How to Grow an LSM-tree?

Towards Bridging the Gap Between Theory and Practice

Presenters:

Jovan Kascelan

Zhixin Li

Introduction - Breaking Down the Title

How to Grow an LSM-tree?

This is a paper about LSM-trees and their growth schemes.

Towards Bridging the Gap Between Theory and Practice

Theory: “the horizontal scheme can provide better overall read-write trade-off”

Practice: “most recent key-value stores … tend to adopt the vertical scheme” due to practical limitations

Introduction - The Concept of a LSM-Tree

LSM-tree is a key-value data structure,

extensively employed as the backbone of various Key-Value Storage Systems (such as RocksDB, Cassandra, TiKV).

When data enters LSM-tree:

Growing schemes decide when to trigger compaction;

Compaction policies (Leveling & tiering) decides how to compact

Expanding on that...

Introduction - The Concept of a LSM-Tree

Recall: Leveling and Tiering

Governs how should the tree perform compaction when needed

- Leveling: immediately compact to the next level via merge-sort
- Tiering: send a sorted-run to the next level, merge if needed later

vs.

Growth scheme governs a key question:

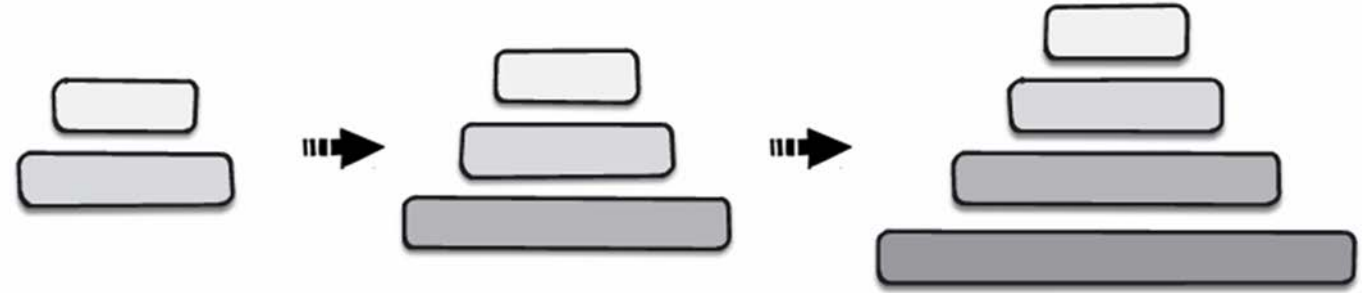
How should the tree expand as more data arrives? This is also to ask: when to compact?

"Currently, the growth schemes adopted in mainstream LSM-based key-value stores fall into two major categories":

- Vertical: expand the number of levels
- Horizontal: expand the size of levels

Introduction - The Two Growth Scheme Catagories

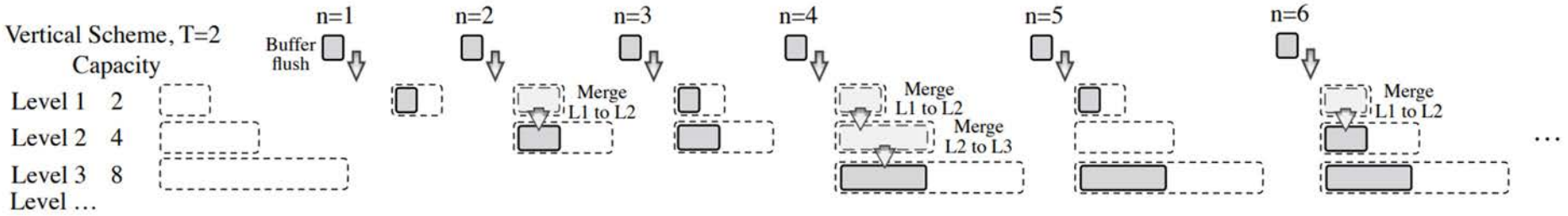
(a) Vertical Scheme: fixed level capacities, growing number of levels



The capacity of each level:
    Fixed to exponential of the previous level
The number of levels:
    Grows over time

Introduction - The Two Growth Scheme Catagories

(a) Vertical Scheme: fixed level capacities, growing number of levels



When data entries are flushed from buffer:

If any level i reached full capacity, merge to the next level i +1;

If there don't exist a next level, create a new level to allow compaction.

Introduction - The Two Growth Scheme Catagories

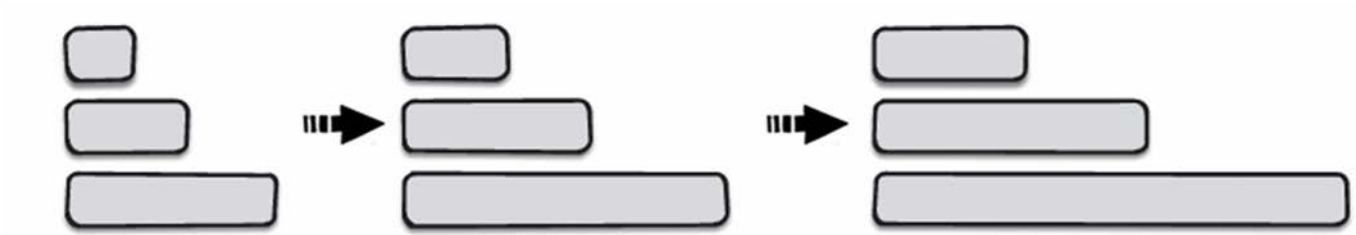
(b) Horizontal Scheme: fixed number of levels, growing level capacities

The capacity of each level:

Grows over time

The number of levels:

Fixed



Wh

Decide whether to compact by heuristics

Introduction - The Two Growth Scheme Catagories

(b) Horizontal Scheme: fixed number of levels, growing level capacities

When data entries is flushed from buffer:

Let C_i be the times of compactions conducted from Level i – 1 to Level i since the previous compaction from Level i to Level i + 1 …

A full compaction from Level i to Level i + 1 is triggered whenever $C\_i > C\_i+1$

Algorithm 1: Horizontal Scheme

Input: Maximum number of levels ℓ

```
for i = 1 to ℓ do
    C_i ← 0
for each flush from buffer do
    C_1 ← C_1 + 1;
    for i = 1 to ℓ − 1 do
        if C_i > C_{i+1} then
            Trigger a full compaction from Level i to Level i + 1;
            C_{i+1} ← C_{i+1} + 1, C_i ← 0;
```

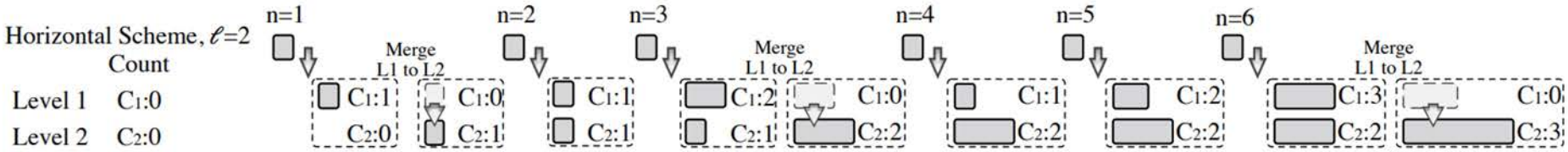
Introduction - The Two Growth Scheme Catagories

(b) Horizontal Scheme: fixed number of levels, level capacities grow

When data entries is flushed from buffer:

Let C_i be the times of compactions conducted from Level i – 1 to Level i since the previous compaction from Level i to Level i + 1 …

A full compaction from Level i to Level i + 1 is triggered whenever $C\_i > C\_i$+1



Introduction - Theory and Practice

Horizontal growth scheme has theoretical "better overall read-write trade-off" due to:

1) Full compactions are expensive
2) Horizontal Scheme has decreasing compaction frequency

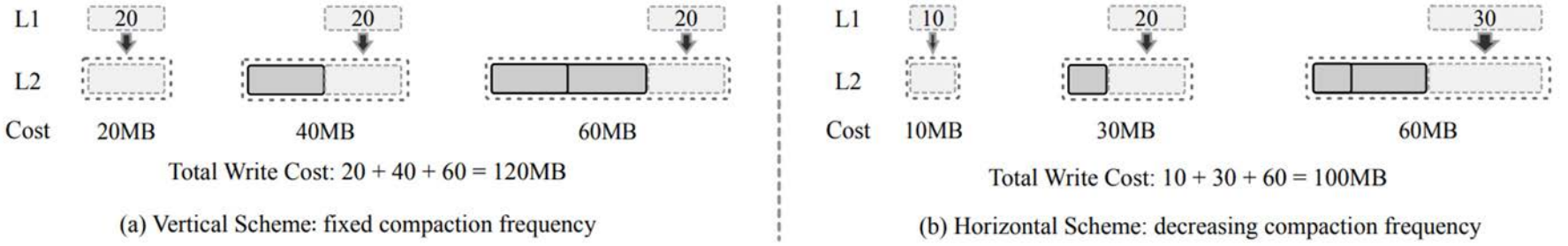


Fig. 3. Illustrating the cost of different schemes.

Also proven by Bentley and Saxe [7] (later refined by Mathieu et al. [45] for LSM-tree context)

Introduction - Theory and Practice

In practice horizontal schemes underperform or are avoided, “most recent key-value stores … tend to adopt the vertical scheme” due to practical limitations:

1) Limited capability in handling update-heavy workloads ✗
    a) Update-deavy workload is common in practical applications
    b) Vertical schemes handle this by using tiering, while Horizontal schemes are not previously designed to do that
    c) Horizontal schemes preforms sub-optimal if directly use tiering
2) Lack of partial compaction ✗
    a) File-level partial compaction granularity is the standard in most industrial systems

The Problem - Imperfect Growth Scheme Hurts System Performance

A suboptimal growth scheme

Leads to - Inappropriate compaction timing

Therefore - Higher read/write amplification

Requiring more disk scans

Consequently - Shortening hardware lifespan

Raised cloud costs

Increased latency and space usage

The Problem - Intuition 1

Currently, leveling is widely adopted as the default;

Recent design that adopt tiering with Vertical Growth Scheme “increas[es] its flexibility in balancing read[-]write trade-offs”

However, Tiering with Horizontal Growth Scheme is undiscovered.

Challenge: has not been accomplished before & notoriously difficult

Growth Scheme	Vertical	Horizontal
Leveling Merge	LevelDB, RocksDB, WiredTiger, BadgerDB ...	BigTable, HBase, AsterixDB
Tiering Merge	Cassandra, ScyllaDB	Not exist

Table 1. The landscape of the two growth schemes.

The Problem - Intuition 2

Can we combine the two different growth schemes?

Is it possible to hybridize the two methods?

Challenge:

- Difficult to fully leverages the strengths of both
- Difficult to adapt to different workloads

Proposed Solutions

The paper proposes two novel techniques to address the challenges:

1. Horizontal-Tiering
2. Vertiorizon

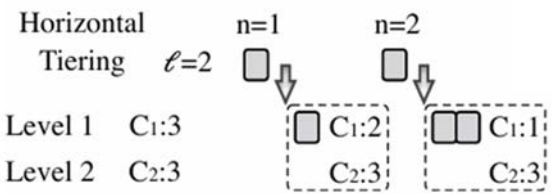
Designs	Vertical-Leveling	Vertical-Tiering	Horizontal-Leveling	Horizontal-Tiering **(ours)**	VERTIORIZON **(ours)**
Update-Heavy	★	★★★★	★★★	★★★★★	★★★★★
Lookup-heavy	★★★★★	★★	★★★★★	★★★	★★★★★
Space	★★★★	★★★★	★★★	★★★	★★★★

Table 2. Comparing existing growth schemes and ours.

Horizontal-Tiering

Enhanced horizontal scheme proposed: Horizontal-Tiering

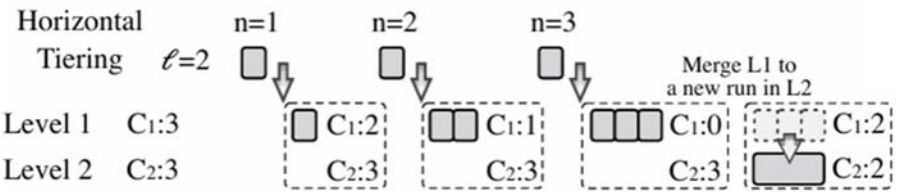
Extends horizontal applicability to tiering merge policy



Horizontal-Tiering

Enhanced horizontal scheme proposed: Horizontal-Tiering

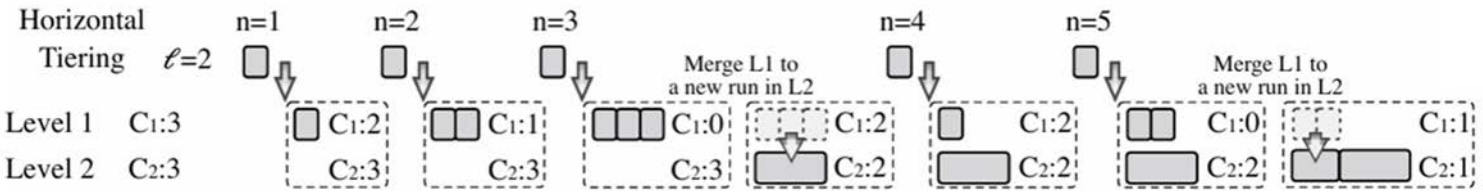
Extends horizontal applicability to tiering merge policy



Horizontal-Tiering

Enhanced horizontal scheme proposed: Horizontal-Tiering

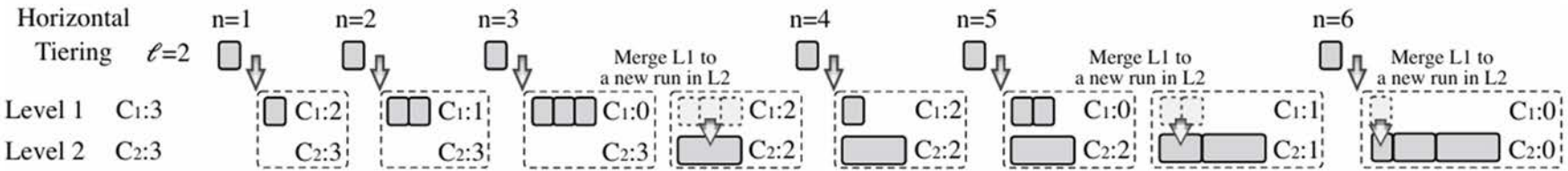
Extends horizontal applicability to tiering merge policy



Horizontal-Tiering

Enhanced horizontal scheme proposed: Horizontal-Tiering

Extends horizontal applicability to tiering merge policy



Analysis

ℓ: number of levels

k: initial counter value

N: total data size

B: buffer size

LEMMA 4.1. *Under the horizontal-tiering scheme, if we initially set the compaction counters of all levels to k, then after* $\binom{k+\ell-1}{\ell}$ *buffer flushes, the compaction counters of all levels will decrease to zero.*

Find smallest k that satisfies the inequality $\frac{N}{B} \leq \binom{k+\ell-1}{\ell}$

Analysis: Part 2

PROBLEM 1. *Consider a workload that would incur a total of n sequential buffer flushes in the LSM-tree, which has a maximum of ℓ levels. Our objective is to find the optimal compaction sequence $S^* = \{p_1^*, p_2^*, ...\}$ to minimize the total read cost of processing this workload. We define this problem as $\psi(n, \ell)$.*

THEOREM 4.2. *When there are total data of size $N = \binom{k+\ell-1}{\ell} \cdot B$ written into the LSM-tree, the compaction operations in the horizontal-tiering scheme described in Algorithm 2 is optimal for Problem 1.*

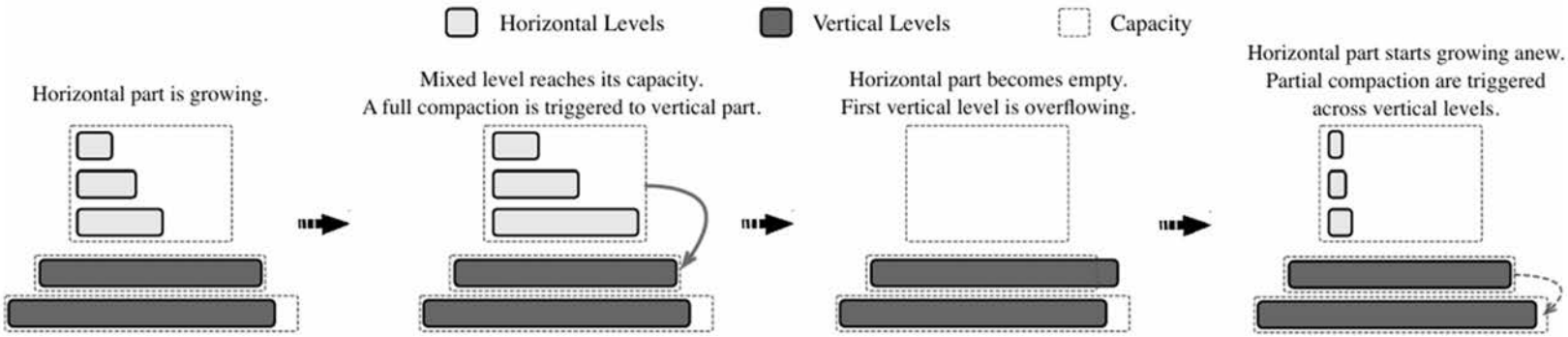
Vertiorizon

Hybrid approach:

Upper levels have horizontal-style

Lower levels have vertical-style

Dynamically increased capacity for the horizontal part (n=n*(1+1/T))



Optimizing Write Amplification

Default write amplification for the vertical part: T+(T+1)/2=3T/2+1/2

Change size ratio from T,T to $T', T^2/T'$

AM-GM inequality: $\frac{x+y}{2} \geq \sqrt{xy}$

New write amplification is $T' + (\frac{T^2}{T'} + 1)/2 \geq 2\sqrt{T' \cdot \frac{T^2}{2T'}} + \frac{1}{2} = \sqrt{2}T + \frac{1}{2}$

Equality achieved when $T' = \frac{T^2}{2T'}$ or $T' = T/\sqrt{2}$

Navigating Toward the Best Design

Minimizing the weighted sum of:

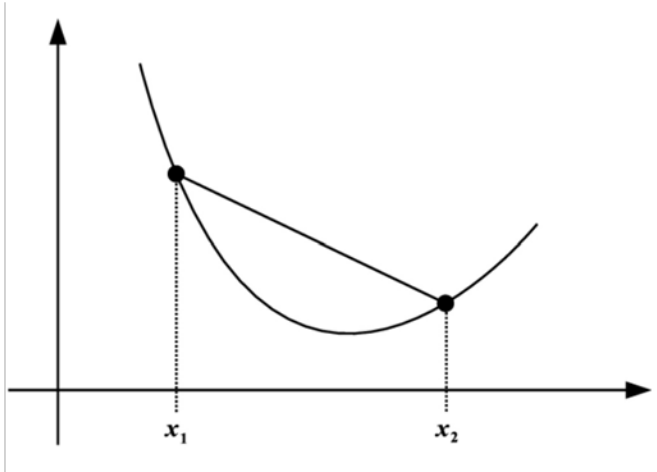
1) Update cost
2) Point lookup cost
3) Range lookup cost

$$\zeta = w \cdot W + r \cdot R + q \cdot Q$$

Optimal setting:

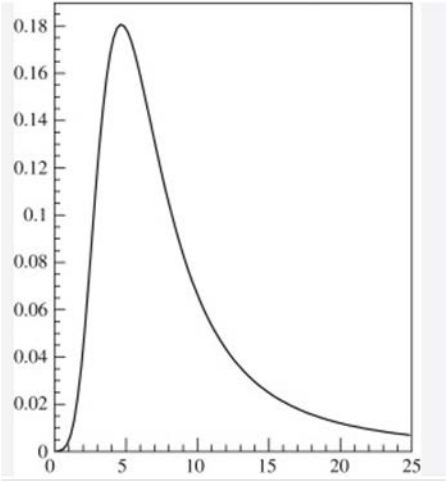
Binary search to find the saddle point

of the convex cost function

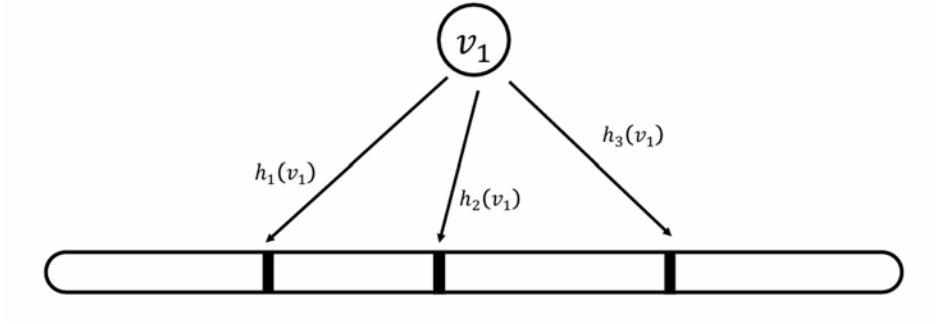


Additional optimizations

Perform compactions less frequently for skewed workloads



Dynamic bloom filters that are reconstructed during full compactions



Analysis: Point lookup cost

l: number of levels

f: Bloom filter false positive rate

n: capacity of the horizontal part

m: integer that satisfies $\binom{m}{\ell} \leq n \leq \binom{m+1}{\ell}$

$$R_l = \ell \cdot f$$

$$R_t = \left(\ell \cdot \binom{m}{\ell+1} + (m - \ell + 1) \cdot \left(n - \binom{m}{\ell}\right)\right) \cdot \frac{f}{n}$$

Analysis: Range lookup cost

1) Point lookup cost

$$R_l = \ell \cdot f \qquad R_t = \left(\ell \cdot \binom{m}{\ell+1} + (m - \ell + 1) \cdot \left(n - \binom{m}{\ell}\right)\right) \cdot \frac{f}{n}$$

1) Range lookup cost

$$Q_l = \frac{R_l}{f} \qquad Q_t = \frac{R_t}{f}$$

Analysis: Update cost

l: number of levels

n: capacity of the horizontal part

m: integer that satisfies $\binom{m}{\ell} \leq n \leq \binom{m+1}{\ell}$

P: page size in entries

$$W_l = \left(\ell \cdot \binom{m+1}{\ell+1} + (m+1) \cdot \left(n - \binom{m}{\ell}\right) - (\ell - 1) \cdot n\right) \cdot \frac{1}{n \cdot P} \qquad W_t = \frac{\ell}{P}$$

Experimental Setting

Write buffer size: 2MB

Block cache size: 32 MB

Bloom filter bits per key: 5

$T = 6$

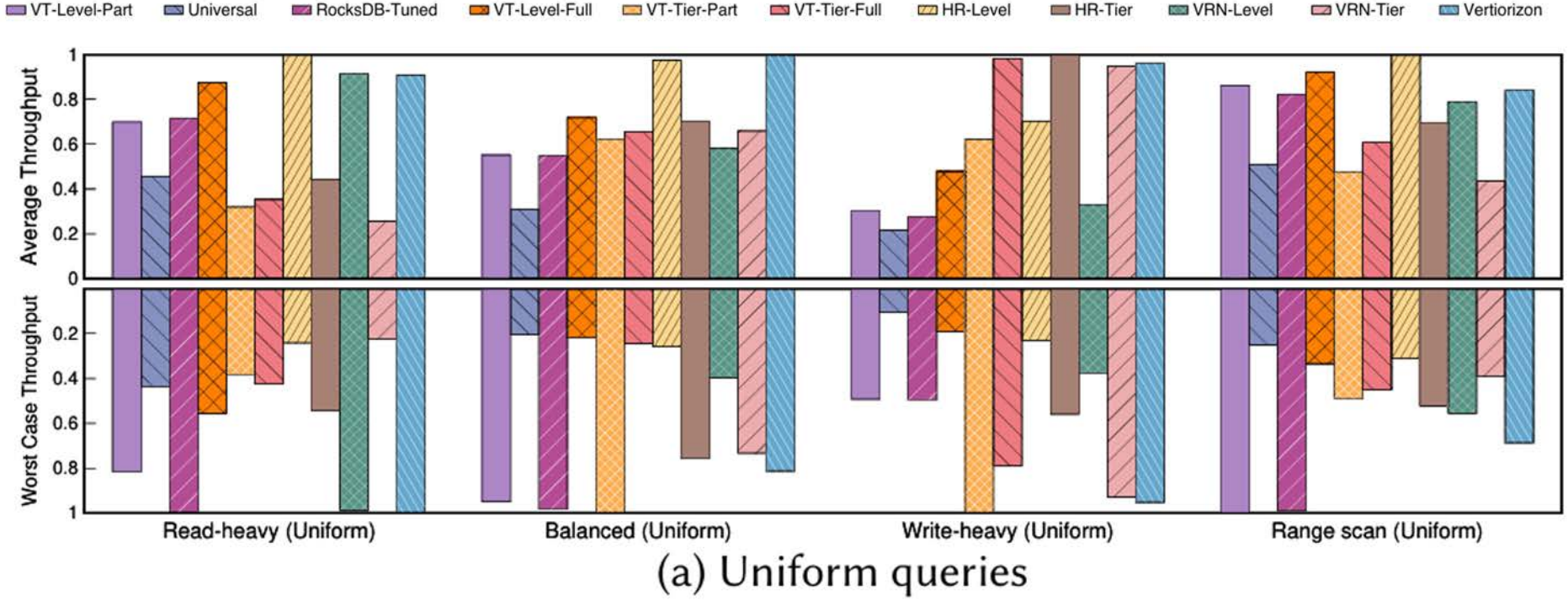
Initially bulk-load 10 million key-value entries of 1KB (YCSB uniform or zipfian)

Workload with 4 million operations (lookup or update)

Worst-case throughput measured as minimum among windows of 100k operations

Results: Uniform queries

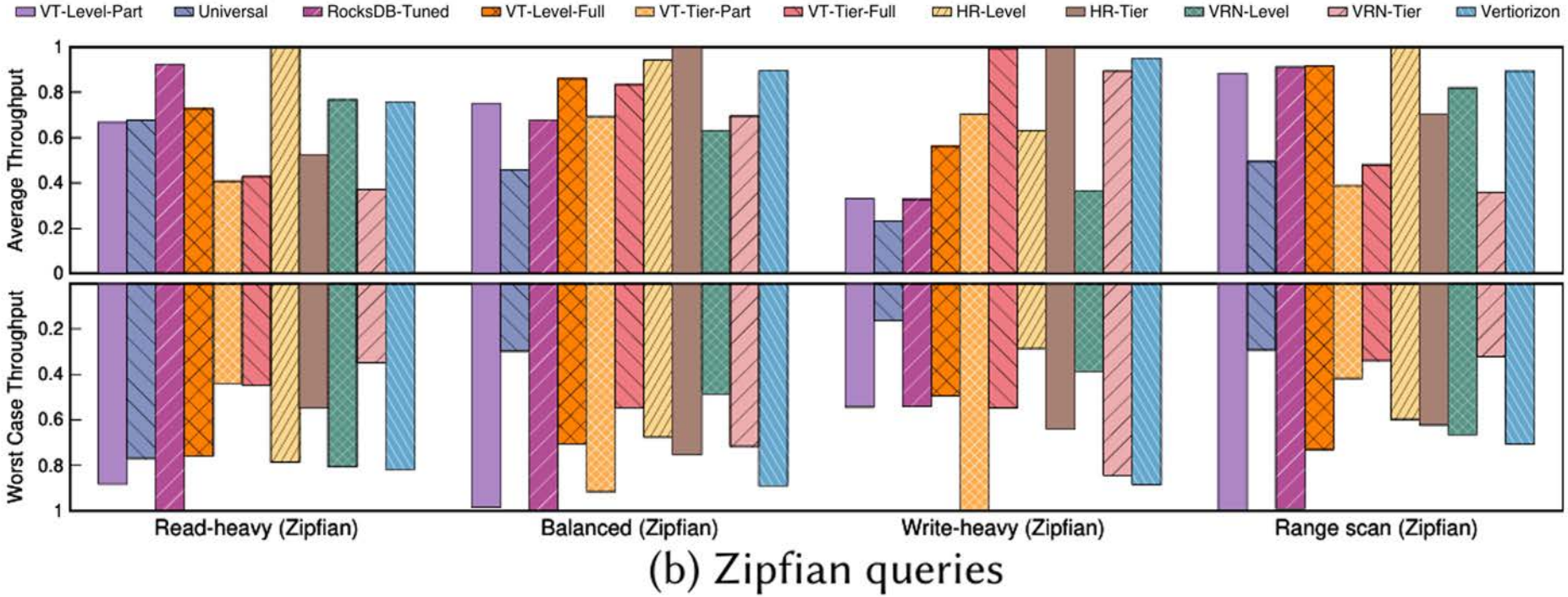
Best average ranking across all types of workloads



(a) Uniform queries

Results: Zipfian queries

Best average ranking across all types of workloads



(b) Zipfian queries

Large data scale experimental setting

Buffer size is increased to 64 MB

Total preloaded data is increased to 500 million key-value entries

Workload of 200 million operations

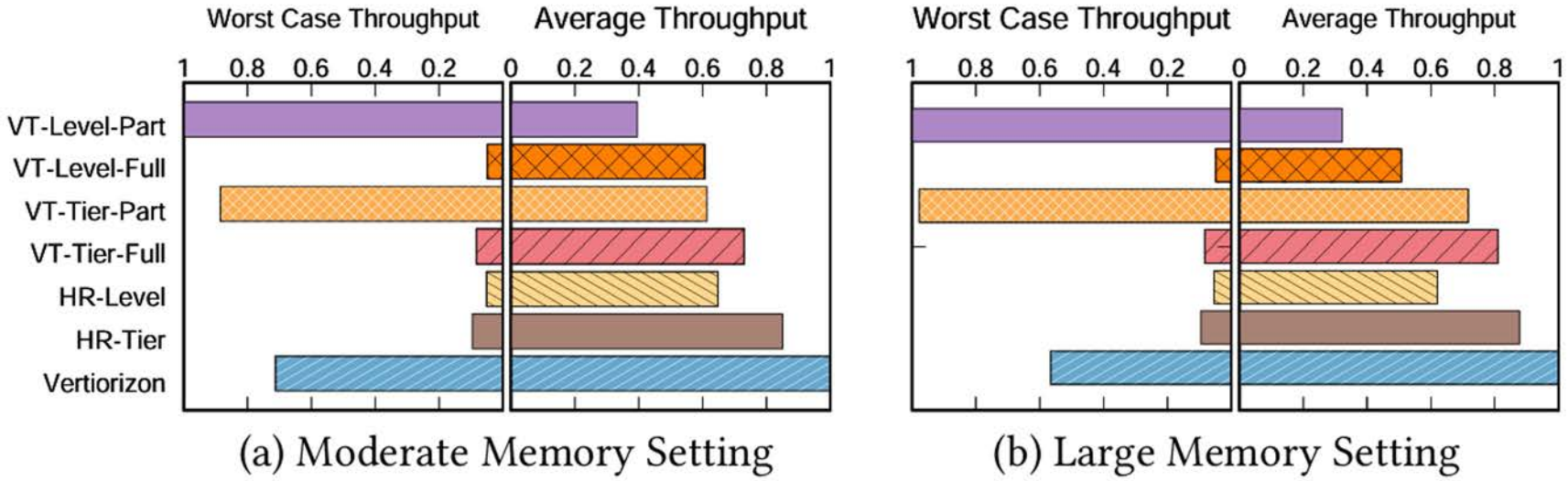
Read-write balanced workload with a uniform key distribution

Moderate memory configuration with 32MB block cache and 10 bits per key Bloom filter

Larger memory configuration with 64GB block cache and 20 bits per key Bloom filter

Results: Large data scale

Vertiorizon offers best average throughput for large data



Additional results

Vertiorizon keeps the best average rank under the following settings:

1) Increasing Bloom filter bits per key up to 20
2) Increasing the block cache size up to 64GB

Embedding Vertiorizon into any lazy-leveling design improves performance

Conclusion

Revisited horizontal and vertical LSM-tree growth schemes

Introduced horizontal tiering and Vertiorizon

Vertiorizon achieves robust performance across a wide range of scenarios and metrics

Q&A

Critics:
Andrew Briasco-Stewart

Proponents:

Liang Yu Lin

Ruoxi Cao

Q1: Do tombstones and snapshots affect the optimality claim of horizontal-tiering?

The paper's cost model focuses on point and range queries.

In practice, deletes with tombstones require extra compaction work to garbage-collect, and snapshots can block compaction from reclaiming stale versions.

Do these affect the optimality claim of horizontal-tiering, or are they orthogonal to the growth scheme analysis?

Q1: Do tombstones and snapshots affect the optimality claim of horizontal-tiering?

Largely orthogonal, with bounded practical impact

- Theorem 4.2 is proven over compaction timing, not data content — tombstones are just entries
- Both tombstones and snapshots affect horizontal and vertical schemes symmetrically
- Relative Pareto advantage of horizontal-tiering is preserved
- Acknowledged limitation: a cost model incorporating tombstone density and snapshot lifespan would be more complete → future work

Q2: Is partial compaction fundamentally incompatible with the horizontal scheme?

What exactly breaks when partial compaction is applied to a horizontal scheme?

Is the incompatibility fundamental, or is it mainly a modeling and design choice in this paper?

Q2: Is partial compaction fundamentally incompatible with the horizontal scheme?

Yes, fundamentally — not just a design choice

- Bentley-Saxe optimality assumes full level merges — partial compaction breaks this assumption
- Counter logic (Ci > Ci+1) loses accuracy when only a file subset is merged → data distribution becomes irregular
- Result: uncontrolled read amplification and space amplification
- Vertiorizon's response: confine full compaction to the small horizontal part, apply partial compaction only where the vertical structure can handle it correctly

Q3: Beyond Vertiorizon's hybrid design, what else can mitigate full compaction latency spikes?

Full compaction still has latency spikes, so it seems problematic for tail latency of a workload.

How problematic is this?

What else can be done?

Q3: Beyond Vertiorizon's hybrid design, what else can mitigate full compaction latency spikes?

Vertiorizon makes structural progress; complementary techniques exist

- Vertiorizon already reduces write stalls by factor ~T by confining full compaction to the horizontal part
- Remaining spike: full compaction from horizontal part → first vertical level
- Complementary approaches:
  - IO scheduling
  - Incremental full compaction: batch and interleave with normal writes
  - Proactive scheduling: anticipate capacity, compact during low-traffic windows
- All orthogonal to growth scheme → directly layerable on Vertiorizon

Q4: Does using RocksDB default settings favor or disadvantage Vertiorizon? How do you ensure fair comparison?

RocksDB exposes a very large configuration space. The paper holds most orthogonal parameters at default values.

Does this actually favor or disadvantage Vertiorizon relative to the baselines, and how should we think about fair comparison in systems with so many tuning knobs?

Q4: Does using RocksDB default settings favor or disadvantage Vertiorizon? How do we ensure fair comparison?

Defaults are principled; Vertiorizon's strength is robustness, not peak tuning

- Defaults follow RocksDB's developer-optimized general SSD settings
- RocksDB-Tuned baseline explicitly accounts for best-configured vertical scheme
- Vertiorizon exposes the same tuning knobs (T, ℓ, merge policy) as baselines — no asymmetry
- Self-tuning (Section 5.2) navigates configuration space automatically
- Key insight: a scheme that performs robustly without manual tuning is more valuable in practice
- Evidence: Vertiorizon ranks #1 overall across all workloads and metrics (Figure 7d) under uniform default settings

Why should users adopt this more complicated system? Is the gain really worth it?

In industry, the optimal solutions aren't always used if they are very complicated, hard to understand and implement, or hard to maintain.

Is this additional complexity really necessary, or can similar performance be achieved with a simpler design?

Why should users adopt this more complicated system? Is the gain really worth it?

1. It is modular architecture

(Makes the design easier to understand and integrate)

1. Complexity is Necessary Trade-off

(Static designs cannot achieve optimal performance)

Does this design introduce significant tuning overhead?

How much additional work does a database admin need to do if they use the design described in the paper?

Modern databases already have a significant number of tuning knobs that an admin must contend with, will this design make the problem worse?

Cost Model: $(\zeta = wW + rR + qQ)$

Does this design introduce significant tuning overhead?

1. Benefits Outweigh the Cost
   (Improved read/write balance)

1. Adaptive Tuning is Beneficial
   (Real-world workloads are dynamic)

Where do you see this line of inquiry going? Are there other improvement that can be made?

What next steps do you see?

How can the system potentially be further improved to better adapt to changing workloads and real-world system conditions?

Where do you see this line of inquiry going? Is there another improvement that can be made?

1. Make the system more adaptive
    - Move toward fully self-tuning LSM systems
    - Dynamically adjust growth scheme based on workload (read-heavy vs write-heavy)

2. Better integration with real-world systems
    - Incorporate hardware characteristics (e.g., SSD/ZNS behavior)
    - Consider concurrency, caching, buffer management