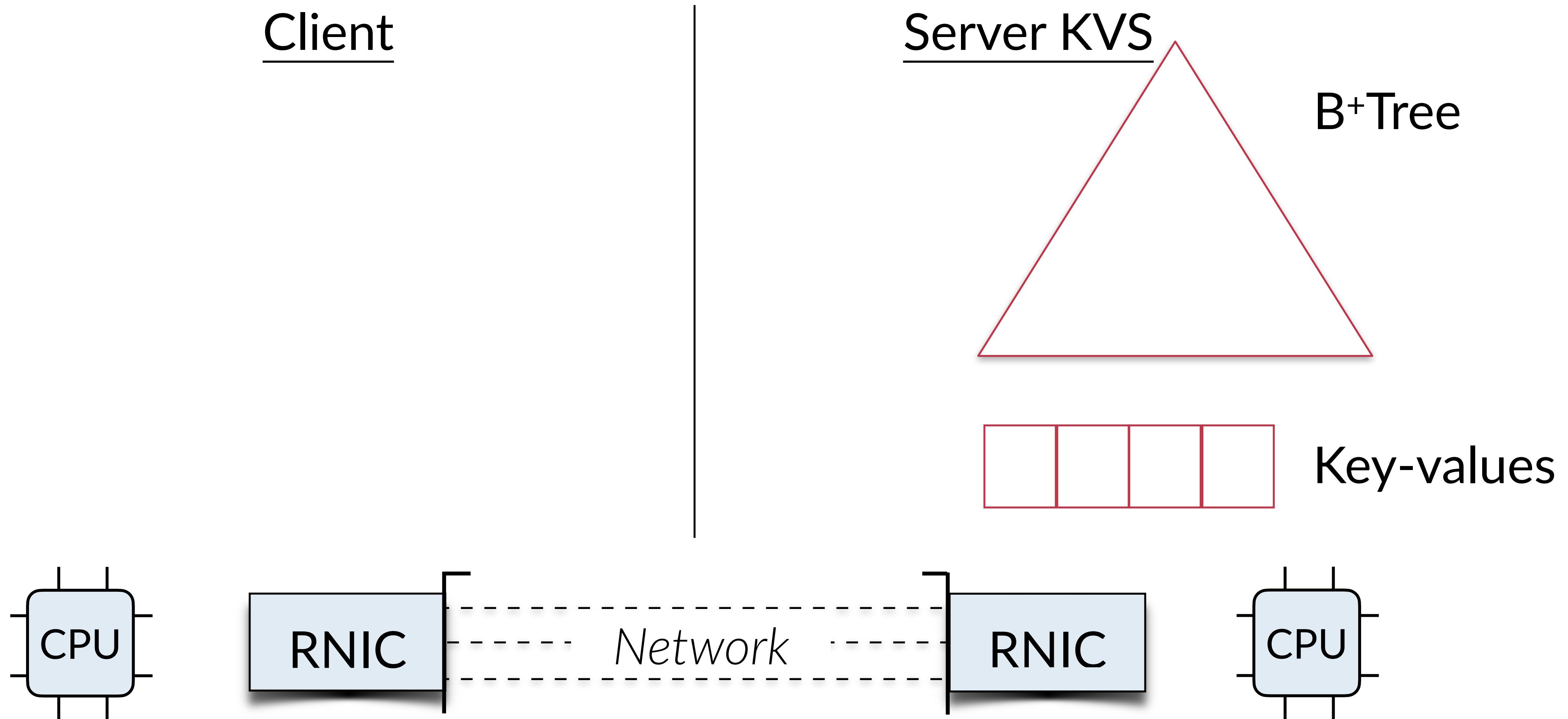
cassandra



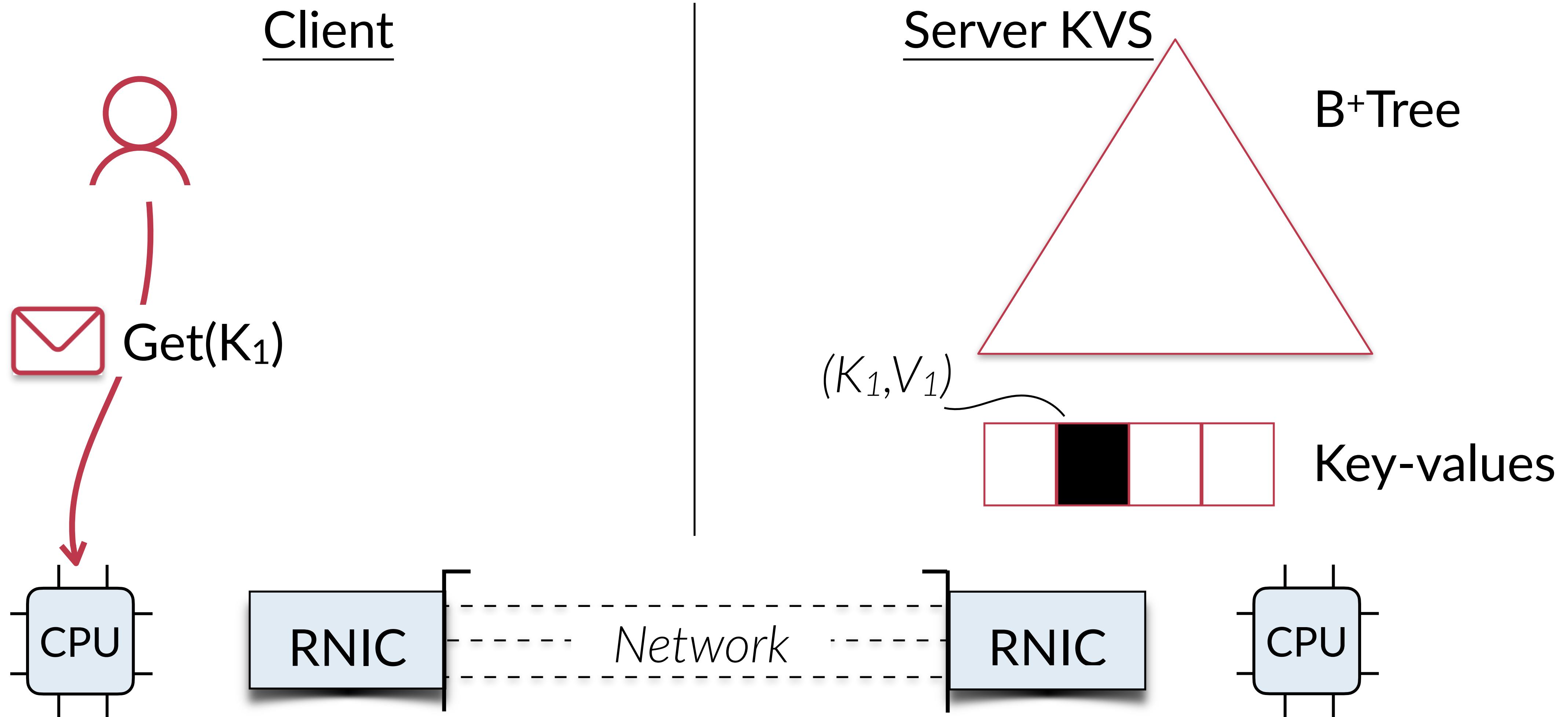
KVS: key pillar for distributed systems



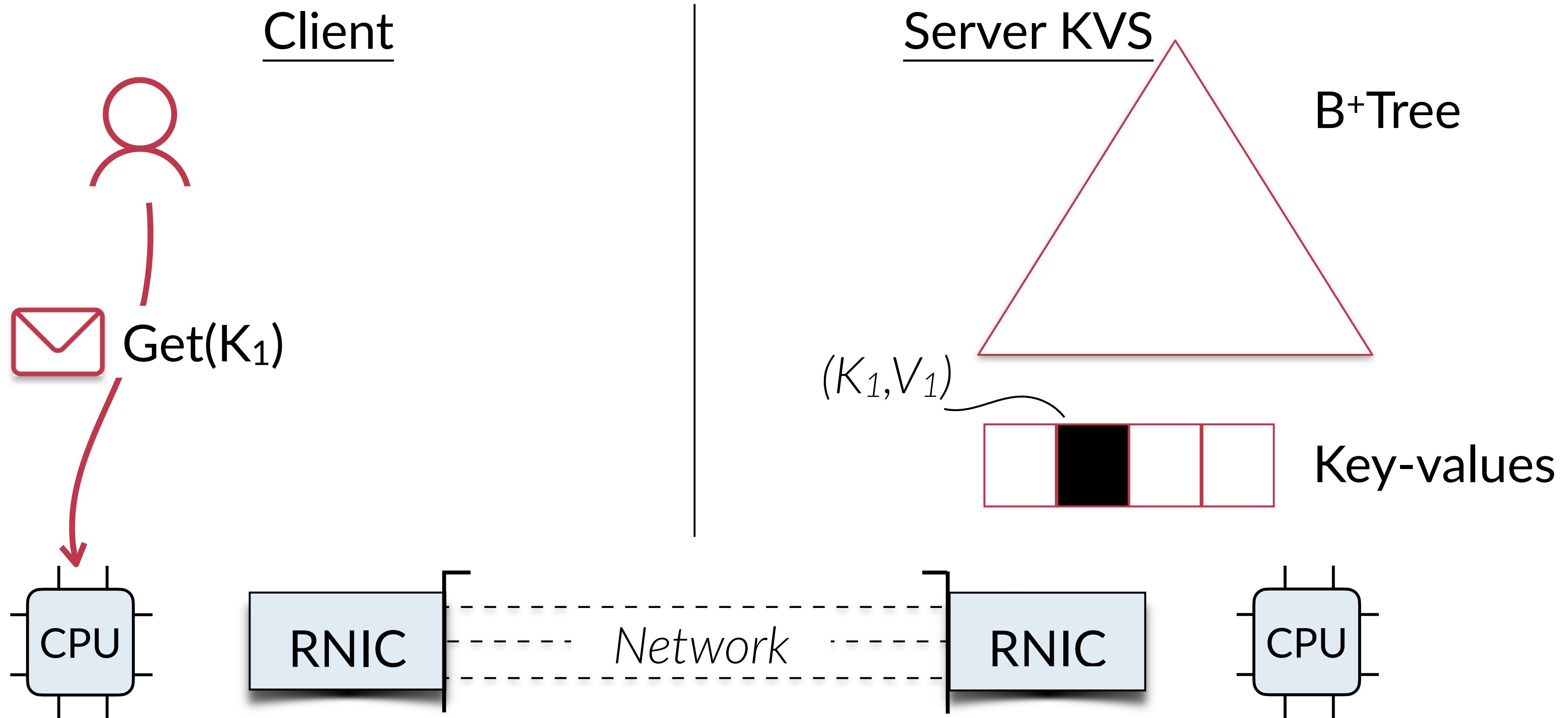
KVS: key pillar for distributed systems



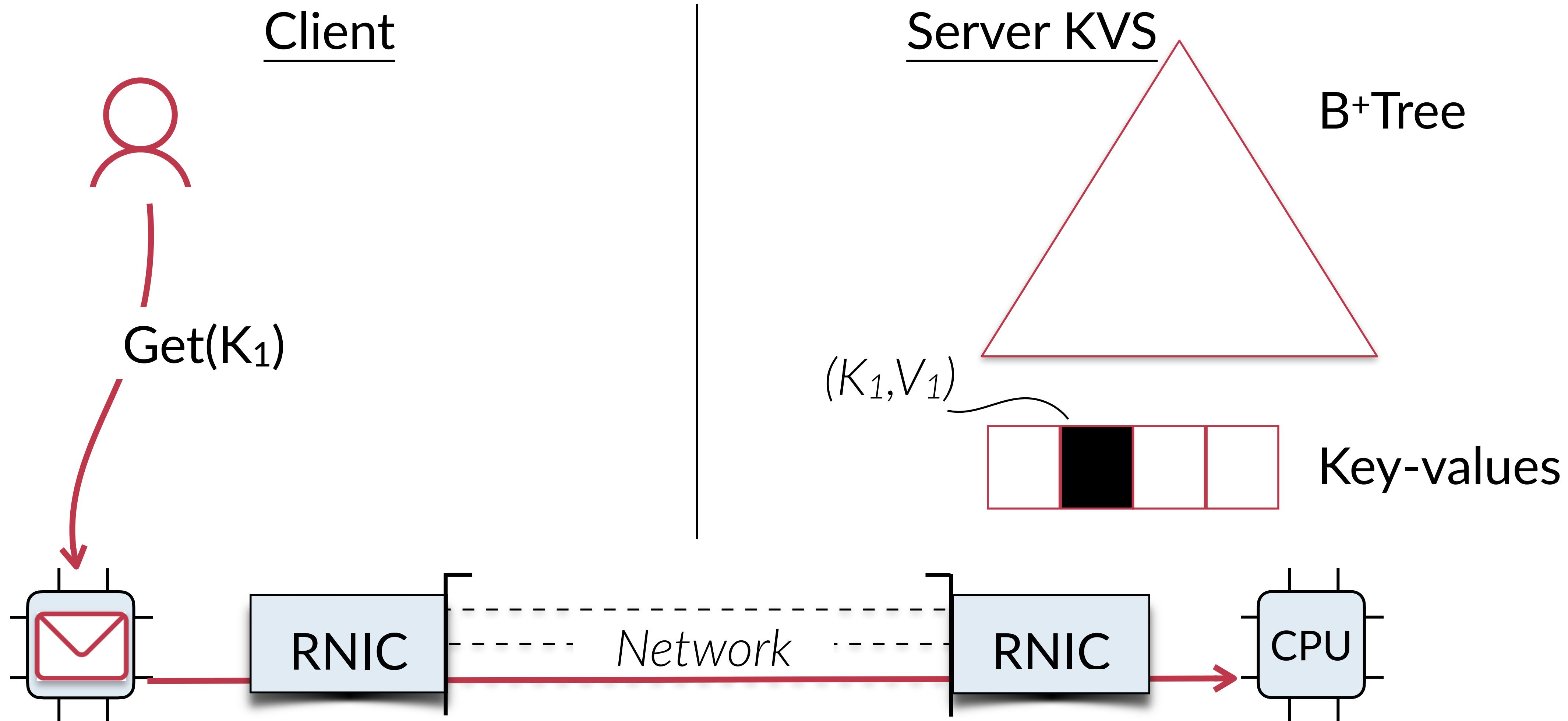
Traditional KVS uses RPC (Server-centric)



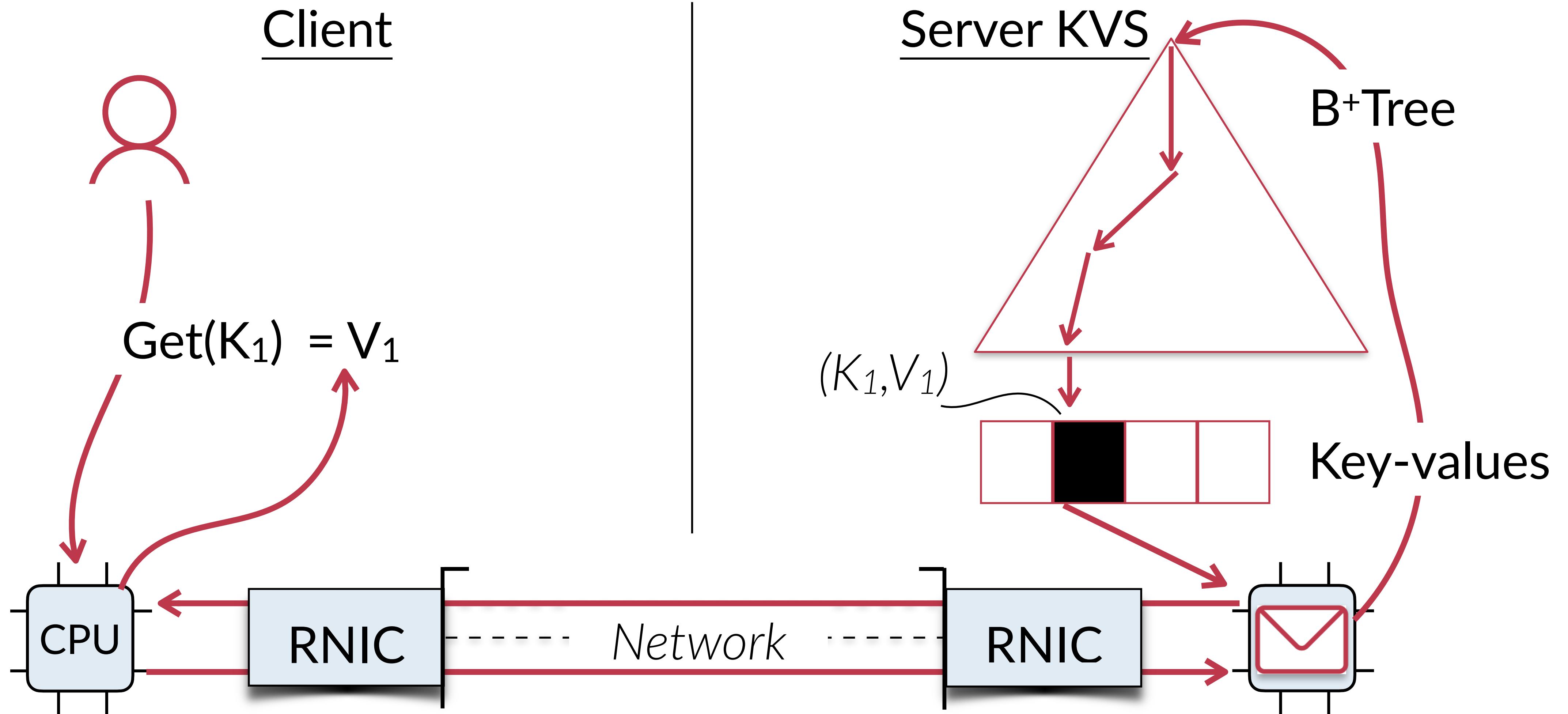
Traditional KVS uses RPC (Server-centric)



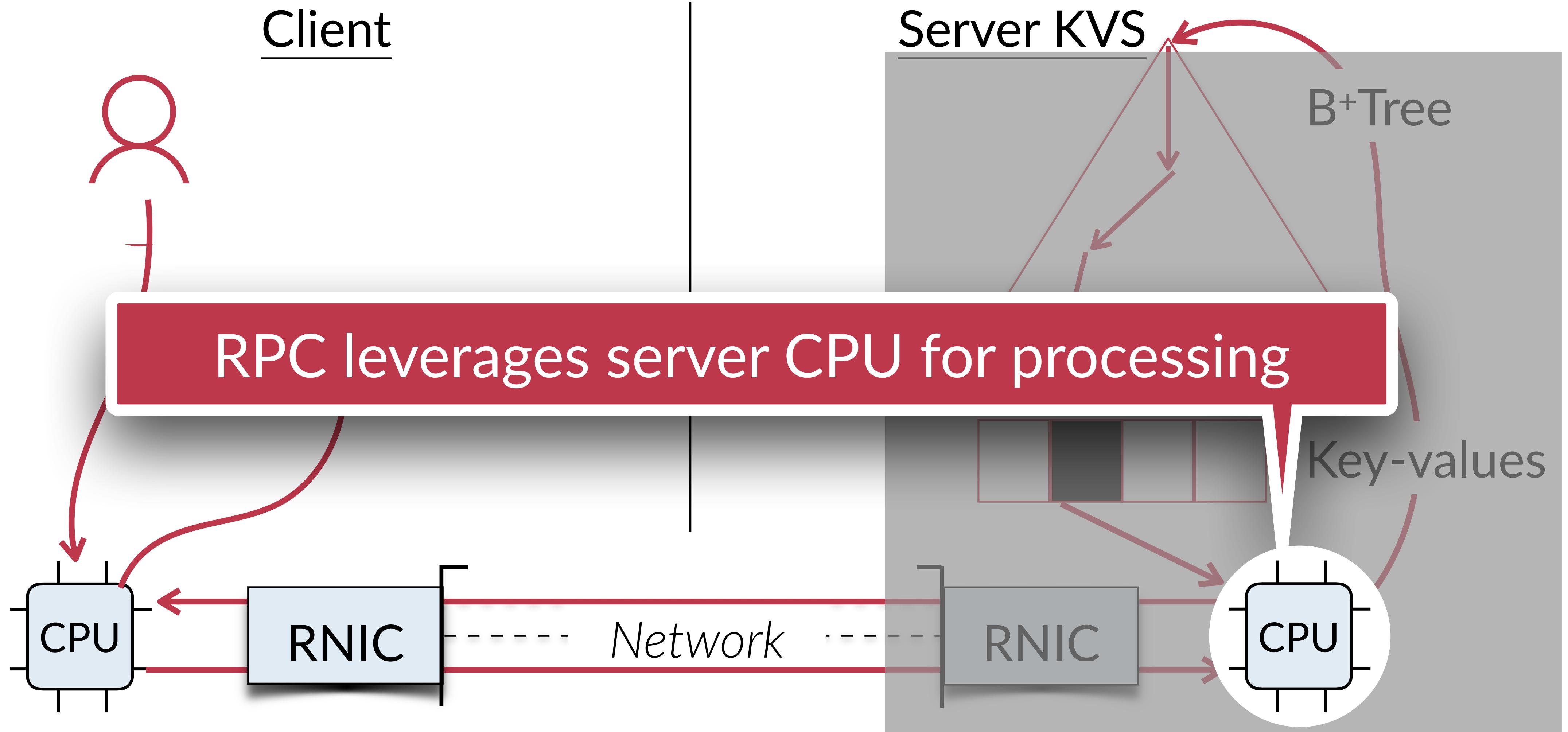
Traditional KVS uses RPC (Server-centric)



Traditional KVS uses RPC (Server-centric)



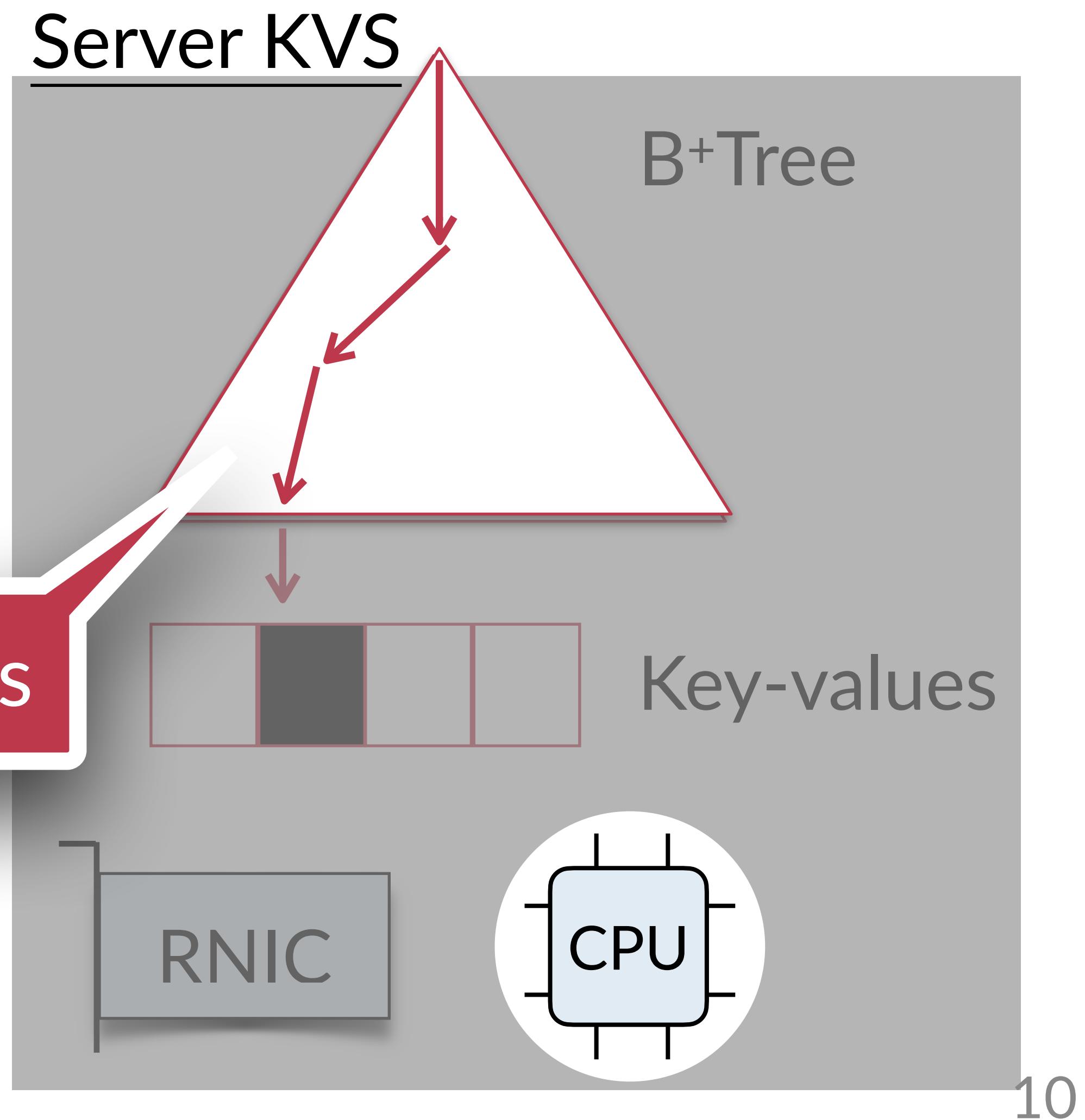
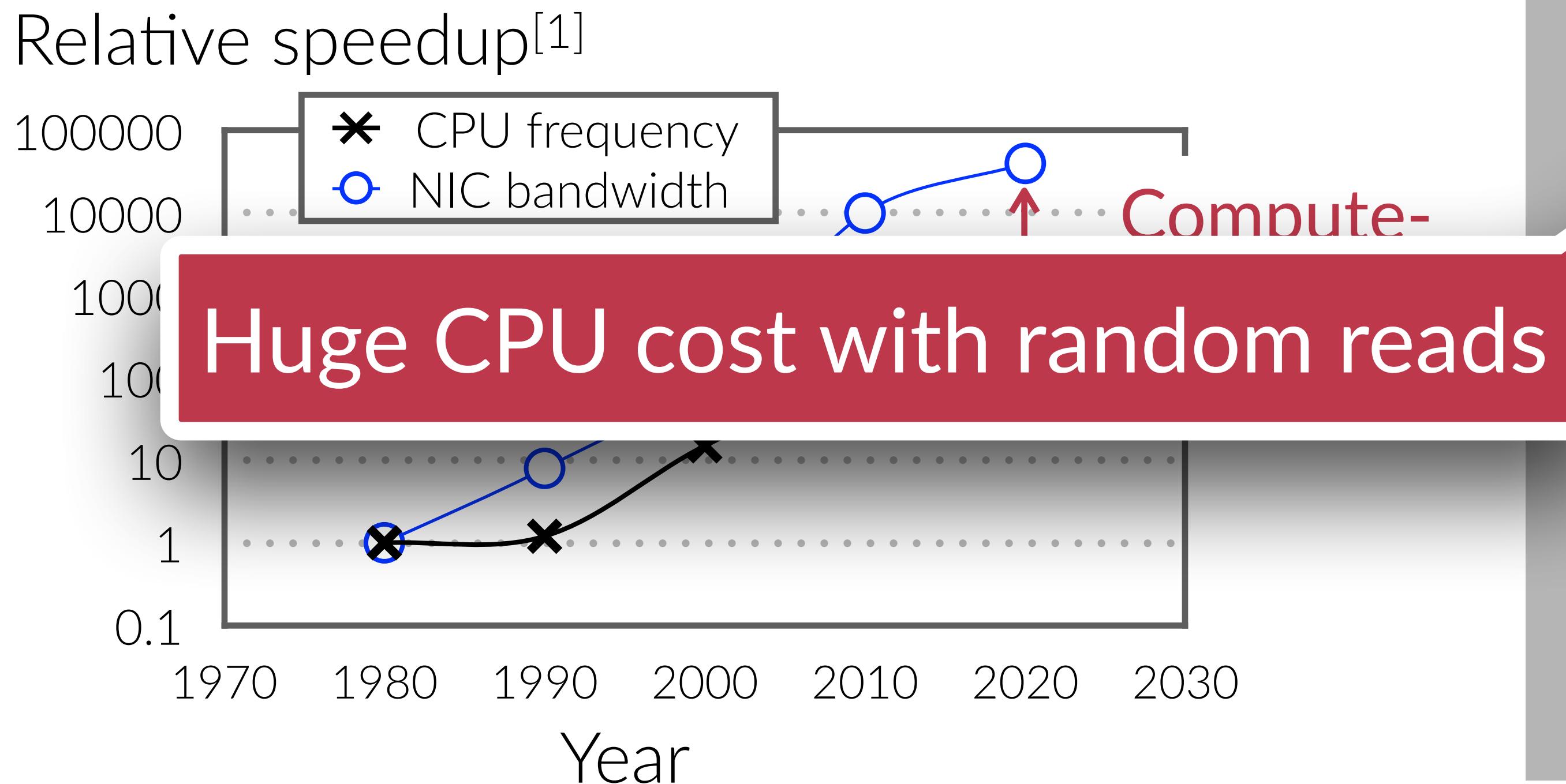
Traditional KVS uses RPC (Server-centric)



Server CPU is becoming the bottleneck

Increasing **CPU-NIC gap**

♪ NIC's speed is growing faster !

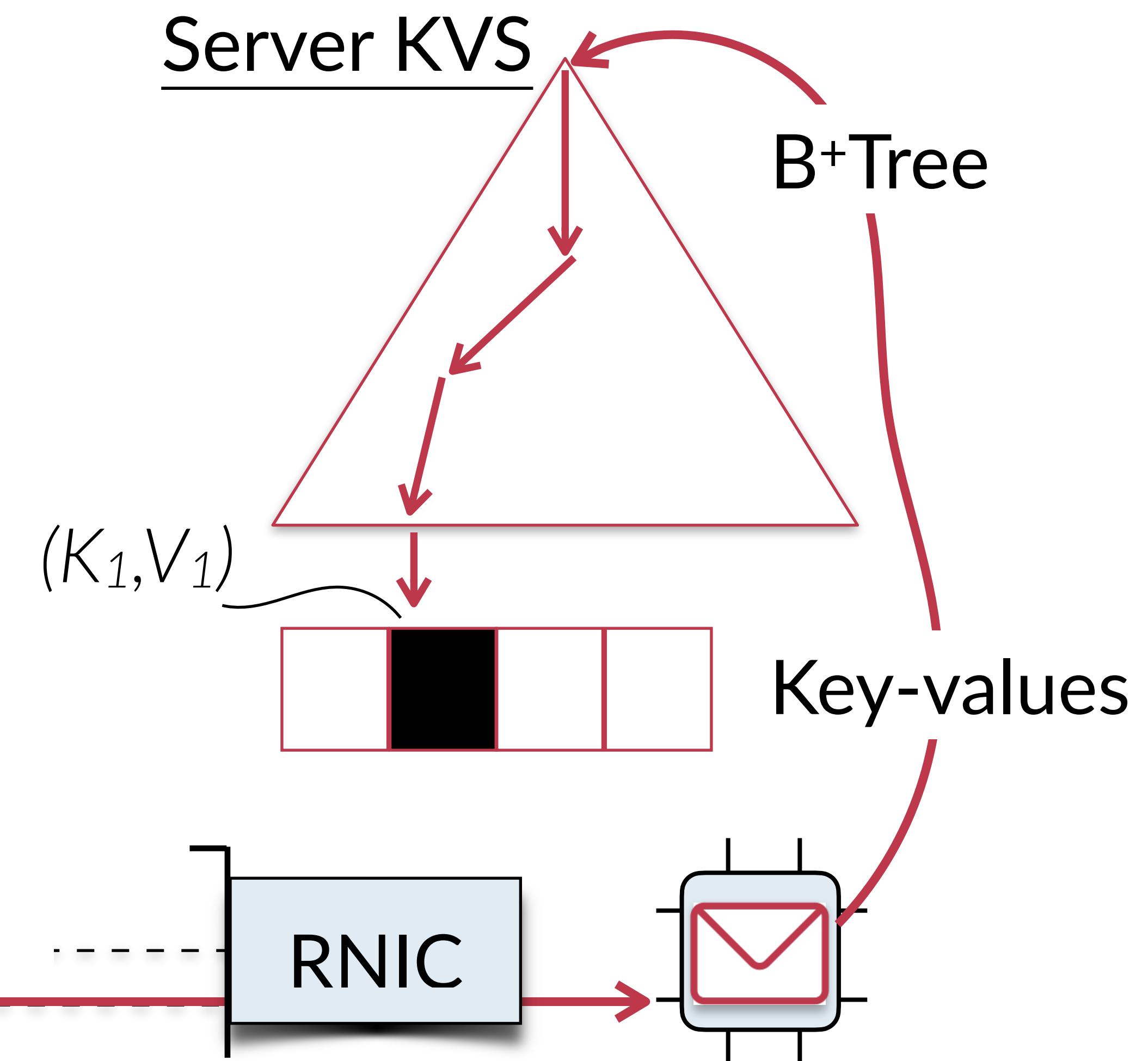
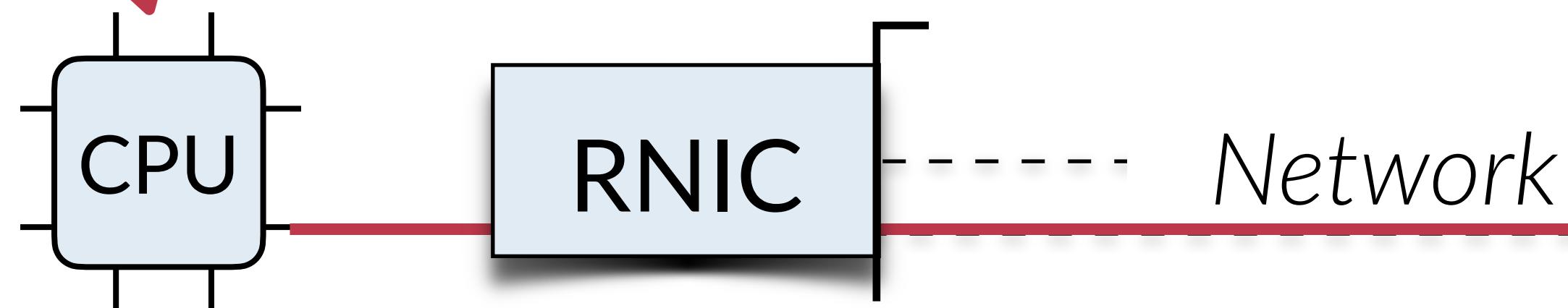


Opportunity: one-sided RDMA (Client-direct)

NIC directly reads/writes memory

- ⌚ Offload index traversal to NIC
- ⌚ Totally bypass server CPU

$\text{Get}(K_1) = V_1$

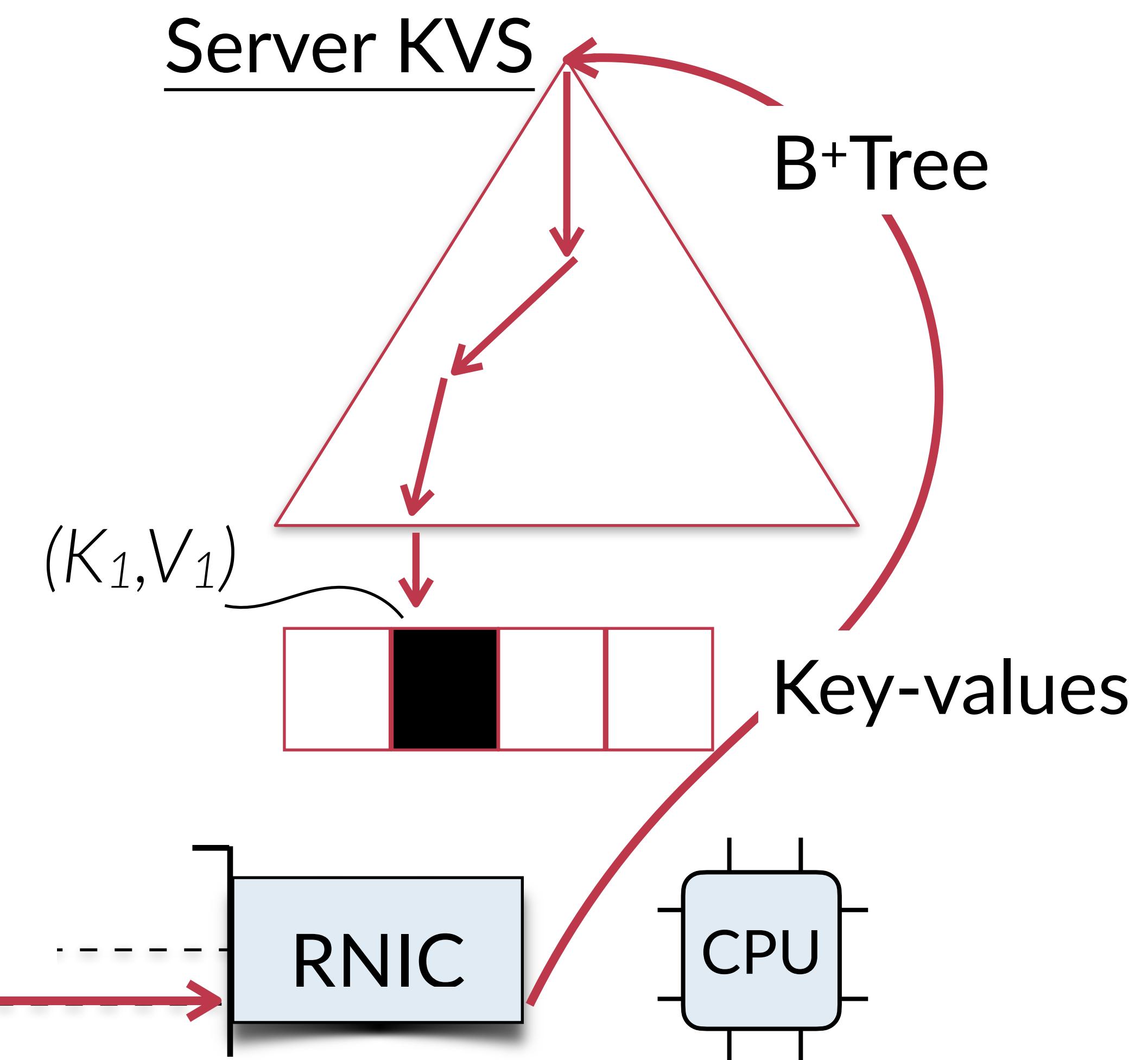
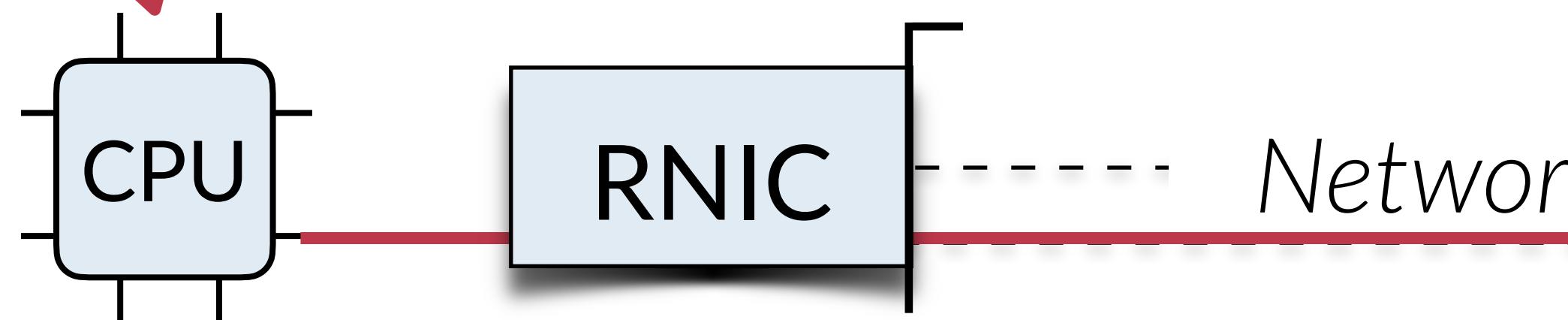


Opportunity: one-sided RDMA (Client-direct)

NIC directly reads/writes memory

- ⌚ Offload index traversal to NIC
- ⌚ Totally bypass server CPU

$\text{Get}(K_1) = V_1$



Challenge: limited NIC abstraction

NIC only has *simple* abstractions

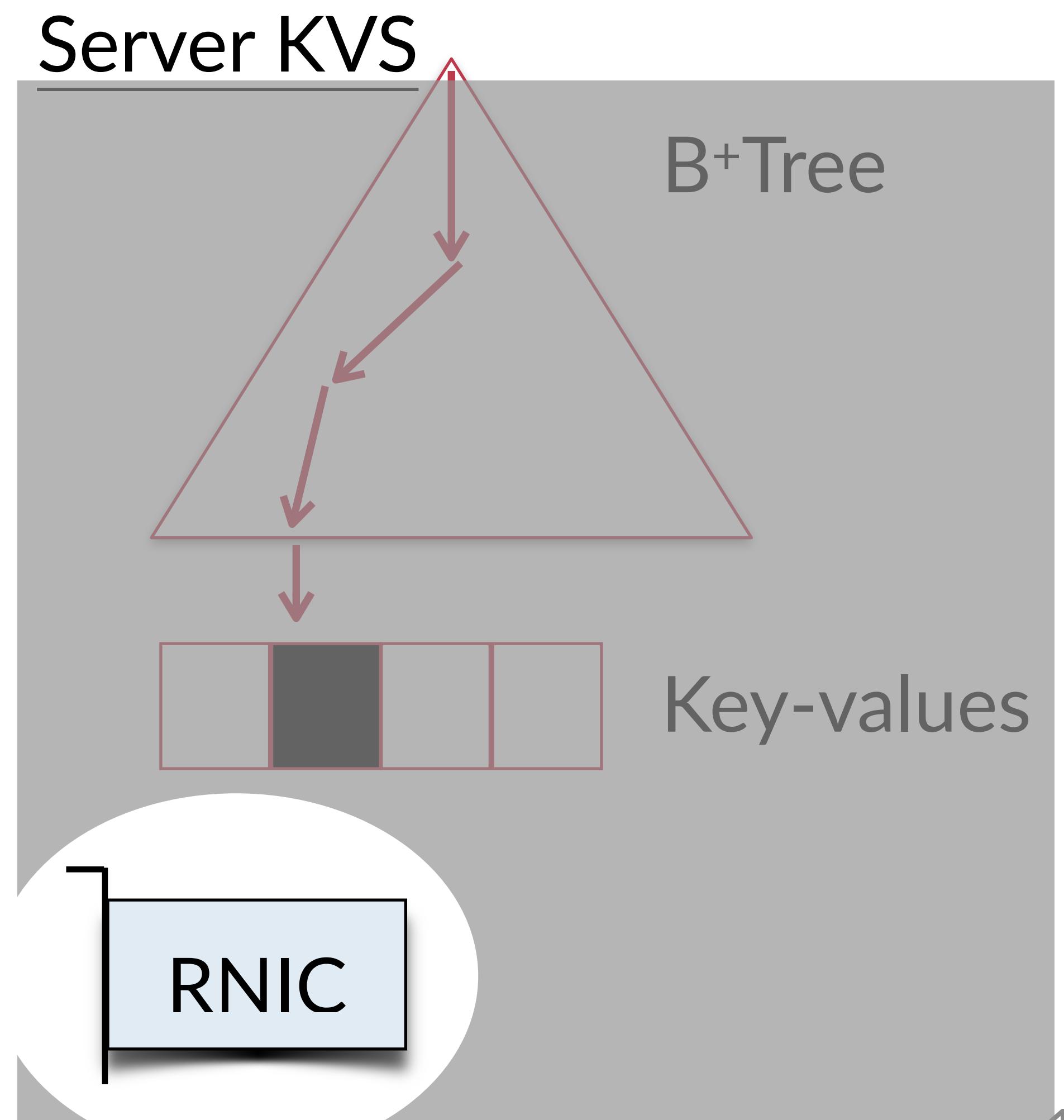
⌚ e.g., memory *read/write*

Works well for simple index structure

⌚ e.g. *HashTable, O(1)* network RTT^[1]

Inferior for complex index structure

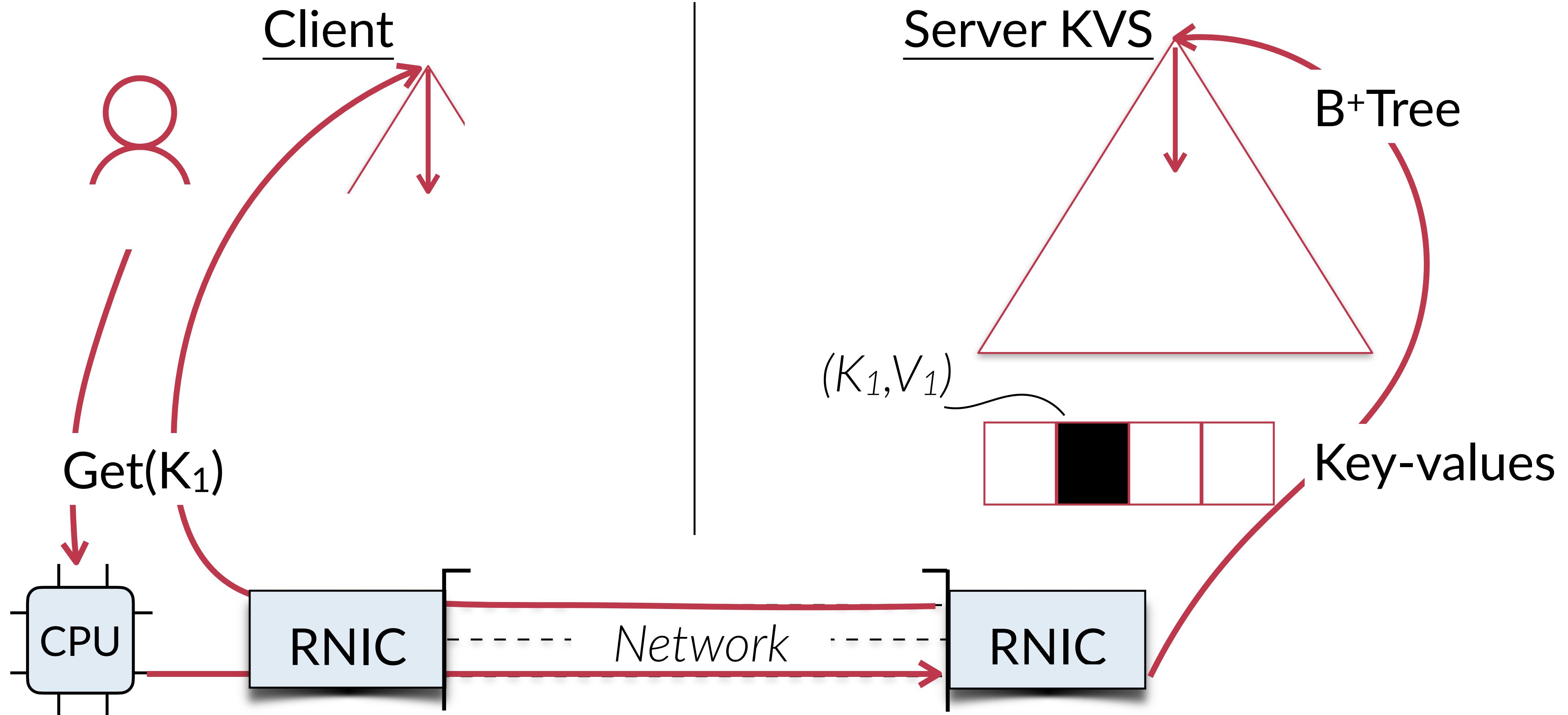
⌚ e.g., *B⁺Tree, O(log(n))*^[2] network RTT



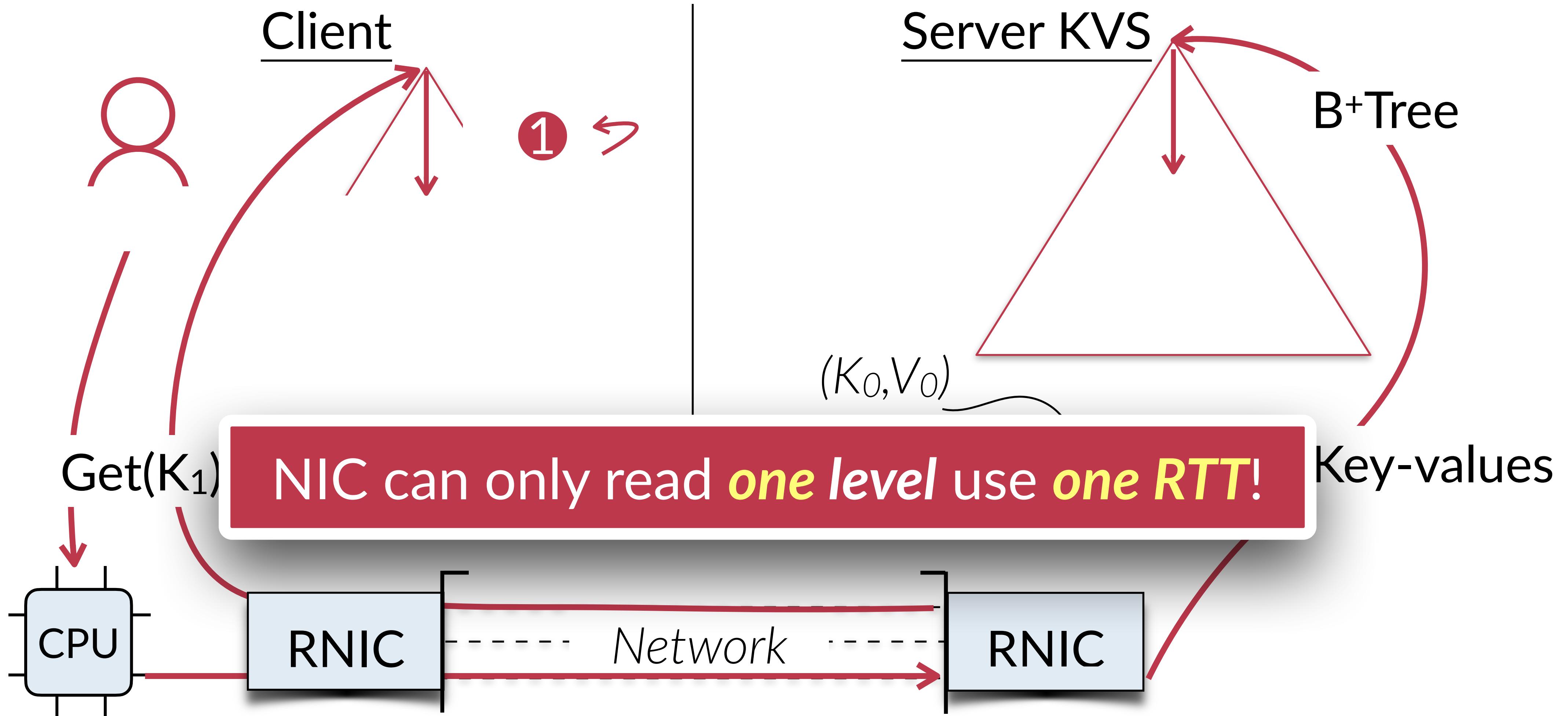
[1] RTT: roundtrip time

[2] n:the scale of the KVS

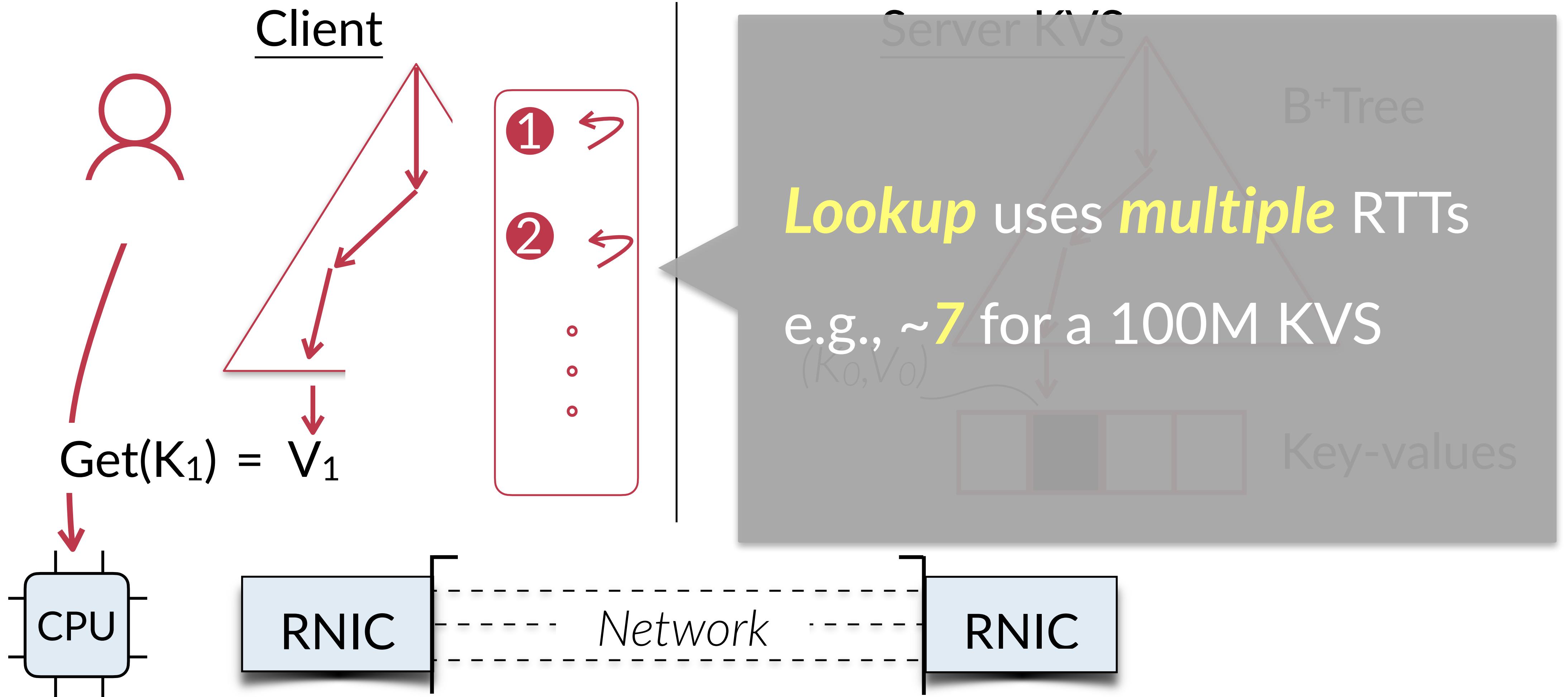
Challenge: limited NIC abstraction



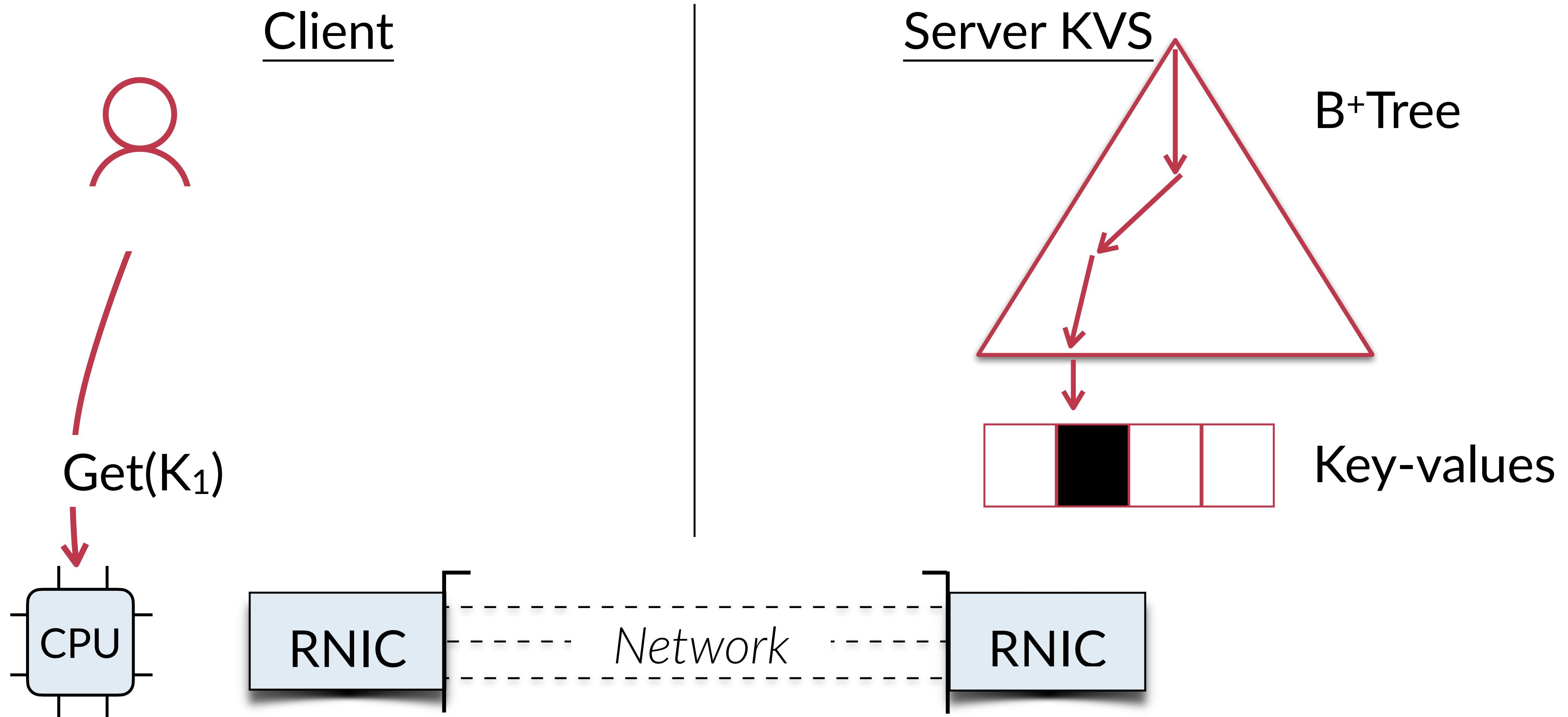
Challenge: limited NIC abstraction



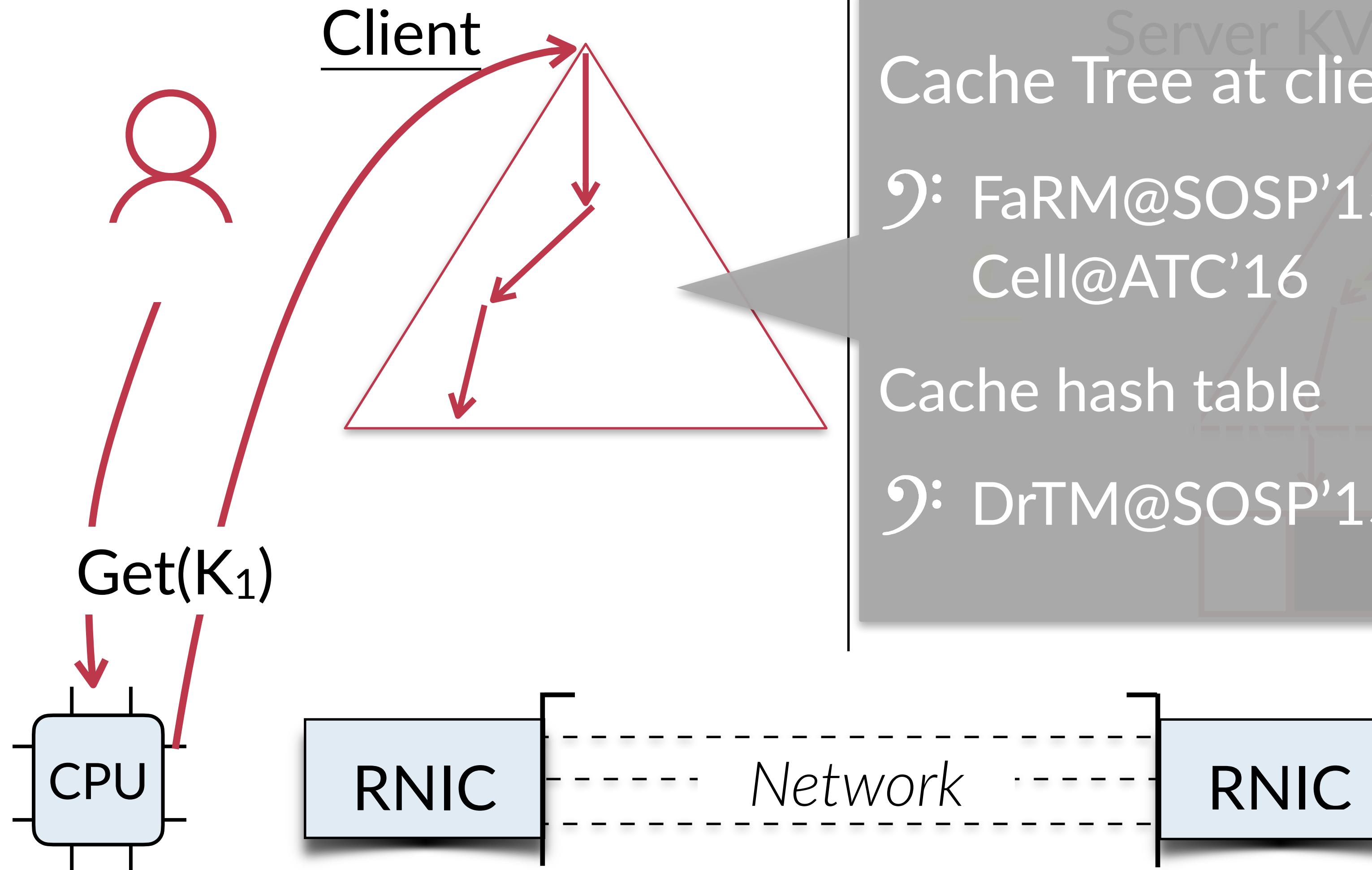
Challenge: limited NIC abstraction



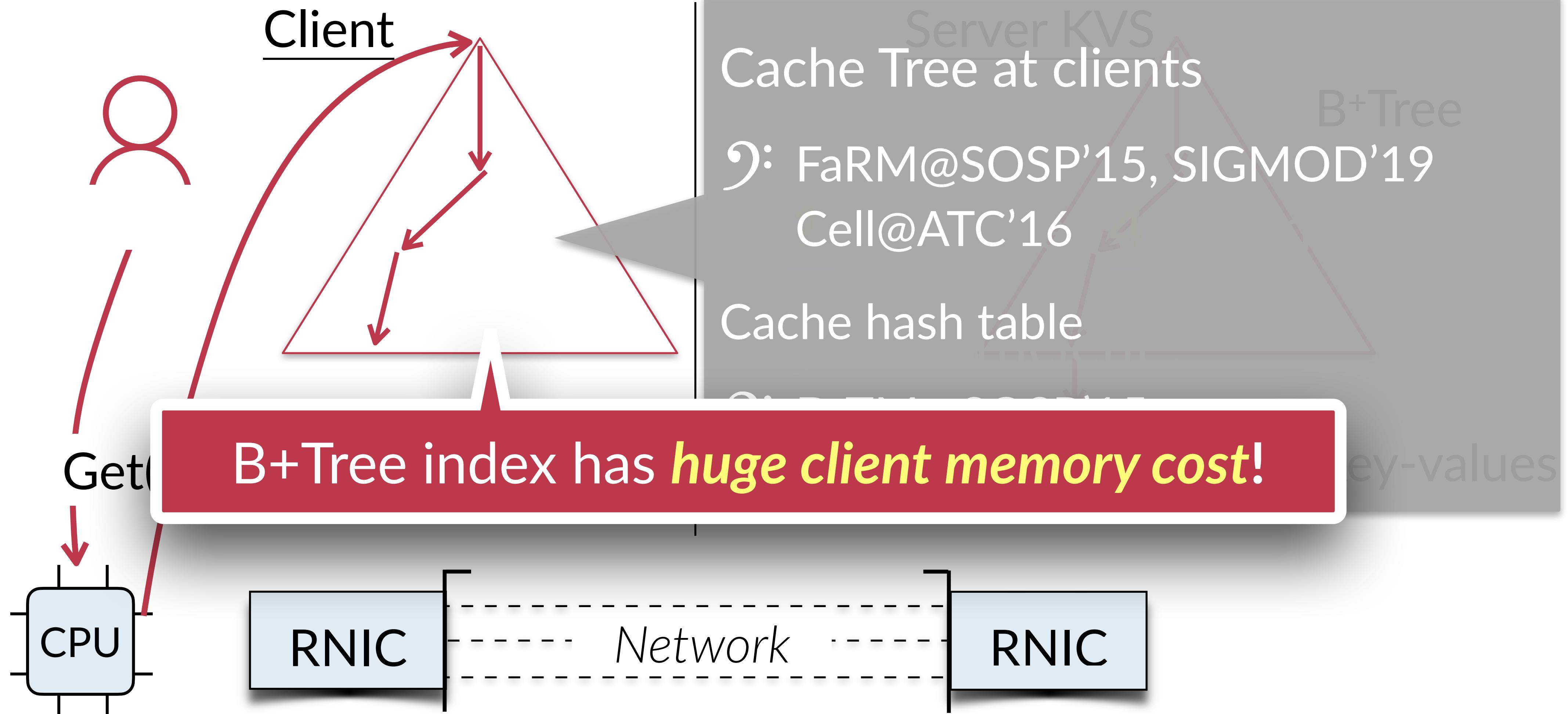
Existing systems adopt caching



Existing systems adopt caching



Existing systems adopt caching



High cache miss cost for caching tree

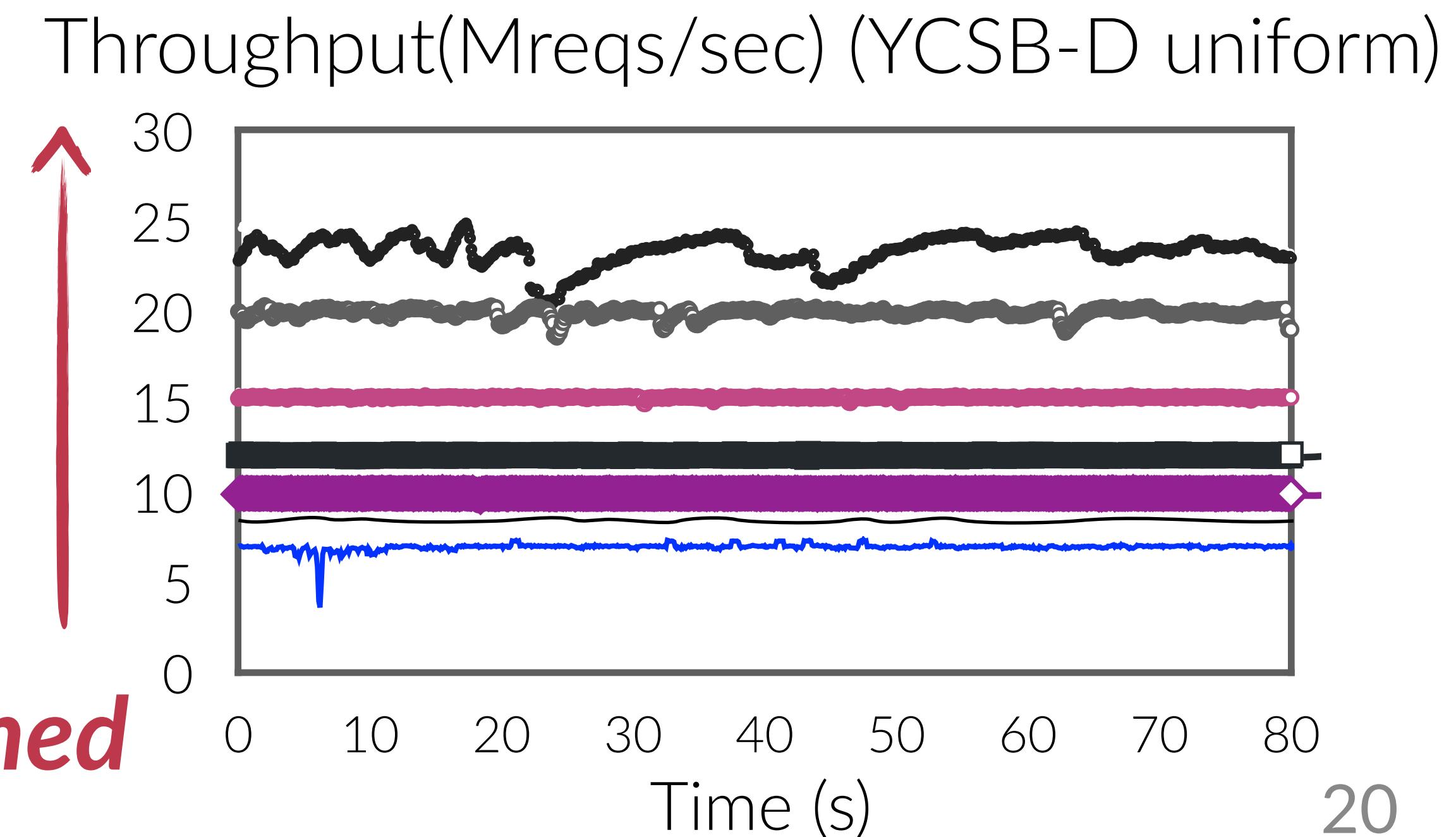
Tree node size can be *much larger* than the KV

- ⌚ e.g., 1K vs. 8B

Recursive invalidation under insertions

- ⌚ When cache more tree layers

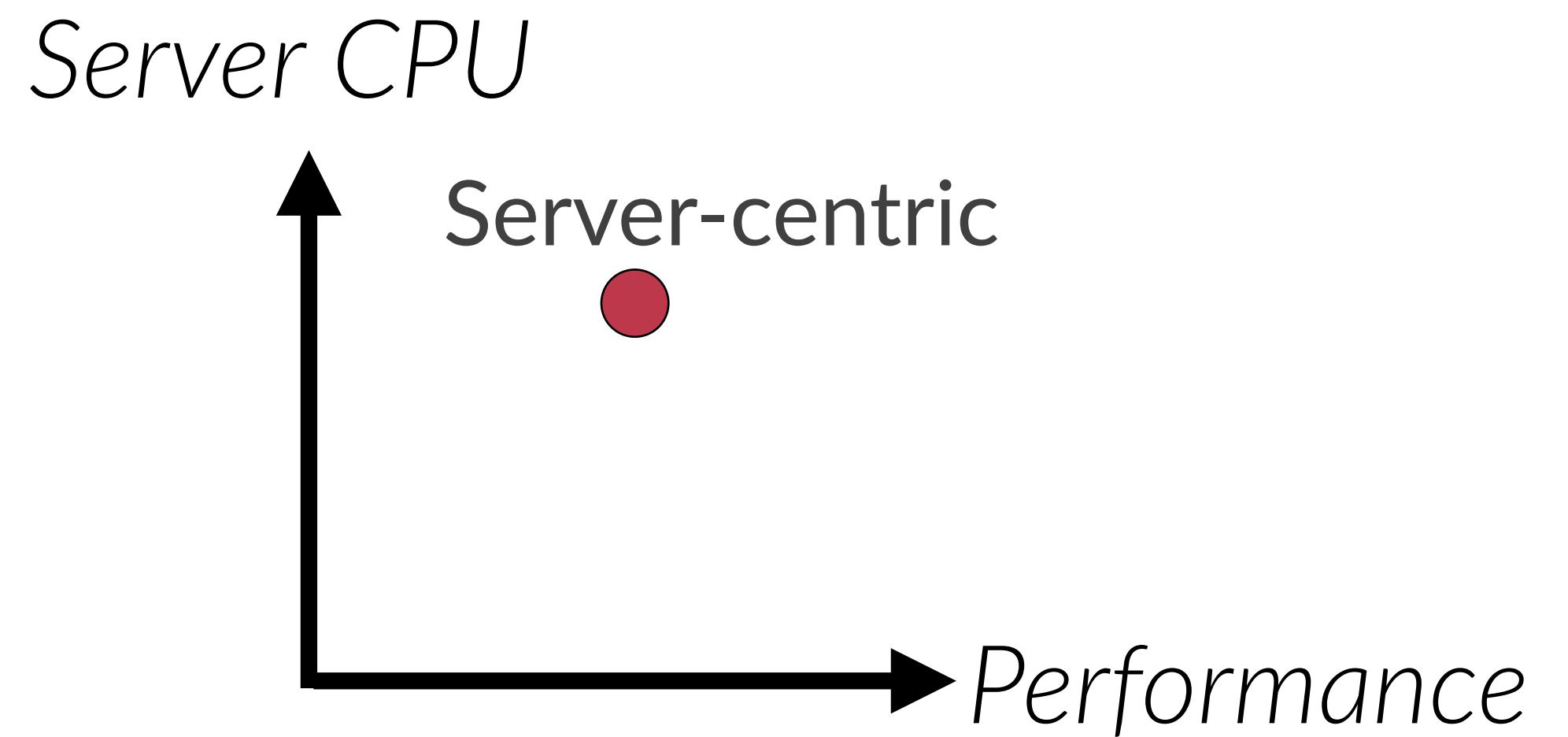
More layer cached



Trade-off of existing KVS

Server-centric KVS

⌚ High CPU utilizations



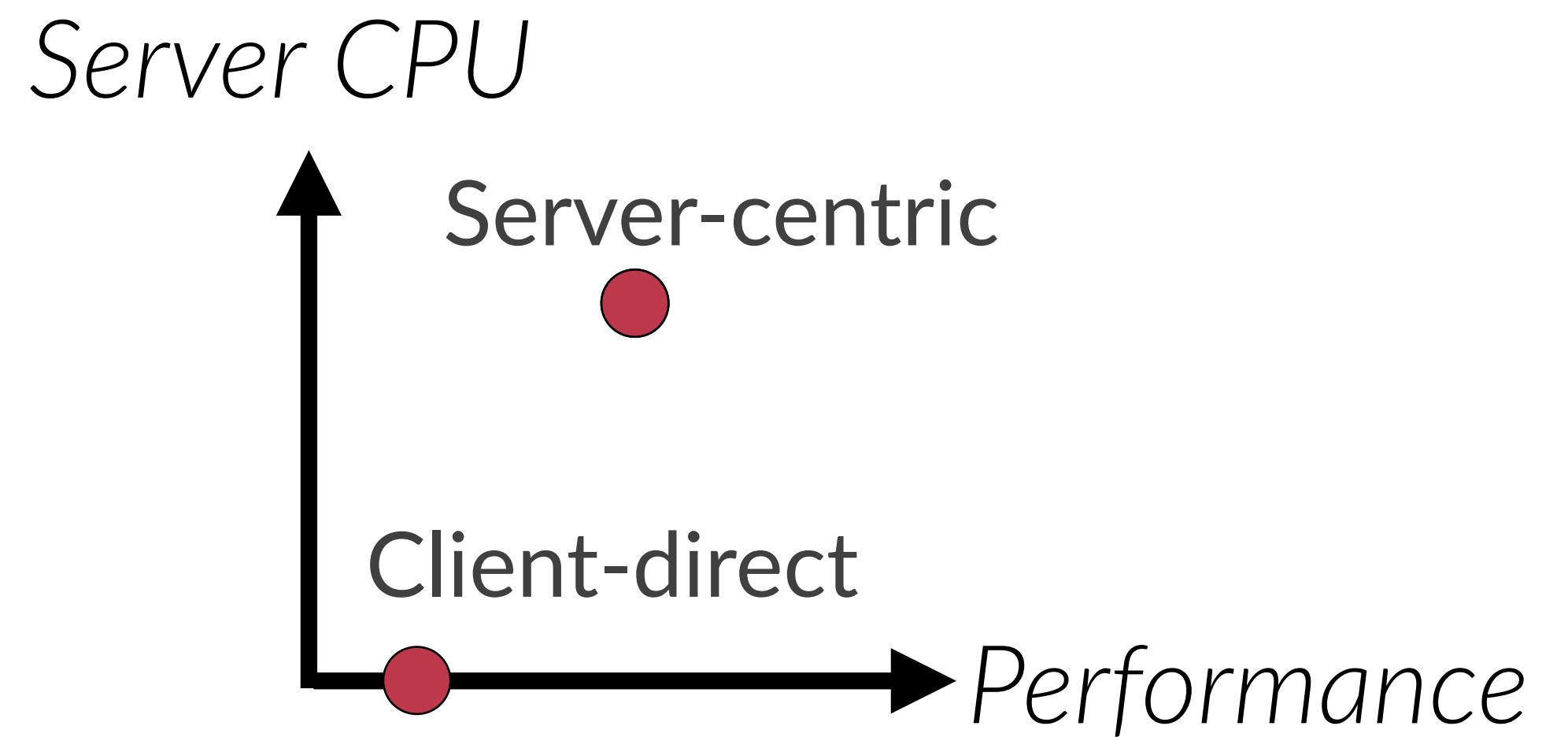
Trade-off of existing KVS

Server-centric KVS

⌚ High CPU utilizations

Client-direct KVS

⌚ Poor performance



Trade-off of existing KVS

Server-centric KVS

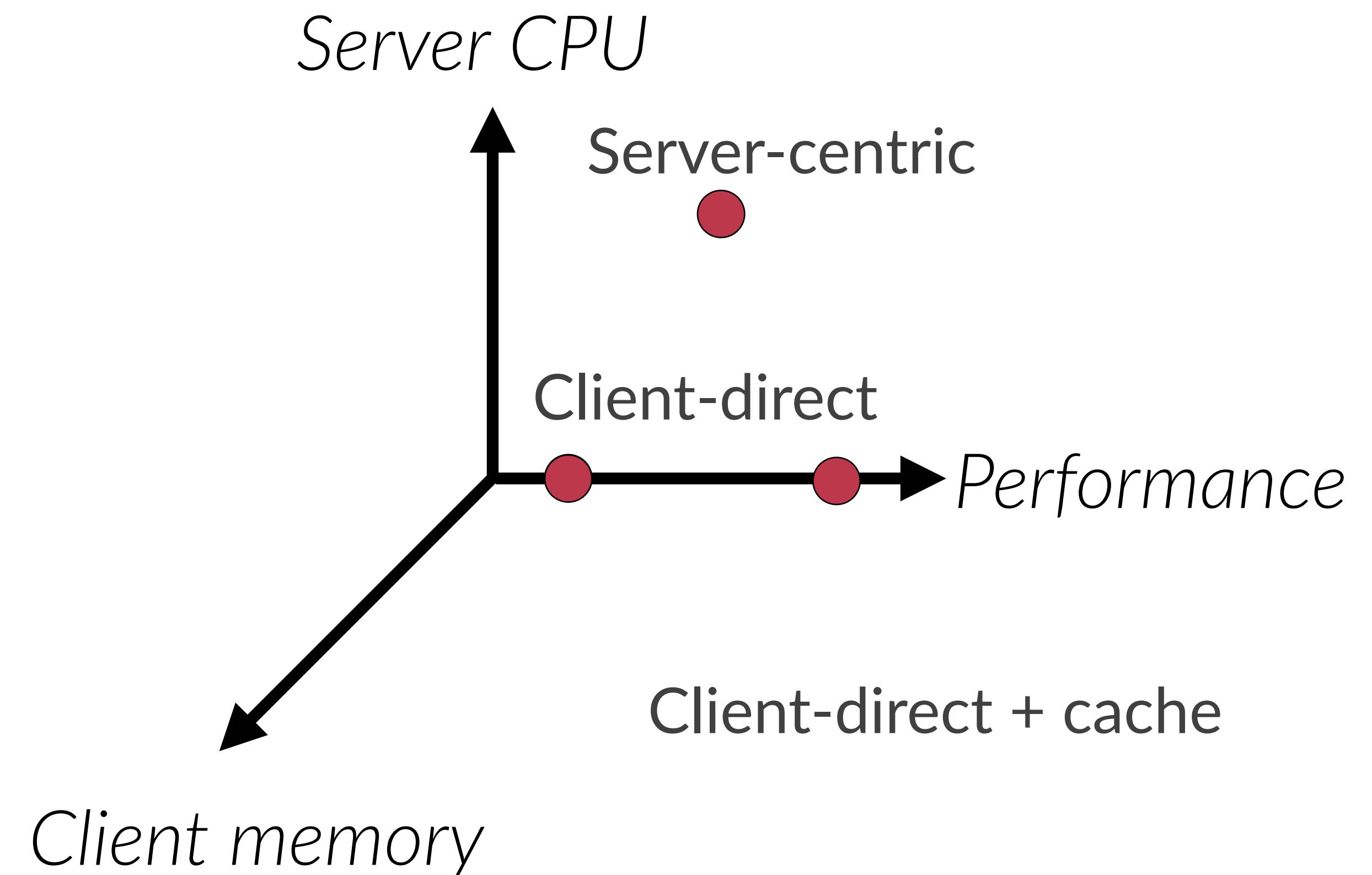
- ⌚ High CPU utilizations

Client-direct KVS

- ⌚ Poor performance

Client-direct KVS + cache

- ⌚ High memory usage



Trade-off of existing KVS

Server-centric KVS

⌚ High CPU utilizations

Client-direct KVS

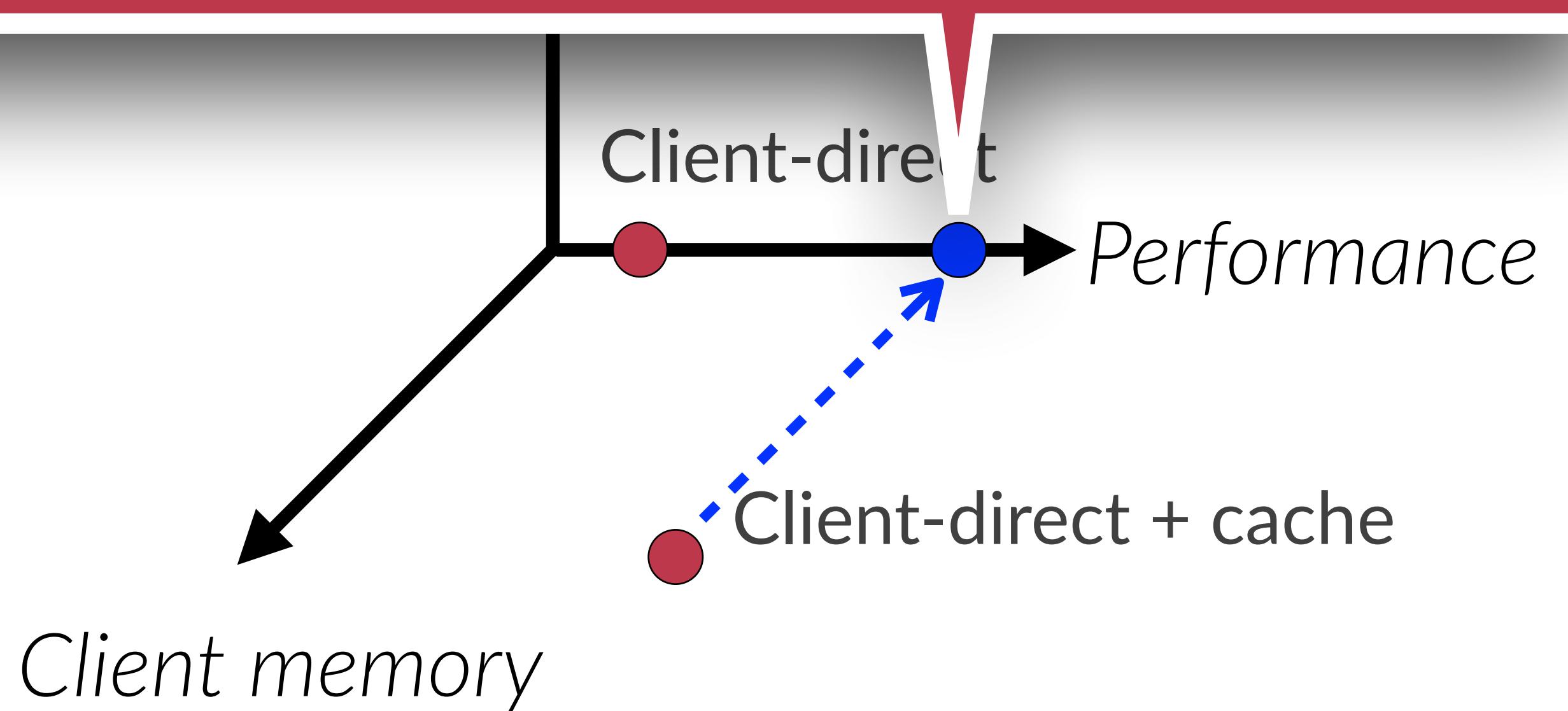
⌚ Poor performance

Client-direct KVS + cache

⌚ High memory usage

Server CPU

Can we achieve all these properties ?



Trade-off of existing KVS

Server-centric KVS

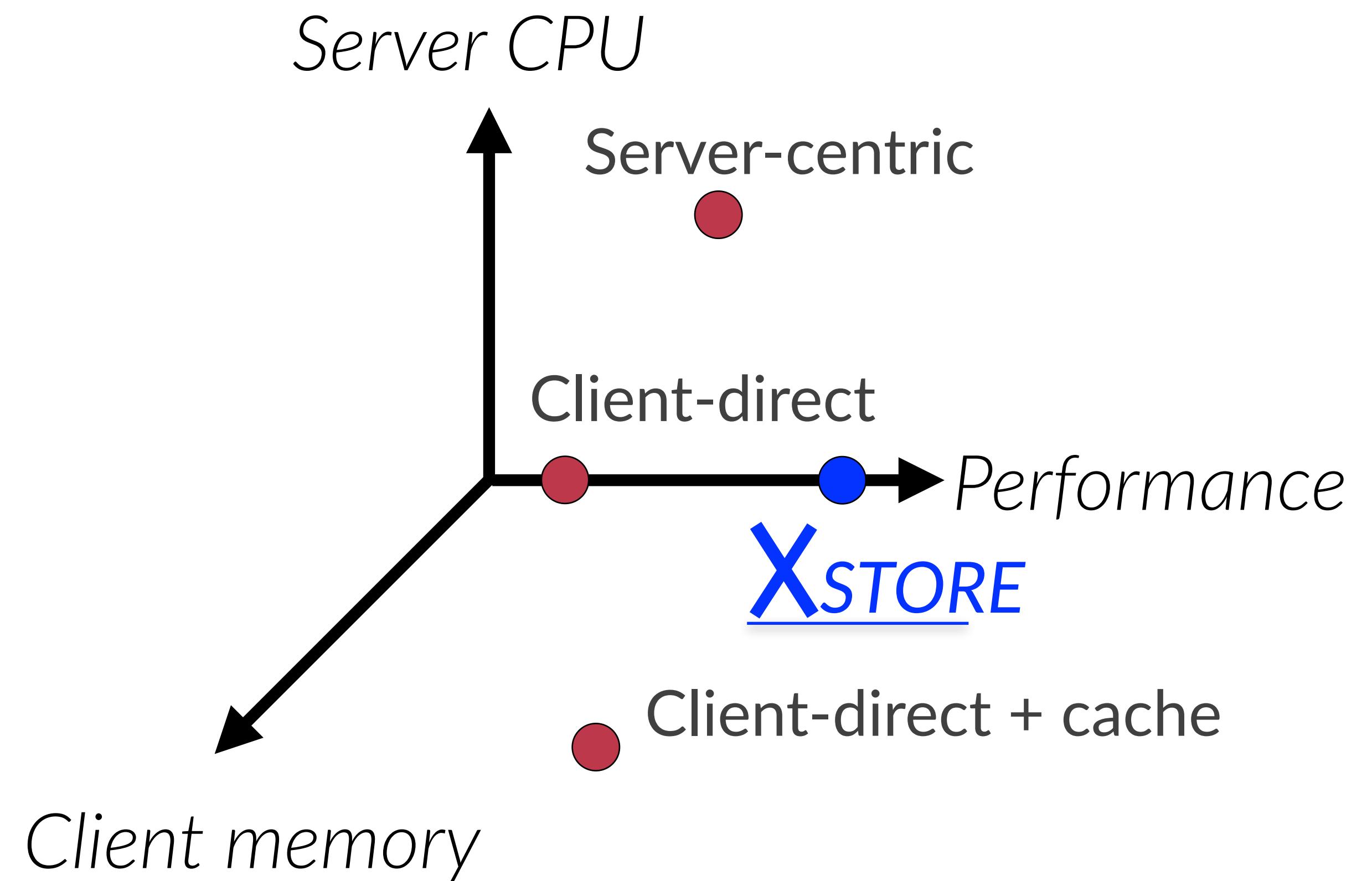
- ⌚ High CPU utilizations

Client-direct KVS

- ⌚ Poor performance

Client-direct KVS + cache

- ⌚ High memory usage

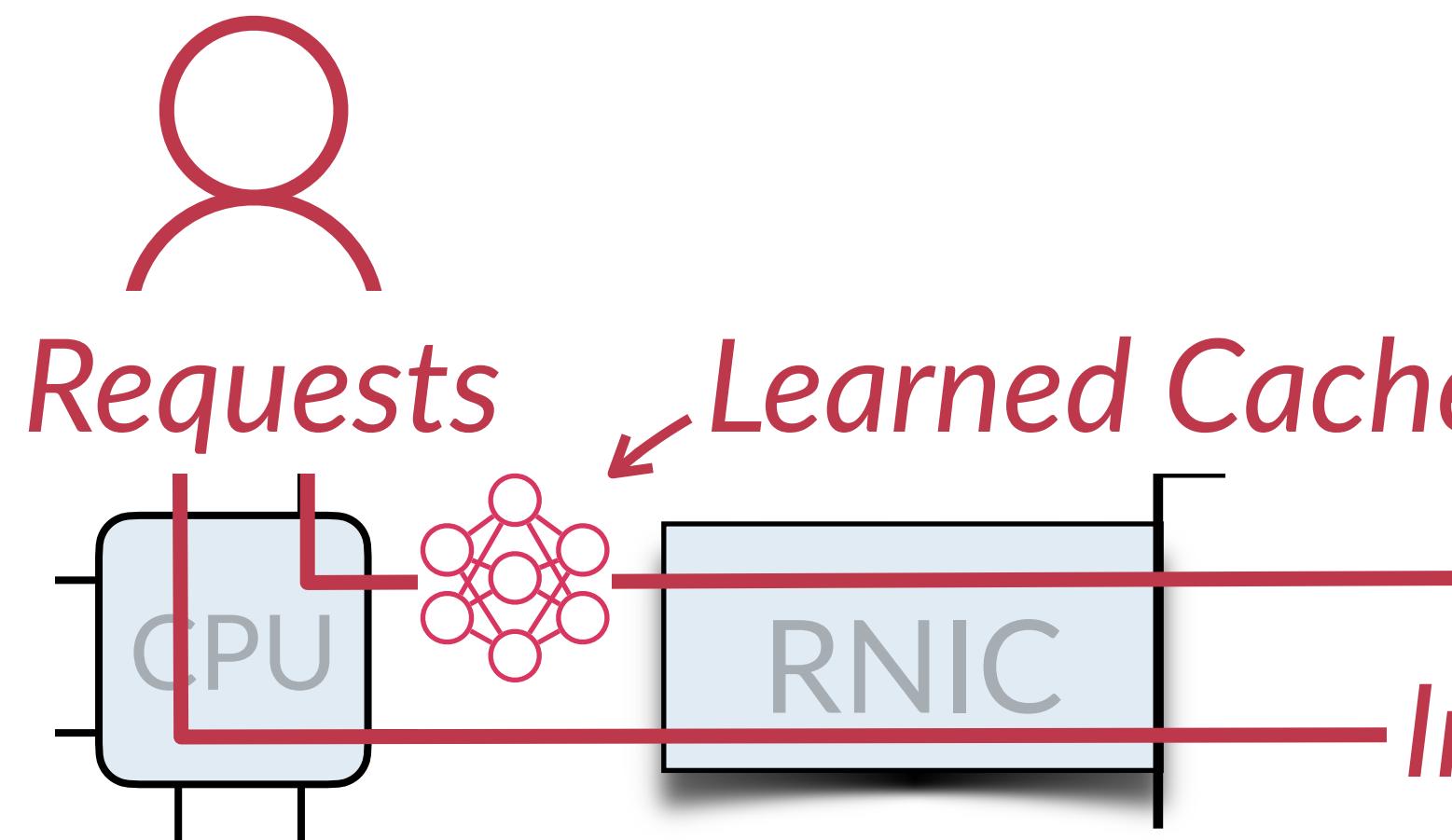
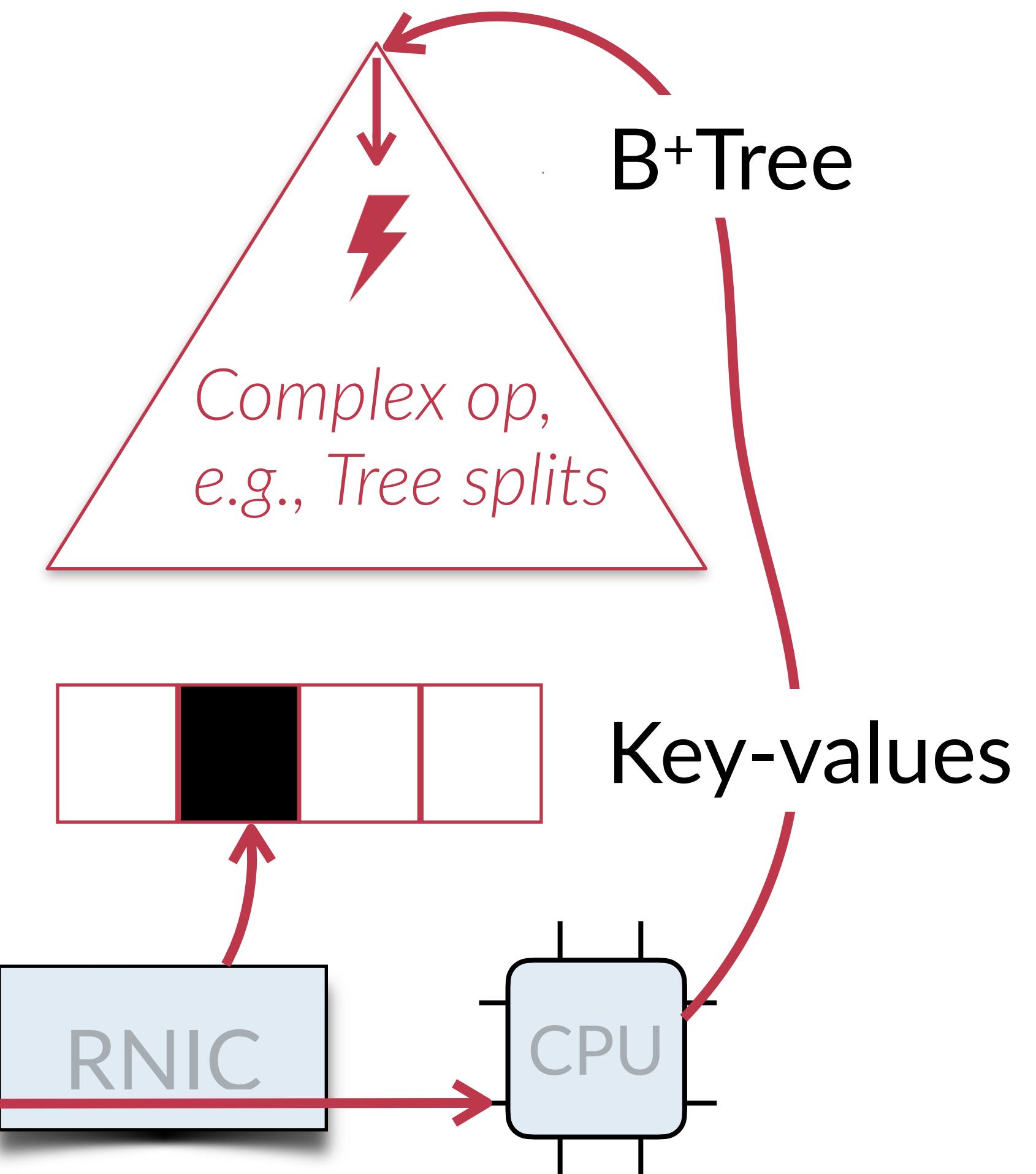


Overview of XSTORE

Hybrid architecture [1]

- ⌚ **Server-centric** updates
- ⌚ Because one-sided has **simple semantic**

O(1) Client-direct Get,Scan

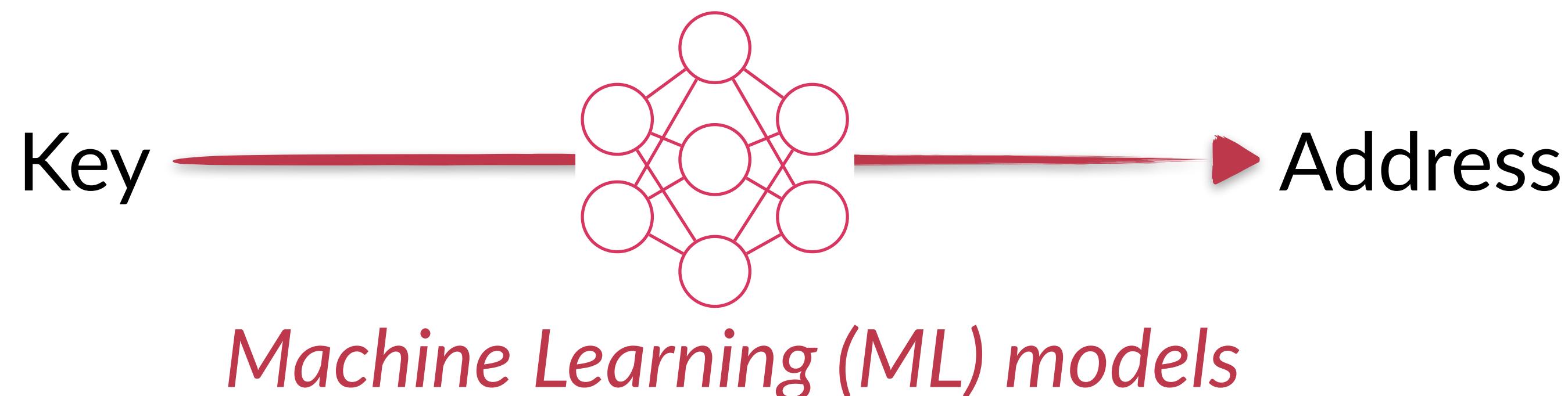


Our approach: Learned cache

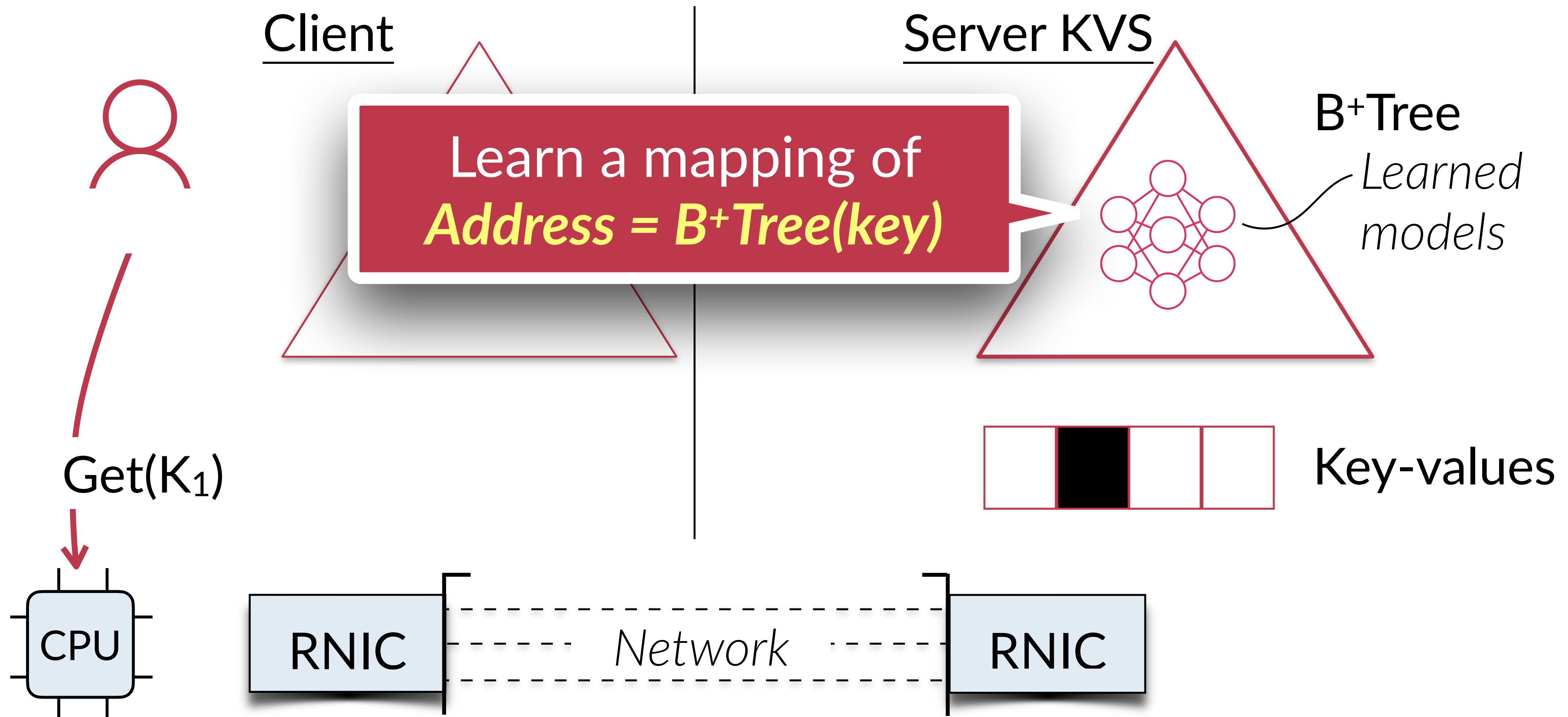
Using **ML** as the cache structure for tree-based index

Motivated by the **learned index**^[1]

- ⌚ Replace **index traversal** with **calculation**
- ⌚ The ML model can be **orders of magnitude smaller** than tree



Client-direct Get() using learned cache

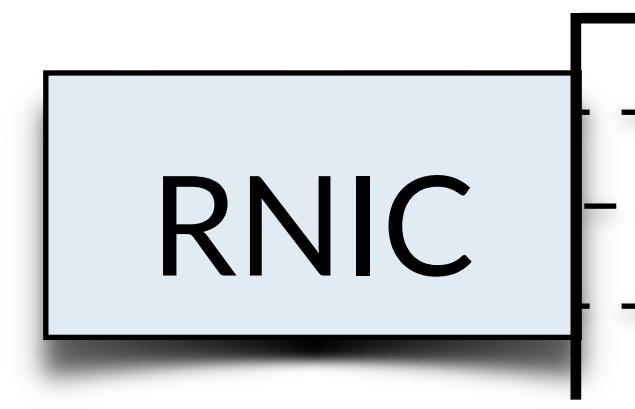
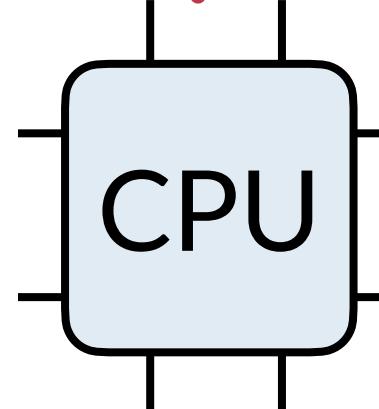


Client-direct Get() using learned cache

Learned model with
small memory

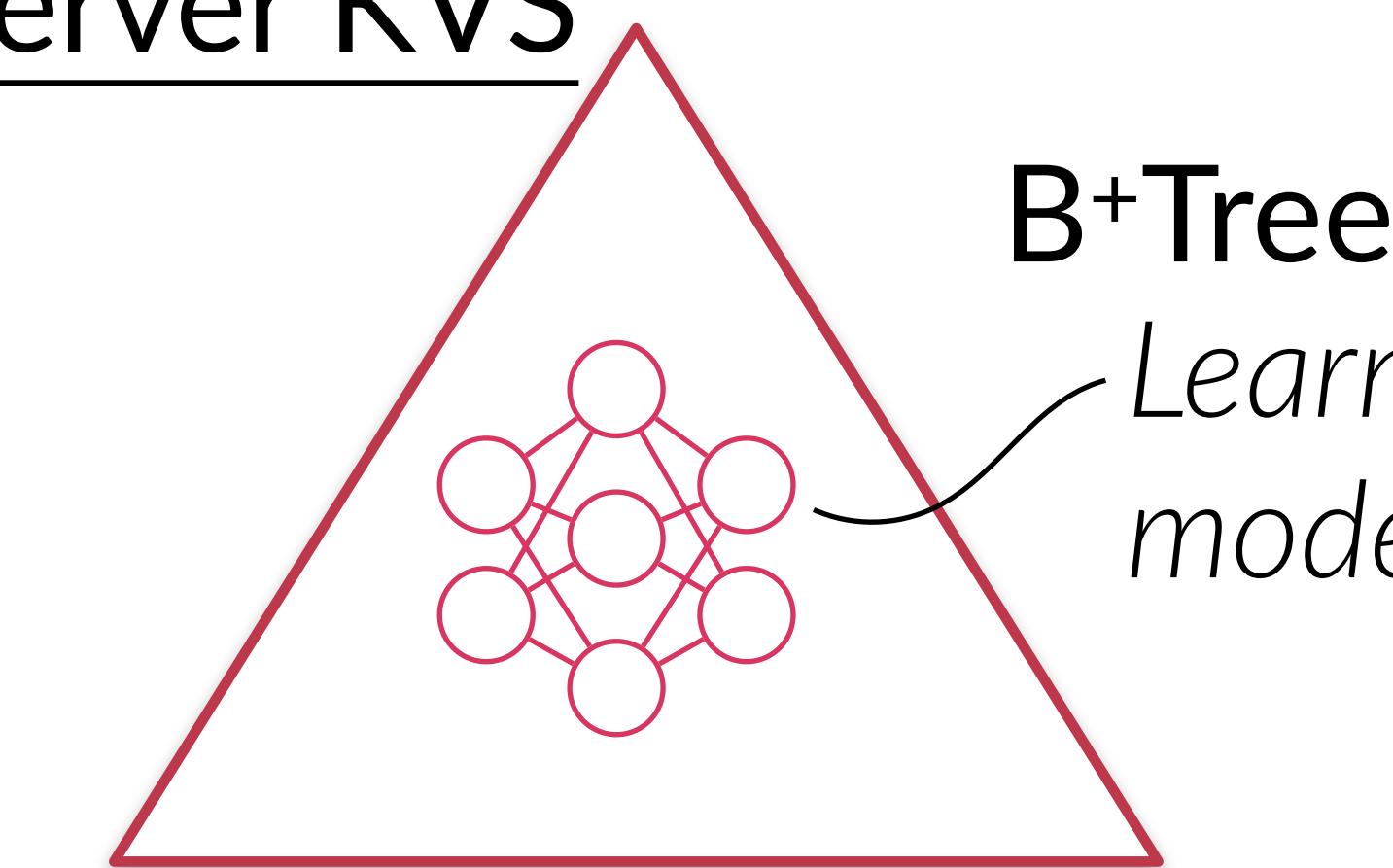


Get(K_1)



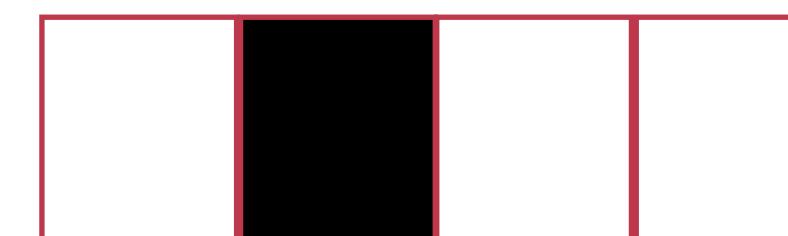
Network

Server KVS



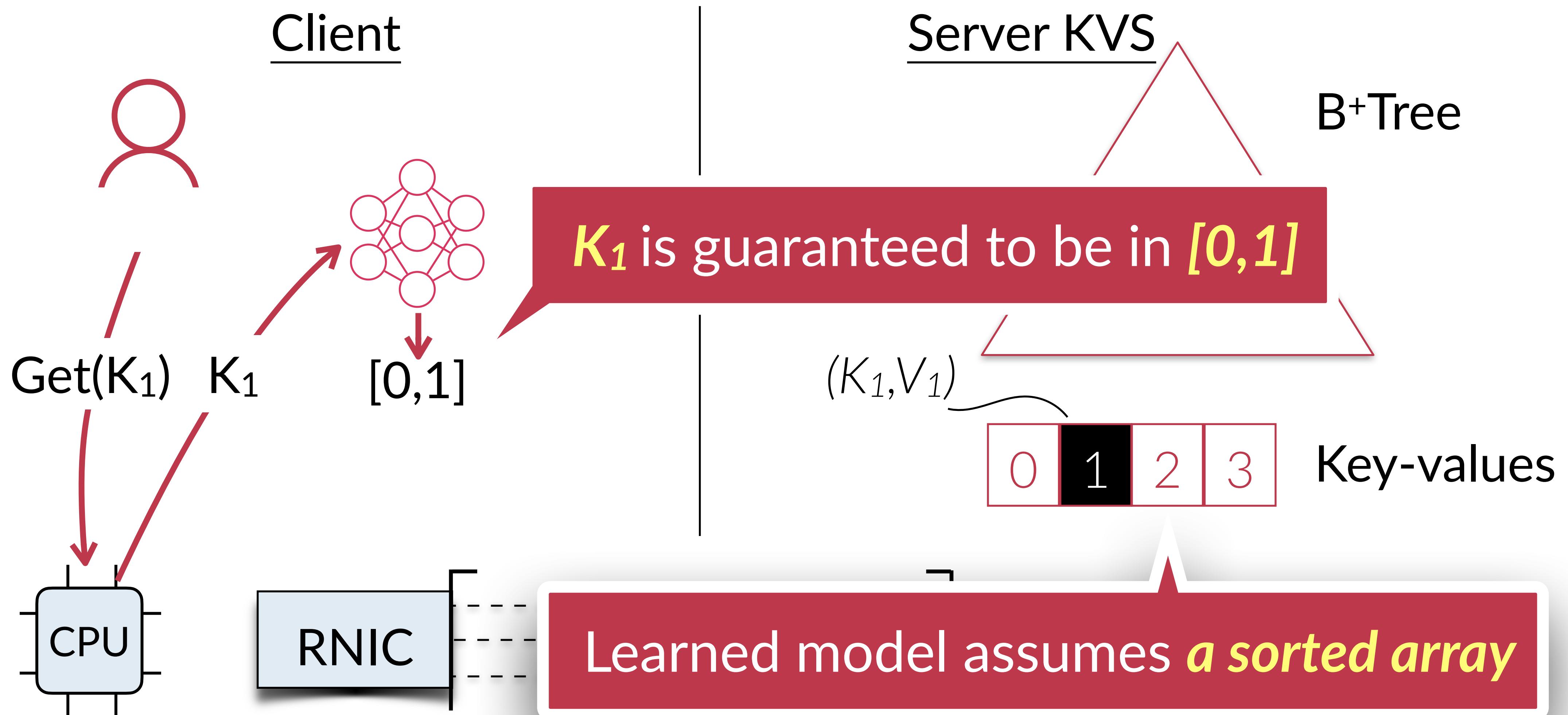
B⁺Tree

Learned
models

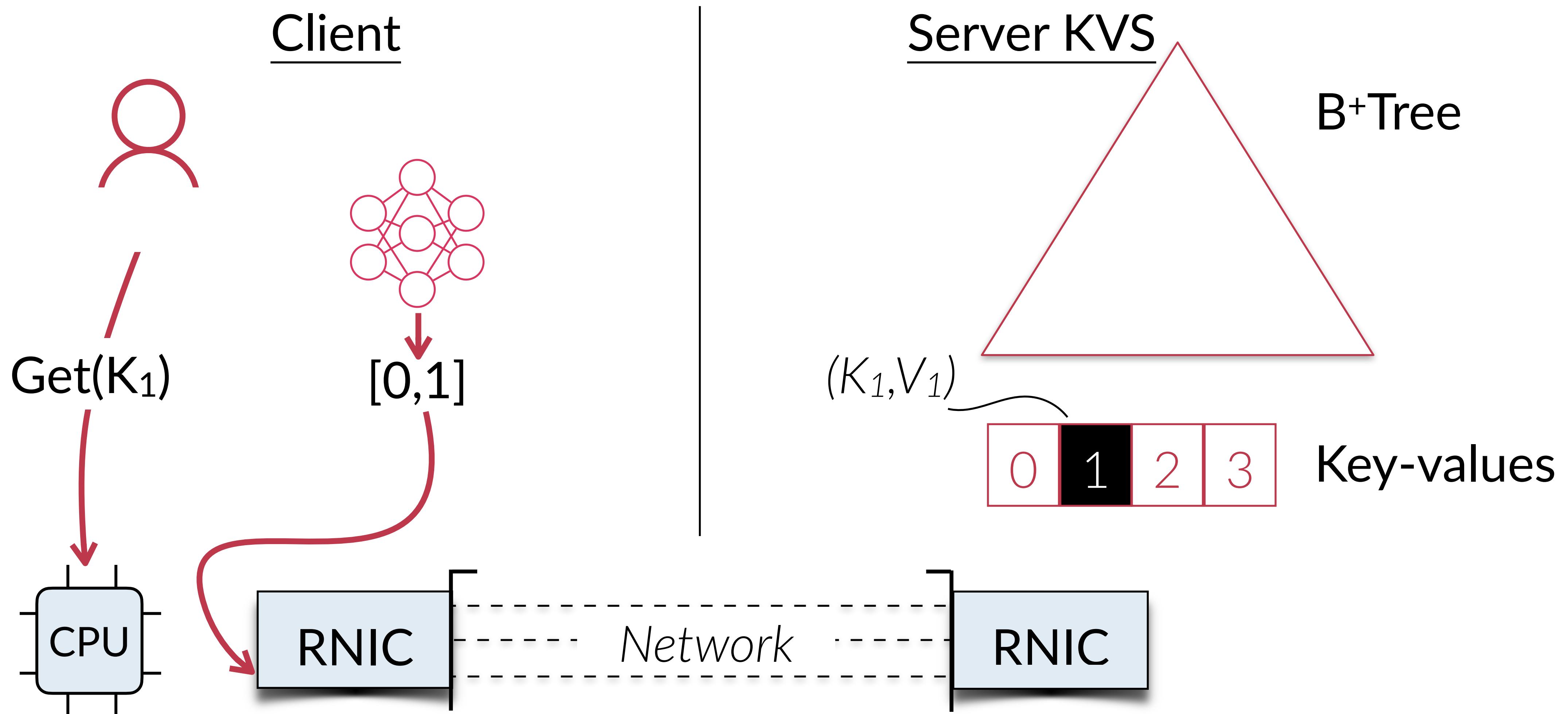


Key-values

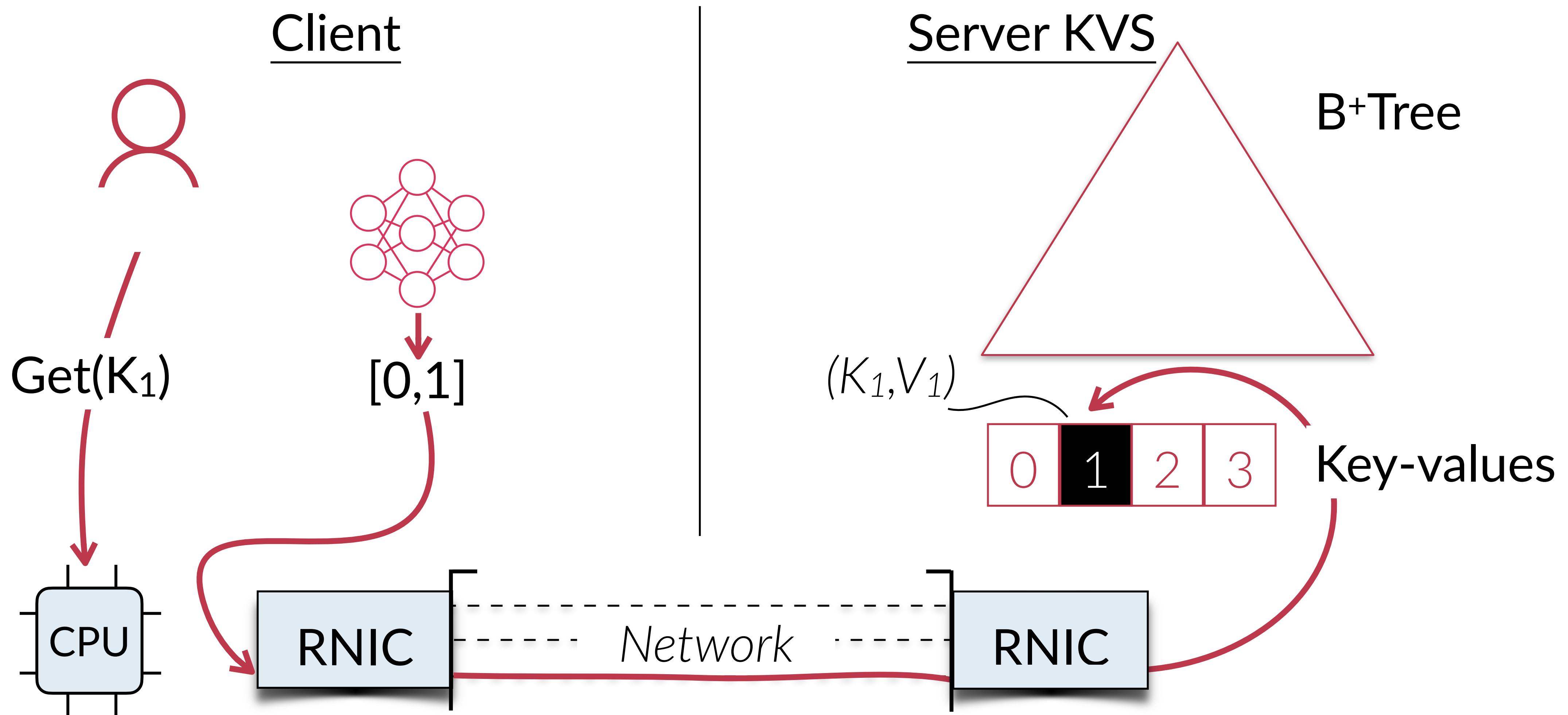
Client-direct Get() using learned cache



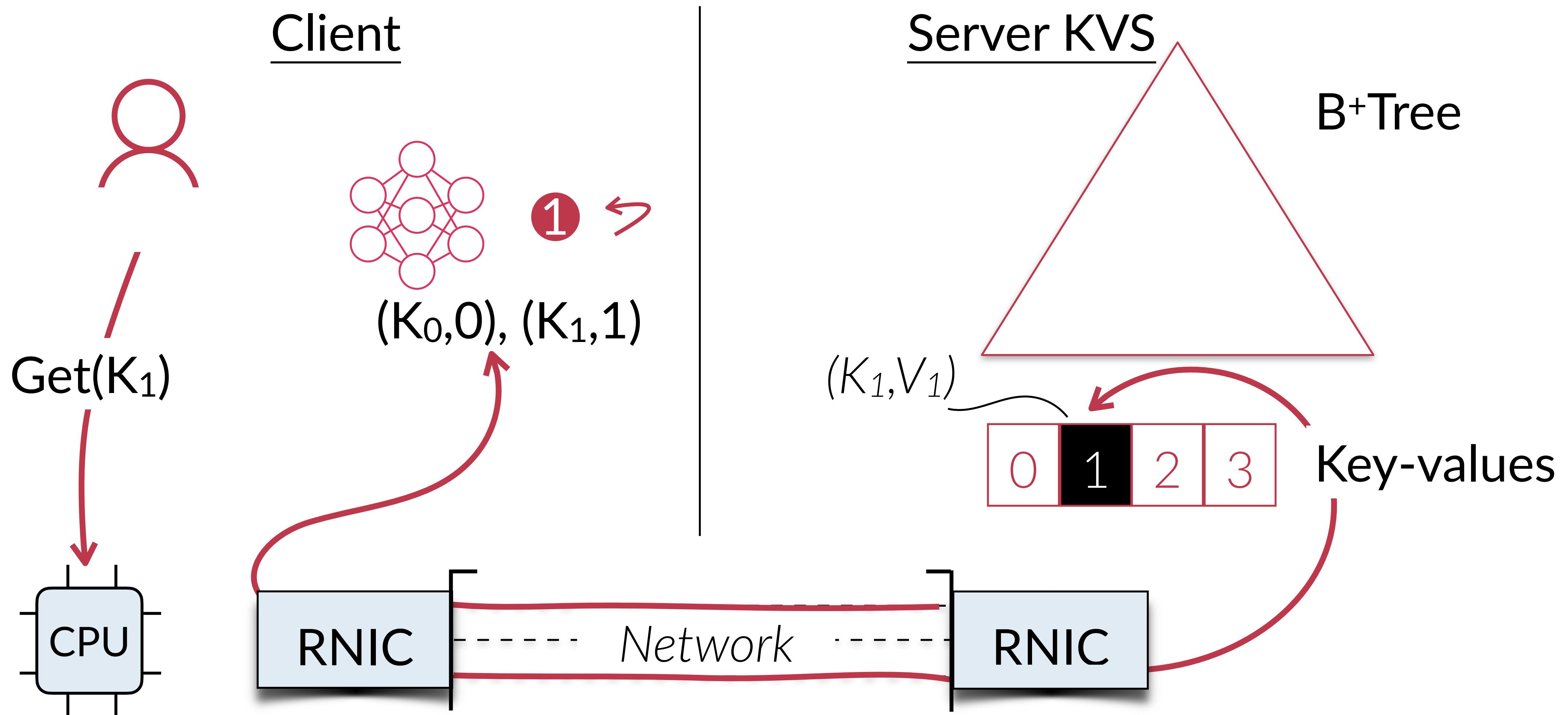
Client-direct Get() using learned cache



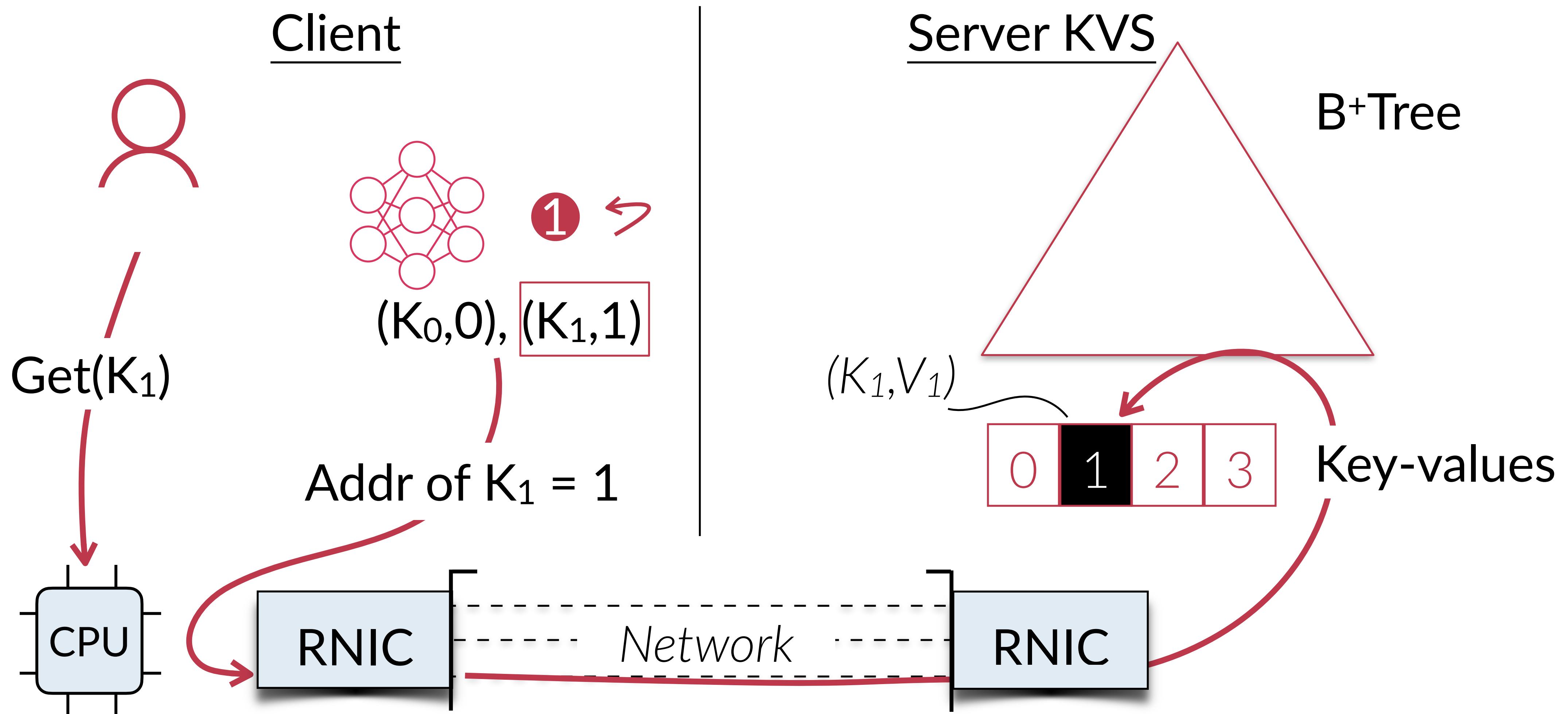
Client-direct Get() using learned cache



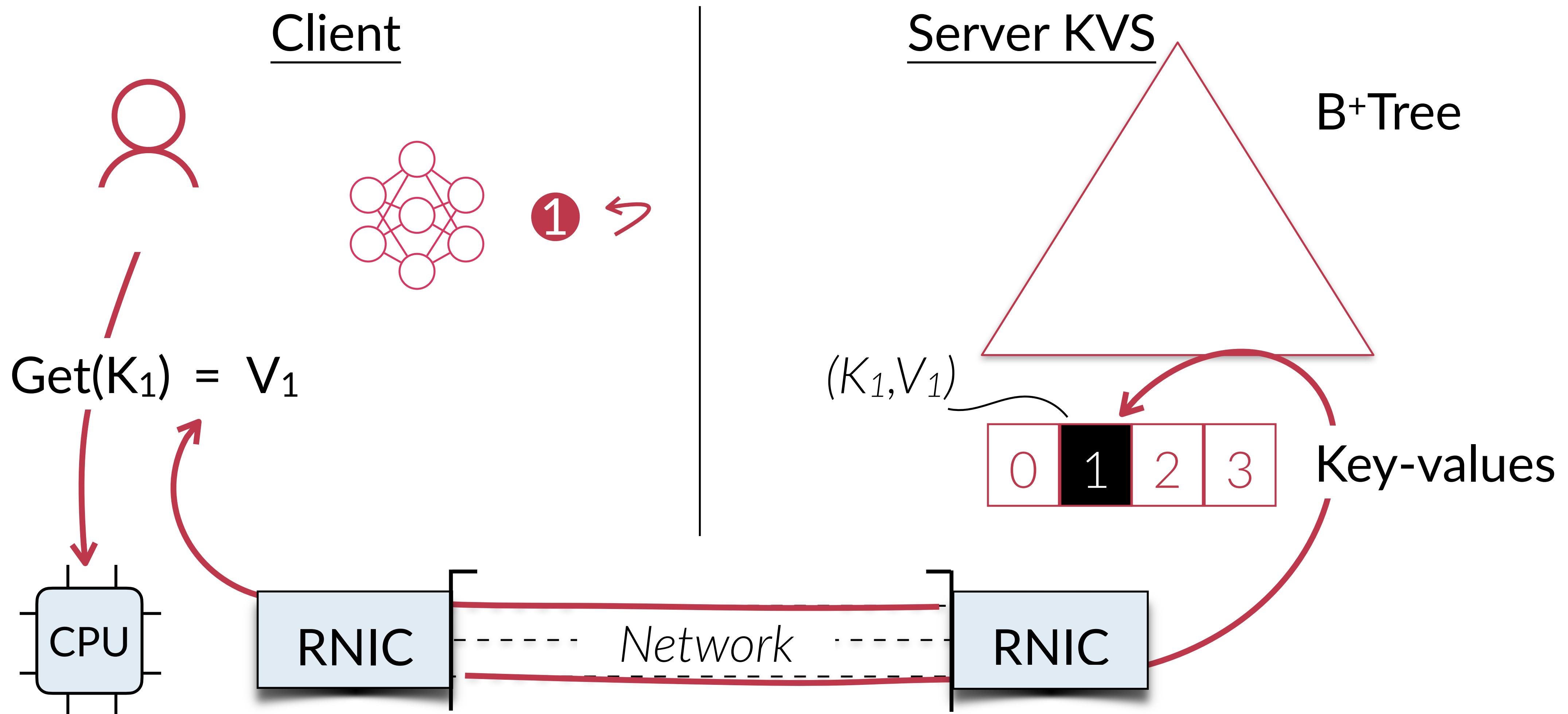
Client-direct Get() using learned cache



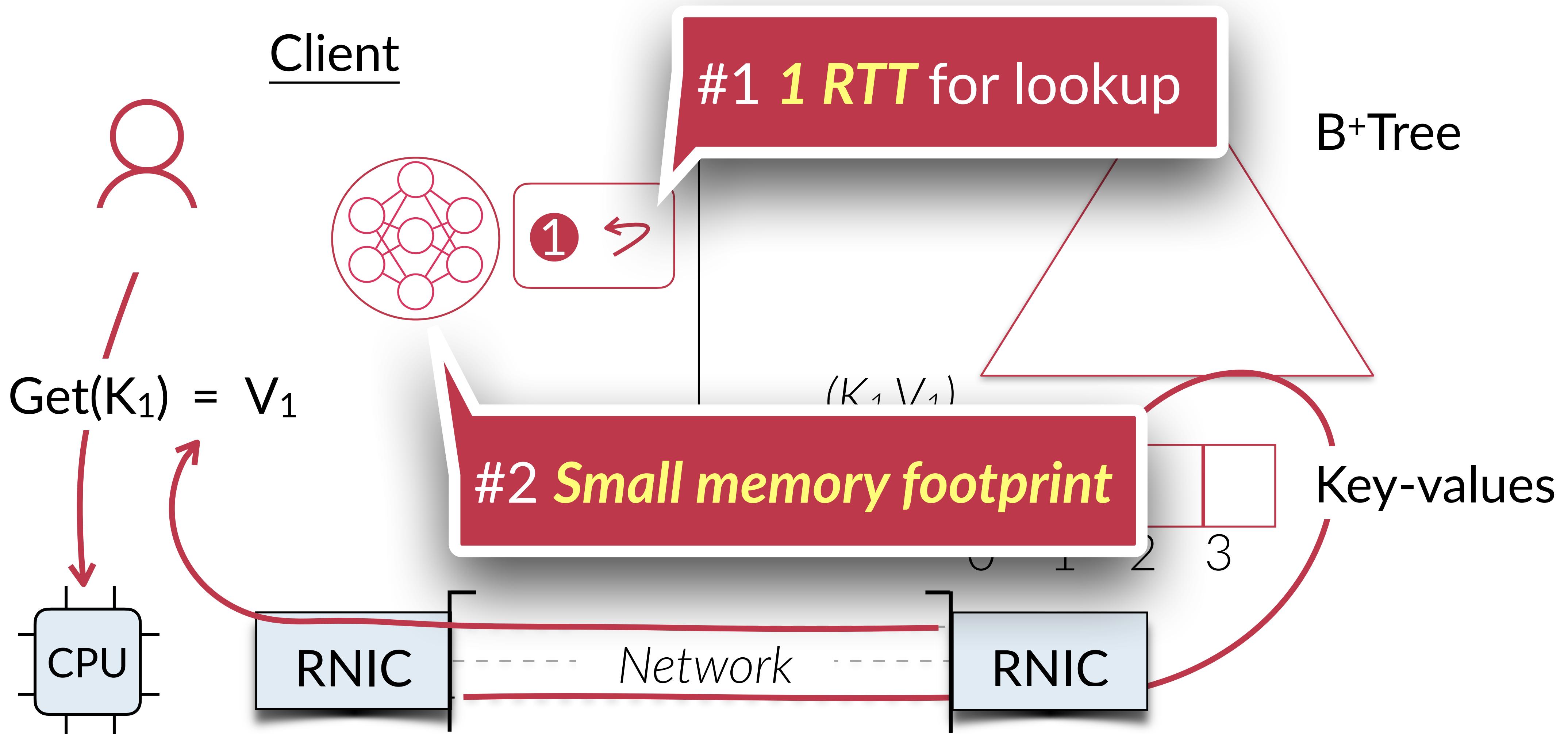
Client-direct Get() using learned cache



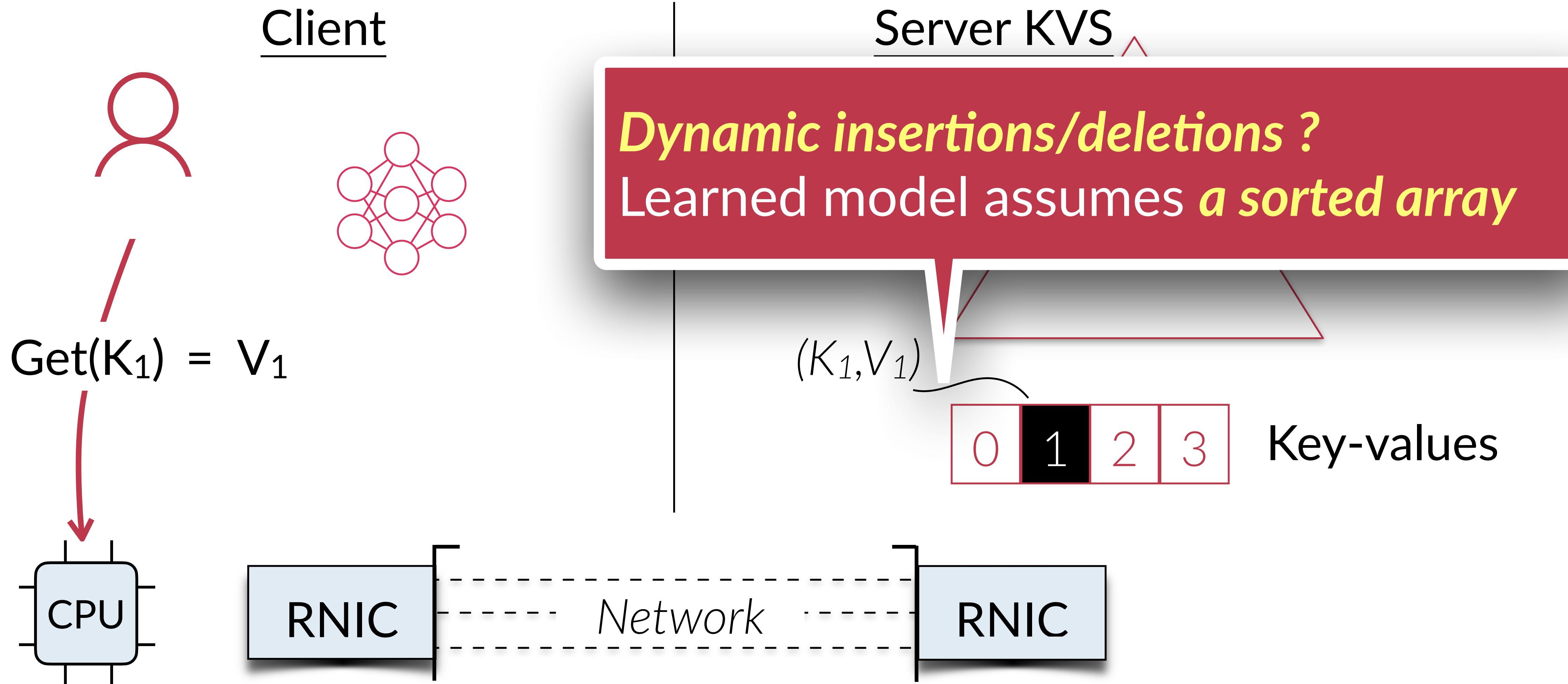
Client-direct Get() using learned cache



Benefits of the learned cache



Challenges of learned cache



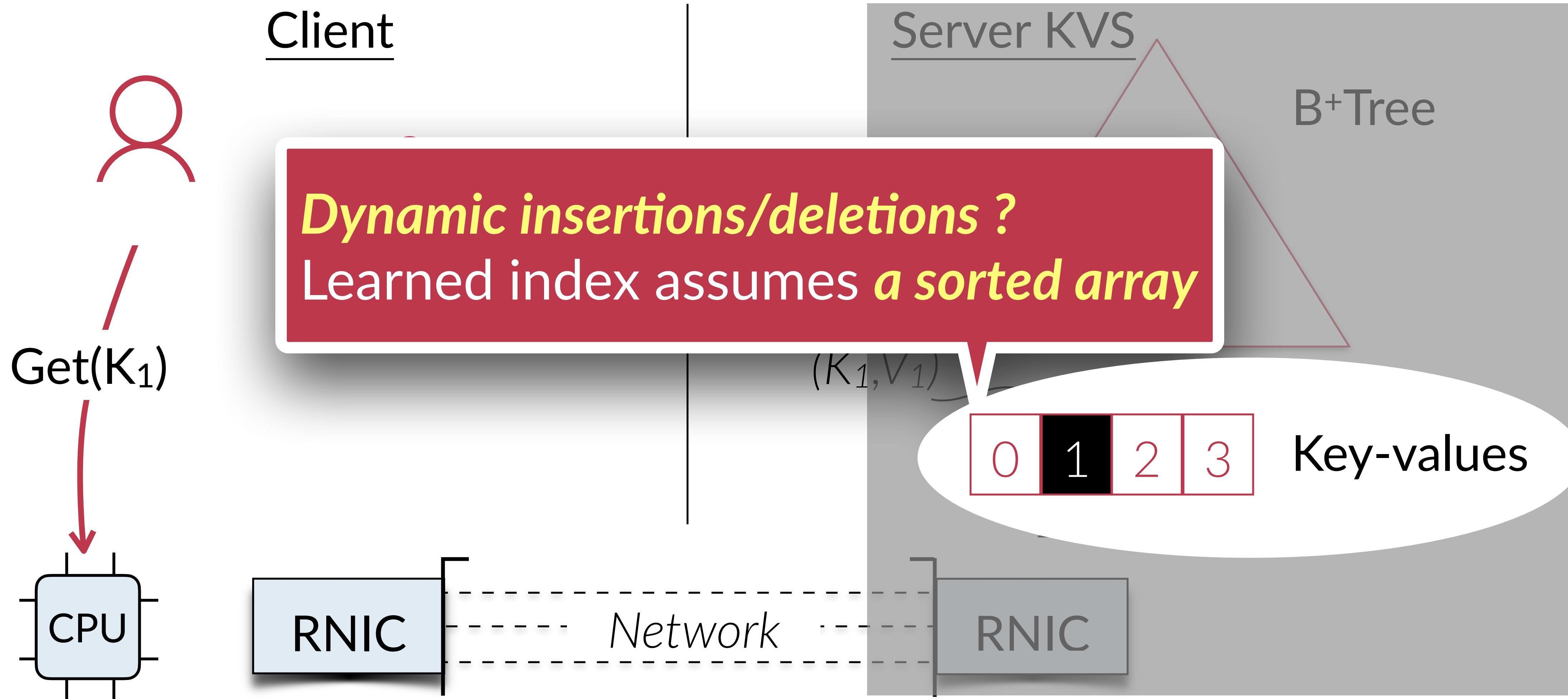
Outline of the remaining content

Server-side data structure for dynamic workloads

Client-side learned cache & TT

Performance evaluation of XSTORE

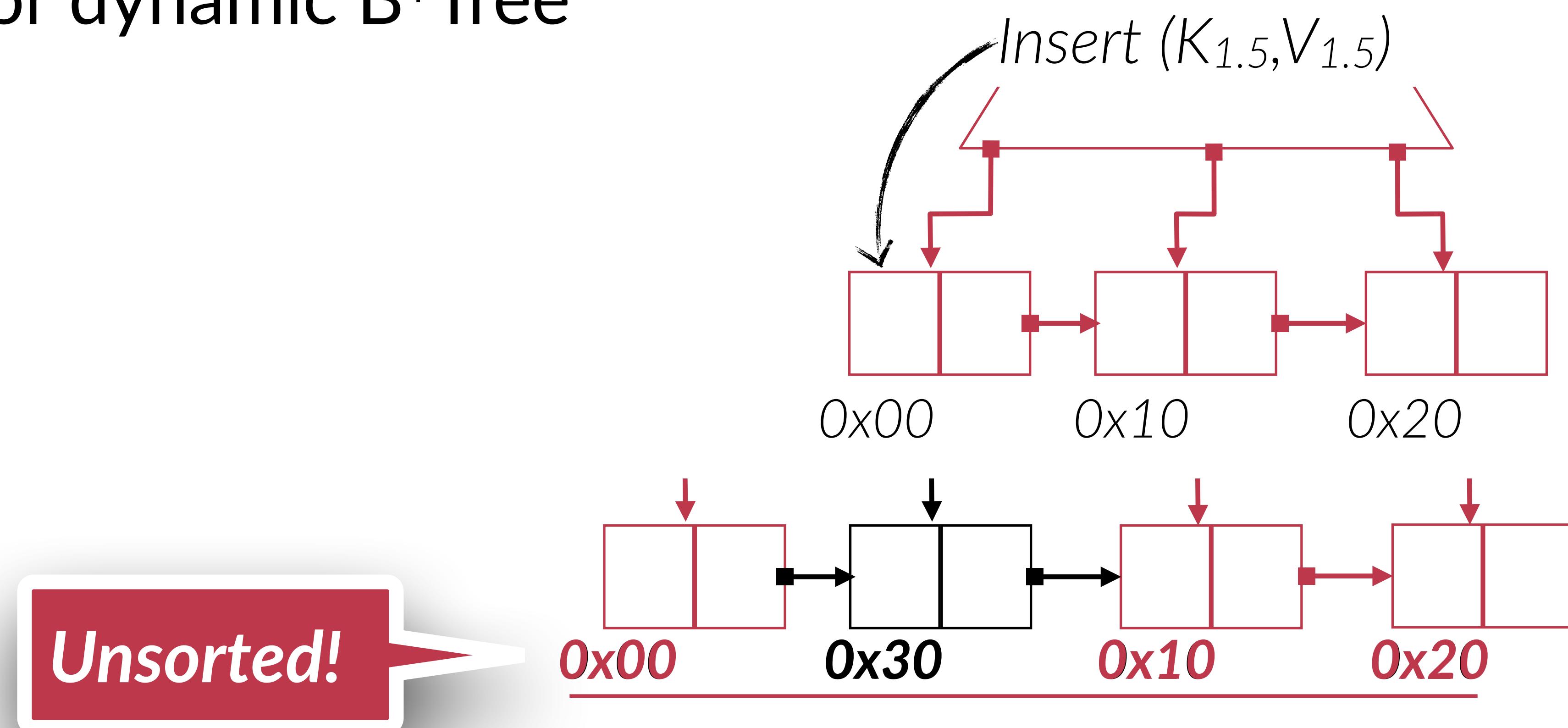
XSTORE stores KVs in B⁺Tree leaf nodes



Models cannot learn dynamic B+Tree address

Can only learn when the addresses are *sorted*

Not the case for dynamic B+Tree



Solution: another layer of indirection

Observation: leaf nodes are logically sorted

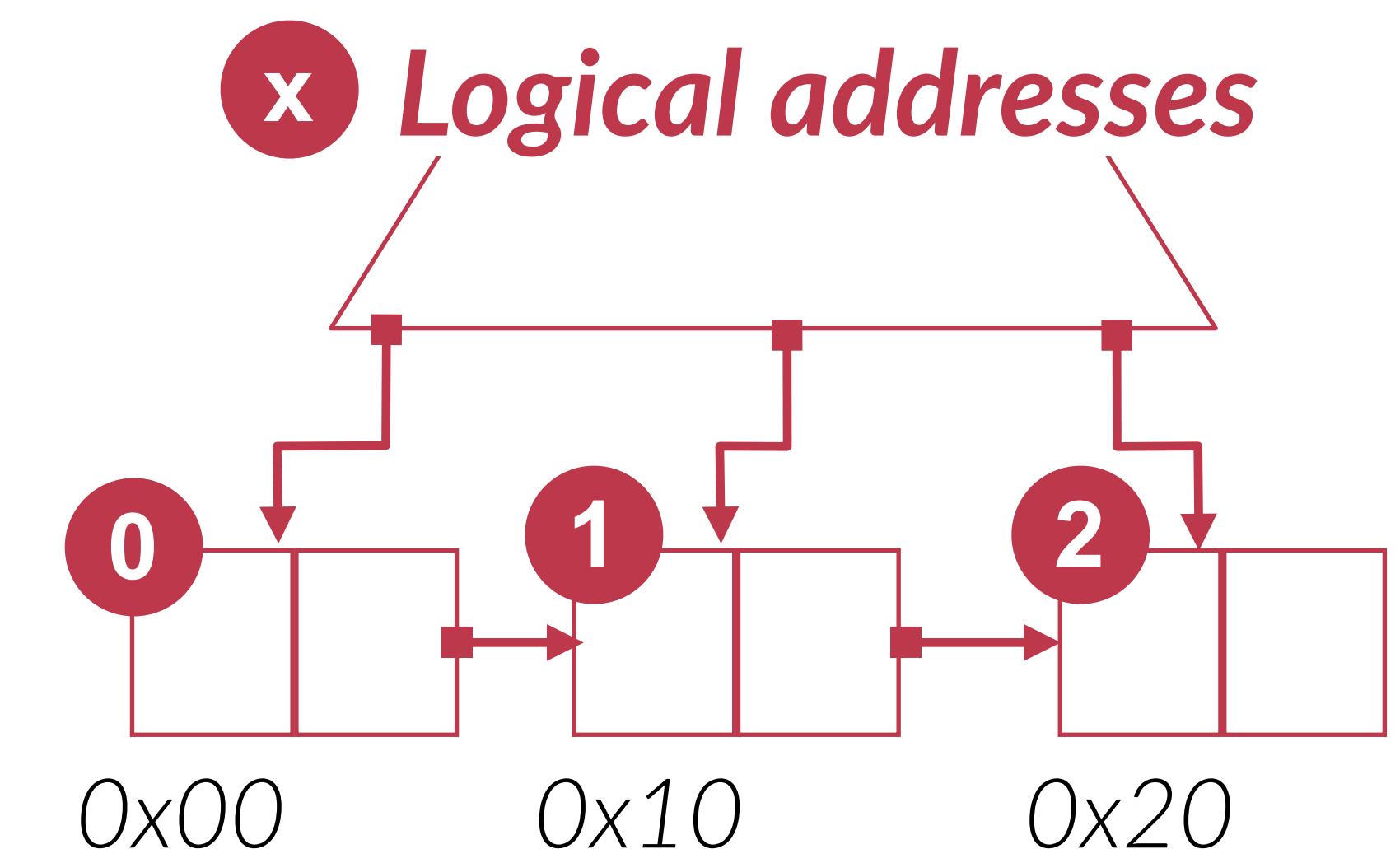
⌚ Assign logical addresses to leaf nodes

ML: key → logical

⌚ Translation table (TT): logical → physical

Translation Table

0x00	0x10	0x20
0	1	2



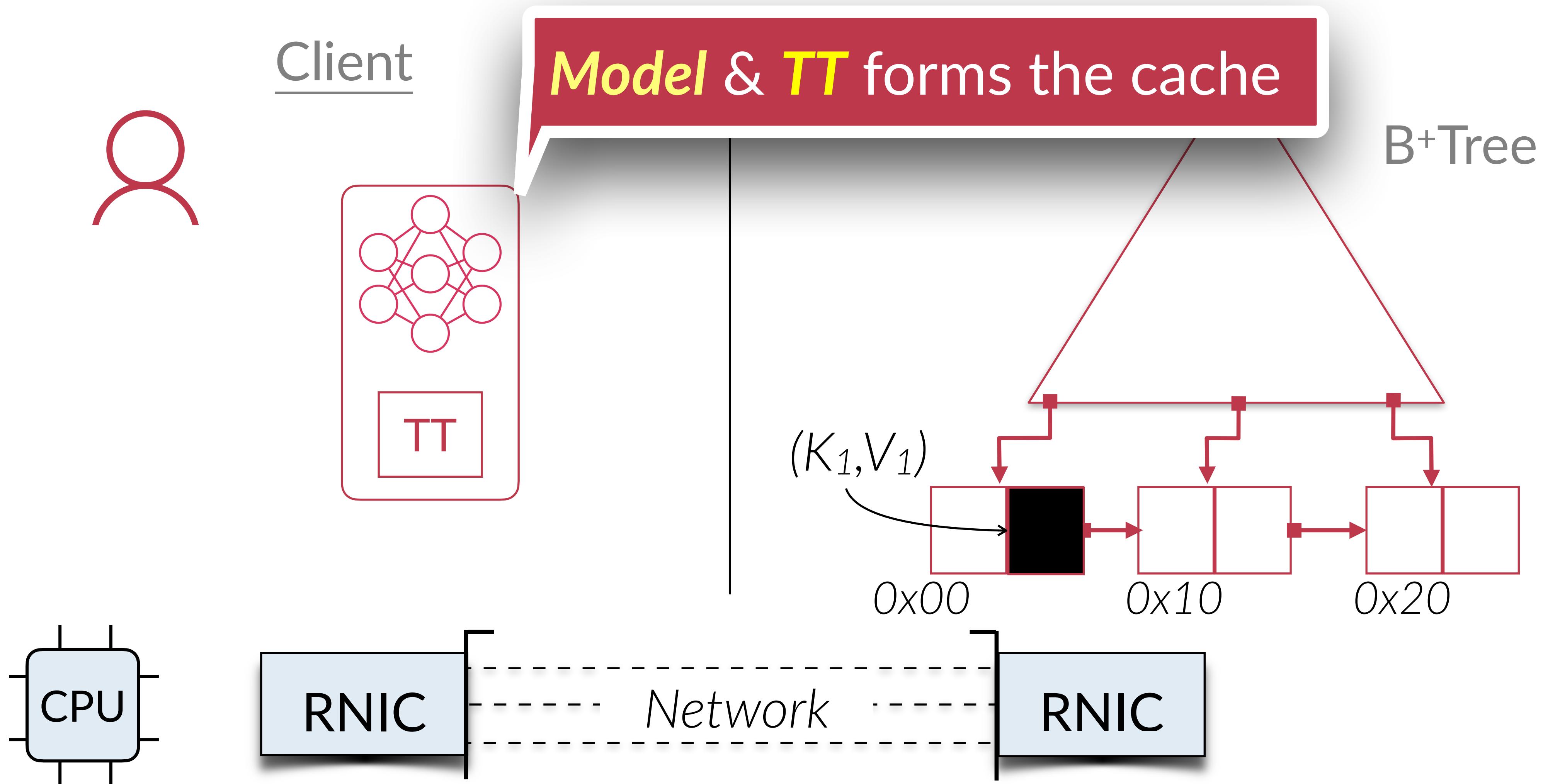
Outline of the remaining content

Server-side data structure for dynamic workloads

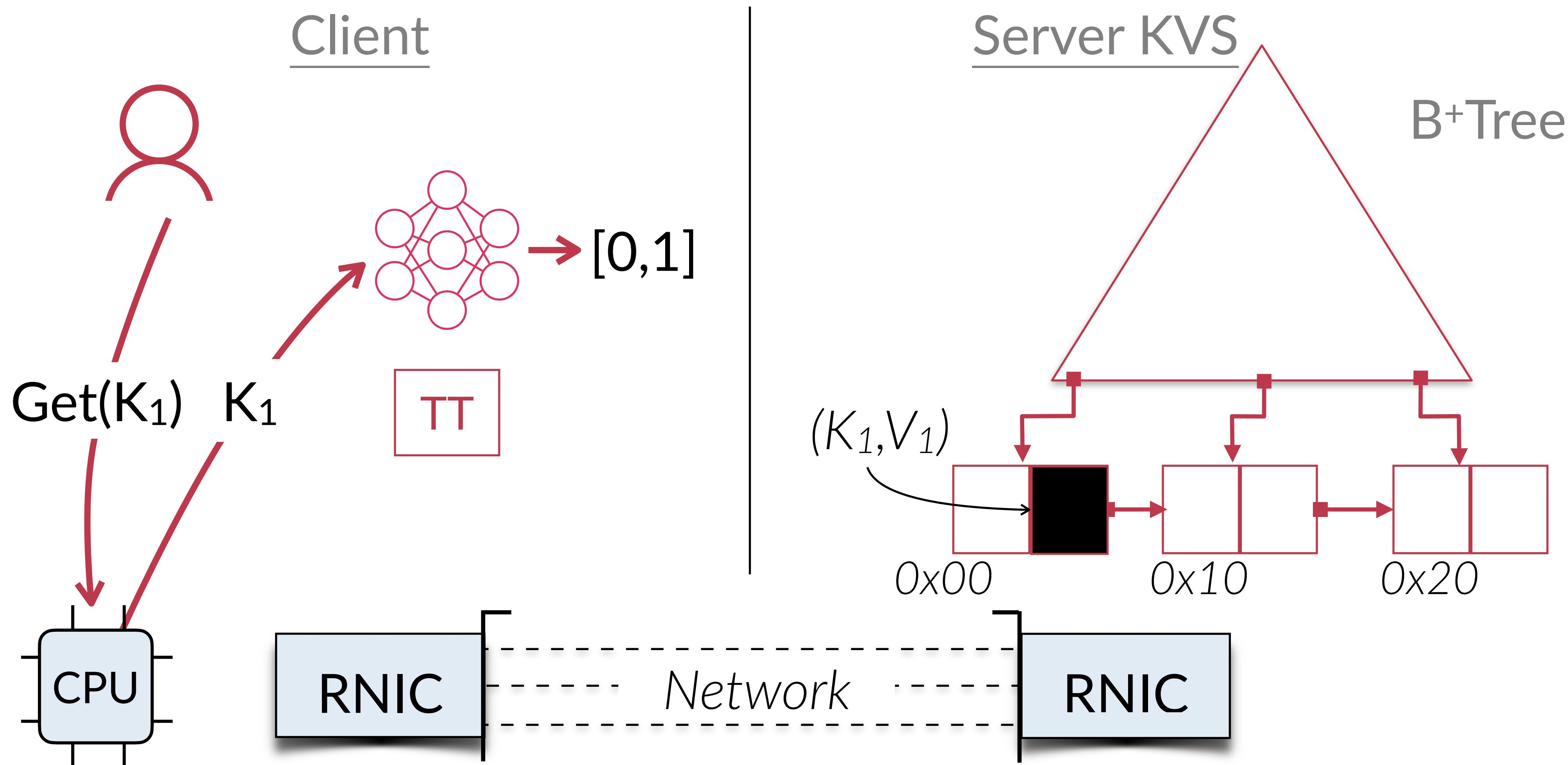
Client-side learned cache & TT

Performance evaluation of XSTORE

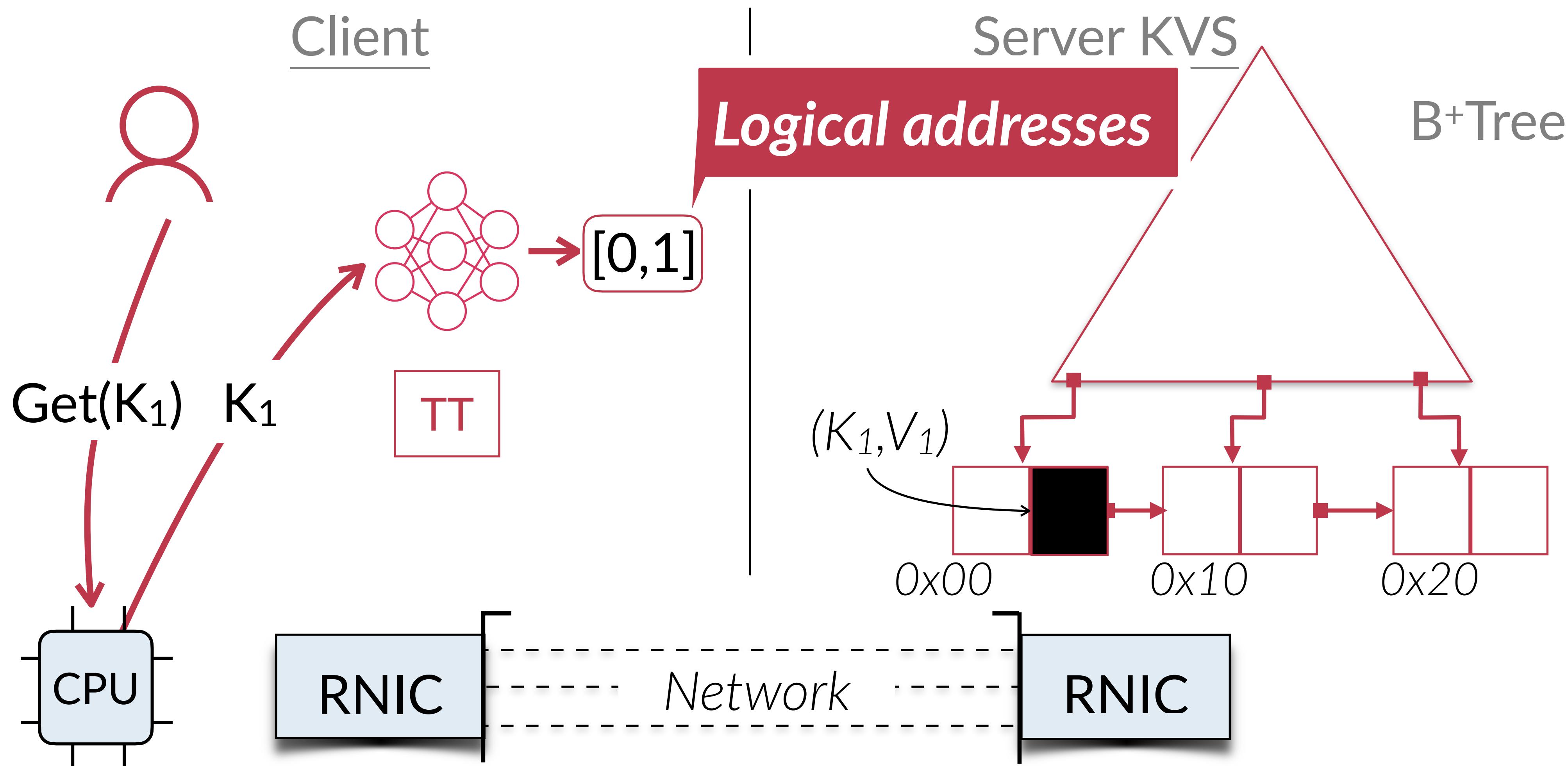
Client-direct Get() using model & TT



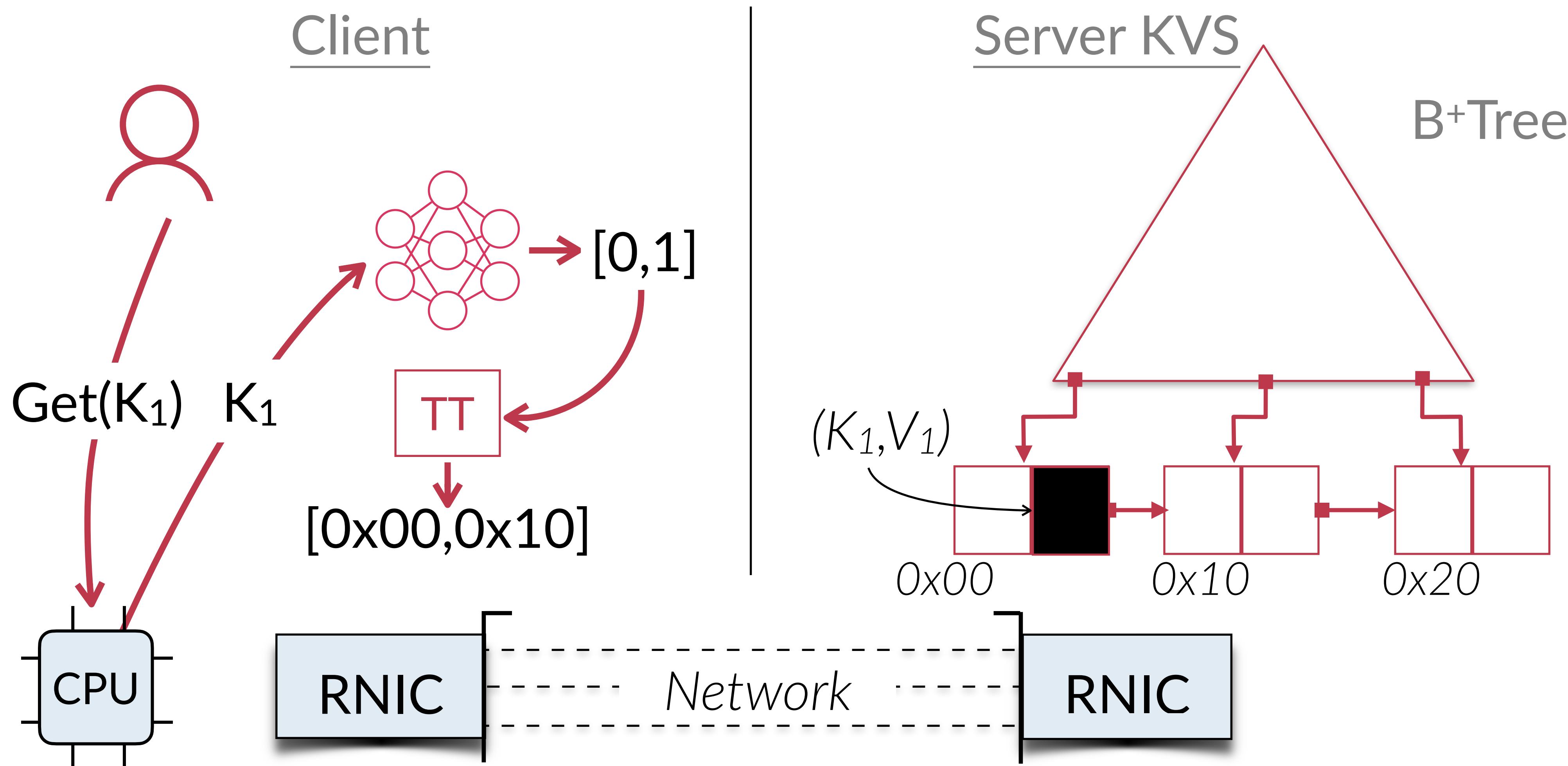
Client-direct Get() using model & TT



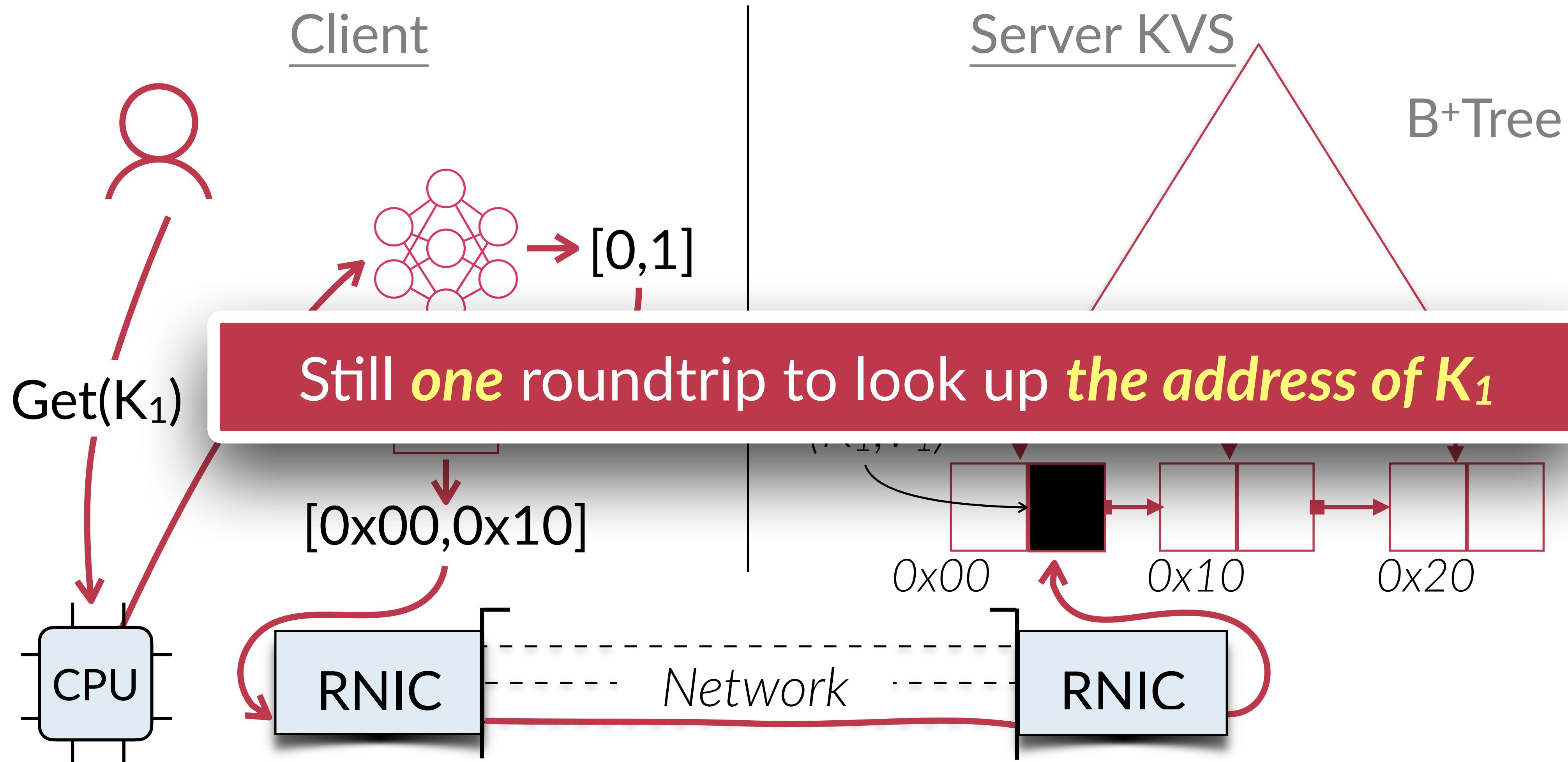
Client-direct Get() using model & TT



Client-direct Get() using model & TT



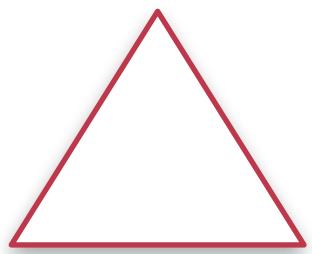
Client-direct Get() using model & TT



Model retraining

Model is retrained at server in background threads

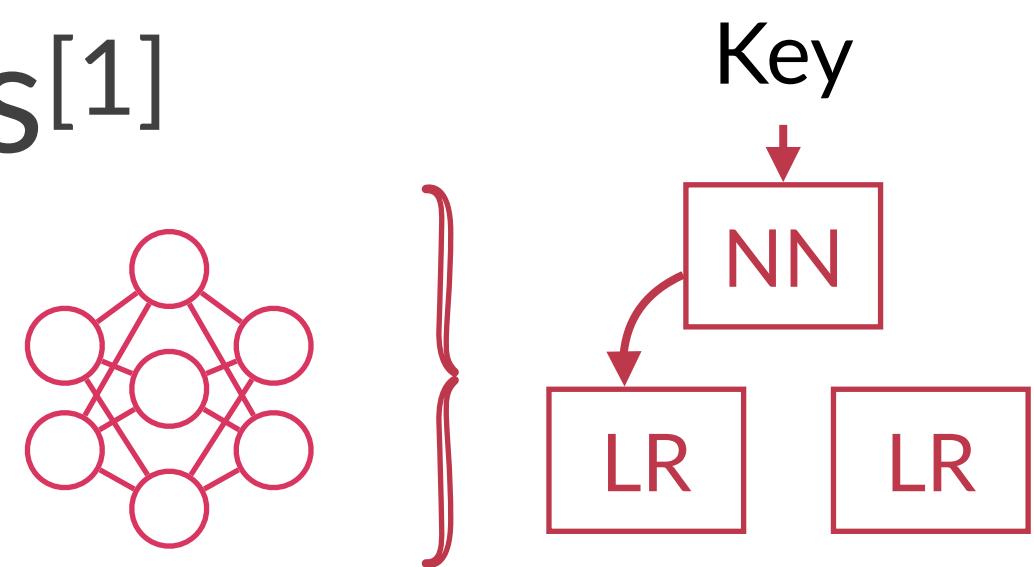
- ⌚ Small cost & extra CPU usage at the server



Server KVS

XSTORE uses a two-layer RMI to organize models^[1]

- ⌚ **Fine-grained** model retraining



Stale model handling

Background update causes *stale learned models*

But stale learned models & TT could correctly *find most keys*

- ⌚ If the key is *not moved*, a stale Model & TT still maintains correct

Key → Logical → Physical

Many other design details & optimizations

Server-side operations

Find *non-trained* keys

Optimizations of speculative execution

Dynamic model expansion

Fault tolerance of XSTORE

Scale-out XSTORE

by incarnation mismatch between the leaf node and cached TT entry (line 17 in Fig. 7) and results in a fallback, which ships the operation to the server. The client will update XCACHE with a retrained model and its translation table fetched by the fallback. Noted that concurrent splits will not affect model retraining in progress and just make it stale. The new incarnation of the split leaf node ensures the client with this new (stale) model to realize the change of concurrent splits. Each split will issue a retraining task. The training thread currently does not merge or optimize the pending tasks to the same sub-model since it happened.

Optimization: speculative execution. A split node must still be mapped to this node or it moves the second half of key-value pairs to its new sibling leaf node. Therefore, the pre- LN_{101} and LN_{327} in Fig. 9b. Based on this observation, speculative execution is enabled to handle the lookup key in the keys of a stale TT entry (i.e., incarnation check is will still find the lookup key in the keys will. If not found, the client will split leaf node. It means there is no pointer to fetch (the second half) keys from RDMA READ). It means there is no chance to avoid incurring a performance penalty. This optimization only consider one sibling before using cascading split happens rarely. This optimization for insert-dominant workloads (e.g., Y operations and retraining tasks might fall back to the same sub-model since it happens to the same sub-model since it happens to a retraining task. The training stage of concurrent tasks does not merge or optimize the pending

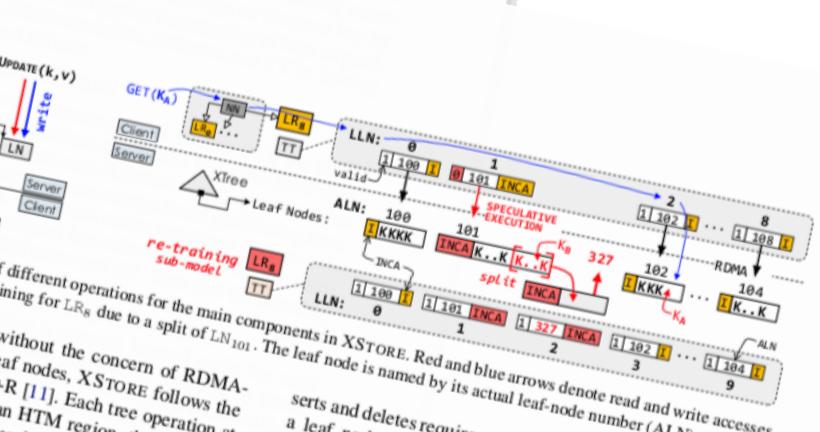
expansion. The growing size of key model will likely increase the predictive performance degradation. Prior work [4] model split to adapt its learned structures, which demands physical data node replacement. In response to the *ports model expansion* that increases in XMODEL at once (e.g., double exceeding a threshold of min- an expansion requires a complete train- build a new version of XMOD. Expansion will not affect any re- server and the client for seve- nals will not change or move be trained over incomplete model retraining could be can use the originally learned. Moreover, after deleting X-STORE can also resize X-models using a similar |

can also resize X -models using a similar writes function.

For $UPDATE(K, V)$, the server first traverses the tree to find the leaf node corresponding to key K . It then updates the value stored at that node to V . Note that this update will not change the learned model, as it only affects static workloads.

INSERT and DELETE
INSERT(K, V) and DELETE(K) are shipped to the server performed on XTREE, as is usual on B+-tree. The *in-place*

hey will not involve in logging and recovery. In addition, XMODEL and TT are tightly associated with XTREE (e.g., virtual address). Thus, they should be rebuilt after recovery. To ensure correct recovery from a machine failure, XTREE can reuse the existing durability mechanism in the current tree-based index extended by XTREE, like version numbers [50, 33]. Each worker thread at the server appends (key, value, and version) to its in-memory log buffer. A corresponding logging thread, sharing the same core with a worker thread, writes out the log buffer to its log file in



9
-rows denote read and write accesses.
rts and deletes require moving many key-value pairs within
leaf node to preserve the order of keys. The
oses not to keep key-value pairs in
ch can avoid this.

key-value pairs sorted within a leaf node, can avoid moving key-value pairs and reduces work in the HTM region. Note that the lookup based named cache will not be affected since it fetches all keys from a leaf node. For $\text{DELETE}(K)$, we always overwrite the value slot for K with the last key-value pair in the leaf node. For $\text{DELETE}(K)$, we directly append K and V to the end of the leaf node if K does not exist in the leaf node. For $\text{INSERT}(K, V)$, we directly append K and V to the end of the leaf node if K does not exist in Fig. 9b). Inserting a key-value pair into a full leaf node will not be reclaimed to avoid thrashing and model a node split (see K_B in Fig. 9b). A new result in a node split (see K_B in Fig. 9b). A new node is allocated, and all key-value pairs (plus the new key-value pairs) are moved to the new node. The original leaf node should increment its index. The clients realize the split and execute on the two nodes.

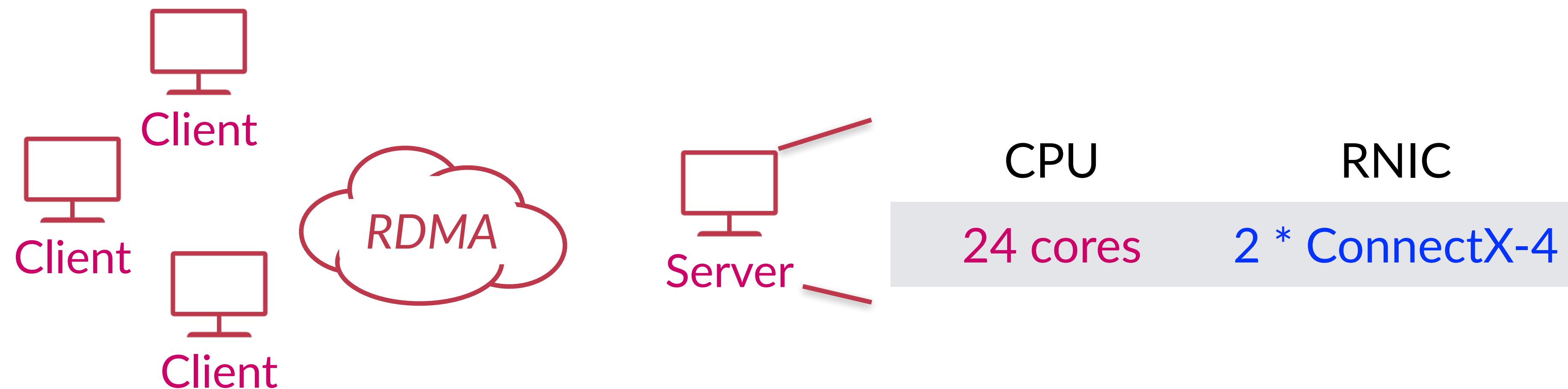
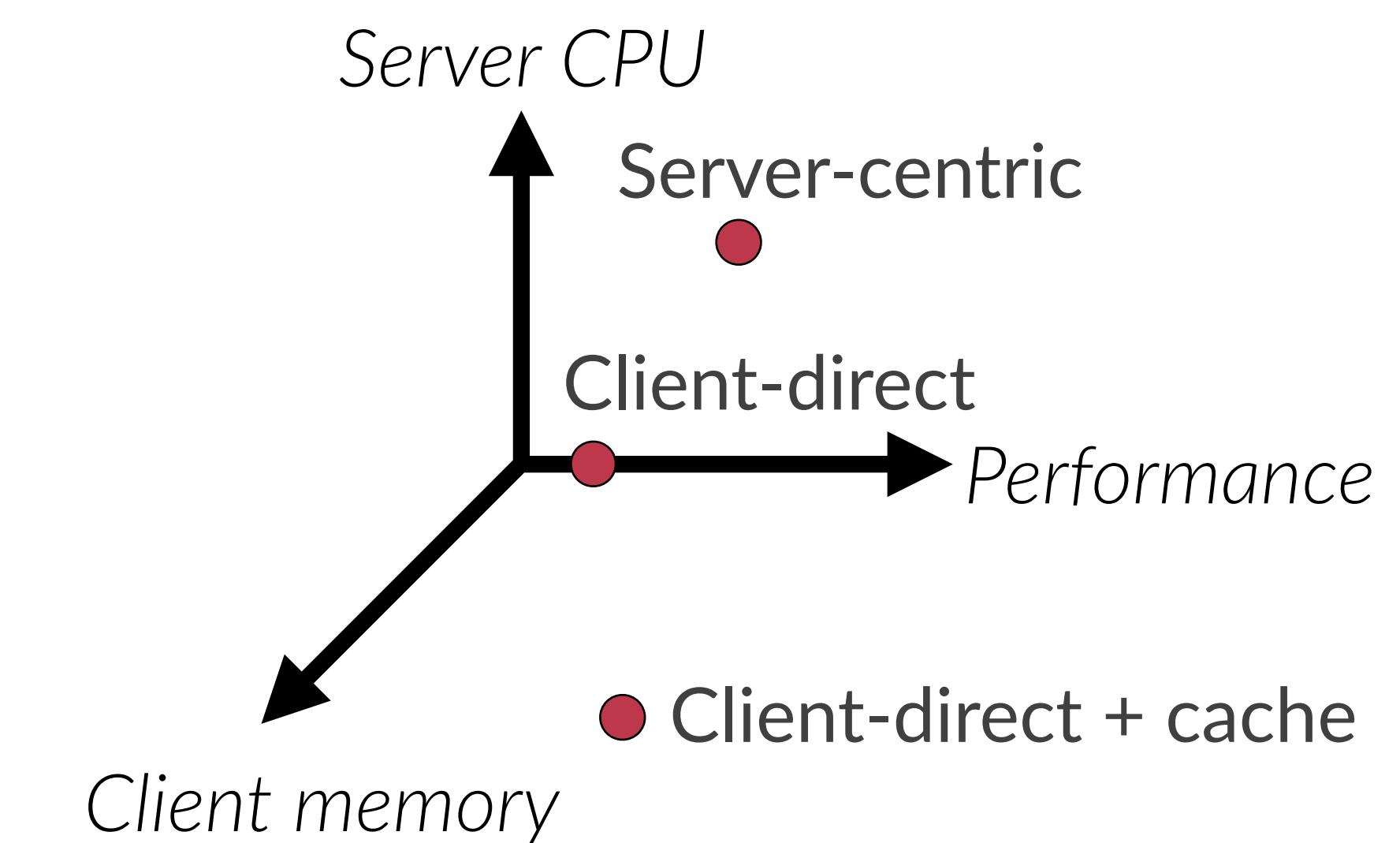
invalidation. The insert of a new leaf node breaks the sorted (logical) order of leaf nodes. An interesting observation before retraining. An interesting observation is that TT decouples model retraining from and allows a *stale* combination of XMODEL to make a *correct* prediction for the lookup key that overlapped with a split. This is because cause data movement across leaf nodes, for example, LR_8 initially maps K_A to leaf node LN_2 , which stores the leaf's physical address 327, the latest logical node after retraining. Yet, the stale TT still address 102, the correct position of K_A . I will use a combination of stale models as long as they are not overlapped split, the

Based on this, after a split, the server will *individually* retrain the sub-model and its translation table in the background (see TRAIN_SUBMODEL in Fig. 6) and perform all kinds of operations as usual based on XTREE. Meanwhile, the clients can still directly perform read-only operations based on XCACHE. The incorrect prediction can be detected

Evaluation of XSTORE

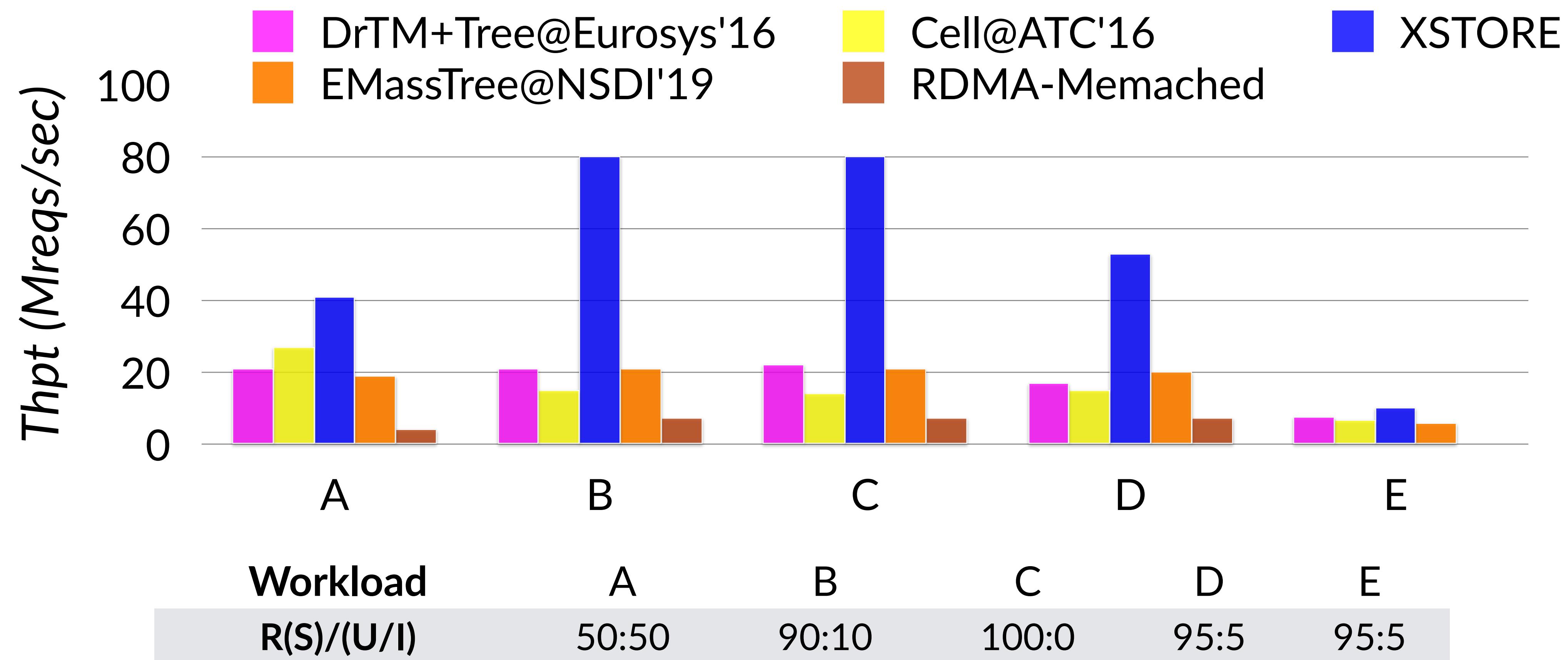
We answer the following questions:

- ⌚ Comparing to server-centric designs?
- ⌚ Comparing to client-direct designs?
- ⌚ Does XStore provide better trade-off?



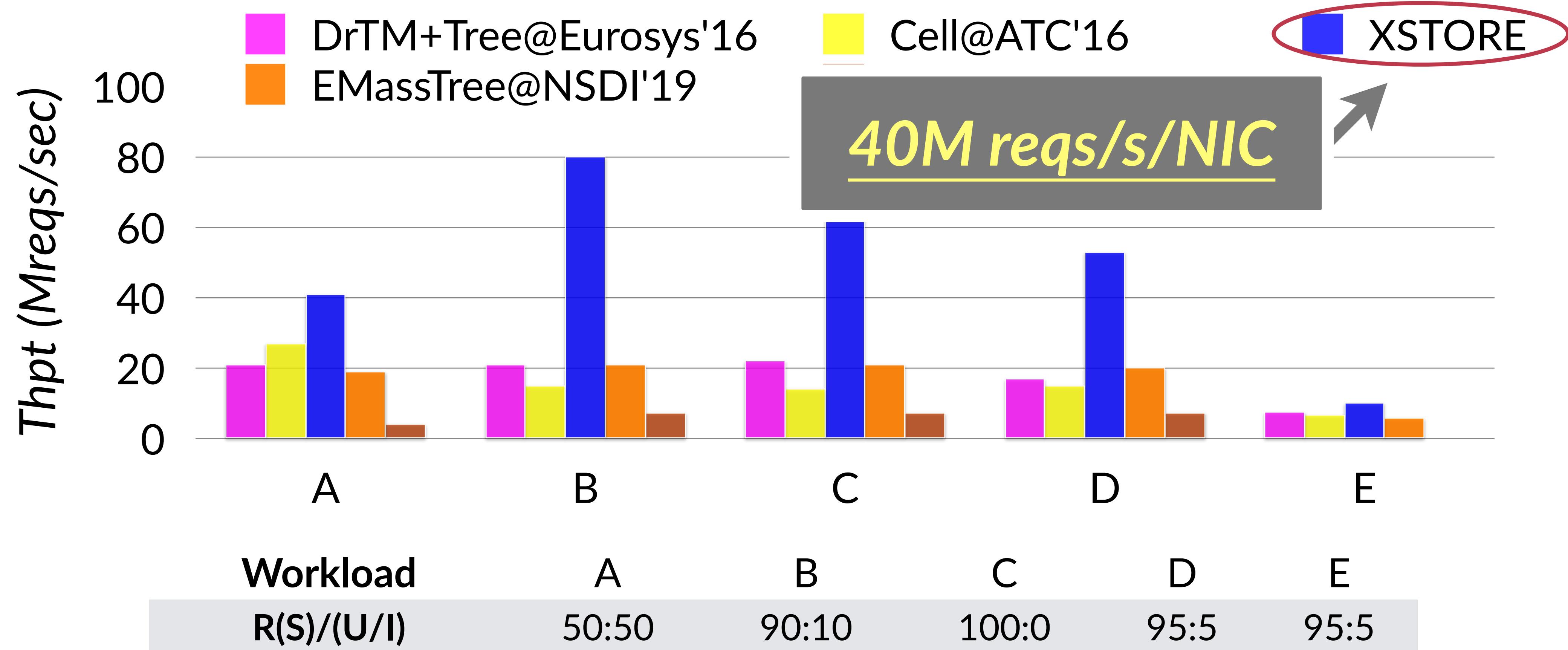
Performance of XSTORE on YCSB

100M KVs, uniform workloads



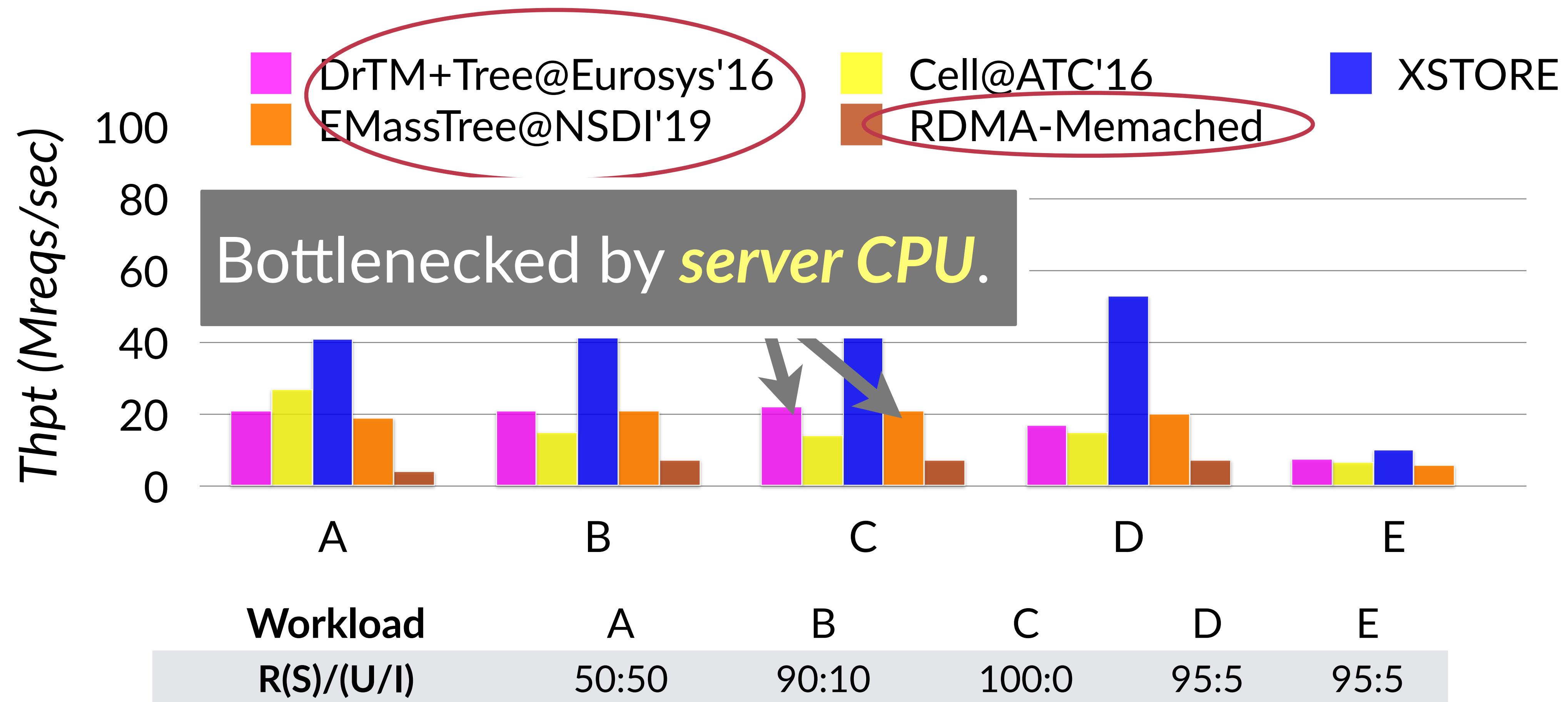
Performance of XSTORE on YCSB

100M KVs, uniform workloads



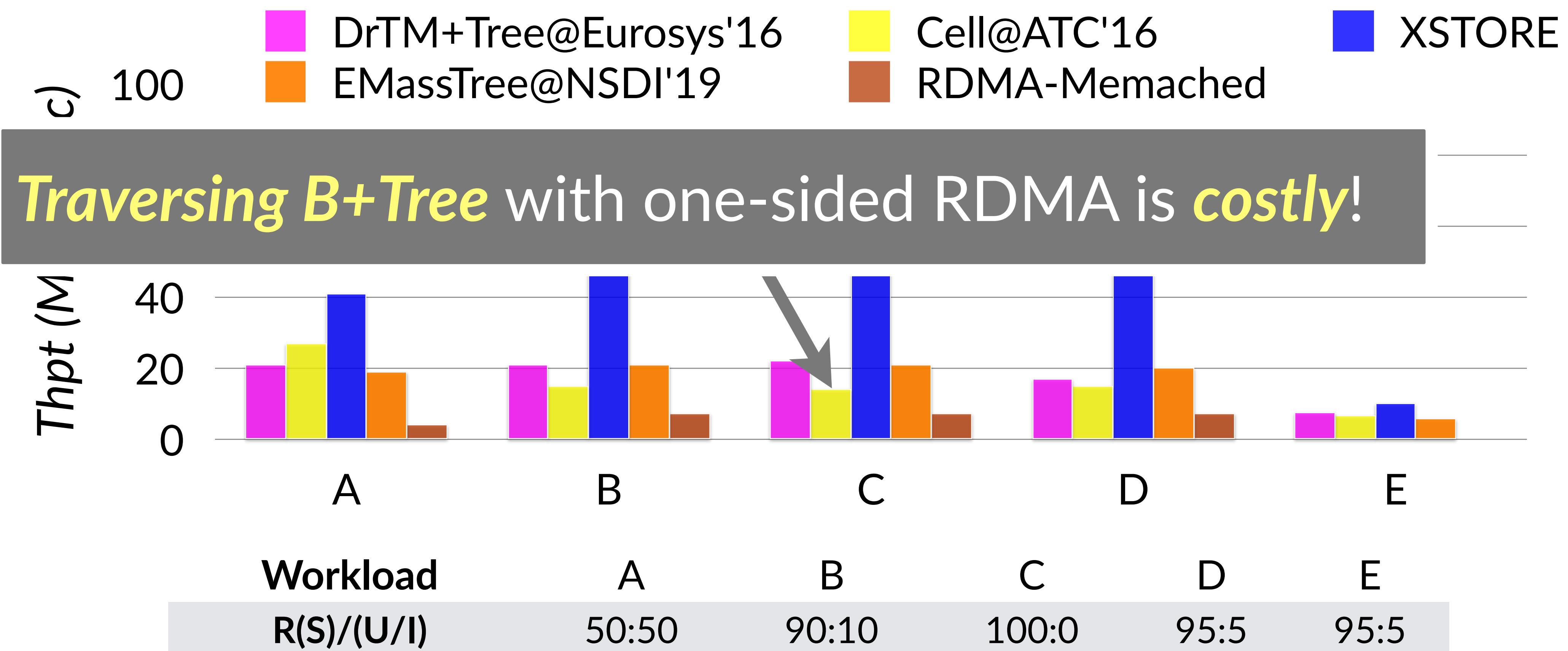
Performance of XSTORE on YCSB

100M KVs, uniform workloads



Performance of XSTORE on YCSB

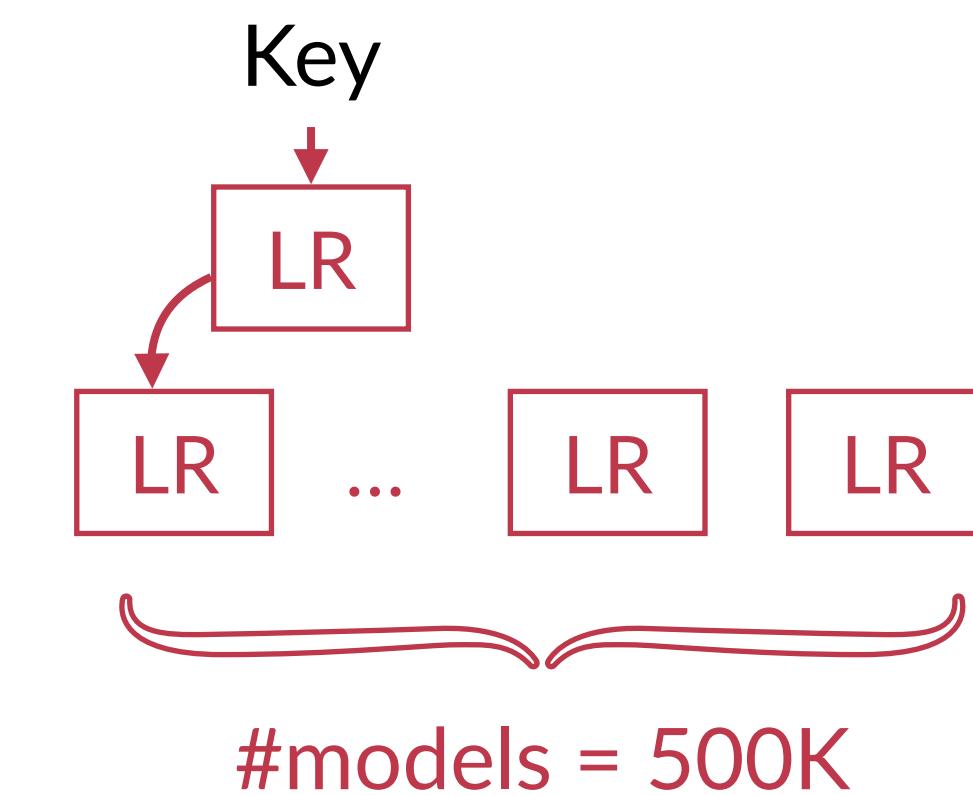
100M KVs, uniform workloads



The XCache in details

For a 100M KVs YCSB dataset

- ⌚ 500K Linear regression as models, each 14B
- ⌚ ~ 8 μ s to retrain each model
- ⌚ ~ 8s to train the entire cache

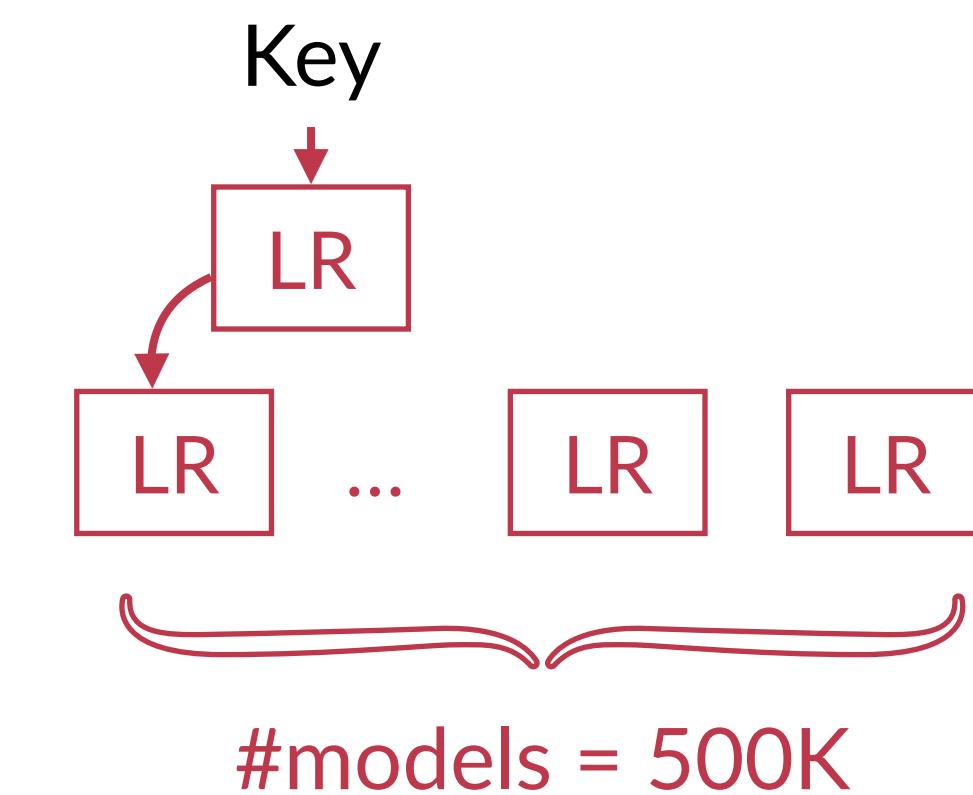


The XCache in details

For a 100M KVs YCSB dataset

- ⌚ **500K** Linear regression as models, each 14B
- ⌚ ~ 8 μ s to retrain each model
- ⌚ ~ 8s to train the entire cache

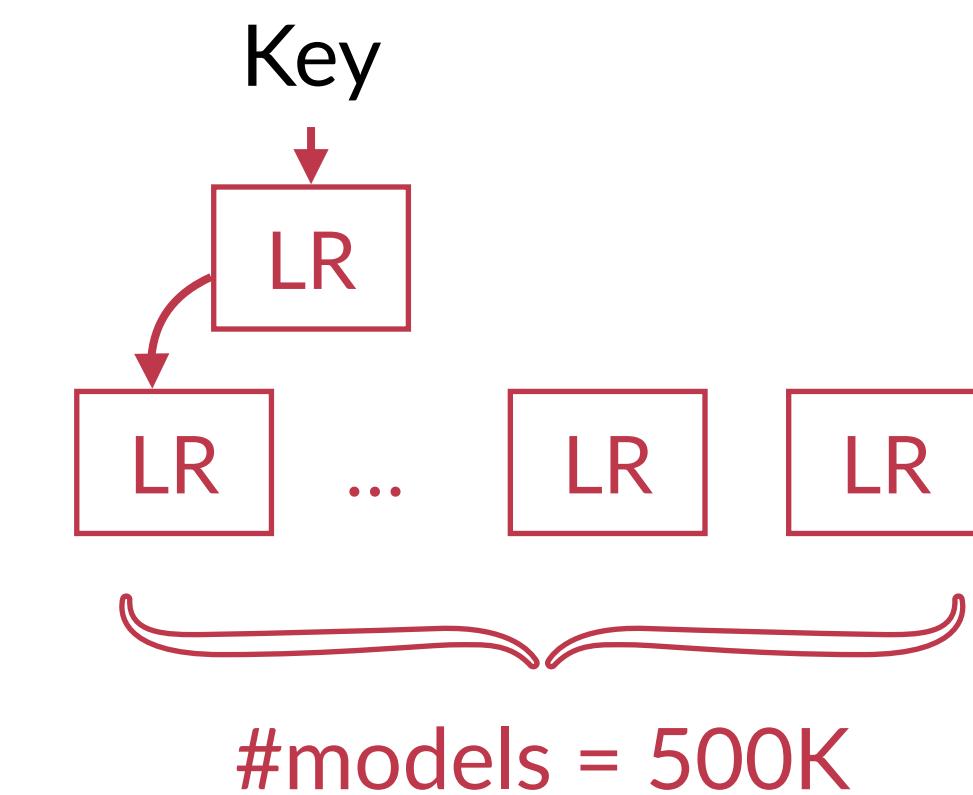
Small model to fit the dataset



The XCache in details

For a 100M KVs YCSB dataset

- ⌚ 500K Linear regression as models, each 14B
- ⌚ ~ 8 μ s to retrain each model
- ⌚ ~ 8s to train the entire cache



Quick retrain under dynamic workload

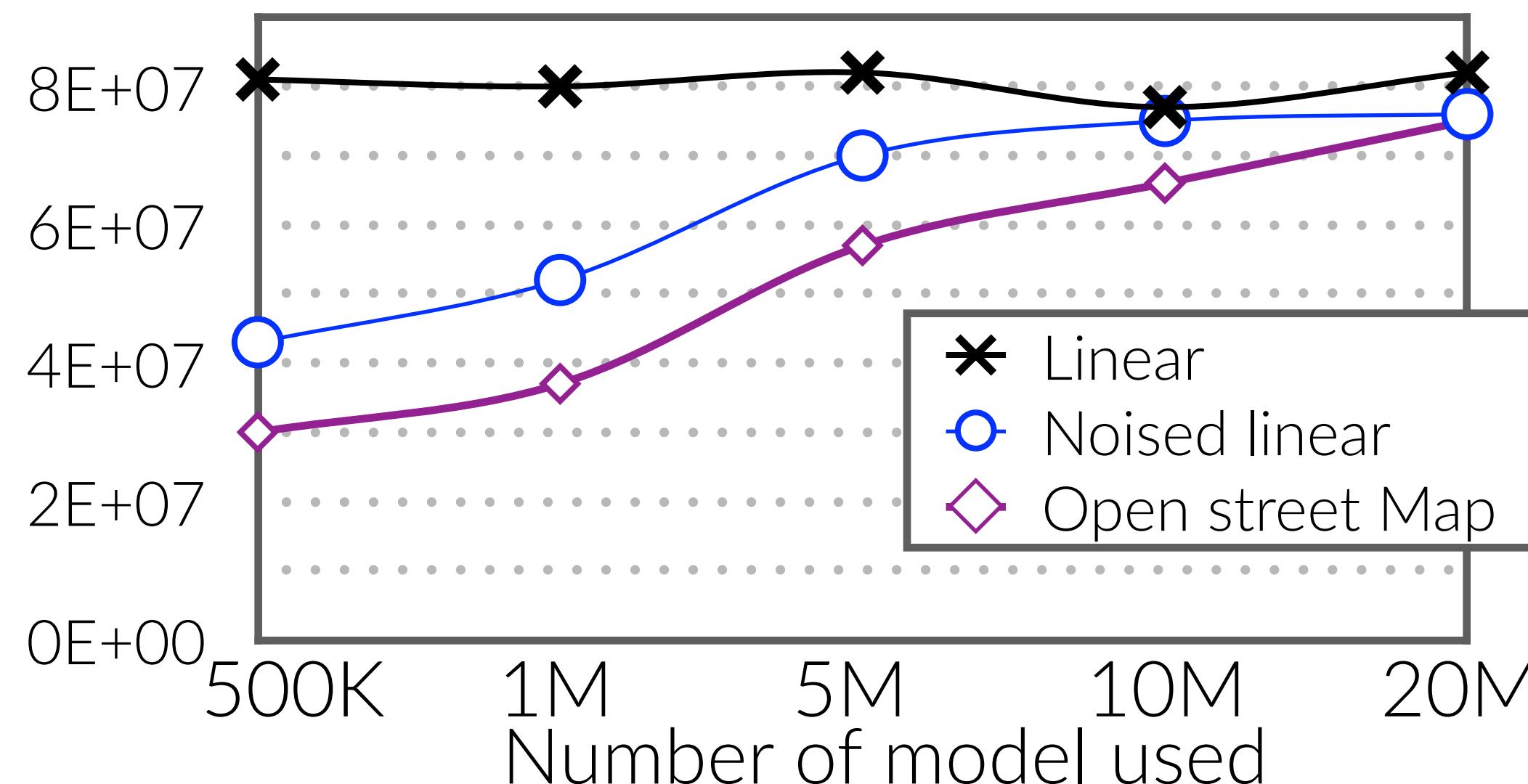
Sensitive to the dataset

Different dataset has different accuracy

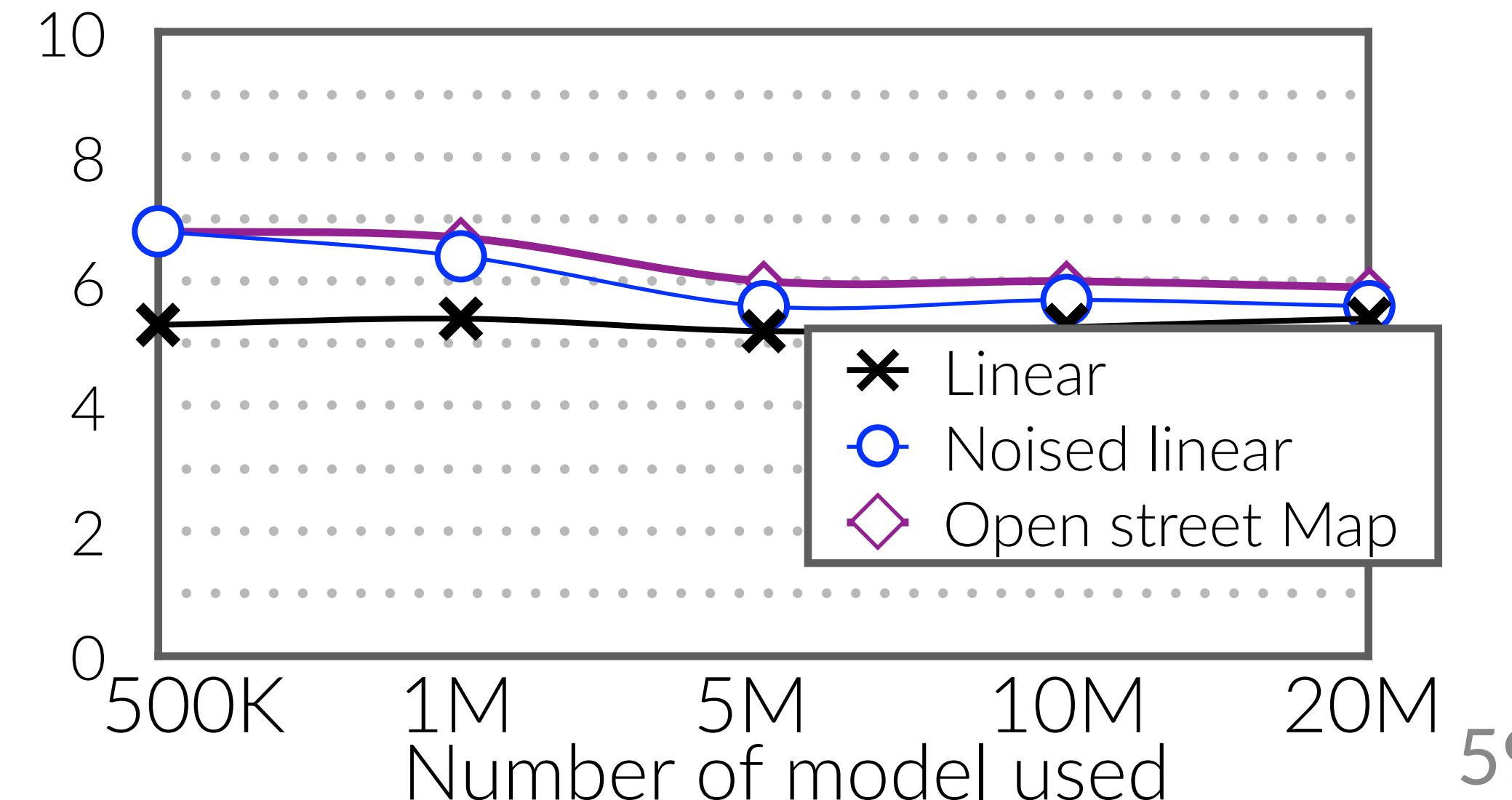
⌚ May affect the performance

Throughput drop due to increased error for complex dataset

Peak throughput (100M dataset)



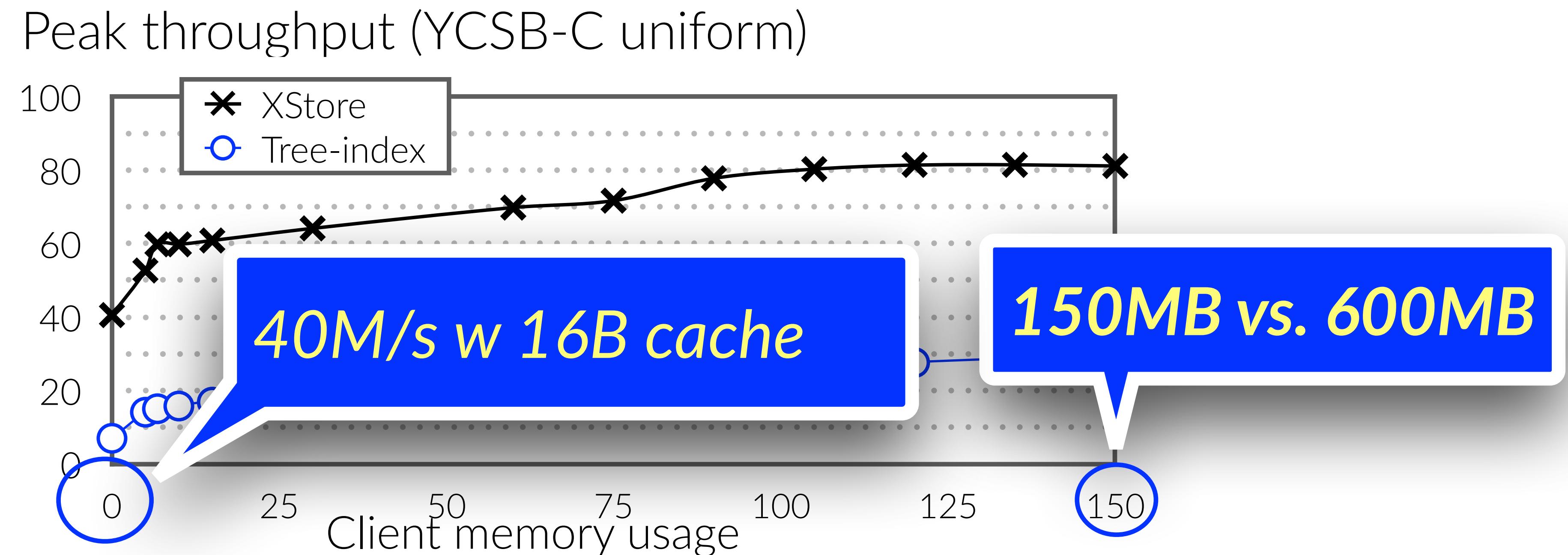
Average latency (μ s)



Learned cache vs. Tree-based cache

XStore provides better *memory-performance trade-off*

⌚ YCSB-C uniform workload



Current limitations and future work

XSTORE currently only supports fixed-length keys

⌚ Our paper describes our plan to support variable-length keys

Focus on simple models (e.g., LR)

⌚ Efficient upon retraining under dynamic workloads
⌚ May result in huge error for complex data distribution
⌚ Trade-off: retraining speed vs. accuracy vs. memory

Orthogonal to the design of XSTORE

Conclusion



IPADS
INSTITUTE OF PARALLEL
AND DISTRIBUTED SYSTEMS

XSTORE provides *a new design* for RDMA-enabled KVS

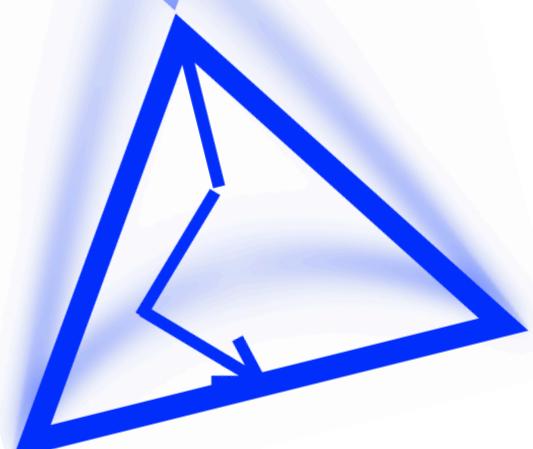
• First adopts the learned models for one-sided RDMA READ

XSTORE provides better trade-offs:

• *Server-side CPU vs. Client-side memory vs. Performance*

Please check XSTORE@

• <https://ipads.se.sjtu.edu.cn/projects/xstore>



Thanks & QA