EasyChair Preprint

№ 6129

# On Lowering Merge Costs of an LSM Tree

Dai Hai Ton That, Mohammadsaleh Gharehdaghi, Alexander Rasin and Tanu Malik



July 21, 2021

# On Lowering Merge Costs of an LSM Tree

Dai-Hai Ton That, Mohammadsaleh Gharehdaghi, Alexander Rasin, Tanu Malik
School of Computing
DePaul University
Chicago, IL, USA
{dtonthat,mgharehd,arasin,tanu.malik}@depaul.edu

## ABSTRACT

In column stores, which ingest large amounts of data into multiple column groups, query performance deteriorates. Commercial column stores use log-structured merge (LSM) tree on projections to ingest data rapidly. LSM tree improves ingestion performance, but for column stores the sort-merge maintenance phase in an LSM tree is I/O-intensive, which slows concurrent queries and reduces overall throughput. In this paper, we present a simple heuristic approach to reduce the sorting and merging cost that arise when data is ingested in column stores. We demonstrate how a Min-Max heuristic can construct *buckets* and identify the level of sortedness in each range of data. Filled and relatively-sorted buckets are written out to disk; unfilled buckets are retained to achieve a better level of sortedness, thus avoiding the expensive sort-merge phase. We compare our Min-Max approach with LSM tree and production columnar stores using real and synthetic datasets.

## KEYWORDS

Min-Max, column-oriented, write-optimized

## 1 INTRODUCTION

Column-oriented databases are a dominant backend DBMSes for supporting business decision-making processes [6, 22]. Column-stores, unlike their row-store counterparts, store entire columns contiguously, often in compressed form. Applications using column-oriented databases, typically, coalesce columns into groups. Using column groups significantly reduces the amount of data to be read, achieving high read performance for analytic (range-query) workloads in which most queries reference a column group.

In the era of big data, applications also ingest high volume data, often, arriving at high velocity. Log-structured merge tree (LSM tree) logs incoming data in a buffer and periodically sort-merges the data [15, 18] into larger sorted runs. Typically used in wide-column NoSQL databases [2, 5, 11, 13], LSM trees are increasingly available in column-store databases for fast writes and high throughput. Each group of columns in a column-store requires storage maintenance, thus column-stores have a greater need for a write-optimized index than row-stores. However, using an LSM tree index structure, which itself has a significant write amplification in a column-store database, reduces query performance.


*SSDBM 2021, July 6-7, 2021, Tampa, Florida, USA*

ACM ISBN XXX-X-XXXXX-XXX-X.

In a commercial system, such as Vertica [14, 19], all column groups must be indexed with an LSM tree to preserve the row order in accordance with the primary key of the groups. Thus a write to a column group must coordinate with writes in other associated column groups for consistency. Queries, however, are often not uniform across column groups. Some column groups are more popular and queried more often than other less popular groups. Query performance of popular groups rapidly deteriorates due to concurrent writes across all groups. One of the most important reasons to use column-oriented databases is to divide a table into column groups, but slow write performance across groups defeats the purpose of dividing them into groups in the first place.

One naive strategy is to split an insert into multiple individual inserts on column groups; this, however, will forsake the consistency and row order between column groups. An alternate strategy is to optimize the expensive sort-merge phase of LSM trees. The key idea is that a fewer I/Os during sort-merge will lead to improved query performance and maintain consistency.

Current methods optimize this phase by adding summary structures within the buffer [4], improving when to merge [9, 10], and by measuring overlaps between buffer and on-disk data [3]. In every proposed approach, however, the sort-merge phase sorts *all* key values at periodic intervals of time. We show, analytically and experimentally, that this complete sorting of keys causes a large fraction of the I/O in an LSM tree. In column stores this increase in I/O during inserts, reduces concurrent query performance, but, more importantly, this I/O due to sorting is redundant for answering analytical workloads. Our strategy, thus, is to compensate for the sort-merge phase of an LSM tree by approximately sorting the incoming data. To approximately sort we select the *most compact bucket* in the buffer to write to disk. The advantage of writing compact buckets to disk is that unlike LSM tree we do not need to wait for periodic intervals to merge and can write out incoming data as fast as it arrives; the disadvantage is that buckets may be written out that eventually requires a merge.

In this paper, we present *Min-Max* approach, which caches data in a buffer, and then evicts the most compact bucket. The most compact bucket is decided based on a sliding window method that computes the smallest *[Min, Max]* range over all windows. Written buckets on disk are indexed with an interval B-tree to improve query performance and avoid merge. We show that for real datasets with stable distributions, such a strategy compensates for the sort-merge cost of an LSM tree, but still maintains good query performance.

The rest of this paper is organized as follows. We present an example in Section 2. Section 3 analyses the cost of merge in LSM trees. We introduce the *Min-Max* approach in Section 4. We show the experiments in Section 5, and discuss about the related work in Section 6. Finally, we conclude the paper in Section 7.

| Order | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Key | 9 | 10 | 3 | 18 | 1 | 13 | 20 | 24 | 12 | 4 | 2 | 7 | 19 | 30 | 26 | 14 | 1 | 29 | 8 | 4 | 15 | 25 | 18 | 9 | 2 | 8 | 11 | 6 | 16 | 5 | 12 | 28 |

**Figure 1: Sample data $D$ to insert.**

## 2 AN EXAMPLE: LSM TREE VS *MIN-MAX*

In this section, we present an example to contrast the LSM Tree and the *Min-Max* approach. There are two types of LSM trees, leveled [15] (proactive merging) and tiered (delayed merging) [9]. We illustrate the more commonly implemented leveled LSM tree in this example; the recently proposed tiered LSM tree example is presented in Appendix 8.1. However, we compare the costs of both tiered and leveled LSM trees throughout this paper.

Figure 1 lists a sequence $D$ of 32 incoming tuples (a tuple consists of a key and data, although only the key is shown) and the order in which they arrive.

When a buffer or a level is full, LSM tree merges sorted runs of tuples into larger sorted runs either in the same level or in the higher level of the tree. Figure 2a shows the state of the leveled LSM tree before and after a merge. The data arrives at the in-memory buffer level $L0$ with a size of $B = 4$ (4 tuples). In this example, LSM tree is configured with a level ratio of $T = 3$ (representing the maximum size ratio between levels in LSM tree). If a level is full, LSM tree recursively merges data runs at a higher level.

Figure 2a illustrates five merging steps performed by the LSM tree approach. The first four tuples $(9, 10, 3, 18)$ are already sorted in memory and written to disk at $L1$ (see row 1). In row 1, the sorted buffer $(1, 13, 20, 24)$ is merged with $L1$ sorted run $(3, 9, 10, 18)$ to write out the sorted sequence of eight tuples $(1, 3, 9, 10, 13, 18, 20, 24)$ at $L1$. A recursive merge is shown in row 4, in which a merge between the levels $L0$ and $L1$ leads to a full $L1$ level and is thus followed by a merge between the levels $L1$ and $L2$. The ratio of data between any two levels is always maintained at a ratio of 3.

In cases where data is mostly sorted, an LSM tree chooses to append sorted runs instead of merging them. For example, in row 5, a merge between levels $L0$ and $L1$ can be replaced by appending levels $L0$ and $L1$ instead. In this case, skipping the merge causes a small reduction in data sortedness (only tuple 5 is out of order). However, the likelihood of such near-sorted alignment depends on the distribution of the data. Although LSM tree can benefit from such distribution, it is not distribution-aware.

The key idea behind the *Min-Max* approach is to determine a window which is the most sorted with respect to the current buffer. Given a window size, which is less than the buffer size, in *Min-Max*, we determine the most sorted window by computing the difference between maximum and minimum values within a window and finding the minimum value over several sliding windows. We illustrate this over the data in Figure 1 considering the same buffer capacity of $B = 4$ and a window size of 2.

For example, in the first row buffer contains tuples $(9, 10, 3, 8)$ and then sort those tuples $(3, 9, 10, 18)$. The window $[9, 10]$ is the most sorted amongst all windows, it is selected and written to disk as a bucket since its value range is smallest. Next, the tuples with keys 1 and 13 will be loaded in the row 2. Every time a window worth of data fills up, a new bucket is written to disk and cleared from the buffer. The last column in Figure 2b summarizes the buckets written during the ingestion of input data.

| Before merge | After merge |
|---|---|
| **L0** 1 13 20 24<br>L1 3 9 10 18 | 1 3 9 10 13 18 20 24 |
| **L0** 2 4 7 12<br>L1 1 3 9 10 13 18 20 24 | 1 2 3 4 7 9 10 12 13 18 20 24 |
| **L0** 1 4 8 29<br>L1 14 19 26 30<br>L2 1 2 3 4 7 9 10 12 13 18 20 24 | 1 4 8 14 19 26 29 30<br>1 2 3 4 7 9 10 12 13 18 20 24 |
| **L0** 9 15 18 25<br>**L1** 1 4 8 14 19 26 29 30<br>L2 1 2 3 4 7 9 10 12 13 18 20 24 | 1 1 2 3 4 4 7 8 9 9 10 12<br>13 14 15 18 18 19 20 24 25 26 29 30 |
| **L0** 5 12 16 28<br>L1 2 6 8 11<br>L2 1 1 2 3 4 4 7 8 9 9 10 12<br>13 14 15 18 18 19 20 24 25 26 29 30 | 2 5 6 8 11 12 16 28<br>1 1 2 3 4 4 7 8 9 9 10 12<br>13 14 15 18 18 19 20 24 25 26 29 30 |

**(a) Inserting the sample data $D$ using leveled LSM trees.**

| Buffer | Buffer after sort | Buffer after evict | Bucket written to disk |
|---|---|---|---|
| 9 10 3 18 | 3 **9** **10** 18 | 3 18 | [9, 10] |
| 3 1 13 18 | **1** **3** 13 18 | 13 18 | [1, 3] |
| 20 24 13 18 | 13 **18** **20** 24 | 13 24 | [18, 20] |
| 13 12 4 24 | 4 **12** **13** 24 | 4 24 | [12, 13] |
| 4 2 7 24 | **2** **4** 7 24 | 7 24 | [2, 4] |
| 19 30 7 24 | 7 **19** **24** 30 | 7 30 | [19, 24] |
| 7 26 14 30 | 7 14 **26** **30** | 7 14 | [26, 30] |
| 7 14 1 29 | **1** **7** 14 29 | 14 29 | [1, 7] |
| 8 4 14 29 | **4** **8** 14 29 | 14 29 | [4, 8] |
| 15 25 14 29 | **14** **15** 25 29 | 25 29 | [14, 15] |
| 18 9 25 29 | 9 18 **25** **29** | 9 18 | [25, 29] |
| 9 18 2 8 | 2 **8** **9** 18 | 2 18 | [8, 9] |
| 2 11 6 18 | **2** **6** 11 18 | 11 18 | [2, 6] |
| 16 5 11 18 | 5 11 **16** **18** | 5 11 | [16, 18] |
| 5 11 12 28 | 5 **11** **12** 28 | 5 28 | [11, 12] |

**(b) Inserting the sample data $D$ using *Min-Max*.**

**Figure 2: The behavior of leveled LSM tree and Min-Max.**

In this particular example, unlike LSM Tree, *Min-Max* runs do not require a merge as the distribution already writes nearly-sorted buckets. Table 1 shows the cost of the LSM-Tree and the *Min-Max* in number of merges and number of I/O operations. Leveled-LSM requires 6 merges with 36 writes and 20 reads; Tiered-LSM requires 2 merges with 28 writes and 8 reads. Meanwhile, *Min-Max* does not require any merges. The number of writes is equal to the total number of written buckets ($32/2 = 16$ buckets)

Next, we examine read query performance using key-range queries. Without loss of generality queries $Q1$, $Q2$ and $Q3$ have key-ranges $[10, 12]$, $[19, 20]$ and $[14, 20]$, respectively. Table 2 presents number of I/Os needed for each query. Tiered-LSM requires more

**Table 1: Merge cost with different methods**

| | Leveled-LSM | Tiered-LSM | Min-Max |
|---|---|---|---|
| #of merges | 6 | 2 | 0 |
| #of I/Os (Writes) | 36 I/O | 28 I/O | 16 I/O |
| #of I/Os (Reads) | 20 I/O | 16 I/O | 0 I/O |

I/O than both Leveled-LSM and *Min-Max* in all three queries. Leveled-LSM exhibits the best performance, but *Min-Max* query performance is equivalent for $Q1$, $Q2$ and $Q3$.

**Table 2: Query cost with different methods**

| Range Query | Leveled-LSM | Tiered-LSM | Min-Max |
|---|---|---|---|
| $Q1 : [19, 20]$ | 2 I/O | 2 I/O | 2 I/O |
| $Q2 : [11, 12]$ | 2 I/O | 3 I/O | 2 I/O |
| $Q3 : [14, 20]$ | 4 I/O | 5 I/O | 4 I/O |

Despite lower merge and I/O costs, there are several challenges for the *Min-Max* approach. First, we need to efficiently determine the most sorted bucket. In this example we used a sliding window approach, but in general, sliding a window, data tuple at a time is computationally expensive. Second, it is equally likely that a bucket which is most sorted currently is considered poorly sorted as new data arrives. Third, the function *Min-Max* itself might be a poor estimator of sortedness depending on the attribute. In Section 4 we propose an efficient index using *Min-Max*, which uses a reverse index to address the first issue and an interval B-tree to address the second issue. We leave the analysis of the quality of the $|max - min|$ function as part of future work.

## 3 THE MERGE COSTS IN LSM TREE

In the previous example we compared the merge cost in LSM trees with a *Min-Max* bucket eviction approach on a small example data. In this section, we mathematically determine the number of merges in a LSM tree, a quantity which influences the total I/O cost of LSM trees.

Table 3 presents the parameters governing the merge cost in an LSM tree. Given $N$ data entries and a buffer of size $B$, the total number of buffers that will be ingested by an LSM tree is $M = \lfloor N/B \rfloor$. Given $T$ as the ratio of data entries between two consecutive levels, the total number of levels in an LSM tree is $L = \lfloor log_T M \rfloor + 1$.

Number of merges is recursively defined based on $L$, the number of levels in an LSM tree and $M$, the total number of buffers. To compute the number of merges, we observe that $\frac{M}{T^i}$ is the number of buffers at any $L-i$ level. We consider the floor of $\frac{M}{T^i}$ because when the last buffer arrives, *i.e.,* the $\left\lceil \frac{M}{T^i} \right\rceil^{th}$ buffer, it is nto accomodated in the current level, but instead the merge with the higher level happens. In tiering merges happen whenever this last buffer arrives and so number of times merges happen is simply the sum of number of buffers at any $L-i$ level (Equation 2). In leveling, a merge happens every time data from the previous level is merged with level *i i.e.,* $\left\lfloor \frac{M}{T^{i-1}} \right\rfloor$. However, this factor must be discounted by $\left\lceil \frac{M}{T^i} \right\rceil$ which is the number of times level $i$ is empty, and therefore no merge is needed (Equation 1).

**Table 3: Notations used in this paper**

| Parameter | Description |
|---|---|
| $N$ | The total number of data entries |
| $B$ | The buffer size (Level $L_0$) |
| $M = \lfloor N/B \rfloor$ | The total number of buffers |
| $T$ | The ratio between two consecutive levels |
| $L = \lceil log_T M \rceil + 1$ | The total number of levels |
| $m_{level}, m_{tier}$ | The total number of merges in Leveled/Tiered-LSM |

$$m_{level} = \sum_{i=1}^{L-1} \left( \left\lfloor \frac{M}{T^{i-1}} \right\rfloor - \left\lceil \frac{M}{T^i} \right\rceil \right) \quad (1)$$

$$m_{tier} = \sum_{i=1}^{L-1} \left\lfloor \frac{M}{T^i} \right\rfloor \quad (2)$$

The number of I/Os in level and tiered LSM tree are proportional to the number of merges respectively for each merging policy. Appendix 8 presents how read and write costs are related to number of merges.

## 4 MIN-MAX

The basic idea in *Min-Max* is to buffer data as in LSM tree but determine data bucket(s) to write to disk, which will lead to least merge cost. To determine this bucket, *Min-Max* uses a sliding window approach in which it computes the difference between the maximum and minimum values in the window. The window which has the smallest difference value represents the most sorted *bucket* amongst the window choices in the buffer, which will be written to the disk. Therefore, *Min-Max* evicts this window as a bucket. The min-max approach is an online approach to determining the most relevant window, without assuming anything about the incoming data distribution. A formal proof about the competitive ratio of this approach is beyond the scope of the current paper.

Algorithm 1 presents the *Min-Max* approach. *Min-Max* maintains a sorted cache $C$ with a window $W$. The cache accepts windows(s) worth of data at a time, and maintains data according to key-order (Line 2). If cache size reaches its limit, window(s) are evicted (Lines 4-6) based on the smallest $|Max - Min|$ value. Sliding the window on a per eviction basis can be computationally expensive. In our implementation, we maintain a reverse index pointing to windows where data was inserted so that windows which do not have incoming data need not be computed for $|Max - Min|$ value again. In our implementation, we also set the window size as equal as the I/O size in LSM tree (bucket size).

The *Min-Max* approach does not completely sort the data on disk as in an LSM tree. There are two choices for Min-Max to carry maintenance tasks: one, sort the data at periodic intervals, or alternatively let it remain unsorted and use indexes to ameliorate the cost of computing queries. We use an interval B-tree (IB-tree) to index the written out buckets. In an IB-tree, buckets are indexed

**Algorithm 1:** Min-Max

```
1 Min-Max ():
    input: Sorted cache C, cache size Size
    event: Every time a data tuple D is inserted
2   Find a position in sorted cache C to insert D
3   Size += 1
4   if (Size == CACHE_LIMIT) then
5       Evict-Window(C, Size)
6       Size -= WINDOW_SIZE
```

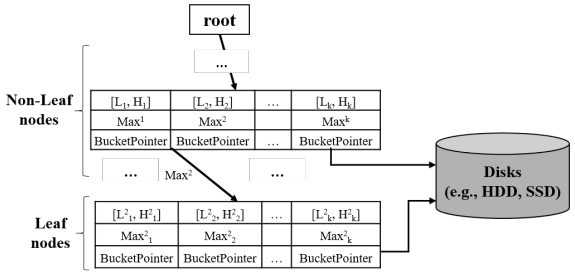


**Figure 3: The data structure of IB-Tree.**

using their min-max key ranges. The leaves and internal nodes of this tree store interval ranges sorted on the low value of the interval ranges. The range of values $[L, H]$ of an internal node of a tree is the key range of the low key boundary of buckets in its sub-tree. Meanwhile, $Max_i$ of a node indicates the highest value of the high key boundary of buckets in its sub-tree. Using this max value bounds the search. The size of node is a factor of page size (i.e., $k*4KB$) so that all nodes in IB-Tree can be efficiently stored on disk. The node's entry in a IB-Tree keeps its data (*i.e.,* $BucketPointer$) [1] in all types of nodes (leaf or non-leaf nodes).

There are a few of advantages of using IB-Tree. First, IB-tree indexes on the bucket (i.e., group of tuples) instead of projection tuple, its size is reduced by the number of tuples in a bucket. Second, it reduces the time to search for a bucket from $O(N)$ to $O(logN)$ [1, 7] (similar to the search cost in B-Tree). Third, IB-Tree is designed for either full or partial loading in memory.

# 5 EXPERIMENT

**Systems** Experiments were conducted on a desktop computer with an Intel Core i7-3770 3.4Ghz (8 cores), 8GB of main memory, 1 TB SATA HDD and 256 GB Intel SATA SSD 600p, and Ubuntu 16.04 64-bit operating system.

**Dataset & Queries** Experiments used real-world data from the New York City Taxi (NYC) dataset [17] with nearly 148 million of data records. Three different projections were deployed for each indexing approach: $Projection1$ (29 columns, indexed on $ID$), $Projection2$ (4 columns, indexed on $trip\_distance$) and $Projection3$ (5 columns, indexed on $total\_amount$). Table 4 summarizes the set of read queries used to measure performance. Key range refers to a delta in the indexed attribute value, *i.e.,* range between $X$ and $X+ < keyrange >$, where $X$ refers to the value of the indexed attribute ($trip\_distance$, $ID$, or $total\_amount$). Selectivity refers to the ratio between the number of query result tuples to the total database tuples. Our implementation is available at [12]

Note that the cost of IB-Tree in our implementation is small ($\leq 100KB$) and considered as negligible (IB-Tree only keeps the pointer of buckets and their key ranges).

**Table 4: Query Range and Selectivity.**

| Range Query | Key Range | Selectivity |
|---|---|---|
| Q1 | 0.003 | 0.003 |
| Q2 | 0.005 | 0.006 |
| Q3 | 0.020 | 0.015 |
| Q4 | 0.030 | 0.026 |
| Q5 | 0.055 | 0.050 |

## 5.1 The Merge costs of LSM tree in Columnar databases

Columnar databases store data tables by column where each column is stored separately. This allows a query to access that precise data that it needs. In general, each column can be stored separately, but this leads to high tuple reconstruction cost. Column grouping (or projection in C-Store [19] or Vertica [14]) is one way to reduce the tuple reconstruction cost. The idea is to group a subset of columns together, to benefit query operations that accesses all these columns. This group of columns is called *projection* in C-Store [19] or Vertica [14]. Column stores trade storage for improved tuple reconstruction cost and query access. For instance, it is possible to replicate columns across projections as well as support a *superprojection* with all columns. We assume a simplified columnar store model in which there are partitioned projections with no replication of columns across projections, and no superprojections.

Figure 4 shows both the number of merges (bar charts) and the amount of read and write data (lines) of different data-set sizes. The y-axis on the left presents the amount of read and write data of the lines; meanwhile the y-axis on the right presents the total number of merges of the bar charts. Clearly tiered LSM causes fewer merges than levelled LSM but in both the cases the number is dominated by the constant factors that are multiplied on a per level basis. These constants play a significant role in a column database as shown in Figure 4. In this experiment we measure the total number of merges with different size of data in a columnar store. The column store has two configurations: 5 projections with each projection has 3 columns, and 1 projection with all columns. As shown in this Figure, while the number of merges of single LSM Tree is quite reasonable, those of multiple LSM Trees are multiplied by the factor of the number of projections and the $T$ factor. The amount of read and write data of those candidates are proportional to the number of merges and number of projections. In contrast, *Min-Max* has negligible merges.

## 5.2 Loading Costs

Figures 5 summarizes the loading costs for the *Min-Max* and LSM tree approaches on both HDD with the different data-set sizes. The

[1] $BucketPointer$ points to physical bucket stored on disk (composed of $fileID$ and $offset$).

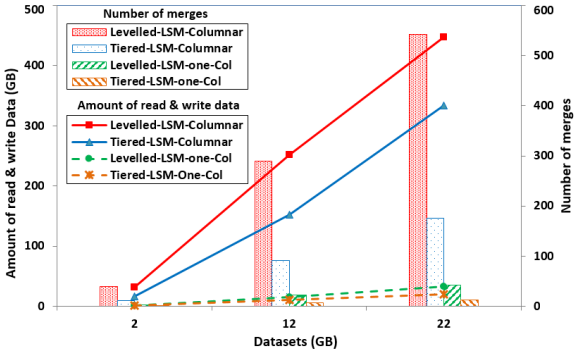


**Figure 4: The total number of merges during data-loading with tiered and levelled LSM trees. Multiple LSM trees on columnar stores, single LSM tree on one column and *Min-Max*. The columnar has 5 projections each of projection has 3 columns; whereas on-column keep the data in only one projection (all columns).**

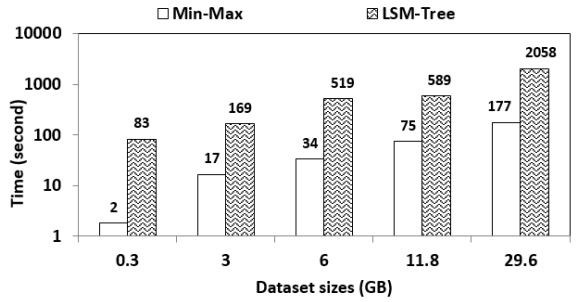


**Figure 5: Loading runtimes on HDD with different dataset sizes.**

LSM tree runtimes include the costs for reorganizing data in all projections. It's worth emphasizing that the results of LSM tree are for one-time bulk-load ingestion, whereas continuous data ingestion or incremental data loading will likely lead to additional clustering/optimization overheads. Meanwhile, *Min-Max* is an on-line process that does not incur any additional overheads for incremental data loading.

We observed that *Min-Max* significantly improved data load time for the column-oriented DBMS on an HDD; it was, on average, 17 times faster than LSM tree across all tested data-sets.

## 5.3 Read Query I/O

We evaluate our method by using the list of queries mentioned in Table 4. Figure 6 presents the total number of accessed I/O of *Min-Max* and LSM tree. As shown in this figure, LSM tree have better query performance in all five queries. *Min-Max* takes more I/O than LSM tree. On average, *Min-Max* was 49.7% percent higher LSM tree. This is obvious since data in LSM tree are regularly ordered in all levels.

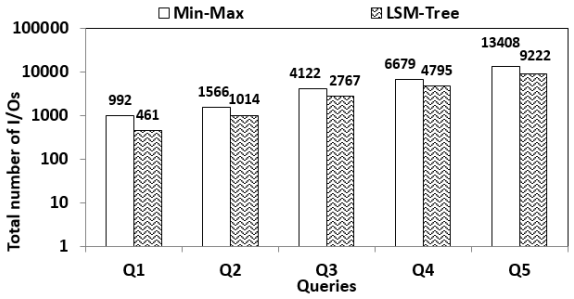


**Figure 6: Total number of I/Os of different queries.**

# 6 RELATED WORK

While many DBMSes including key-value stores, relational DBMSes or columnar stores suffer from poor write performance, the log-structure merge tree (LSM tree) [15, 18] is a common solution for this problem. The main idea is to transform small random writes into large sequential writes by buffering and reorganizing data in a large buffer before flushing them to disk at a batch. For example, LevelDB [11], BigTable [5], HBase [2], Cassandra [13] are some key-value stores apply LSM tree, while MySQL [16] and SQLite4 [8] are relational DBMSes that support LSM tree indexing. Other enhanced variations of LSM tree used in Monkey [9] and Dostoevsky [10] to further improve the DBMS performance by using BloomFilter and changing merging policies. However, the main drawback of LSM tree, i.e., large write amplification [2] still remains.

In [20, 21], we introduced the primary idea of approximate sorting to lower maintenance costs. However, these index structures do not exploit available buffer cache to sort data in main memory. Using main memory efficiently for improved write performance is explored in TRIAD [3]. TRIAD (i) keeps hot-entries longer in the main memory; (ii) optimizes the tiered merging policy by allowing for overlaps between runs in a level; and (iii) improves the efficiency of writes in a commit-log. TRIAD, however, still relies on LSM tree at the back end. It also does not address the query lookup performance. In this work we have suggested an alternate way for compensating the merge cost while efficiently using main memory.

# 7 CONCLUSION

In this paper, we presented a simple, yet efficient strategy (*Min-Max*) for lowering the maintenance costs of an LSM-tree. Often maintenance costs increase with the size of data. In *Min-Max* approach we show that if distribution is stable the maintenance cost is significantly lowered by observing distribution and using simple statistics for writing data to disk. We have shown this as a viable method in column stores, which have a high cost of using LSM trees.

# ACKNOWLEDGMENTS

This work is supported by National Science Foundation under grants ICER-1639759 and ICER-1661918.

[2]Write amplification is the ratio of total write IO performed by the DBMS to the total data in the DBMS. High write amplification increases the loading cost on storage devices.

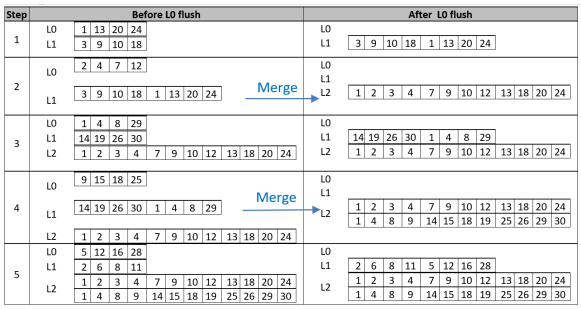


**Figure 7: Data loading with tired and tiered LSM-Trees.**

# REFERENCES


[1] Chuan-Heng Ang and Kok-Phuang Tan. 1995. The Interval B-tree. *Inf. Process. Lett.* 53, 2 (Jan. 1995), 85–89. https://doi.org/10.1016/0020-0190(94)00176-Y

[2] Apache. [n.d.]. HBase. https://github.com/google/leveldb. [Online; accessed 25-Mar-2019].

[3] Oana Balmau and et. al. 2017. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *USENIX ATC 17*. USENIX Association, Santa Clara, CA.

[4] Edward Bortnikov and et. al. 2018. Accordion: Better Memory Organization for LSM Key-value Stores. *Proc. VLDB Endow.* 11, 12 (Aug. 2018), 13. https://doi.org/10.14778/3229863.3229873

[5] Fay Chang and et. al. 2006. Bigtable: A Distributed Storage System for Structured Data. In *OSDI'06* (Seattle, WA). USENIX Association, Berkeley, CA, USA, 1.

[6] Columnar Database. 2019. Columnar Database: A Smart choice for data warehouses. https://www.columnardatabase.com/. [Online; accessed 25-Mar-2019].

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[8] D. Richard Hipp. [n.d.]. SQLite4. https://sqlite.org/src4/doc/trunk/www/index.wiki. [Online; accessed 25-Mar-2019].

[9] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *SIGMOD'17* (Chicago, Illinois, USA). ACM, New York, NY, USA, 16.

[10] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *SIGMOD '18* (Houston, TX, USA). ACM, New York, NY, USA, 16.

[11] Google. [n.d.]. LevelDB. https://github.com/google/leveldb. [Online; accessed 25-Mar-2019].

[12] DePaul Database Group. 2021. Indexing implementation. https://github.com/TonHai1111/DB_Implementation/tree/hai-data-distribution/Simulation/Distribution/tools. [Online; accessed 8-Jun-2021].

[13] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 6.

[14] Andrew Lamb and et. al. 2012. The Vertica Analytic Database: C-store 7 Years Later. *Proc. VLDB Endow.* 5, 12 (Aug. 2012), 12.

[15] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-structured Merge-tree (LSM-tree). *Acta Inf.* 33, 4 (June 1996), 351–385. https://doi.org/10.1007/s002360050048

[16] Oracle. [n.d.]. MySQL. https://github.com/mysql/mysql-server. [Online; accessed 25-Mar-2019].

[17] Todd W. Schneider. 2016. Unified New York City Taxi and Uber data. https://github.com/toddwschneider/nyc-taxi-data. [Online; accessed 18-Aug-2017].

[18] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: A General Purpose Log Structured Merge Tree. In *SIGMOD '12* (Scottsdale, Arizona, USA). ACM, New York, NY, USA, 12.

[19] Mike Stonebraker and et. al. 2005. C-store: A Column-oriented DBMS. In *VLDB'05* (Trondheim, Norway). 12.

[20] Dai Hai Ton That, James Wagner, Alexander Rasin, and Tanu Malik. 2018. PLI$^+$: Efficient Clustering of Cloud Databases. *Distributed and Parallel Databases* (2018). in the third round of DAPD.

[21] James Wagner, Alexander Rasin, Dai Hai Ton That, and Tanu Malik. 2017. PLI: Augmenting Live Databases with Custom Clustered Indexes. In *SSDBM'17* (Chicago, IL, USA). ACM, New York, NY, USA, Article 36, 6 pages. https://doi.org/10.1145/3085504.3085582

[22] Wikipedia. 2019. Business Intelligence. https://en.wikipedia.org/wiki/Business_intelligence. [Online; accessed 25-June-2019].


# 8 APPENDIX

## 8.1 Example: Tiered LSM-Tree

The Figure 7 shows the Tiered LSM-Tree behavior during the data ingestion (Sample data D - Figure 1).

## 8.2 LSM-Tree merge costs

The the number of writes and reads of Tiered-LSM and Leveled-LSM are presented in Formulas ( 3 and 5) and ( 4 and 6).

$$n_w^R = \sum_{i=1}^{L-1} \left( \left\lfloor \frac{M}{T^{i-1}} \right\rfloor - \left\lfloor \frac{M}{T^i} \right\rfloor \right) \frac{BT^{i-1}}{P} \tag{3}$$

Proof. Formula 3 can be proved as follows. The Tier-LSM has L levels (*i.e.*, $[0; L-1]$). Since Level 0 is the head of LSM resided main-memory we only consider levels in $[1; L-1]$. Given $M = \lfloor N/B \rfloor$, at level $L = 1$, we have $M$ times level $L = 0$ get full and flush a run to level $L = 1$. Among these $M$ times, there are $\lfloor \frac{M}{T} \rfloor$ times level $L = 1$ get full and all data will be written to level $L = 2$ instead of level $L = 1$. In level $L = 1$, $run = B$. Therefore, we have: $\frac{run}{P} * (M - \lfloor \frac{M}{T} \rfloor) = \frac{B}{P} * (M - \lfloor \frac{M}{T} \rfloor)$ write IO. At level $L = 2$: we have $\lfloor \frac{M}{T} \rfloor$ times level $L = 1$ get full and flush a run to level $L = 2$. Among these $\lfloor \frac{M}{T} \rfloor$ times, there are $\lfloor \frac{M}{T^2} \rfloor$ times level $L = 2$ get full and all data will be written to level $L = 3$ instead of level $L = 2$. In level $L = 2$, $run = BT$. Therefore, we have: $\frac{run}{P} * (\lfloor \frac{M}{T} \rfloor - \lfloor \frac{M}{T^2} \rfloor) = \frac{BT}{P} * (\lfloor \frac{M}{T} \rfloor - \lfloor \frac{M}{T^2} \rfloor)$ write IO. At Level $L = i$ $(1 \leq i \leq L-1)$: we have $\lfloor \frac{M}{T^{i-1}} \rfloor$ times level $L = i-1$ get full and flush a run to level $L = i$. Among these $\lfloor \frac{M}{T^{i-1}} \rfloor$ times, there are $\lfloor \frac{M}{T^i} \rfloor$ times level $L = i$ get full and all data will be written to level $L = i+1$ instead of level $L = i$. In level $L = i$, $run = BT^{i-1}$. Therefore, we have: $run * (\lfloor \frac{M}{T^{i-1}} \rfloor - \lfloor \frac{M}{T^i} \rfloor) = \frac{BT^{i-1}}{P} * (\lfloor \frac{M}{T^{i-1}} \rfloor - \lfloor \frac{M}{T^i} \rfloor)$ write IO. Therefore the total number of writes:
$n_w^R = \sum_{i=1}^{L-1} (\lfloor \frac{M}{T^{i-1}} \rfloor - \lfloor \frac{M}{T^i} \rfloor) * \frac{BT^{i-1}}{P}$.

□

$$n_w^L = \sum_{i=1}^{L-1} \left[ \left\lfloor \frac{M}{T^i} \right\rfloor \frac{(T+1)T^iB}{2P} + \left( \left\lfloor \frac{M}{T^{i-1}} \right\rfloor \%T \right) \left( \left\lfloor \frac{M}{T^{i-1}} \right\rfloor \%T + 1 \right) \frac{T^{i-1}B}{2P} \right] \tag{4}$$

$$n_r^R = \sum_{i=1}^{L-1} \left[ \left\lfloor \frac{M}{T^i} \right\rfloor \frac{(T-1)B}{P} \right] = m^T \left[ \frac{(T-1)B}{P} \right] \tag{5}$$

$$n_r^V = \sum_{i=1}^{L-1} \left[ \left\lfloor \frac{M}{T_i} \right\rfloor \frac{(T-1)T^iB}{2P} + \left( \left\lfloor \frac{M}{T^{i-1}} \right\rfloor \%T \right) \left( \left\lfloor \frac{M}{T^{i-1}} \right\rfloor \%T - 1 \right) \frac{T^{i-1}B}{2P} \right] \tag{6}$$

Similar methods can be applied to have the total number of IOs (write) in Leveled-LSM (Formulas 4) and (read) for Tiered-LSM and Leveled-LSM shown in Formulas 5 and 6.