LDI: Learned Distribution Index for Column Stores

Dai-Hai Ton That*
University of Alabama, Huntsville
daihai.tonthat@uah.edu

Mohammadsaleh Gharehdaghi*, Alexander Rasin, Tanu Malik
DePaul University
{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 improves ingestion performance, but in column stores the sort-merge phase is I/O-intensive, which slows concurrent queries and reduces overall throughput. In this paper, we aim to reduce the sorting and merging cost that arise when data is ingested in column stores. We present LDI, a learned distribution index for column stores. LDI learns a frequency-based data distribution and constructs a *bucket* worth of data based on the learned distribution. Filled buckets that conform to the distribution are written out to disk; unfilled buckets are retained to achieve the desired level of sortedness, thus avoiding the expensive sort-merge phase. We present an algorithm to learn and adapt to distributions, and a robust implementation that takes advantage of disk parallelism. We compare LDI with LSM and production columnar stores using real and synthetic datasets.**

I. Introduction

Column-oriented databases are a dominant backend DBMSes for supporting business decision-making processes [1], [2]. 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 [3], [4] into larger sorted runs. Typically used in wide-column NoSQL databases [5]–[8], LSM-trees are increasingly available in column-store databases for fast writes and high throughput, such as Vertica [9], [10]. 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 index structure, which itself has a significant write amplification[1] in a column-store database can also reduce query performance.

Current methods optimize this phase by adding summary structures within the buffer [11], improving strategies for merge [12], [13], and by measuring overlaps between buffer and on-disk data [14]. In every proposed approach, however, the sort-merge phase sorts *all* key values at periodic time intervals. We show, analytically and experimentally, that this complete sorting of keys causes a large fraction of the I/O cost 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 eliminates the sort-merge phase of an LSM-tree and sorts the incoming data *approximately* instead. The advantage of approximate sorting is that unlike LSM 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 we must know in advance if the data being written out is *sufficiently* sorted. A similar approach has been studied by using Min-Max windows [15] to bucketize incoming data. However, this method cannot learn different types of incoming data distribution, and thus it's still suffering low query performance.

In this paper, we present LDI, a low cost index for column stores, which logs data based on a learned distribution of data. If the incoming data conforms to the learned distribution, and the distribution remains stable, no further sorting will be needed for logged disk blocks. If the incoming data does not conform to the distribution, disk blocks will be approximately sorted and will need to be reorganized later. We show that for real datasets, such strategy localizes the reorganization instead of incurrcing the full cost of a sort-merge as in an LSM-tree.

LDI constructs a distribution similar to a dynamic histogram where it accumulates incoming data distribution every time window t. We present a learning algorithm that decides when to adjust the intervals based on incoming data, and show how interval counts can be maintained incrementally. LDI has the advantage of writing disk blocks continuously as data arrives. We present an I/O handler that takes advantage of in-built parallelism in HDD and SSD storage devices to support such a strategy. Finally, we present an extensive set of experimental results comparing LDI with LSM and commercial column databases, using both real and synthetic datasets.

The rest of this paper is organized as follows. We present an example in Section II. We introduce learned distribution index structure (LDI) in Section III, and its implementation details in Section IV. We show the experiments in Section V, and discuss about the related work in Section VI.

II. An Example: the behavior of LSM and LDI

We illustrate the difference between LSM-Tree and LDI approach through an example in this section. There are two types of LSM trees, leveled (proactive merging) and tiered (delayed merging) [12], [16]. We select leveled LSM tree in this example; tiered LSM tree example is presented in Appendix VIII-A.

*Most of the work done while the authors worked at DePaul University.

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

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

Fig. 1: Sample data D to insert.

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

Step	Before Merge	After Merge
1	L0: 1 13 20 24 L1: 3 9 10 18	1 3 9 10 13 18 20 24
2	L0: 2 4 7 12 L1: 1 3 9 10 13 18 20 24 L2:	1 2 3 4 7 9 10 12 13 18 20 24
3	L0: 1 4 8 29 L1: 14 19 26 30 L2: 1 2 3 4 7 9 10 12 13 18 20 24	1 4 8 14 19 26 29 30 1 2 3 4 7 9 10 12 13 18 20 24
4	L0: 9 15 18 25 L1: 1 4 8 14 19 26 29 30 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 13 14 15 18 18 19 20 24 25 26 29 30
5	L0: 5 12 16 28 L1: 2 6 8 11 L2: 1 1 2 3 4 4 7 8 9 9 10 12 13 14 15 18 18 19 20 24 25 26 29 30	2 5 6 8 11 12 16 28 1 1 2 3 4 4 7 8 9 9 10 12 13 14 15 18 18 19 20 24 25 26 29 30

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

Step	Buffer	Leaf intervals [1÷2]	[3÷6]	[7÷12]	[13÷19]	[20÷30]	Buckets written to disk
1	**9 10 3 18**		(3)	(9,10)	(18)		[9,10]
2	**1 13** 3 18	(1)	(3)		(18,13)		[18,13]
3	1 **20** 3 **24**	(1)	(3)			(20,24)	[20,24]
4	1 **12** 3 **4**	(1)	(3,4)	(12)			[3,4]
5	1 12 **2** **7**	(1,2)		(7,12)			[1,2];[7,12]
6	**19 30 26 14**				(14,19)	(26,30)	[14,19];[26,30]
7	**1 29 8 4**	(1)	(4)	(8)		(29)	**[1,4]**
8	**15** 29 8 **25**			(8)	(15)	(29,25)	[29,25]
9	15 **18** 8 **9**			(8,9)	(15,18)		[8,9];[15,18]
10	**2 8 11 6**	(2)	(6)	(8,11)			[8,11]
11	2 **16 5** 6	(2)	(6,5)		(16)		[6,5]
12	2 16 **12 28**	(2)		(12)	(16)	(28)	**[12,16]**
13	2	(2)				(28)	

(b) Inserting the sample data D using LDI.

Fig. 2: The behavior of LSM and LDI.

Figure 2a illustrates five merging steps performed by the LSM approach. The first four tuples $(9, 10, 3, 18)$ are already sorted in memory and written to disk at $L1$ before step 1. In step 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 step can trigger multiple consecutive merges: for example, in step 4, a merge between levels $L0$ and $L1$ leads to a full $L1$ level and is thus followed by a merge between levels $L1$ and $L2$. The ratio of data between levels is always maintained to be 3.

When data is mostly sorted, LSM tree can choose to append sorted runs instead of merging them. For example, in step 5, a merge between levels $L0$ and $L1$ can be replaced by appending $L0$ 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 can benefit from such distribution, it is not distribution-aware.

Intuitively, LDI designs the buffering process based on data distribution to ensure that data runs are mostly sorted and can be appended without a merging cost. Figure 2b illustrates LDI behavior for the data in Figure 1 with the same main buffer capacity $B = 4$. Leaf ranges are skewed based on the input distribution. Each leaf bucket contains pointers to tuples stored in the buffer. For example in the first row buffer contains tuples $(9, 10, 3, 8)$ and the interval $[3 \div 6]$ points to tuple 3, while $[13 \div 18]$ interval contains pointers for tuples 9 and 10.

The maximum number of pointers in a leaf interval is n (in this example $n = 2$, and a bucket can hold 2 tuples). Every time an interval fills up, a new bucket is written to disk and cleared from the buffer. For instance, at step 1, the leaf interval $[7 \div 12]$ has two pointers (9 and 10). As a result, a new bucket $[9, 10]$ is created and written to disk. Tuples 9 and 10 are removed from the buffer and from the leaf storage. The last row in Figure 2b summarizes the buckets written during the ingestion of input data.

Unlike LSM-Tree, LDI runs do not require a merge as the distribution already writes nearly-sorted buckets. Table I shows the insertion cost of LSM-Tree and LDI in number of merges and the 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, LDI does not require any merges. The number of writes is equal to the total number of written buckets ($32/2 = 16$ buckets). It's worth to emphasize that both LSM-Tree and LDI use their data structures for handling data ingestion and organizing data on disk. The data layout and its indexing structure will determine the performance of query.

TABLE I: Merge cost with different methods

	Leveled-LSM	**Tiered-LSM**	**LDI**
#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

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

layout of Tiered-LSM requires more I/O than both those of Leveled-LSM and LDI in all three queries. Leveled-LSM exhibits the best performance, but LDI query performance is equivalent for $Q1$ and $Q3$ and is only one I/O higher than Leveled-LSM for $Q2$.

TABLE II: Query cost with respect to the data layouts of different methods

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

LDI data buckets can overlap because data is not strictly sorted and does not use merges. Therefore, some queries will read extraneous data at a higher cost. Ultimately, the goal of LDI design is to minimize these extra penalties by achieving good bucket compactness.

In the Appendix VIII-B, we experimentally detail the merge costs of LSM and LDI on columnar databases.

III. Learned Distribution Index

The basic idea of LDI is to use a fixed amount of training data to learn a given distribution, and then continuously update the learned distribution as new data comes in.

LDI is conservative in updating its distribution, since a change in the distribution will affect query costs. In order to learn, it maintains two distributions: a *global* distribution based on which buckets are created, and a *local* distribution, which reflects the state of incoming data. Local distribution may change more rapidly but bucket creation is determined by the global distribution. Only when sufficient evidence about the local distribution is collected, and a *drift* is detected, LDI updates the global distribution. In LDI, a normalized frequency is used to create a distribution and is thus similar to V-optimal histograms but unlike dynamic V-optimal histograms [17] which keep tuning configurable, LDI decides when to tune.

We now describe in detail how the distribution is initialized and updated based on drifts. We then present the data maintenance in LDI that keeps data compacted in order to improve the query performance.

A. Initializing a distribution

LDI distribution is represented by an array of n contiguous intervals $\{[b_i, b_{i+1})\}$, where b_i and b_{i+1} are interval boundaries. For each interval, we maintain two values: a normalized frequency, denoted dis and a count of values denoted $load$.

Interval boundaries and frequency counts are initialized by inserting a fixed amount of data into a k-ary B$^+$-tree, and using the min/max key values in the leaf nodes as interval boundaries to determine how many values fall into these leaf nodes. To reduce the cost of tree traversal, we prefer to create a wide B$^+$-tree by using a high value of k (e.g., $k = 128$).

B. Updating a distribution

Table III summarizes the variables maintained by LDI. Interval boundaries and frequency counts are updated as new data arrives. Local (denoted by L) and Current (denoted by C) have no value initially (distribution or load); LDI determines the frequency counts using a time window t with $size$ data entries. For each window t, we record the normalized frequency and the count of keys falling into each interval in C. Once time window t is ended, the recorded normalized frequency $C.dis_i$, and the current total count $C.load_i$ are updated. These current quantities ($C.dis_i$ and $C.load_i$) are then accumulated into local normalized frequency (denoted by $L.dis_i$) and local count (denoted by $L.load_i$).

We define the distribution drift$(L.dis_i, G.dis_i)$ as the ratio between the weighted estimate of the global normalized frequency and the local normalized frequency for the ith interval. Intuitively, a ratio of 1 represents a data distribution that has not changed and deviation from 1 represents an increase or decrease in the data seen for this interval. If the drift is positive, more data is arriving, and we might be creating buckets that are too narrow; if the drift is negative, less data is arriving, and we might be creating buckets that are too wide.

$$\text{drift}(i) = \frac{\text{w}_{L_i} * L.dis_i + \text{w}_{G_i} * G.dis_i}{G.dis_i} \tag{1}$$

TABLE III: Notation used in this paper

Parameter	Description
$G.b_i$	Global boundary at interval i^{th}
$G.dis_i$	Global normalized frequency at interval i^{th}
$G.load_i$	Global count at interval i^{th}
$L.b_i$	Local boundary at interval i^{th}
$L.dis_i$	Local normalized frequency at interval i^{th}
$L.load_i$	Local count at interval i^{th}
$C.b_i$	Current boundary at interval i^{th}
$C.dis_i$	Current normalized frequency at interval i^{th}
$C.load_i$	Current count at interval i^{th}
drift$(L.dis_i, G.dis_i)$	The change in the distribution (local vs global) at interval i^{th}
Φ^{max}	The upper boundary (split) threshold
Φ^{min}	The lower boundary (merge) threshold
$size$	The total number of data tuples (data entries) in a time window
$count_i$	The number of data tuples (entries) falling to the interval i^{th} in a time window

The weights for local and global normalized frequency are based on the ratio between the amount of data observed so far. More data represents stronger evidence for L or G:

$$\text{w}_{L_i} = \frac{L.load_i}{L.load_i + G.load_i} \tag{2}$$

$$\text{w}_{G_i} = \frac{G.load_i}{L.load_i + G.load_i} \tag{3}$$

where $L.load_i$, $L.dis_i$, $G.load_d$ and $L.dis_i$ are described in Table III. The relative distribution drift(i) in Equation 1 is used to determine whether the interval ranges should be modified. While drift remains near 1, no change is needed; but as the

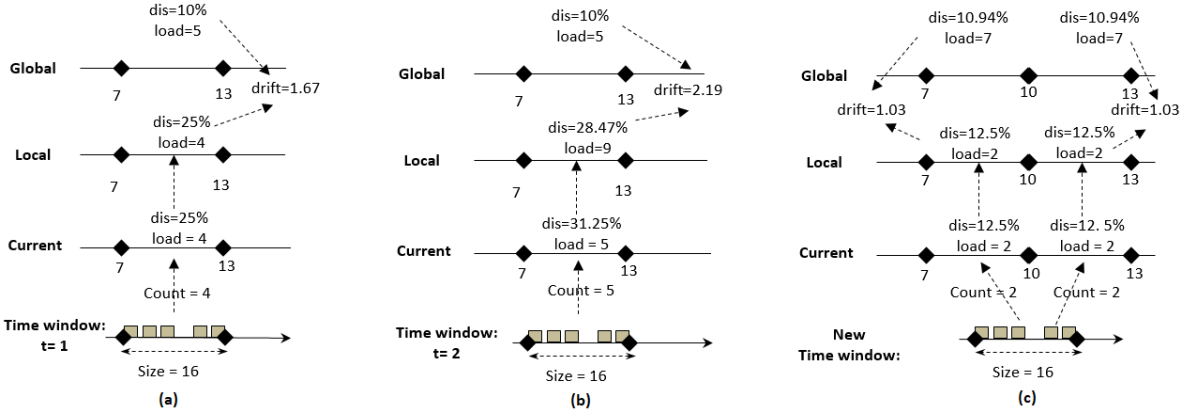

Fig. 3: Online interval tuning Algorithm applied on the interval $[7 \div 12]$ of LDI shown in Figure 2b.

value goes above Φ^{max} or below Φ^{min}, the intervals are split or merged accordingly:

$$Action(i) = \begin{cases} \text{drift}(i) \geq \Phi^{max}, & \text{Split.} \\ \Phi^{max} > \text{drift}(i) > \Phi^{min}, & \text{Skip.} \\ \text{drift}(i) \leq \Phi^{min}, & \text{Merge.} \end{cases} \quad (4)$$

In practice, our current thresholds were set as $\Phi^{max} = 2$ and $\Phi^{min} = 0.3$. In Algorithm 1, after each time window t, once the $count_i$ data tuples are accumulated to local L, the drift for interval i will be calculated to determine whether a split or a merge is needed ($Phase2$ – lines [11–21]). If a split is required at an interval i (lines [13–16]), the current data distribution in local L at this position will be added to the global distribution G before a split is performed (the split happens in the global level G, i.e. the list of boundaries $G.b_i$ are adjusted). Similarly, if a merge is required at an interval i (lines [18–21]), the current data distribution in local L at this position will be incorporated into the global distribution G (lines [18–20]), merging the current interval with its left or its right neighbor. Finally, if there is any change in the global G, then we propagate this change to the current distribution C (lines [22–23]), to make sure the intervals $C.b_i, L.b_i, G.b_i$ are same.

Figure 3 shows an example of how distributions are updated and applied to the interval boundary $[7 \div 12]$ (the third interval $i = 3$) of LDI shown in Figure 2b. In this example, we consider three time windows $t = 1$, $t = 2$ and $t = 3$, where each window has 16 data entries. The sample data is shown in Figure 1. Figure 3(a) shows the behavior of the algorithm when receiving the first 16 data entries. Since in the first 16 data entries (see Figure 1), there are 4 keys falling into $[7, 13)$ range, the current normalized frequency is $C.dis_3 = 4/16 = 25\%$ and the current count $C.load_3 = 4$. These current distribution and load will be accumulated to the local L. Next, the distribution $drift(L.dis_3, G.dis_3) = 1.67$. Since this $drift$ value does not go above Φ^{max} or below Φ^{min}, there is no split or merge after this time window.

In the next time window $t = 2$ (see Figure 3), there are 5 keys falling into $[7, 13)$. Thus, the current normalized frequency is $C.dis_3 = 5/16 = 31.25\%$ and workload $C.load_3 = 5$ will be accumulated with the local (new values: $L.dis_3 = 28.47\%$ and $L.load_3 = 9$). Since $drift(L.dis_3, G.dis_3) = 2.19$ is higher than $\Phi^{max} = 2$, there will be a split in this interval 3 after this time window. Figure 3(c) shows the new intervals after the split and the new values in those intervals when new data arrives.

C. LDI Maintenance

LDI writes buckets based on the global distribution, but sometimes may need to perform some in-memory merging to write out buckets. Going back to our example in Figure 2b, step 7, data buffer $(1, 29, 8, 4)$ is full but none of the leaf ranges are full. As a result, LDI has to combine data from two sibling leaves with intervals $[1 \div 2]$ and $[3 \div 6]$ to write a bucket $[1, 4]$. This bucket is called non-compacted because it contains a range of data that is larger than a its leaf range. Similarly, in step 12, a non-compacted bucket $[12, 16]$ is created that will have to be rebuilt. In both these situations the data that arrived fell into each respective interval, but there was not enough data to create a compacted bucket. Based on pigeon-hole principle such a situation is unavoidable even if the global distribution is perfectly learned. In these cases, non-compacted bucket will be created and will need to be compacted by a background data maintenance process, which reads and re-writes buckets.

Merging: As defined in [18], the compactness of an approximately sorted dataset is defined through the average relative bucket range factor (ARB):

$$ARB = \frac{\sum_{i=1}^{K} \left| Range\left(Bucket_i^{sorted}\right) \right|}{\sum_{i=1}^{K} \left| Range\left(Bucket_i^{LDI}\right) \right|} \quad (5)$$

ARB is the ratio of the sum of the range between minimum and maximum values in a bucket for perfectly sorted data versus the range between minimum and maximum values in the LDI bucket. Range is defined as the difference between largest and smallest value in a bucket. Data is considered in bucket units; if approximately sorted data does not have any values in a wrong bucket, ARB will have a perfect score of 1.

LDI performs a sort-merge operation by combining all non-compacted buckets, in order to approximate an improved ARB

Algorithm 1: Dynamic-interval tuning process

***Interval-Tuning ()*:**
  input : Global partitioning G, Local distribution O, Current distribution C
  output: New G, O
  $G = \{G.b_i, G.dis_i, G.load_i\}$; //Global intervals
  $L = \{L.b_i, L.dis_i, L.load_i\}$; //Local intervals
  $C = \{C.b_i, C.dis_i, C.load_i\}$; //Current intervals
  Phase 1: Merge current distribution to local intervals
  foreach $(\{L.dis_i, L.load_i\}, \{C.dis_i, C.load_i\})$ ***in*** $\{L, C\}$ **do**
    $L.dis_i$ = ($L.dis_i$ * $L.load_i$ + $C.dis_i$ * $C.load_i$) / ($L.load_i$ + $C.load_i$);
    $L.load_i$ += $C.load_i$;
  Phase 2: Tune the global, intervals if necessary
  foreach $(\{L.dis_i, L.load_i\}, \{G.dis_i, G.load_i\})$ ***in*** $\{L, G\}$ **do**
    if *(drift*$(i) \geq \Phi^{max}$*)* **then**
      $G.dis_i$ = ($G.dis_i$ * $G.load_i$ + $L.dis_i$ * $L.load_i$) / ($G.load_i + L.load_i$);
      $G.load_i$ += $L.load_i$; $L.load_i$ = 0; $L.dis_i$ = 0;
      Split current interval i;
    else if *(drift*$(i) \leq \Phi^{min}$*)* **then**
      $G.dis_i$ = ($G.dis_i$ * $G.load_i$ + $L.dis_i$ * $L.load_i$) / ($G.load_i + L.load_i$);
      $G.load_i$ += $L.load_i$; $L.load_i$ = 0; $L.dis_i$ = 0;
      Merge the current interval i to the left or to the right;
  if *there is a split or a merge in* G **then**
    $C = G$;

measure. In LDI, a bucket is considered to be non-compacted if it has at least one data key out of the key range where this bucket is indexed (i.e., the bucket had to be merged with another leaf interval). For instance, as shown in the Figure 2b, buckets [1-4] and [12-16] are non-compacted bucket as the data key 4 and 16 are out off the key ranges (i.e., [1-2] and [7-12]) where those buckets are indexed.

IV. LDI Architecture

Figure 4 gives an overview architecture of LDI: (i) Data-clustering component generates buckets with data distribution as discussed in Section III; (ii) Bucket indexing indexes the buckets using interval B-Tree (Section IV-A; and (iii) I/O handler (Section IV-B). LDI receives data tuples and generates buckets. The created buckets are then written to disk by I/O handler, while buckets metadata are indexed in Bucket indexing.

A. Bucket indexing for Query Performance

Buckets generated by LDI are stored on disk and indexed by interval B-Trees [19] (i.e., IB-Trees), in order to further improve query performance. In LSM, Bloom filters are typically

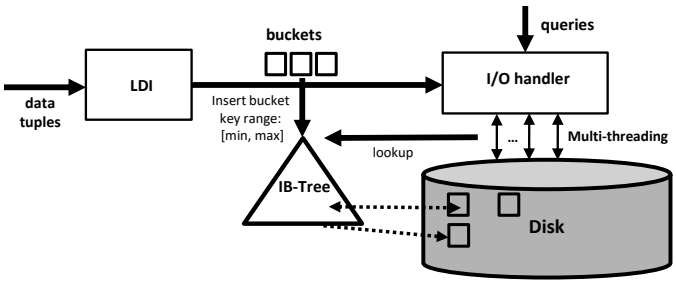

Fig. 4: The overview architecture of LDI.

used to improve query performance. Since analytic workloads are predominantly range queries, an IB-Tree is more suitable. Incoming buckets are indexed using their key ranges. The range of values $[L, H]$ of a node 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.

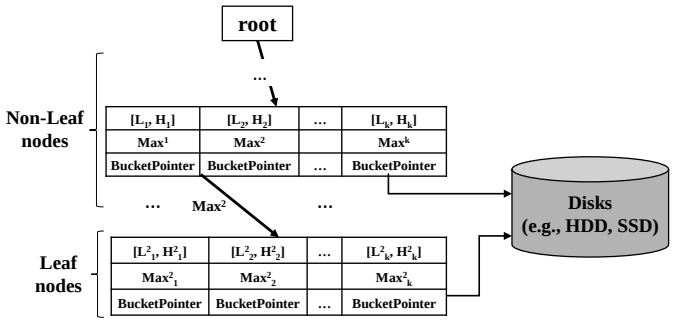

Fig. 5: The data structure of IB-Tree.

Figure 5 illustrates the adaptation of IB-Tree applied in our context. The size of node is a multiple of page size (i.e., $k * 4KB$) so that all nodes in IB-Tree can be efficiently stored on disk. IB-Tree keeps its data (i.e., $BucketPointer$)[2] in all types of nodes, including both leaf or non-leaf nodes.

There are a few of advantages of using IB-Tree. First, IB-Tree indexes at the granularity of a bucket (i.e., a group of tuples), so the index size is reduced based on the average number of tuples in a bucket. Second, it reduces the time to search for a bucket from $O(N)$ to $O(logN)$ [19], [20] (similar to the search cost in B-Tree). Third, IB-Tree is designed for either full or partial loading in memory.

B. I/O handler

In most computer system, the storage layer is the major performance bottleneck; more so with large-scale database systems. While reducing amount of I/O should improve query performance, the strategy for performing I/O operations also plays crucial role in determining query execution time. Current solutions optimize I/O operations by favoring large granularity I/O and reducing random I/O [21]–[23]. The I/O handler

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

is designed to further improve I/O through: (i) avoiding interference between concurrent read and write operations; (ii) supporting concurrent transactions; and (iii) exploiting internal parallelism of storage devices. We detail those techniques in Appendix VIII-C.

V. Evaluation

Purpose: Our experiments 1) demonstrate the significant improvement in data loading performance for *LDI* index, 2) compare read query performance for *LDI* index to current state-of-the-art approaches, and 3) present the benefits of parallelism achieved by our I/O handler in *LDI*.

Setup: In our evaluation, we implemented two versions of basic column-oriented DBMS with *LDI* in C++: one without disk-level parallelism (*LDI*) and other with our I/O handler parallelism optimization (*LDI_Par*). We also implemented a basic column-oriented DBMS (*Sorted_Col*), in which all projections are sorted on the indexed key. Queries executed in *Sorted_Col* are expected to have the best possible performance as columns are fully sorted. We also deployed a column-store with LSM-Tree-like indexing (*Col_LSM-like*) that used the same datasets as well as *Min-Max* approach [15]. The comparison with other variations of LSM-Tree such as [12], [13] is a part of our future work.

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.

We reported our experiment on SSD in this section, the same experiments on HDD were also conducted and reported in Appendix Sections VIII-D, VIII-E and VIII-F.

TABLE IV: NYC dataset sizes.

Table	T_A	T_B	T_C	T_D	T_E
Records (M)	1.5	15	30	59	148
Size (GB)	0.3	3	6	11.8	29.6

Dataset & Queries: Experiments used real-world data from the New York City Taxi (NYC) dataset [24]. Table IV summarizes the five table sizes used. 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 V 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.

TABLE V: Query Range and Selectivity.

Range Query	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**
Key Range	0.003	0.005	0.020	0.030	0.055
Selectivity	0.003	0.006	0.015	0.026	0.050
Range Query	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**
Key Range	0.065	0.075	0.085	0.100	0.110
Selectivity	0.067	0.083	0.100	0.130	0.150

A. Loading Costs

Figure 6 summarizes the loading costs for the three approaches on SSD (*Min-Max*'s loading cost is similar to *LDI*). To provide a baseline reference, we also include Write$_{MAX}$, the maximum sequential write speed for each storage device. The *Sorted* runtimes include the load time plus a one-time clustering operation performed at the end of a bulk-load. Similarly, the *Col_LSM-like* runtimes include the costs for reorganizing data in all projections. It's worth emphasizing that the results are for one-time bulk-load ingestion, whereas continuous data ingestion or incremental data loading will likely lead to additional clustering/optimization overheads in both *Sorted* and *Col_LSM-like*. Meanwhile, *LDI* is an online process that does not incur any additional overheads for incremental data loading.

We observed that the load times for *LDI* were within a factor of three compared to the disk capacity Write$_{MAX}$. *LDI* significantly improved data load time for the column-oriented DBMS on an SSD; it was, on average, 21X times faster than *Sorted* across all tables (note that the load times are shown on a logarithmic scale). The advantage of *LDI* comes from avoiding data maintenance and clustering overhead during ingestion. *Sorted* includes the loading time and a cost of clustering at the end of bulk-loading. Similarly, loading runtimes in *Col_LSM-like* include merge costs. Moreover, incremental loading of data (versus a one-time bulk load) can only degrade the performance of both *Sorted* and *Col_LSM-like*. In contrast, *LDI* load cost is always based on the amount of data it ingests.

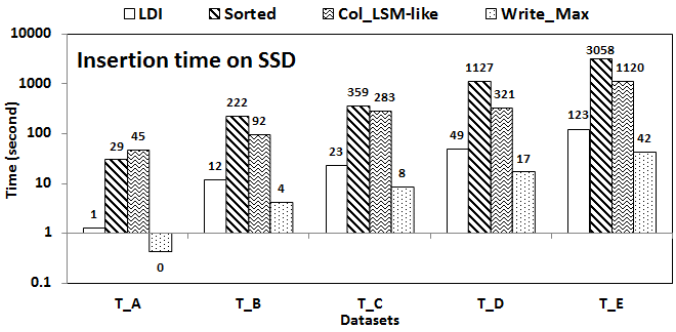

Fig. 6: Loading runtimes on SSD with different tables.

B. Read Query Performance

LDI eliminates clustering reorganization (sort and merge) costs associated with data loading by creating approximately clustered buckets. Since *LDI* reads queried data in approximately sorted buckets, it may read extraneous tuples stored in the buckets that contain some of the data in queried range. As a result, *LDI* queries may incur a small penalty in comparison to approaches that strictly cluster data. In this evaluation, we compare *LDI* to the best possible query performance of *Sorted*. We ran queries against two different projections: $Projection1$ (29 columns) and $Projection2$ (4 columns) with all competitors: *LDI*, *LDI_par* (with parallelism optimization), *Sorted*,

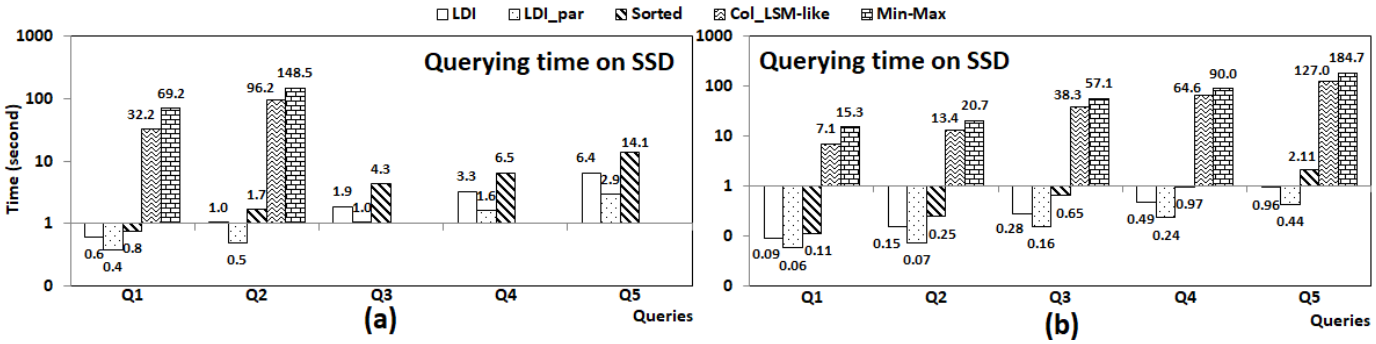

Fig. 7: Runtimes of different queries on SSD at different projections: (a) Runtimes on $Projection1$ with SSD; and (b) Runtimes on $Projection2$ with SSD.

Col_LSM-like and Min-Max. Query runtimes for SSD are shown in Figure 7.

The first observation is that *Sorted* typically has the fastest query runtimes in queries on SSD (i.e., Figure 7(a) and 7(b)) among other competitors except *LDI*. This is because data is perfectly sorted in *Sorted*. *LDI_par* has better query performance than *Sorted* on all evaluated queries and on SSD (see Figure 7 (a) and (b)). *LDI_par* can outperform Sorted by leveraging our parallelism optimization, which is most effective on larger queries (that access the most data) and on SSD.

As expected, the query runtimes of *LDI* (without parallelism) are slightly slower than those of *Sorted* because: 1) *LDI* data is approximately sorted in buckets, causing it to read some extraneous data that is not required by the query, and 2) read operations are done in bucket granularity instead of sequential reads in *Sorted*. However, on SSD, when the performance of random I/O and sequential I/O are almost similar, *LDI* performance is better than *Sorted* with large queries (see Figure 7 (b)). This comes from our I/O handler in *LDI* that optimizes the query performance by grouping buckets which are physically stored in close proximity (see Appendix VIII-C). In other words, our I/O handler combines many 'neighbor' I/Os into a single big I/O, leading to accessing to more data, but reducing the data accessing time.

Col_LSM-like and *Min-Max* shows the slowest query performance among other evaluated approaches, for all tested queries. It takes a long time to query $Projection1$ (see Figure 7(a)). For better readability we only show the runtimes of queries $Q1$ and $Q2$, as other queries were much slower. *Col_LSM-like* and *Min-Max* runtimes on $Projection2$ are faster (see Figure 7(b)), but still slower than other methods.

C. Effectiveness of concurrence and parallel processing

The experiments in this section aim to evaluate the effectiveness of concurrence and parallel processing in read queries. In order to show how *LDI* behaves in presence of concurrence requests, by de-duplicating overlapping data and by leveraging concurrency, we created a set of 40 queries with selectivity ranging from 0.2% to 5%. We then run these queries, varying the number of threads from 1 to 32. Figure 8 presents the

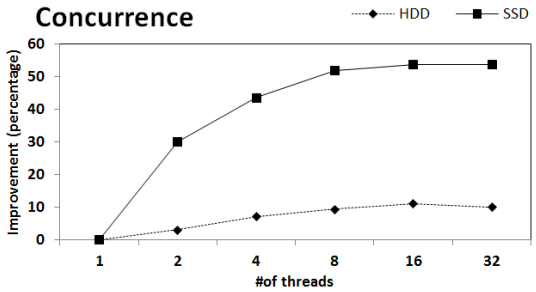

Fig. 8: Runtime improvements on HDD and SSD.

runtime improvements (percentage improvement on y-axis) for both HDD and SSD as the number of threads increases (x-axis).

As Figure 8 shows, while improvement increases (the runtime decreases) as the number of threads increases, the best number of threads in HDD is 8 or 16 threads (improvement of 9% or 11%), while on SSD the best number of threads is 16 or 32 (improvement of 53%). The results confirm that 1) concurrent processing improves query performance both in HDD and in SSD, and 2) SSD has a greater capability for exploiting concurrency compared to HDD.

We also observe that increasing the number of threads eventually becomes counterproductive. For example, in our experiments 32 threads on HDD and 64 threads on SSD lead to query runtime deterioration. In fact, the effectiveness of concurrence comes from parallelism and de-duplication. However when as the number of threads increases, the overheads of multi-threading will negate the benefits of concurrence. Furthermore, a large number of threads may require a significant amount of memory to store temporary query results.

In order to examine the impact of the number of threads in reading/writing from/to storage devices, we vary the number of threads that are allowed to read/write at the same time using queries from Table V. Figure 9 shows the runtimes of different queries as we vary the number of parallel threads (x-axis shows the number of threads, y-axis shows the execution time in seconds).

Figure 9 shows the results for Q6 through Q10. As shown, the best number of threads on SSD is 16 threads. Continuing

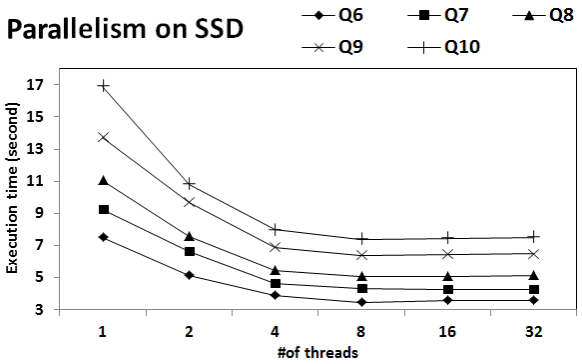

Fig. 9: Parallel on SSD.

to increase the number of threads offers very little benefit or potentially begins to slow queries down due to the overhead costs of multi-threading.

VI. Related Work

Columnar Stores: The ideas of column-oriented hierarchical have been widely applied in both academic [9], [25] and commercial [10], [26]–[28]. MonetDB [25] is an in-memory columnar relational DBMS that leverages the large main memories of modern computer systems in query processing, while the database is persistently stored on disk, showing outstanding performance. However, similar to other in-memory column-oriented DBMSes such as SAP HANA [29], Peloton [30], DB2 BLU [27] or Microsoft SQL Server Column stores [28]. Recently, in the same context as MonetDB, Vectorwise [26] was proposed, aiming at vectorizing execution to operate on vector of data instead of separate tuples and further improving in storage model. However, the applications of those methods are limited due to many reasons: (i) require huge amount of main memory; and (ii) the volatility of main memory.

C-Store [9] and its recent commercial extension (i.e., Vertica [10]) are on-disk columnar databases that apply different sort orders on each projections (i.e., groups of columns), different compression method for each column in order to improve the compression ratio, fully support aggregation operations on compressed data. However, similar to other column stores, incremental data loading may require heavy data clustering/reorganization processes. Even though, these processes can be run as background processes in Vertica, it negatively impacts DBMS performance.

Log Structured Merge Trees While many DBMSes including columnar stores suffer from poor write performance, the log-structure merge tree (LSM-Tree) [3], [4] is a common solution for this problem. Other enhanced variations of LSM-Tree used in Monkey [12] and Dostoevsky [13] 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 still remains.

Similarly in TRIAD [31], couple of improvement techniques have been applied to reduce the write amplification such as (i) keeping the hot-entries longer in the main memory; (ii) changing the tired merging policy by considering the overlapping between runs in a level; and (iii) optimize the write in commit-log. Unfortunately, the gain comes with a price. Lower write amplification is archived by using a variation of tiered policy (i.e., favor the write performance) and by scarifying the look up performance (the higher the merge overlapping threshold, the more degradation in look up). Furthermore, similar to Dostoevsky [13], this method only reduces the high write amplification of LSM-Tree, but cannot completely avoid this issue.

Data distribution Knowing data distributions is crucial important in database systems and data streaming. However, accurately record data distributions is expensive, leading to many approaches for approximately capture data distribution (called incremental histograms) [32]–[34]. For example, Gibbons et al. [33] proposed an approximate histograms maintained in the present of data insertion and a merge and split technique for adjusting histogram buckets according to the data insertion. Meanwhile, Mousavi et al. [32] introduced an approximate approach for incrementally approximate compute equi-depth histograms over sliding windows.

Those histogram methods have been applied in many aspects of database systems such as selectivity estimation (query optimization), approximate query answering, join query execution. However, as mentioned in [17], [35], traditional indexing using B^+-Tree is not suitable to serve as non-equi-depth histograms. Particularly, straightforwardly applying basic index trees to serve non-equi-depth histograms should significantly degrade indexing performance due to its unbalanced structure or low node occupancy [17]. To the best of our knowledge, there is no work applying histogram information in database indexing.

VII. Conclusion

In this paper, we presented a learned distribution index (LDI) structure as an alternative for LSM tree structure, which is advocated for ingesting data rapidly into projections of a columnar database to improve insertion performance. Learning distributions, as we show, avoids the expensive sort-merge phase of LSM. As a result, data ingestion performance increases by more than an order of magnitude in comparison with other methods, while query performance remains comparable to LSM-like indexing approaches. The cost of learning distribution may lead to some non-compacted buckets, causing additional maintenance in LDI. We believe this cost of maintenance is reasonable as only the non-compacted buckets are need to be inverted. Further study on LDI's maintenance remains as part of our future work.

Acknowledgment

This work is supported by National Science Foundation under grants CNS-1846418, NSF ICER-1639759, ICER-1661918 and a Department of Energy Fellowship.

REFERENCES

[1] Wikipedia, "Business intelligence," https://en.wikipedia.org/wiki/Business_intelligence, 2019, [Online; accessed 25-June-2019].

[2] Columnar Database, "Columnar database: A smart choice for data warehouses," https://www.columnardatabase.com/, 2019, [Online; accessed 25-Mar-2019].

[3] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Inf.*, vol. 33, no. 4, pp. 351–385, Jun. 1996. [Online]. Available: http://dx.doi.org/10.1007/s002360050048

[4] R. Sears and R. Ramakrishnan, "bLSM: A general purpose log structured merge tree," in *SIGMOD '12*. New York, NY, USA: ACM, 2012, pp. 217–228. [Online]. Available: http://doi.acm.org/10.1145/2213836.2213862

[5] Google, "Leveldb," https://github.com/google/leveldb, [Online; accessed 25-Mar-2019].

[6] 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 *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 15–15.

[7] Apache, "Hbase," https://github.com/google/leveldb, [Online; accessed 25-Mar-2019].

[8] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773922

[9] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik, "C-store: A column-oriented dbms," in *VLDB'05*, 2005.

[10] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear, "The vertica analytic database: C-store 7 years later," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1790–1801, Aug. 2012. [Online]. Available: http://dx.doi.org/10.14778/2367502.2367518

[11] E. Bortnikov, A. Braginsky, E. Hillel, I. Keidar, and G. Sheffi, "Accordion: Better memory organization for lsm key-value stores," *Proc. VLDB Endow.*, vol. 11, no. 12, pp. 1863–1875, Aug. 2018. [Online]. Available: https://doi.org/10.14778/3229863.3229873

[12] N. Dayan, M. Athanassoulis, and S. Idreos, "Monkey: Optimal navigable key-value store," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. New York, NY, USA: ACM, 2017, pp. 79–94. [Online]. Available: http://doi.acm.org/10.1145/3035918.3064054

[13] N. Dayan and S. Idreos, "Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: ACM, 2018, pp. 505–520. [Online]. Available: http://doi.acm.org/10.1145/3183713.3196927

[14] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka, "TRIAD: Creating synergies between memory, disk and log in log structured key-value stores," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, Jul. 2017, pp. 363–375. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/balmau

[15] D. H. Ton That, M. Gharehdaghi, A. Rasin, and T. Malik, "On lowering merge costs of an lsm tree," in *33rd International Conference on Scientific and Statistical Database Management*, ser. SSDBM 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 253–258. [Online]. Available: https://doi-org.ezproxy.depaul.edu/10.1145/3468791.3468820

[16] B. C. Kuszmaul, "A comparison of fractal trees to log-structured merge (lsm) trees," *White Paper*, 014.

[17] D. Barbara, W. duMouchel, C. Faloutsos, P. J. Haas, J. M. Hellerstein, Y. Ioannidis, H. V. Jagadish, T. Johnson, R. Ng, V. Poosala, K. A. Ross, and K. C. Sevcik, "The new jersey data reduction report," *IEEE DATA ENG. BULL*, pp. 3–45, 1997.

[18] D. H. Ton That, J. Wagner, A. Rasin, and T. Malik, "PLI$^+$: Efficient clustering of cloud databases," *Distributed and Parallel Databases*, 2018, in the third round of DAPD.

[19] C.-H. Ang and K.-P. Tan, "The interval b-tree," *Inf. Process. Lett.*, vol. 53, no. 2, pp. 85–89, Jan. 1995. [Online]. Available: http://dx.doi.org/10.1016/0020-0190(94)00176-Y

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[21] D. H. Ton-That, I. Sandu-Popa, and K. Zeitouni, "TRIFL: A generic trajectory index for flash storage," *ACM Trans. Spatial Algorithms Syst.*, vol. 1, no. 2, pp. 6:1–6:44, Jul. 2015. [Online]. Available: http://doi.acm.org/10.1145/2786758

[22] I. Sandu Popa, K. Zeitouni, V. Oria, D. Barth, and S. Vial, "Indexing in-network trajectory flows," *The VLDB Journal*, vol. 20, no. 5, pp. 643–669, Oct. 2011. [Online]. Available: http://dx.doi.org/10.1007/s00778-011-0236-8

[23] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi, "Tree indexing on solid state drives," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 1195–1206, Sep. 2010. [Online]. Available: http://dx.doi.org/10.14778/1920841.1920990

[24] T. W. Schneider, "Unified new york city taxi and uber data," https://github.com/toddwschneider/nyc-taxi-data, 2016, [Online; accessed 18-Aug-2017].

[25] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten, "Monetdb: Two decades of research in column-oriented database architectures," *IEEE Data Engineering Bulletin*, vol. 35, no. 1, pp. 40–45, 2012.

[26] M. Zukowski and P. A. Boncz, "Vectorwise: Beyond column stores," *IEEE Data Eng. Bull.*, vol. 35, pp. 21–27, 2012.

[27] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm, and L. Zhang, "Db2 with blu acceleration: So much more than just a column store," *Proc. VLDB Endow.*, vol. 6, no. 11, pp. 1080–1091, Aug. 2013. [Online]. Available: http://dx.doi.org/10.14778/2536222.2536233

[28] P. Larson, C. Clinciu, C. Fraser, E. N. Hanson, M. Mokhtar, M. Nowakiewicz, V. Papadimos, S. L. Price, S. Rangarajan, R. Rusanu, and M. Saubhasik, "Enhancements to sql server column stores," in *SIGMOD '13*, June 2013.

[29] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "Sap hana database: Data management for modern business applications," *SIGMOD Rec.*, vol. 40, no. 4, pp. 45–51, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2094114.2094126

[30] J. Arulraj, A. Pavlo, and P. Menon, "Bridging the archipelago between row-stores and column-stores for hybrid workloads," in *SIGMOD '16*. New York, NY, USA: ACM, 2016, pp. 583–598. [Online]. Available: http://doi.acm.org/10.1145/2882903.2915231

[31] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka, "TRIAD: Creating synergies between memory, disk and log in log structured key-value stores," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 363–375. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/balmau

[32] H. Mousavi and C. Zaniolo, "Fast and accurate computation of equi-depth histograms over data streams," in *Proceedings of the 14th International Conference on Extending Database Technology*, ser. EDBT/ICDT '11. New York, NY, USA: ACM, 2011, pp. 69–80. [Online]. Available: http://doi.acm.org/10.1145/1951365.1951376

[33] P. B. Gibbons, Y. Matias, and V. Poosala, "Fast incremental maintenance of approximate histograms," *ACM Trans. Database Syst.*, vol. 27, no. 3, pp. 261–298, Sep. 2002. [Online]. Available: http://doi.acm.org/10.1145/581751.581753

[34] D. Donjerkovic, Y. Ioannidis, and R. Ramakrishnan, "Dynamic histograms: Capturing evolving data sets," in *Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073)*, 2000, pp. 86–.

[35] Y. Ioannidis, "The history of histograms (abridged)," in *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, ser. VLDB '03. VLDB Endowment, 2003, pp. 19–30. [Online]. Available: http://dl.acm.org/citation.cfm?id=1315451.1315455

[36] F. Chen, R. Lee, and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, Feb 2011, pp. 266–277.

[37] S. ATA, "Native command queuing," https://sata-io.org/developers/sata-ecosystem/native-command-queuing, 2018, [Online; accessed 7-Nov-2018].

VIII. Appendix

A. Example: Tiered LSM-Tree

The Figure 10 shows the Tiered LSM-Tree behavior during the data ingestion (Sample data D - Figure 1). As shown, there are only two merges at the steps 2 and 4.

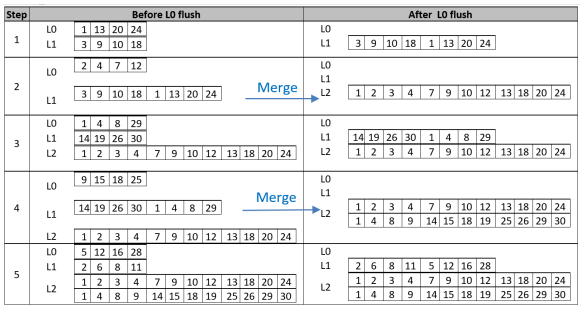

Fig. 10: Data loading with tired and tiered LSM-Trees.

B. An Experiment: LSM vs LDI in Column Databases

In the previous example we compared the performance of LSM trees with LDI on a small example data. In this section, we measure it experimentally on columnar databases based on analytic results previously formulated [15].

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 [9] or Vertica [10]) 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 [9] or Vertica [10]. 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.

LSM tree merge cost. We refer the reader to [15] for the analytical cost of levelling and tiering merge policies, but determine these costs experimentally over here.

We measure the total number of merges with different size of data in a columnar store. In general, tiered LSMs cause fewer merges than levelled LSMs, 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 11. In Figure, the column store has two configurations: 5 projections with each projection has 3 columns, and 1 projection with all columns. As shown in 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. In contrast, a LDI has close to zero merges.

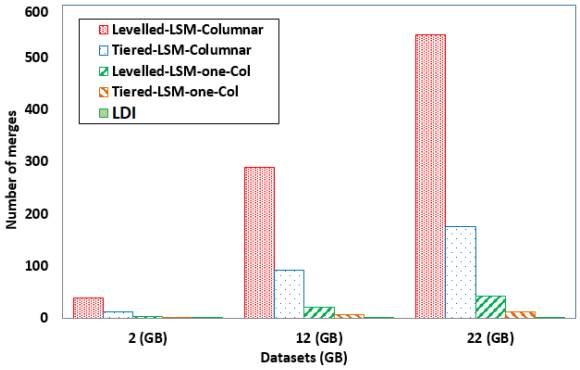

Fig. 11: The total number of merges during data-loading with tiered and levelled LSM trees. Multiple LSM trees on columnar stores, single LSM tree on one column and LDI. The columnar has 5 projections each of projection has 3 columns; whereas on-column keep the data in only one projection (all columns)

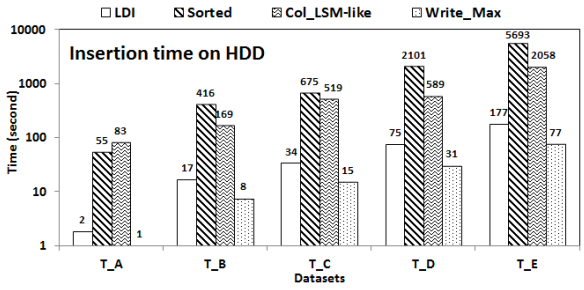

Fig. 12: Loading costs on HDD for different tables.

C. LDI: I/O Handler design principles

Avoid interference between concurrent read and write operations: Many DBMSes support concurrent execution of insert and read query requests. However, recent experiments show that mixing of parallel reads and writes has a strong negative impact on throughput of storage devices [36]. In our approach, concurrent insertion and read queries are allowed, but there is no mixing of insertion and read queries.

Concurrent transactions: Concurrent handling of queries can be exploited to eliminate transfer of data which is redundant among requests. Although multiple identical queries sent to a DBMS at the same time are unlikely, it is common to have overlap in data access by different concurrent requests. To reduce query read cost multiple requests for overlapping data is eliminated by combining requests for the same buckets.

Moreover, requests for different data that are physically stored in close proximity should be grouped, since modern storage devices can efficiently combine such access. To this end, we collect the list of desired buckets and group them into co-located bucket groups (note that bucket is the unit of access, interchangeable with block). Since random I/O access incurs additional overhead, we automatically switch to reading an entire group of buckets (including some extraneous data), based on a threshold θ. The θ threshold is determined for each

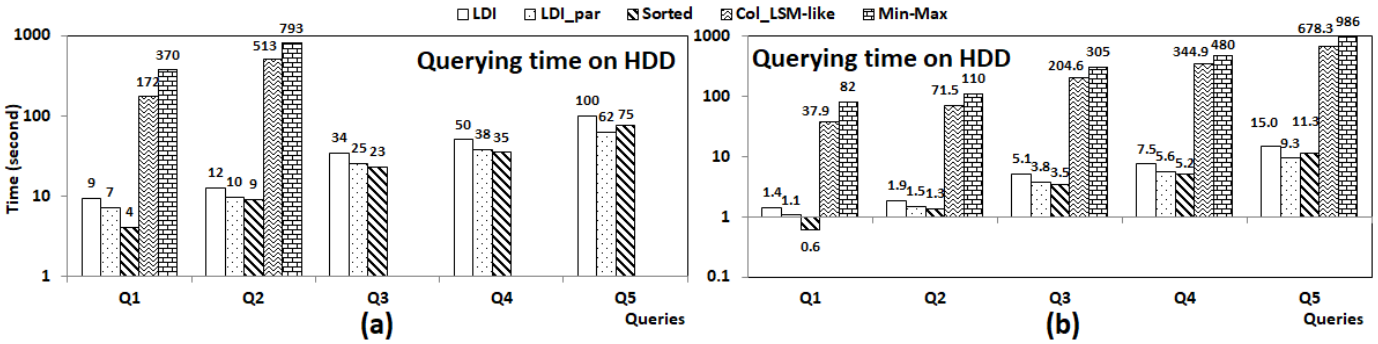

Fig. 13: Runtimes of different queries on HDD at different projections: (a) Runtimes on $Projection1$ with HDD; and (b) Runtimes on $Projection2$ with HDD.

specific storage device, using the following formula:

$$\theta = \frac{T_{Read}(Group)}{T_{Read}(Bucket)} \tag{6}$$

where $T_{Read}(Group)$ is the time to read a bucket group and T_{Read} is the time to read a bucket.

Exploit internal parallelism of storage devices: We explain how we choose the thread levels in SSDs. SSD has many internal levels of parallelism. Different manufactures or even different devices products may have variations in channels, chips, dies, planes, allocation schemes (how data is allocated on SSD) and so on. Furthermore, as a user, we have no control on how data is physically placed on SSD, which depends on firmware implementation and current usage of SSD.

Nevertheless, choosing the right number of threads can improve the number of planes used, taking advantage of high-bandwidth I/O channel bus and parallelism channel. Given SSD specifications, Formula 7 defines the number of threads used in our model. The idea is to increase the probability of using many planes, while keeping the number of threads low due to the overhead of switching threads.

$$N_{thread} = \frac{N_{channel} * N_{chip} * N_{die} * N_{plane}}{\lambda} \tag{7}$$

Where $N_{channel}$, N_{chip}, N_{die} and N_{plane} are the number of channels, chips, dies and planes in an SSD and λ is the overhead costs of planning, switching, and buffering temporary results of multiple thread in an SSD-controller.

In theory, CPU and main memory overheads in an SSD controller are considered negligible. Thus, in our evaluation we set $\lambda = 1$. In practice, these costs may have a noticeable effect on performance ($\lambda > 1$). In order to determine a specific value of λ for a drive, one would rely on an SSD-specific third party benchmark. For HDDs, we rely on Native Command Queuing (NCQ) [37] for sending parallel requests.

D. Experiment: Loading costs on HDD

Figure 12 details the loading costs of all four approach on HDD with different datasets. As with HDD, *LDI* significantly outperforms *Sorted* and *Col_LSM-like* (it is, on average, 27X and 18X times faster than *Sorted* and *Col_LSM-like* across all tables, respectively). The advantage of *LDI* on HDD load times is similar to its advantages in the SSD evaluation, because the overheads of *Sorted* and *Col_LSM-like* approaches are caused by mostly sequential extra I/O operations.

E. Experiment: Read Query Performance on HDD

We ran queries against two different projections: $Projection1$ (29 columns) and $Projection2$ (4 columns) on HDD with all competitors: *LDI*, *LDI_par* (with parallelism optimization), *Sorted*, *Col_LSM-like* and Min-Max. Query runtimes for HDD are shown in Figure 13.

The differences in query runtimes between *LDI_par* and other competitors are even larger on HDD. This is because the throughput of SSD is much higher than throughput of HDD. All performance of all methods were reduced on HDD. *Sorted* typically has the fastest query runtimes in queries on HDD (i.e., Figure 13(a) and 13(b)). This is because data is perfectly sorted in *Sorted*. *LDI_par* is only slightly slower than than *Sorted* on smaller queries and on HDD (see Figure 13 (a) and (b)). However, *LDI_par* actually outperforms *Sorted* with larger queries. *LDI_par* can outperform *Sorted* by leveraging our parallelism optimization, which is most effective on larger queries (in terms of amount of data).

As with SSD, *Col_LSM-like* and *Min-Max*shows the slowest query performance among evaluated approaches, for all tested queries and on HDD. We only show the runtimes of queries $Q1$ and $Q2$, as other queries take even longer. *Col_LSM-like* and *Min-Max* runtimes on $Projection2$ are faster (see Figure 13(a) and (b)), but still slower than other methods.

F. Experiment: Effectiveness of concurrence and parallel processing on HDD

Figure 14 shows the runtimes of different queries as we vary the number of parallel threads (x-axis shows the number of threads, y-axis shows the execution time in seconds).It presents the results for Q1 through Q5 (queries described in Table V). Both HDD works better with multiple threads. The best number of threads on HDD is 8 threads. Similar to SSD, continuing to increase the number of threads offers very little benefit or potentially begins to slow queries down due to the overhead costs of multi-threading.

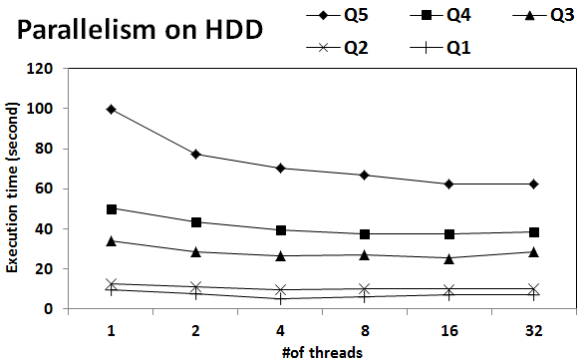

Fig. 14: Parallel on HDD.

Additionally, the results in Figures 14 and 9 further confirms that SSD has better support for parallelism as compared to HDD, since the benefits of parallelism are larger on SSD vs HDD. In particular, the improvement of parallelism optimization on SSD is around 54% (at 16 threads), while on HDD the improvement is around 25% (at 8 threads).