BoDS: A Benchmark on Data Sortedness

Aneesh Raman[1], Konstantinos Karatsenidis[1], Subhadeep Sarkar[1], Matthaios Olma[2], and Manos Athanassoulis[1]

[1] Boston University, MA, USA
{aneeshr,karatse,ssarkar1,mathan}@bu.edu
[2] Microsoft Research, WA, USA
maolma@microsoft.com

Abstract. Indexes in data systems accelerate data access by adding structure to otherwise unstructured data at the cost of index construction and maintenance. Data systems, and particularly, the underlying indexing data structures are designed to offer favorable ingestion (and query) performance for the two extremes of data sortedness, i.e., *unsorted* data (often assumed to follow a uniform random distribution) or *fully-sorted* data. However, in practice, data may arrive with an intermediate degree of pre-sortedness. In such cases, where data arrives nearly (but not necessarily fully) sorted, the intuition is that the indexing cost should be lower than when ingesting unsorted data. Such sortedness-aware index designs lack from the literature. In fact, there is a need for a framework to explore how index designs may be able to exploit pre-existing sortedness during data ingestion to amortize the index construction cost.

In this paper, we present *Benchmark on Data Sortedness*, *BoDS* for short, that highlights the performance of data systems in terms of index construction and navigation costs when operating on data ingested with variable sortedness. To quantify *data sortedness*, we use the state-of-the-art (K, L)-sortedness metric. Specifically, BoDS benchmarks the indexing performance of a data system as we vary the two fundamental components of the metric: (i) K, that measures *how many* elements are out-of-order in a data collection; and (ii) L, that measures by *how much* the out-of-order entries are displaced from their respective in-order positions; as well as (iii) the distribution of L. We present in detail the benchmark, and we run it on PostgreSQL, a popular, production-grade relational database system. Unsurprisingly, we observe that PostgreSQL cannot exploit data sortedness; however, through our experiments we show the headroom for improvement, and we lay the groundwork for experimentation with sortedness-aware index designs. The code for BoDS is available at: https://github.com/BU-DiSC/bods.

Keywords: Sortedness · Indexing · Databases.

1 Introduction

To facilitate efficient query processing, database systems often utilize indexes that are gradually populated as new data is ingested [3,4,6,9,10,18]. Indexes accelerate data access for both analytical and transactional workloads by efficiently supporting selective queries. They improve query performance by adding

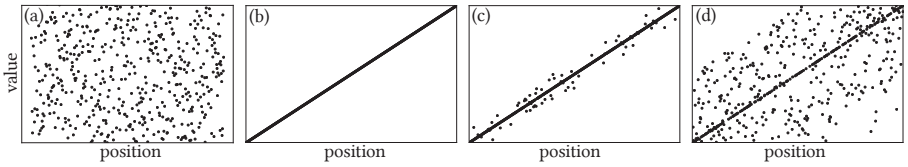

Fig. 1: Classical data organization techniques like indexing in databases focus on the two extremes of data sortedness, i.e., (a) scrambled data, and (b) fully-sorted data. However, there is a lack of a clear framework to evaluate the indexing performance of systems with intermediary degrees of data sortedness as shown in (c) and (d), where data is ordered to some extent.

structure to an otherwise unstructured data collection at the expense of index construction and maintenance.

Data Sortedness. The goal of a range index is to create a fully sorted version of the ingested data (on the indexed attribute). In practice, in several real-life use cases, data arrives with some pre-existing structure, i.e., data may be *near-sorted*, but not necessarily fully sorted on the indexing attribute. For example, in a typical data warehousing benchmark like TPC-H [19], one of the main tables (`lineitem`) has three date columns (`shipdate`, `commitdate`, and `receiptdate`), and when data arrives as ordered on the `shipdate`, the other two date columns on `commitdate` and `receiptdate` are also very close to being sorted (but not fully-sorted) [3]. Near sortedness can also be found in time-series, stock market data, and monitoring measurements that are part of complex hybrid transactional/analytical pipelines. Further, near-sorted data collections often result from a previous query or join operation or sorting based on another naturally correlated attribute [5]. We also have to index near-sorted data when a relation is already sorted based on a collection of attributes, and the index built is a superset of the ranking attributes. In addition to classifying data as *fully sorted* or *not sorted*, there is a wealth of intermediate states of data sortedness. These are captured by *sortedness metrics* [5,8,12,14] which create a continuum between the two extremes. To populate this continuum, we use (K, L)-sortedness [5] that allows us to vary *how many* entries are out-of-order and by *how much*.

Problem: Lack of Sortedness Benchmark for Indexes. When ingesting data in a heap file, we only need to append it at the end of the file. However, when an index is involved, an *additional* index ingestion effort goes to establish a complete order of the ingested data (based on the index attributes) to facilitate future queries. For workloads that arrive with some pre-existing degree of sortedness, one would expect that the intrinsic data sortedness would reduce the extra effort spent to establish total order to the ingested data and speedup index construction. Despite the *natural correlation between data sortedness and index construction cost*, state-of-the-art database indexing techniques are not (*yet*) designed to take advantage of any intermediate degree of sortedness when ingesting near sorted data. Classical index structures (e.g., B^+-trees) focus on the two extremes of data sortedness, i.e., *unsorted* data (regular insertions) and *fully sorted* data [2,7] (visualized in Fig. 1a and 1b), but do not consider the case

of near-sorted data (Fig. 1c and 1d). While it is intuitive that indexes should be able to perform better with increasing sortedness, as *less* effort is required to establish total order, practical performance evaluations for indexes and data systems are unexplored. In order to bridge this gap, we propose a new benchmarking framework that analyzes the performance of indexes and data systems by varying data sortedness in a continuum - from scrambled (unsorted) data to fully sorted data. Specifically, in this work, we focus on quantifying indexing performance when ingesting data that is nearly sorted on the indexed attribute.

Contributions. To this end, we formalize the **Benchmark on Data Sortedness**[3] (BoDS) that varies data sortedness using the (K, L)-sortedness metric [5]. As a first step toward constructing the benchmark, we present a variable-sortedness data generator that builds (K, L)-sorted data collections using a user-specified distribution for L. The benchmark tests five different workload types: (i) pure bulk loading, (ii) one-by-one insertion only, (iii) mixed inserts and queries without pre-loading, (iv) mixed inserts and queries after preloading using bulk loading, and (v) mixed inserts and queries after preloading using one-by-one insertions. For each of the five workload types, a spectrum of different data ingestion orders are generated using the (K, L)-sortedness metric and tested to quantify the combined impact of data sortedness and access pattern types. We highlight that BoDS tests both **bulk insertion** and **transactional read-/write mixed workloads** with a **varying degree of (ingestion) sortedness**. As an example, we benchmark PostgreSQL, a state-of-the-art production-grade database system, and present key observations regarding adapting data systems and indexing to be sortedness-aware.

2 Data Sortedness Metrics

In order to vary, study, and exploit data sortedness, we first need a way to quantify it. To that end, several metrics have been proposed and used in literature [5,8,12,14]. Some of these include *inversions* that measure the number of pairs in the incorrect order, *runs* that measure the number of contiguous increasing subsequences in the collection, and *exchanges* that quantify the least number of swaps needed to bring the data in order [12]. While these metrics are intuitive to quantify data sortedness, they have certain drawbacks that make them unsuitable to use in a sortedness benchmark for indexing. For example, *inversions* fail to capture global disorder where the data collection contains monotonically increasing sorted sequences, but the sequences are placed out of order; *runs* fail to capture local disorder, where each entry in the data collection can simply be swapped with its adjacent entry to establish total order [14].

(K, L)-Sortedness Metric. Ben-Moshe *et al.* proposed to quantify data sortedness using a combination of two parameters: K, which captures the number of elements that are out of order, and L, which captures the maximum displacement in the position of the out-of-order elements [5]. Essentially, the (K, L)-sortedness metric captures a data collection's sortedness in terms of *how many elements* (K)

[3] BoDS codebase: https://github.com/BU-DiSC/bods.

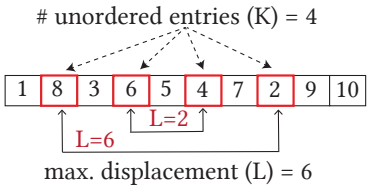

Fig. 2: A sample data collection having 10 elements with $K = 4$ and $L = 6$.

are in the *wrong* position, and crucially, *by how much* (L). The combination of both K and L in this metric underlines the *effort* it would take to establish total order in a data collection, while also overriding drawbacks of one-dimensional metrics (discussed above) with respect to global or local disorder. Fig. 2 visualizes the (K, L)-sortedness metric for a data collection of 10 elements with 4 elements out of order ($K = 4$) and a maximum displacement of 6 positions ($L = 6$). The original definition refers to L as maximum displacement, meaning that even if all out-of-order elements are off by one position and only one is off by a higher number, say max_disp, then $L = max\_disp$. When considering files and indexing, one element in a wrong location is not considered detrimental. Hence, we consider one more dimension of near-sortedness in our benchmarking metric: the *distribution of the displacement of the out-of-order entries*.

3 Generating (K, L)-Sorted Data

The most important part of the proposed benchmark is generating data collections with a varying degree of sortedness. To that end, we build a synthetic data generator that creates data collections adhering to specific values of the (K, L)-sortedness metric. The data generator takes as input user-specified values for the K and L parameters of the sortedness metric as a fraction of the total number of entries (N), as well as the displacement distribution (on L). The total size of the generated data collection can be controlled using the number of entries to be generated (N) and the payload size (P). The payload is a randomly generated string of a given size. Table 1 summarizes the input parameters to the workload generator.

Variable-Sortedness Data Generator. The data generator initially creates a fully sorted data collection and induces "unsortedness" as required. The overall process is described in detail in Algorithm 1. Unordered entries are generated

Parameter	Description
N	No. of entries in the data collection
P	Size of the payload for every key
K	No. of out-of-order entries in the data collection
L	Maximum displacement of an out-of-order entry
$B(\alpha, \beta)$	Beta distribution for displacement (on L)
S	Seed value
o	output directory path

Table 1: Overview of input arguments to the sortedness data generator.

Algorithm 1: Generate (K, L, B)-sorted keys

```
   Input: Fully sorted array arr, N ≥ 0; K ≥ 0; L ≥ 0; B(α, β), num_tries1 > 0, num_tries2 > 0
   Output: (K, L, B)-sorted array arr
 1 Sources ← Generate_Sources(N, K) ;                          /* using Algorithm 2 */
 2 dest <>;                                                    /* set of destinations */
 3 left <>;                                                    /* set of left out sources */
 4 for x ∈ Sources do
 5     while num_tries1 > 0 do
 6         r ← Pick_dest(N, K, x, B);                          /* using Algorithm 3 */
 7         num_tries1 ← num_tries1 − 1;
 8         if r ∈ dest or r ∈ Sources;                         /* destination already used */
 9          then
10             if num_tries1 == 0;                            /* retrials exhausted, moving r to leftovers */
11              then
12                 insert r to left;
13             end
14             continue;
15         else
16             insert r in dest;
17             swap arr[x] with arr[r];
18             break;
19         end
20     end
21 end
22 for x ∈ left;                                               /* randomized re-attempt for leftovers */
23  do
24     while num_tries2 > 0 do
25         r ← Pick_dest(N, K, x, B);                          /* using Algorithm 3 */
26         num_tries2 ← num_tries2 − 1;
27         if r ∈ dest or r ∈ Sources;                         /* destination already used */
28          then
29             continue;
30         else
31             insert r in dest;
32             swap arr[x] with arr[r];
33             remove x from left;
34             break;
35         end
36     end
37 end
38 Perform_Brute_Force(arr, left, dest, L) ;                   /* using Algorithm 4 */
```

by swapping elements, i.e., each swap generates two out-of-order elements that contribute to the K parameter. Thus, for a given K, the data generator first picks $K/2$ sources for swaps (Alg. 2), and for each source, a destination position (up to L positions away) is randomly picked to swap with (Alg. 3).

Algorithm 2: Generate $K/2$ swap sources

```
   Input: N ≥ 0; K ≥ 0
   Output: A set of source swaps X
 1 cnt ← 0;
 2 while cnt < K/2 do
 3     pick a random index r ∈ [0, n − 1];
 4     if r ∈ X then
 5         continue;
 6     else
 7         insert r in X;
 8         cnt + +;
 9     end
10 end
```

Displacement Distribution. While the (K, L)-sortedness metric quantifies the effort required to bring the data collection to fully-sorted order, it does

Algorithm 3: Pick a destination

```
   Input: N ≥ 0; K ≥ 0; source x; B(α, β)
   Output: Destination d
 1 low_jump ← −L;
 2 high_jump ← L;
 3 if position + high_jump ≥ N ;                    /* Sanity checks for out-of-bounds */
 4  then
 5  |  high_jump ← N − 1 − x;
 6 end
 7 if position + low_jump < N then
 8  |  low_jump ← −x;
 9 end
10 pick r ∈ [0, 1];
11 initialize beta distribution object distr(B);
12 rd ← quantile(distr, r);                         /* picks number in [0,1] range */
13 jump ← low_jump + ((high_jump − low_jump) ∗ rd);
14 ret ← position + jump;
```

Algorithm 4: Brute-Force Swap

```
   Input: Array arr, leftovers left, destination swaps dest, L
   Output: Array arr after swapping leftovers
 1 for x ∈ left;                                    /* final brute-force attempt for leftovers */
 2  do
 3  |  pick rnd ∈ [0, 1] ;                          /* coin toss for forward/backward run */
 4  |  if rnd < 0.5;                                /* move forward */
 5  |   then
 6  |  |  start ← x − L; end ← x + L;
 7  |  else
 8  |  |  start ← x + L; end ← x − L;               /* move backward */
 9  |  end
10  |  for r ∈ [start, end];                        /* loop and pick first valid spot */
11  |   do
12  |  |  if r ∈ dest or r == x or r ∈ X;          /* check for cascading swaps */
13  |  |   then
14  |  |  |  continue;
15  |  |  else
16  |  |  |  insert r in dest;
17  |  |  |  swap arr[x] with arr[r];
18  |  |  |  break;
19  |  |  end
20  |  end
21 end
```

not specify how the unordered entries are distributed within the data collection. The L parameter captures only the maximum displacement among all unordered entries, hence, we may have only one entry that is displaced by L, whereas other entries have a much smaller displacement. Thus, to offer fine-grained control on the distribution of the displacement among the unordered entries, we use an additional parameter to capture the distribution of data sortedness through a generalized beta distribution with fixed bounds between $-L$ and L. Fig. 3 shows examples of the probability density (PDF) of the beta distribution. Note that the beta distribution maps to uniform for $\alpha = \beta = 1$ (Fig. 3a) and that it maps to a variable degree of skewed distributions for different values of α and β (Fig. 3b, 3c and 3d). Alg. 3 is already capable of generating swaps with a distance that follows a user-defined distribution (lines 10-14). To do this, we use a generalized beta distribution with user-specified α and β values. For this, we use the C++ `boost` library that provides a beta distribution function and a quantile function for picking a number per the beta distribution using inverse transform sampling.

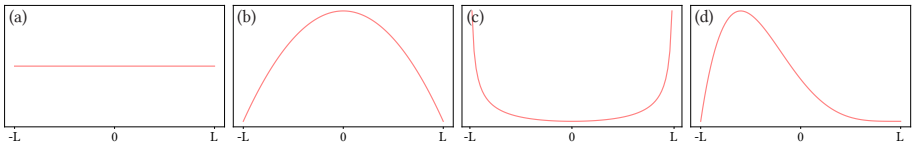

Fig. 3: Probability density (PDF) of β-distribution bounded between $[-L, L]$. Using the β-distribution offers fine-grained control on the L parameter. In (a) that the displacements are uniformly distributed ($\alpha = \beta = 1$), while in (b) the displacements are centered around the mean $= 0$ ($\alpha = \beta = 2$). Skewness in the distribution of displacements can be introduced like (c) and (d), where they are closer to the maximum displacement ($\alpha = \beta = 0.5$), or biased toward one direction ($\alpha = 2, \beta = 5$).

Examples of (K, L)-Sorted Data. The data generator is capable of generating data collections with variable K, L, and displacement distribution. For example, a dataset with either $K = 0\%$ or $L = 0\%$ is fully sorted. Similarly, a dataset with $K = 10\%$ and $L = 2\%$ will have 10% of its total entries out of order, and each out-of-order entry is displaced within a distance equivalent to 2% (at most) of the total entries from its ideal position. Figure 4 shows an example (K, L, B)-sorted data collections. Here, we have used $\alpha = \beta = 1$ (uniform distribution) for Fig. 4a−4d, while we use $\alpha = \beta = 0.5$ (skewed) for Fig. 4e−4h. Consequently, we observe that a higher number of unordered entries in Fig. 4e−4h are displaced by $\sim L$ as compared to the former set of figures.

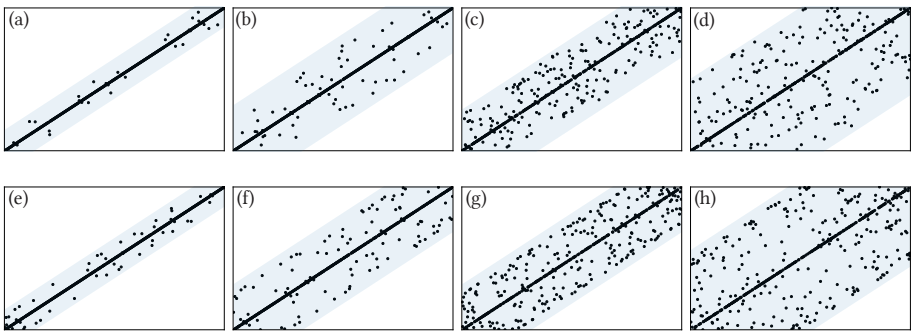

Fig. 4: Examples of benchmark data with different K and L combinations for data sortedness. The first row of visuals (a−d) are generated using a uniform distribution ($\alpha = \beta = 1$), while the second row of visuals (e−h) are generated using a skewed distribution concentrated close to the maximum displacement ($\alpha = \beta = 0.5$). The following figures correspond to sortedness levels: for the following sortedness levels: (a)&(e) K=10%, L=10%, (b)&(f) K=20%, L=25%, (c)&(g) K=50%, L=25%, (d)&(h) K=50%, L=50%.

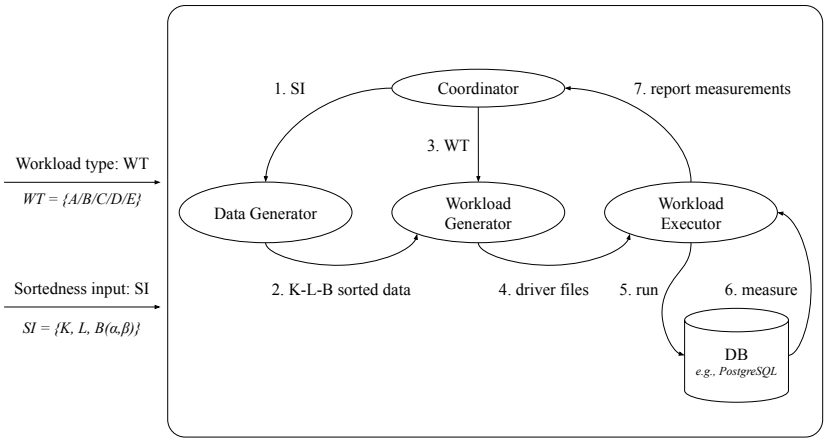

Fig. 5: High-level architecture of the proposed Benchmark on Data Sortedness (BoDS) workload instance that uses the (K, L)-sortedness metric.

4 The Benchmark on Data Sortedness

The Benchmark on Data Sortedness is a suite of workloads that compares data ingestion and transactional (mixed read/write) accesses on indexing data structures and data systems, for a variable degree of sortedness. In this section, we put together all the pieces introduced earlier to present the architecture of the benchmark and its important components.

Overall Architecture. A BoDS deployment consists of five principal components: (i) the coordinator, (ii) the data generator, (iii) the workload generator, (iv) the workload executor and (v) the tested database system. When running an instance of the benchmark, we first decide the workload type which controls whether data ingestion is performed as bulk loading or via individual inserts and whether there is a mix of reads and writes to the system. Next, a data file is created per the degree of sortedness given by the K, L, and $B(\alpha, \beta)$ input to the data generator as described in Section 3. The workload generator then takes as input the data file and the workload type to generate driver files that prepare the workload according to the system's interface. When testing full-blown relational systems, the system driver files contain SQL statements. A file named `preload.sql` loads the initial data and preconditions the database, and a second file named `operations.sql` contains the interleaved inserts or queries. Finally, the workload executor executes the driver files on the tested system and monitors their runtime. To try variable data sortedness, the workload executor is re-instantiated with different (K, L) inputs. Figure 5 illustrates the high-level architecture of a BoDS workload instance.

Supported Workloads. We briefly summarize the workload types in Table 2. Specifically, the benchmark supports five workload types:

Workload	Data Loading		Operations	
	Method	% of data	R/W ratio	% of data
A	BL	100%	-	-
B	II	100%	-	-
C	-	0%	83%/17%	100%
D	BL	80%	50%/50%	20%
E	II	80%	50%/50%	20%

Table 2: Overview of workloads supported by the Sortedness Benchmark. Workloads A and D use Bulk loading (BL) for the data loading phase, while workloads B and E use Individual inserts (II).

(A) bulk load: where we insert data into the system using the bulk load functionality (e.g., copy in PostgreSQL);
(B) individual inserts: where we ingest data into the system one by one;
(C) mixed inserts and read queries with no preloading: where we insert data interleaved with (20%) reads;
(D) mixed inserts and read queries after bulk loading (a combination of A and C): where we pre-load the system with a portion of the data using bulk loading and perform interleaved reads and writes;
(E) mixed inserts and read queries after individual inserts (a combination of B and C): where we pre-load the system with a portion of the data using individual insertions and perform interleaved reads and writes;

For the mixed workloads (D) and (E), we use 80% of the to-be-ingested data to preload the system and perform mixed reads and writes on the remaining data. Also, the lookup keys are uniformly chosen and may consist of both empty and non-empty queries within the key domain. Essentially, workload C model performance during the initialization-state of the index, while workloads D and E capture the steady-state performance.

5 BoDS in Action

We run the Benchmark on Data Sortedness (BoDS) on PostgreSQL, a popular row-store and present key observations regarding its performance when ingesting nearly-sorted data.

Experimental Setup. We run the experiments on Amazon Web Services (AWS) `EC2` instances of `t2.medium` instance type. Each instance has 2 virtual Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz CPUs, 4GB DIMM RAM and 40GB root storage using general purpose SSDs (`gp2`) with 120 provisioned IOPS on EBS storage. The instances run Ubuntu Server 22.04 LTS (HVM) that use a 64-bit (x86) architecture.

Default System and Index Setup. We use PostgreSQL 14.3 to execute the benchmark with a modified buffer pool (shared buffer) space of 1GB. In all experiments, we use *unlogged* tables to avoid overheads due to write-ahead logging

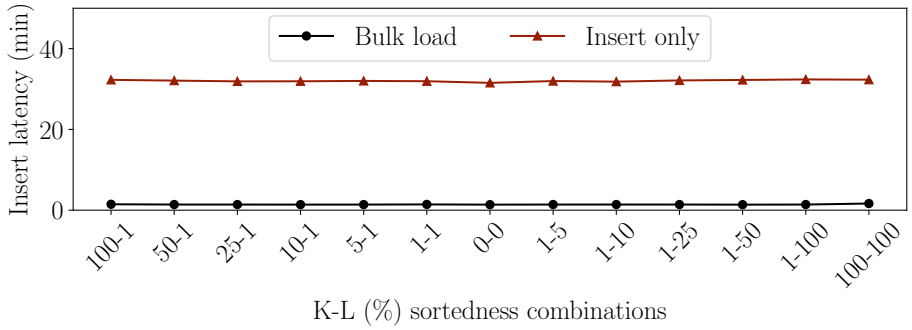

Fig. 6: Comparison of bulk loading and insert performance of PostgreSQL with near-sorted data of 16M rows (4GiB).

(WAL) and isolate index performance. We ingest data in a table containing two attributes: (i) `id_col`, and (ii) `payload`. A B-tree is created (before data ingestion/loading) on the `id_col` attribute. In each experiment, we drop the existing table (if it exists) along with its corresponding index, and recreate them.

Default Data Setup. We create multiple data collections (of size 4GB) with varying K and L values using a uniform distribution ($\alpha = \beta = 1$). Each data collection is created with 16M key-value pairs having an entry size of 256B, where each key is 4B and the payload is 252B.

Default Testing Suite. BoDS supports execution of any combination of (K, L) and $B(\alpha, \beta)$ for a particular workload type. By default, we run the benchmark for the following (K, L)-near-sorted combinations: $(100, 1)$, $(50, 1)$, $(25, 1)$, $(10, 1)$, $(5, 1)$, $(1, 1)$, $(0, 0)$, $(1, 5)$, $(1, 10)$, $(1, 25)$, $(1, 50)$, $(1, 100)$, and $(100, 100)$. This way, we ensure to compare the systems for a spectrum of near-sortedness, in addition to the two extremes of *fully-sorted* data and *unsorted* data.

Evaluation Metrics. We measure using BoDS the performance of PostgreSQL measuring: (i) ingestion latency, and (ii) overall operational latency in case of a mixed workload with reads and writes.

5.1 Raw Ingestion Performance

In this set of experiments, we compare the performance of the PostgreSQL (Postgres) system while bulk loading and one-by-one insert into a database table using a B-tree index. For bulk loading, we use the `COPY` command to load the table with the entire data file. In the case of individual inserts, we write an `INSERT` command for each row of the data collection. In both cases, the commands are written to the `load.sql` file, and we measure the overall execution time while executing this file on Postgres. Fig. 6 shows the comparison of bulk loading and individual inserts with near-sorted data, using a B-tree index for Postgres.

Bulk Loading is Extremely Fast. As expected, we observe from Fig. 6 that bulk loading (black line) in PostgreSQL is extremely fast, with an average ingestion latency of $5.62\mu s$ per inserted row. This is because bulk loading populates

the index bottom-up by first creating fully-occupied leaf and internal nodes of the b-tree index. This allows PostgreSQL to avoid expensive node splits for both leaf and internal nodes as well as re-organizing the data layout. However, bulk loading the data entails sorting the entire collection up front. This is why ingesting a fully sorted (K=L=0) data set ($82s$) takes 19.51% less time as compared to ingesting unsorted (K=L=100) data ($98s$). Note that this improvement in performance is owing only to a smaller sorting overhead before bulk loading.

PostgreSQL is Agnostic Toward Data Sortedness. Fig. 6 (red line) also shows that when performing individual insertions, PostgreSQL is unable to take advantage of any intermediary data sortedness. The sortedness-agnostic ingestion performance of PostgreSQL can be attributed to the underlying B-tree index, the construction cost of which is $O(log_F N)$ (since there is no bulk loading happening in this case) due to tree-traversal. The B-tree does not use inherent *sortedness* in already sorted or near-sorted data to reduce the ingestion cost, rather, ends up doing *extra* work in establishing order in data that already had some degree of inherent sortedness. Thus, regardless of the data sortedness, PostgreSQL ends up spending $\sim 20\times$ ($108\mu s$) more time during data ingestion through individual insertions compared to the bulk loading time.

Column-Store Systems Require a Fundamental Redesign to Support Data Sortedness. Most column-store systems primarily optimize for the read performance of analytical queries through vertical partitioning, vectorization, compression, tight for-loops, and cache efficiency and do not rely on secondary indexes [1]. Further, when loading data, one cannot enforce the system to maintain the data fully sorted (similar to what a secondary B-tree index would do in a row-store), hence, column-store systems would require fundamental design changes to try and accommodate a variable degree of data sortedness. For example, MonetDB supports two types of indexes, *imprints* [17] and *ordered indexes* (essentially a sorted version of the desired columns), which are both invalidated after any insert, update, or delete on the corresponding tables [15]. We run workloads A and B on MonetDB using the ordered index with auto-commit off. As we vary the underlying sortedness for each workload, we observe that the runtime does not change since we only pay the cost of vertically partitioning the incoming data and populating the table's columns. Note that the index is invalidated after the first update and is not maintained thereafter.

We now briefly discuss two commercial column-store systems, Vertica and Actian Vector. Vertica supports *projections*, which are similar to ordered indexes and do not support live updates [13]. Hence, we expect to have similar behavior to MonetDB. Actian Vector supports live updates in a sorted column through an approach called Positional Delta Tree (PDT) [11]. PDT is essentially a variation of an in-memory B-tree with positional information. Hence, based on our experimentation with an in-memory B-tree [16], we do not expect significant performance differences when varying sortedness. However, more experimentation is needed to fully assess Vertica's and Actian Vector's capability to exploit sortedness, which we leave as future work.

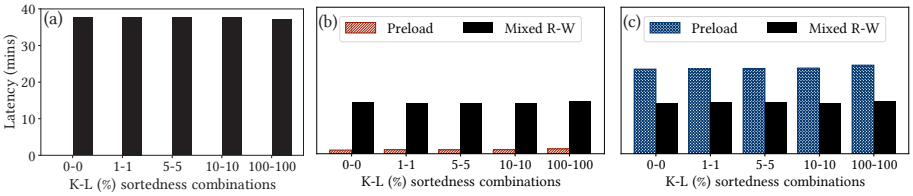

Fig. 7: Performance of PostgreSQL with mixed workloads: (a) Mixed workload with no pre-loading (b) Mixed workload using bulk loading for preloading (c) Mixed workload using individual inserts for preloading.

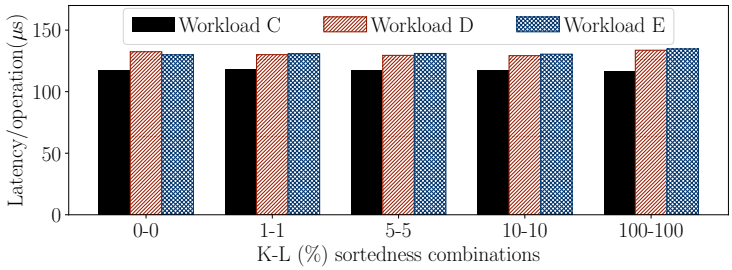

Fig. 8: Comparison of latency per operation among the mixed workloads C, D and E.

5.2 Mixed Workload Performance

In this set of experiments, we compare the performance of PostgreSQL under the three mixed workload settings supported by our benchmark: (i) mixed with no preloading (workload C); (ii) mixed after bulk load (workload D); and (iii) mixed after individual insertions (workload E). For workload C, we perform 16M insertions interleaved uniformly with 3.2M point queries, while for the workloads D and E we first preload the system with 12.8M data rows and then perform 3.2M inserts interleaved uniformly with 3.2M point queries. Again, the *preloading phase* is written to an intermediary file `preload.sql` that is either empty (for workload C) or contains a `COPY` command (for workload D) or 12.8M `INSERT` statements (for workload E); while in the *operations phase* the `operations.sql` file contains the interleaved `INSERT` and `SELECT` statements. We then execute both files on PostgreSQL and measure the workload execution latency. Fig. 7 shows the comparison of phase-wise execution time for all three workloads.

PostgreSQL Cannot Harness Sortedness as a Resource. When ingesting a mixed workload with uniformly interleaved reads and writes, even for a completely sorted data collection (K=L=0) the performance of PostgreSQL mimics the performance of ingesting unsorted data (K=L=100), as seen from Fig. 7(a). When performing individual insertions, the underlying B-tree index is unable to identify if the ingested data is completely sorted. This is because state-of-the-art index structures lack a mechanism to assess data sortedness of ingested entries

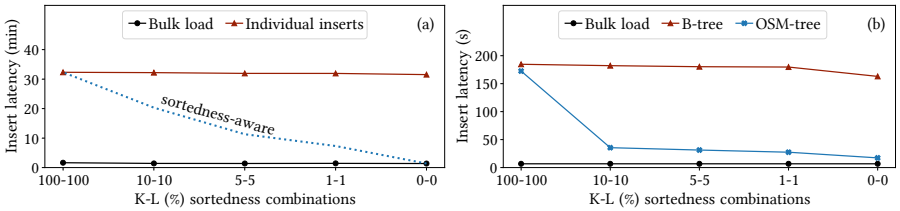

Fig. 9: (a) A sortedness-aware system should adapt to data characteristics. (b) OSM-tree index is an example of an index that adapts to data sortedness to offer favorable ingestion.

on the fly. In fact, this trend can also be observed in the operational latency for workload D, as well as both phase-wise latencies in workload E.

Fig. 8 shows the average latency-per-operation for mixed workloads C, D, and E as we vary the data sortedness. While the average operational latency remains largely unaffected by data sortedness, we observe that workload C shows ∼15-16% lower latency as compared to mixed workloads D and E. Workload C contains no preloading phase, and hence, initial operations are performed on a smaller database (and subsequently, a smaller index) by size. On the other hand, workloads D and E do contain a preloading phase where the database and index are warmed-up with 3.2GB data (80% of 4GB), and thus, every insert or query needs to traverse the index which exacerbates the operational cost. Note that workload C represents the initialization-state of the index/database, while workloads D and E represent the steady-state.

6 Toward Sortedness Awareness

From our experiments with PostgreSQL, we have observed that the system is *unsurprisingly* not sortedness-aware. Fig. 9a summarizes the results from §5, where we maintain a B-tree index during data ingestion with increasing sortedness. PostgreSQL is agnostic to sortedness due to the inability of the underlying B-tree to *use data sortedness as a resource*. We highlight the vast headroom for improvement during index construction by comparing the individual insert latency (red line in Fig. 9a) against the bulk loading latency (black line). As we pointed out in §5, state-of-the-art index data structures like B-tree lack the means to assess sortedness on the fly during ingestion and end up paying the standard construction cost (i.e., worst-case performance) even for nearly sorted data. Ideally, a sortedness-aware index and, in turn, a sortedness-aware system should *lower the index construction cost* when ingesting data with increasing sortedness and, thus, follow a trend similar to the dotted blue line in Fig. 9a.

Existing literature on indexes does not explore sortedness when optimizing index construction. Providing classical indexes with the means to capture data sortedness during ingestion (most notably, buffering) would incur a read overhead. Hence, we expect a tradeoff between optimizing index construction by

harnessing data sortedness and read performance. However, an ideal sortedness-aware index should be able to navigate this performance tradeoff. It should offer near-optimal ingestion (bulk loading) in the presence of high data sortedness, while falling back to the current baseline otherwise. Further, it should amortize any read overheads incurred to offer better overall performance. A first design that uses sortedness as a resource is the OSM-tree [16] which uses a buffer to capture data sortedness in memory and to maximize bulk loading. Fig. 9b shows that the OSM-tree acts like a B-tree when ingesting unsorted data (left-end of the x-axis), and mimics bulk loading when ingesting fully sorted data (right-end of the x-axis), while bridging the gap for data sortedness between the two extremes. Similarly to the experiments with PostgreSQL, we ingest 16M keys and allocate a buffer pool of 25% of the total data size to both OSM-tree and B-tree. We use workload A for the bulk loading line, and workload B for the individual inserts to the B-tree and the OSM-tree, to which we allocate an in-memory buffer of 1% of the total data size. This small in-memory buffer investment, along with the other design elements of OSM-tree (e.g., partial opportunistic bulk loading), leads to $\sim$9$\times$ faster ingestion for fully sorted data, making individual inserts almost as efficient as bulk loading. Even for lower data sortedness (K=L=10%), OSM-tree offers a $\sim$5$\times$ improvement over B-tree, while mimicking the baseline for unsorted data. Overall, having a sortedness-aware design will allow data systems to build their indexes faster, e.g., via batching inserts that can be opportunistically bulk loaded. Such a sortedness-aware system may offer orders of magnitude better ingestion performance in the presence of data sortedness, which will be beneficial even for mixed read/write workloads.

7 Conclusion

The Benchmark on Data Sortedness lays the groundwork to test systems and index data structures that will be designed to harness data sortedness. We expect the trend shown in Fig. 9 to match the observed performance for sortedness-aware indexing data structures and systems, with variations that will correspond to the degree each approach is optimized for interleaving reads and writes, accurately capturing sortedness, and memory availability. Overall, we expect that new sortedness-aware data structure designs and data systems will emerge and will employ BoDS to show how they cope with variable data sortedness.

Acknowledgements. This work is funded by NSF Grants IIS-2144547 and IIS-1850202, a Facebook Faculty Research Award, and a Meta gift.

References

1. Abadi, D.J., Boncz, P.A., Harizopoulos, S.: Column-oriented Database Systems. Proceedings of the VLDB Endowment **2**(2), 1664–1665 (2009), `http://dl.acm.org/citation.cfm?id=1687553.1687625`
2. Achakeev, D., Seeger, B.: Efficient Bulk Updates on Multiversion B-trees. Proceedings of the VLDB Endowment **6**(14), 1834–1845 (2013), `http://www.vldb.org/pvldb/vol6/p1834-achakeev.pdf`

3. Athanassoulis, M., Ailamaki, A.: BF-Tree: Approximate Tree Indexing. Proceedings of the VLDB Endowment **7**(14), 1881–1892 (2014), http://www.vldb.org/pvldb/vol7/p1881-athanassoulis.pdf
4. Athanassoulis, M., Yan, Z., Idreos, S.: UpBit: Scalable In-Memory Updatable Bitmap Indexing. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (2016), https://dl.acm.org/citation.cfm?id=2915964
5. Ben-Moshe, S., Kanza, Y., Fischer, E., Matsliah, A., Fischer, M., Staelin, C.: Detecting and Exploiting Near-Sortedness for Efficient Relational Query Evaluation. In: Proceedings of the International Conference on Database Theory (ICDT). pp. 256–267 (2011), http://doi.acm.org/10.1145/1938551.1938584
6. Bender, M.A., Farach-Colton, M., Jannen, W., Johnson, R., Kuszmaul, B.C., Porter, D.E., Yuan, J., Zhan, Y.: An Introduction to Bϵ-trees and Write-Optimization. White Paper (2015), http://supertech.csail.mit.edu/papers/BenderFaJa15.pdf
7. den Bercken, J.V., Seeger, B.: An Evaluation of Generic Bulk Loading Techniques. In: Proceedings of the International Conference on Very Large Data Bases (VLDB). pp. 461–470 (2001), http://www.vldb.org/conf/2001/P461.pdf
8. Carlsson, S., Chen, J.: On Partitions and Presortedness of Sequences. In: Acta Informatica. vol. 29, pp. 267–280 (1992), https://doi.org/10.1007/BF01185681
9. Graefe, G.: B-tree indexes, interpolation search, and skew. In: Proceedings of the International Workshop on Data Management on New Hardware (DAMON) (2006), http://dl.acm.org/citation.cfm?id=1140402.1140409
10. Graefe, G.: Modern B-Tree Techniques. Foundations and Trends in Databases **3**(4), 203–402 (2011), http://dx.doi.org/10.1561/1900000028
11. Héman, S., Zukowski, M., Nes, N.J.: Positional Update Handling in Column Stores. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. pp. 543–554 (2010), http://dl.acm.org/citation.cfm?id=1807167.1807227
12. Knuth, D.E.: The art of computer programming, Volume I: Fundamental Algorithms (3rd Edition). Addison-Wesley (1997), http://www.worldcat.org/oclc/312910844
13. Lamb, A., Fuller, M., Varadarajan, R.: The Vertica Analytic Database: C-Store 7 Years Later. Proceedings of the VLDB Endowment **5**(12), 1790–1801 (2012), http://dl.acm.org/citation.cfm?id=2367518
14. Mannila, H.: Measures of Presortedness and Optimal Sorting Algorithms. IEEE Transactions on Computers (TC) **34**(4), 318–325 (1985), https://doi.org/10.1109/TC.1985.5009382
15. MonetDB: Index Definitions. https://www.monetdb.org/documentation-Jan2022/user-guide/sql-manual/data-definition/index-definitions/ (2022)
16. Raman, A., Sarkar, S., Olma, M., Athanassoulis, M.: OSM-tree: A Sortedness-Aware Index. CoRR **abs/2202.0** (2022), https://arxiv.org/abs/2202.04185
17. Sidirourgos, L., Kersten, M.L.: Column Imprints: A Secondary Index Structure. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. pp. 893–904 (2013), http://dl.acm.org/citation.cfm?id=2463676.2465306
18. Stonebraker, M.: The Case for Partial Indexes. ACM SIGMOD Record **18**(4), 4–11 (1989), https://doi.org/10.1145/74120.74121
19. TPC: TPC-H benchmark. http://www.tpc.org/tpch/ (2021)