Fractured Indexes: Improved B-trees To Reduce Maintenance Cost And Fragmentation

Newton Ikhariale
newtonik@cs.brown.edu
Computer Science Department, Brown University, RI, USA.
Advised by Stanley B. Zdonik

January 7, 2010

Abstract

Database warehouses support processing very large amounts of data. The workload is primarily composed of SELECT queries; updates are less frequent and typically come in batches. And yet the INSERT operation can be expensive and have a significant impact on SELECT query performance. Inserted data has to be placed in its respective location based on presence of indexing structures; excepting the index based on the arrival time stamp, the updates are likely to manifest all over the data range. If the modification do not exhibit an access locality, each one will touch one or more different disk pages causing a high performance penalty (irrespective of any batching). The relevant page has to be read, modified and eventually written back to disk once the memory buffer fills up. In this paper we propose a *Fractured Index* mechanism to alleviate the penalties associated with the update costs while still being able to use B-Tree indexes. By creating a mini-Index for each batch of data, we ensure insert locality (all data within the mini-Index is freshly inserted) and keep the indexes read-only (achieving better fragmentation of these indexes and providing compression opportunities). The idea of batching is frequently used in databases; storing insert batches in the B-Tree structures is a logical extension of the same approach. Although earlier work has presented a similar delayed update approach, we believe that our idea is much simpler to implement. As we will show in this paper, this minimalistic implementation achieves two orders of magnitude improvement of INSERTs and one order of magnitude improvement of SELECT queries.

1 Introduction

The B-tree [BM72, Knu73] is one of the most widely used data structures to organize data in databases. B-tree provides a fast read access to data by sorting the data by indexed attribute (key) values and hierarchically organizing the data into intermediate and leaf nodes. B-tree splits and merges nodes upon insertions and deletions to keep the tree balanced.

Most database products provide two types of B-tree indexes, primary (*clustered*) and secondary (*non-clustered*). A primary B-tree index stores tuple data in its leaf nodes while a secondary B-tree index stores only pointers to the primary index (or heap if no primary index exists). A primary B-tree index provides a faster read access than a secondary B-tree index, especially for Online Analytical Processing (OLAP) [CD97] style queries because an access path using a secondary B-tree index needs a random disk seek for each tuple to be fetched while OLAP queries are often not selective. A primary B-tree index requires only one seek to locate the needed tuples and then performs a sequential scan. This is substantially more efficient because one random disk seek takes several milliseconds while a sequential disk access takes only microseconds to read a tuple.

Although reads on B-tree in its initial state are efficient, writes on B-tree create two problems. First, each write operation requires random disk seeks to access and overwrite data in B-tree. Second, splitting and merging nodes can cause severe fragmentation of B-tree structure on disk. This can significantly degrade the read performance on B-tree because each fragment requires a jump (seek) even for sequential read on a primary index. These problems become more and more significant as the random disk seek performance has not been increasing over a few decades while disk capacity and sequential disk access performance are rapidly increasing.

In this paper, we propose a new data structure, Fractured Index, which overcomes the problems of B-tree by using only sequential accesses on disk. Fractured Index is a few orders of magnitude faster than B-tree to maintain and also an order of magnitude faster to query after heavy write operations because Fractured Index causes no fragmentation on disk. It is especially beneficial for OLAP type queries which are less selective. In this paper, we mainly focus on the context of OLAP, but the same technique could be used to Online Transaction Processing (OLTP) setting to achieve the better maintenance performance with a comparatively more performance penalties on read queries.

The contributions of this paper are:

- An introduction of the architecture of Fractured Index that speeds up write operations on B-tree and eliminates fragmentation on disk
- A proposal on how to use Fractured Index for answering queries and how to reorganize it for speeding up queries after several fractures are made.
- A verification of the advantage of Fractured Index over the unmodified B-tree on maintenance and query performance experiments on MySQL

The remainder of this paper is organized as follows. Section 2 defines the architecture and use of Fractured Index. Section 3 demonstrates the experimental result to verify the advantage of Fractured Index. Section 4 lists related prior work and Section 5 concludes with future work.

2 Architecture

The key idea of Fractured Index is to sequentially output small B-trees to reduce the overhead of inserts into tables that are organized as primary and secondary B-trees.

Inserting new tuples to random places of the B-tree is a costly operation which cause random disk seeks and fragmentation due to splits and merges of B-tree nodes. DBMS uses a buffer pool to alleviate this problem by keeping dirty (modified) pages in memory and later writing them back to disk, but it is impossible for the buffer pool to contain all of the necessary leaf pages when millions of new tuples are inserted to the main table. The leaf pages are continuously being swapped between the disk and buffer pool as inserts continue to come in unless new tuples are coming in the order of index keys, which does not occur in general case. This results in a large number of disk seeks and fragmented B-trees.

The Fractured Index mitigates these problems by sequentially recording data changes (insertions and deletions) into small B-trees separated from the main table. In order to sequentially write small B-trees (*mini-index*), the Fractured Index buffers insertions and deletions into the database in *Insert Buffer* and dumps the data changes to a new mini-index when the current insert buffer becomes full. When reading the database, the query processor reads from each mini-index in addition to the main table.

Details of this architecture are described as follows. We explain how a *Fractured Index* is stored on disk in Section 2.1, how a Fractured Index buffers data changes and generates mini-indexes in Section 2.2, how the query processor accesses the mini-indexes in addition to the main table for answering queries in Section 2.3 and finally how a Fractured Index merges mini-indexes to improve the query performance when there are too many mini-indexes in Section 2.4.

2.1 Fractures on Disk

The mini-indexes are called *Fractures*. Each fracture resembles the main table in every aspect. It has the same primary and secondary B-trees. The only difference is in the set of tuples that it contains. Each fracture amounts to only 100MB or less of most recently inserted data so that the memory buffer can retain the latest on-going fracture in memory.

Figure 1 illustrates the difference between the traditional and fractured approach. The main and mini B-trees, except the latest one (insert buffer) are read-only. Future insertions and deletions do not change them. This disk layout avoids fragmentation of B-trees which would cause a severe query performance degradation in traditional B-trees.

2.2 Buffering Insertions and Deletions

As the Figure 2 shows, Fractured Index buffers both insertions and deletions to the database until the buffer becomes full. When the buffer becomes full, Fractured Index outputs the current set of buffered insertions and deletions to a new fracture (mini-index) which consists of the same set of secondary indexes and heap file as the base table. This dump operation is efficient because it does only sequential writes in contrast with traditional B-trees which cause random disk seeks for each tuple to be updated. Also, our approaches causes no fragmentation on disk as described in the previous section.

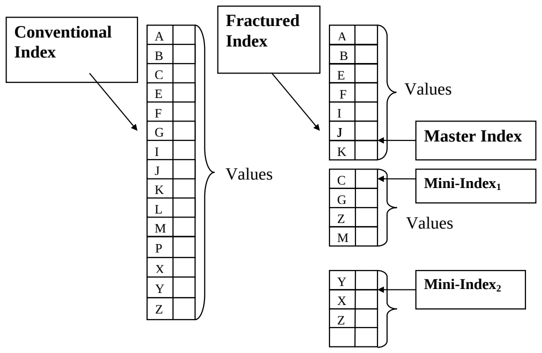

Figure 1: Main Indexes and Fractured Indexes

Deletions are handled similarly to insertions in Fractured Index. Each fracture has a *delete set* which is a set of tuples deleted during the period. Fractured Index efficiently dumps the delete sets when the buffer becomes full in the same way as insertions. The query executor reads the delete sets when the fracture is accessed. Updates are handled as composites of deletions and insertions.

2.3 Answering Queries

A query on the table is rewritten to a set of queries by the query optimizer to access the fractured indexes. The rewritten queries look up on each fracture and the main table, merging the partial results from each fracture. As described in previous section, each fracture has the same set of primary and secondary indexes. Each of the rewritten query uses the same access path, for example primary index seek or secondary index seek.

This approach does not add significant overhead because the query cost is generally determined by the total size of the table and the total number of retrieved tuples. Fractured Indexes have a smaller size in each fracture and also the same number of retrieved tuples in total. Therefore the only additional overhead of SELECT queries on Fractured Indexes comes from the execution of the query on each fracture, which includes query parsing, query optimization and launching cost on each fracture. This does not cause a significant overhead as far as the number of fractures is not tremendous especially in OLAP setting where one query accesses many more tuples than OLTP and almost all costs purely come from disk I/O.

However, when the number of fractures becomes large over time, the query performance could deteriorate. In that case, Fractured Indexes *Merge* fractures to reduce the number of fractures.

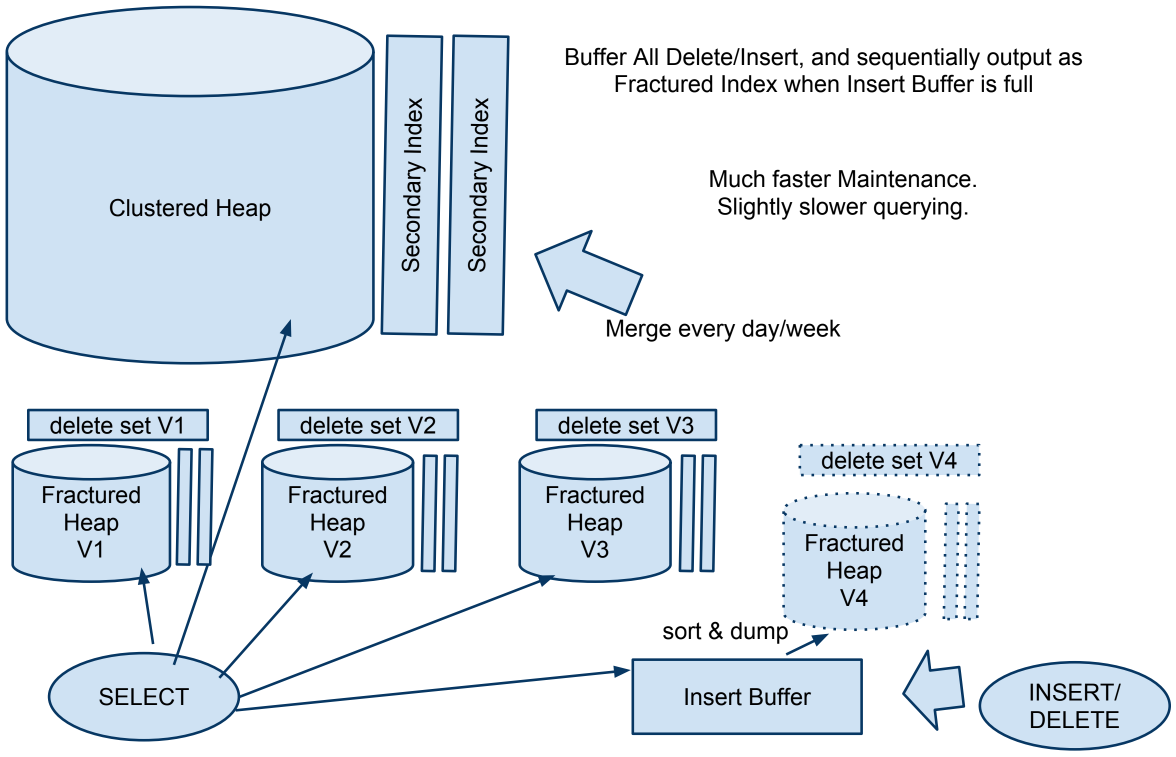

Figure 2: Buffering Insertions and Deletions

2.4 Merging

The merging process cannot apply inserts directly to the main table as the non-fractured approach does because it leads precisely to the same problems our new approach is attempting to mitigate. Batch inserts of the fractured table data into the main table will lead to the same insert overhead and further fragmentation of the main table.

The proposed method is to merge the tables by performing a sort-merge to combine all of the data to one table. The data from each individual fracture is pre-sorted on the clustered key, so Fractured Indexes can be efficiently merged. The sort-merge proceeds as follows. The merge operation must always compare the top (lowest ordered row) of each table to find the next row. This is important because the clustered key is likely random and the consecutive ordered rows will likely be in different tables. The size of the main table prevents it from being loaded to memory. Therefore, the data has to sorted on the database server and then retrieved using a reasonable fetch size. The sorting can be accomplished by using an **ORDER by *primarykey*** on the retrieval query.

The fetch size must be set to a reasonable size based on the available server memory.

The result from the merge is loaded back into the database sequentially because tuples are ordered by clustered key in the sort-merge step. The newly loaded data will provide efficient ordering of data on disk and fill factor in the B+ Trees. The secondary indexes are created after the inserts are finished. This process should restore the query performance to optimal levels.

3 Experimental Results

The experiments are designed to test the performance of the Fractured Index approach compared to existing methods. It is important to observe how both insertion and select queries behave in this new setting compared to the standard approaches. To evaluate the performance of the fractured index approach, we compared it against the performance of MySQL B-trees. One of the distinguishing features of MySQL from other relational databases is its support for multiple storage engines. They include MyISAM (default engine), Falcon, Federated, BerkeleyDB, InnoDB. Unless otherwise noted, we are using the InnoDB engine in our experiments. This is because the InnoDB database engine has a component called the Insert Buffer. The insert buffer is responsible for buffering inserts when the indexes that are being modified are not already present in the buffer pool. Inserts to secondary indexes are buffered to reduce I/O operations on disk. While there is no official documentation detailing how the Insert Buffer actually works, at the high level it works the following way: when an insert is executed, InnoDB first checks the buffer pool for the page that needs to be modified. If the relevant page is in the buffer pool already the insert will proceed as usual. However, if the page is not available, the database engine will insert the record into the insert buffer instead. Then, at certain time intervals the insert buffer is merged into the main file. The MySQL manual claims that *insert buffering* can speed up inserts by up to a factor of 15 [ABa].

3.1 Setup

The experiments are run on a Debian based Linux computer. The specifications of the computer and MySQL.

Hardware

- CPU: Core 2 Duo E6600 2.66 GHz
- Memory: 6GB, Swap 3GB
- Hard Drive: 320 GB SATA 7200rpm
- OS: Ubuntu 9.04 x64

MySQL InnoDB

- MySQL Community Server version 5.1.37

- InnoDB Plugin 1.0.4
- MySQL Buffer Pool Size 138MB
- Page Size 16KB

Relevant InnoDB System Variables

- Insert Buffer Size 65MB
- innodb_fast_shutdown 0
- innodb_flush_method O_DIRECT
- innodb_autoextend_increment 128MB

3.1.1 InnoDB Plugin

In order to validate the performance improvement achieved by InnoDB's insert buffering, we compared InnoDB's performance with and without the Insert Buffer. To setup up this test, *insert buffering* had to be controlled. There is no option to control Insert Buffer component in currently available MySQL distributions. However, an InnoDB plugin with the option to control insert buffering was available. InnnoBase, the developer of InnoDB, also releases the storage engine as a plugin. The plugin sometimes includes new and improved features compared to the version built into into MySQL server by default. It can be dynamically installed on compatible MySQL installations. In the InnoDB plugin version 1.0.4, the global variable *insert buffering* is used to turn off the insert buffering feature when necessary for our experiments. It was also necessary to ensure that the insert buffer and buffer pool were flushed between runs. To flush, the server was restarted with the *fast_shutdown* set as 0 rather than the default of 1. When set to 1 the InnoDB flushes the buffer pool and completes all merge operations in the insert buffer.

3.1.2 Inserted Data

The data used to insert were randomly generated strings (explained below) with a length of 5. Each inserted row had 20 columns. The schema of the database table is as follows.

The main table was updated to have ten secondary indexes (B+ Trees) on string columns. It is important to note that inserted rows have random primary key values. This was done to simulate more realistic scenarios, auto increment is not very common in data warehouses. It also prevents the buffering when inserts are done using auto increment on the clustered index.

```
CREATE TABLE book
(id BIGINT NOT NULL,       publishedyear INT          NOT NULL,
    Name    VARCHAR(255)    NOT    NULL,         col1         VARCHAR(255) NOT NULL,
    col2    VARCHAR(255)    NOT    NULL,         col3         VARCHAR(255) NOT NULL,
    col4    VARCHAR(255)    NOT    NULL,         col5         VARCHAR(255) NOT NULL,
    col6    VARCHAR(255)    NOT    NULL,         col7         VARCHAR(255) NOT NULL,
    col8    VARCHAR(255)    NOT    NULL,         col9         VARCHAR(255) NOT NULL,
    col10   VARCHAR(255)    NOT    NULL,         col11        VARCHAR(255) NOT NULL,
    col12   VARCHAR(255)    NOT    NULL,         col13        VARCHAR(255) NOT NULL,
    col14   VARCHAR(255)    NOT    NULL,         col15        VARCHAR(255) NOT NULL,
    col16   VARCHAR(255)    NOT    NULL,         col17        VARCHAR(255) NOT NULL,
    col18   VARCHAR(255)    NOT    NULL, PRIMARY KEY(id)) Engine=InnoDB;
```

Table 1: Base Index Table (Query)

Base Table The base table is generated by inserting 500 batches of 20K rows until it reaches 10 million rows. The data in each column was randomly generated, however, to control the selectivity of our queries, purely random words were not used. 10 million five character words were randomly generated and written to a file. The data in this file was then used to generate words for all the experiments. All 10 million words are loaded to memory at the beginning of a test script. An insert row can randomly choose a number within the allowed range (1 to 1 million) as a position at which we can get a random word. The purpose of this method is to be able to control the number of rows retrieved when a SELECT query is executed on the database. For example, given a column with smaller cardinality (smaller distinct word count), the following steps were taken. Load the list of strings to an array. The array can be defined as mockdata[100000]. For a column 1,

    String randword = mockdata[rand.nextInt(100)] ;

However for column 2, the following was used to get random data.

    String randword = mockdata[rand.nextInt(10000)] ;

Column 2 will have more distinct words that Column 1 because it has a larger set of words to choose random words. This helps when writing SELECT queries to test because generating a column because by varying the available set you can vary number of Results in a query. An sample SELECT query,

    SELECT * from book WHERE col = “word”

the number of rows returned depends on the size of the set used to generate the column. This assumes that “word” is in the available set.

The initial base table will be restored at the start of every experiment. This provides a consistent base for the tests to run. The process of restoring the data structures is a copy operation that takes advantage of the fact that MySQL data files are stored in a folder. The folder was backed up after the initial load of the base table. Resetting the database to the initial state only requires an overwrite of the data directory.

3.1.3 Insert Buffer versus No Insert Buffer

The comparison of the innoDB database engine with or without the insert buffer was accomplished by performing a series of insert batches into the database. To measure the difference, we measured the insertion time of multiple batches to the database. The database is initialized with the base table as described above. There are 20 sequential batches inserted into the base table. Each insert batch consisted of 50K rows inserted in a batch. The 10 million rows of data and 10 secondary indexes in the base index combine to form approximately 3.6 GB of disk space in the system. Relevant information such as MySQL global status variables were stored between runs. Figure 3 shows the results of the test run. The insert times below produce up to 8.2 times speedup in inserts. The speedup is not as good as the stated by MySQL documentation [ABa] but it does demonstrate that the MySQL InnoDB *insert buffering* significantly improves insert performance.

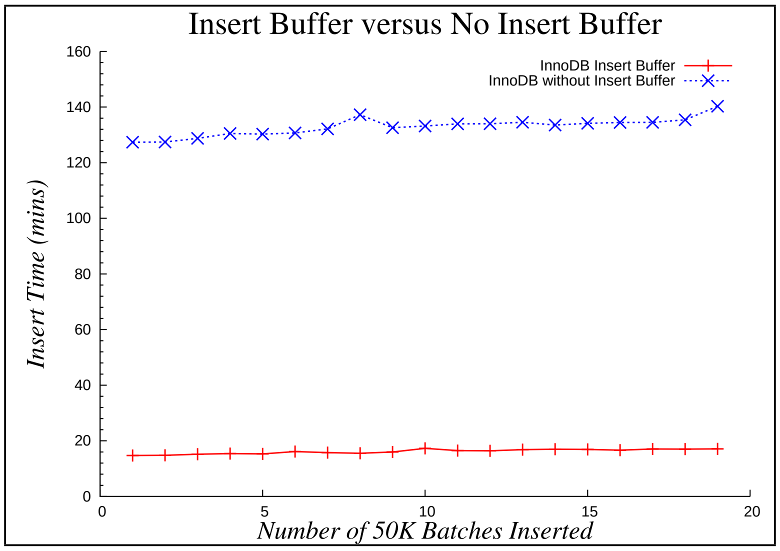

Figure 3: Comparison of Batch Inserts between Insert Buffer and No Insert Buffer on a Base table of 10 million rows

3.2 Maintenance Time

After observing InnoDB's improvement on insert performance because of *insert buffering*, the next step was to determine the difference in performance between the InnoDB engine with insert buffering (we used insert buffering of InnoDB in all of the following experiments) and the fractured index approach. This experiment involves inserting same number of rows using the standard InnoDB table and the fractured tables. In this scenario, the assumption is that the *fractured indexes* provide faster insertion times because the tuples are not inserted directly into the large main table. Each batch is consists of 40K rows of random data.

Figure 4 compares the basic insert of 40K batches between unmodified *InnoDB* (with insert buffer) and *Fractured Indexes*, running the same number of inserts as the previous experiment. The results show a significant improvement (a factor of 10) in insert time when measuring just the insertion cost of the rows into mini-indexes. It would be a factor of 100 speed up compared with the InnoDB without the insert buffer.

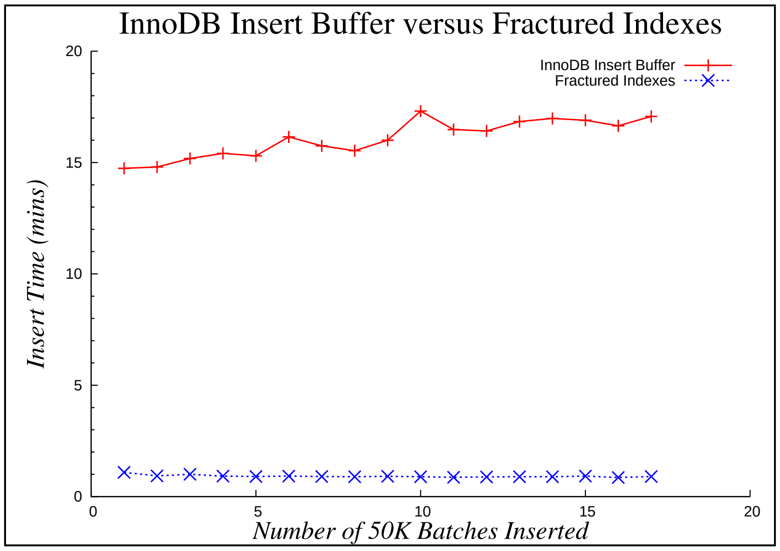

Figure 4: Batch Insert Costs of unmodified InnoDB and Fractured Indexes

3.3 Select Time

We measured the performance of SELECT queries performed on the table in both the basic approach and our fractured index approach. We rewrote queries to access each of the individual mini tables. We then merged the results together. For example, the following query on primary index key (*id*),

```
SELECT (*) FROM book WHERE id BETWEEN x AND y
```

is rewritten to include the fractured indexes, book_0, book_1, book_2.

```
  SELECT (*) FROM book WHERE id BETWEEN x AND y
+ SELECT (*) FROM book_0 WHERE id BETWEEN x AND y
+ SELECT (*) FROM book_1 WHERE id BETWEEN x AND y
+ SELECT (*) FROM book_2 WHERE id BETWEEN x AND y
```

In this experiment, we chose *x* and *y* so that 1% of tuples are retrieved from the table. The results of all queries are merged to get the complete result. The comparison of the fractured and the non-fractured approach is done by starting with the base index of 10 million rows and then inserting batches of 40K rows. The select times are then measured after each insert batch is completed. The buffers on the server are flushed between every insert or select operation. The flushing is done to ensure that the inserts are not interfering with select query performance.

The results in the Figure 5 show the query performance of both non-fractured and fractured approach after the insertion of each tuple batch.

The query performance of non fractured approach deteriorates more rapidly than fractured approach even though fractured approach has to query each fracture in addition to the main table. The query runtime after 10 insert batches is more than 4 times slower in the non fractured case. This is because the **f**ragmentation of the table. Non fractured approach inserts new tuples to random places on disk and causes data fragmentation when B-tree nodes split. The query uses the primary index to locate the range of id to be scanned, and then performs a sequential sweep to retrieve tuples until the id is out of the range. In the fractured case, the cost is purely one B-tree lookup and a sequential scan cost. However, in the non-fractured case, the cost could be one B-tree lookup and many random disk seeks because of the fragmentation. This result shows that fractured approach could be also beneficial for SELECT performance because of fragmentation.

However, note that the query performance of both fractured and non-fractured approach deteriorates over time. This is because fractured approach needs to apply the same query to more and more fractures as more rows are inserted and the overhead (query parsing, query optimization, seeks for each fracture) becomes higher.

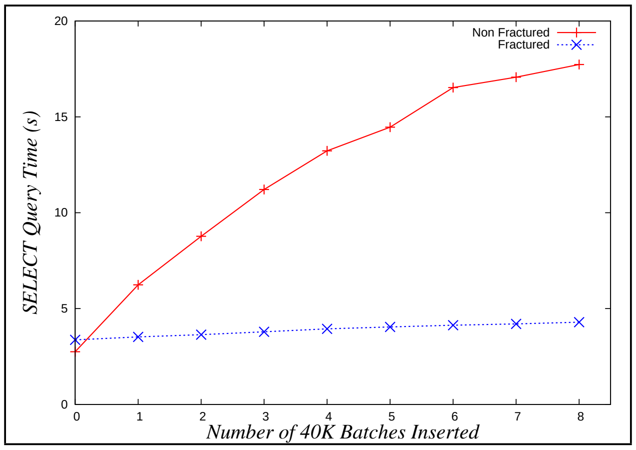

Figure 5: SELECT performance comparison between Non-Fractured and Fractured Approach. 1% Selectivity.

3.4 Merging

The previous experiment shows that the query performance could deteriorate over time because of more fractures. We have implemented the sort-merge of fractured indexes to overcome this problem as described in Section 2.4. We sequentially retrieved tuples in the order of primary key from MySQL with a connection for each fracture and merged them in a sort-merge fashion.

To enable a fair comparison, we applied a similar reorganization technique to the non-fractured approach. As described in previous section, the quick query slow down of non-fractured approach comes from fragmentation of the table. Thus, defragging the table should have a similar improvement on query performance in the non-fractured case. We defragged the table by retrieving tuples from MySQL and then loading them back to MySQL [ABb] in the sorted order. We did not use ALTER TABLE command to defrag because we found that the command does not eliminate fragmentation well if the table is larger than buffer pool.

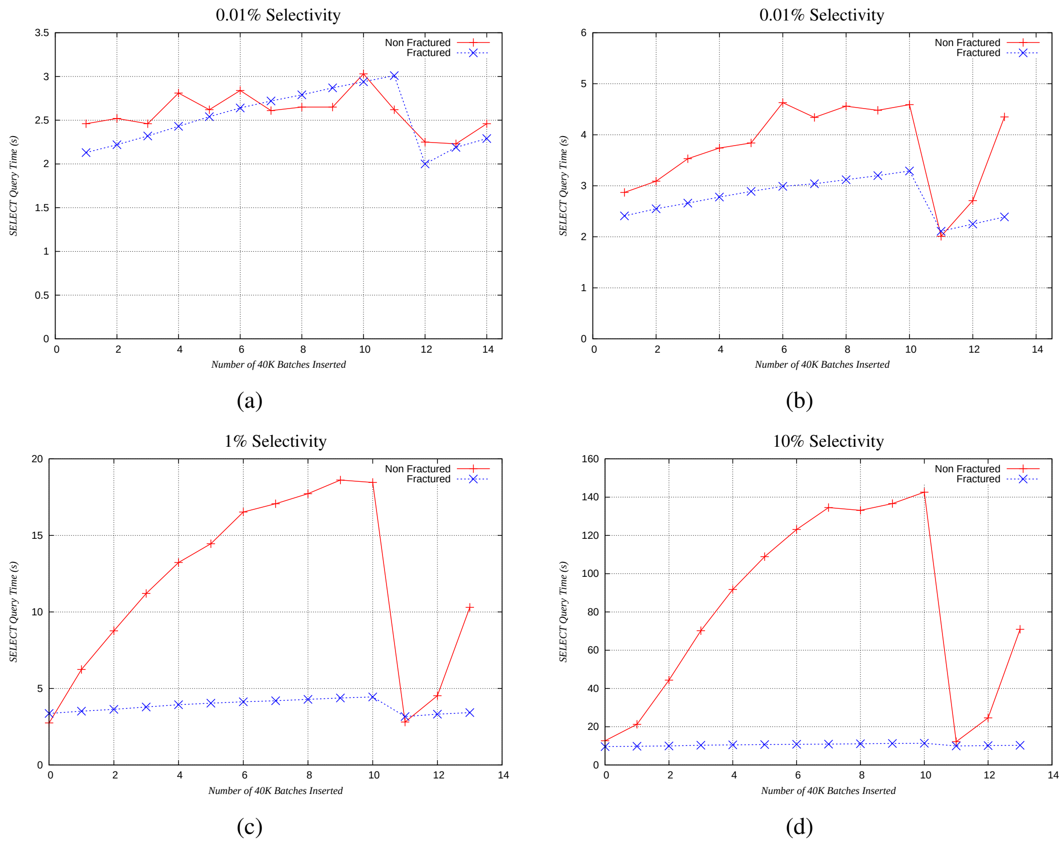

Figure 6: Query performance improvement. Databases are reorganized after 10 insert batches (Merging for fractured case, Defragging for non-fractured case)

Figure 6 shows the effect of reorganization (merging or defragging) of both fractured and non-fractured approach. We used the same query as in previous experiment, but used four queries with different selectivity settings (10%, 1%, 0.1%, 0.01%). The following steps are taken for both fractured and non-fractured table. We measured the query runtime of each query 10 times with cold cache and took average of them.

1. Start with the base index.
2. Insert 40K batches 13 times.
3. Perform a SELECT query after every insert batch.
4. Perform a Reorganization after 10 batches.

The query performance of both approaches improved after reorganization. The query performance is still slightly worse than the initial query runtime, but this is simply because the table has more data after several insert batches.

One interesting difference of the two methods is that the query slowdown in the Non-Fractured case over time is substantially milder in more selective cases such as 0.01% selectivity. This is because a selective query retrieves fewer tuples from the primary indexes, limiting the effects of data fragmentation. For example, if the query accesses only a few tuples which fits in one leaf page of the primary index, the query performance is not affected by fragmentation at all. In such a case, the overhead of fractured indexes for each fracture becomes significant and the query performance of fractured indexes could be same or worse than non-fractured B-trees. On the other hand, fractured indexes perform significantly better than non-fractured B-trees for less selective queries which is common in OLAP applications.

Another difference is that the query slowdown in non-fractured index becomes mild after several insert batches (e.g., after 6-7 batches in 1%, 10% selectivities) because the B-trees are already highly fragmented and almost every leaf page in the primary index causes a disk seek anyway. The query slowdown in fractured index grows constantly as the number of fractures to be accessed grows.

Finally, we measured the cost of reorganization operation in both approaches. The result is shown in Table 2. Although the fractured index takes a longer time to reorganize, the overhead is comparable.

Table 2: Comparison of Reorganization overhead

Approach	Load Table (hr)	Create Indexes (hr)	Total Time (hr)
Non Fractured B-tree	1.7	0.3	2.6
Fractured B-tree	1.8	0.3	2.8

4 Related Work

There has been a very large body of work on improving B-Tree query performance and maintenance cost. Most of this work tries to reduce the fragmentation of the data layout resulting from updates as well as minimize the cost of re-organizing the modified B-Tree. Although the cost of SELECT queries is usually considered, we typically try to optimize for workloads that contain a significant amount of inserts. In the absence of updates B-Tree is an extremely efficient structure that is difficult to improve upon.

In [TP02] it was observed that the B-Tree can be re-organized on disk to speed up an observed set of SELECT queries. If the values that are typically looked up together by a query can be placed sequentially on disk, the runtime of that query is reduced, as we are able to avoid expensive seeks. The advantage of applying this approach to a B-Tree is that it maintains a balance and has to be reorganized every time one of the intermediate nodes has too many (or too few) values due to updates. Thus we can continuously adjust the B-Tree at little extra cost. Note that this solution does not improve the cost of inserts.

[JDO99] presented an alternative B-Tree-like structure (called *Y-Tree*) that allows for a much faster insert rate while exhibiting a similar SELECT behavior. In Y-Tree it is possible to make sure that a batch of d values can be inserted at a cost (I/O-wise) of a single insert. This is done by increasing the size of the intermediate nodes and having value “buckets” between key values. Once the d values are inserted in their appropriate place at the top level, the largest (containing most values) bucket is drained by propagating its values to the lower level of the Y-tree. The values pushed down are not necessarily the same as the values that were inserted (since inserted values likely touch multiple intermediate nodes and we are only draining a single node); however, the number of values pushed down is the same as number of values inserted and thus the Y-Tree is able to maintain its balance invariant similar to that of a B-Tree. This solution can require very large intermediate nodes (depending on value of d) and will typically slow down SELECT queries to some degree. However, it can speed up insert queries by a factor of a 100x.

An approach very similar to ours was presented in [JNS$^+$97] - the incoming inserts are batched in memory and when the buffer fills up, its contents are written into a B-Tree, similar to our index fragment. These B-Trees are organized in a hierarchy: every time *(K-1)* B-Trees of the same size are formed they are merged into a single larger B-Tree. The hierarchical structure and merging process is very similar to the external sort algorithm and the data organization idea builds on the work in [OCGO96] by adding multiple merge levels. The advantage of this approach (as well as ours) is that it relies on a B-Tree structure rather than on a custom structure, simplifying the implementation in a database. In this work we evaluate the cost of INSERTs and realistic SELECT queries rather than single-value lookup evaluated in [JNS$^+$97]

A more general efficient storage mechanism is presented in [JOY07]. The idea is to segment the indexed data based on key ranges and then store each key-range partition in an exponential file. The exponential file keeps the data in multiple fixed size “levels” where every time a level fills up, it is merged into a next larger level. The top level of every segmented is pinned in memory to speed up INSERT operations. The work in [JOY07] evaluates SELECT and INSERT performance for several workload mixes, including a plain B-Tree, Y-Tree and LSM structure. The evaluation is done assuming a skew in inserts, while in this work we assume a uniform insert distribution. [JOY07] solution relies on a custom data structure implementation to be applied in a database.

Rose [SCB08] presents a system that extends the work in [OCGO96] and [JOY07] by storing the data in multiple index structures simultaneously and providing snapshot isolation for the queries. In addition to that, the performance is improved by applying compression that is typically used in column store databases (such as [SAB$^+$05] and MonetDB [BGvK$^+$06]).

To simplify our experiments and minimize the cost of index merge we perform a single pass merge of index fracture. However, the work in [SWSZ05] has evaluated a practical way to perform lazy merging. Lazy merging lets us to delay the merge and only merge the data as it is accessed by queries (thus avoiding some work). In addition to that a lazily merged tree is trivially available while the merge takes place – in our work we assume that one of the offline merge methods described in [SWSZ05] can be applied. The offline methods include disabling the structure undergoing a merge or doing the merge in the background and queuing the changes to be applied when the merge is done.

Although we only present experiments using simple SELECT queries in this work, most of the typical data warehouse queries can be rewritten to work in this setting. The work in [YYTM10] summarizes how to refactor and re-assemble query results. The middle-ware layer used in [YYTM10] work is very similar to our approach, although their goal is load balancing and query restart rather than improving query performance.

5 Conclusion and Future Work

We introduced the idea of using *fractured indexes* to alleviate the impact of INSERT operations in data warehouses. It is an approach that uses mini replica indexes to delay the inserts to the main table. The performance of the fractured indexes was compared to MySQL InnoDB storage engine.

Our implementation shows significant improvement in executing large batches of inserts. The speedup in previous section shows as a factor of 10 improvement over the already improved performance of the InnoDB insert buffer. We also observed the effect of using the fractured index with SELECT queries. The performance of SELECT queries degrades as the number of insert batches (mini-indexes) increase in the system. This result led to the need to merge the data in the mini-indexes back into the main table. The merging was accomplished by a sort merge algorithm which retrieves all the rows from the different tables and merges them into a new table. The merge rebuilds the table into a very efficient state that improves future queries because it performs inserts in clustered key order.

Although fractured approach system proves to be effective, more work will be needed to make it ready for use in today's databases. The merging process can be improved as it takes almost three hours to merge 11 million rows. It might be a good idea to evaluate split merges, by merging many fractured indexes into a smaller intermediate fractured tables. This will reduce the immediate merge cost compared to always merging the fractures with the main table, since the main table is usually multiple orders of magnitude larger than the fractures. Also, it would be very useful to know the optimal number of fractures needed and how often they need to be merged. This can be based on size of the data, the merge time and the hardware of the database server.

Another approach that is worth comparing to what we have already presented is the one described in [SWSZ05]. Although it clearly requires a more sophisticated implementation (requests need to be intercepted and analyzed), the lazy merging approach allows us to reuse work already done by the queries. So as queries access the main table we can merge certain parts of the fractured indexes into the part of the table that has already been read. Note that this would cause the same fragmentation that is already affecting performance in the non-fractured case. Therefore it might be a good idea to apply lazy approach the fracture merging described earlier. For example every time a query reads all or most parts of some fractures, these fractures might be merged into a single (larger) fracture.

Despite the fact that our approach is more useful in Data Warehouses where the cost of a single query is relatively high, the availability and performance of the system is still important. Therefore, some research or planning needs to be done to ensure that the data remains online and at acceptable performance while the merging occurs. This may be solved by merging in a backup database.

References

[ABa] MySQL AB. *MySQL 5.1 Reference Manual*. Sun Microsystems.

[ABb] MySQL AB. *MySQL 5.1 Reference Manual InnoDB Defragmenting*. Sun Microsystems.

[BGvK$^+$06] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. Monetdb/xquery: a fast xquery processor powered by a relational engine. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 479–490, New York, NY, USA, 2006. ACM.

[BM72] R. Bayer and EM McCreight. Organization and maintenance of large ordered indexes. *Acta informatica*, 1(3):173–189, 1972.

[CD97] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *ACM Sigmod record*, 26(1):65–74, 1997.

[JDO99] Chris Jermaine, Anindya Datta, and Edward Omiecinski. A novel index supporting high volume data warehouse insertion. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 235–246, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[JNS$^+$97] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and Rama Kanneganti. Incremental organization for data recording and warehousing. In *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 16–25, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[JOY07] Christopher Jermaine, Edward Omiecinski, and Wai Gen Yee. The partitioned exponential file for database storage management. *The VLDB Journal*, 16(4):417–437, 2007.

[Knu73] D.E. Knuth. *The art of computer programming. Vol. 3, Sorting and Searching*. Addison-Wesley Reading, MA, 1973.

[OCGO96] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, June 1996.

[SAB$^+$05] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: a column-oriented dbms. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[SCB08] Russell Sears, Mark Callaghan, and Eric Brewer. Rose: compressed, log-structured replication. *Proc. VLDB Endow.*, 1(1):526–537, 2008.

[SWSZ05] Xiaowei Sun, Rui Wang, Betty Salzberg, and Chendong Zou. Online b-tree merging. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 335–346, New York, NY, USA, 2005. ACM.

[TP02] Yufei Tao and Dimitris Papadias. Adaptive index structures. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 418–429. VLDB Endowment, 2002.

[YYTM10] C. Yang, C. Yen, C. Tan, and S.R. Madden. Osprey: Implementing MapReduce-Style Fault Tolerance in a Shared-Nothing Distributed Database. In *ICDE '10: Proceedings of the 26th International Conference on Data Engineering*, 2010.