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

Hyeontaek Lim and David G. Andersen, *Carnegie Mellon University;*
Michael Kaminsky, *Intel Labs*

https://www.usenix.org/conference/fast16/technical-sessions/presentation/lim

This paper is included in the Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16).
February 22–25, 2016 • Santa Clara, CA, USA
ISBN 978-1-931971-28-7

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

Hyeontaek Lim,[1] David G. Andersen,[1] Michael Kaminsky[2]
[1]*Carnegie Mellon University,* [2]*Intel Labs*

Abstract

Multi-stage log-structured (MSLS) designs, such as LevelDB, RocksDB, HBase, and Cassandra, are a family of storage system designs that exploit the high sequential write speeds of hard disks and flash drives by using multiple append-only data structures. As a first step towards accurate and fast evaluation of MSLS, we propose new analytic primitives and MSLS design models that quickly give accurate performance estimates. Our model can almost perfectly estimate the cost of inserts in LevelDB, whereas the conventional worst-case analysis gives 1.8–3.5X higher estimates than the actual cost. A few minutes of offline analysis using our model can find optimized system parameters that decrease LevelDB's insert cost by up to 9.4–26.2%; our analytic primitives and model also suggest changes to RocksDB that reduce its insert cost by up to 32.0%, without reducing query performance or requiring extra memory.

1 Introduction

Log-structured store designs provide fast write and easy crash recovery for block-based storage devices that have considerably higher sequential write speed than random write speed [37]. In particular, multi-stage versions of log-structured designs, such as LSM-tree [36], COLA [2], and SAMT [42], strive to balance read speed, write speed, and storage space use by segregating fresh and old data in multiple append-only data structures. These designs have been widely adopted in modern datastores including LevelDB [19], RocksDB [12], Bigtable [8], HBase [45], and Cassandra [27].

Given the variety of multi-stage log-structured (MSLS) designs, a system designer is faced with a problem of plenty, raising questions such as: Which design is best for this workload? How should the systems' parameters be set? How sensitive is that choice to changes in workload? Our goal in this paper is to move toward answering these questions and more through an improved—both in quality and in speed—analytical method for understanding and comparing the performance of these systems. This analytical approach can help shed light on how different design choices affect the performance of today's systems, and it provides an opportunity to optimize (based on the analysis) parameter choices given a workload. For example, in Section 6, we show that a few minutes of offline analysis can find improved parameters for LevelDB that decrease the cost of inserts by up to 9.4–26.2%. As another example, in Section 7, we reduce the insert cost in RocksDB by up to 32.0% by changing its system design based upon what we have learned from our analytic approach.

Prior evaluations of MSLS designs largely reside at the two ends of the spectrum: (1) asymptotic analysis and (2) experimental measurement. *Asymptotic analysis* of an MSLS design typically gives a big-O term describing the cost of an operation type (e.g., query, insert), but previous asymptotic analyses do not reflect real-world performance because they assume the worst case. *Experimental measurement* of an implementation produces accurate performance numbers, which are often limited to a particular implementation and workload, with lengthy experiment time to explore various system configurations.

This paper proposes a new evaluation method for MSLS designs that provides accurate and fast evaluation without needing to run the full implementations. Our approach uses new analytic primitives that help model the dynamics of MSLS designs. We build upon this model by combining it with a nonlinear solver to help automatically optimize system parameters to maximize performance.

This paper makes four key contributions:

- New analytic primitives to model creating the log structure and merging logs with redundant data (§3);
- System models for LevelDB and RocksDB, representative MSLS designs, using the primitives (§4, §5);
- Optimization of system parameters with the LevelDB model, improving real system performance (§6); and
- Application of lessons from the LevelDB model to the RocksDB system to reduce its write cost (§7).

Section 2 describes representative MSLS designs and common evaluation metrics for MSLS designs. Section 8 broadens the applications of our analytic primitives. Section 9 discusses the implications and limitations of our method. Appendix A provides proofs. Appendices B and C include additional system models.

2 Background

This section introduces a family of multi-stage log-structured designs and their practical variants, and explains metrics commonly used to evaluate these designs.

2.1 Multi-Stage Log-Structured Designs

A multi-stage log-structured (MSLS) design is a storage system design that contains multiple append-only data structures, each of which is created by sequential writes; for instance, several designs use sorted arrays and tables that are often called *SSTables* [19, 42]. These data structures are organized as *stages*, either logically or physically, to segregate different classes of data—e.g., fresh data and old data, frequently modified data and static data, small items and large items, and so forth. *Components* in LSM-tree [36] and *levels* in many designs [2, 19, 42] are examples of stages.

MSLS designs exploit the fast sequential write speed of modern storage devices. On hard disk and flash drives, sequential writes are up to an order of magnitude faster than random writes. By restricting most write operations to incur only sequential I/O, MSLS can provide fast writes.

Using multiple stages reduces the I/O cost for data updates. Frequent changes are often contained within a few stages that either reside in memory and/or are cheap to rewrite—this approach shares the same insight as the generational garbage collection used for memory management [25, 28]. The downside is that the system may have to search in multiple stages to find a single item because the item can exist in any of these stages. This can potentially reduce query performance.

The system moves data between stages based upon certain criteria. Common conditions are the byte size of the data stored in a stage, the age of the stored data, etc. This data migration typically reduces the total data volume by merging multiple data structures and reducing the redundancy between them; therefore, it is referred to as "compaction" or "merge."

MSLS designs are mainly classified by how they organize log structures and how and when they perform compaction. The data structures and compaction strategy significantly affect the cost of various operations.

2.1.1 Log-Structured Merge-Tree

The log-structured merge-tree (LSM-tree) [36] is a write-optimized store design with two or more components, each of which is a tree-like data structure [35]. One component (C_0) resides in memory; the remaining components (C_1,C_2,...) are stored on disk. Each component can hold a set of items, and multiple components can contain multiple items of the same key. A lower-numbered component always stores a newer version of the item than any higher-numbered component does.

For query processing, LSM-tree searches in potentially multiple components. It starts from C_0 and stops as soon as the desired item is found.

Handling inserts involves updating the in-memory component and merging data between components. A new entry is inserted into C_0 (and is also logged to disk for crash

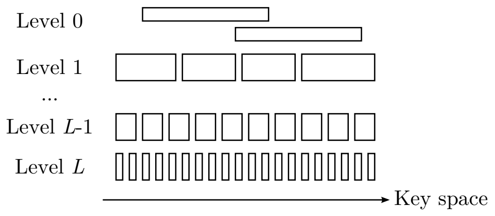

Figure 1: A simplified overview of LevelDB data structures. Each rectangle is an SSTable. Note that the x-axis is the key space; the rectangles are not to scale to indicate their byte size. The memtable and logs are omitted.

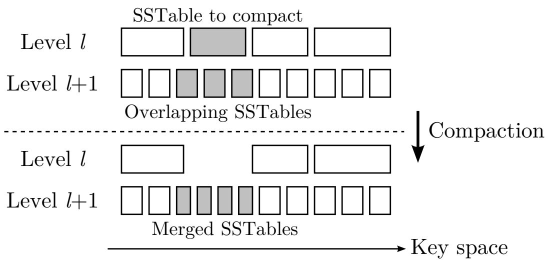

Figure 2: Compaction between two levels in LevelDB.

recovery), and the new item is migrated over time from C_0 to C_1, from C_1 to C_2, and so on. Frequent updates of the same item are coalesced in C_0 without spilling them to the disk; cold data, in contrast, remains in C_1 and later components which reside on low-cost disk.

The data merge in LSM-tree is mostly a sequential I/O operation. The data from C_l is read and merged into C_{l+1}, using a "rolling merge" that creates and updates nodes in the C_{l+1} tree incrementally in the key space.

The authors of LSM-tree suggested maintaining component sizes to follow a geometric progression. The size of a component is r times larger than the previous component size, where r is commonly referred to as a "growth factor," typically between 10 and 20. With such size selection, the expected I/O cost per insert by the data migration is $O((r+1)\log_r N)$, where N is the size of the largest component, i.e., the total number of unique keys. The worst-case lookup incurs $O(\log_r N)$ random I/O by accessing all components, if finding an item in a component costs $O(1)$ random I/O.

2.1.2 LevelDB

LevelDB [19] is a well-known variant of LSM-tree. It uses an in-memory table called a memtable, on-disk log files, and on-disk SSTables. The memtable plays the same role as C_0 of LSM-tree, and write-ahead logging is used for recovery. LevelDB organizes multiple levels that correspond to the components of LSM-tree; however, as shown in Figure 1, LevelDB uses a set of SSTables instead of a single tree-like structure for each level, and LevelDB's first level (level-0) can contain duplicate items across multiple SSTables.

Handing data updates in LevelDB is mostly similar to LSM-tree with a few important differences. Newly inserted data is stored in the memtable and appended to a log file. When the log size exceeds a threshold (e.g., 4 MiB[1]), the content of the memtable is converted into an SSTable and inserted to level-0. When the table count in level-0 reaches a threshold (e.g., 4), LevelDB begins to migrate the data of level-0 SSTables into level-1. For level-1 and later levels, when the aggregate byte size of SSTables in a level reaches a certain threshold, LevelDB picks an SSTable from that level and merges it into the next level. Figure 2 shows the compaction process; it takes all next-level SSTables whose key range overlaps with the SSTable being compacted, replacing the next-level SSTables with new SSTables containing merged items.

The SSTables created by compaction follow several invariants. A new SSTable has a size limit (e.g., 2 MiB), which makes the compaction process incremental. An SSTable cannot have more than a certain amount of overlapping data (e.g., 20 MiB) in the next level, which limits the future cost of compacting the SSTable.

LevelDB compacts SSTables in a circular way within the key space for each level. Fine-grained SSTables and round-robin SSTable selection have interesting implications in characterizing LevelDB's write cost.

There are several variants of LevelDB. A popular version is RocksDB [12], which claims to improve write performance with better support for multithreading. Unlike LevelDB, RocksDB picks the largest SSTable available for concurrent compaction. We discuss the impact of this strategy in Section 7. RocksDB also supports "universal compaction," an alternative compaction strategy that trades read performance for faster writes.

We choose to apply our analytic primitives and modeling techniques to LevelDB in Section 4 because it creates interesting and nontrivial issues related to its use of SSTables and incremental compaction. We show how we can analyze complex compaction strategies such as RocksDB's universal compaction in Section 5.

2.2 Common Evaluation Metrics

This paper focuses on analytic metrics (e.g., per-insert cost factors) more than on experimental metrics (e.g., insert throughput represented in MB/s or kOPS).

Queries and inserts are two common operation types. A query asks for one or more data items, which can also return "not found." An insert stores new data or updates existing item data. While it is hard to define a cost metric for every type of query and insert operation, prior studies extensively used two metrics defined for the *amortized I/O cost per processed item*: read amplification and write amplification.

Read amplification (RA) is the expected number of random read I/O operations to serve a query, assuming that the total data size is much larger than the system memory size, which translates to the expected I/O overhead of query processing [7, 29]. RA is based on the fact that random I/O access on disk and flash is a critical resource for query performance.

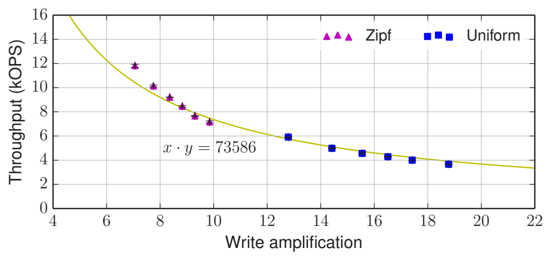

Figure 3: Write amplification is an important metric; increased write amplification decreases insert throughput on LevelDB.

Write amplification (WA) is the expected amount of data written to disk or flash per insert, which measures the I/O overhead of insert processing. Its concept originates from a metric to measure the efficiency of the flash translation layer (FTL), which stores blocks in a log structure-like manner; WA has been adopted later in key-value store studies to project insert throughput and estimate the life expectancy of underlying flash drives [12, 19, 29, 30, 43].

WA and insert throughput are inversely related. Figure 3 shows LevelDB's insert throughput for 1 kB items on a fast flash drive.[2] We vary the total data volume from 1 GB to 10 GB and examine two distributions for the key popularity, uniform and Zipf. Each configuration is run 3 times. Workloads that produce higher WA (e.g., larger data volume and/or uniform workloads) have lower throughput.

In this paper, our main focus is WA. Unlike RA, whose effect on actual system performance can be reduced by dedicating more memory for caching, the effect of WA cannot be mitigated easily without changing the core system design because the written data must be eventually flushed to disk/flash to ensure durability. For the same reason, we do not to use an extended definition of WA that includes the expected amount of data *read* from disk per insert. We discuss how to estimate RA in Section 8.

3 Analytic Primitives

Our goal in later sections is to create simple but accurate models of the write amplification of different MSLS designs. To reach this goal, we first present three new analytic primitives, Unique, Unique^{-1}, and Merge, that form the basis for those models. In Sections 4 and 5, we show how to express the insert and growth behavior of LevelDB and RocksDB using these primitives.

[1] Mi denotes 2^{20}. k, M, and G denote 10^3, 10^6, and 10^9, respectively.

[2] We use Intel® SSDSC2BB160G4T with `fsync` enabled for LevelDB.

3.1 Roles of Redundancy

Redundancy has an important effect on the behavior of an MSLS design. Any given table store (SSTable, etc.) contains at most one entry for a single key, no matter how many inserts were applied for that key. Similarly, when compaction merges tables, the resulting table will also contain only a single copy of the key, no matter how many times it appeared in the tables that were merged. Accurate models must thus consider redundancy.

Asymptotic analyses in prior studies ignore redundancy. Most analyses assume that compactions observe no duplicate keys from insert requests and input tables being merged [2, 19]. The asymptotic analyses therefore give the same answer regardless of skew in the key popularity; it ignores whether all keys are equally popular or some keys are more popular than others. It also estimates only an upper bound on the compaction cost—duplicate keys mean that less total data is written, lowering real-world write amplification.

We first clarify our assumptions and then explain how we quantify the effect of redundancy.

3.2 Notation and Assumptions

Let K be the key space. Without loss of generality, K is the set of all integers in $[0, N-1]$, where N is the total number of unique keys that the workload uses.

A discrete random variable X maps an insert request to the key referred to by the request. f_X is the probability mass function for X, i.e., $f_X(k)$ for $k \in K$ is the probability of having a specific key k for each insert request, assuming the keys in the requests are independent and identically distributed (i.i.d.) and have no spatial locality in popularity. As an example, a Zipf key popularity is defined as $f_X(h(i)) = (1/i^s)/(\sum_{n=1}^{N} 1/n^s)$, where s is the skew and h maps the rank of each key to the key.[3] Since there is no restriction on how f_X should look, it can be built from a key popularity distribution inferred by an empirical workload characterization [1, 38, 49].

Without loss of generality, $0 < f_X(k) < 1$. We can remove any key k satisfying $f_X(k) = 0$ from K because k will never appear in the workload. Similarly, $f_X(k) = 1$ degenerates to a workload with exactly one key, which is trivial to analyze.

A table is a set of the items that contains no duplicate keys. Tables are constructed from a sequence of insert requests or merges of other tables.

L refers to the total number of standard levels in an MSLS design. *Standard* levels include only the levels that follow the invariants of the design; for example, the level-0 in LevelDB does not count towards L because level-0 contains overlapping tables, while other levels do not, and has a different compaction trigger that is based on the table count in the level, not the aggregate size of tables in a level. L is closely related to read amplification; an MSLS design may require L random I/Os to retrieve an item that exists only in the last level (unless the design uses additional data structures such as Bloom filters [5]).

[3]Note that $s = 0$ leads to a *uniform* key popularity, i.e., $f_X(k) = 1/N$. We use $s = 0.99$ frequently to describe a "skewed" or simply "Zipf" distribution for the key popularity, which is the default skew in YCSB [11].

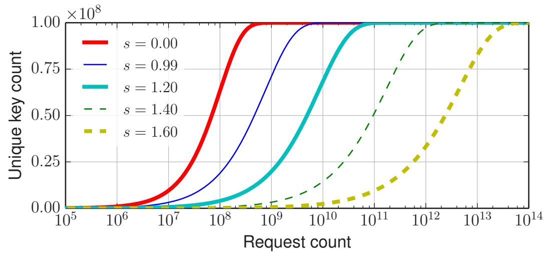

Figure 4: Unique key count as a function of request count for 100 million unique keys, with varying Zipf skew (s).

To avoid complicating the analysis, we assume that all items have equal size (e.g., 1000 bytes). This assumption is consistent with YCSB [11], a widely-used key-value store benchmark. We relax this assumption in Section 8.

3.3 Counting Unique Keys

A sequence of insert requests may contain duplicate keys. The requests with duplicate keys overwrite or modify the stored values. When storing the effect of the requests in a table, only the final (combined) results survive. Thus, a table can be seen as a set of distinct keys in the requests.

We first formulate Unique, a function describing the expected number of unique keys that appear in p requests:

Definition 1.

$$\text{Unique}(p) := N - \sum_{k \in K} (1 - f_X(k))^p \text{ for } p \geq 0.$$

Theorem 1. Unique(p) *is the expected number of unique keys that appear in p requests.*

Figure 4 plots the number of unique keys as a function of the number of insert requests for 100 million unique keys ($N = 10^8$). We use Zipf distributions with varying skew. The unique key count increases as the request count increases, but the increase slows down as the unique key count approaches the total unique key count. The unique key count with less skewed distributions increases more rapidly than with more skewed distributions until it is close to the maximum.

In the context of MSLS designs, Unique gives a hint about how many requests (or how much time) it takes for a level to reach a certain size from an empty state. With no or low skew, a level quickly approaches its full capacity and the system initiates compaction; with high skew, however, it can take a long time to accumulate enough keys to trigger compaction.

We examine another useful function, Unique^{-1}, which

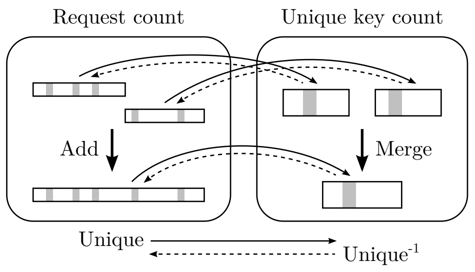

Figure 5: Isomorphism of Unique. Gray bars indicate a certain redundant key.

is the inverse function of Unique. $\text{Unique}^{-1}(u)$ estimates the expected number of requests to observe u unique keys in the requests.[4] By extending the domain of Unique to the real numbers, we can ensure the existence of Unique^{-1}:

Lemma 1. $\text{Unique}^{-1}(u)$ exists for $0 \le u < N$.

We further extend the domain of Unique^{-1} to include N by using limits. $\text{Unique}(\infty) := \lim_{p\to\infty} \text{Unique}(p) = N$; $\text{Unique}^{-1}(N) := \lim_{u\to N} \text{Unique}^{-1}(u) = \infty$.

It is straightforward to compute the value of $\text{Unique}^{-1}(u)$ by solving $\text{Unique}(p) = u$ for p numerically or by approximating Unique and Unique^{-1} with piecewise linear functions.

3.4 Merging Tables

Compaction takes multiple tables and creates a new set of tables that contain no duplicate keys. Nontrivial cases involve tables with overlapping key ranges. For such cases, we can estimate the size of merged tables using a combination of Unique and Unique^{-1}:

Definition 2. $\text{Merge}(u,v) := \text{Unique}(\text{Unique}^{-1}(u) + \text{Unique}^{-1}(v))$ for $0 \le u,v \le N$.

Theorem 2. $\text{Merge}(u,v)$ *is the expected size of a merged table that is created from two tables of sizes u and v.*

In worst-case analysis, merging tables of size u and v results in a new table of size $u+v$, assuming the input tables contain no duplicate keys. The error caused by this assumption grows as u and v approach N and as the key popularity has more skew. For example, with 100 million (10^8) total unique keys and Zipf skew of 0.99, $\text{Merge}(10^7, 9\times10^7) \approx 9.03\times10^7$ keys, whereas the worst-case analysis expects 10^8 keys.

Finally, Unique is an isomorphism as shown in Figure 5. Unique maps the length of a sequence of requests to the number of unique keys in it, and Unique^{-1} does the opposite. Adding request counts corresponds to applying Merge to unique key counts; the addition calculates the length of concatenated request sequences, and Merge obtains the number of unique keys in the merged table. Translating the number of requests to the number of unique keys and vice versa makes it easy to build an MSLS model, as presented in Section 4.

[4] Unique^{-1} is similar to, but differs from, the generalized coupon collector's problem (CCP) [16]. Generalized CCP terminates *as soon as* a certain number of unique items has been collected, whereas Unique^{-1} is merely defined as the inverse of Unique. Numerically, solutions of the generalized CCP are typically smaller than those of Unique^{-1} due to CCP's eager termination.

3.5 Handling Workloads with Dependence

As stated in Section 3.2, our primitives assume i.i.d., that insertions are independent, yet real-world workloads can have dependence between keys. A common scenario is using composite keys to describe multiple attributes of a single entity [26, 34]: [`book100|title`], [`book100|author`], [`book100|date`]. Related composite keys are often inserted together, resulting in dependent inserts.

Fortunately, we can treat these dependent inserts as independent if each insert is independent of a large number of (but not necessarily all) other inserts handled by the system. The dependence between a few inserts causes little effect on the overall compaction process because compaction involves many keys; for example, the compaction cost difference between inserting keys independently and inserting 10 related keys sharing the same key prefix as a batch is only about 0.2% on LevelDB when the workload contains 1 million or more total unique keys (for dependent inserts, 100,000 or more independent key groups, each of which has 10 related keys). Therefore, our primitives give good estimates in many practical scenarios which lack strictly independent inserts.

4 Modeling LevelDB

This section applies our analytic primitives to model a practical MSLS design, LevelDB. We explain how the dynamics of LevelDB components can be incorporated into the LevelDB model. We compare the analytic estimate with the measured performance of both a LevelDB simulator and the original implementation.

We assume that the dictionary-based compression [10, 17, 21] is not used in logs and SSTables. Using compression can reduce the write amplification (WA) by a certain factor; its effectiveness depends on how compressible the stored data is. Section 8 discusses how we handle variable-length items created as a result of compression.

Algorithm 1 summarizes the WA estimation for LevelDB. **unique**() and **merge**() calculate Unique and Merge as defined in Section 3. **dinterval**() calculates DInterval, defined in this section.

4.1 Logging

LevelDB's write-ahead logging (WAL) writes roughly the same amount as the data volume of inserts. We do not need to account for key redundancy because logging does not perform redundancy removal. As a consequence,

```
 1 // @param  L     maximum level
 2 // @param  wal   write-ahead log file size
 3 // @param  c0    level-0 SSTable count
 4 // @param  size  level sizes
 5 // @return       write amplification
 6 function estimateWA_LevelDB(L, wal, c0, size[]) {
 7   local l, WA, interval[], write[];
 8 
 9   // mem -> log
10   WA = 1;
11 
12   // mem -> level-0
13   WA += unique(wal) / wal;
14 
15   // level-0 -> level-1
16   interval[0] = wal * c0;
17   write[1] = merge(unique(interval[0]), size[1]);
18   WA += write[1] / interval[0];
19 
20   // level-l -> level-(l+1)
21   for (l = 1; l < L; l++) {
22     interval[l] = interval[l-1] + dinterval(size, l);
23     write[l+1] = merge(unique(interval[l]),
        size[l+1]) + unique(interval[l]);
24     WA += write[l+1] / interval[l];
25   }
26 
27   return WA;
28 }
```

Algorithm 1: Pseudocode of a model of WA of LevelDB.

logging contributes 1 unit of WA (line #10). An advanced WAL scheme [9] can lower the logging cost below 1 unit.

4.2 Constructing Level-0 SSTables

LevelDB stores the contents of the memtable as a new SSTable in level-0 whenever the current log size reaches a threshold *wal*, which is 4 MiB by default.[5] Because an SSTable contains no redundant keys, we use Unique to compute the expected size of the SSTable corresponding to the accumulated requests; for every *wal* requests, LevelDB creates an SSTable of $\text{Unique}(wal)$, which adds $\text{Unique}(wal)/wal$ to WA (line #13).

4.3 Compaction

LevelDB compacts one or more SSTables in a level into the next level when any of the following conditions is satisfied: (1) level-0 has at least $c0$ SSTables; (2) the aggregate size of SSTables in a level-l ($1 \le l \le L$) reaches $\text{Size}(l)$ bytes; or (3) an SSTable has observed a certain number of seeks from query processing. The original LevelDB defines $c0 = 4$ SSTables[6] and $\text{Size}(l) = 10^l$ MiB. The level to compact is chosen based on the ratio of the current SSTable count or level size to the triggering condition, which can be approximated as prioritizing levels

[5]We use the byte size and the item count interchangeably based on the assumption of fixed item size, as described in Section 3.2.

[6]LevelDB begins compaction with 4 level-0 SSTables, and new insert requests stall if the compaction of level-0 is not fast enough that the level-0 SSTable count reaches 12.

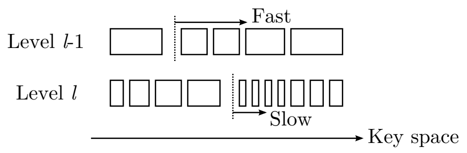

Figure 6: Non-uniformity of the key density caused by the different compaction speed of two adjacent levels in the key space. Each rectangle represents an SSTable. Vertical dotted lines indicate the last compacted key; the rectangles right next to the vertical lines will be chosen for compaction next time.

in their order from 0 to L in the model. The seek trigger depends on the distribution of queries as well as of insert requests, which is beyond the scope of this paper.

We examine two quantities to estimate the amortized compaction cost: a certain interval (a request count) that is large enough to capture the average compaction behavior of level-l, denoted as $\text{Interval}(l)$; and the expected amount of data written to level-$(l+1)$ during that interval, denoted as $\text{Write}(l+1)$. The contribution to WA by the compaction from level-l to level-$(l+1)$ is given by $\text{Write}(l+1)/\text{Interval}(l)$ by the definition of WA (line #18, #24).

4.3.1 Compacting Level-0 SSTables

LevelDB picks a level-0 SSTable and other level-0 SSTables that overlap with the first SSTable picked. It chooses overlapping level-1 SSTables as the other compaction input, and it can possibly choose more level-0 SSTables as long as the number of overlapping level-1 SSTables remains unchanged. Because level-0 contains overlapping SSTables with a wide key range, a single compaction commonly picks multiple level-0 SSTables; to build a concise model, we assume that all level-0 SSTables are chosen for compaction whenever the trigger for level-0 is met.

Let $\text{Interval}(0)$ be the interval of creating $c0$ SSTables, where $\text{Interval}(0) = wal \cdot c0$ (line #16). Compaction performed for that duration merges the SSTables created from $\text{Interval}(0)$ requests, which contain $\text{Unique}(\text{Interval}(0))$ unique keys, into level-1 with $\text{Size}(1)$ unique keys. Therefore, $\text{Write}(1) = \text{Merge}(\text{Unique}(\text{Interval}(0)), \text{Size}(1))$ (line #17).

4.3.2 Compacting Non-Level-0 SSTables

While the compaction from level-l to level-$(l+1)$ ($1 \le l < L$) follows similar rules as level-0 does, it is more complicated because of how LevelDB chooses the next SSTable to compact. LevelDB remembers $\text{LastKey}(l)$, the last key of the SSTable used in the last compaction for level-l and picks the first SSTable whose smallest key succeeds $\text{LastKey}(l)$; if there exists no such SSTable, LevelDB picks the SSTable with the smallest key in the level. This compaction strategy chooses SSTables in a circular way in the key space for each level.

Non-uniformity arises from round-robin compaction.

Compaction removes items from a level, but its effect is localized in the key space of that level. Compaction from a lower level into that level, however, tends to push items across the key space of the receiving level: the lower level makes faster progress compacting the entire key space because it contains fewer items, as depicted in Figure 6. As a result, the recently-compacted part of the key space has a lower chance of having items (low density), whereas the other part, which has not been compacted recently, is more likely to have items (high density). Because the maximum SSTable size is constrained, the low density area has SSTables covering a wide key range, and the high density area has SSTables with a narrow key range.

This non-uniformity makes compaction cheaper. Compaction occurs for an SSTable at the dense part of the key space. The narrow key range of the dense SSTable means a relatively small number of overlapping SSTables in the next level. Therefore, the compaction of the SSTable results in less data written to the next level.

Some LevelDB variants [22] explicitly pick an SSTable that maximizes the ratio of the size of that SSTable to the size of all overlapping SSTables in the next level, in hope of making the compaction cost smaller. Interestingly, due to the non-uniformity, LevelDB already *implicitly* realizes a similar compaction strategy. Our simulation results (not shown) indicate that the explicit SSTable selection brings a marginal performance gain over LevelDB's circular SSTable selection.

To quantify the effect of the non-uniformity to compaction, we model the density distribution of a level. Let $\text{DInterval}(l)$ be the expected interval between compaction of the same key in level-l. This is also the interval to merge the level-l data into the entire key space of level-$(l+1)$. We use d to indicate the unidirectional distance from the most recently compacted key $\text{LastKey}(l)$ to a key in the key space, where $0 \le d < N$. $d = 0$ represents the key just compacted, and $d = N-1$ is the key that will be compacted next time. Let $\text{Density}(l,d)$ be the probability of having an item for the key with distance d in level-l. Because we assume no spatial key locality, we can formulate Density by approximating $\text{LastKey}(l)$ to have a uniform distribution:

Theorem 3. Assuming $\text{P}(\text{LastKey}(l) = k) = 1/N$ for $1 \le l < L$, $k \in K$, then $\text{Density}(l,d) = \text{Unique}(\text{DInterval}(l) \cdot d/N)/N$ for $1 \le l < L$, $0 \le d < N$.

We also use a general property of the density:

Lemma 2. $\sum_{d=0}^{N-1} \text{Density}(l,d) = \text{Size}(l)$ for $1 \le l < L$.

The value of $\text{DInterval}(l)$ can be obtained by solving it numerically using Theorem 3 and Lemma 2.

We see that $\text{DInterval}(l)$ is typically larger than $\text{Unique}^{-1}(\text{Size}(l))$ that represents the expected interval of compacting the same key *without* non-uniformity. For example, with $\text{Size}(l) = 10$ Mi, $N = 100$ M (10^8), and

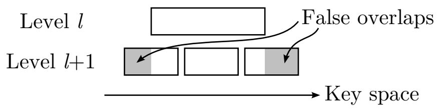

Figure 7: False overlaps that occur during the LevelDB compaction. Each rectangle indicates an SSTable; its width indicates the table's key range, not the byte size.

a uniform key popularity distribution, $\text{DInterval}(l)$ is at least twice as large as $\text{Unique}^{-1}(\text{Size}(l))$: 2.26×10^7 vs. 1.11×10^7. This confirms that non-uniformity does slow down the progression of $\text{LastKey}(l)$, improving the efficiency of compaction.

$\text{Interval}(l)$, the actual interval we use to calculate the amortized WA, is cumulative and increases by DInterval, i.e., $\text{Interval}(l) = \text{Interval}(l-1) + \text{DInterval}(l)$ (line #22). Because compacting lower levels is favored over compacting upper levels, an upper level may contain more data than its compaction trigger as an *overflow* from lower levels. We use a simple approximation to capture this behavior by adding the cumulative term $\text{Interval}(l-1)$.

False overlaps are another effect caused by the incremental compaction using SSTables in LevelDB. Unlike non-uniformity, they increase the compaction cost slightly. For an SSTable being compacted, overlapping SSTables in the next level may contain items that lie outside the key range of the SSTable being compacted, as illustrated in Figure 7. Even though the LevelDB implementation attempts to reduce such false overlaps by choosing more SSTables in the lower level without creating new overlapping SSTables in the next level, false overlaps may add extra data writes whose size is close to that of the SSTables being compacted, i.e., $\text{Unique}(\text{Interval}(l))$ for $\text{Interval}(l)$. Note that these extra data writes caused by false overlaps are more significant when Unique for the interval is large, i.e., under low skew, and they diminish as Unique becomes small, i.e., under high skew.

Several proposals [30, 41] strive to further reduce false overlaps by reusing a portion of input SSTables, essentially trading storage space and query performance for faster inserts. Such techniques can reduce WA by up to 1 per level, and even more if they address other types of false overlaps; the final cost savings, however, largely depend on the workload skew and the degree of the reuse.

By considering all of these factors, we can calculate the expected size of the written data. During $\text{Interval}(l)$, level-l accepts $\text{Unique}(\text{Interval}(l))$ unique keys from the lower levels, which are merged into the next level containing $\text{Size}(l+1)$ unique keys. False overlaps add extra writes roughly as much as the compacted level-l data. Thus, $\text{Write}(l+1) = \text{Merge}(\text{Unique}(\text{Interval}(l)), \text{Size}(l+1)) + \text{Unique}(\text{Interval}(l))$ (line #23).

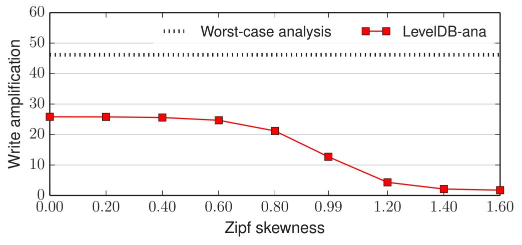

Figure 8: Effects of the workload skew on WA. Using 100 million unique keys, 1 kB item size.

4.4 Sensitivity to the Workload Skew

To examine how our LevelDB model reacts to the workload skew, we compare our WA estimates with worst-case analysis results. Our worst-case scenarios make the same assumption as prior asymptotic analyses [2, 36, 39], that the workload has no redundancy; therefore, merging two SSTables yields an SSTable whose size is exactly the same as the sum of the input SSTable sizes. In other words, compacting levels of size u and v results in $u+v$ items in the worst case.

Figure 8 plots the estimated WA for different Zipf skew parameters. Because our analytic model ("LevelDB-ana") considers the key popularity distribution of the workload in estimating WA, it clearly shows how WA decreases as LevelDB handles more skewed workloads; in contrast, the worst-case analysis ("Worst-case analysis") gives the same result regardless of the skew.

4.5 Comparisons with the Worst-Case Analysis, Simulation, and Experiment

We compare analytic estimates of WA given by our LevelDB model with the estimates given by the worst-case analysis, and the measured cost by running experiments on a LevelDB simulator and the original implementation.

We built a fast LevelDB simulator in C++ that follows the LevelDB design specification [20] to perform an item-level simulation and uses system parameters extracted from the LevelDB source code. This simulator does not intend to capture every detail of LevelDB implementation behaviors; instead, it realizes the high-level design components as explained in the LevelDB design document. The major differences are (1) our simulator runs in memory; (2) it performs compaction synchronously without concurrent request processing; and (3) it does not implement several opportunistic optimizations: (a) reducing false overlaps by choosing more SSTables in the lower level, (b) bypassing level-0 and level-1 for a newly created SSTable from the memtable if there are no overlapping SSTables in these levels, and (c) dynamically allowing more than 4 level-0 SSTables under high load.

For the measurement with the LevelDB implementation, we instrumented the LevelDB code (v1.18) to report

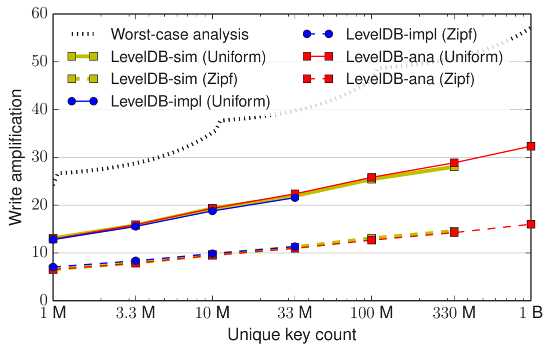

Figure 9: Comparison of WA between the estimation from our LevelDB model, the worst-case analysis, LevelDB simulation, and implementation results, with a varying number of total unique keys. Using 1 kB item size. Simulation and implementation results with a large number of unique keys are unavailable due to excessive runtime.

the number of bytes written to disk via system calls. We use an item size that is 18 bytes smaller than we do in the analysis and simulation, to compensate for the increased data writes due to LevelDB's own storage space overhead. For fast experiments, we disable `fsync` and checksumming,[7] which showed no effects on WA in our experiments. We also avoid inserting items at an excessive rate that can overload level-0 with many SSTables and cause a high lookup cost.

Both LevelDB simulator and implementation use a YCSB [11]-like workload generator written in C++. Each experiment initializes the system by inserting all keys once and then measures the average WA of executing random insert requests whose count is 10 times the total unique key count.

Figure 9 shows WA estimation and measurement with a varying number of total unique keys. Due to excessive experiment time, the graph excludes some data points for simulation ("LevelDB-sim") and implementation ("LevelDB-impl") with a large number of unique keys. The graph shows that our LevelDB model successfully estimates WA that agrees almost perfectly with the simulation and implementation results. The most significant difference occurs at 330 M unique keys with the uniform popularity distribution, where the estimated WA is only 3.0% higher than the measured WA. The standard worst-case analysis, however, significantly overestimates WA by 1.8–3.5X compared to the actual cost, which highlights the accuracy of our LevelDB model.

Figure 10 compares results with different *write buffer* size (i.e., the memtable size), which determines how much data in memory LevelDB accumulates to create a level-0 SSTable (and also affects how long crash recovery may

[7]MSLS implementations can use special CPU instructions to accelerate checksumming and avoid making it a performance bottleneck [23].

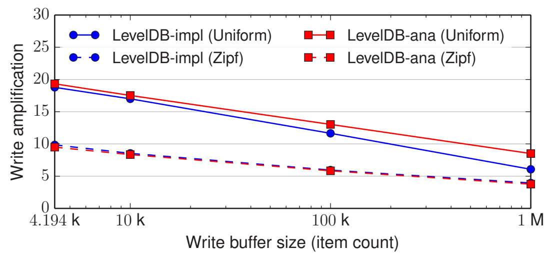

Figure 10: Comparison of WA between the estimation from our LevelDB model and implementation results, with varying write buffer sizes. Using 10 million unique keys, 1 kB item size.

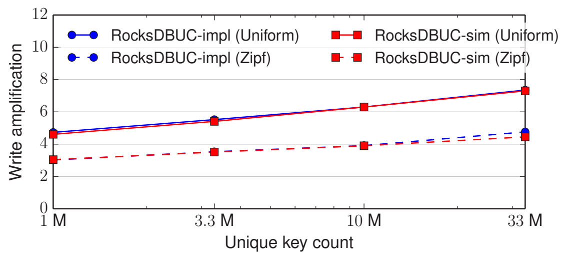

Figure 11: Comparison of WA between the estimation from our table-level simulation and implementation results for RocksDB's universal compaction, with a varying number of total unique keys. Using 1 kB item size.

take). In our LevelDB model, *wal* reflects the write buffer size. We use write buffer sizes between LevelDB's default size of 4 MiB and 10% of the last level size. The result indicates that our model estimates WA with good accuracy, but the error increases as the write buffer size increases for uniform key popularity distributions. We suspect that the error comes from the approximation in the model to take into account temporal overflows of levels beyond their maximum size; the error diminishes when level sizes are set to be at least as large as the write buffer size. In fact, avoiding too small level-1 and later levels has been suggested by RocksDB developers [14], and our optimization performed in Section 6 typically results in moderately large sizes for lower levels under uniform distributions, which makes this type of error insignificant for practical system parameter choices.

5 Modeling Universal Compaction

This section focuses on how we can model complex compaction strategies such as "universal compaction" implemented in RocksDB [15]. Section 7 revisits RocksDB to compare its "level style compaction" with LevelDB.

Universal compaction combines three small compaction strategies. RocksDB keeps a list of SSTables ordered by the age of their data, and compaction is restricted to adjacent tables. Compaction begins when the SSTable count exceeds a certain threshold (*Precondition*). First, RocksDB merges all SSTables whose total size minus the last one's size exceeds the last one's size by a certain factor (*Condition 1*); second, it merges consecutive SSTables that do not include a sudden increase in size beyond a certain factor (*Condition 2*); third, it merges the newest SSTables such that the total SSTable count drops below a certain threshold (*Condition 3*). Condition 1 avoids excessive duplicate data across SSTables, and Conditions 2 and 3 prevent high read amplification.

In such a multi-strategy system, it is difficult to determine how frequently each condition will cause compaction and what SSTables will be chosen for compaction.

We take this challenge as an opportunity to demonstrate how our analytic primitives are applicable to analyzing a complex system by using a *table-level simulation*. Unlike full simulators that keep track of individual items, a table-level simulator calculates only the SSTable size. It implements compaction conditions as the system design specifies, and it estimates the size of new or merged SSTables by using our analytic primitives. Dividing the total size of created SSTables by the total number of inserts gives the estimated WA. Unlike our LevelDB model that understands incremental compaction, a model for universal compaction does not need to consider non-uniformity and false overlaps. Interested readers may refer to Appendix C for the full pseudocode of the simulator.

Figure 11 compares WA obtained by our table-level simulation and the full RocksDB implementation. We use the default configuration, except for the SSTables count for compaction triggers set to 12. The simulation result ("RocksDBUC-sim") is close to the measured WA ("RocksDBUC-impl"). The estimated WA differs from the measured WA by up to 6.5% (the highest error with 33 M unique keys and skewed key inserts) though the overall accuracy remains as high as our LevelDB model presented in Section 4.

6 Optimizing System Parameters

Compared to full simulators and implementations, an analytic model offers fast estimation of cost metrics for a given set of system parameters. To demonstrate fast evaluation of the analytic model, we use an example of optimizing LevelDB system parameters to reduce WA using our LevelDB model.

Note that the same optimization effort could be made with the full LevelDB implementation by substituting our LevelDB model with the implementation and a synthetic workload generator. However, it would take prohibitively long to explore the large parameter space, as examined in Section 6.4.

6.1 Parameter Set to Optimize

The level sizes, Size(l), are important system parameters in LevelDB. They determine when LevelDB should initi-

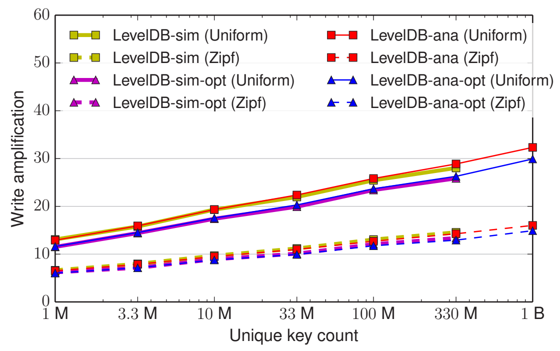

Figure 12: Improved WA using optimized level sizes on our analytic model and simulator for LevelDB.

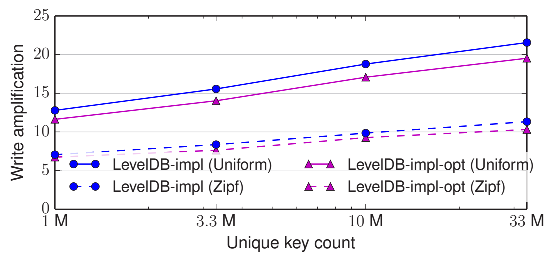

Figure 13: Improved WA using optimized level sizes on the LevelDB implementation.

ate compaction for standard levels and affect the overall compaction cost of the system. The original LevelDB design uses a geometric progression of $\text{Size}(l) = 10^l$ MiB. Interesting questions are (1) what level sizes different workloads favor; and (2) whether the geometric progression of level sizes is the optimal for all workloads.

Using different level sizes does not necessarily trade query performance or memory use. The log size, level-0 SSTable count, and total level count—the main determinants of query performance—are all unaffected by this system parameter.

6.2 Optimizer

We implemented a system parameter optimizer based on our analytic model. The objective function to minimize is the estimated WA. Input variables are $\text{Size}(l)$, excluding $\text{Size}(L)$, which will be equal to the total unique key count. After finishing the optimization, we use the new level sizes to obtain new WA estimates and measurement results on our analytic model and simulator. We also force the LevelDB implementation to use the new level sizes and measure WA. Our optimizer is written in Julia [4] and uses Ipopt [47] for nonlinear optimization. To speed up Unique, we use a compressed key popularity distribution which groups keys with similar probabilities and stores their average probability.[8]

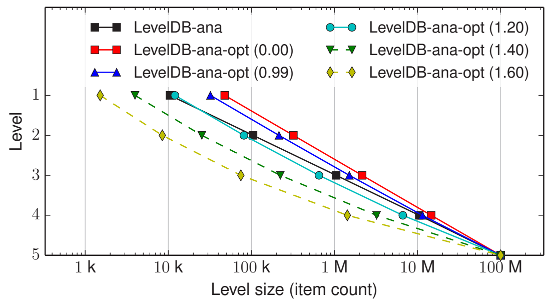

Figure 14: Original and optimized level sizes with varying Zipf skew. Using 100 million unique keys, 1 kB item size.

Source	Analysis		Simulation	
	No opt	Opt	No opt	Opt
mem→log	1.00	1.00	1.00	1.00
mem→level-0	1.00	1.00	1.00	1.00
level-0→1	1.62	3.85	1.60	3.75
level-1→2	4.77	4.85	4.38	4.49
level-2→3	6.22	4.82	6.04	4.66
level-3→4	6.32	4.65	6.12	4.58
level-4→5	4.89	3.50	5.31	3.93
Total	25.82	23.67	25.45	23.41

Table 1: Breakdown of WA sources on the analysis and simulation without and with the level size optimization. Using 100 million unique keys, 1 kB item size, and a uniform key popularity distribution.

6.3 Optimization Results

Our level size optimization successfully reduces the insert cost of LevelDB. Figures 12 and 13 plot WA with optimized level sizes. Both graphs show that the optimization ("LevelDB-ana-opt," "LevelDB-sim-opt," and "LevelDB-impl-opt") improves WA by up to 9.4%. The analytic estimates and simulation results agree with each other as before, and the LevelDB implementation exhibits lower WA across all unique key counts.

The optimization is effective because level sizes differ by the workload skew, as shown in Figure 14. Having larger lower levels is beneficial for relatively low skew as it reduces the size ratio of adjacent levels. On the other hand, high skew favors smaller lower levels and level sizes that grow faster than the standard geometric progression. With high skew, compaction happens more frequently in the lower levels to remove redundant keys; keeping these levels small reduces the cost of compaction. This result suggests that it is suboptimal to use fixed level sizes for different workloads and that using a geometric progression of level sizes is not always the best design to minimize WA.

[8]For robustness, we optimize using both the primal and a dual form of the LevelDB model presented in Section 4. The primal optimizes over $\text{Size}(l)$ and the dual optimizes over $\text{Unique}^{-1}(\text{Size}(l))$. We pick the result of whichever model produces the smaller WA.

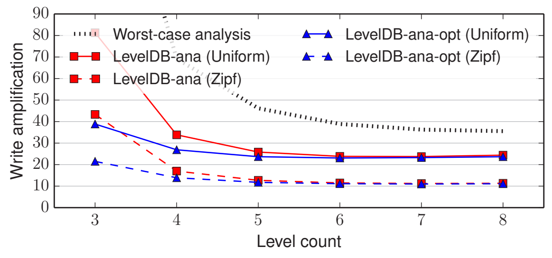

Figure 15: WA using varying numbers of levels. The level count excludes level-0. Using 100 million unique keys, 1 kB item size.

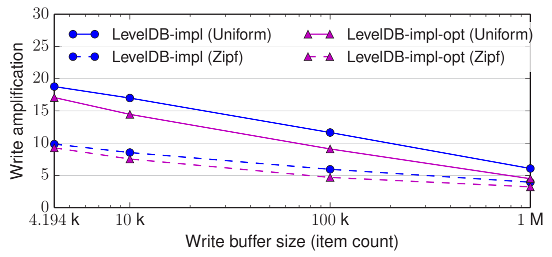

Figure 16: Improved WA using optimized level sizes on the LevelDB implementation, with a large write buffer. Using 10 million unique keys, 1 kB item size.

Table 1 further examines how the optimization affects per-level insert costs, using the LevelDB model and simulation. Per-level WA tends to be more variable using the original level sizes, while the optimization makes them relatively even across levels except the last level. This result suggests that it may be worth performing a runtime optimization that dynamically adjusts level sizes to achieve the lower overall WA by reducing the variance of the per-level WA.

By lowering WA, the system can use fewer levels to achieve faster lookup speed without significant impact on insert costs. Figure 15 reveals how much extra room for query processing the optimization can create. This analysis changes the level count by altering the growth factor of LevelDB, i.e., using a higher growth factor for a lower level count. The result shows that the optimization is particularly effective with a fewer number of levels, and it can save almost a whole level's worth of WA compared to using a fixed growth factor. For example, with the optimized level sizes, a system can use 3 levels instead of 4 levels without incurring excessively high insert costs.

A LevelDB system with large memory can further benefit from our level size optimization. Figure 16 shows the result of applying the optimization to the LevelDB implementation, with a large write buffer. The improvement becomes more significant as the write buffer size increases, reaching 26.2% of WA reduction at the buffer size of 1 million items.

6.4 Optimizer Performance

The level size optimization requires little time due to the fast evaluation of our analytic model. For 100 million unique keys with a uniform key popularity distribution, the entire optimization took 2.63 seconds, evaluating 17,391 different parameter sets (6,613 evaluations per second) on a server-class machine equipped with Intel® Xeon® E5-2680 v2 processors. For the same-sized workload, but with Zipf skew of 0.99, the optimization time increased to 79 seconds, which is far more than the uniform case, but is less than 2 minutes; for this optimization, the model was evaluated 16,680 times before convergence (211 evaluations per second).

Evaluating this many system parameters using a full implementation—or even item-level simulation—is prohibitively expensive. Using the same hardware as above, our in-memory LevelDB simulator takes 45 minutes to measure WA for a *single* set of system parameters with 100 million unique keys. The full LevelDB implementation takes 101 minutes (without `fsync`) to 490 minutes (with `fsync`), for a smaller dataset with 10 million unique keys.

7 Improving RocksDB

In this section, we turn our attention to RocksDB [12], a well-known variant of LevelDB. RocksDB offers improved capabilities and multithreaded performance, and provides an extensive set of system configurations to temporarily accelerate bulk loading by sacrificing query performance or relaxing durability guarantees [13, 14]; nevertheless, there have been few studies of how RocksDB's *design* affects its performance. We use RocksDB v4.0 and apply the same set of instrumentation, configuration, and workload generation as we do to LevelDB.

RocksDB supports "level style compaction" that is similar to LevelDB's data layout, but differs in how it picks the next SSTable to compact. RocksDB picks the largest SSTable in a level for compaction,[9] rather than keeping LevelDB's round-robin SSTable selection. We learned in Section 4, however, that LevelDB's compaction strategy is effective in reducing WA because it tends to pick SSTables that overlap a relatively small number of SSTables in the next level.

To compare the compaction strategies used by LevelDB and RocksDB, we measure the insert cost of both systems in Figure 17. Unfortunately, the current RocksDB strategy produces higher WA ("RocksDB-impl") than LevelDB does ("LevelDB-impl"). In theory, the RocksDB approach may help multithreaded compaction because large tables may be spread over the entire key space so that they facilitate parallel compaction; this effect, how-

[9]Recent versions of RocksDB support additional strategies for SSTable selection.

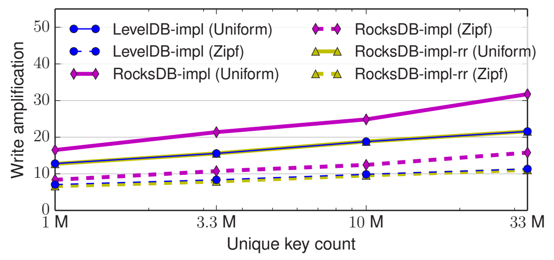

Figure 17: Comparison of WA between LevelDB, RocksDB, and a modified RocksDB with LevelDB-like SSTable selection.

ever, was not evident in our experiments using multiple threads. The high insert cost of RocksDB is entirely caused by RocksDB's compaction strategy; implementing LevelDB's SSTable selection in RocksDB ("RocksDB-impl-rr") reduces RocksDB's WA by up to 32.0%, making it comparable to LevelDB's WA. This result confirms that LevelDB's strategy is good at reducing WA as our analytic model predicts.

We have not found a scenario where RocksDB's current strategy excels, though some combinations of workloads and situations may favor it. LevelDB and RocksDB developers may or may have not intended any of the effects on WA when designing their systems. Either way, our analytic model provides quantitative evidence that LevelDB's table selection will perform well under a wide range of workloads despite being the "conventional" solution.

8 Estimating Read Amplification

This section presents read amplification (RA) estimation.

We introduce a *weighted* variant of our analytic primitives. A per-key weight w, which is nontrivial (i.e., $w(k) \neq 0$ for some k) and nonnegative, specifies how much contribution each key makes to the result:

Definition 3.
$\text{Unique}(p,w) := \sum_{k \in K} [1-(1-f_X(k))^p]\, w(k)$ for $p \geq 0$.

We construct $w(k)$ to indicate the probability of having key k for each query. For level-l, let $s(l)$ and $e(l)$ be the expected age of the newest and oldest item in level-l in terms of the number of inserts, obtained by using the system model presented in Section 4. We find $c(l)$, the expected I/O cost to perform a query at level-l. The expected I/O cost to perform queries that finish at level-l is $[\text{Unique}(e(l),w) - \text{Unique}(s(l),w)] \cdot c(l)$. Adding the expected I/O cost of each level gives the overall RA.

As another use case, the weighted variant can add support for *variable-length* items to the system models presented in Sections 4 and 5. By setting $w(k)$ to the size of the item for key k, Unique returns the expected size of unique items instead of their expected count. Because weighted Unique is still strictly monotonic, weighted Unique^{-1} and Merge exist.

9 Discussion

Analyzing an MSLS design with an accurate model can provide useful insights on how one should design a new MSLS to exploit opportunities provided by workloads. For example, our analytic model reveals that LevelDB's byte size-based compaction trigger makes compaction much less frequent and less costly under skew; such a design choice should be suitable for many real-world workloads with skew [1].

A design process complemented with accurate analysis can help avoid false conclusions about a design's performance. LevelDB's per-level WA is less (only up to 4–6) than assumed in the worst case (11–12 for a growth factor of 10), even for uniform workloads. Our analytical model justifies LevelDB's high growth factor, which turns out to be less harmful for insert performance than standard worst-case analysis implies.

Our analytic primitives and modeling are not without limitations. Assumptions such as independence and no spatial locality in requested keys may not hold if there are dependent keys that share the same prefix though a small amount of such dependence does not change the overall system behavior and thus can be ignored as discussed in Section 3.5. Our modeling in Section 4 does not account for time-varying workload characteristics (e.g., flash crowds) or special item types such as tombstones that represent item deletion, while the simulation-oriented modeling in Section 5 can handle such cases. We leave extending our primitives further to accommodate remaining cases as future work.

Both design and implementation influence the final system performance. Our primitives and modeling are useful for understanding the *design* of MSLS systems. Although we use precise metrics such as WA to describe the system performance throughout this work, these metrics are ultimately not identical to implementation-level metrics such as operations per second. Translating a good system design into an efficient implementation is critical to achieving good performance, and remains a challenging and important goal for system developers and researchers.

10 Related Work

Over the past decade, numerous studies have proposed new multi-stage log-structured (MSLS) designs and evaluated their performance. In almost every case, the authors present implementation-level performance [2, 12, 18, 19, 22, 29, 30, 32, 39, 40, 41, 42, 43, 46, 48]. Some employ analytic metrics such as write amplification to explain the design rationale, facilitate design comparisons, and generalize experiment results [12, 22, 29, 30, 32, 39, 43], and most of the others also use the concept of per-operation costs. However, they eventually rely on the experimental measurement because their analysis fails to offer sufficiently high accuracy to make meaningful performance comparisons. LSM-tree [36], LHAM [33], COLA [2], bLSM [39], and B-tree variants [6, 24] provide extensive analysis on their design, but their analyses are limited to asymptotic complexities or always assume the worst case.

Despite such a large number of MSLS design proposals, there is little active research to devise improved evaluation methods for these proposals to fill the gap between asymptotic analysis and experimental measurement. The sole existing effort is limited to a specific system design [31], but does not provide general-purpose primitives. We are unaware of prior studies that successfully capture workload skew and the dynamics of compaction to the degree that the estimates are close to simulation and implementation results, as we present in this paper.

11 Conclusion

We present new analytic primitives for modeling multi-stage log-structured (MSLS) designs, which can quickly and accurately estimate their performance. We have presented a model for the popular LevelDB system, which estimates write amplification very close to experimentally determined actual costs; using this model, we were able to find more favorable system parameters that reduce the overall cost of writes. Based upon lessons learned from the model, we propose changes to RocksDB to lower its insert costs. We believe that our analytic primitives and modeling method are applicable to a wide range of MSLS designs and performance metrics. The insights derived from the models facilitate comparisons of MSLS designs and ultimately help develop new designs that better exploit workload characteristics to improve performance.

Acknowledgments

This work was supported by funding from National Science Foundation under awards IIS-1409802 and CNS-1345305, and Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC). We would like to thank anonymous FAST reviewers for their feedback and James Mickens for shepherding this paper. We appreciate Eddie Kohler, Mark Callaghan, Kai Ren, and anonymous SOSP reviewers for their comments on early versions of this work.

References

[1] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the SIGMETRICS'12*, June 2012.

[2] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-oblivious streaming B-trees. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, 2007.

[3] J. L. Bentley and J. B. Saxe. Decomposable searching problems I: Static-to-dynamic transformations. *Journal of Algorithms*, 1(4):301–358, Dec. 1980.

[4] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A fresh approach to numerical computing. Dec. 2014. http://arxiv.org/abs/1411.1607.

[5] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[6] G. S. Brodal and R. Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2003.

[7] M. Callaghan. Read Amplification Factor. http://mysqlha.blogspot.com/2011/08/read-amplification-factor.html, 2011.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. 7th USENIX OSDI*, Nov. 2006.

[9] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson. From ARIES to MARS: Transaction support for next-generation, solid-state drives. In *Proc. 24th ACM Symposium on Operating Systems Principles (SOSP)*, Nov. 2013.

[10] Y. Collet. LZ4. https://github.com/Cyan4973/lz4, 2015.

[11] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. 1st ACM Symposium on Cloud Computing (SOCC)*, June 2010.

[12] Facebook. RocksDB. http://rocksdb.org/, 2015.

[13] Facebook. Performance Benchmarks. https://github.com/facebook/rocksdb/wiki/Performance-Benchmarks, 2014.

[14] Facebook. RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide, 2015.

[15] Facebook. RocksDB Universal Compaction. https://github.com/facebook/rocksdb/wiki/Universal-Compaction, 2015.

[16] P. Flajolet, D. Gardy, and L. Thimonier. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discrete Applied Mathematics*, 39(3), Nov. 1992.

[17] J.-L. Gailly and M. Adler. zlib. http://www.zlib.net/, 2013.

[18] M. Ghosh, I. Gupta, S. Gupta, and N. Kumar. Fast compaction algorithms for NoSQL databases. In *Proc. the 35th IEEE International Conference on Distributed Computing Systems (ICDCS)*, June 2015.

[19] Google. LevelDB. https://github.com/google/leveldb, 2014.

[20] Google. LevelDB file layout and compactions. https://github.com/google/leveldb/blob/master/doc/impl.html, 2014.

[21] Google. Snappy. https://github.com/google/snappy, 2015.

[22] HyperDex. HyperLevelDB. http://hyperdex.org/performance/leveldb/, 2013.

[23] Intel SSE4 programming reference. https://software.intel.com/sites/default/files/m/8/b/8/D9156103.pdf, 2007.

[24] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter. BetrFS: A right-optimized write-optimized file system. In *Proc. 13th USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2015.

[25] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 2011.

[26] B. Kate, E. Kohler, M. S. Kester, N. Narula, Y. Mao, and R. Morris. Easy freshness with Pequod cache joins. In *Proc. 11th USENIX NSDI*, Apr. 2014.

[27] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating System Review*, 44:35–40, Apr. 2010.

[28] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6), June 1983.

[29] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2011.

[30] L. Marmol, S. Sundararaman, N. Talagala, and R. Rangaswami. NVMKV: A scalable, lightweight, FTL-aware key-value store. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)*, July 2015.

[31] P. Menon, T. Rabl, M. Sadoghi, and H.-A. Jacobsen. Optimizing key-value stores for hybrid storage architectures. In *Proceedings of CASCON*, 2014.

[32] C. Min, W.-H. Kang, T. Kim, S.-W. Lee, and Y. I. Eom. Lightweight application-level crash consistency on transactional flash storage. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)*, July 2015.

[33] P. Muth, P. O'Neil, A. Pick, and G. Weikum. The LHAM log-structured history data access method. *The VLDB Journal*, 8(3-4):199–221, Feb. 2000.

[34] NoSQL data modeling techniques. https://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques/, 2012.

[35] P. E. O'Neil. The SB-tree: An index-sequential structure for high-performance sequential access. *Acta Inf.*, 29(3):241–265, 1992.

[36] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, 1996.

[37] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[38] T. Saemundsson, H. Bjornsson, G. Chockler, and Y. Vigfusson. Dynamic performance profiling of cloud caches. In *Proc. 5th ACM Symposium on Cloud Computing (SOCC)*, Nov. 2014.

[39] R. Sears and R. Ramakrishnan. bLSM: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012.

[40] R. Sears, M. Callaghan, and E. Brewer. Rose: Compressed, log-structured replication. *Proc. VLDB Endowment*, 1(1), Aug. 2008.

[41] P. Shetty, R. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok. Building workload-independent storage with VT-trees. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[42] R. P. Spillane, P. J. Shetty, E. Zadok, S. Dixit, and S. Archak. An efficient multi-tier tablet server storage architecture. In *Proc. 2nd ACM Symposium on Cloud Computing (SOCC)*, Oct. 2011.

[43] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. RIPQ: Advanced photo caching on flash for Facebook. In *Proc. 13th USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2015.

[44] The Apache Software Foundation. Apache Cassandra. https://cassandra.apache.org/, 2015.

[45] The Apache Software Foundation. Apache HBase. https://hbase.apache.org/, 2015.

[46] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi. LogBase: A scalable log-structured database system in the cloud. *Proc. VLDB Endowment*, 5(10), June 2012.

[47] A. Wächter and L. T. Biegler. On the implementation of an interior-point filter line-search algorithm

for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, 2006.

[48] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems*, 2014.

[49] J. Wires, S. Ingram, Z. Drudi, N. J. A. Harvey, and A. Warfield. Characterizing storage workloads with counter stacks. In *Proc. 11th USENIX OSDI*, Oct. 2014.

A Proofs

This section provides proofs for theorems presented in this paper.

Theorem 1. *(in Section 3.3)* Unique(p) *is the expected number of unique keys that appear in p requests.*

Proof. Key k counts towards the unique key count if k appears at least once in a sequence of p requests, whose probability is $1-(1-f_X(k))^p$. Therefore, the expected unique key count is $\sum_{k\in K}(1-(1-f_X(k))^p) = N-\sum_{k\in K}(1-f_X(k))^p = \text{Unique}(p)$. □

Lemma 1. *(in Section 3.3)* $\text{Unique}^{-1}(u)$ exists for $0 \le u < N$.

Proof. Suppose $0 \le p < q$. $(1-f_X(k))^q < (1-f_X(k))^p$ because $0 < 1-f_X(k) < 1$. $\text{Unique}(q)-\text{Unique}(p) = -\sum_{k\in K}(1-f_X(k))^q + \sum_{k\in K}(1-f_X(k))^p > 0$. Thus, Unique is a strictly monotonic function that is defined over $[0,N)$. □

Theorem 2. *(in Section 3.4)* Merge(u,v) *is the expected size of a merged table that is created from two tables whose size is u and v.*

Proof. Let p and q be the expected numbers of insert requests that would produce tables of size u and v, respectively. The merged table is expected to contain all $k \in K$ except those missing in both request sequences. Therefore, the expected merged table size is $N-\sum_{k\in K}(1-f_X(k))^p(1-f_X(k))^q = \text{Unique}(p+q)$. Because $p = \text{Unique}^{-1}(u)$ and $q = \text{Unique}^{-1}(v)$, $\text{Unique}(\text{Unique}^{-1}(u)+\text{Unique}^{-1}(v)) = \text{Merge}(u,v)$. □

Theorem 3. *(in Section 4.3)* Assuming $\text{P}(\text{LastKey}(l) = k) = 1/N$ for $1 \le l < L$, $k \in K$, then $\text{Density}(l,d) = \text{Unique}(\text{DInterval}(l)\cdot d/N)/N$ for $1 \le l < L$, $0 \le d < N$.

Proof. Suppose $\text{LastKey}(l) = k \in K$. Let k' be $(k-d+N) \bmod N$. Let r be $\text{DInterval}(l)\cdot d/N$. There are r requests since the last compaction of k'. Level-l has k' if any of r requests contains k', whose probability is $1-(1-f_X(k'))^r$.

By considering all possible k and thus all possible k', $\text{Density}(l,d) = \sum_{k\in K}(1/N)(1-(1-f_X(k))^r) = \text{Unique}(\text{DInterval}(l)\cdot d/N)/N$. □

Lemma 2. *(in Section 4.3)* $\sum_{d=0}^{N-1}\text{Density}(l,d) = \text{Size}(l)$ for $1 \le l < L$.

Proof. The sum over the density equals to the expected unique key count, which is the number of keys level-l maintains, i.e., $\text{Size}(l)$. □

Theorem 4. *(in Section 8) The expected I/O cost to perform queries that finishes at level-l is given by* $[\text{Unique}(e(l),w)-\text{Unique}(s(l),w)]\cdot c(l)$, *where w describes the query distribution and* $c(l)$ *is the expected I/O cost to perform a query at level-l.*

Proof. For key k to exist in level-l and be used for query processing (without being served in an earlier level), it must appear in at least one of $e(l)-s(l)$ requests and in none of other $s(l)$ requests. The first condition ensures the existence of the key in level-l, and the second condition rejects the existence of the key in an earlier level (otherwise, queries for key k will be served in that level). Thus, the probability of such a case is $(1-(1-f_X(k))^{e(l)-s(l)})\cdot(1-f_X(k))^{s(l)} = (1-f_X(k))^{s(l)}-(1-f_X(k))^{e(l)}$.

The expected I/O cost to perform a query for key k that finishes at level-l is $\left[(1-f_X(k))^{s(l)}-(1-f_X(k))^{e(l)}\right]\cdot c(l)$.

Because the fraction of the queries for key k among all queries is given by $w(k)$, the expected I/O cost to perform queries that finishes at level-l is $\sum_{k\in K}\left[(1-f_X(k))^{s(l)}-(1-f_X(k))^{e(l)}\right]c(l)w(k) = \sum_{k\in K}\left[\left((1-(1-f_X(k))^{e(l)}\right)-\left(1-(1-f_X(k))^{s(l)}\right)\right]w(k)$ $c(l) = [\text{Unique}(e(l),w)-\text{Unique}(s(l),w)]\cdot c(l)$. □

B Modeling COLA and SAMT

The cache-oblivious lookahead array (COLA) [2] is a generalized and improved binomial list [3]. Like LSM-tree, COLA has multiple levels whose count is $\lceil \log_r N \rceil$, where r is the growth factor. Each level contains zero or one SSTable. Unlike LSM-tree, however, COLA uses the merge count as the main compaction criterion; a level in COLA accepts $r-1$ merges with the lower level before the level is merged into the next level.

COLA has roughly similar asymptotic complexities to LSM-tree's. A query in COLA may cost $O(\log_r N)$ random I/O per lookup if looking up a level costs $O(1)$ random I/O. COLA's data migration costs $O((r-1)\log_r N)$ I/O per insert. r is usually chosen between 2 and 4.

The Sorted Array Merge Tree (SAMT) [42] is similar to COLA but performs compaction differently. Instead of eagerly merging data to have a single log structure per level, SAMT keeps up to r SSTables before merging them and moving the merged data into the next level. Therefore, a lookup costs $O(r\log_r N)$ random I/O, whereas the per-update I/O cost decreases to $O(\log_r N)$.

A few notable systems implementing a version of COLA and SAMT are HBase [45] and Cassandra [27, 44].

Algorithm 2 presents models for COLA and SAMT. Both models assume that the system uses write-ahead log files whose count is capped by the growth factor r.

```
1  // @param  L    maximum level
2  // @param  wal  write-ahead log file size
3  // @param  r    growth factor
4  // @return      write amplification
5  function estimateWA_COLA(L, wal, r) {
6    local l, j, WA, interval[], write[];
7    // mem -> log
8    WA = 1;
9    // mem -> level-1; level-l -> level-(l+1)
10   interval[0] = wal;
11   for (l = 0; l < L - 1; l++) {
12     interval[l + 1] = interval[l] * r;
13     write[l + 1] = 0;
14     for (j = 0; j < r - 1; j++)
15       write[l + 1] += merge(unique(interval[l]),
          unique(interval[l] * j));
16     WA += write[l + 1] / interval[l + 1];
17   }
18   // level-(L-1) -> level-L
19   WA += unique(∞) / interval[L - 1];
20   return WA;
21 }
22
23 function estimateWA_SAMT(L, wal, r) {
24   local l, WA, interval[], write[];
25   // mem -> log
26   WA = 1;
27   // mem -> level-1; level-l -> level-(l+1)
28   interval[0] = wal;
29   for (l = 0; l < L - 1; l++) {
30     interval[l + 1] = interval[l] * r;
31     write[l + 1] = r * unique(interval[l]);
32     WA += write[l + 1] / interval[l + 1];
33   }
34   // level-(L-1) -> level-L
35   WA += unique(∞) / interval[L - 1];
36   return WA;
37 }
```

Algorithm 2: Pseudocode of models of WA of COLA and SAMT.

In COLA, line #15 calculates the amount of writes for a level that has already accepted j merges ($0 \leq j < r - 1$). Compaction of the second-to-last level is treated specially because the last level must be large enough to hold all unique keys and has no subsequent level (line #19). The SAMT model is simpler because it defers merging the data in the same level.

C Modeling Universal Compaction

Algorithm 3 models RocksDB's universal compaction using a table-level simulation presented in Section 5. Line #15 estimates the size of a new SSTable created from insert requests. Line #26, #43, and #55 predict the outcome of SSTable merges caused of different compaction triggers.

merge_all() takes a list of (multiple) SSTable sizes and returns the expected size of the merge result (i.e., $\text{Unique}(\sum_i \text{Unique}^{-1}(\text{sizes}[i]))$).

```
// @param  wal                                write-ahead log file size
// @param  level0_file_num_compaction_trigger number of files to trigger compaction
// @param  level0_stop_writes_trigger         maximum number of files
// @param  max_size_amplification_percent     parameter for Condition 1
// @param  size_ratio                         parameter for Condition 2
// @param  tables                             list of initial SSTable sizes
// @param  num_inserts                        number of inserts to simulate
// @return                                    write amplification
function estimateWA_UC(wal, level0_file_num_compaction_trigger, level0_stop_writes_trigger,
     max_size_amplification_percent, size_ratio, tables, num_inserts) {
  local inserts, writes, done, last, start_i, last_i, i, candidate_count, candidate_size, table_size;
  inserts = writes = 0;
  while (inserts < num_inserts) {
    if (len(tables) < level0_stop_writes_trigger) {
      // a new SSTable
      table_size = unique(wal);
      writes += wal;                  // mem -> log
      writes += table_size;           // mem -> level-0
      inserts += wal;
      tables = [table_size] + tables;
    }
    // Precondition
    if (len(tables) >= level0_file_num_compaction_trigger) {
      last = len(tables) - 1;
      // Condition 1
      if (sum(tables[0...last-1]) / tables[last] > max_size_amplification_percent / 100) {
        table_size = merge_all(tables);
        tables = [table_size];
        writes += table_size;         // level-0 -> level-0
      } else {
        done = false;
        // Condition 2
        for (start_i = 0; start_i < len(tables); start_i++) {
          candidate_count = 1;
          candidate_size = tables[start_i];
          for (i = start_i + 1; i < len(tables); i++) {
            if (candidate_size * (100 + size_ratio) / 100 < tables[i])
              break;
            candidate_size += tables[i];
            candidate_count++;
          }
          if (candidate_count >= 2) {
            last_i = start_i + candidate_count - 1;
            table_size = merge_all(tables[start_i...last_i]);
            tables = tables[0...start_i-1] + [table_size] + tables[last_i+1...last]);
            writes += table_size;     // level-0 -> level-0
            done = true;
            break;
          }
        }
        // Condition 3
        if (done == false) {
          candidate_count = len(tables) - level0_file_num_compaction_trigger;
          if (candidate_count >= 2) {
            last_i = candidate_count - 1;
            table_size = merge_all(tables[0...last_i]);
            tables = [table_size] + tables[last_i+1...last];
            writes += table_size;     // level-0 -> level-0
          }
        }
      }
    }
  }
  return writes / inserts;
}
```

Algorithm 3: Pseudocode of a table-level simulation of RocksDB's universal compaction.