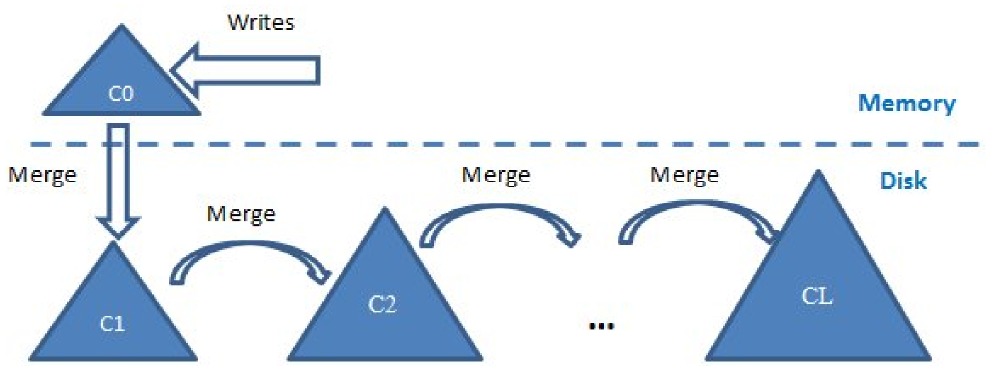
On Performance Stability in LSM-based Storage Systems

Check the website https://ahacad.github.io/LSM-pre for online slides.

A look ahead

On Performance Stability in LSM-based Storage Systems



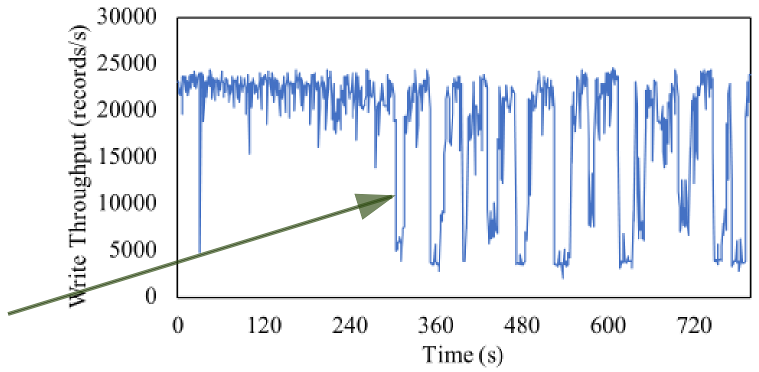


Figure 1: Instantaneous write throughput of RocksDB: writes are periodically stalled to wait for lagging merges

- LSM-tree is good at writing
- BUT memory FASTER THAN disk
- **writes stalls** happen when manipulating disks, and it affects usability
- let's try to solve it by tweaking **merge schedulers**

Table of contents

Roadmap

DATABASE STORAGE ENGINES

B-TREE

ORACLE

Microsoft SQL Server

IBM DB2

PostgreSQL

MySQL

mongoDB

Couchbase

LSM TREE

cassandra

APACHE HBASE

Google Cloud Bigtable

elasticsearch

influxdb

LEVELDB

RocksDB

yugabyteDB

LSM (Log-Structured Merge-Tree) - 1: Basics

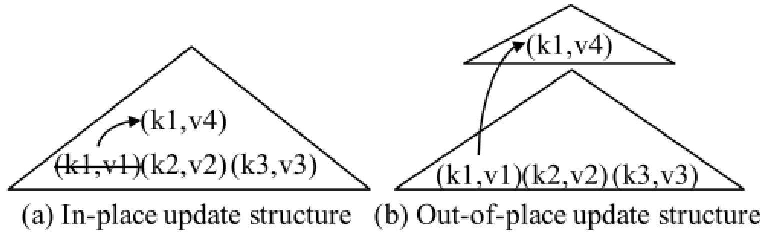


Fig. 1

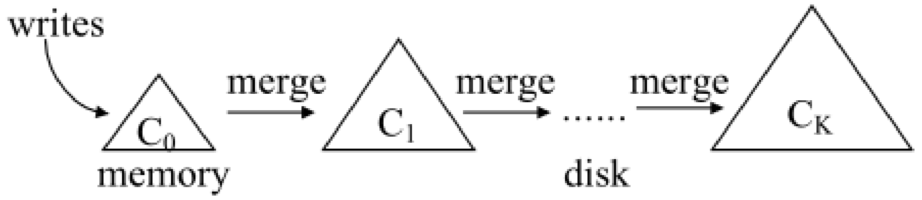


Fig. 2

- unlike B+ tree (in-place update) which overwrites old entries, we store updates into new locations (Fig. 1)
- only **append** operations (insert OR update), delete = anti-matter entry, like writing everything into a "log"
- buffers all writes in memory (C_0), and **flushes** them to disk and **merges** later (do compactions)(Fig. 2)
- quick write (due to sequential I/Os), but possibly slow read

LSM (Log-Structured Merge-Tree) - 2: Merging

- today's LSM-tree use **merging** to reduce components examined when querying (compaction)
- two merging policies: leveling merge policy & tiering merge policy
- **leveling**: 1 component, merged with Level $i-1$ until big enough and then merged into Level $i+1$
- **tiering**: T components at Level i, then merged together to Level $i+1$

- we'll analyze this in Part III

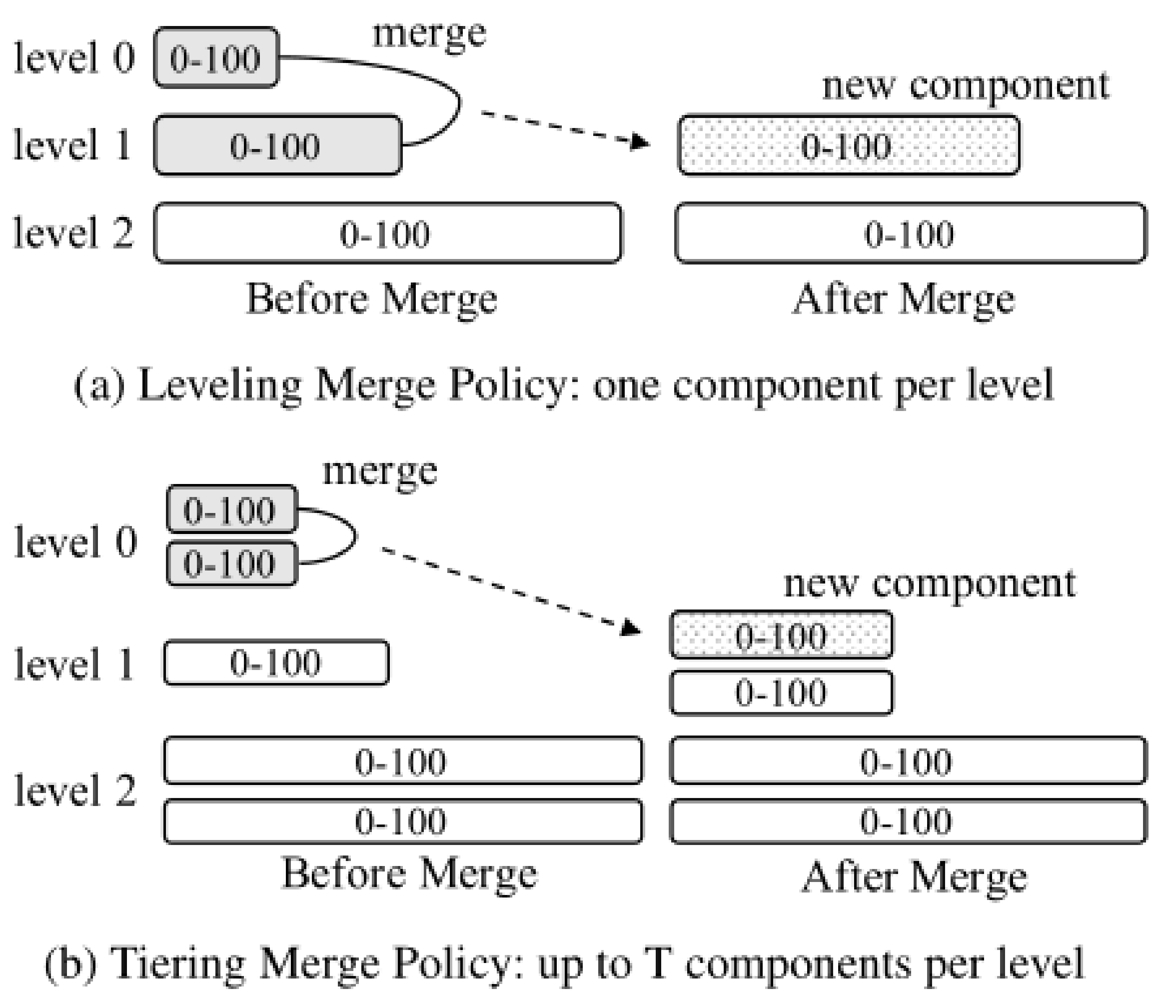
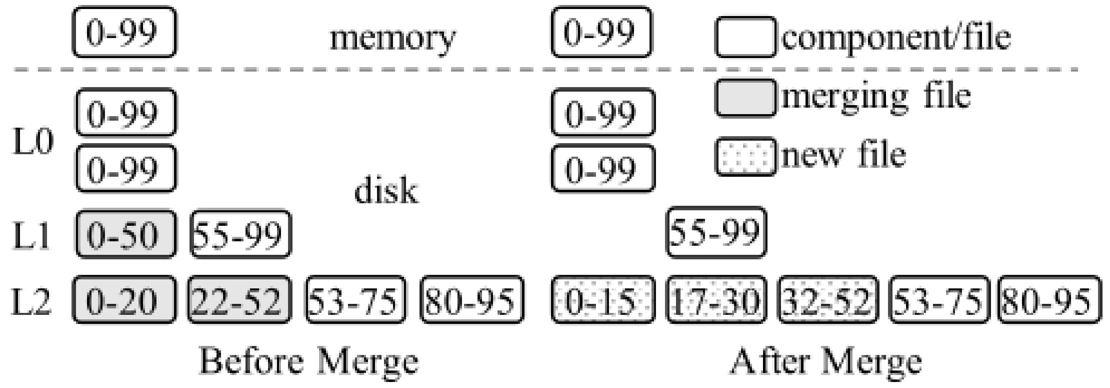


Fig. 1

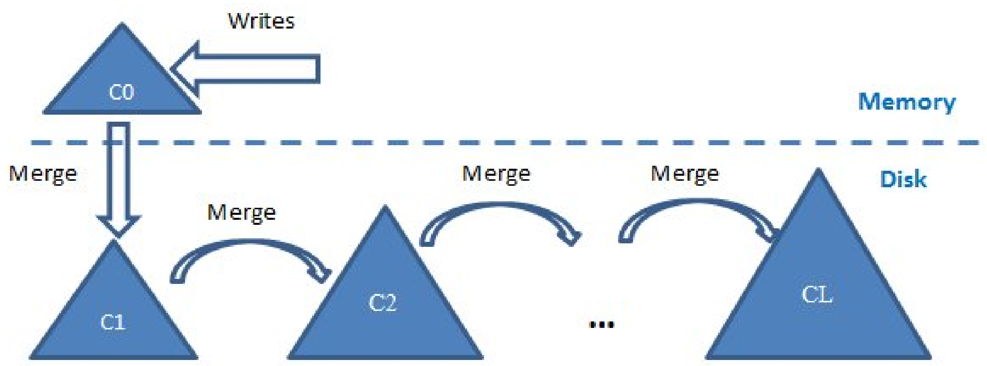
LSM (Log-Structured Merge-Tree) - 3: Partitioning



Partitioned LSM-tree, Leveling Merge Policy

- **Partitioning**: large LSM disk component range-partitioned into multiple files for optimization
- partitioning and merge policies can be used together, currently LevelDB and RocksDB use partitioned leveling policy
- **Write Stalls**: memory speed faster than I/Os, writing to memory will be **stalled** (the *write stall* problem)
- **merges** are major cause of stalls, since components are merged multiple times, but writes only flush once
- we'll analyze this in Part IV

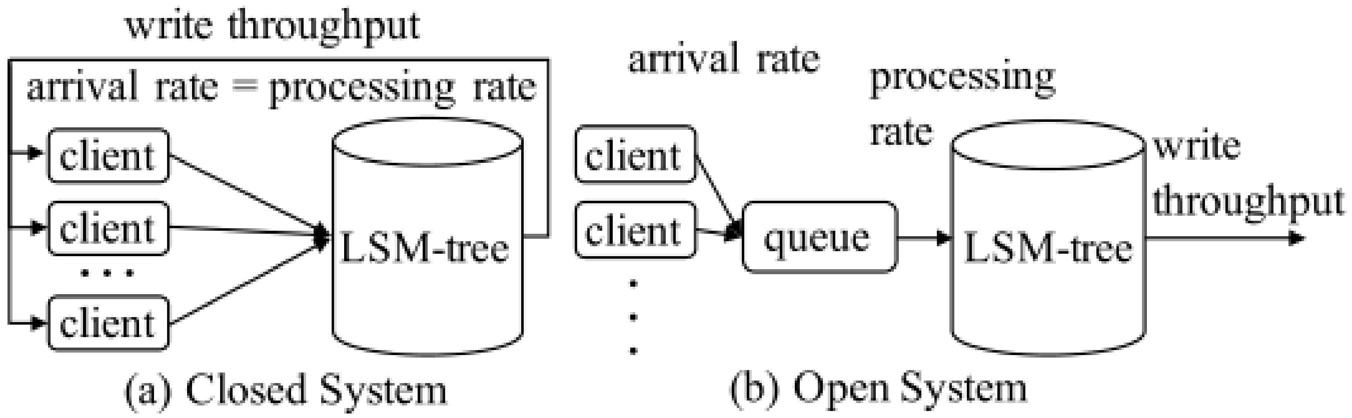
LSM (Log-Structured Merge-Tree): Recap



- multi-level, only append, flush and merge
- 2 merge policies: leveling and tiering
- 1 optimization: partitioning
- some keywords: component, write throughput, write stalls,
- We'll analyze LSM-tree later

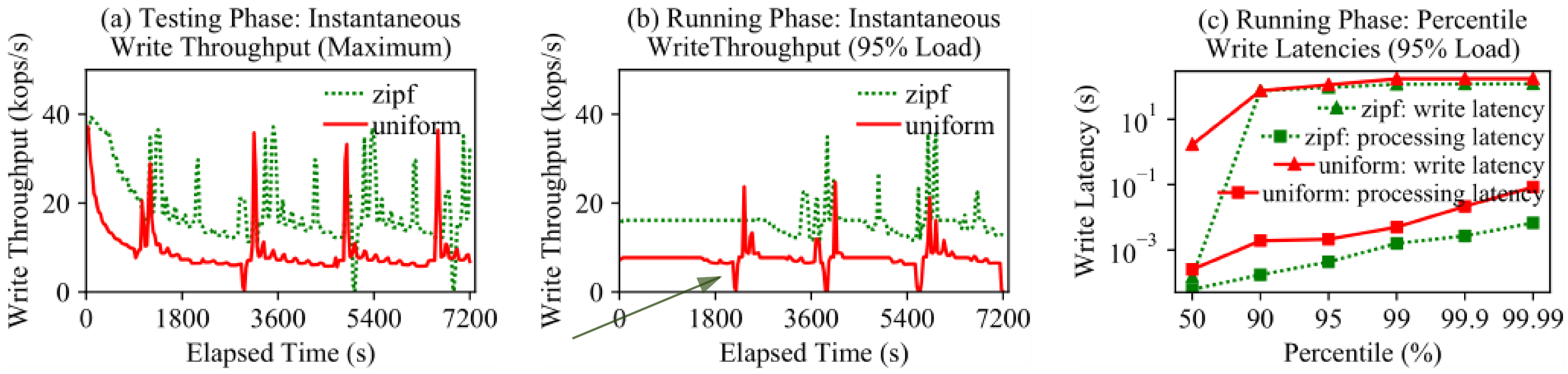
Roadmap

Measuring Latency: 2 phases



- testing phase: use the closed system to measure *maximum write throughput* (M)
- running phase: use the open system with a 95% M to see if the write latency will be stable
- previous experiments used only the closed system, but the open system is more realistic
- so we combine them both

Measuring Latency: Experiment



- write latency = processing latency + queuing latency
- clear write stalls
- let’s try to solve it

Roadmap

Full Merges Analyses - 1: Scheduling

- **global component constraint** better than local component constraint
- **process writes** as quickly as possible minimizes write latency
- **concurrent merges** matter:
  - process merges on each level concurrently
- finally, the paper proposes a **greedy scheduler** for better variability performances
  - 3 schedulers for comparison: the *single-thread* scheduler, the *fair* scheduler, and the **greedy** scheduler
  - a single-thread scheduler does not work concurrently
  - a *fair* scheduler allocate I/O bandwidth to all merges equally
  - a **greedy scheduler** prefers the merge with smallest remaining bytes first
  - the greedy scheduler minimizes number of components
- we are going to see *many* pictures next

Full Merges Analyses - 2: Variability

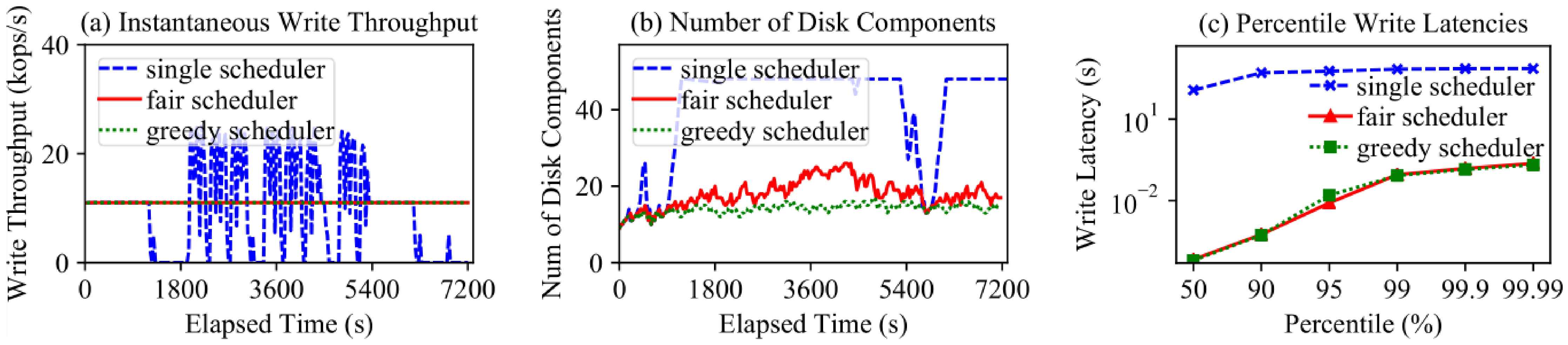


Figure 8: Running Phase of Tiering Merge Policy (95% Load)

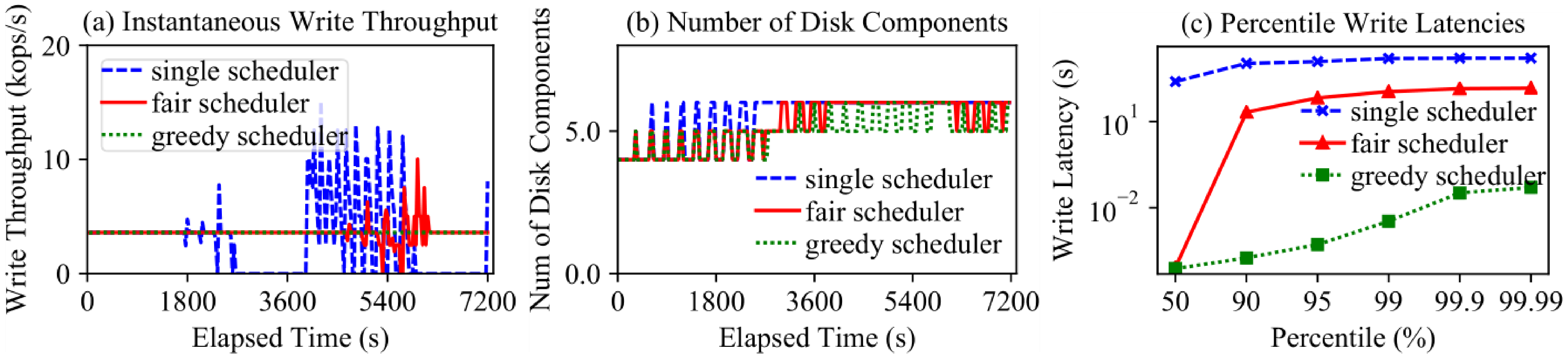


Figure 9: Running Phase of Leveling Merge Policy (95% Load)

- **stable write throughput** can be achieved
- the greedy scheduler does well

Full Merges Analyses - 3: Global Constaint

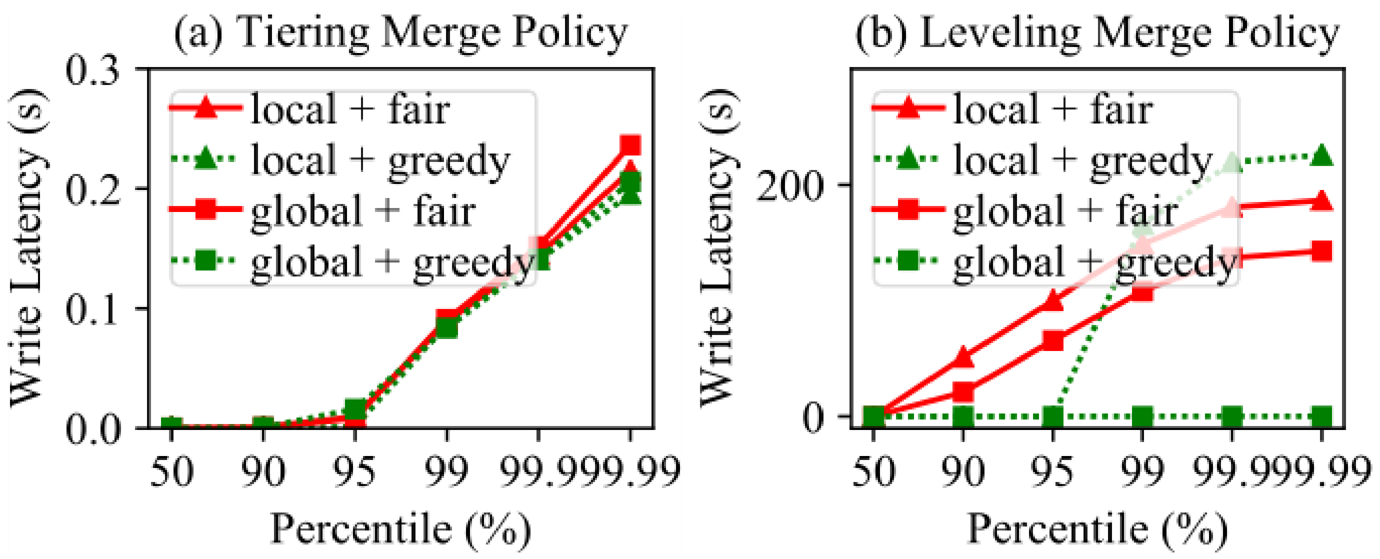


Figure 11: Impact of Enforcing Component Constraints on Percentile Write Latencies

- **global component constraint** better than local component constraint:
  - component constraint: limit on disk componet number
  - more components = less write stalls BUT worse query performance and takes more space
  - local constraint: limit on each level
  - global constraint: limit on all components

Full Merges Analyses - 4: Write Quickly

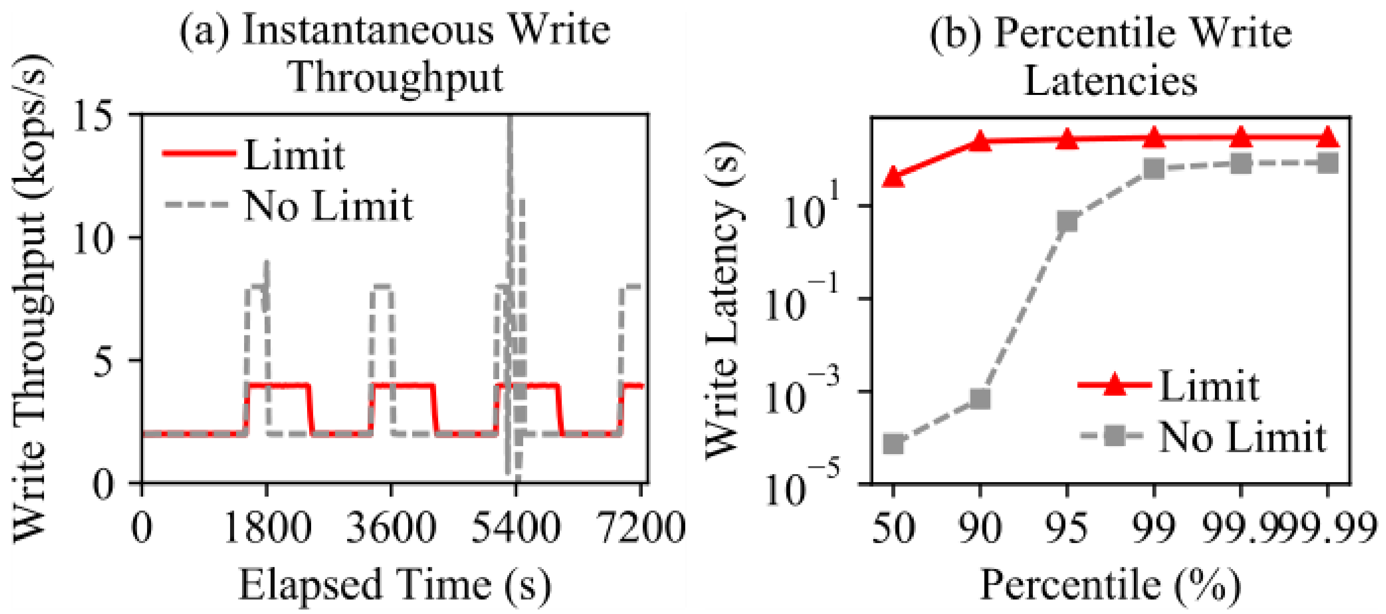


Figure 12: Running Phase with Burst Data Arrivals

- **process writes** as quickly as possible minimizes write latency:
  - when *component constraint* violated, needs to slow down or stop writes
  - current implementations (LevelDB, RocksDB, bLSM) prefers slowdown

Full Merges Analyses - 5: Query Performance Analyses

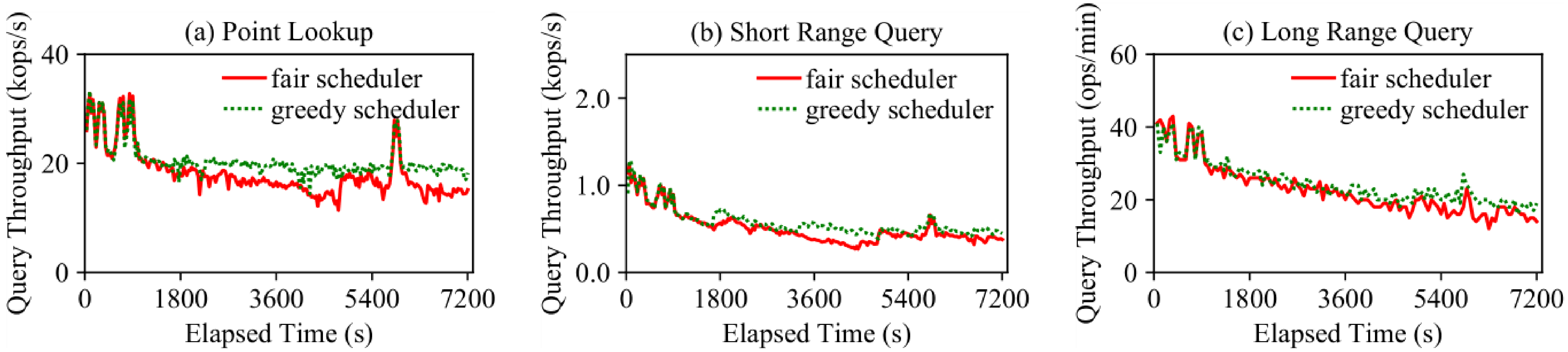


Figure 13: Instantaneous Query Throughput of Tiering Merge Policy

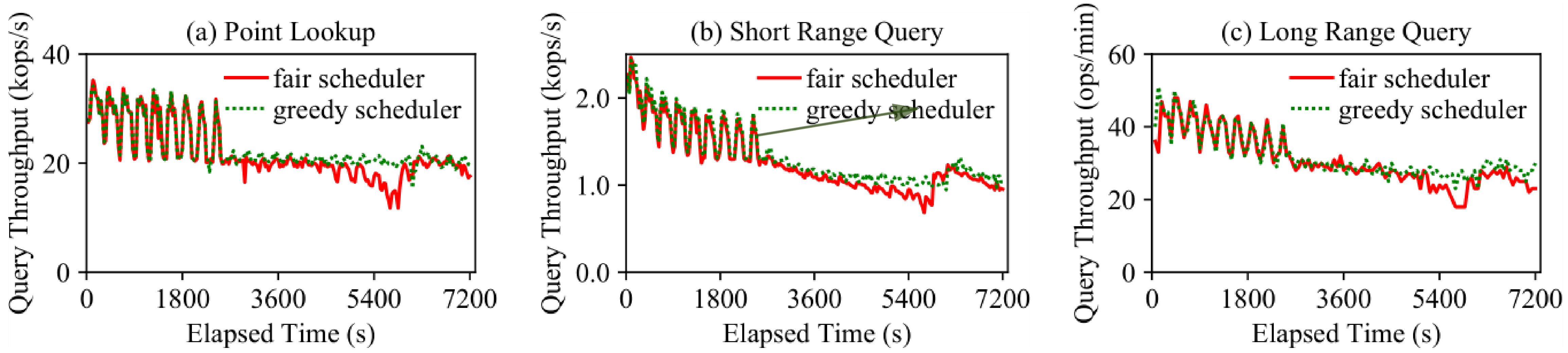


Figure 14: Instantaneous Query Throughput of Leveling Merge Policy

- use Bloom filters to speed up point lookup
- greedy scheduler has better performance because it minimizes components

Full Merges Analyses - 6: Size Ratio

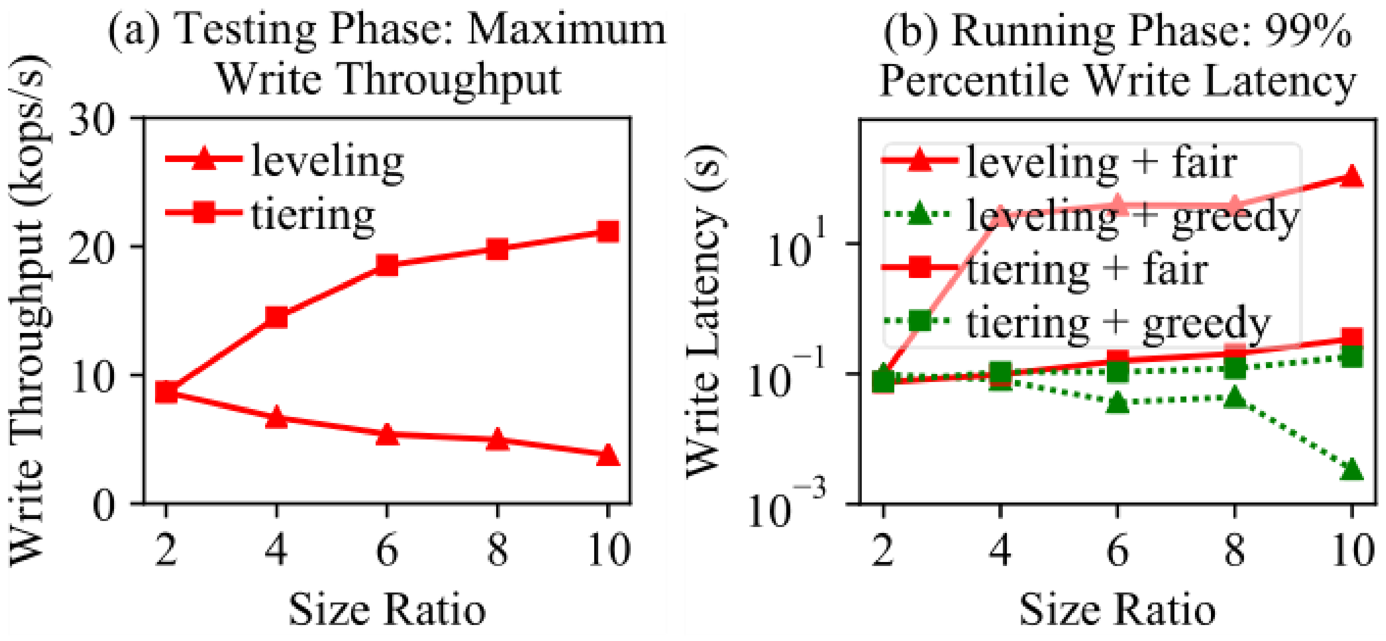


Figure 10: Impact of Size Ratio on Write Stalls

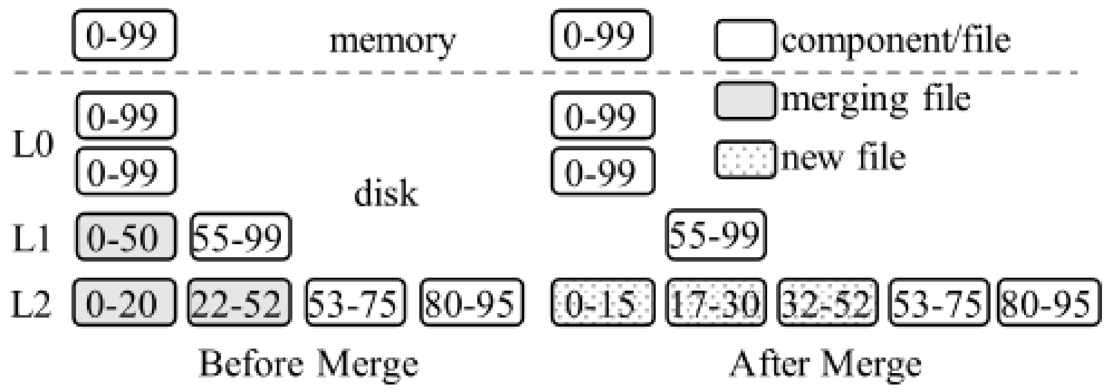
- higer size ratio means fewer merges for tiering

Full Merges Analyses: Recap

- utilize **concurrency** schedulers
- a proposed **greedy scheduler** works well
- some other analyses for possible improving:
  - gloabl component constraint
  - write quickly
  - size ratio (T)

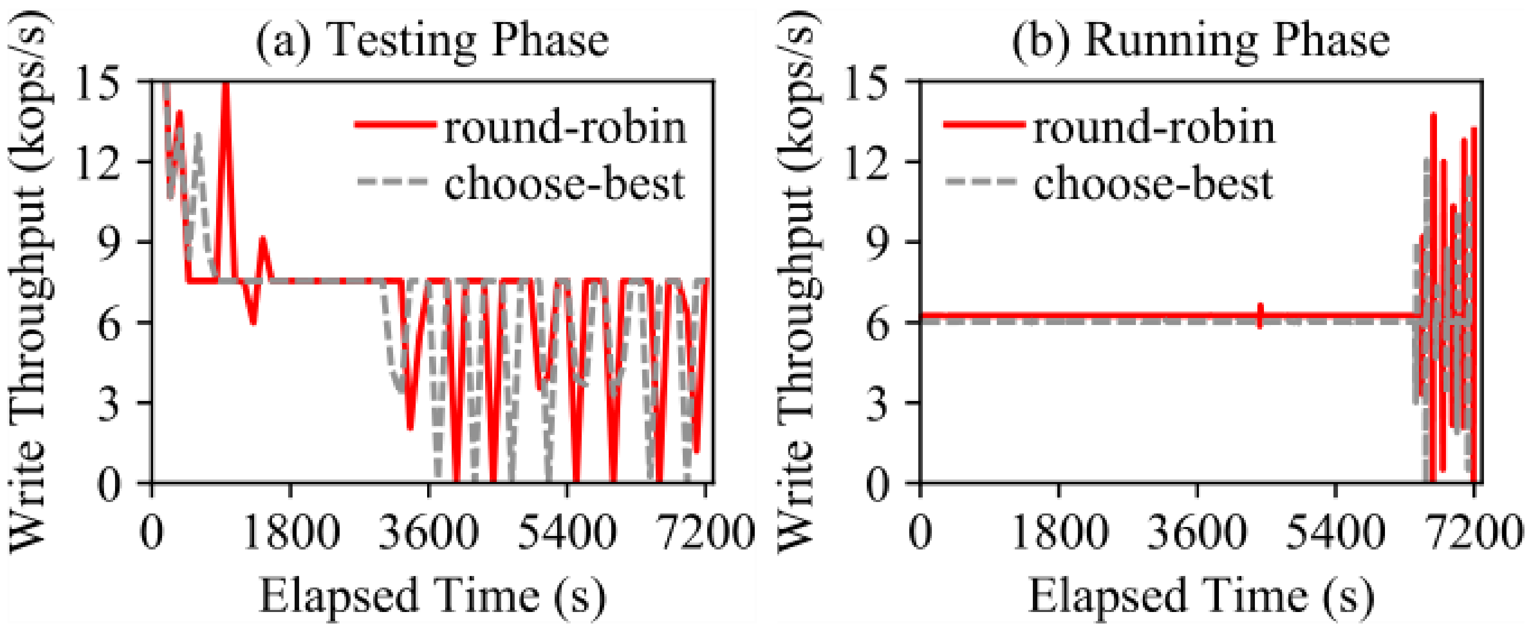
Roadmap

Partitioned Merges Analyses - 1



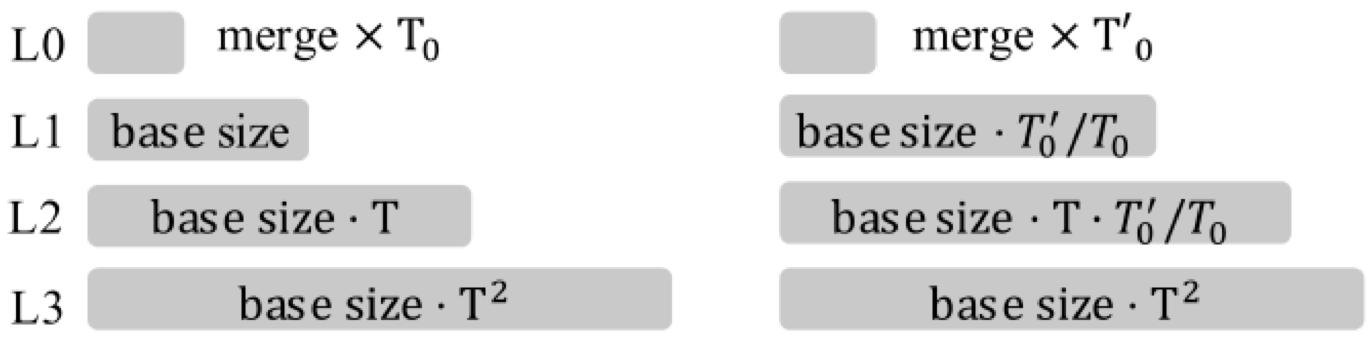
- single-threaded scheduler will be enough, since merges happen once a level is full
- use LevelDB to analyze

Partitioned Merges Analyses - 2



- it's ok, but emmm… not so good

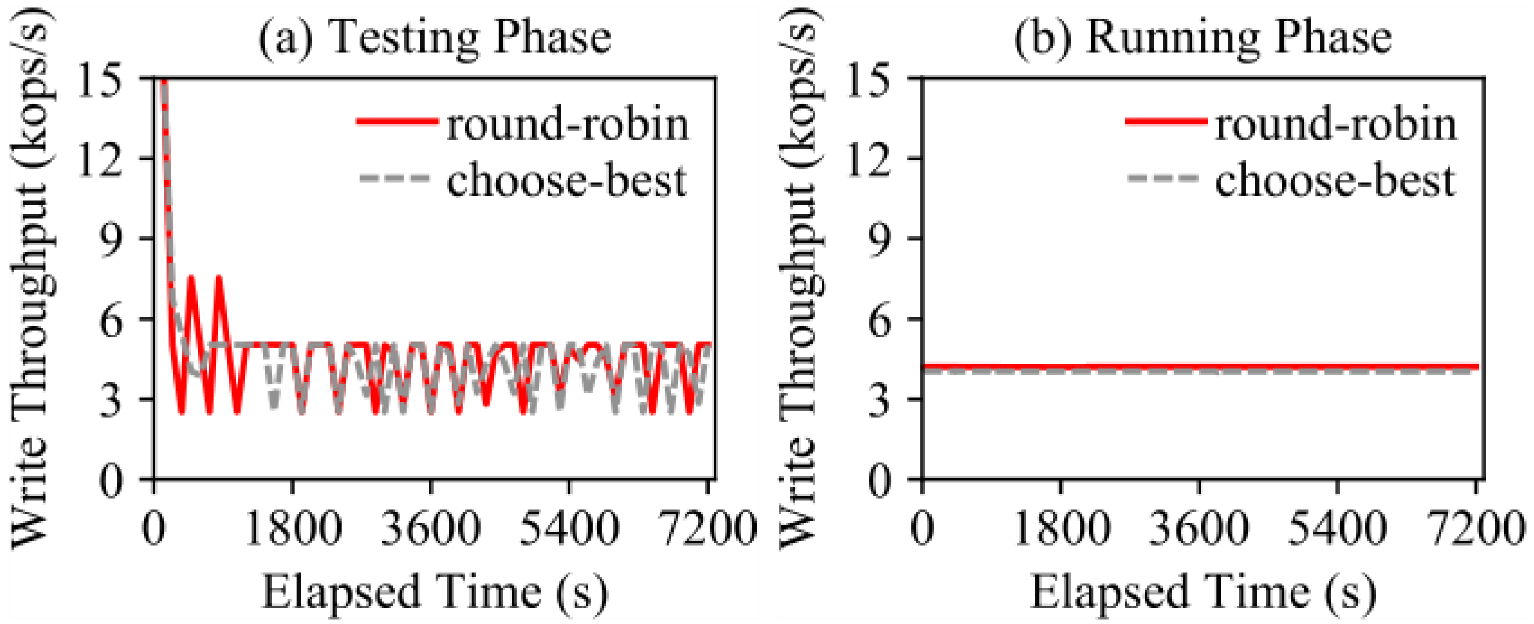
Partitioned Merges Analyses - 3



(a) Expected LSM-tree (b) Actual LSM-tree

- T_0 : minimum number of mergeable components, T'_0 : maximum number of mergeable components
- the problem: in the tesing phase a LSM-tree shifts to (b) because writes come quickly
- causes:
  - may cause write stalls in the running phase
  - suboptimal trade-offs because ratio no longer the same[3]
  - more disk usage
- we just explicitly ensure that always merge T_0 components

Partitioned Merges Analyses - 4



- fixed the problem last slide
- the single-threaded scheduler is enough to achieve stable write throughput

Roadmap

Lessons and Conclusions

- consider **performance variance** together with write throughput for usability
- use the new two-phase approach to evaluate the impact of write stalls
- a good scheduler can help achieve stable write throughput:
  - for full merges, the proposed **greedy scheduler**
  - for partitioned merges, the single-threaded scheduler is enough

Roadmap

References

- [1]C. Luo and M. J. Carey, “On performance stability in LSM-based storage systems,” Proc. VLDB Endow., vol. 13, no. 4, pp. 449–462, Dec. 2019, doi: 10.14778/3372716.3372719.
- [2] C. Luo and M. J. Carey, “LSM-based Storage Techniques: A Survey,” The VLDB Journal, vol. 29, no. 1, pp. 393–418, Jan. 2020, doi: 10.1007/s00778-019-00555-y.
- [3] P. O’Neil, E. Cheng, D. Gawlick, and E. O’Neil, “The log-structured merge-tree (LSM-tree),” Acta Informatica, vol. 33, no. 4, pp. 351–385, Jun. 1996, doi: 10.1007/s002360050048.
- [4] R. Sears and R. Ramakrishnan, “bLSM: a general purpose log structured merge tree,” in Proceedings of the 2012 international conference on Management of Data - SIGMOD ’12, Scottsdale, Arizona, USA, 2012, p. 217. doi: 10.1145/2213836.2213862.
- [5] P. Guo, “Log Structured Merge Tree.” [Online]. Available: https://lrita.github.io/images/posts/database/lsmtree-170129180333.pdf
- [6] “The Log-Structured Merge-Tree (LSM Tree) | the morning paper.” https://blog.acolyer.org/2014/11/26/the-log-structured-merge-tree-lsm-tree/ (accessed May 30, 2021).

References

- [7] “RocksDB | A persistent key-value store,” RocksDB. http://rocksdb.org/ (accessed Jun. 05, 2021).
- [8] google/leveldb. Google, 2021. Accessed: Jun. 05, 2021. [Online]. Available: https://github.com/google/leveldb
- [9] K. Rott, intfrr/lsmtree. 2021. Accessed: Jun. 05, 2021. [Online]. Available: https://github.com/intfrr/lsmtree
- Tip: search "lsm-tree" or similar keywords on GitHub for community implementations
- See more detailed parameters analyses in the paper!