高性能高可靠键值存储系统

High-Performance and Reliable Key-Value Stores

李永坤 副教授

http://staff.ustc.edu.cn/~ykli/

中国科学技术大学

University of Science and Technology of China

Joint work with Qiang Zhang, Patrick P. C. Lee, Yinlong Xu, Si Wu

Background

- Data is growing exponentially and diversified
  - Total amount of data in the wild will reach 175ZB by 2025
  - Unstructured data is dominant (> 75% of all data)
- Key-value (KV) stores are widely used
  - Flexible data model & high scalability
  - Simple interface: Put, Get, Scan,…

Google

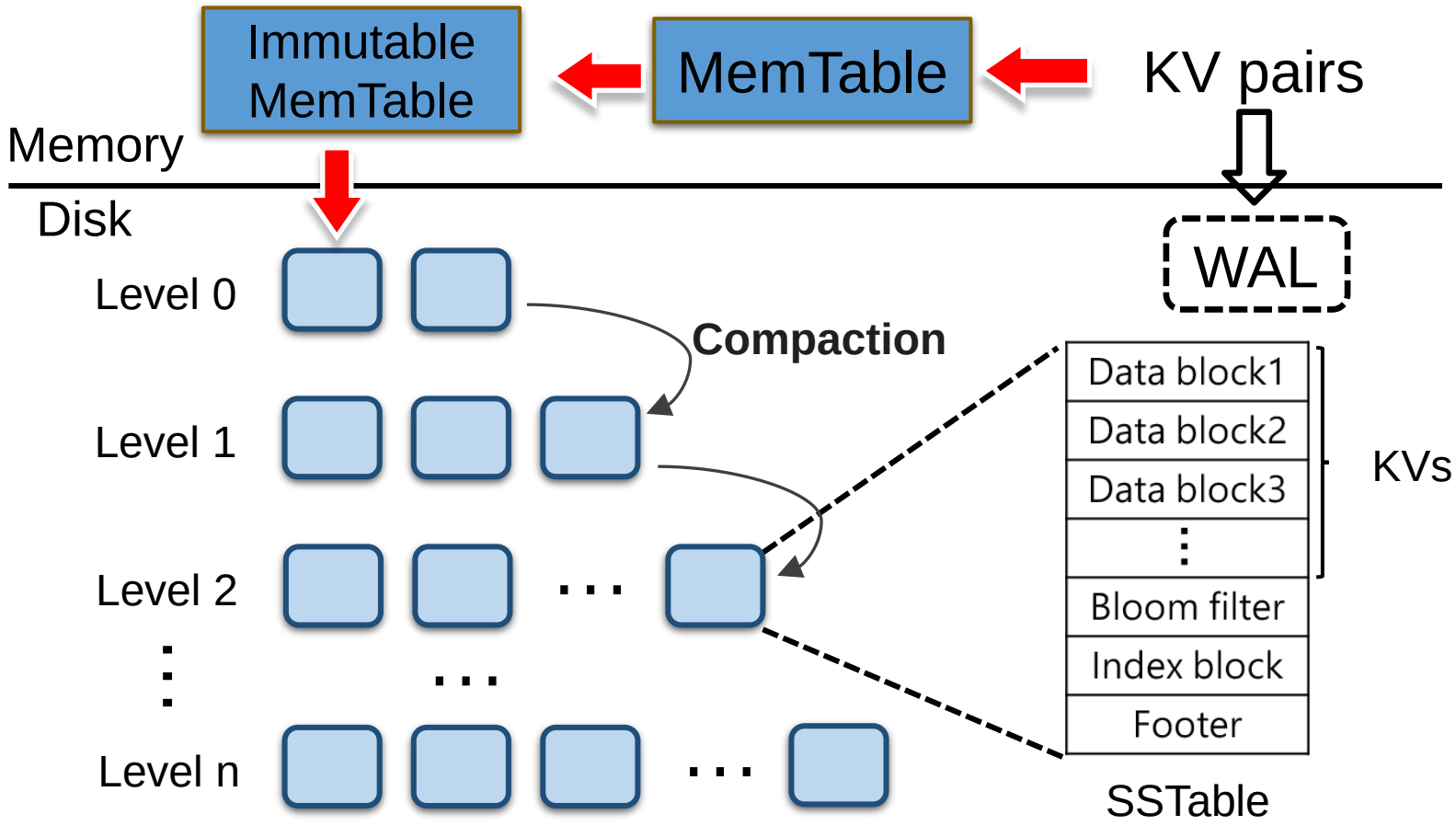
Facebook

Apache

PingCAP

Background

➢ Most KV stores build on **log-structured merge tree (LSM-tree)**

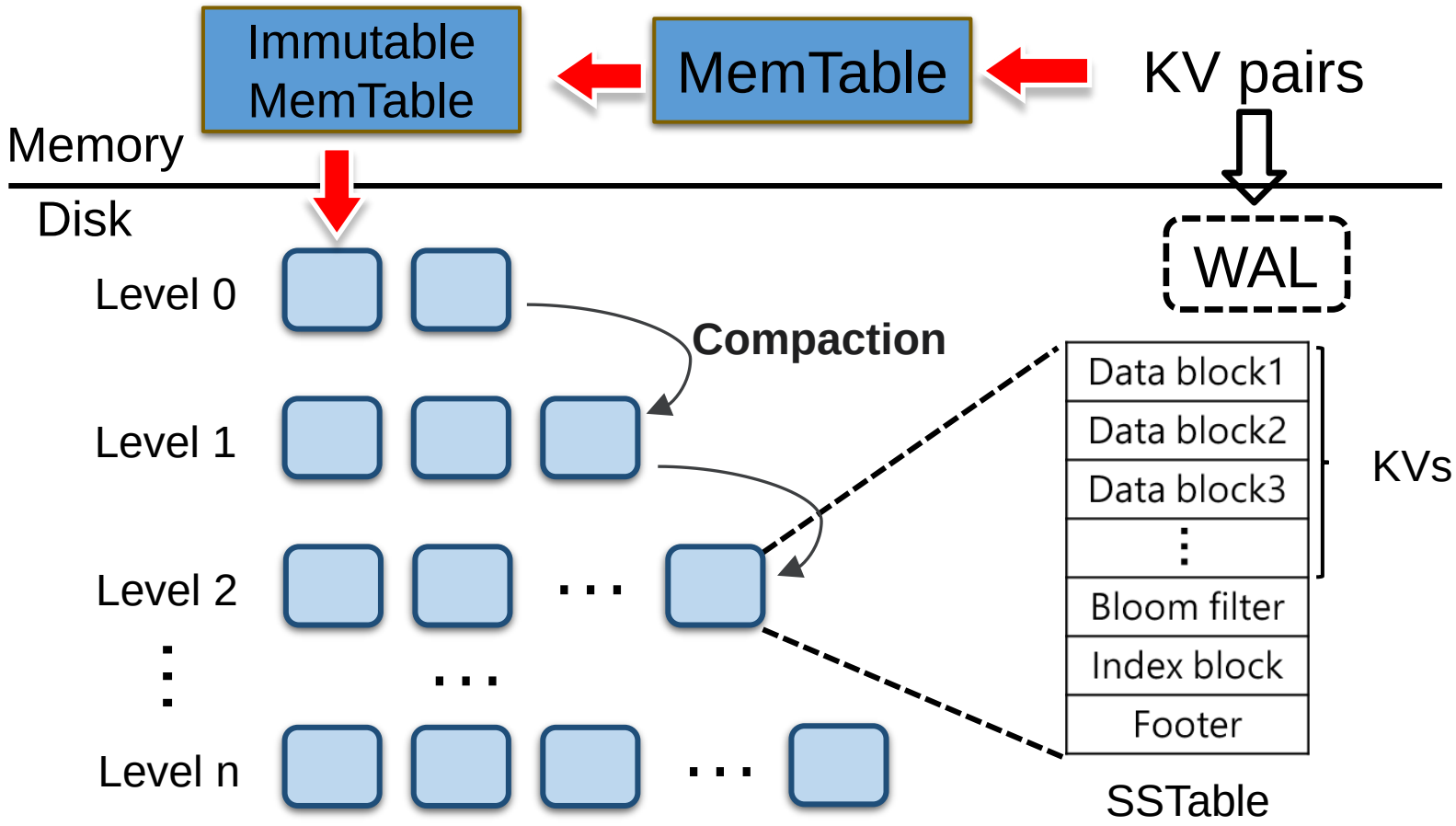


Multiple levels
Log-structured writes
Sorted in each level

Advantages:
Efficient writes, scans
high scalability

Background

➢ Limitations of **LSM-tree** based KV stores



High Read Amplification

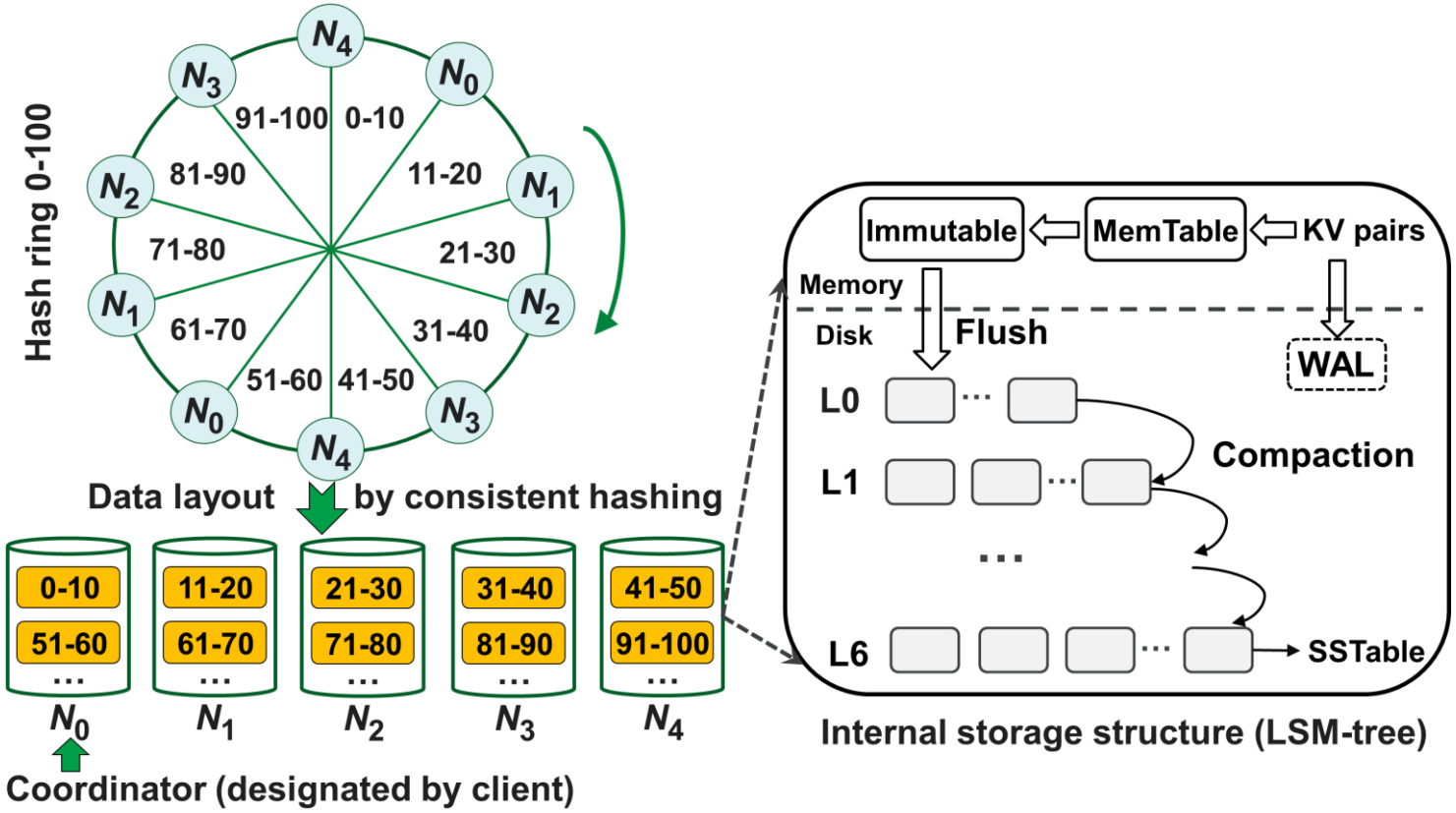
High Write Amplification

UniKV@ICDE 2020

DiffKV@ATC 2021

Background

- Distributed KV stores
  - KV pairs are partitioned based on **consistent hashing**
  - Each node stores KV pairs in LSM-tree



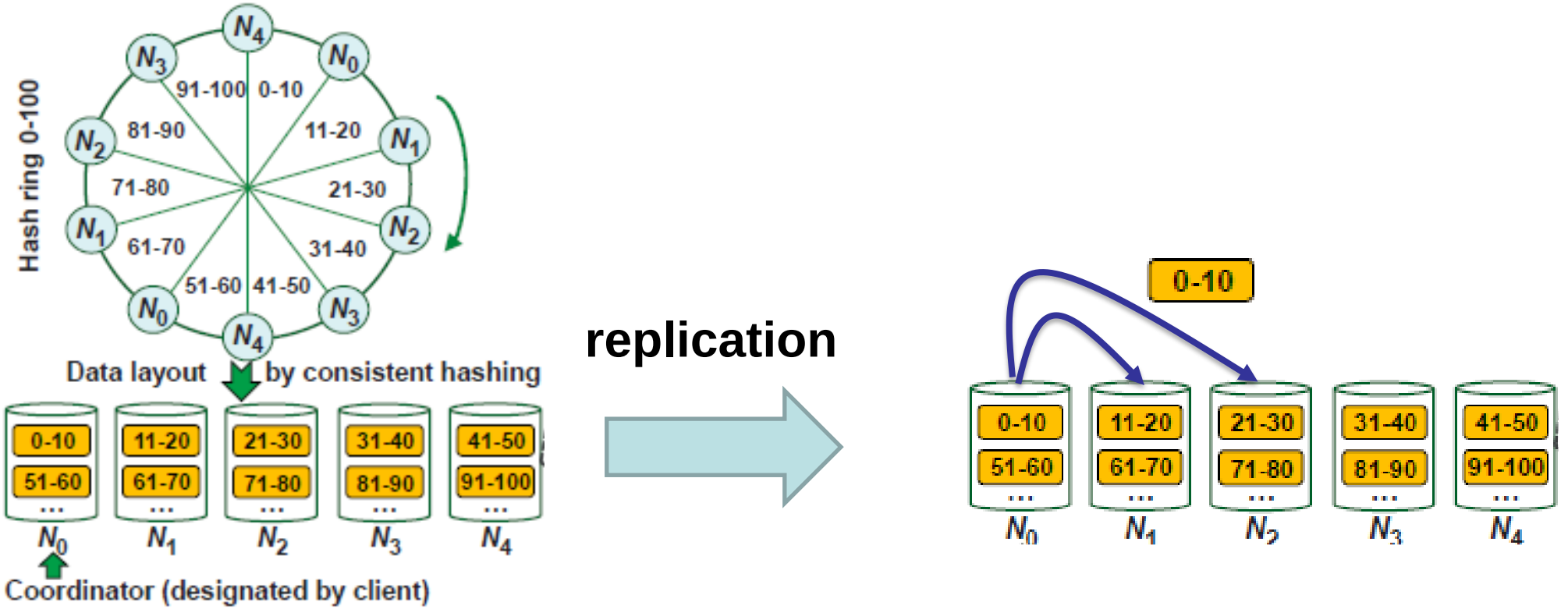
DynamoDB, Cassandra,

ScyllaDB, HyperDex, TiKV, …

Take Cassandra as an example

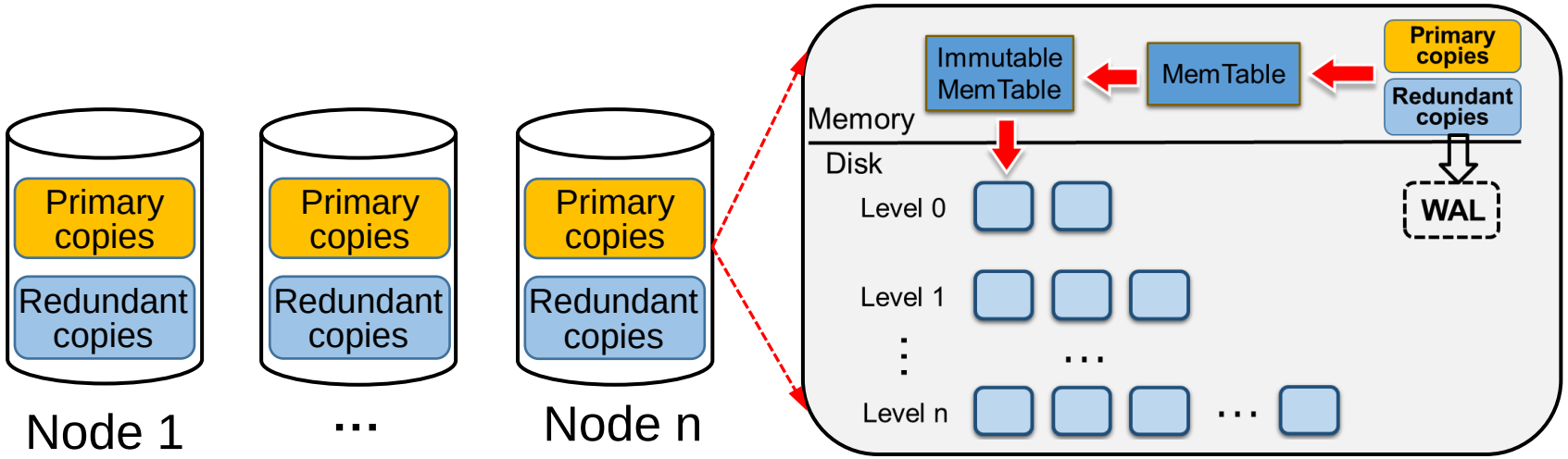
Motivation

- Replication makes replicas for each KV pair for fault tolerance
  - **Primary copy**: the main replica, mapped to nodes by consistent hashing
  - **Redundant copies**: remaining replicas, mapped by replication strategy



Motivation

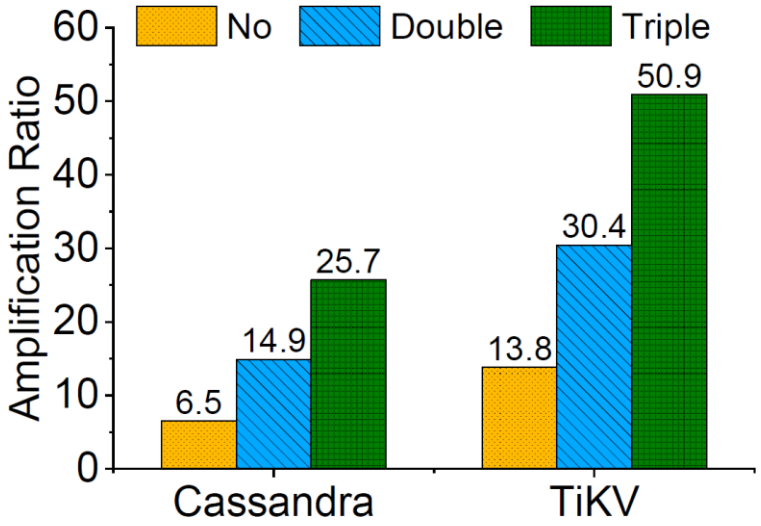
- Replication makes replicas for each KV pair for fault tolerance
  - **Primary copy**: the main replica, mapped to nodes by consistent hashing
  - **Redundant copies**: remaining replicas, mapped by replication strategy
  - **Uniform indexing**: each node stores primary and redundant copies together in **a single LSM-tree** → aggravate read & write amplifications



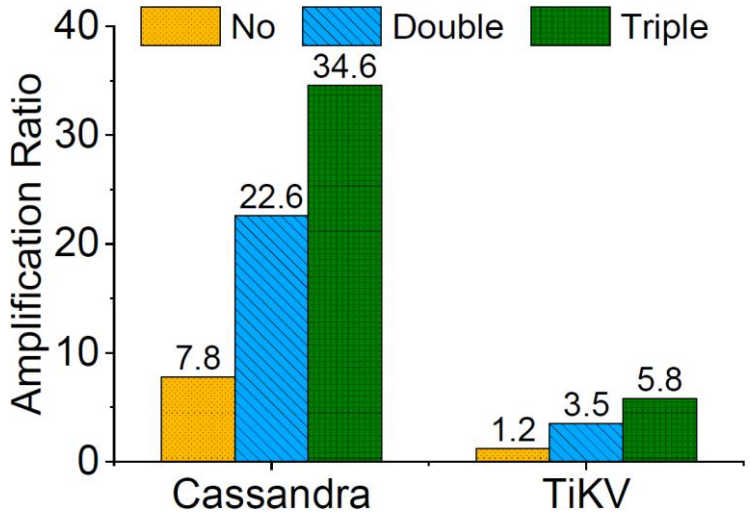
e.g., Cassandra, ScyllaDB, TiKV, HBase, HyperDex

Motivation

- Experimental verification on Cassandra & TiKV



(a) **Write amplification** under different number of replicas



(b) **Read amplification** under different number of replicas

"No": one replica
"Double": two replicas
"Triple": three replicas

A 5-node cluster
Write 300GiB KV pairs
Read 30GiB KV pairs

A larger replication factor implies higher write/read amplifications

Our Idea

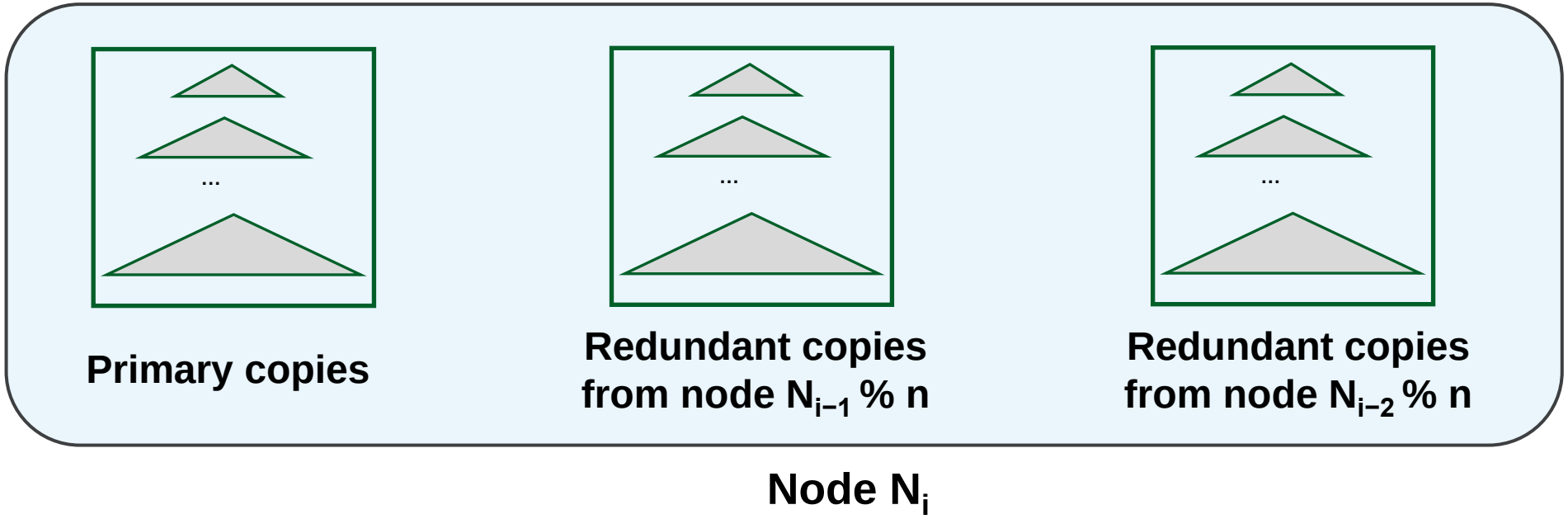
Decoupling **primary** and **redundant copies** in storage layer

- **Avoid interaction** between primary and redundant copies during reads and writes

How to manage primary and redundant copies separately after replica decoupling?

Naïve Approaches

- Replica decoupling with **multiple LSM-trees (mLSM)**
  - Each node manages k replicas with **k LSM-trees**
  - If k=3, a node uses **one LSM-tree** to store **primary copies**, and **two LSM-trees** to store **redundant copies** from two other nodes



***n**: the number of physical nodes*

Naïve Approaches

➢ Limitations of mLSM

- **L1: k× memory overhead**: each LSM-tree has its MemTable

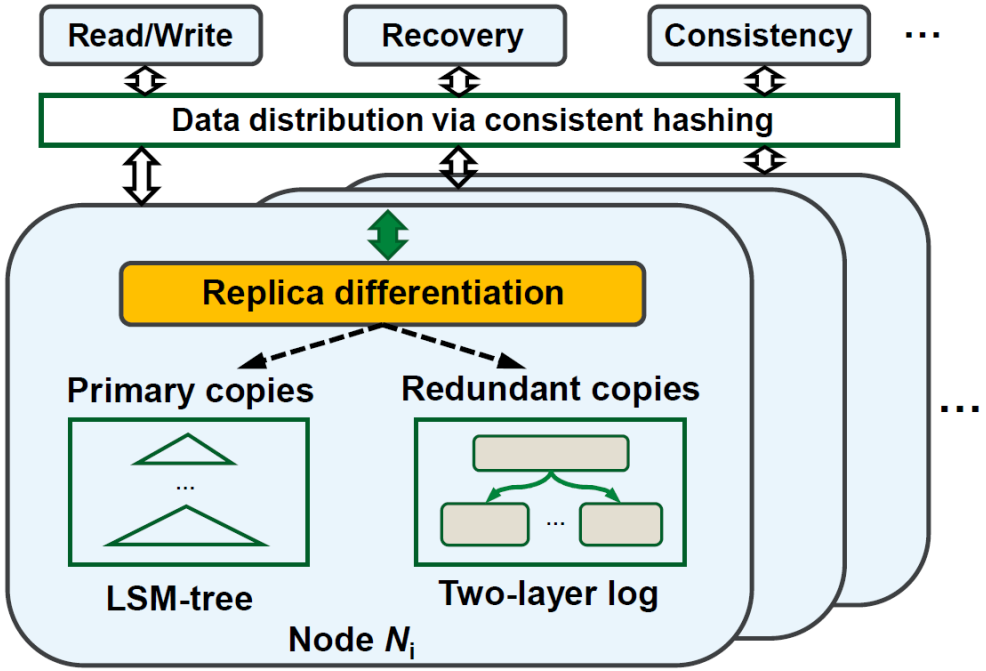
> If MemTable is **m MiB** and the cluster size is **n**, memory cost of **k LSM-trees** is **k×m×n MiB**

- **L2:** limited reduction of **compaction overhead**: each LSM-tree executes frequent compactions to keep each level fully-sorted

> Client writes 200GiB data under triple replication, total compaction sizes of **Cassandra** and **mLSM** are **3.46TiB** and **2.72TiB**, mLSM only reduces compaction size by **21%**

Our Design

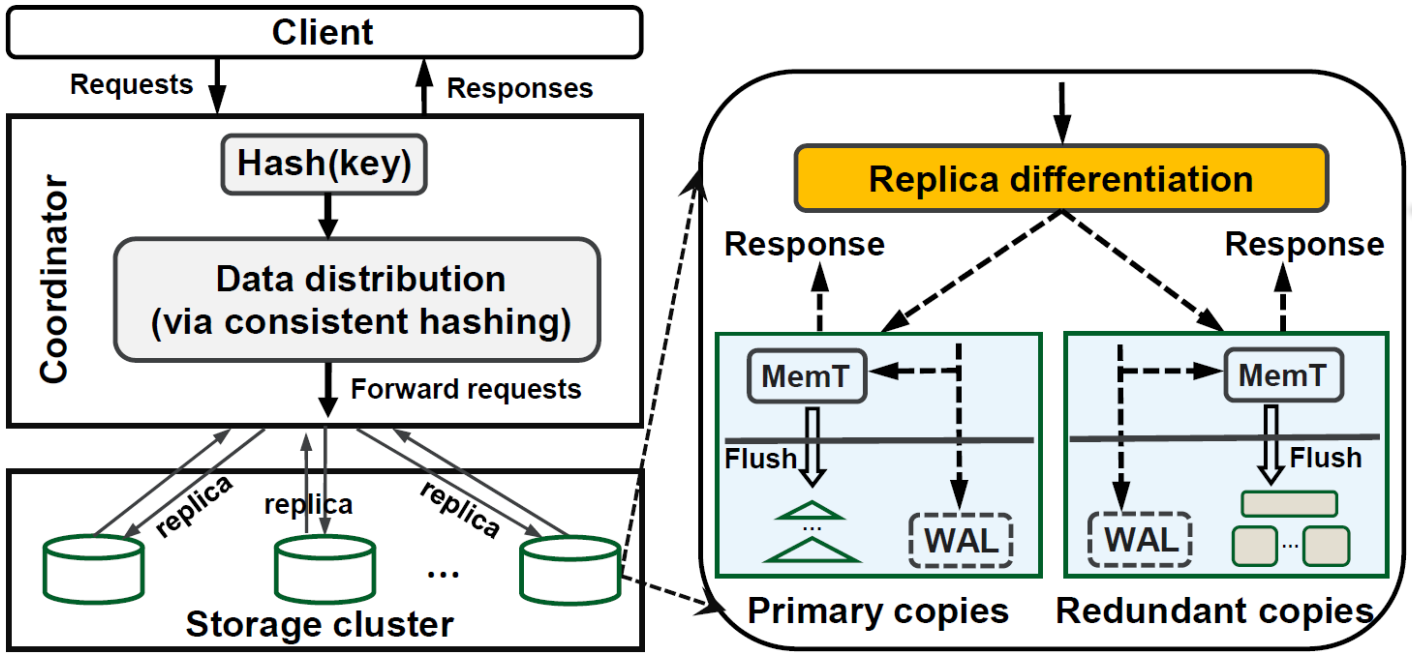
- **DEPART**: Replica decoupling for distributed Key-Value storage
  - **Primary copies**: LSM-tree → efficient writes, reads and scans
  - **Redundant copies**: two-layer log → fast writes, tunable trade-off
- Key design points
  - Replica differentiation
  - Two-layer log with tunable ordering
  - Parallel recovery scheme
- Implementation atop Cassandra



Replica Differentiation

➢ How to differentiate primary and redundant copies?

Lightweight replica differentiation



Based on simple hash computation → **Low overhead**

A node receives a KV pair, computes hash(key) → Get **node ID** to which the KV is mapped by **consistent hashing**

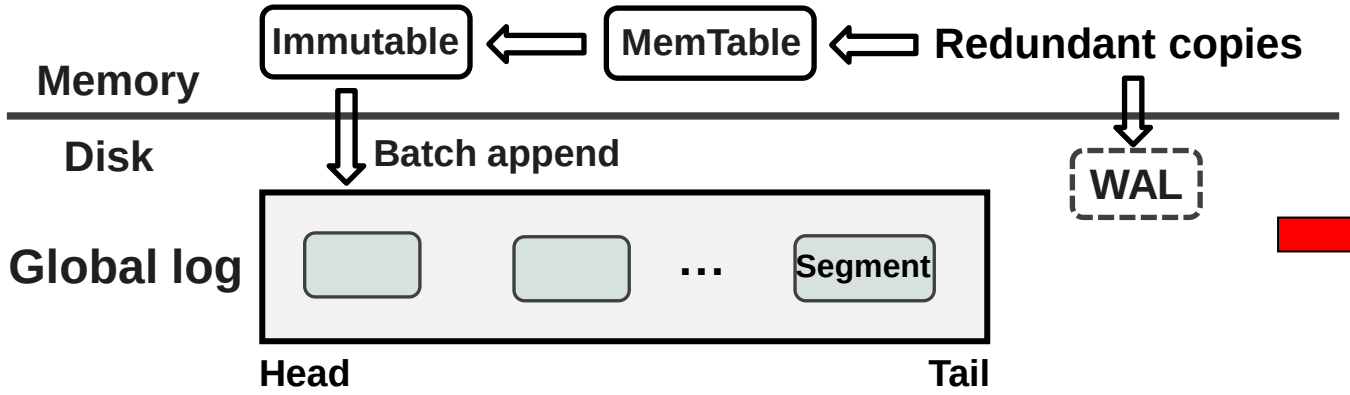
If **node ID=current node**, the KV is a primary copy; Otherwise, it is a redundant copy

Two-layer Log

➢ How to manage redundant copies efficiently?

Two-layer log with tunable ordering

- **Global log**: all redundant copies are appended in a batched manner



Write to global log in a batched & appended manner

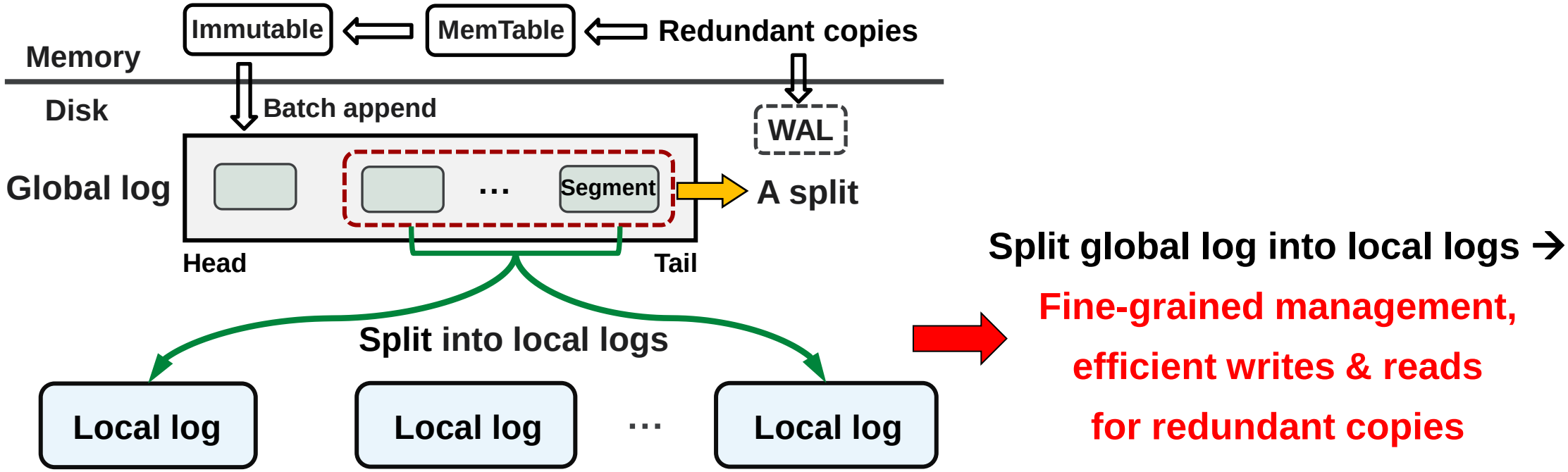
→ efficient writes

Two-layer Log

➢ How to manage redundant copies efficiently?

Two-layer log with tunable ordering

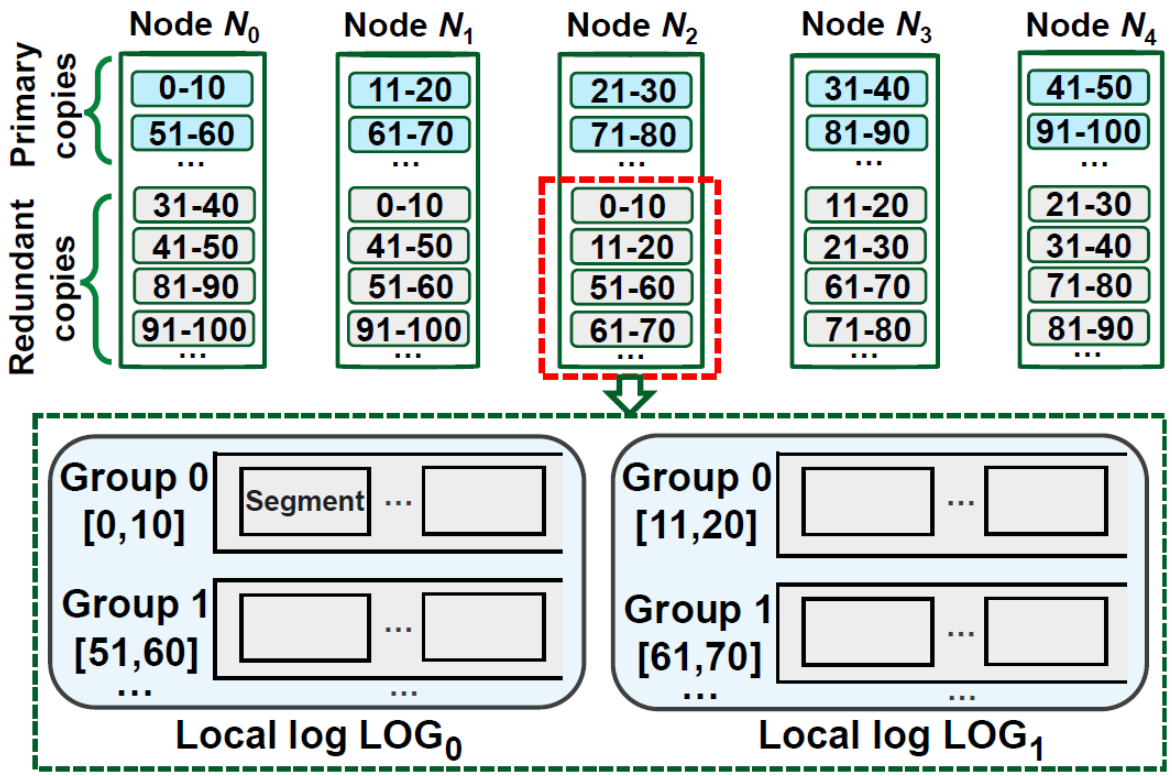
- **Local logs:** split global log into different local logs in the background



LOGi: stores redundant copies whose primary reside in node i

Two-layer Log

- How to manage redundant copies efficiently?
  - **Range-based data grouping within local logs**



- Each node stores several ranges → divide a local log into different groups
- **Benefits of range grouping:**

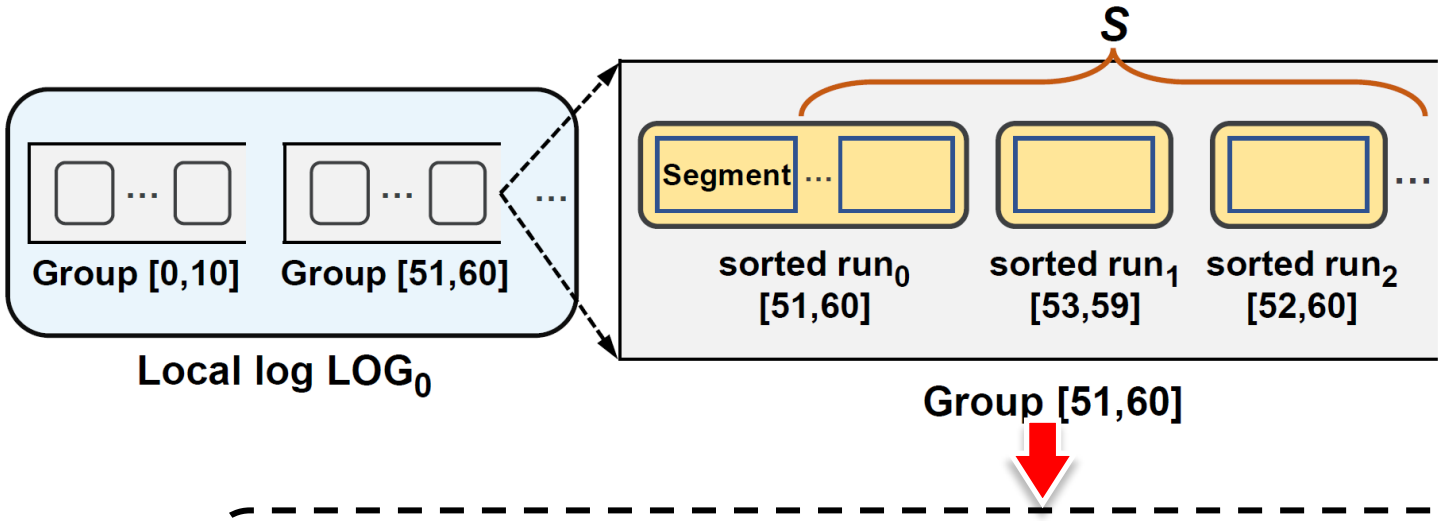
Efficient GC: each GC only selects one group, avoid scanning the whole local log

Efficient recovery: recovery reads only the corresponding group

Two-layer Log

➢ How to manage redundant copies efficiently?

- **Data organization within each group**
  - Each group contains **several sorted runs**
  - KV pairs **within** a sorted run are **fully sorted**, but **unsorted across** sorted runs



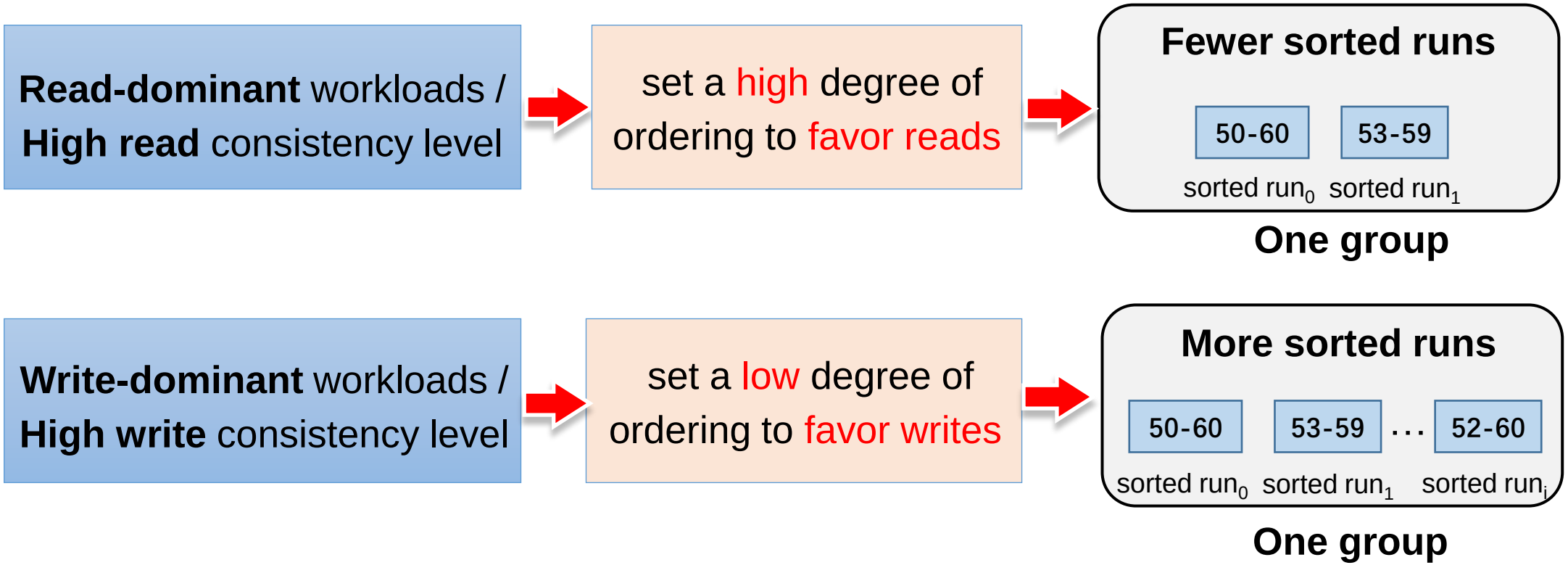
Adjust number of sorted runs (S)

Fewer sorted runs: high ordering, better read performance

More sorted runs: low ordering, better write performance

Two-layer Log

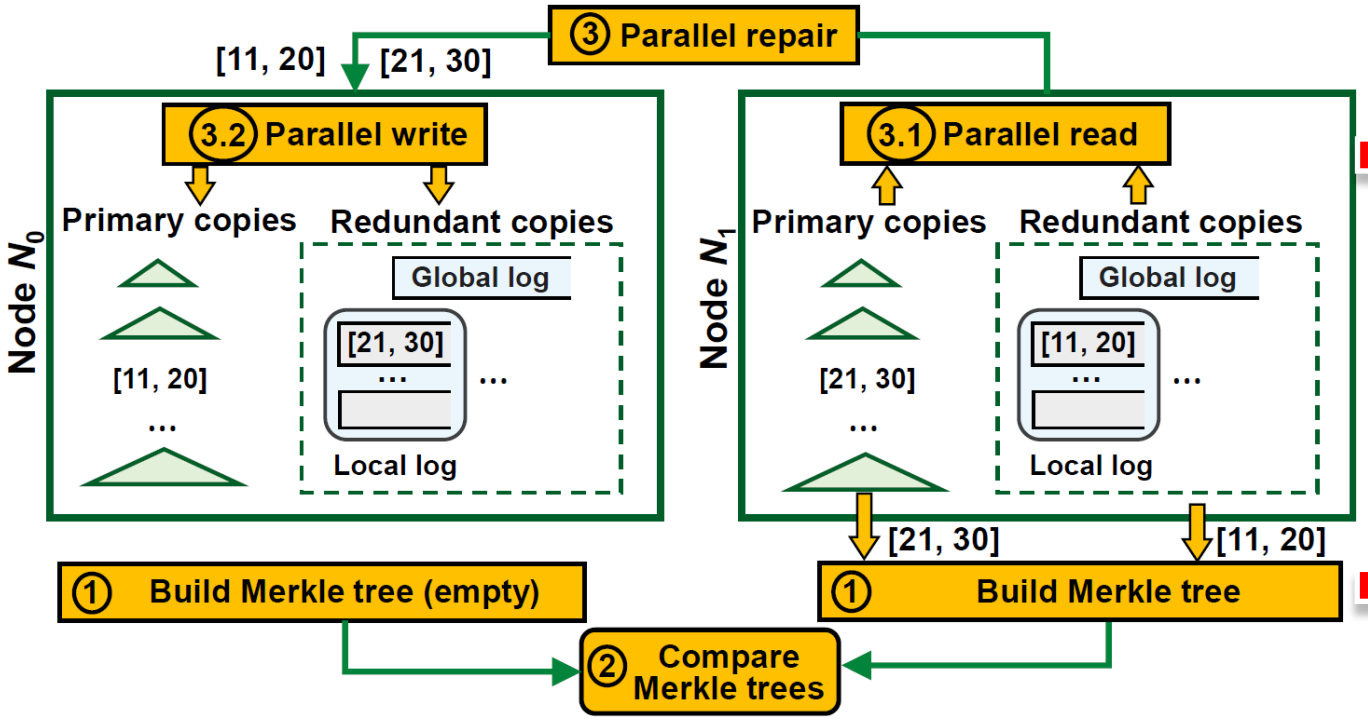
- How to adjust the ordering of the two-layer log?
  - Set different degrees of ordering based on performance requirements



Parallel Recovery

- How to perform recovery efficiently?

Parallel recovery to accelerate data repair



When repairing multiple ranges, **reading primary and redundant copies in parallel**

When building Merkle trees to detect lost data, two threads to **read primary** and **redundant copies in parallel**

Experiments

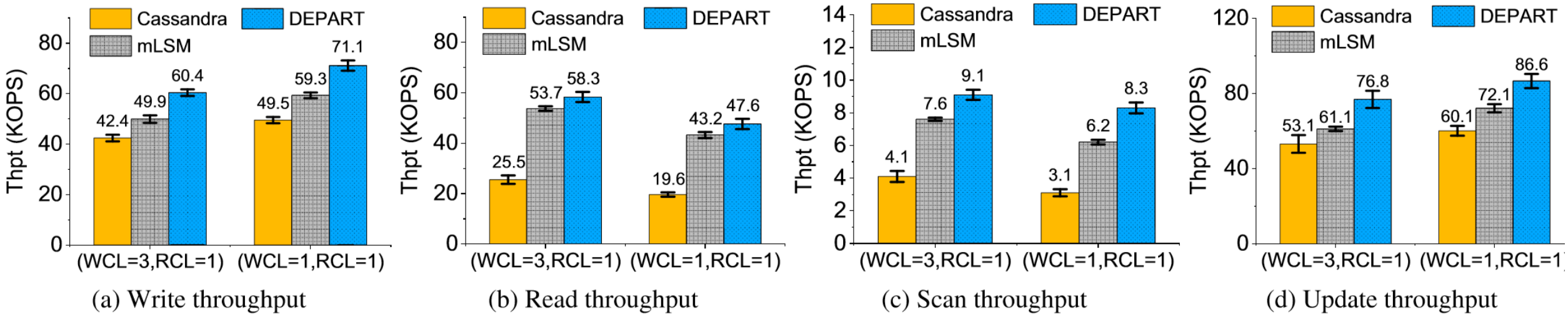
- Setup:
  - 6 nodes (5 storage nodes + 1 client node), 10 Gb/s Ethernet switch
  - **Workloads**: YCSB 0.15.0, 1KB KV pairs, Zipf distribution (0.99)
  - **Parameters**: three replicas, (WCL=ONE, RCL=ONE) by default
- Comparisons:
  - **Cassandra v3.11.4** VS **multiple LSM-trees (mLSM)** VS **DEPART**
  - DEPART builds on Cassandra v3.11.4

Server configuration

Machine	CPU	Memory	Disk	OS
6 nodes	12-core Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20 GHz 4	32-GiB DDR4 2400 MHz	500 GiB SSD	CentOS 7.6.1810 64-bit Linux kernel 3.10.0

Micro-benchmarks

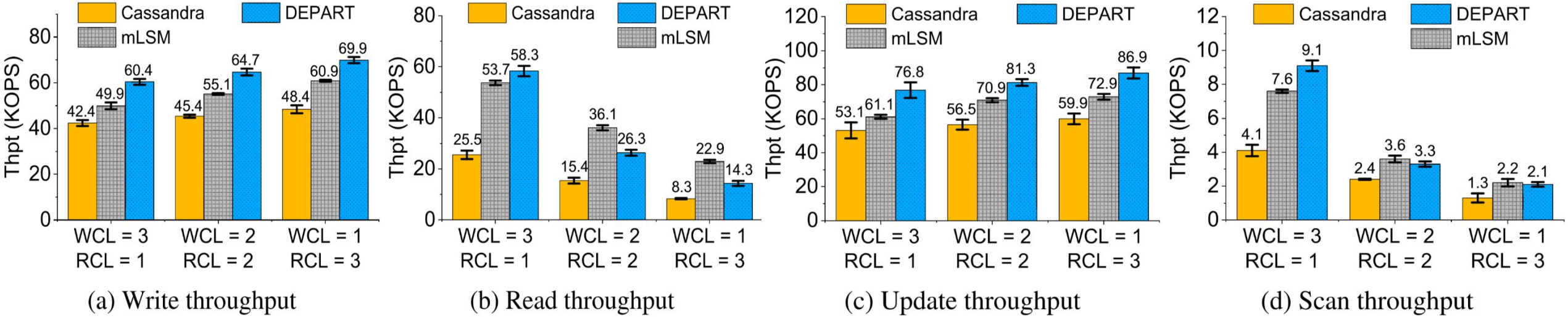
- Client first writes 200M KV pairs, followed by 20M reads, 2M scans, and 200M updates



Compared to **Cassandra**, **DEPART** improves **writes, reads, scans and updates** up to **1.43X, 2.43X, 2.68X and 1.44X;**
mLSM notably improves reads, but marginal improvements on writes

Consistency Configurations

- For strong consistency, configurations under triple replication:
  (WCL=3, RCL=1), (WCL=2, RCL=2), (WCL=1, RCL=3)

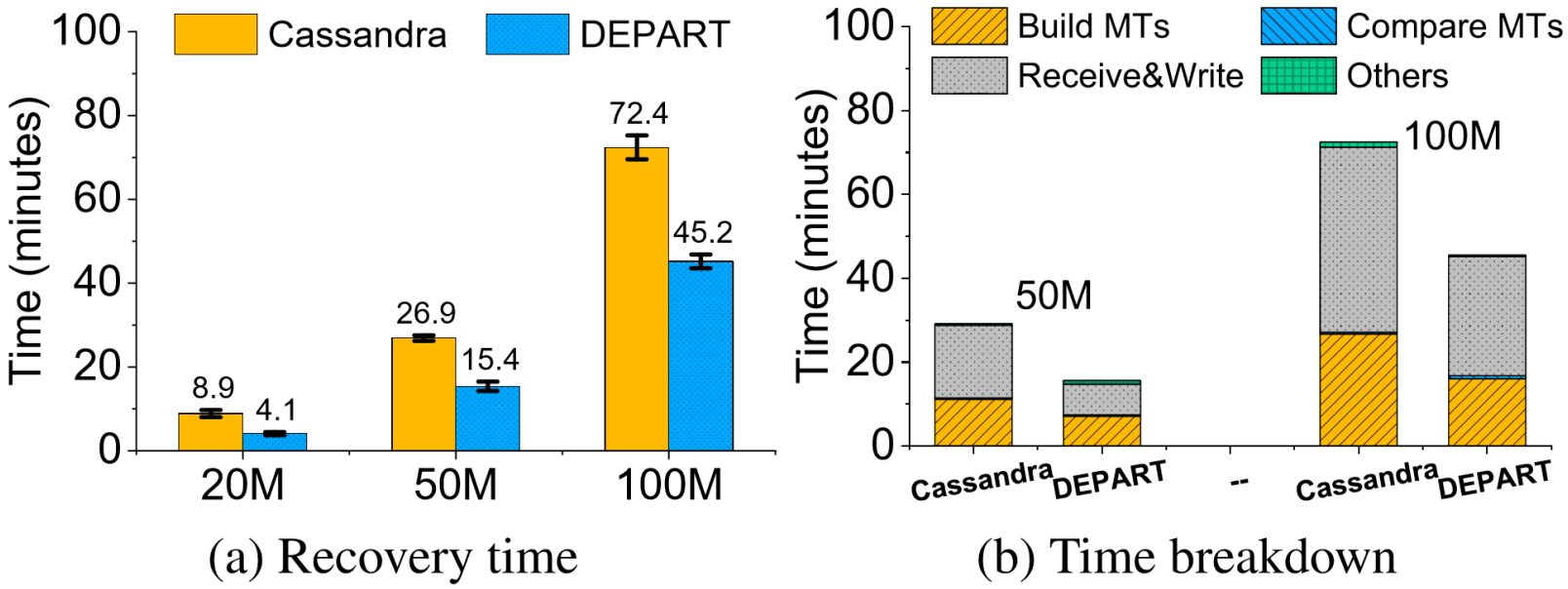


(a) Write throughput (b) Read throughput (c) Update throughput (d) Scan throughput

DEPART consistently improves writes, reads, scans, updates over **Cassandra**;
For **RCL ≥ 2**, the read gains of **DEPART** over Cassandra become smaller

Recovery Performance

- Write 20M, 50M,100M KV pairs; erase data in a node; recover



DEPART reduces recovery time of Cassandra **by 38-54%**; and DEPART reduces the time costs of Build MTs and Receive&Write **by nearly one half**

Impact of Ordering Degree S

- Write & read performance versus **ordering degree S**

S	Write thpt (KOPS)	Read thpt (KOPS)
1	37.2	42.3
10	57.2	31.5
20	64.7	23.1
$\rightarrow \infty$	78.4	7.6
Cassandra	45.4	15.4

Increasing **S from 1 to** ∞, the ordering of the two-layer log is relaxed and hence merge-sort overhead becomes smaller, so **write thpt increases but read thpt decreases**

Conclusions

- **DEPART**: **Replica decoupling** for **high-performance & reliable** distributed KV storage
  - Lightweight replica differentiation
  - Two-layer log design for redundant copies for fast writes & recovery
  - Tunable ordering to balance reads and writes
  - Parallel recovery for fast recovery

- More evaluation results and analysis are in the paper
- The source code is at https://github.com/ustcadsl/depart

Thanks for your attention!

Q&A

ykli@ustc.edu.cn

中国科学技术大学

University of Science and Technology of China