Distributed and Parallel Databases
https://doi.org/10.1007/s10619-018-7252-2

PLI$^+$: Efficient Clustering of Cloud Databases

**Dai Hai Ton That · James Wagner ·
Alexander Rasin · Tanu Malik**

Received: January 15, 2018 / Accepted: September 25, 2018

Abstract Commercial cloud database services increase availability of data and provide reliable access to data. Routine database maintenance tasks such as clustering, however, increase the costs of hosting data on commercial cloud instances. Clustering causes an I/O burst; clustering in one-shot depletes I/O credit accumulated by an instance and increases the cost of hosting data. An unclustered database decreases query performance by scanning large amounts of data, gradually depleting I/O credits.

In this paper, we introduce Physical Location Index Plus (PLI^+), an indexing method for databases hosted on commercial cloud. PLI^+ relies on internal knowledge of data layout; it builds a physical location index, which maps a range of physical co-locations with a range of attribute values to create approximately sorted buckets. As new data is inserted, writes are partitioned in memory based on incoming data distribution. The data is written to physical locations on disk in block-based partitions to favor large granularity I/O. Incoming SQL queries on indexed attribute values are rewritten in terms of the physical location ranges. As a result, PLI^+ does not decrease query performance on an unclustered cloud database instance; DBAs may choose to cluster the instance when they have sufficiently large I/O credit available for clustering thus delaying the need for clustering. We evaluate query performance over PLI^+ by comparing it with clustered, unclustered (secondary) indexes, and log-structured merge trees on real datasets. Experiments show that PLI^+ significantly delays clustering, and yet does not degrade query performance – thus achieving higher level of sortedness than unclustered indexes and log-structured merge trees. We also evaluate the quality of clustering by introducing a measure of interval sortedness, and the size of index.

Keywords clustered indexes, relational databases, scientific data and computing

College of Computing and Digital Media, DePaul University, Chicago, IL, USA
E-mail: dtonthat@depaul.edu, jwagne32@depaul.edu, arasin@depaul.edu, tmalik1@depaul.edu

1 Introduction

Database management systems (DBMS) offer users a secure and a reliable solution to store and share data. There is an increased interest in hosting data within the cloud. Cloud-based DBMSes[1] provide a cost effective solution by allocating processing and storage resources on demand and deallocating these resources when they are not needed. This is particularly evident for scientific databases, such as Plenar.io [8], the 1000 Genomes Project [9], GenBank [7], and the NASA NEX [18]. Scientific databases such as these sporadically ingest high volumes of data, and require high availability and support for multi-user access. For example, if a scientific database stores information about road traffic, it is reasonable to expect that a high volume of data will be collected around rush hour and a low volume of data will be collected at midnight. Therefore, there are times when the DBMS requires increased resources to load and manage data, but there are also periods of time when these resources are unnecessary.

Even though cloud-based DBMSes offer a cost-effective scaling solution for such databases, the cost of these services can still be prohibitive due to maintenance demands. The majority of cloud-based DBMS vendors charge users based on the number of I/O operations performed, in addition to the storage and processing costs. Therefore, it is important for a cloud-based DBMS to efficiently manage storage to reduce I/O operations.

Clustering is a common and well-known technique used in DBMSes to physically sort data in persistent storage. When a table is clustered (or sorted), records are read with minimal I/O operations. Therefore, it is reasonable to claim that clustering will reduce costs for a cloud-based DBMS. Section 2 provides an overview of the different clustering methods supported by DBMSes.

While clustering results in optimal I/O costs for the read-only queries, there remains a steep trade-off in the number of I/O operations needed to maintain a clustered table. Most relational databases implement clustering as an external merge sort operation, which has a worst-case disk I/O cost of $2*N*[\lceil log_{B-1}\lceil \frac{N}{B} \rceil \rceil + 1]$ for a table of size N blocks [22]. Thus every clustering maintenance operation is I/O and memory-intensive, causing at least double the number of I/Os based on the size of the database and available memory, and often much higher due to temporary indexes created during clustering. In a non-cloud DBMS, this entire operation is assumed to be free of monetary cost. When databases are hosted on the cloud, clustering generates an I/O burst, depleting available I/O credits of the cloud instance. If tables are to be clustered frequently, which is inevitable when new datasets are inserted, instances must be either over-provisioned in capacity adding to hosting costs or the application throughput must be capped when credits deplete. For example, the costs can be as high as \$1000 per month for modest size databases (100 - 500GB). For funds and resource-crunched data-intensive science projects,

[1] A DBMS deployed on a cloud platform

these costs can soon add up to be significant. Section 3 demonstrates the high monetary costs needed to implement clustering on a cloud-based DBMS.

Example 1 A federal law enforcement agency wants to study trends in crime around the nation. The Plenar.io [8] dataset, which is stored on a cloud-based DBMS, serves as a valuable resource since it stores crime statistics for cities around the world. The federal agents primarily access this crime data based on the geographical attributes. Therefore, it would be reasonable to cluster (or sort) the crime data based on the geographical attributes to reduce the I/O for the queries issued by the federal agents. Assuming that crimes continuously happen around the country (or city), data would be ingested based on the order in which the events occur. To support efficient query responses for the federal agents, the DBMS must cluster (or re-sort) the table as it is ingested. While the I/O remains low for the read queries issued by federal agent, the DBMS requires a high number of I/O operations to re-order the data based on the geographical attributes. This high number of I/O operations increases the monetary costs to host the DBMS on a cloud service.

Our previous work [30] sought to address the problem demonstrated in Example 1 with an index structure called a **physical location index** (PLI). PLI allowed for a *delayed* sorting approach in that as data was ingested, it would be appended to a table by an insertion order. PLI maintains a mapping between approximate physical locations and the values ingested. This approach does not experience any costs to organize the data. So for example, if it takes 100 seconds to scan a table and a query accessed 5% of the table based on the sorted attribute, then a PLI could provide a query runtime of approximately 5 seconds. If the table size grew by 10%, then the query that accessed 5% of the table based on the sorted attribute would now take approximately 15 seconds. However, the read query response time can quickly decay as large amounts of data is continuously ingested, such as in Plenar.io. Section 4 explains how PLI is deployed and used.

In this paper, we expand upon our previous work from [30] by proposing a solution called PLI^+. PLI^+ builds upon PLI in the following ways: 1) it achieves the optimal read query runtimes and I/O provided by a PLI and native clustering in spite of large number of inserts, and 2) it supports a delayed clustering approach that does not require excessive I/O, yet does not experience the query degradation observed for a PLI. We discuss our proposed solution, PLI^+, in Sect. 5. We thoroughly evaluate PLI^+ in Sect. 6. Our contributions are as follows:

- We analyze the costs associated with implementing clustering for a cloud-based DBMS. This demonstrates that clustering produces unexpected monetary costs due to high number of I/O operations. (Sect. 3)
- We describe our previous work on PLI including its advantages and shortcomings. PLI offers competitive runtimes for read queries, and delays clustering maintenance approach. (Sect. 4)
- We describe our proposed solution, PLI^+, which builds upon PLI. PLI^+ also takes a delayed clustering approach, but it additionally buffers incom-

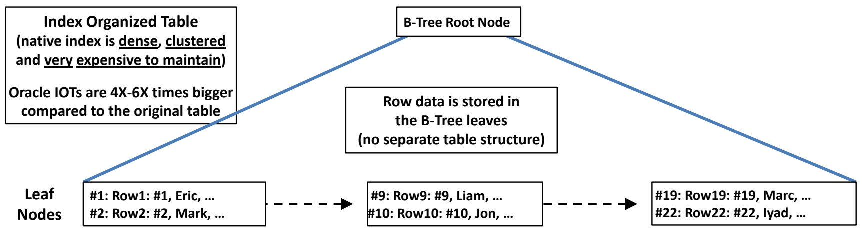

Index Organized Table
(native index is dense, clustered and very expensive to maintain)
Oracle IOTs are 4X-6X times bigger compared to the original table
B-Tree Root Node
Row data is stored in the B-Tree leaves (no separate table structure)
Leaf Nodes
#1: Row1: #1, Eric, …
#2: Row2: #2, Mark, …
#9: Row9: #9, Liam, …
#10: Row10: #10, Jon, …
#19: Row19: #19, Marc, …
#22: Row22: #22, Iyad, …

Fig. 1: Storage layout of native database clustered index based on an IOT.

ing data and performs in memory sorting, which avoids the query degradation of PLI. (Sect. 5)
– We provide a detailed and sound experimental analysis of PLI^+ including a thorough comparison with Log-structured Merge (LSM) trees, secondary, and clustered indexes. (Sect. 6)

2 Related Work

In this section, we review clustering-based methods as provided by cloud database services, and as described in current research.

2.1 Database systems on the cloud

Clustering in a cloud database engines depends upon the type of provisioned databases instances. We classify them as B-Tree and non-B-Tree based systems.

B-tree-based database systems (e.g., Postgres, DB2) store related items logically adjacent in the B-tree, but B-tree structure does not guarantee that logically-adjacent items will be physically adjacent. As a B-tree ages, leaves become scattered across the disk due to node splits from insertions and node merges from deletions. In an aged B-tree, there is little correlation between the logical and physical order of the leaves, and the cost of reading a new leaf involves both the data-transfer cost and the seek cost. The only way to improve the correlation is by manually clustering the database.

Several DBMSes (e.g., Oracle and MySQL) implement *Index Organized Tables* (IOT) [20], an augmented B-Tree structure that simultaneously serves as a clustered table. Instead of maintaining two independent database structures, a table and its index, IOT is a merged structure with rows of the table spliced into the leaf nodes of the B-Tree index itself. When new rows are inserted, IOTs maintain logical clustering as table data is stored along the leaves of the IOT structure. However, this solution comes at the price of slower inserts and a deteriorating read access performance. The leaves of the B-Tree data structure form a logically sorted linked list (as shown in Figure 1), which is not guaranteed to maintain a physical ordering as a clustered table does. For example, after a B-Tree node overflows, two split nodes may have to be written in a different physical location on disk. Thus, IOTs incur the overheads of other B-tree-based indexing systems, with the additional storage and access overhead compared to a regular B-Tree, due to the additional row data incorporated into the leaf nodes. Additionally, an IOT can only be organized on the primary key, while a PLI^+index can be built on any column(s).

2.2 Clustering with Write-optimized Indexing

Write-optimized indexes, such as log-structured merge trees [19] and its variants [17, 26] maintain local clustering of blocks at multiple levels, each of which is organized as a sorted sequential structure for the purpose of efficient lookups. Incoming writes are first sorted in memory and when full, data is merged into the first level on the disk. When the first level is full, its data will be gradually merged to the second level, and so on. The entire clustering process is a sequence of merges level by level. During a merge, only sequential I/O operations are involved. However, since all levels are sorted separately, and key spaces of different levels can overlap, LSM trees incur random I/O during query time. Implementations of LSM-trees make point queries efficient by using Bloom-filters which help search for levels that contain the keys. However, range queries remain slow since searches for each point need to be performed in each level.

Since write-optimized indexing requires time to search, B$^\epsilon$-trees use ϵ amount of space within internal for searching [13]. Here ϵ is a tunable parameter that selects how much space internal nodes use for searching. B$^\epsilon$-trees experimentally show better read/write performance than IOT structures. Unlike LSM and B$^\epsilon$-trees, which strictly sort the data (with merge sort), in $PLI^+$the goal is to approximately sort the data. Thus, PLI^+maintains physical locations of blocks that contain the specific ranges of data (min and max in a block), giving read queries an advantage by reducing the number of seeks needed for range querying. However, this introduces additional overhead when new data is inserted. We describe a main-memory sorting technique that determines the physical location of where new data is inserted.

2.3 Clustering with Queries

There are several ways to reduce random I/O of the query workload. The ferris-wheel approach [32, 33] queues queries if they access data out of index order. Queries can also be used to determine the structure of an index. Generalized partial indexing builds unclustered indexes around records defined by the user, leaving some records not indexed [27]. In contrast, PLI and PLI^+ provide the benefit of indexing all records, and approximately sort the data across buckets. In both methods, index maintenance cost is reduced by only recording access or reorganizing data that benefits queries. Database cracking [15] indexes by reorganizing individual columns (DB cracking is proposed for column-stores); the column itself serves as an index, physically reorganized to speed up query access. The reorganization happens dynamically as columns are accessed by user queries. Similar to database cracking, PLI table data is organized across but not within individual buckets. Kimura et al. proposed dividing a table into buckets as a scan unit with a correlation map (CM) index [16]. Representing a table as a sequence of buckets of rows allows for a lightweight index structure which is easily cached, reducing costs of index storage and maintenance. Similar to CMs, our method records the ranges of values stored for each bucket and implements indexing with query rewrite. Unlike CMs, our method relies only on internal row identifier for query rewrite – while CMs require a built-in clustered index and the presence of correlation in data (specifically, a correlation between the indexed column and the clustering key is required).

2.4 Horizontal Partitioning

Horizontal partitioning (HP) methods [1, 2, 10] allow tables, indexes and views to be partitioned into a disjoint sets of rows physically stored separately. But PLI and PLI^+ use buckets in which key-range values overlap. Horizontal partitioning is also an offline processes that requires processing for changes to workloads or incoming data [14]. The indexing methods proposed in this article are online, light-weight indexes and can be created to satisfy a mixed point and range query workload. Some DBMSes support both a clustered index and a partitioned index. Similarly, a PLI (or PLI^+) can be used in parallel with a partitioned index.

3 Clustering on a cloud database instance

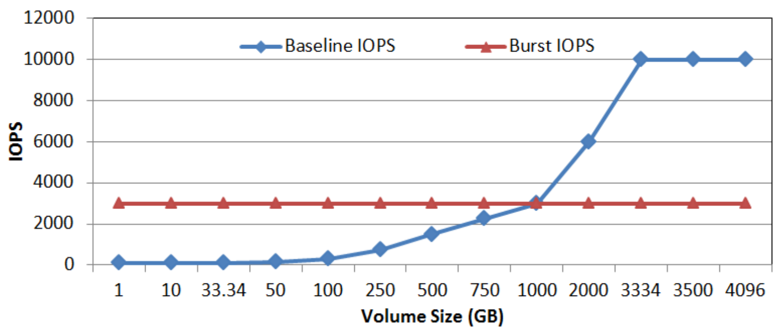


Fig. 2: The burst performance and baseline performance of different volume sizes on RDS [5]. Maximum infinite IOPS is at 10K IOPS.

In this section, we analyze the monetary costs of clustering relational database services (RDS) offered on the cloud. We use Amazon Web Services

(AWS) as an example to illustrate the high cost of clustering on the cloud. Other competing cloud database services (e.g., Microsoft Azure) use a similar pricing structure. The cost of storing data on the cloud (not including the cost of purchasing the Amazon EC2 instance) is twofold. First, there is a cost to purchase the disk space on cloud platform (storage cost). Second, there is a cost to access this data with high throughput (throughput cost). All cloud services, including AWS, adopt a burst model for I/O throughput measured in number of input and output operations per second (IOPS). Burst throughput is maximum throughput, which is available for a fixed period of time. Even though burst credit can be replenished after 24 hours, burst period is often short and cloud instance reverts to a baseline throughput after the burst period. Operating within the baseline throughput can dramatically degrade the clustering process, leading to the DBMS remaining blocked for a long period. Baseline throughput can be improved by extending the volume size (i.e., by over-provision). Figure 2 shows the relationship between the baseline throughput and burst throughput in terms of IOPS. The baseline throughput grows as the volume size increases. It's equal to burst throughput at 1TB and reaches a maximum IOPS at about 3TB (i.e., 10K IOPS). The burst model follows the pay-as-you-go model adopted for cloud services that charges for the amount of I/O capacity used [3]. The model, however, greatly penalizes the access pattern used by database maintenance tasks.

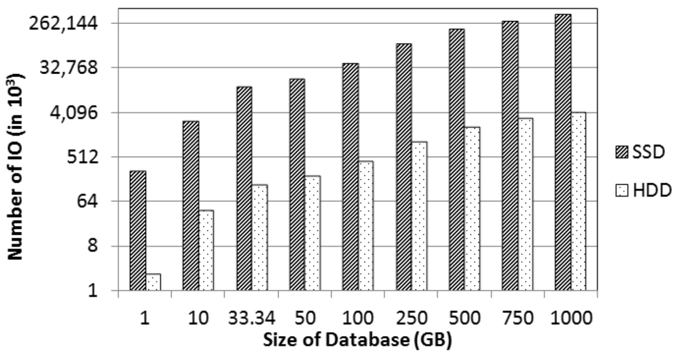


Fig. 3: Total number of I/Os of clustering on AWS with HDD and SSD storage (SSD based on 16KB I/O size, HDD based on 1MB I/O size).

The I/O cost of clustering using external merge sort algorithm requires $M = 2 * N * [\lceil log_{B-1} \lceil \frac{N}{B} \rceil \rceil + 1]$ I/O accesses, in which N is the number of pages in a table, and B is the size of main memory buffer available for external merge operation [22]. Figure 3 shows the cost of clustering for different table sizes in term of number of I/Os on a logarithmic scale. The I/O size is 16KB on SSD storage, and 1MB for HDD, resulting in different values of total numbers of I/Os for SSDs and HDDs in Fig. 3. Also note that we chose log base-2 for Y-axis as the best log scale to present results in most of our subsequent figures.

If database applications are conservative and under-provision by choosing an IOPS value based on database size, then the large number of I/O required for the clustering process will quickly deplete the burst throughput. The clustering after that is done at a baseline throughput, reducing query performance and requiring much larger time to cluster. In Figure 4, the solid line with squares shows the time different sized tables require to be clustered at a baseline throughput after depleting burst I/O credits. If the application over-provisions by using estimated high throughput I/O values, they must over-provision significantly to get maximum throughput for a period of time in which the entire clustering can finish. The solid line with triangles in Fig. 4 shows the time to cluster when the storage is over-provisioned. Alternatively, if storage is under-provisioned, clustering can take days to finish during which time query performance will be impacted. If the storage is over-provisioned to maximum throughput, a 100GB applications may need to pay for extra storage of about 3TB, while infinite IOPS are available at an extra cost of \$350 each month or \$4200 per year[2]. The results, reported for RDS operating on solid-state disk drives, remain similar if EC2 instances are used or if hard disk drives are used. In this paper, we show that PLI^+can amortize the clustering cost by initializing with a one time disk scan of the database and as little as a 100MB memory buffer.

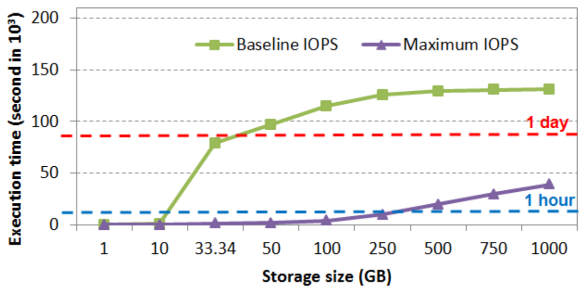


Fig. 4: Time to Cluster on AWS RDS with SSD storage.

4 PLI: A Physical Location Index for Delayed Clustering

We first describe the basic idea of PLI through an example, which we continue to refer to in the rest of this section. Consider Table T in Fig. 5 with attributes {`ID, Name`}. Let T be physically clustered on attribute {`ID`} into seven pages, i.e., the pages are in sequential order on the disk. The table also records the physical location of each row which is marked with an internal {`RowID`} column. Clustering this column on {`ID`} will sort the attribute {`ID`} and physically cluster the sorted result. Note that in order to minimize

[2] We use the cost estimates from [4]; Each GB at this time costs about \$0.125

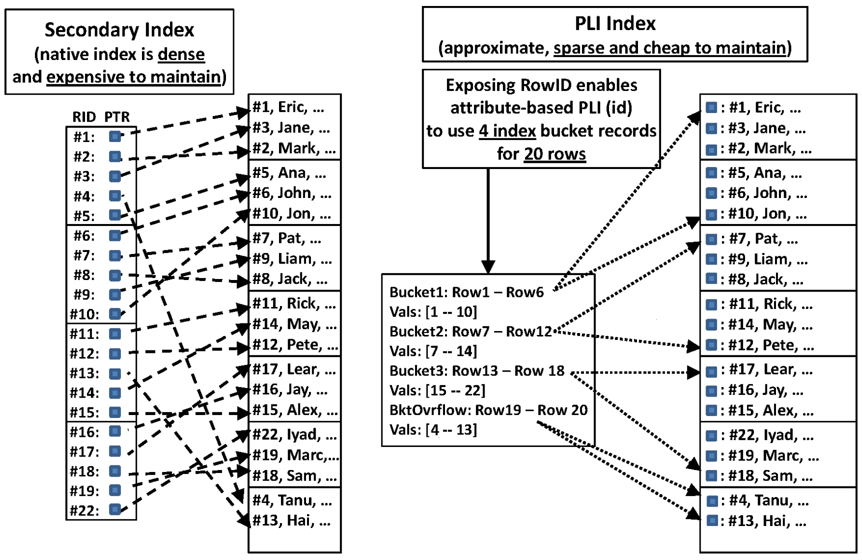

Secondary Index
(native index is dense and expensive to maintain)
RID PTR
PLI Index
(approximate, sparse and cheap to maintain)
Exposing RowID enables attribute-based PLI (id) to use 4 index bucket records for 20 rows
Bucket1: Row1 – Row6
Vals: [1 -- 10]
Bucket2: Row7 – Row12
Vals: [7 -- 14]
Bucket3: Row13 – Row 18
Vals: [15 -- 22]
BktOvrflow: Row19 – Row 20
Vals: [4 -- 13]

Fig. 5: Storage layout of native database indexes and *PLI*.

maintenance costs, the clustering on {`ID`} in Figure 5 example is not strict but rather approximate. Consider a query that accesses values based on `ID BETWEEN #1 and #6`. The secondary index will look up the matching keys, reading a number of index pages (intermediate levels) and two pages from leaf level of the index (incurring several seeks *before* accessing the table itself). First three pointers (Row1, Row3, Row2) will access the first page, which will be cached after the Row1 lookup. Fourth match (Row4) will require a seek and a read of a seventh page at the end. Finally, fifth and sixth match will correspond to pointers (Row5, Row6) causing yet another seek and reading of the second page in the table. A more efficient access path would recognize that five out of six matched values are in fact co-clustered in first two pages, with one outlier (#4) that resides in the overflow page and avoid seeking back and forth. While the above example assumes a separate B^+-Tree over the table, index-organized tables (IOTs) lead to a similar higher number of seeks for the same query.

The only way to take advantage of this seek reduction is by determining the level of physical co-clustering within attribute {`ID`}, information which is maximally available through the `RowID` column of the table. Thus for instance, if the database was indexed on `RowID`, with each range of `RowID` values consisting of six table rows, then such an index will quickly determine the physical co-clustering and lead to two seeks instead of three seeks. In general, the performance difference can be much larger. The sparse index on the right in Fig. 5 illustrates that fewer seeks are possible by knowing the state of physical clustering. Using this basic idea, *PLI* implements a sparse index and automatically rewrites SQL queries to include the database-internal row

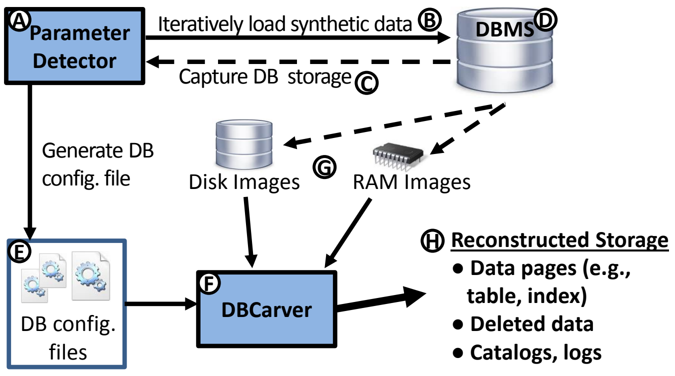


Fig. 6: Architecture of `DBCarver`.

identifier column to perform range scans. *PLI* can be implemented for both regular attributes, or order-preserving expressions on attributes.

4.1 Physical Data Organization

The physical layout of table data within a database file must to be known to build a *PLI*. In most DBMSes (e.g., Oracle and PostgreSQL), the location of rows can be determined by accessing the internal {`RowID`} column. However, the organization of data within a file may be different than that of the disk image since the file may not be written to consecutive sectors.

To verify the fragmentation of DBMS files on disk (fragmentation of the file cannot be determined through {`RowIDs`}), we use an implementation of database page carving, `DBCarver` [29]. Database page carving reconstructs the contents of relational database pages without relying on the file system or DBMS itself. This method is inspired by traditional file carving [23, 11] techniques that reconstruct data (active and deleted) from disk images or RAM snapshots without the need for a live system. Figure 6 provides an overview of `DBCarver`, which consists of two main components: the parameter collector(A) and the carver(F). The parameter detector calibrates `DBCarver` for the identification and reconstruction of DBMS pages. To do this, the parameter detector loads synthetic data(B) into a working version of the particular DBMS, and it captures underlying storage(C). The parameter detector then learns the layout of the database pages, and describes this layout with a set of parameters, which are written to a configuration file(E). For example, the parameter detector records the location of row directory, the endianness of addresses, and the size of each address (typically a 16-bit number) as parameters in the configuration file. A configuration file only needs to be generated once for each specific DBMS and version, and it is likely that a configuration file will work for multiple DBMS versions as page layout is rarely changed between versions. `DBCarver` has been tested against ten different databases: PostgreSQL, Oracle, SQLite, DB2, SQL Server, MySQL, Apache Derby, Firebird, Maria DB, and Greenplum. It can parse disk storage and describe the exact physical layout (based on disk address) of each database table.

4.2 *PLI* Structure and Maintenance

The structure of *PLI* is similar to that of a traditional sparse primary index. A regular sparse index will direct access to the correct page or sequence of pages instead of referencing particular rows. For example, in Figure 5, *PLI* consists of 3 buckets of approximately sorted data and an overflow bucket for a total of 20 rows in the table. Instead of storing 20 index entries, *PLI* only contains 4; the first bucket covers first two pages with six rows – *PLI* structure knows that all indexed values in that range are between #1 and #10 (without knowing the exact order) and can direct the query to scan this range if the predicate matches. The following two pages belong to bucket two which includes range between #7 to #14; note that approximate nature of sorting can result in overlap between buckets, e.g., *PLI* does not know whether #8 is in the first or second bucket and will direct the query to scan both buckets for this value. Thus, *PLI* can conceptually tolerate any amount of out-of-orderness, but performance will deteriorate accordingly. In addition to the indexed buckets, we also include the overflow bucket (values [5–13]) which contains recent inserts.

We next discuss maintenance costs. Interestingly, *PLI*'s approach requires *no maintenance* for deletes. Sparse bucket-based indexing knowingly permits false-positive matches that will be filtered out by the query after I/O was performed. Therefore, the index does not change when rows are deleted (e.g., in Figure 5, deletion of #6 will not change the first bucket in any way). Update queries can be viewed as `DELETE` + `INSERT`, permitting us to treat updates as inserts as well.

A new insert would typically be appended at the end of table storage, unless there is unallocated space on one of the existing pages and the database is willing to make in-place overwrite (Oracle has a setting to control page utilization, while PostgreSQL avoids in-place overwrite inserts). If the insert is appended, the overflow bucket needs to be updated only if the range of values in the bucket changes. For example, in Figure 5 overflow bucket is [5–13] and thus does not need to be changed when #10 is inserted into overflow.

There are several ways to determine the location of the newly inserted row to update *PLI* (RowID is the *internal* database identifier that reflects location of the row). Our current prototype queries the DBMS for it (`SELECT CTID` in PostgreSQL or `SELECT ROWID` in Oracle). However, for bulk inserts we can also use `DBCarver` to inspect the storage and determine the RowID ourselves. The new insert may overwrite a previously deleted row at any position (as we are avoiding maintenance overheads of clustering), which could potentially widen range of values in that bucket creating more false-positives on read access. The degradation is gradual, but eventually the table will need to be reorganized.

This storage reorganization can be done by targeting specific rows (executing commands that will cause out-of-order rows to be re-appended) or by resorting the whole table.

The storage size and the cost to maintain the *PLI* structure is proportional to the number of buckets that it uses. We have experimented with different granularities and bucket sizes – and, in practice, having a bucket of fewer than 12 disk pages does not improve query performance. Assuming about 80 rows per page, *PLI* structure only needs one bucket per one thousand (1000) rows. An index structure of this size can be kept in RAM and used or maintained at a negligible overhead cost.

4.3 Query rewrite

In order to use *PLI* index, incoming SQL queries are rewritten to take full advantage of the current layout of the table. *PLI*-based predicates are added to the query to restrict the disk scan range to specific buckets; bucket-based indexing is approximate by nature and provides a superset range in which data of interest resides. For example, consider Figure 5 – the following query predicate:

```
id BETWEEN #1 AND #6
```

is rewritten into:

```
id BETWEEN #1 AND #6
AND (CTID BETWEEN Row1 and Row6)
AND (CTID BETWEEN Row19 and Row20)
```

The first introduced condition represents a range of regular buckets (in that case the first bucket from *PLI*) and the second condition corresponds to the special overflow bucket. This access range results in a more efficient scan pattern of disk by minimizing seeks and by removing the overhead of a secondary index. The *PLI* condition does include false-positives (specifically, id #10 at Row6 and #13 at Row20) but they will be filtered out by the original query predicate (`id BETWEEN #1 AND #6`). *PLI* query rewrite relies on an internal RowID pseudo-column, exposed by nearly all DBMSes (known as ROWID in Oracle and CTID in PostgreSQL). In PostgreSQL (but not in Oracle), this internal pseudo-column should also be indexed for the efficient execution of *PLI*-rewritten queries; we note that in PostgreSQL 10 indexing of CTID column has been disallowed.

4.4 Architecture

The architecture of *PLI* operation is shown in Figure 7. We rely on the native database table(A) with no modifications or assumptions about DBMS engine features (e.g., underlying DBMS may not even support clustering). Initially, we use `DBCarver` to inspect table layout as it currently exists. As shown in [29], looking for specific pages in a table is orders of magnitude faster compared to

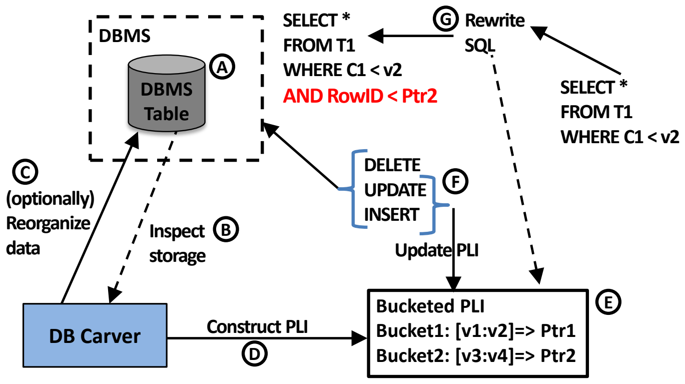


Fig. 7: Architecture of PLI.

full reconstruction of disk image. If the table is sufficiently (approximately) organized in the desired fashion and can be represented as a sequence of bucket ranges (e.g., first 10 pages contain function values $[0 - 10]$, next 10 pages contain function values $[9 - 12]$, etc.), then PLI can be built immediately; otherwise, we need to reorganize the table. If the table is not already sorted as we prefer, we impose the ordering by recreating that table structure. In either case we discard the existing secondary index (as PLI will replace it). Database user can choose arbitrary ordering that need not be unique or strict; any function or rule supported by ORDER BY clause would be acceptable. To order table T on function of columns (A-C), we create a new structure as:

```
CREATE TABLE T_PLI AS
SELECT * FROM T
ORDER BY (A-C).
```

This new table structure replaces the original table and requires very little maintenance from the host DBMS (since new rows can be appended at the end of the table). Note that any sorting function supported by DBMS can be chosen (e.g., income-expenses or $\sqrt{income}$).

Once the sorted table is created, we use DBCarver to validate table's storage sorting at the physical level. The table is likely to be sorted (or at least mostly-sorted) as the ORDER BY clause specified as non-clustered tables are generally stored in order of insertion. However, although such sorting is not guaranteed – in practice, new table may be stored differently on disk (most notably in Oracle). Using the underlying sorting, we next generate a bucket mapping structure, recording RowID boundaries for each bucket and creating the PLI structure.

4.5 PLI Validation Through Initial Experimentation

Experiments were performed on PostgreSQL 9.6 and Oracle 12c DBMSes. The limited availability of the database-internal RowID pseudo-column prevented us from using other DBMSes. Also, as previously mentioned, newer versions of PostgreSQL (i.e., 10 and on) do not allow the CTID pseudo-column to be indexed, making them inapplicable. We used data from the Unified New York City Taxi Data Set [24]. The experiments reported here were performed on servers with an Intel X3470 2.93 GHz processor and 8GB of RAM running Windows Server 2008 R2 Enterprise SP1 or CentOS 6.5.

4.5.1 Experiment 1: Regular Clustering

The objective of this experiment is to compare the performance of a table with a native clustered index and a table with a *PLI*. In Part-A, we collected query runtimes using a predicate on the sorted attribute. In Part-B, we compare the time to batch insert data into each table. In Part-C, we repeat the queries from Part-A.

Part A We began with 16M rows (2.5GB) from the `Green_Trips` table sorted by the `Trip_Distance` column. For each DBMS, we created one table that implemented the native clustering technique and another table that implemented *PLI*. Since an Oracle IOT can only be organized by the primary key, we prepended the `Trip_Distance` column to the original primary key. We then ran three queries, which performed sequential range scans, with selectivities of 0.10, 0.20, and 0.30. Table 1 summarizes the runtimes, which are normalized with respect to a full table scan (i.e., 100% is the cost of scanning the table without using the index) and the number of I/O over different query selectivity. Note that the I/O access is mandated by the PLI buckets accessed by the query; therefore, the number of I/Os is proportional to the reported runtimes. Since our goal is to evaluate a generalized database approach, the absolute time of a table scan is irrelevant; we are concerned with the runtime improvement resulting from indexing. In PostgreSQL, both approaches exhibited comparable performance, a few percent slower than the optimal runtime (e.g., for 0.20 selectivity the optimal runtime would be 20% of the full table scan). *PLI* remained competitive with native PostgreSQL clustering – the slight edge in *PLI* performance is due to not having the overhead of accessing the secondary index structure. PostgreSQL has to read the index and the table, while *PLI* access only reads the table (*PLI* structure itself is negligible in size). In Oracle, *PLI* significantly outperformed the IOT for the range scans. The queries that used a *PLI* were about three times faster than those that used an IOT. Oracle performance is impacted by lower average page utilization (and unused space) in the nodes of the IOT B-Tree. In all cases, the I/O cost was similarly reduced through the use of *PLI*.

Part B Next, we bulk loaded 1.6 million additional rows (250MB or 10% of the table) into each `Green_Trips` from Part-A. In PostgreSQL, the records were loaded in 263 seconds for the table that implemented native clustering and 62 seconds for the table that implemented a *PLI*. Clustering is a one-time operation in PostgreSQL and ordering is not maintained as inserts are

		Runtimes			**#of I/O**		
		Query Selectivity			**Query Selectivity**		
DBMS	**Index Type**	**0.10**	**0.20**	**0.30**	**0.10**	**0.20**	**0.30**
PostgreSQL	Clustered	15%	26%	38%	2,412	4,181	6,111
	PLI	13%	25%	36%	2,091	4,021	5,790
Oracle	Clustered	31%	57%	86%	4,986	9,167	13,831
	PLI	12%	21%	32%	1,930	3,377	5,146

Table 1: Query runtimes as percent of a full table scan and the number of I/O (clustered on attribute vs *PLI*).

performed. Therefore, the observed overhead was primarily associated with the clustered index itself. A *PLI* does not have a significant maintenance cost due to its sparse and approximate nature. In Oracle, the records were loaded in 713 seconds for the IOT, and 390 seconds for the table that implemented a *PLI*. Since IOT used a B-Tree to order records, the observed high overhead was caused by maintenance of the B-Tree as new records were inserted. Note that the I/O costs are a lower-bound approximation based on query runtimes. The implementation of IOT in Oracle is database-specific and not publicly available. IOT may have incurred further I/O overheads.

		Runtimes			**#of I/O**		
		Query Selectivity			**Query Selectivity**		
DBMS	**Index Type**	**0.10**	**0.20**	**0.30**	**0.10**	**0.20**	**0.30**
PostgreSQL	Clustered	90%	115%	139%	15,922	20,344	24,590
	PLI	23%	33%	44%	4,069	5,838	7,784
Oracle	Clustered	123%	238%	347%	21,760	42,104	61,387
	PLI	20%	31%	40%	3,538	5,484	7,076

Table 2: Query runtimes as percent of a full table scan and the number of I/O (clustered on attribute vs *PLI* after bulk insert).

Part C To evaluate the maintenance approach for each index, we re-ran the queries from Part-A. Table 2 summarizes the resulting runtimes. For both DBMSes, the queries that used a *PLI* incurred a penalty of 10% or less compared to Part-A, which is consistent with Part-B inserting 10% worth of new rows. All newly inserted records were appended to the end of the table and were therefore incorporated into the overflow bucket (requiring minimal maintenance in the process and causing limited query performance deterioration). In PostgreSQL, the queries using the native clustered index slowed down by a factor of about 4 due to the interleaving seeks inefficiency discussed in Sect. 1. In Oracle, the queries using native clustering also slowed down by a factor of about 4, albeit for a different reason. While the IOT maintains logically sorted records within the leaf node pages, these leaf node pages are not necessarily ordered on disk during B-Tree re-organization resulting in an increased number of seeks for the queries.

4.5.2 Experiment 2: Expression Clustering

The objective of this experiment is to expand upon Experiment 1 by evaluating an expression-based (rather than attribute-based) index to demonstrate the extendability and flexibility of the *PLI* approach. In Part-A, we collected query runtimes using a predicate on the sorted attribute. In Part-B, we compare the time to batch insert data into each table. In Part-C, we re-run the same queries from Part-A.

Part A We began with 16M rows (2.5GB) from the `Green_Trips` table, and we sorted the table on $\frac{Tip\_Amount}{Trip\_Distance}$ function (i.e., tip-per-mile for each trip as our order-preserving function). For each DBMS, we created one table that implemented the native clustering technique and another table that implemented *PLI*. As Oracle does not support function-based indexes, we created a computed column, and prepended this computed column to the primary key so an IOT could be built. We then ran three queries, which performed sequential range scans with selectivities of 0.10, 0.20, and 0.30.

Table 3 summarizes the number of I/Os and the runtimes, with runtimes normalized with respect to a full table scan over different query selectivity values. These baseline performance results are very similar the result from Experiment 1: Part-A demonstrating that query access for the function based index does not impose a significant penalty for any of the approaches. The runtimes for the Oracle IOT were slightly higher, which we believe were caused by additional storage space used by the computed column.

		Runtimes			**#of I/O**		
		Query Selectivity			**Query Selectivity**		
DBMS	**Index Type**	**0.10**	**0.20**	**0.30**	**0.10**	**0.20**	**0.30**
PostgreSQL	Clustered	13%	25%	37%	2,091	4,021	5,951
	PLI	14%	24%	36%	2,252	3,860	5,790
Oracle	Clustered	30%	62%	100%	4,825	9,971	16,082
	PLI	11%	21%	32%	1,769	3,377	5,146

Table 3: Query runtimes as percent of a full table scan and the number of I/O (clustered on expression-based index vs *PLI*).

Part B Next, we bulk loaded 1.6 million additional rows (250MB or 10% of the table) into each `Green_Trips` from Part-A. For the Oracle IOT containing the computed column, we previously generated the value, and we stored it in the raw data file. In PostgreSQL, the records were loaded in 917 seconds for the table that implemented native clustering, and 70 seconds for the table that implemented a *PLI*. This demonstrates that a traditional expression-based index is far more expensive to maintain than a regular index, producing much higher overheads. *PLI* requires very minimal maintenance – same as in Experiment 1, without an expression-based clustering. The insert cost into the table itself is using append and is thus comparable for both. In Oracle, the records were loaded in 1527 seconds for the IOT, and 408 seconds for the table that implemented a *PLI*. This drastic overhead increase in the time to load the data (compared to Experiment 1: Part-B) can be explained by data distributed. The data in Experiment 1 was more uniform requiring less B-Tree rebuilding, while computed ordering was much more scattered resulting in more B-Tree restructuring.

Part C To evaluate the maintenance penalties for each index, we re-ran the queries from Part-A as summarized in Table 4. Just as in Experiment 1, the queries that used *PLI* increased in cost by about 10% of a full table scan – as expected because inserted records were appended to the overflow bucket causing queries to scan additional 10% of overflow data. In PostgreSQL, the runtimes for the native expression-based clustered index increased by about a factor of 3 due to interleaving seeks as in Experiment 1. Interestingly, the penalty caused by computed index and storage fragmentation was not nearly as significant as regular built-in clustered index. We expect that PostgreSQL makes some additional effort to mitigate the overhead of interleaving seeks when utilizing an expression-based clustered index. In Oracle, the queries using the IOT increased by a factor of about 7, which is significantly more than Experiment 1: Part-C. This difference can be attributed to a greater amount of fragmentation caused by the B-Tree restructuring in Part-B.

		Runtimes			**#of I/O**		
		Query Selectivity			**Query Selectivity**		
DBMS	**Index Type**	**0.10**	**0.20**	**0.30**	**0.10**	**0.20**	**0.30**
PostgreSQL	Clustered	52%	79%	93%	9,199	13,976	16,452
	PLI	23%	32%	44%	4,069	5,661	7,784
Oracle	Clustered	259%	461%	706%	45,819	81,554	124,896
	PLI	20%	30%	40%	3,538	5,307	7,076

Table 4: Query runtimes as percent of a full table scan and the number of I/O (clustered on expression-based vs *PLI* after bulk insert).

5 PLI^+: An In-memory Physical Location Index for Delayed Clustering

5.1 Limitations of PLI

We demonstrated that PLI is competitive with a clustered index when performing read-only range scan queries on approximately sorted data. However, as inserted data is added to the overflow-page (or overflow bucket), PLI performance degrades (since the entire overflow bucket is always scanned). To address this limitation, we propose an extension of PLI called PLI^+. PLI^+ implements bucket-based reads similar to PLI, but additionally maintains performance for insert-intensive workloads. We describe PLI^+ in the remainder of this paper.

5.2 Overview of PLI^+

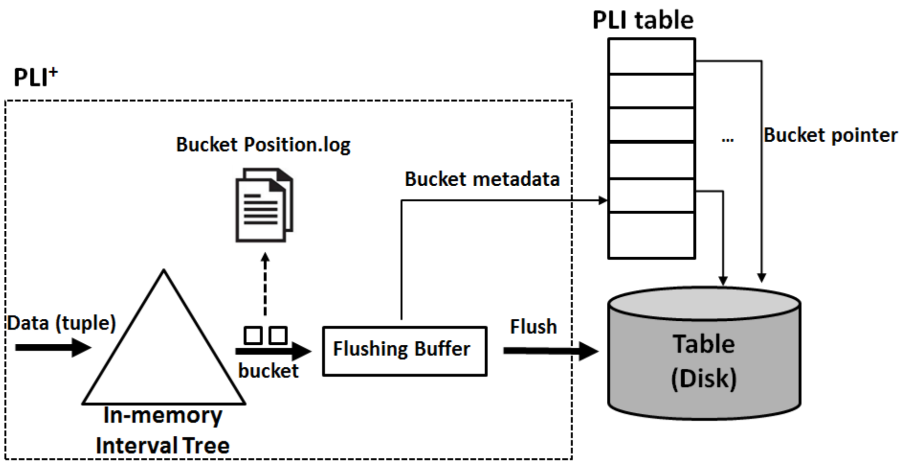


Fig. 8: The Architecture of PLI^+.

PLI^+ is a write-optimized external index for PLI that live databases can use to approximately sort the data in memory and reduce the number of I/Os. PLI^+ addresses two primary shortcomings of PLI. First, as the size of overflow bucket in PLI increases, there is an increased need to cluster, reducing the effectiveness of the index. PLI^+ further delays the need to cluster by approximate sorting of incoming data in memory. Current techniques to sort data either sort data locally in memory, *i.e.,* no global sorting of data requiring an expensive merge step later [19], or require large buffers to sort data in its entirety. PLI^+ approximately sorts tables by feeding incoming data into several intervals and maintaining a tree of intervals in memory. Data within

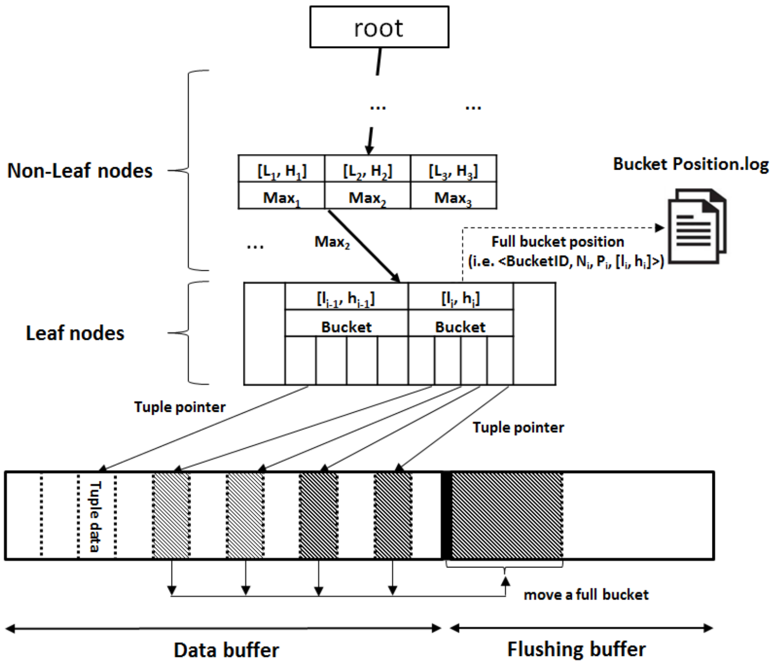


Fig. 9: The Internal Structure of PLI^+.

each interval is not sorted. In subsequent section, we describe the structure of this in-memory interval tree, how it is initialized, and maintained. Second, PLI^+ improves on PLI insertion granularity; PLI inserts new data at a page level (typically at 32KB or 64 KB). PLI^+ reduces the number of I/Os required for clustering by favoring large granularity of I/O (i.e., page size is larger than 512KB) [17].

5.3 PLI^+ Structure, Initialization and Querying

Figure 8 shows how PLI^+ operates along with PLI table. PLI^+ consists of three components, the in-memory interval tree, the buckets and their positions in a log file, and the flushing buffer. Figure 9 describes how the three components relate to each other. First, a path of the in-memory interval tree[3] is shown from root to leaf nodes. Leaves consist of multiple data buckets, with each bucket identified by an interval $[l_i, h_i]$. Note that the intervals of all leaves are disjoint. Also, in order for the ranges of in-memory tree to cover all input keys, we set the intervals of leftmost and rightmost leaf nodes with very large and very small numbers representing infinity (i.e., $((-\infty, h_0]$ and $[l_n, +\infty))$. Buckets consist of pointers to data tuples stored in a data buffer. Non-leaf

[3] This is a skeleton tree. At the initial stage, all tree nodes are initialized with meta data (e.g., intervals, pointers, etc.) but nodes initially do not contain any data.

nodes simply record the $Interval[Low_i, High_i]$ over all buckets of the subtree pointed by it. Thus technically, all of the data is stored in the leaves, while intermediate nodes are only used to store broader interval ranges and guide the search operation to narrower intervals in the leaves.

As a new data tuple (each tuple has a key) arrives, in-memory tree will be traversed using the tuple's key to find out the appropriate leaf node to append this new tuple. When a bucket is full, its data (i.e., tuples) will be immediately moved from the data buffer to a *flushing buffer*, its metadata is indexed in PLI table and the interval that previously contained this bucket is cleared. This means that there are no full leaf nodes in the in-memory tree. In other words, it is always possible to find an appropriate leaf node to insert a new tuple.

Flushing buffer contains full buckets which are ready to be flushed onto the database disk. We distinguish between data buffer and flushing buffer as PLI^+ favors large granularity I/O and aggregates data in flushing buffer before writing it out to disk. For example, in our most optimal setting a typical bucket size is around 256KB, and a flushing buffer is of size 20MB, so one disk I/O is performed when approximately 80 buckets are full. PLI^+ also records which interval ranges within leaves generate more full buckets. These are logged in a *bucket-position* data structure. This information will be used for maintaining the structure of PLI^+ so as to utilize maximum available memory for organizing the data in intervals (see Sect. 5.4). In other words, more memory must be allocated for intervals that have more data distributed within them. Currently, we initialize the intervals of the in-memory tree from an interval B-Tree [6], which provides historical initialization. Alternatively, uniform or Gaussian distribution can also be used. The next section describes how these distributions can be used to determine initial intervals.

5.3.1 In-memory Interval Tree Initialization

We support three methods to initialize the in-memory interval tree's structure: (i) historical initialization, (ii) uniform distribution and (iii) Gaussian distribution. We will briefly describe these methods and their assumptions; all notations used in this section are described in Table 5. Historic initialization was used for experiments.

Historical initialization: This method relies on the assumption that the data distribution in the injection flow follows a consistent pattern. Therefore, the structure of in-memory interval tree should adapt to the history of data distribution. Based on the knowledge of data distribution log file we build an interval B-Tree [6] and then copy its structure (i.e., all node metadata and its intervals) to in-memory interval tree. At the beginning stage (when there is no history of data distribution), we consider data to be using uniform distribution that will be discussed in the next part.

Table 5: Notations used in the paper.

Notation	**Description**
N_{Node}	Number of nodes
N_B	Maximum number of buckets in Buffer
N_I	Maximum number of buckets in a leaf node
B_T	Number of buckets in database
C_{Size}	Buffer cache size
B_{Size}	Bucket size
$[X_L; X_H]$	Range value of indexed key
$f^u(x)$	uniform distribution function
$f^G(x)$	Gaussian distribution function
θ	Buffer cache usage factor
μ	The mean value
δ	The standard division
$B(N_i, P_j)$	Number of generating buckets at node N_i, position P_j during time window W^T
$\overline{M}$	Average number of generating buckets
B_h	Upper-bound number of generating buckets
B_l	Lower-bound number of generating buckets

Uniform distribution: The uniform distribution [31] is defined by Formula 1. The uniform distribution in indexed keys of the insertion leads to equal interval division for the in-memory interval tree. Hence, we initialize the structure of the in-memory interval tree with uniform intervals. Particularly, the total number of nodes (N_{Node}) and maximum number of buckets in buffer (N_B) of the in-memory interval tree are determined by Formula 2. Interval information of each bucket is defined by Formula 3.

$$f^u(x) = \begin{cases} \frac{1}{b-a} & for \;\; b \geq x \geq a \\ 0 & for \;\; x > b \;\; or \;\; x < a \end{cases} \tag{1}$$

$$N_B = \theta \frac{C_{Size}}{B_{Size}} \qquad and \qquad N_{Node} = \frac{N_B}{N_I} = \theta \frac{C_{Size}}{B_{Size}.N_I} \tag{2}$$

$$Interval_i = (a_i, b_i) \;, \quad \begin{cases} a_i = a + i.\frac{b-a}{N_B} \\ b_i = a_i + \frac{b-a}{N_B} \end{cases} \tag{3}$$

Gaussian distribution: Gaussian distribution or normal distribution [31] is defined by Formula 4. The division of intervals is made following Gaussian distribution as defined in 5, where N_I is determined by Formula 2. The main idea is to have small interval in the high rate distribution range and vice versa.

$$f^G(x) = \frac{1}{\sqrt{2\pi\delta^2}} e^{-\frac{(x-\mu)^2}{2\delta^2}} \tag{4}$$

Where μ is the mean value and δ is the standard deviation.

$$Interval_i = (a_i, b_i) \;, \quad \begin{cases} a_i = X_L + i.\frac{1}{f^G(x).N_B} \\ b_i = a_i + \frac{1}{f^G(x).N_B} \end{cases} \tag{5}$$

Even though the in-memory tree structure (described in the next part) will dynamically adjust to the current status of input data distribution, it is important to select a suitable initialization technique at the beginning. First, in-memory tree will not immediately arrive to the optimal structure, instead gradually adapting – and a better in-memory tree may only be obtained after several maintenance calls. Second, all of the buckets generated before the adjustment takes place are not compact and if stored on disk, will degrade the scanning performance.

5.3.2 PLI^+ Querying

Algorithm 1 presents the search operation in PLI^+. It searches for all tuples in a database table where keys belong to a given interval. Query answer in PLI^+ is a combination of the results of two searches (Line 5): tuple scanning in the in-memory PLI^+ (Lines 6-17) and bucket scanning in the PLI index table (Lines 18-21). Searching in PLI^+ returns tuples that are part of the query result, while the results (in buckets) from PLI index table are set of on-disk buckets which need to be further filtered to eliminate irrelevant tuples (Line 4). To search over interval ranges, the search value is compared with the maximum high value over all interval ranges in the subtree rooted at a non-leaf node (Line 15).

Algorithm 1: Search for all tuples whose keys belong to a given interval.

```
1  Search(interval, Output):
2      SearchTree(root, interval, Tuples1);
3      SearchPLI(PLIRoot, interval, Bucket);
4      Tuples2 ⟵ Load and filter the list of buckets in Buckets
5      Output ⟵ Combine Tuples1 and Tuples2
6  SearchTree(node, interval, Output):
7      if (node.isLeaf) then
8          foreach (entry in node) do
9              if (entry.key is inside interval) then
10                 Output ⟵ Add entry to the result list;
11     else
12         for ($i = 0$ to node.length) do
13             if ($interval[1] < node.child[i].interval[0]$) then
14                 break;
15             if ($interval[0] \leq node.max[i]$) then
16                 /*Search in each child*/
17                 SearchTree(node.child[i], interval, Output);
18 SearchPLI(PLIroot, interval, Output):
19     foreach (bucket in PLIroot.Table) do
20         if (bucket.interval intersects with interval) then
21             Output ⟵ Add bucket to the result list;
```

5.4 Maintaining PLI^+

The intervals of the in-memory interval tree are created during the initialization stage. However, the distribution of indexed attribute values may change rapidly, requiring adjustment in the interval ranges in the in-memory interval tree. The in-memory interval tree resides on a restricted pool of data buffer. Each new incoming data tuple is appended to the end of this buffer. A bucket for an interval range is formed when a fixed number of data tuples (e.g., 1000 or 2000) are placed into the buffer. When the bucket is full, it is moved to a flushing buffer. Many interval ranges in an incoming data distribution do not receive a sufficient number of tuples to become full, and thus must be grouped or merged to form a full bucket, so that it can be moved to the flushing buffer. Similarly, if a specific interval range received a large number of tuples that form too many full buckets, this interval must be split because the large number of full buckets corresponds to a high density of data in this interval. As a result, the chosen interval range might be too broad for high selectivity queries to perform well, i.e., they may read more data than required.

To adjust the in-memory interval tree, currently we adopt a greedy heuristic. We note a single parameter, the number of times a given interval range becomes full to form a bucket, in the bucket position log data structure, and use this to merge or split bucket. If a given interval range in the leaf node has generated a number of buckets that is larger than the maximum value of a threshold, it is split, else if the number if greater than the minimum value of a threshold, it is merged with the neighboring bucket. Algorithm 2 describes the procedure. First, a data buffer to keep the new tree will be initialized (Line 3). Next, it begins with the left leaf-node of the tree (Line 2) and goes through all remaining leaf-nodes. At a position (i.e., an entry in a node), this process counts the history of generating buckets for this position in the tree (Line 7). If the large number of generating buckets (compared to the average number of buckets) is counted, then a split is applied on this position. Alternatively, if there is a small number of buckets generated at an entry, it should be merged with its sibling entries (Line 8). Finally, the new tree will be built on top of the chain of new leaves (Line 18).

It is to be emphasized the PLI^+ is a tree for delaying clustering in databases that requires clustering due to large number of bulk-inserts, and the user queries are predominantly read-only. This is the case of most scientific databases in which large number of inserts but few deletes and updates are present. Consequently, delete and update operations do not need to be optimized and PLI^+ treats them similarly to PLI.

6 Experiments

In Sect. 4.5, we have shown the effectiveness of PLI for a variety of queries and DBMSes. In this section, we evaluate the performance of PLI^+ as well as demonstrate its usability.

Algorithm 2: Reshape in-memory interval Tree's structure

```
1  Reshaping():
       Input  : The root of the tree, Bucket Position.log
       Output: The root of the new tree
2      $leafNode \longleftarrow$ Get the left leaf-node of the tree
3      $buffer[]$ $\longleftarrow$ initialize a data buffer for the new tree
4      while (leafNode != NULL) do
5          $buffer.cur \longleftarrow$ Load $leafNode$ to the first available position in buffer
6          foreach ($entry$ in $buffer.cur.data$) do
7              1. Count the number of generated bucket at this current position of
               $entry$ and $leafNode$ in the tree using data from $BucketPosition.log$
8              2. Determine which actions (i.e., Split, Merge or Keep) will be applied
               to $entry$ according to above number
9          $leafNode \longleftarrow leafNode.sibling$
10     /*Merge operations*/
11     foreach ($entry$ that requires a merge in $buffer$) do
12         1. Get a sibling entry having criteria: not null, small number of generated
           buckets.
13         2. Merge $entry$ with its sibling entry
14     /*Split operations*/
15     foreach ($entry$ that requires a split in $buffer$) do
16         split this entry into 2 sub-entries
17     Clear content of $root$ of the tree
18     $root \longleftarrow$ Rebuild the new root from the chain of leaf nodes in $buffer$
```

(a) NYC Datasets

Table	Records(M)	Size(GB)
NYC_T1	1.5	.23
NYC_T2	15	2.2
NYC_T3	30	4.4
NYC_T4	59	8.8
NYC_T5	148	22

(b) HEP Datasets

Table	Records(M)	Size(GB)
HEP_T1	28	3
HEP_T2	289	30
HEP_T3	587	61
HEP_T4	1200	122
HEP_T5	2900	305

Table 6: NYC and HEP Dataset Sizes.

Initial Setup Our experiments were performed against a PostgreSQL RDS cloud instance running Ubuntu 16.04 64-bit OS with an Intel Core i7-3770 3.4GHz CPU, 8GB of main memory, and a 1TB SATA HDD. We have selected two real-world data sets: New York City (NYC) Taxi dataset of year 2016 [25] and High Energy Physics (HEP) dataset [21]. We used various table sizes from each dataset, which are summarized in Tables 6(a) and 6(b). To perform

		Selectivity	
Range Query	**Key Range**	**NYC**	**HEP**
Q1	0.0025	0.028	0.031
Q2	0.005	0.035	0.059
Q3	0.010	0.067	0.104
Q4	0.020	0.111	0.175
Q5	0.050	0.191	0.268

Table 7: Query Range and Selectivity.

experiments for PLI^+, we initialized a table with 1GB of data clustered on the *trip_distance* column in NYC Taxi dataset and the P_t of $Muon$ column in HEP dataset, and bulk loaded the remaining raw data. A set of read-only (i.e., `SELECT`) queries that we used to measure performance of our indexes is summarized in Table 7. In Table 7, key range refers to delta difference in the value of the clustered attribute, i.e., range between X and $X+ < keyrange >$, where X refers to the value of the attribute. The selectivity refers to the ratio of the number of tuples in the query result to the total number of tuples in the database. Selectivity is larger than the key range due to higher density of the data overlapping the key range (e.g., in Q4, a 2% range query covers 11.1% and 17.5% in NYC and HEP datasets respectively).

Data Distribution To demonstrate the benefit of PLI^+, we examine the entropy of the indexed attribute, i.e., *trip_distance* column in NYC Taxi dataset and the P_t of $Muon$ column in HEP dataset as present in the downloaded dataset. The entropy [12] is computed based on Equation 6. In Equation 6, p_i is the frequency of occurrence of a given value of the indexed attribute over the total number of values in the domain range. Entropy, E, ranges from 0 to 1, with a random distribution at $E = 1$.

$$E = -\sum p_i * log_n(p_i) \tag{6}$$

To compute entropy, we constructed 20 1-million-tuple windows, with each window containing around one million of tuple insertions. Figure 10 shows the entropy values for NYC data remain around 0.71 for all 20 windows, and for HEP periodically ranges from 0.75-0.85. Nevertheless, entropy for both datasets is high, showing that the data values are mostly random.

Comparative Indexing Methods We selected three different indexing approaches that we believe provide a representative competition against PLI^+: secondary index, PLI [30], and LSM-Tree [19]. The secondary index served as a baseline comparison since it is the most commonly utilized indexing technique. We re-used our previous work to implement PLI, since PLI^+ is essentially an extension of this work. We applied 100MB of buffer for all evaluated candidates. We implemented LSM-Tree since this approach was designed for massive data

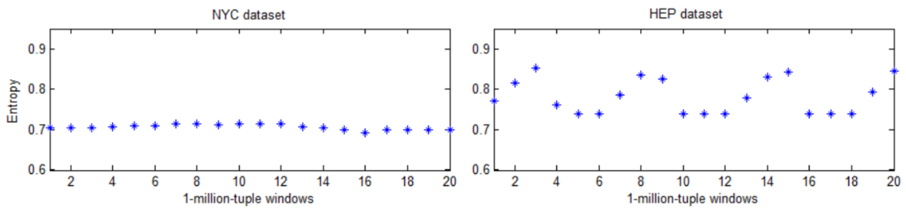


Fig. 10: The entropy of indexed key in NYC and HEP datasets.

(a) NYC Datasets

Table	Clustering (min)
NYC_T1	0.9
NYC_T2	6.8
NYC_T3	11
NYC_T4	34.5
NYC_T5	93.6

(b) HEP Datasets

Table	Clustering (hour)
HEP_T1	0.4
HEP_T2	1.5
HEP_T3	2.5
HEP_T4	8
HEP_T5	21.6

Table 8: Time to cluster NYC Taxi and HEP dataset tables.

ingestion. To ensure fairness of our comparison, we applied the same configuration to all indexing approaches.

It is important to emphasize that range scan query performance of a clustered index is always optimal, since the data in table is always physically ordered on disk. However, keeping a table clustered is unrealistic as the table size and the query throughput increases. The cost to maintain a clustered index is proportional to the table size as shown in Table 8. During the clustering maintenance operation time, the database may experience a downtime and the table may become inaccessible for querying.

6.1 Comparison of Query Execution Performance

We examine the impact on query performance for different indexing methods i.e., clustered index, unclustered index, table scan, PLI, PLI^+, and LSM-Tree. We furthermore considered PLI^+ at different bucket size settings (i.e., 250 tuples/bucket, 500 tuples/bucket, 1000 tuples/bucket and 2000 tuples/bucket). For this experiment, we indexed the NYC_T5 dataset, which is 22GB in size, and run our queries with different data selectivity varying from 0.01 to 0.57. Our primary result is shown in Fig. 11 (we excluded the extremely slow runtimes of unclustered index from the chart for better readability).

The cost of full table scan stands at a constant value of 350 seconds. We selected the cost of table scan as a baseline and calculated the relative performance compared to that option. Figure 11 shows the relative performance, normalized with respect to a full table scan (i.e., 100% is the cost of scanning the entire table without using indexing). At one extreme, unsurprisingly, is the unclustered index, which performs poorly over queries at all selectivities.

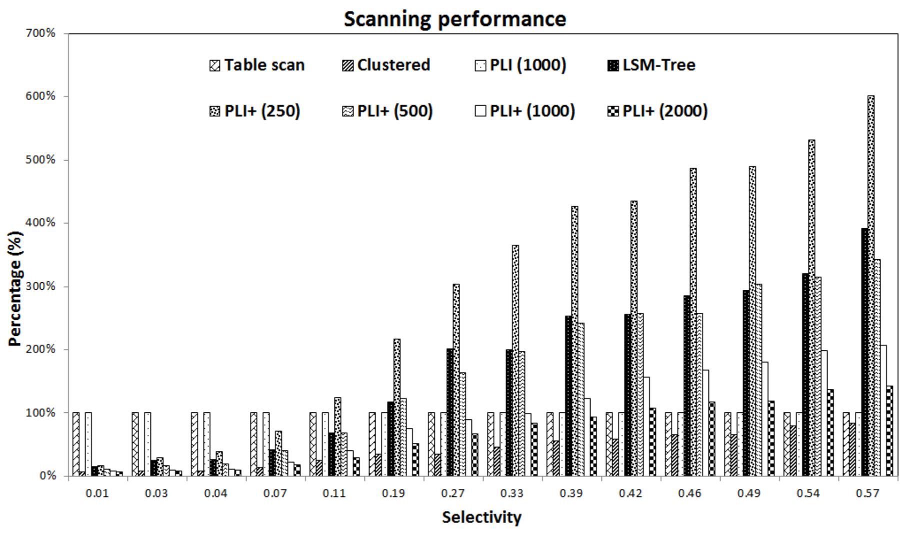


Fig. 11: Relative Performance of Queries at Varying Selectivity with Different Indexing Methods on Hard Disk.

With a high number of random I/Os in the unclustered index, execution time of querying even 0.01 of the table is slower than that of the table scan. The query execution time of unclustered index rapidly grows as query selectivity increases. For example, it is 10X slower than the table scan at query selectivity of 0.03. We therefore excluded the runtimes of unclustered index for better readability of the chart. On the other extreme is the clustered index that always outperforms other indexing methods at all values of query selectivity. This is expected, since table data is physically clustered on disk before querying. However, maintaining physical clustering data is impractical due to its high cost (see details in Sect. 3 and Table 8). The performance of PLI is dominated by the cost of overflow bucket scan. As discussed earlier, PLI is designed for read-oriented database and approximately sorted data. However, this experiment deals with large injection and randomly ordered data. Specifically, we started with 1GB of clustered data and inserted 21GB of unsorted data. Therefore, most of data in PLI is redirected to the overflow-page, degrading the scanning performance of PLI. Scanning overflow-page of PLI is done sequentially with large granularity I/Os, and thus the query execution cost becomes similar to table scan over query selectivities higher than 0.01.

The performance of queries using PLI^+ is significantly better than other candidates and tends to approach the performance of the clustered index (optimal performance) for high selectivity queries (0.01-0.15) over bucket sizes larger than 1000. The larger the size of bucket used in PLI^+, the closer its query execution times are able to track clustered index runtime, improving PLI^+ runtime for larger ranges of query selectivity. For example, PLI^+(2000) shows the best performance among the tested buckets, with better performance

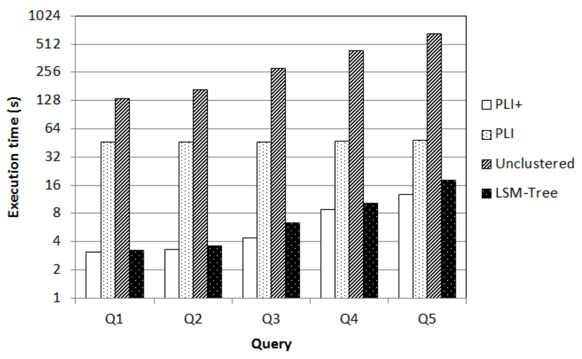


Fig. 12: Performance of Queries with Different Indexing Methods on SSD.

than a table scan for query selectivity as high as 0.4. It is to be emphasized that in large scientific databases full table scans or queries with low selectivity are infrequent; often scientific users are looking for needles in haystack [28].

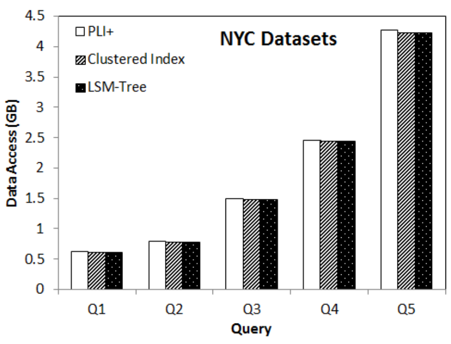
We can achieve better performance in PLI^+ by increasing the size of the bucket. However, having very large bucket size requires larger buffer for in-memory interval-tree to be loaded. Furthermore, there was no difference in the bucket compactness value ARB (see Sect. 6.3) for bucket sizes larger than 1000 tuples/bucket. The minimum amount of data accessed by queries with low selectivity is expected to be larger as bucket size is increased (since at least 1 full bucket must be scanned by all query), and thus a very large bucket size can degrade overall PLI^+ performance. In our later evaluation, to avoid using large buffer for in-memory interval tree, we fix the bucket size at 1000 tuples/bucket, as our compactness measure presented in Sect. 6.3 reaches the maximum value at this bucket size.

We also evaluated query runtimes on the NYC Taxi dataset (i.e., NYC_T5) for each indexing approach on SSD drives. Figure 12 summarizes these query runtimes. First, the performance of all methods is improved on SSD, since the throughput of SSD is much better than HDD. Also, because the cost of random I/O is similar to sequential I/O on SSD, we observed a significant improvement in query runtimes of both unclustered index and LSM-Tree indexes compared to HDD results. PLI^+ offers the smallest execution time in all types of queries on SSD due to large granularity of reading data, which is favored in SSDs; SSD performance decreases at page I/O which is performed in PLI and LSM.

6.2 Amount of Data Accessed

The objective of this experiment is to compare the the amount of data accessed by the different indexing approaches.

To compare the amount of data accessed by each indexing approach, we executed our set of queries (Table 7) against the NYC Taxi and the HEP datasets. We measured the data access for each index by collecting the read



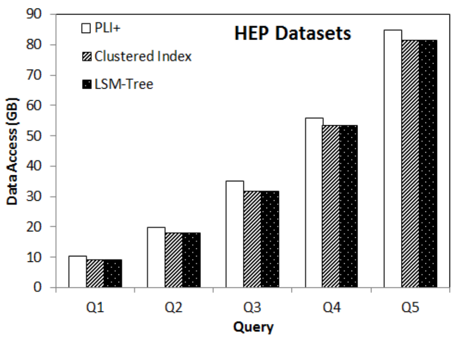


Fig. 13: Amount of data accessed by different indexing methods. We eliminated PLI due to very large amount of data accessed.

data based on our implementation of PLI^+ and LSM-Trees, and clustered index. Before we executed each query, we flushed both the DBMS and OS caches.

Figure 13 summarizes the results for the data access. As expected, the LSM-Trees and the clustered index accessed a similar amount of data for all queries from both the NYC and HEP datasets. This is because data in the LSM-Trees and the clustered index are completely clustered. Meanwhile, PLI^+ accessed a slightly higher amount of data compared to the LSM-Trees and clustered index. The explanation is that PLI^+ uses approximately sorted intervals thus reading some unnecessary data. Also, PLI^+ reads data in a bucket units rather than individual pages, thus an upper bound on the number of buckets covering matching data may be accessed instead of the exact number of pages. This performance in PLI^+ (in amount of data accessed) shows that the quality of buckets in PLI^+ is close to optimal intervals for the queries (i.e., high level of compactness, see Sect. 6.3 for the details) leading to fetching only the relevant data in PLI^+. Finally, PLI accessed significantly more data than the other indexing approaches in this experiment. We eliminated PLI from Fig. 13 due to its very large amount of data accessed. PLI incurred a significant penalty for all queries due to the inserted data being appended to the overflow bucket, and because the entire overflow bucket is always scanned.

6.3 The Quality of Buckets in PLI^+

PLI^+ inherits the idea of using bucket instead of tuple/row from PLI to favor the large granularity in I/O access. However, the exact amount of accessed data depends how distributed the data is in the buckets of PLI^+. In this experiment, we show that in fact PLI^+'s interval indexing generates compact buckets. To measure compactness, we define a metric that compares the range of minimum and maximum values in a bucket in for completely sorted data and the range of minimum and maximum values in same-sized PLI^+ bucket.

Bucket size (tuple/bucket)	**Raw input data**	PLI^+
250	0.00316	0.382
500	0.00529	0.519
1000	0.00869	0.9996
2000	0.01865	0.9999

Table 9: Compactness of Buckets in PLI^+.

For a given bucket size, we sum this over all buckets, as shown in Equation 7. The average relative bucket range factor (ARB) is:

$$ARB = \frac{\sum_{i=1}^{K} |Range(Bucket_i^{sorted})|}{\sum_{i=1}^{K} |Range(Bucket_i^{PLI^+})|} \tag{7}$$

in which K is the total number of buckets in a table, $|Range(Bucket_i^{optimal})|$ and $|Range(Bucket_i^{real})|$ are the range of the indexed values in a bucket of same size that is completely sorted and in PLI^+, respectively. $ARB \rightarrow 1$ (ARB approaches 1) means the buckets are highly compact, similar to perfectly sorted data; whereas $ARB \rightarrow 0$ (ARB approaches 0) means the values are distributed at random. Note that PLI^+ does not care if individual buckets are sorted internally, and therefore we do not consider the order of tuples in a bucket. Rather, we are concerned with how many values from other buckets have been injected into a given bucket due to sub-optimal merge and split operations in PLI^+.

The evaluation of the compactness of buckets in PLI^+ over different bucket sizes is presented in Table 9. We used the data from the NYC_T2 table. Our results show that compactness improves as bucket size increases—1000 and 2000 rows/bucket are closest to the ideal value of 1. While we are still working on proving the result formally, intuitively at smaller sizes even a deviation of a few values between buckets, makes the ARB value gravitate to 0. A larger-sized bucket has less potential of scattering its values in other buckets (e.g., a bucket set to the size of the entire table will always have an ARB value of 1). However, there are two penalties associated with larger bucket sizes. First, a larger bucket requires more memory to store in the in-memory interval tree; second, highly selective queries will experience additional I/O that is not needed to satisfy the query (e.g., if bucket size is equal to the size of the table, all queries will scan the entire table regardless of their selectivity).

6.4 The Size of the Constructed Index

The objective of this experiment is to evaluate the storage requirements for PLI^+. For this we compare the index sizes used by each indexing approach. To compare the index sizes, we collected the storage size of each index for our datasets when all data was inserted, i.e., around 22GB on NYC Taxi dataset and 305GB on HEP dataset. Figures 14(a) and 14(b) summarize the index sizes for each indexing approach in NYC Taxi and HEP datasets, respectively.

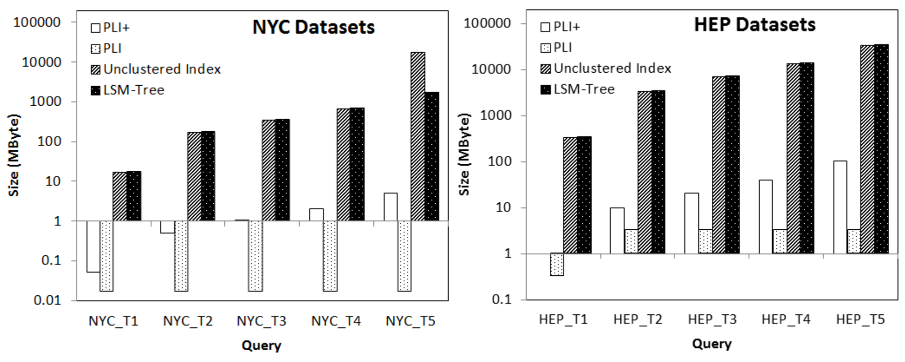


Fig. 14: Sizes of Constructed Index on NYC Taxi and HEP Datasets.

As shown in these figures, PLI has the smallest index size; whereas secondary index and LSM-Tree have large sizes in all datasets. While one may argue that with large memory sizes, the amount of memory that LSM consumes is small, a closer observation shows that in larger datasets (belonging to HEP), LSM consumes gigabytes of memory, which can be very expensive on a cloud instance. In PLI and PLI^+ data is grouped and indexed by buckets. Bucket interval is selected as indexed key. This means the size of index is reduced by the number of tuples in bucket. Compared to PLI, PLI^+ organizes the overflow page into buckets and indexes them, thus the number of buckets indexed in PLI^+ is larger than that in PLI.

7 Conclusion

We have presented PLI – a generalized clustered indexing approach that can be integrated into a live relational database using ROWID column. This indexing approach uses a bucket-based sparse indexing structure, which results in a very lightweight and easy-to-maintain index. The sparse pointers into the table can easily tolerate approximate clustering (i.e., reordering within the bucket is irrelevant) and trivially allows PLI variations to use an expression-based index to match query predicate. DBMSes could expose ROWID column further to make custom clustered index creation simple for the user – or this approach can be used to create a generation of better clustered indexes inside the database engine, as existing engines do not implement true (i.e., textbook-like) sparse clustering indexes. Moreover, we also propose an extension of PLI (PLI^+), a bucket-clustered indexing method for live relational databases that can be applied to both read-intensive workflows or huge injection workflows. This extension overcomes the overflow limitation in PLI by applying the approximation sorting with an efficient B-Tree-like structure to re-organize the data into buckets before storing and indexing them on secondary storage.

In the future, we plan to apply PLI^+ in the context of spatio-temporal data as well as using this technique in NoSQL context. We would also like to endow PLI^+ with an improved self-tuning algorithm that allows it to adapt to any kind of data input distribution.

Acknowledgments

The authors would like to acknowledge support for this work from the National Science Foundation under grants NSF ICER-1639759, ICER-1661918, ICER-1440327, and ICER-1343816.

References

1. Agrawal S, Narasayya V, Yang B (2004) Integrating vertical and horizontal partitioning into automated physical database design. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, ACM, New York, NY, USA, SIGMOD '04, pp 359–370, DOI 10.1145/1007568.1007609
2. Agrawal S, Chaudhuri S, Kollar L, Marathe A, Narasayya V, Syamala M (2005) Database tuning advisor for microsoft sql server 2005: Demo. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, ACM, New York, NY, USA, SIGMOD '05, pp 930–932, DOI 10.1145/1066157.1066292
3. Amazon (2017) Amazon EBS Product Details. `https://aws.amazon.com/ebs/details/`, [Online; Accessed 11-Dec-2017]
4. Amazon (2017) Amazon RDS for PostgreSQL Pricing. `https://aws.amazon.com/rds/postgresql/pricing/`, [Online; Accessed 11-Dec-2017]
5. Amazon (2017) Storage for Amazon RDS. `https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_Storage.html`, [Online; Accessed 11-Dec-2017]
6. Ang CH, Tan KP (1995) The Interval B-tree. Information Processing Letters 53(2):85–89
7. National Center for Biotechnology Information UNLoM (2017) Genbank. `https://www.ncbi.nlm.nih.gov/genbank/`
8. Catlett C, Malik T, Goldstein B, Giuffrida J, Shao Y, Panella A, Eder D, Zanten Ev, Mitchum R, Thaler S, Foster IT (2014) Plenario: An Open Data Discovery and Exploration Platform for Urban Science. IEEE Data Engineering Bulletin 37(4):27–42
9. Consortium GP, et al (2015) A global reference for human genetic variation. Nature 526(7571):68
10. Curino C, Jones E, Zhang Y, Madden S (2010) Schism: A workload-driven approach to database replication and partitioning. Proc VLDB Endow 3(1-2):48–57, DOI 10.14778/1920841.1920853
11. Garfinkel SL (2007) Carving contiguous and fragmented files with fast object validation. Digital Investigation 4:2–12
12. Gray RM (1990) Entropy and Information Theory. Springer-Verlag New York, Inc., New York, NY, USA

13. Jannen W, Yuan J, Zhan Y, Akshintala A, Esmet J, Jiao Y, Mittal A, Pandey P, Reddy P, Walsh L, Bender M, Farach-Colton M, Johnson R, Kuszmaul BC, Porter DE (2015) BetrFS: A Right-Optimized Write-Optimized File System. In: USENIX Conference on File and Storage Technologies (FAST)
14. Jindal A, Dittrich J (2012) Relax and let the database do the partitioning online. In: Castellanos M, Dayal U, Lehner W (eds) Enabling Real-Time Business Intelligence, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 65–80
15. Kersten ML, Manegold S, et al (2005) Cracking the database store. In: CIDR, vol 5, pp 4–7
16. Kimura H, Huo G, Rasin A, Madden S, Zdonik SB (2009) Correlation Maps: A Compressed Access Method for Exploiting Soft Functional Dependencies. Proceedings of the VLDB Endowment 2(1):1222–1233
17. Li Y, He B, Yang RJ, Luo Q, Yi K (2010) Tree Indexing on Solid State Drives. Proceedings of the VLDB Endowment 3(1-2)
18. NASA (2018) Nasa earth exchange. https://docs.opendata.aws/nasa-nex/readme.html
19. O'Neil P, Cheng E, Gawlick D, O'Neil E (1996) The Log-structured Merge-tree (LSM-tree). Acta Informatica 33(4)
20. Oracle (2018) Managing index-organized tables. https://docs.oracle.com/cd/B28359_01/server.111/b28310/tables012.htm#ADMIN01506
21. Pivarski J, Elmer P, Bockelman B, Zhang Z (2017) Fast Access to Columnar, Hierarchical Data via Code Transformation. ArXiv e-prints 1708.08319
22. Ramakrishnan R, Gehrke J (2003) Database Management Systems, 3rd edn. McGraw-Hill, Inc., New York, NY, USA
23. Richard III GG, Roussev V (2005) Scalpel: A Frugal, High Performance File Carver. In: Digital Forensic Research Workshop (DFRWS)
24. Schneider T (2016) Unified New York City Taxi and Uber Data. URL https://github.com/toddwschneider/nyc-taxi-data
25. Schneider TW (2016) Unified new york city taxi and uber data. https://github.com/toddwschneider/nyc-taxi-data, [Online; accessed 18-Aug-2017]
26. Sears R, Ramakrishnan R (2012) bLSM: A general purpose log structured merge tree. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp 217–228
27. Seshadri P, Swami A (1995) Generalized Partial Indexes. In: Proceedings of the Eleventh International Conference on Data Engineering, pp 420–427
28. Szalay AS, Gray J, Thakar AR, Kunszt PZ, Malik T, Raddick J, Stoughton C, vandenBerg J (2002) The SDSS Skyserver: Public Access to the Sloan Digital Sky Server Data. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp 570–581
29. Wagner J, Rasin A, Malik T, Heart K, Jehle H, Grier J (2017) Database forensic analysis with DBCarver. In: Conference of Innovative Database Research, pp 84–90

30. Wagner J, Rasin A, That DHT, Malik T (2017) PLI: Augmenting Live Databases with Custom Clustered Indexes. In: Proceedings of the International Conference on Scientific and Statistical Database Management
31. Walck C (1996) Hand-book on Statistical Distributions for experimentalists
32. Wang X, Burns RC, Malik T (2009) LifeRaft: Data-driven, Batch Processing for the Exploration of Scientific Databases. In: Proceedings Biennial Conference on Innovative Data Systems Research(CIDR)
33. Wang X, Perlman E, Burns R, Malik T, Budavári T, Meneveau C, Szalay A (2010) Jaws: Job-aware workload scheduling for the exploration of turbulence simulations. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society, pp 1–11