PLI: Augmenting Live Databases with Custom Clustered Indexes

James Wagner, Alexander Rasin, Dai Hai Ton That, Tanu Malik
School of Computing
DePaul University
Chicago, IL
{jwagne32,arasin,dtonthat,tanu}@cdm.depaul.edu

ABSTRACT

RDBMSes only support one clustered index per database table that can speed up query processing. Database applications, that continually ingest large amounts of data, perceive slow query response times to long downtimes, as the clustered index ordering must be strictly maintained. In this paper, we show that application slowdown or downtime, however, can often be avoided if database systems expose the physical location of attributes that are completely or approximately clustered.

Towards this, we propose PLI, a physical location index, constructed by determining the physical ordering of an attribute and creating approximately sorted buckets that map physical ordering with attribute values in a live database. To use a PLI incoming SQL queries are simply rewritten with physical ordering information for that particular database. Experiments show queries with the PLI index significantly outperform queries using native unclustered (secondary) indexes, while the index itself requires a much lower maintenance overheads when compared to native clustered indexes.

CCS CONCEPTS

•**Information systems → Record and block layout; Data scans;** •**Computer systems organization** → *Secondary storage organization;*

KEYWORDS

Clustered index, Custom index, Block layout

ACM Reference format:
James Wagner, Alexander Rasin, Dai Hai Ton That, Tanu Malik. 2017. PLI: Augmenting Live Databases with Custom Clustered Indexes. In *Proceedings of ACM Scientific and Statistical Database Management (SSDBM) Conference, Chicago, Illinois USA, June 27-29, 2017 (SSDBM '17),* 6 pages.
DOI: 10.1145/3085504.3085582

1 INTRODUCTION

Indexing is a primary technique in Relational Database Management Systems (RDBMS) to logically order data. Therefore, it is a key factor for scalable query processing. When the underlying data is clustered in index order, query processing scales up. However, in the absence of clustering a regular secondary index merely improves search performance while potentially incurring random I/O for each page access. In the worst case for an indexed query predicate on an unclustered index, a cost-based optimizer may resort to full sequential table scan for query selectivity as low as 0.01 (1%).

It is well recognized that moving object or sensor data warehouses that continually ingest data, which requires clustered and unclustered indexes to support analytical workloads, face query response time degradation due to indices becoming severely unclustered, i.e., incurring random I/O on each disk access. It is not uncommon for a database warehouse to undertake a downtime to recluster the entire data and improve performance.

However, such slowdowns due to random disk I/O can be reduced if the database shares some information of the physical location of attributes on a disk. We illustrate this reduction through an example in Figure 1. Consider Table T in Figure 1 with attributes {`ID, Name`}. The table is 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 maintenance costs, the clustering on {`ID`} in Figure 1 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 (Row19) will require a seek and a read of a seventh page at the end. Finally, fifth and sixth match will correspond to pointers (Row4, Row5) 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.

The only way to take advantage of this seek reduction is by determining the level of physical co-clustering within attribute {`ID`}, an 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 difference can be much larger. The sparse index on the right of table T illustrates that fewer seeks are possible with physical clustering.

To create an index based on physical location ordering of data rows, one must precisely determine the physical location, i.e., the values of the `RowID` column. While this internal column maps the queryable tuple to the physical location on the disk, commercial

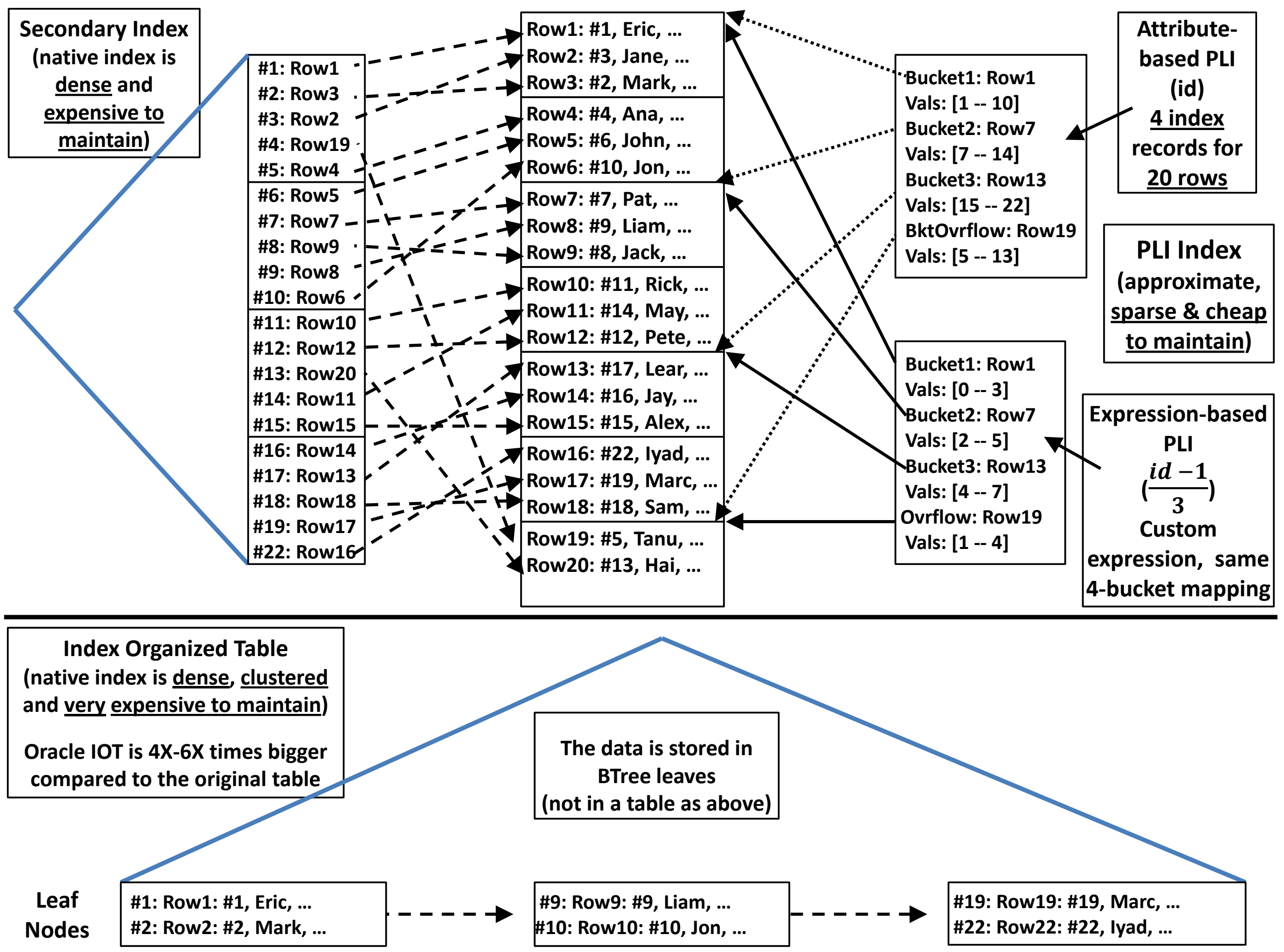

Figure 1: Storage layout of PLI and native database indexes (secondary and index-organized).

databases, due to physical-logical independence do not provide this information transparently. In fact, while the physical location of a row can be determined through a SQL query (e.g., `SELECT ROWID` in Oracle) , given two physical locations, there is no SQL query that determines if the two physical locations are strictly ordered on disk. Ordering a table on the internal `RowID` attribute, and inserting the result into a new table will also not guarantee that the resulting table is strictly ordered. To obtain the precise physical locations of the table rows in this paper we use a forensic tool that can read most commercial database storage files and output rows as they are physically ordered on disk for each table.

We describe a bucket based mechanism that exploits the approximate sorting inherent to Table T in Figure 1. Instead of traditional index that maps one value to one storage location (e.g., {`ID`}=3 to Row2 pointer) `PLI` utilizes range-of-values to range-of-addresses mapping. In our example, there is no entry for {`ID`}=3 specifically; instead, there is Bucket1 that represents a value range [1–10] (min and max for {`ID`}) which is mapped to a storage location range (in our example, Bucket1 corresponds to a physical pointer range of [Row1–Row6]). What Bucket1 tells us is that physical address range [Row1–Row6], which corresponds to first two pages, contains only values between 1 and 10. The index enables us to include (or exclude) this particular bucket without knowing the exact set of values or their specific ordering within the bucket.

Bucket mapped index has significant advantages over a typical secondary index. First, it provides the advantages inherent to a sparse index. That is, it requires one record per bucket (instead of one record per row) and is easier to maintain and use for lookup. Furthermore, bucket ranges defined in terms of `RowID` can be externally used in any database that exposes `RowID` values. This approach can be effectively generalized to multiple databases and attached to a live DBMS. Second, the value range associated with each bucket makes it easy to build expression-based variations of the same structure. For example, in Figure 1 we have a second `PLI` structure constructed on $\frac{ID-1}{3}$ rather than *ID*. The only change that such structure requires is re-mapping the value ranges (e.g., Bucket1 [1–10] becomes Bucket1 [0–3]) and the new `PLI` can be used on matching expression. This is far easier for order-preserving functions and we plan to explore other mappings and effects of inter-column correlation.

Our experiments show that a live database can be augmented with `PLI` using existing `RowID` and achieving query performance competitive to that of a native clustered index (or even exceed native performance because clustered indexes are not implemented to act as a true sparse index). Furthermore, PLI is also associated with surprisingly low overhead and higher tolerance to storage fragmentation (due to inserts) because of its sparse nature and approximate (rather than strict) ordering.

2 RELATED WORK

Some DBMSes (e.g., Oracle and MySQL) implement an *Index Organized Table* (IOT) as a replacement for clustered table. Figure 1 includes an example of IOT compared to a clustered index or PLI. While a traditional clustered index is still an additional structure that happens to be aligned with the sorting order of the table (table and index are two distinct structures), IOT is a merged structure with rows of the table spliced into the leaf nodes of the index itself. IOTs do achieve clustering (textbook definition) in that the table data is now kept sorted as new rows are inserted. However, this solution comes at a price. The leaves of the BTree data structure are logically sorted forming a linked list (each leaf node points to the next sibling). However, such linked list is not guaranteed to maintain a physical ordering as a clustered table usually does. Furthermore, even if a physical ordering of index leaves exists initially, BTree maintenance algorithm cannot maintain such continuity as the tree splits and merges (nor is this the goal of BTree structure).

Kimura et al. proposed dividing a table into buckets as a scan unit with correlation maps (CMs) index [3]. Using buckets to scan a table allows for a compressed index structure, but can result in false positives. A compressed index structure can be cached, reducing I/O operations for index maintenance just like PLI. Similar to CMs, our method records the ranges of values stored for each bucket. Unlike CMs, our method only requires access to internal row identifier – while CMs require a built-in clustered index to operate. As we show in our experiments, built-in clustered index has some practical (database-specific) limitations.

Generalized partial indexing builds unclustered indexes around records defined by the user, leaving some records not indexed [5]. A physical location index is similar in that the user defines which sections, i.e. buckets, of the table to reorder possibly leaving some buckets unordered. However, PLI provide the benefit of indexing to most records, and approximately sorts data across buckets. In both methods, index maintenance cost is reduced by only recording access or reorganizing data that benefits queries.

Database cracking expands on generalized partial indexing by reorganizing a cracker index in pieces accessed by queries [2]. Our work allows the user to reorganize data across units of buckets, where the size of the bucket is determined by the user instead of a query. Similar to database cracking, data is only organized across, not within, a piece or bucket. The major difference between the two is that database cracking requires significant rewrite of the DBMS engine, while we add PLI to a live database.

Cheng et al. implements predicate introduction to improve query performance [1]. Predicate introduction can be used to improve a query by accessing a column with an index, or by reducing the tuples scanned for a join. Instead of rewriting queries based on structures created by the user, our work rewrites queries with constraints on the database internal row identifier in the WHERE clause of the query. Since the row identifier is typically used to access rows when a full table scan is not used, there is no benefit to using user created structures.

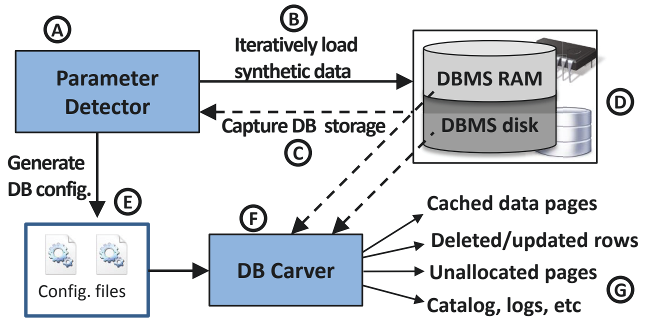

Figure 2: Architecture of DBCarver.

2.1 DBCarver

In this section, we describe the general implementation of database page carving with DBCarver [6]. Figure 2 shows the overall architecture of DBCarver. DBCarver consists of two main components: the parameter detector(A) and the carver(F).

The parameter detector calibrates DBCarver for the identification and reconstruction of different 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.

3 HOW TO BUILD A CUSTOM CLUSTERED INDEX

To create a database-independent clustered in a DBMS, we augment the DBMS with a module that consists of a sparse index structure, a maintenance component and an automated SQL query re-writer. Section 4 presents results using a PostgreSQL and Oracle DBMSes.

3.1 Architecture

The architecture of PLI operation is shown in Figure 3. We rely on the native database table(A) with no modifications or assumptions about DBMS engine features (e.g., DBMS may not even support clustering). Initially, we use DBCarver to inspect table layout as it currently exists. As shown in [6], looking for specific pages in a table is orders of magnitude faster compared to full reconstruction of disk image. If the table is sufficiently (approximately) organized in the desired fashion and can be represented as a sequence of

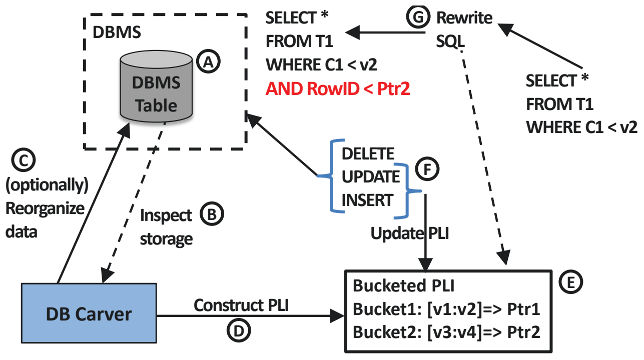

Figure 3: Architecture of PLI.

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 (by creating a replacement table with custom `ORDER BY` clause). Note that any sorting function supported by DBMS can be chosen (e.g., `income-expenses` or $\sqrt{income}$). The `PLI` structure is then constructed(D) as a substitute for the native DBMS index – Figure 1 outlines different index choices. `PLI` is orders of magnitude cheaper to maintain when compared to a regular index for two reasons: 1) `PLI` is *sparse* and thus very small, and 2) `PLI` does not need to track `DELETE` operations (we explain why in this section). Finally, in order to query with `PLI`, we use a simple query rewrite process(G) that occurs outside of the database. An incoming query is augmented to include a predicate on database-specific implementation of the RowID to instruct the DBMS engine which pages to read. The rest of this section discusses creation and use of `PLI` in more detail.

3.2 Initial Setup

The first step in using `PLI` is to inspect the table and organize it (if necessary) according to the desired access pattern. Note that reorganization refers the *table data itself* not to creation of an additional index. Secondary index does not permit sequential access and introduces significant overhead in addition to the original table. Databases clustering functionality is severely restricted in practice (e.g., in many DBMSes clustering index key must be unique). Despite the fact that sparse access is the distinguishing feature of clustering indexes, they are never truly sparse (e.g., using only 1 index entry per 80 rows on a page) when used in practice.

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).

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.

3.3 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 1, `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 1, 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 insert 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 1 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 it 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. The degradation is gradual, but eventually the table will need to be reorganized. The comparison of different reorganization algorithms is beyond the scope of this paper, but 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. A structure of this size can be kept in RAM and used or maintained at a negligible overhead cost.

3.4 Query rewrite

In order to use PLI index, incoming SQL queries are rewritten to take full advantage of the current layout of the table. Additional PLI-based predicates are added to the query to restrict the disk scan range; bucket-based indexing is approximate by nature and provides a superset range within which data of interested resides. For example, consider Figure 1 – a 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)`. In Oracle ROWID will be used instead of CTID in PostgreSQL. The first added condition matches the range of buckets (in that case the first bucket from PLI) and the second condition corresponds to the overflow bucket. This access range results in a more efficient pattern of disk reading by minimizing seeks and by removing the overhead of secondary index use. While this PLI condition does include false-positives (specifically, id #10 at Row6 and #13 at Row20), the original query predicate (`id BETWEEN #1 AND #6`) will elminate false positives.

4 EXPERIMENTS

Due to limitations of available user access to the database-internal RowID attribute, our experiments were limited to two databases, PostgreSQL and Oracle. We used data from the Unified New York City Taxi Data Set [4]. 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.

4.1 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). 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.

		Query Selectivity		
DBMS	**Index Type**	**0.10**	**0.20**	**0.30**
PostgreSQL	Clustered	15%	26%	38%
	PLI	13%	25%	36%
Oracle	Clustered	31%	57%	86%
	PLI	12%	21%	32%

Table 1: Query runtimes as percent of a full table scan (clustered on attribute vs PLI).

Part B. Next, we bulk loaded 1.6m 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 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.

		Query Selectivity		
DBMS	**Index Type**	**0.10**	**0.20**	**0.30**
PostgreSQL	Clustered	90%	115%	139%
	PLI	23%	31%	44%
Oracle	Clustered	123%	238%	347%
	PLI	20%	31%	40%

Table 2: Query runtimes as percent of a full table scan (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 Section 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.

4.2 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., tipper-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 runtimes, which are again normalized with respect to full table scan. 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.

		Query Selectivity		
DBMS	**Index Type**	**0.10**	**0.20**	**0.30**
PostgreSQL	Clustered	13%	25%	37%
	PLI	15%	25%	37%
Oracle	Clustered	30%	62%	100%
	PLI	11%	21%	32%

Table 3: Query runtimes as percent of a full table scan (clustered on expression-based index vs PLI).

Part B. Next, we bulk loaded 1.6m 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.

		Query Selectivity		
DBMS	**Index Type**	**0.10**	**0.20**	**0.30**
PostgreSQL	Clustered	52%	79%	93%
	PLI	22%	31%	44%
Oracle	Clustered	259%	461%	706%
	PLI	19%	30%	40%

Table 4: Query runtimes as percent of a full table scan (clustered on expression-based vs PLI after bulk insert).

5 CONCLUSION

We have presented PLI – a generalized clustered indexing approach that can be added to 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.

ACKNOWLEDGMENTS

This work was partially funded by the US National Science Foundation Grant CNF-1656268.

REFERENCES

[1] Qi Cheng, Jarek Gryz, Fred Koo, TY Cliff Leung, Linqi Liu, Xiaoyan Qian, and Bernhard Schiefer. 1999. Implementation of two semantic query optimization techniques in DB2 universal database. In *VLDB*, Vol. 99. 687–698.

[2] Martin L Kersten, Stefan Manegold, and others. 2005. Cracking the database store. In *CIDR*, Vol. 5. 4–7.

[3] Hideaki Kimura, George Huo, Alexander Rasin, Samuel Madden, and Stanley B Zdonik. 2009. Correlation maps: a compressed access method for exploiting soft functional dependencies. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1222–1233.

[4] Todd Schneider. 2016. Unified New York City Taxi and Uber Data. (2016). https://github.com/toddwschneider/nyc-taxi-data

[5] Praveen Seshadri and Arun Swami. 1995. Generalized partial indexes. In *Data Engineering, 1995. Proceedings of the Eleventh International Conference on.* IEEE, 420–427.

[6] James Wagner, Alexander Rasin, Tanu Malik, Karen Heart, Hugo Jehle, and Jonathan Grier. 2017. Database Forensic Analysis with DBCarver. CIDR.