Towards Accurate and Fast Evaluation of Multi-Stage Log-Structured Designs

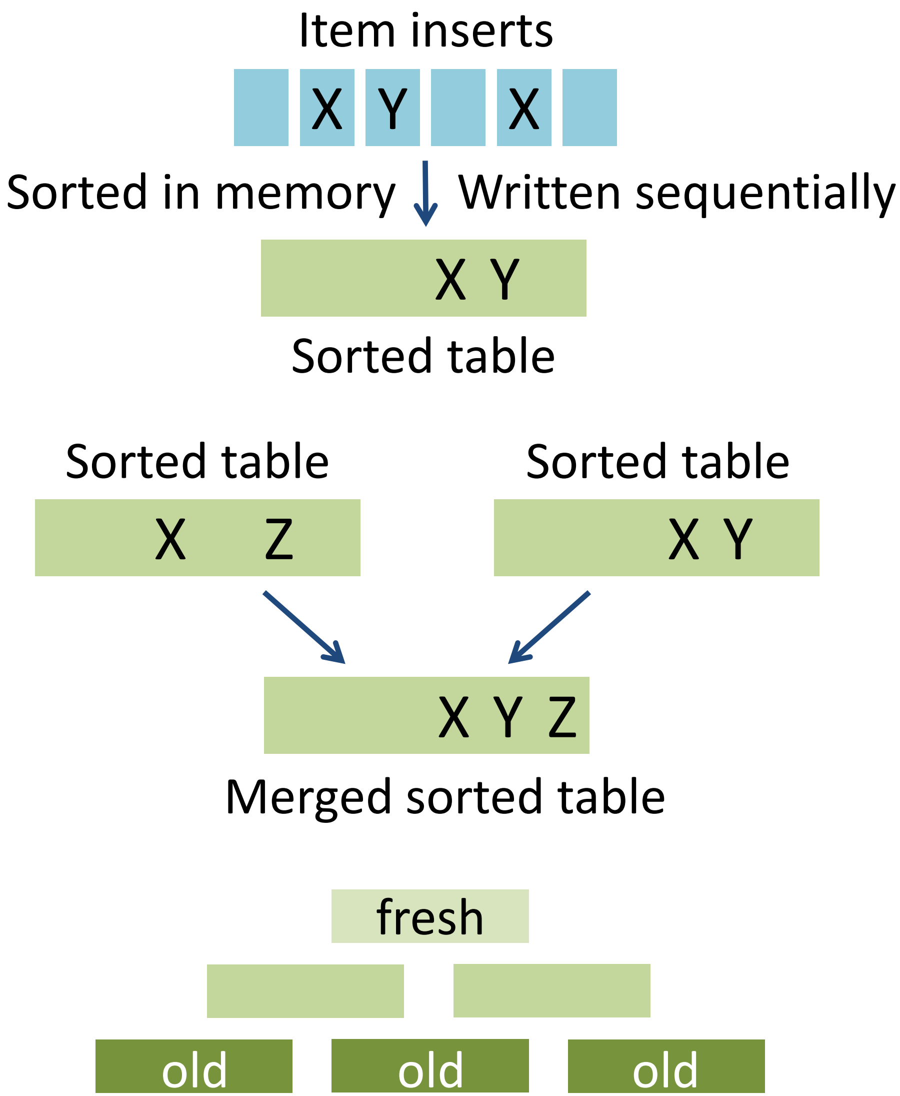
Hyeontaek Lim

David G. Andersen, Michael Kaminsky[†]

Carnegie Mellon University

[†]Intel Labs

Multi-Stage Log-Structured ("MSLS") Designs



Example: LevelDB, RocksDB, Cassandra, HBase, …

(Naïve) Log-structured design

- Fast writes with sequential I/O
- Slow query speed
- Large space use

Compaction

- Fewer table count
- Less space use
- Heavy I/O required

Multi-stage design

- Cheaper compaction by segregating fresh and old data

MSLS Design Evaluation Needed

Mobile app

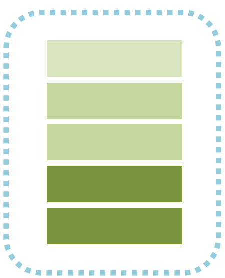
Filesystem

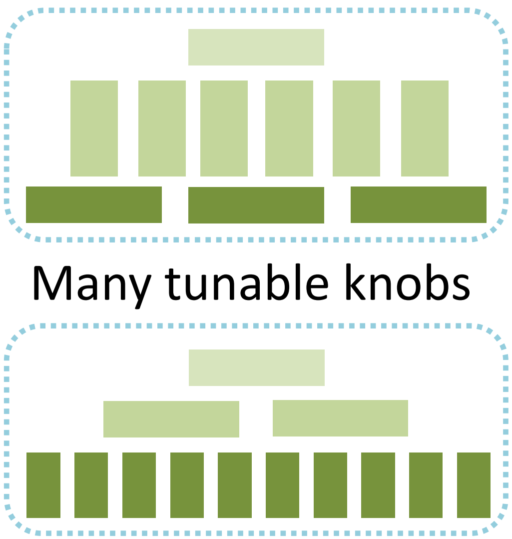
Desktop app

Data-intensive computing

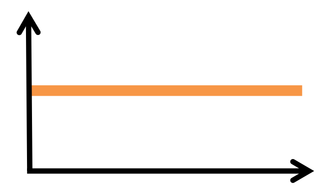
Problem: How to evaluate and tune MSLS designs for a workload?

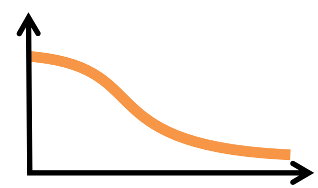
Large design space





Many tunable knobs

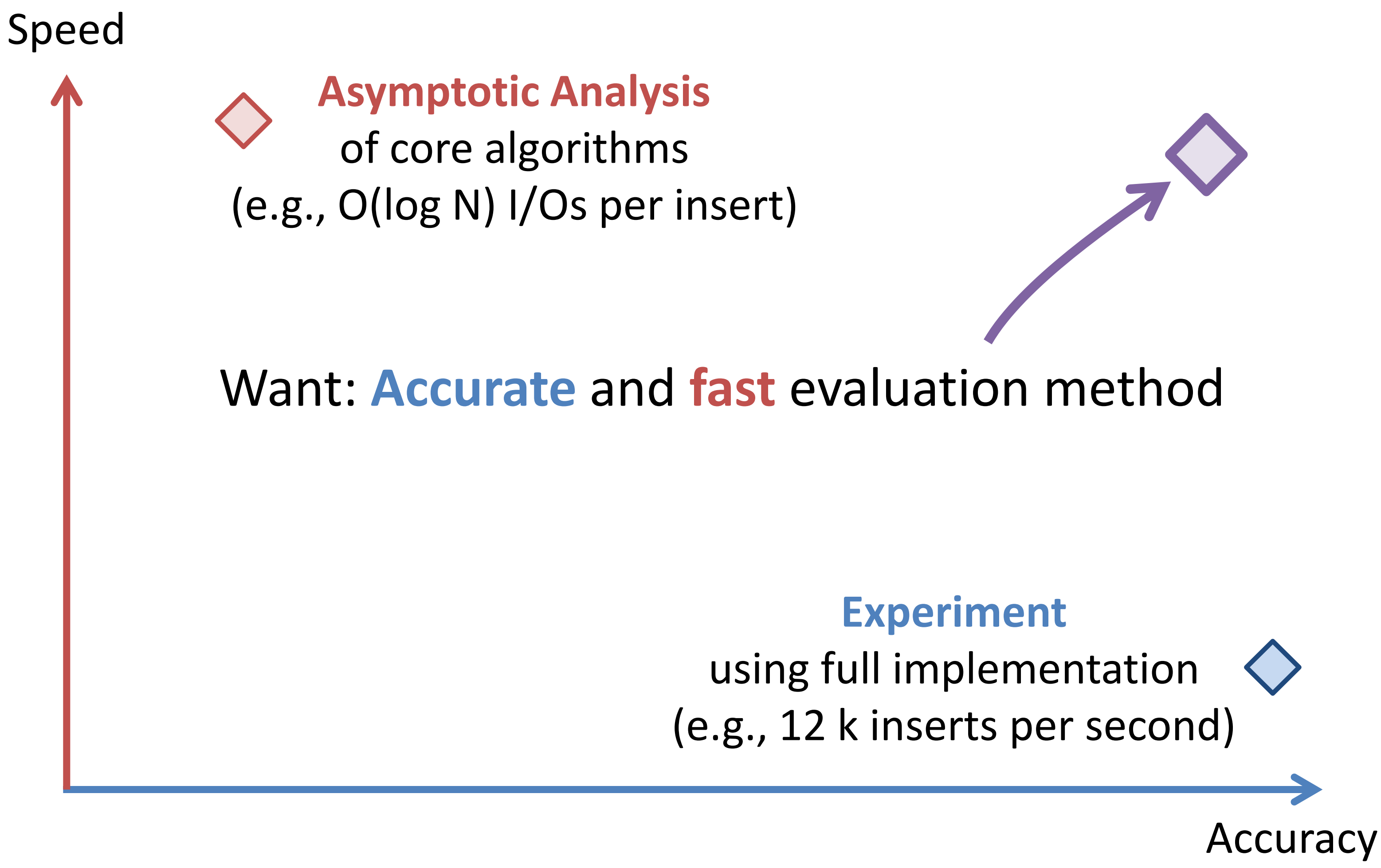




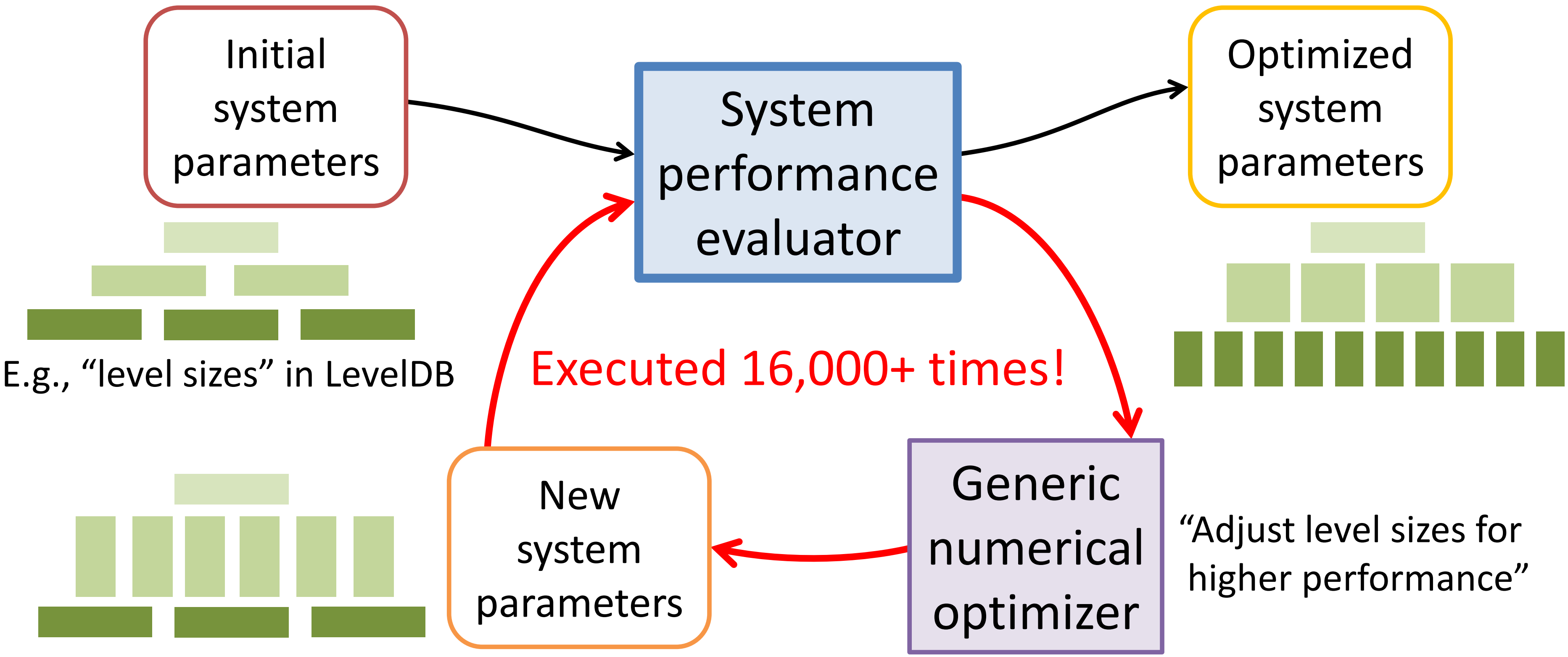
Diverse workloads

Diverse workloads

Two Extremes of Prior MSLS Evaluation



What You Can Do With Accurate and Fast Evaluation



Our level size optimization on LevelDB

- Up to 26.2% lower per-insert cost, w/o sacrificing query performance
- Finishes in 2 minutes (full experiment would take years)

Accurate and Fast Evaluation of MSLS Designs

Analytically model multi-stage log-structured designs using **new analytic primitives** that consider redundancy

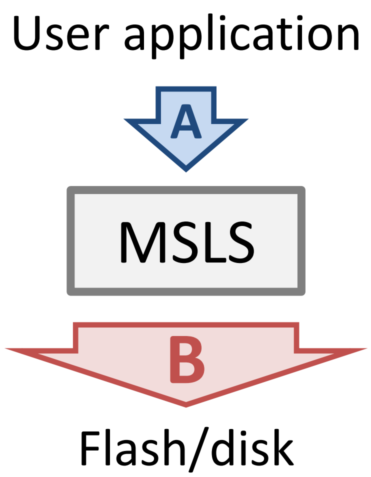
Accuracy: Only ≤ 3–6.5% off from LevelDB/RocksDB experiment

Speed: < 5 ms per run for a workload with 100 M unique keys

Performance Metric to Use

Focus of this talk: **Insert performance** of MSLS designs

- Often bottlenecked by writes to flash/disk
- Need to model amortized write I/O of inserts

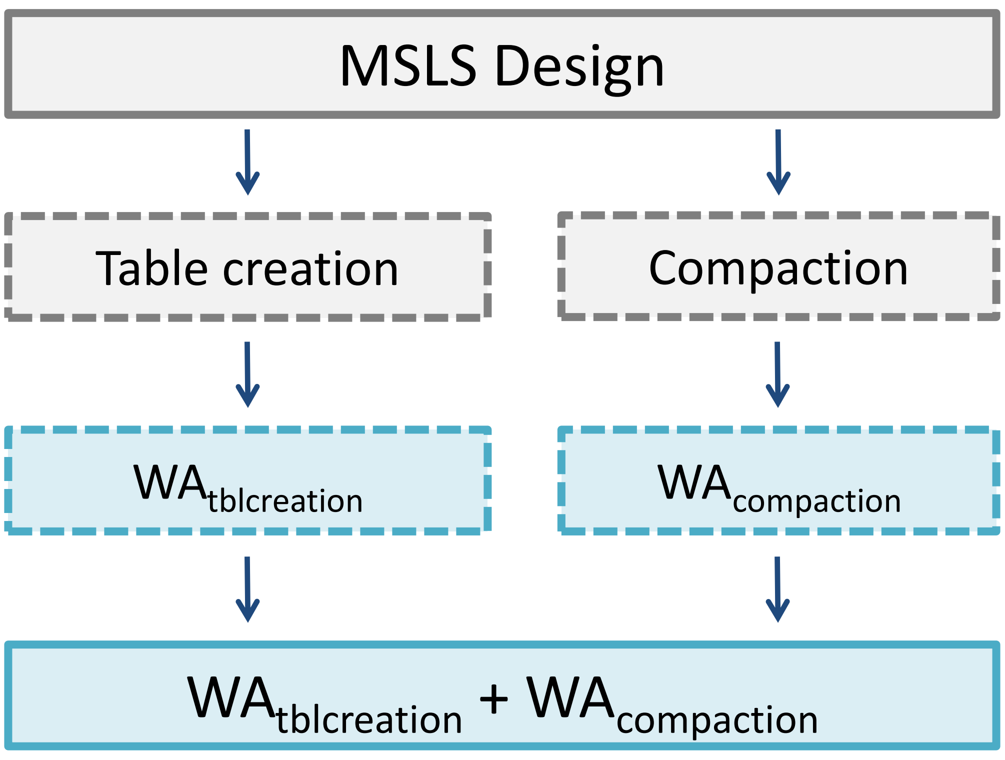


(Application-level) **Write amplification**

$$= \frac{\text{Size of data written to flash/disk (B)}}{\text{Size of inserted data (A)}}$$

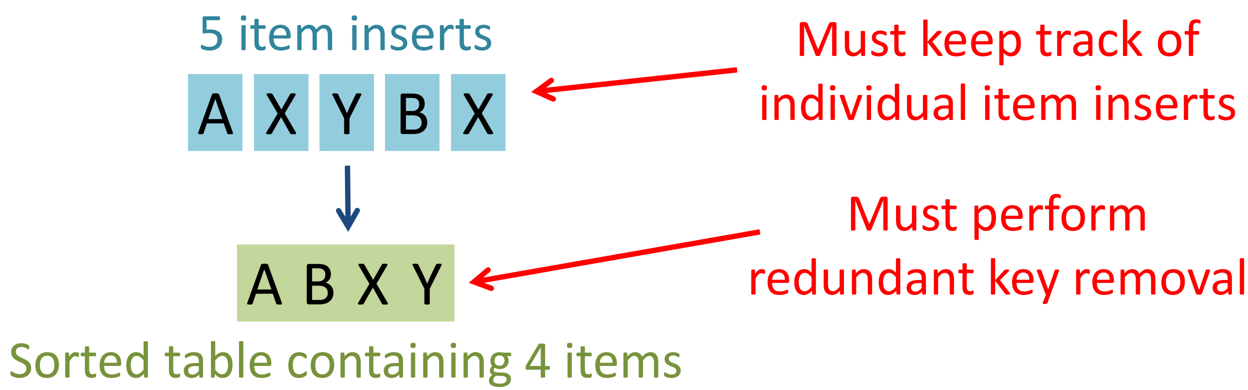
- Easier to analyze than raw throughput
- Closely related to raw throughput: write amplification ∝ 1/throughput

Divide-and-Conquer to Model MSLS Design



1. Break down MSLS design into small components
2. Model individual components' write amplification
3. Add all components' write amplification

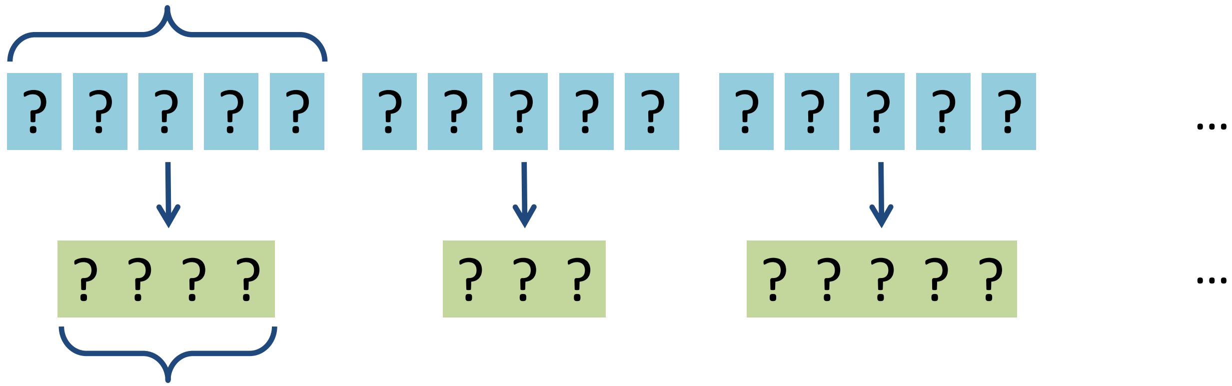
Modeling Cost of Table Creation: Strawman



Write amplification of this table creation event = $\frac{4}{5}$

Modeling Cost of Table Creation: Better Way

bufsize (max # of inserts buffered in memory)

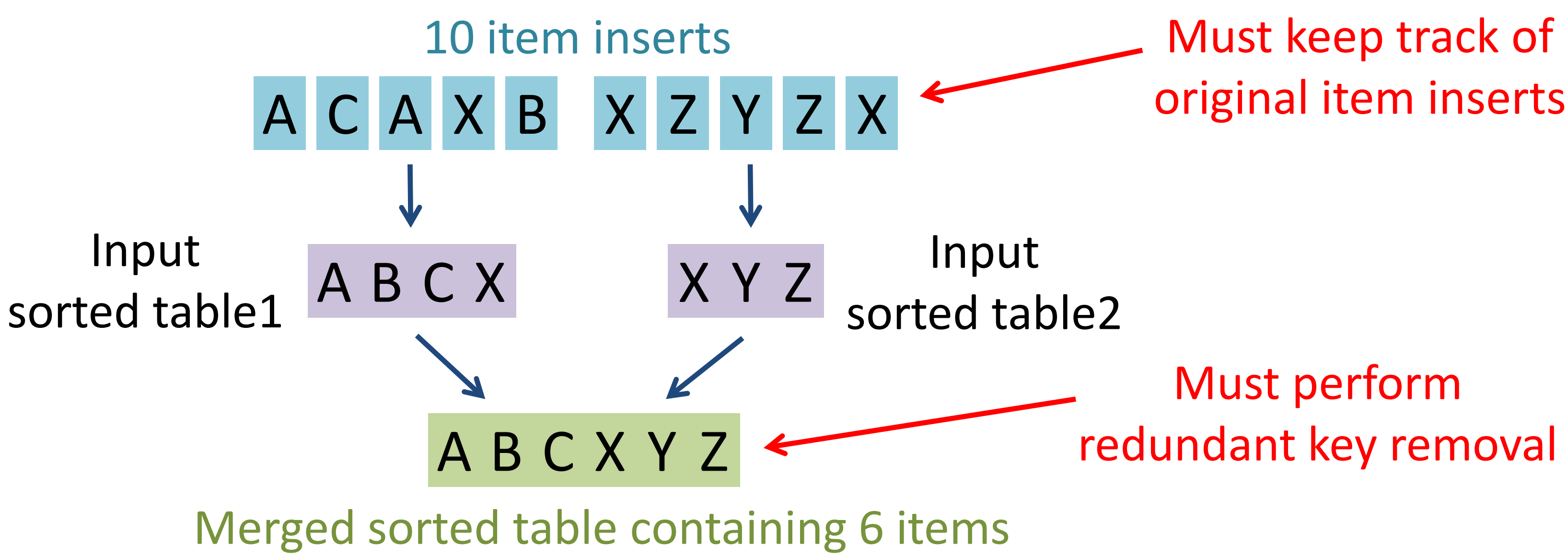


Unique(bufsize): expected # of **unique** keys in bufsize requests

Write amplification of regular table creation = $\frac{\textbf{Unique}(\text{bufsize})}{\text{bufsize}}$

✓ No item-level information required
✓ Estimates general operation cost

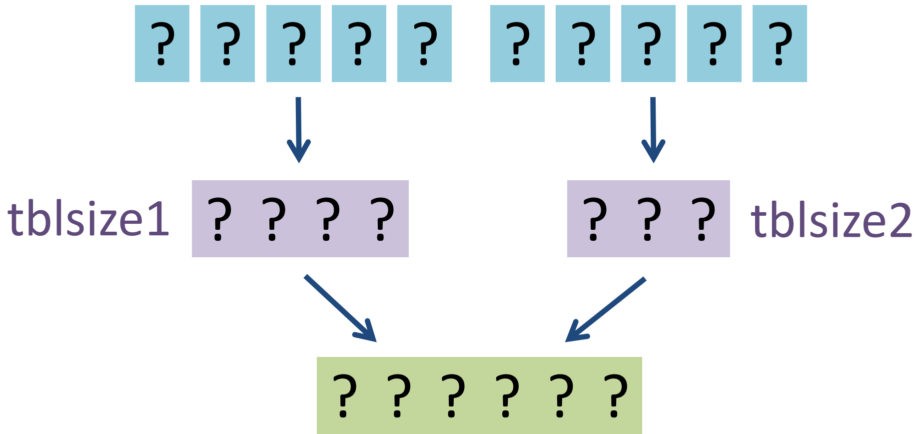
Modeling Cost of Compaction: Strawman



Write amplification of this compaction event = $\frac{6}{10}$

Modeling Cost of Compaction: Better Way

Unique^{-1}(tblsize1) **Unique**$^{-1}$(tblsize2): expected # of requests containing tblsize2 unique keys
i.e., **Unique**(**Unique**$^{-1}$(tblsize2))
= tblsize2



Merge(tblsize1, tblsize2): expected # of unique keys in input tables whose sizes are tblsize1 and tblsize2

$$\text{Write amplification of 2-way compaction} = \frac{\textbf{Merge}(\text{tblsize1}, \text{tblsize2})}{\textbf{Unique}^{-1}(\text{tblsize1}) + \textbf{Unique}^{-1}(\text{tblsize2})}$$

✓ No item-level information required
✓ Estimates general operation cost

New Analytic Primitives Capturing Redundancy

Unique: [# of requests] → [# of unique keys]

Unique^{-1}: [# of requests] ← [# of unique keys]

Merge: [multiple # of unique keys] → [total # of unique keys]

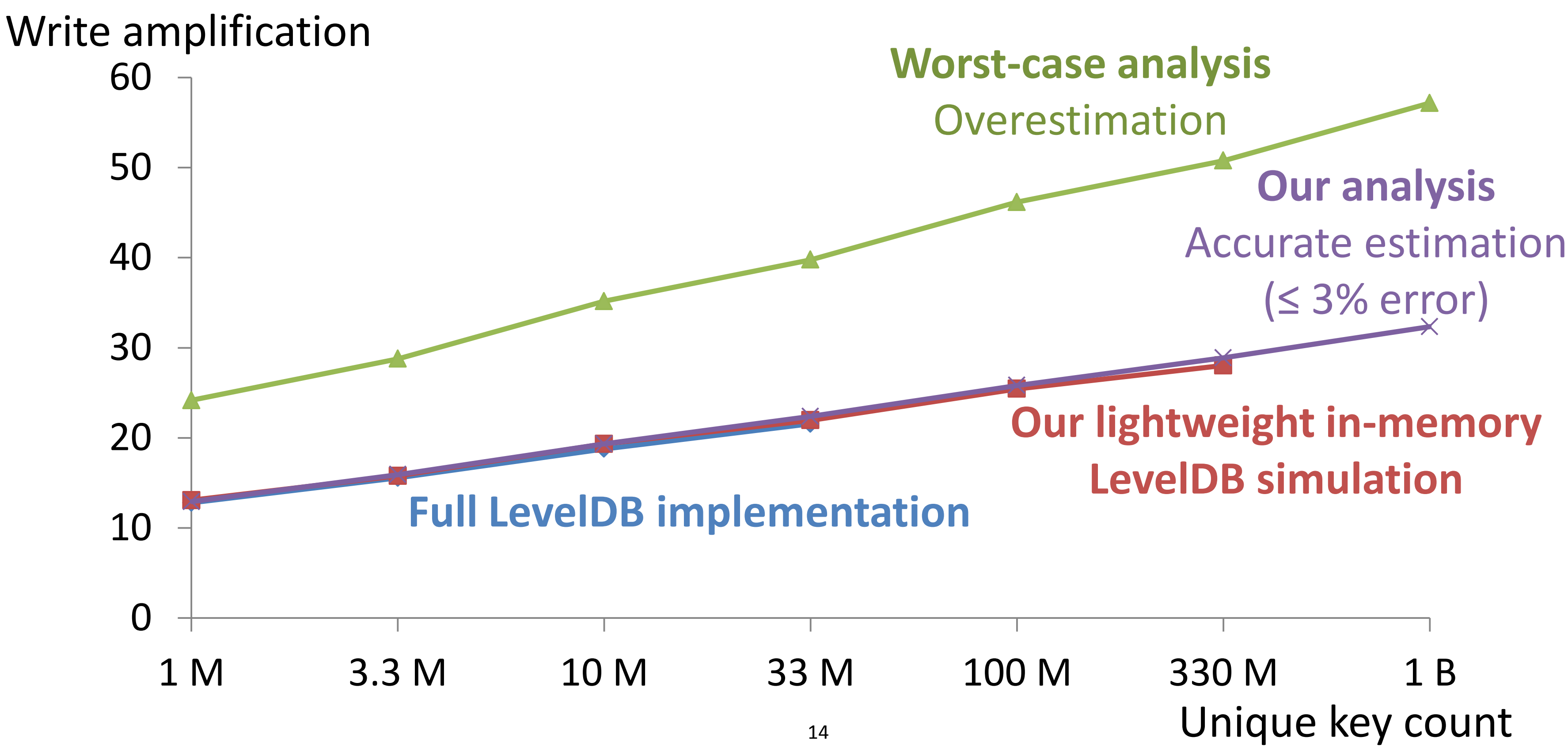
- **Fast** to compute (see paper for mathematical descriptions)
- Consider **redundancy**: Unique(p) ≤ p Merge(u, v) ≤ u + v
- Reflect **workload skew**: [Unique(p) for Zipf] ≤ [Unique(p) for uniform]

- Caveat: Assume no or little dependence between requests

High Accuracy of Our Evaluation Method

Compare measured/estimated write amplification of insert requests on LevelDB

- Key-value item size: 1,000 bytes
- Unique key count: 1 million–1 billion (1 GB–1 TB)
- Key popularity dist.: Uniform



High Speed of Our Evaluation Method

Compare **single-run** time to obtain write amplification of insert requests for a **specific** workload using a **single** set of system parameters

- LevelDB implementation: fsync disabled
- LevelDB simulation: in-memory, optimized for insert processing

Method	Workload size (# of unique keys)	Elapsed time
Experiment using LevelDB implementation	10 M	101 minutes
Experiment using LevelDB simulation	100 M	45 minutes
Our analysis	**100 M**	**< 5 ms**

Summary

- Evaluation method for multi-stage log-structured designs
  - **New analytic primitives** that consider redundancy
  - **System models** using new analytic primitives
- Accurate and fast
  - Only ≤ 3–6.5% error in estimating insert cost of LevelDB/RocksDB
  - Several orders of magnitude faster than experiment
- Example applications
  - Automatic system optimization (~26.2% faster inserts on LevelDB)
  - Design improvement (~32.0% faster inserts on RocksDB)
- Code: github.com/efficient/msls-eval

Backup Slides

Nature of MSLS Operations

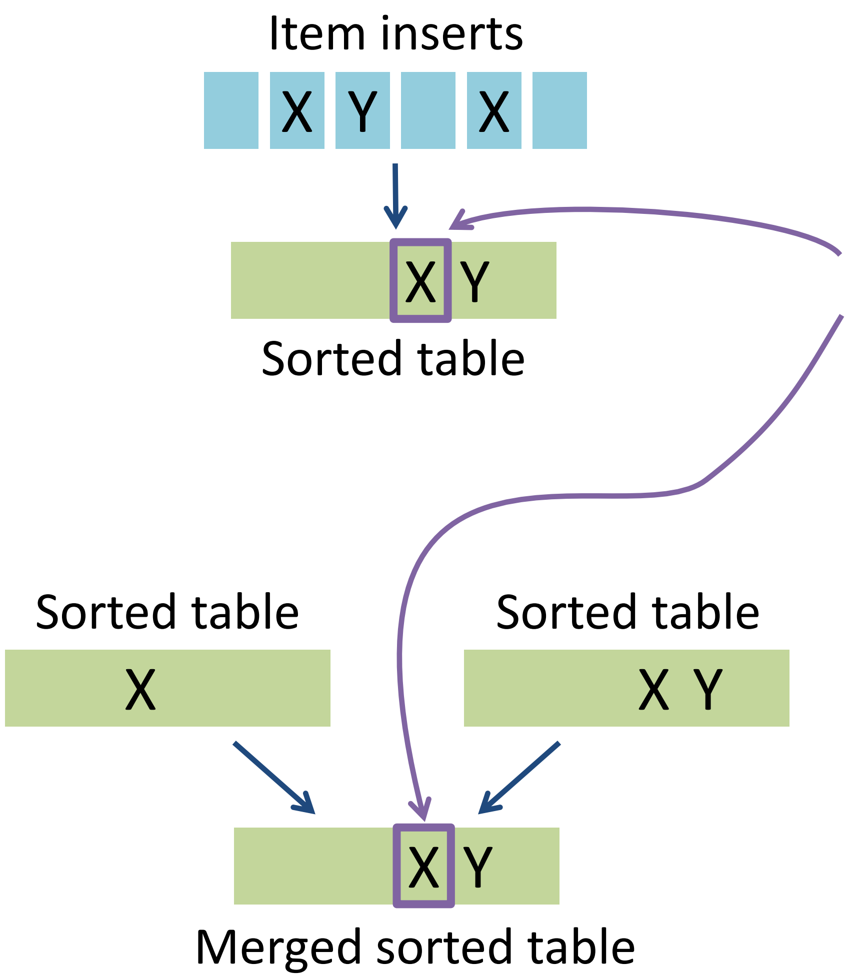


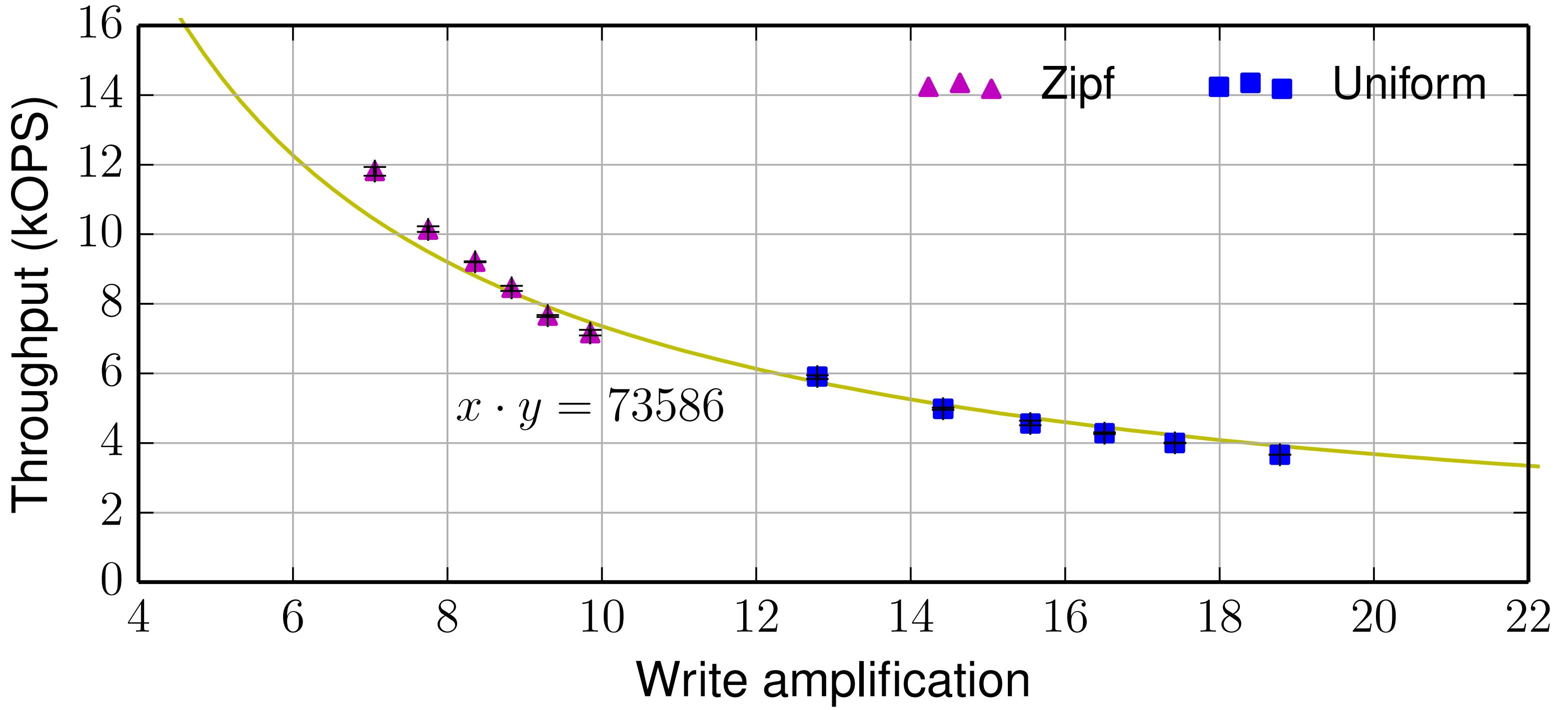
Table creation and compaction: essentially redundancy removal

➪ Modeling operation cost requires considering redundancy

Write Amplification vs. Throughput

Compare measured write amplification/throughput of insert requests on LevelDB

- Key-value item size: 1,000 bytes
- Unique key count: 1 million–10 million (1 GB–10 GB)
- Key popularity dist.: Uniform, Zipf (skew=0.99)



Mathematical Description of New Primitives

Unique: [# of requests] → [# of unique keys]

Unique^{-1}: [# of requests] ← [# of unique keys]

Merge: [multiple # of unique keys] → [total # of unique keys]
Merge(u, v) = Unique(Unique^{-1}(u) + Unique^{-1}(v))

$$\text{Unique}(p) \coloneqq N - \sum_{k \in K} \left(1 - f_X(k)\right)^p$$

Total # of unique keys ($|K|$)

# of requests

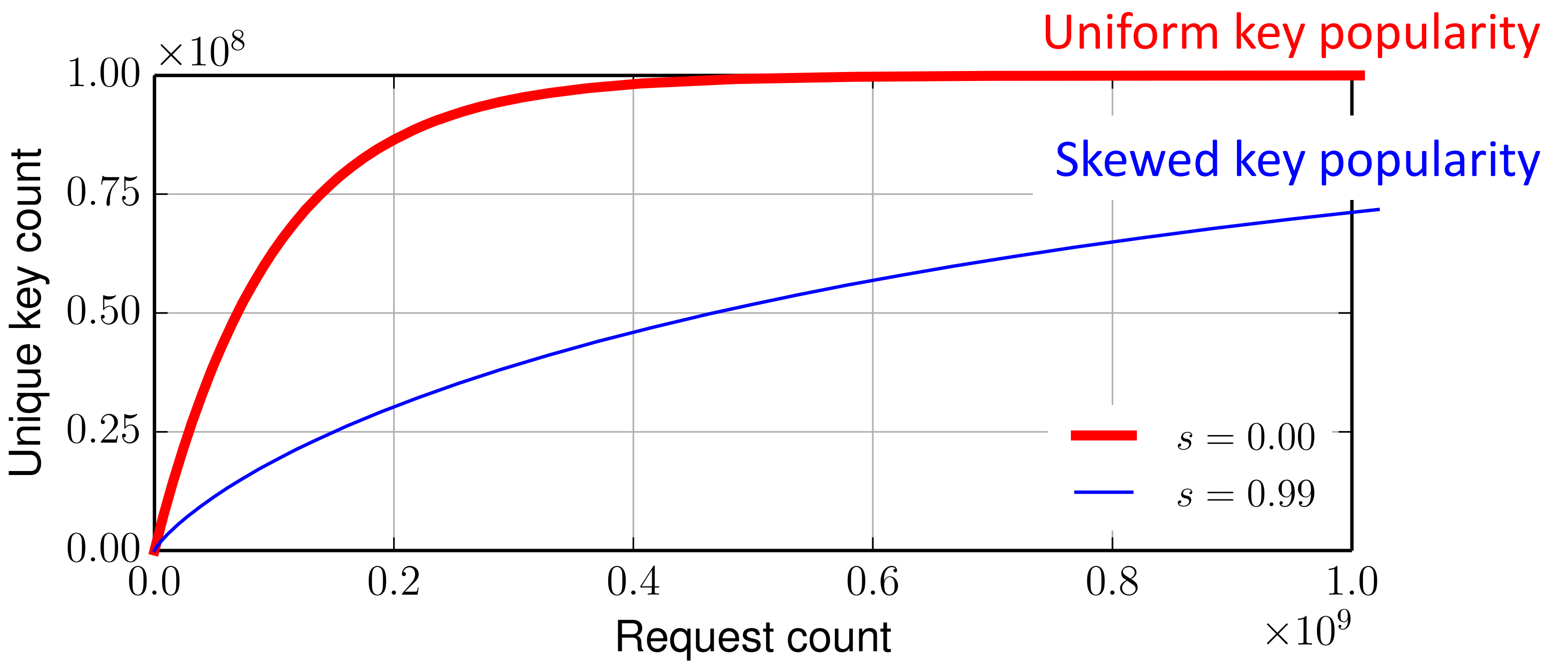
Probability of key k in each request for a **key popularity distribution**

Set of unique keys

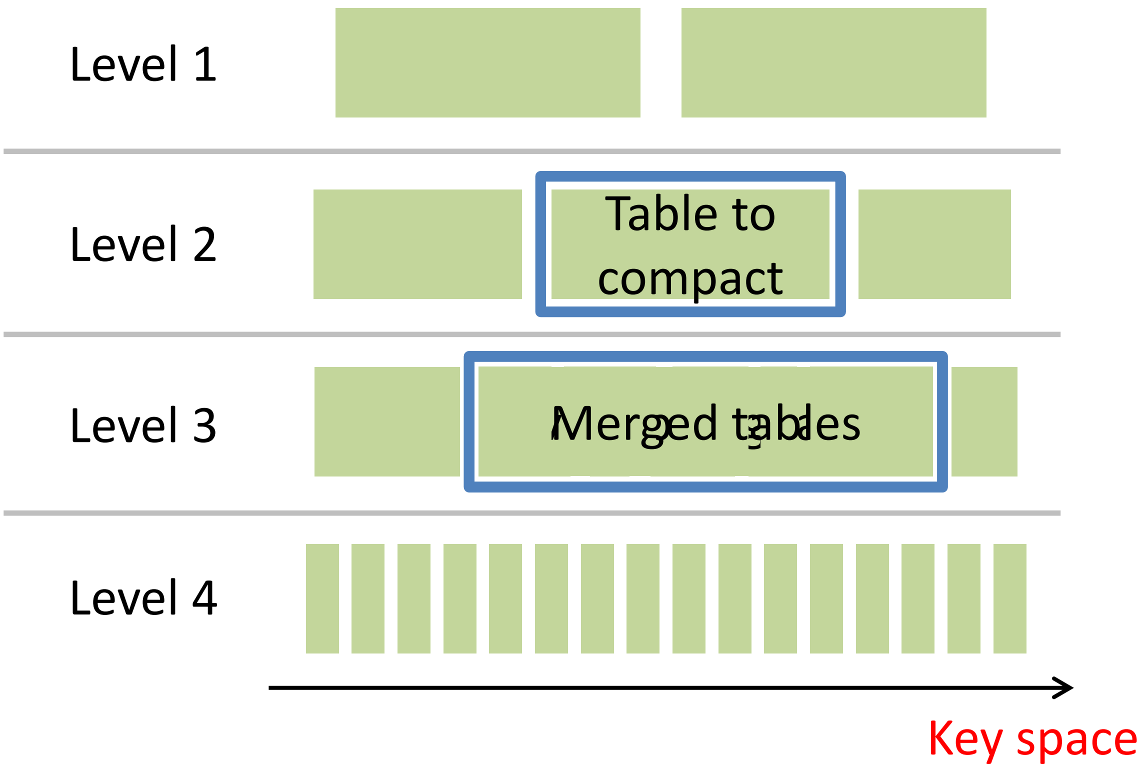
Unique as a Function of Request Count

Compare measured write amplification/throughput of insert requests on LevelDB

- Key-value item size: 1,000 bytes
- Unique key count: 100 M (100 GB)
- Request count: 0–1 billion
- Key popularity dist.: Uniform, Zipf (skew=0.99)



LevelDB Design Overview



Each level's total size
= ~**10**X previous level's

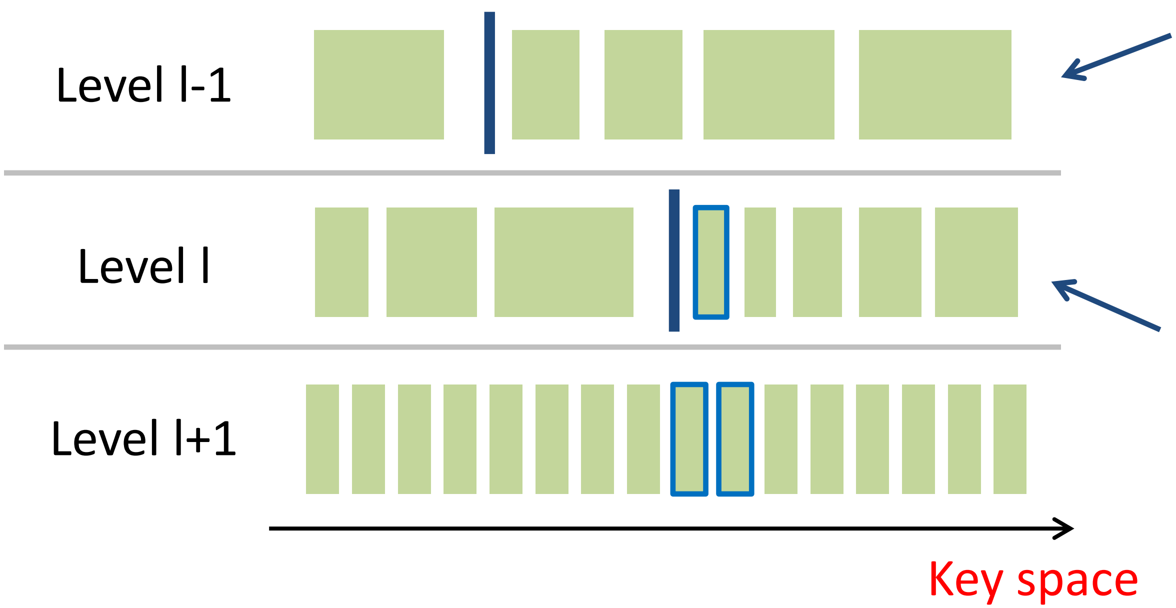
Each level are partitioned
into small tables (~2 MB)
for incremental compaction

Q: Average # of overlaps?
➪ Less than **10**!
("non-uniformity")

(Omitted: memtable, write-ahead log, level 0)

Non-Uniformity in LevelDB

Direction of compaction in key space (round-robin way)



Fast to sweep small level
- ➪ Add new data to next level uniformly across key space

Slow to sweep large level
- ➪ Soon-to-be-compacted region becomes dense, causing non-uniformity
- ➪ Fewer overlapping tables in next level

(Omitted: memtable, write-ahead log, level 0)

Pseudo Code of LevelDB Model

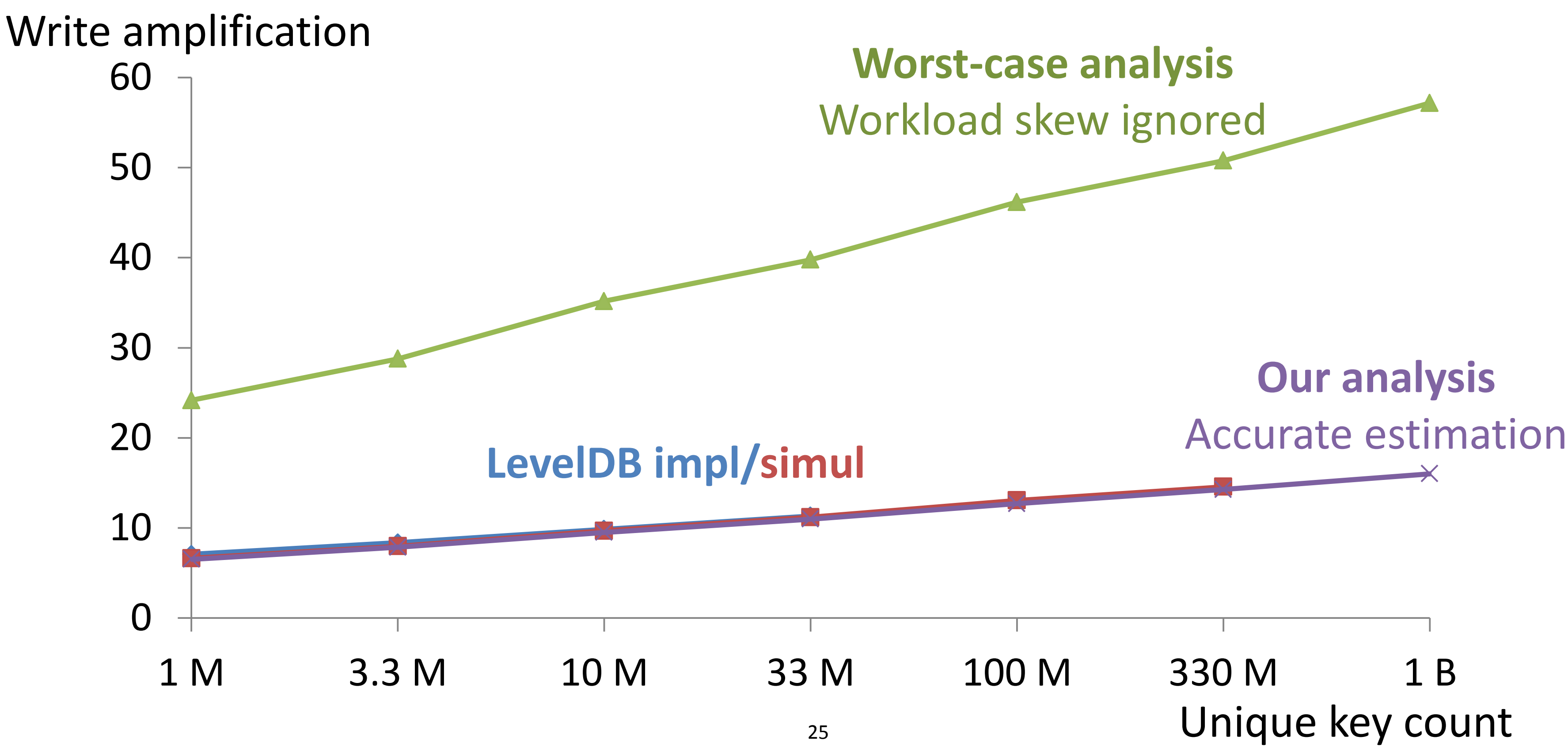
```
// @param  L     maximum level
// @param  wal   write-ahead log file size
// @param  c0    level-0 SSTable count
// @param  size  level sizes
// @return       write amplification (per-insert cost)
function estimateWA_LevelDB(L, wal, c0, size[]) {
  local l, WA, interval[], write[];

  // mem -> log
  WA = 1;

  // mem -> level-0
  WA += unique(wal) / wal;

  // level-0 -> level-1
  interval[0] = wal * c0;
  write[1] = merge(unique(interval[0]), size[1]);
  WA += write[1] / interval[0];

  // level-l -> level-(l+1)
  for (l = 1; l < L; l++) {
    interval[l] = interval[l-1] + dinterval(size, l);
    write[l+1] = merge(unique(interval[l]), size[l+1]) + unique(interval[l]);
    WA += write[l+1] / interval[l];
  }

  return WA;
}
```

LevelDB-specific function
to take into account “non-uniformity”

Sensitivity to Workload Skew

Compare measured/estimated write amplification of insert requests on LevelDB

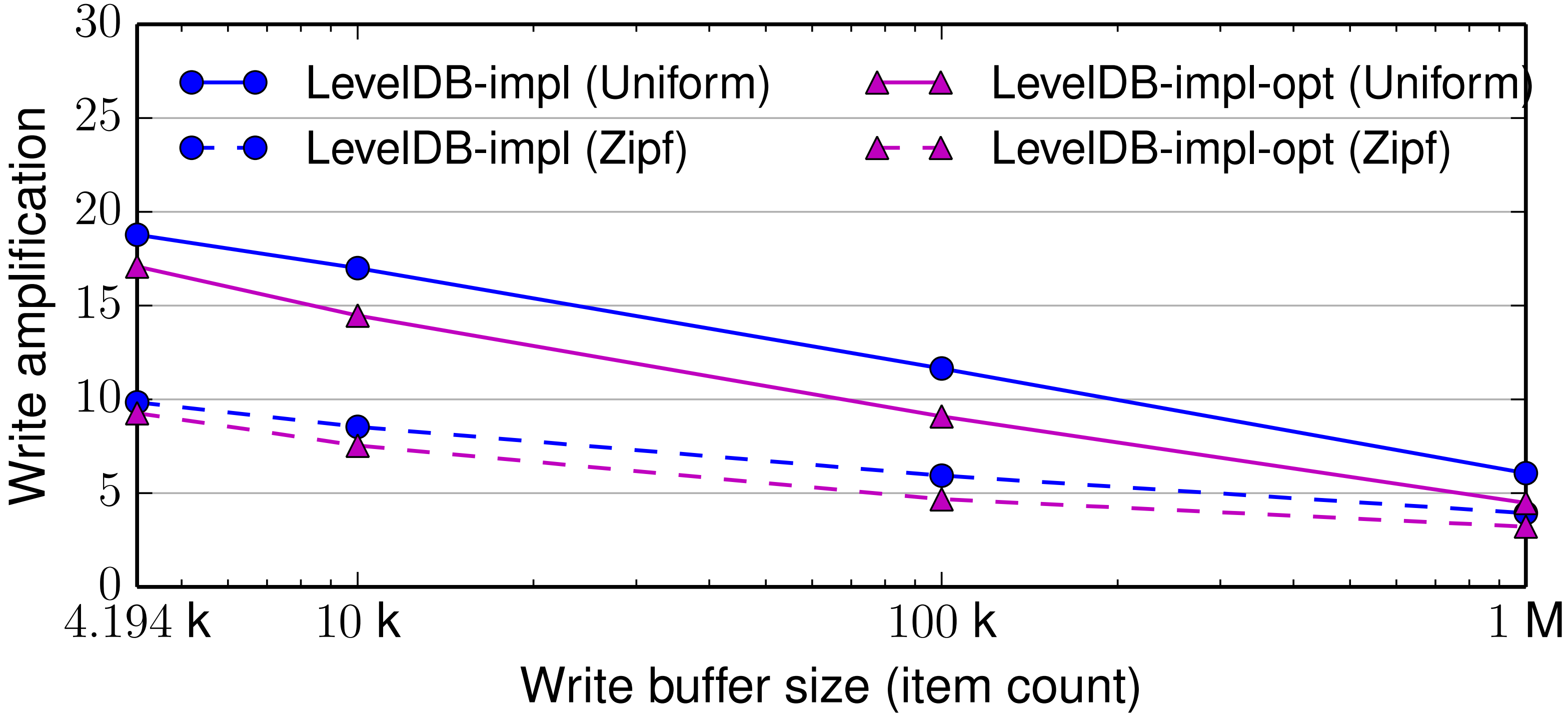
- Key-value item size: 1,000 bytes
- Unique key count: 1 million–1 billion (1 GB–1 TB)
- **Key popularity dist.: Zipf (skew=0.99)**



Automatic System Optimization Result

Compare measured/estimated write amplification of insert requests on LevelDB

- Key-value item size: 1,000 bytes
- Write buffer size: 4 MiB–[10% of total unique key count]
- Unique key count: 10 million (10 GB)
- Key popularity dist.: Uniform, Zipf (skew=0.99)



End of Slides