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

(_naive_) 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

Example: LevelDB, RocksDB, Cassandra, HBase, ...
Problem: How to evaluate and tune MSLS designs for a workload?
Two Extremes of Prior MSLS Evaluation

- **Asymptotic Analysis** of core algorithms (e.g., \(O(\log N)\) I/Os per insert)

- **Experiment** using full implementation (e.g., 12 k inserts per second)

Want: **Accurate** and **fast** evaluation method
What You Can Do With Accurate and Fast Evaluation

- **Initial system parameters**
- **System performance evaluator**
- **Optimized system parameters**
- **New system parameters**
- **Generic numerical optimizer**

E.g., “level sizes” in LevelDB

**Executed 16,000+ times!**

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 $\leq 3\text{--}6.5\%$ off from LevelDB/RocksDB experiment

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

**Write amplification**

\[
\text{Write amplification} = \frac{\text{Size of data written to flash/disk (B)}}{\text{Size of inserted data (A)}}
\]

- Easier to analyze than raw throughput
- Closely related to raw throughput: write amplification \(\propto 1/\text{throughput}\)
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

5 item inserts

\[
\begin{array}{c}
A \\
X \\
Y \\
B \\
X \\
\end{array}
\]

\[
\begin{array}{c}
A \\
B \\
X \\
Y \\
\end{array}
\]

Sorted table containing 4 items

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 = \[
\frac{\text{Unique}(\text{bufsize})}{\text{bufsize}}
\]

✓ No item-level information required
✓ Estimates general operation cost
Modeling Cost of Compaction: Strawman

10 item inserts

Must keep track of original item inserts

Must perform redundant key removal

Write amplification of this compaction event = \frac{6}{10}
**Modeling Cost of Compaction: Better Way**

\[
\text{Unique}^{-1}(\text{tblsize1}) \quad \text{Unique}^{-1}(\text{tblsize2}): \text{expected # of requests containing tblsize2 unique keys i.e., } \text{Unique}(\text{Unique}^{-1}(\text{tblsize2})) = \text{tblsize2}
\]

\[
\text{Merge}(\text{tblsize1}, \text{tblsize2}): \text{expected # of unique keys in input tables whose sizes are tblsize1 and tblsize2}
\]

Write amplification of 2-way compaction =

\[
\frac{\text{Merge}(\text{tblsize1}, \text{tblsize2})}{\text{Unique}^{-1}(\text{tblsize1}) + \text{Unique}^{-1}(\text{tblsize2})}
\]

- ✓ No item-level information required
- ✓ Estimates general operation cost
New Analytic Primitives Capturing Redundancy

**Unique**: [# of requests] $\rightarrow$ [# of unique keys]

**Unique$^{-1}$**: [# of requests] $\leftarrow$ [# of unique keys]

**Merge**: [multiple # of unique keys] $\rightarrow$ [total # of unique keys]

- **Fast** to compute (see paper for mathematical descriptions)
- Consider **redundancy**: $\text{Unique}(p) \leq p$ \hspace{1cm} $\text{Merge}(u, v) \leq u + v$
- Reflect **workload skew**: $[\text{Unique}(p) \text{ for Zipf}] \leq [\text{Unique}(p) \text{ for uniform}]$
- **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

Write amplification

- Worst-case analysis: Overestimation
- Our analysis: Accurate estimation (≤ 3% error)
- Our lightweight in-memory LevelDB simulation

Graph showing write amplification against unique key count.
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

<table>
<thead>
<tr>
<th>Method</th>
<th>Workload size (# of unique keys)</th>
<th>Elapsed time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Experiment using LevelDB implementation</td>
<td>10 M</td>
<td>101 minutes</td>
</tr>
<tr>
<td>Experiment using LevelDB simulation</td>
<td>100 M</td>
<td>45 minutes</td>
</tr>
<tr>
<td><strong>Our analysis</strong></td>
<td>100 M</td>
<td>&lt; 5 ms</td>
</tr>
</tbody>
</table>
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 \(\leq 3–6.5\%\) 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](https://github.com/efficient/msls-eval)
**Nature of MSLS Operations**

Only one instance survives for each key.

Table creation and compaction: essentially **redundancy removal**

⇒ Modeling operation cost requires considering redundancy.
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]

\[
\text{Merge}(u, v) = \text{Unique}(\text{Unique}^{-1}(u) + \text{Unique}^{-1}(v))
\]

**Unique**\(p\) := \(N - \sum_{k \in K} \left(1 - f_X(k)\right)^p\)

- Total # of unique keys (|\(K|\))
- Probability of key \(k\) in each request for a key popularity distribution
- # of requests
- Set of unique keys

---

Probability of key \(k\) in each request for a **key popularity distribution**
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 = ~10X previous level’s

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

Q: Average # of overlaps?
⇒ Less than 10!
(“non-uniformity”)
Non-Uniformity in LevelDB

Just compacted | Soon to be compacted

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

Level l-1

Level l

Level l+1

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

Key space ➔ Fewer overlapping tables in next level

(Omitted: memtable, write-ahead log, level 0)
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-1 -> 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;
}
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)

Write amplification

Worst-case analysis
Workload skew ignored

LevelDB impl/simul

Our analysis
Accurate estimation

Unique key count

0 10 20 30 40 50 60

1 M 3.3 M 10 M 33 M 100 M 330 M 1 B
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