BaSE (Byte addressable Storage Engine) Access Method

Krishnaprasad Shastry
Hewlett-Packard ISO Pvt. Ltd.
Bangalore.
krishnaprasad.shastry@hp.com

Shine Mathew
Hewlett-Packard ISO Pvt. Ltd.
Bangalore.
shinem@hp.com

Sathyanarayanan Manamohan
Hewlett-Packard ISO Pvt. Ltd.
Bangalore.
sathya@hp.com

Goetz Graefe
Hewlett-Packard Laboratories,
Palo Alto, CA, USA.
goetz.graefe@hp.com

Abstract

Non-Volatile Memory (NVM) is an emerging memory technology that combines the best properties of today's hard disks and today's main memory by combining non-volatility, high density, high speed, and byte addressability. This provides an opportunity to redesign systems and their software stacks to improve performance and to reduce the system and software complexity. Present-day database systems are designed and optimized for traditional disks and deep memory hierarchies. This makes them very complex because they have to handle varying levels of storage latencies, from CPU caches to hard disks. Our intention is to build a prototype storage engine optimized for NVM to take advantage of the collapsed memory hierarchy, and to develop this storage engine in an incremental way. In this paper, we discuss the optimizations for the data access module. We modified the B-tree access module of an open source storage engine, which reduced the lock contention by 99.6%, i.e., by a factor of 273.

1. Introduction

At its heart, a typical RDBMS has five main components – Client communication Manager, Process manager, Relational query processor, Transactional storage manager and Shared components and Utilities [HSH07]. Transactional storage manager typically encompasses four deeply intertwined components – Lock manager, log manager, buffer pool and access methods [HSH07]. Our intention is to develop novel technologies to build a transaction storage manager, for the NVM environment in a phased manner. We will call transactional storage manager as transactional storage engine in the remainder of this document.

Non Volatile Memory (NVM) is an emerging memory technology that combines best properties of today's hard disks and main memory and offers non-volatility, high speed and byte addressability.

We have a significant opportunity to redesign transactional storage engine modules for NVM and improve performance substantially. We aim to develop such an optimized storage system by redesigning these modules one by one in a phased manner. In this paper, we focus on optimizing the Access methods.

B-tree lookup and latching are the two significant contributors for Access method performance. Latching was not seen as a major overhead earlier because IO overheads masked it in traditional disk based database systems. Once IO overheads are eliminated, latching becomes a significant overhead. As shown in figure 1, Latching accounts for about 14% of the instructions, and is primarily important in the create record and B-tree lookup portions of the transaction [HAMS08]. Hence minimizing latch contention can significantly improve the concurrency and system performance.

The 19th International Conference on Management of Data (COMAD),
19th-21st Dec, 2013 at Ahmedabad, India.

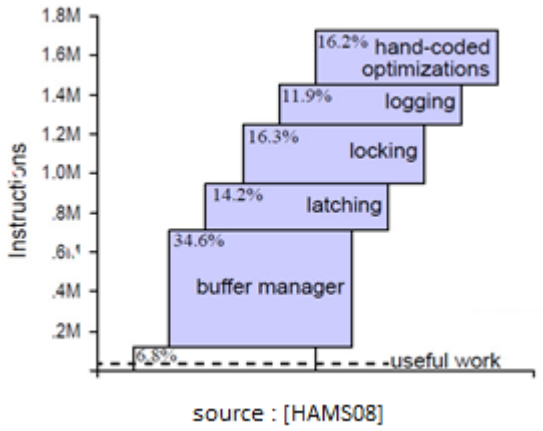


Figure 1

Foster B-trees [GKK12] are a new variant of B-trees that combine advantages of prior B-tree variants optimized for multi-core processors and modern memory hierarchies with flash storage and nonvolatile memory. Foster B-trees reduces the number of latches and the latch duration.

In our work, we demonstrate the applicability of Foster B-trees to a popular database management system. We have chosen the open source DBMS MariaDB and the associated storage engine XtraDB, which are forks of MySQL DBMS and InnoDB storage engine. We integrated Foster B-trees with XtraDB. We optimized XtraDB B-tree access methods and latching to leverage the Foster B-tree design principles.

To optimize the performance, we replace the multi-pass insert algorithm in XtraDB with a single-pass algorithm. We eliminate the "pessimistic insert" method and minimize data redistribution during tree splits. We also implement U-latch to support optimistic adoption of the foster child.

The performance evaluation shows a significant reduction in latch contentions and a good improvement in response time.

When the entire database is available in memory, the behavior exhibited by the storage engine can be compared with the one running on a system built with NVM. This makes the results observed in this experiment applicable for NVM.

1.1 Structure of this document

The remainder of this paper is organized as follows. In section 2, we describe the B-tree implementation in XtraDB and Foster B-tree data structures. In section 3, we discuss how Foster B-tree was integrated with XtraDB and highlight the unique contributions we made. In section 4, we describe how the performance improvement was evaluated and conclude the work in section 5.

2. B-tree Data Structures

2.1 B-tree implementation in XtraDB

This description is based on our analysis of the XtraDB code. To the best of our knowledge, this has not been documented in any publicly available paper.

XtraDB storage engine uses modified B^{link} trees [YB87] to store the data. In XtraDB storage engine, the nodes of the B^{link} tree are often referred as pages, and we follow the same notation in this document. User records are stored only at the leaf pages. Intermediate pages and the root page store several special records known as "node pointers". Each node pointer points to a child of the page. The "node pointers" are built from the value of the indexing "key" of the given index/table and the identification number of its child page. The identification number associated with a page is referred as "page number". At any given level, the pages are connected to the neighboring pages using "next" and "previous" pointers. The leaf pages are always considered as "level 0" and the level increases towards the root page. XtraDB storage engine ensures the page number associated with a root page of any B^{link} tree remains intact as long as the tree exists. Figure 2 illustrates a simplified representation of the B^{link} tree used in XtraDB.

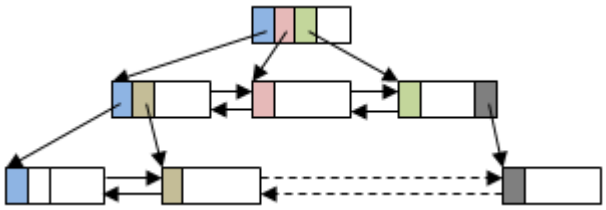


Figure 2

Within a page, records are stored as a linked list in sorted order. To reduce the access time, XtraDB uses a shortcut pointer, which is referred as a "directory slot". The offset of the start of every sixth record in the linked list is stored in one directory slot, and the number of directory slots increases as more records are inserted into the page. Directory slots are built from the values of the indexing keys and the offsets to the corresponding records. Figure 3 provides the graphical representation of "directory slot".

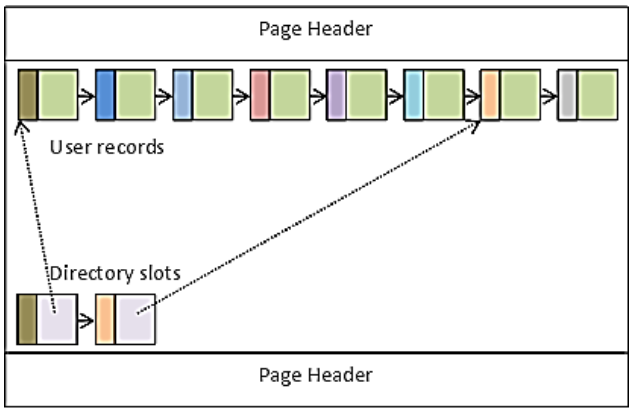


Figure 4

In XtraDB, each page is 16KB in size. Out of this, 128 bytes are used for bookkeeping and the remaining space is available for storing records. The user record grows towards the higher memory address whereas the "directory slots" grow towards the lower memory addresses. Figure 4 illustrates the page layout.

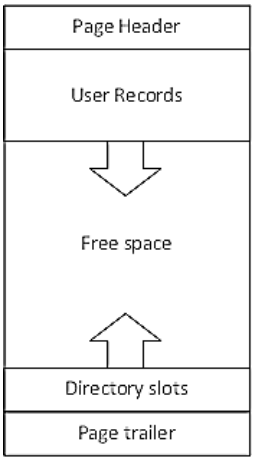


Figure 4

Tree traversal in XtraDB starts at the root page and then, goes down to the leaf level. If a search key is available, then it uses the given search key to identify the child page and repeats the same operation until the cursor reaches the leaf page. In case of tree traversal without a search key, the cursor will be positioned at the left most leaf page for forward scan and right most leaf page for reverse scan. Once the cursor is positioned at a leaf page, XtraDB uses the "next" (or "previous") pointers to move to the next (or previous) page and read the records. Because of the "next" and "previous" pointers in a page, a minimum of four pages (current page, new page, next/previous page and the parent page) must be latched in case of a page split.

In XtraDB, the tree modifications are performed under an x-latch on the index data structure. This can be considered as a tree latch, which blocks other concurrent accesses of the tree. The x-latch on the index becomes a bottleneck in the case of highly concurrent tree modifications. The existing XtraDB implementation holds latches on index data structure, parent page, current page, new page and next or previous page during a page spilt.

2.2 Foster B-trees

Foster B-trees [GKK12] are a new variant of B-trees that combine advantages of prior B-tree variants optimized for multi-core processors and modern memory hierarchies with flash storage and nonvolatile memory. The defining properties of Foster B-trees are a single incoming pointer per node at all times, fence keys in every node, and structural operations similar to B^{link} trees. Single incoming pointer to a node reduces the number of latches required in case of node split, which is ideal for multi-core architectures. Figure 5 provides the pictorial representation of a Foster B-tree.

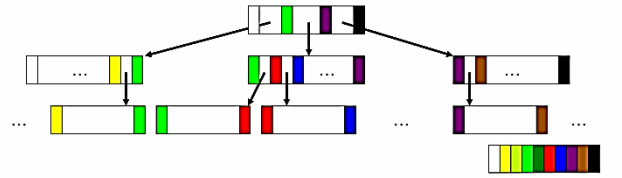


Figure 5

3. Foster B-tree implementation in XtraDB

3.1 Data structure changes for implementing Foster B-tree

We redesigned the B^{link} tree data structures of XtraDB to implement Foster B-tree. Since Foster B-trees do not need next and previous pointers, we removed these from the existing B-tree structure and added new members to store fence keys and foster child pointers. Figure 6 illustrates the new XtraDB page layout.

The pointers to fence keys are kept at a fixed location (x+0x3FF0 and x+0x3FF2 where x is the start address of the page) for quicker access, but the actual fence key can reside anywhere in the user record space. It should be noted that, the fence keys are not part of the user record chain. New methods are implemented to access fence keys. Fence key pointer initialized to 0 indicates +/- infinity as fence key. The fence keys are built from the values of indexing key.

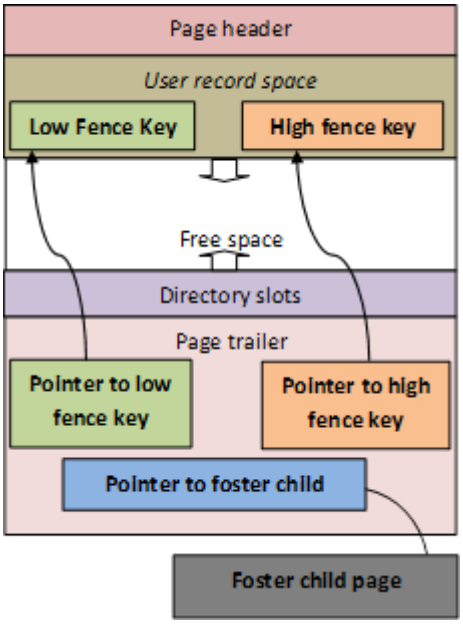


Figure 6

To ensure consistency of new storage engine, every page must satisfy two conditions. The first condition defines the valid range for key values of user records for any given page. The valid key values must be less than the high fence key and must be greater than or equal to the low fence key. The second condition defines the relation between the fence keys of two consecutive pages at any given level. The high fence key of the left page must be equal to the low fence key of the right page.

3.2 Algorithm changes for reducing latch contention

To gain the benefits of Foster B-tree and NVM, some of the basic algorithms are redesigned and a few new algorithms are added. We redesigned the B-tree modification algorithm and added new algorithms for tree traversal.

The traditional insert algorithm is based on a multi-pass approach. This approach works well with the current hardware and software stack. With Foster B trees, a single pass is enough. Table 1 shows a high-level comparison between the old algorithm and the new algorithm. The new algorithm is built into XtraDB's "optimistic insert" method.

The flowcharts of the original XtraDB algorithm and the new algorithm are shown in the Appendix. Flowchart 1 represents original algorithm used by XtraDB and Flowchart 2 represents new algorithm.

In case of traditional XtraDB, the "next" and "previous" pointers are used to move the cursor from one page to another after control reaches the desired level. Foster B-tree has removed the "next" and previous" pointers and the tree traversal always start from the root page. We move from a page to the next using the value of fence keys. High fence key of the current page is used to identify the next page and Low fence is used to identify the previous page.

In the future, the need to start traversal from the root page can be eliminated by using page caching. With this technique, to move to the next or the previous page, we need to move only one level up in the tree. The page cache will be invalidated in the case of a structural change to the B-tree and the traversal must start from the root again.

Table 1

S.I	XtraDB	XtraDB + Foster B tree
1	Multi-pass approach (optimistic[1] insert followed by pessimistic[2] insert)	Single-pass approach (only optimistic insert)
2	Multiple tree traversal	Single tree traversal
3	Holds x-latch on the index for a long duration.	Holds x-latch on the index for a shorter duration. The x-latch on the index is held only during the foster child adoption.
4	In case of page split, a minimum of four nodes are x-latched.	In case of page split, a maximum of two nodes are x-latched.
5	In case of page split, the tree structure changes	In case of page split, foster child is created and attached to current page.
6	New page is attached to the parent page during the page split.	Foster child is adopted during tree traversal.

1. Optimistic insert assumes the new record will fit into the page when it inserts it. If this fails, storage engine will try the second attempt using pessimistic insert algorithm.
2. Pessimistic insert will anticipate a page split and handle it appropriately.

To gain the benefits of Foster B-tree in XtraDB, we redesigned the B-tree modification algorithm and added new algorithms for tree traversal. For inserting new records, XtraDB first attempts an "optimistic insert" algorithm. If this fails, it retries with a "pessimistic insert" algorithm. This multi-pass approach degrades the performance and is not in alignment with design

principles of Foster B-trees. To improve the performance, we eliminated "pessimistic insert" method and redesigned XtraDB "optimistic insert" to handle page splits.

The "pessimistic insert" method is used whenever the insert results in page split. The "pessimistic insert" splits the page, moves half of the records to the new page and then, inserts the new record into the appropriate page. Data redistribution happens in all cases except when the new record is positioned at the end of the current page. XtraDB holds an x-latch on the index for the entire duration of the "pessimistic insert". We added new algorithms to the "optimistic insert" method to handle page splits and data redistribution. If the new record cannot fit into the current page, then we create a foster-child and attach it to the current page while holding an x-latch on the foster-parent and foster-child. This reduces the number of x-latches acquired and eliminates the x-latch on the index during inserts. Our new algorithm always tries to minimize the data movement. We achieved this by not always splitting the page in the middle – instead the page is split optimally based on position of the new record and size of fence keys. Table 1 lists the changes between the existing algorithm and the new algorithm.

We attempt foster adoption during tree traversal and implement opportunistic adoption and forced adoption algorithms to optimize the overall performance. During the tree traversal, we perform opportunistic adoption if an x-latch can be acquired on the index in non-blocking mode. If the attempt to acquire x-latch on the index without blocking fails, foster adoption does not occur. This will eventually lead to the formation of foster chain. Once foster chain length exceeds the pre-defined threshold, we perform forced foster adoption. During forced foster adoption, we block until we acquire the x-latch on the index.

We implemented u-latch, update intended latch, to support opportunistic foster adoption. A successful u-latch elevates an existing s-latch on the index to an x-latch. During tree traversal u-latch is attempted in non-blocking mode. If it is successful, opportunistic foster adoption is performed. During foster adoption only the parent and foster child pages are x-latched. This approach reduces the number of x-latches acquired and the duration of x-latch on the index significantly.

4. Performance evaluation

We evaluated the performance improvements of XtraDB with Foster B-tree Access method in three phases. In the first phase, we measured the reduction in the number of x-latches acquired during the concurrent random inserts. In the second phase, we measured the throughput for a single worker thread inserting random records and, we evaluated the throughput improvement for concurrent random inserts in the last phase.

In XtraDB, only INSERT operations cause page splits and foster child creation. In case of delete operations, XtraDB sets a special flag to mark the records as deleted and does not change the tree structure immediately. Also, the DELETE algorithm for a Foster B-tree acquires the same number of latches as the algorithm for DELETE on a regular B-tree. So we focused on just INSERT operations for our performance measurements. No standard benchmark is available for measuring the insert performance alone for a RDBMS; so we developed a multi-threaded application to perform random inserts and to measure the elapsed time. We instrumented the XtraDB code to monitor the latches acquired during database operations.

We evaluated the performance on a system with 8 Intel cores and 16GB RAM running SuSE Linux 2.6.32. We used MariaDB 5.2.7 release for our evaluation. The test application creates a table with a record length of 80 bytes. The data types of the columns or primary key do not have any effect on the number of x-latches. We chose an integer column as the primary key to make dynamic data generation easy.

To evaluate latch contention, we loaded the table with 1 million records and then, randomly inserted 1 million records each from 16 concurrent threads. The seed values for the random number generator were chosen carefully to minimize the chances of generating non-unique values as keys. If any non-unique keys were generated, the worker threads ignore that record and continue with the rest of the records.

As shown in table 2 and figure 7, Foster B-trees reduce the latch contentions during inserts by 99.6%, i.e., by a factor of 273.

Table 2

	XtraDB	FBT	Improvement
Number of Records	16987225	16987225	
#X-Latch (index + B-tree pages)	884321	3231	99.6%

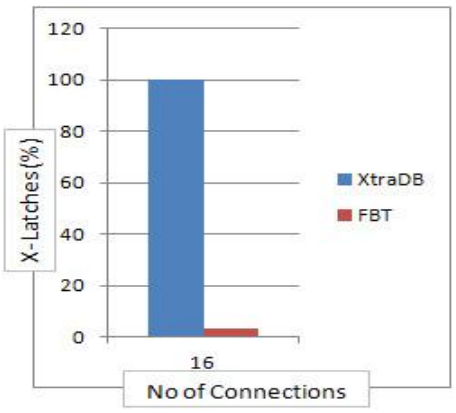


Figure 7

To evaluate the impact of the new tree traversal algorithms we introduced, we ran the same application with one worker thread to insert 1 million, 2 million, 4 million, 8 million and 16 million records. Since there is no latch contention in this experiment, we expected the performance of the Foster B-tree to be slightly worse than the default implementation because of the increased overheads while traversing and maintaining the tree. As shown in table 3 and figure 8, Foster B-trees performed on par with default XtraDB. This shows the overheads introduced by Foster B-tree are negligible.

Table 3

No of Records (million)	Average Time (s) (XtraDB)	Average Time (s) (FBT)
1	76	73
2	151	151
4	299	302
8	617	606
16	1219	1214

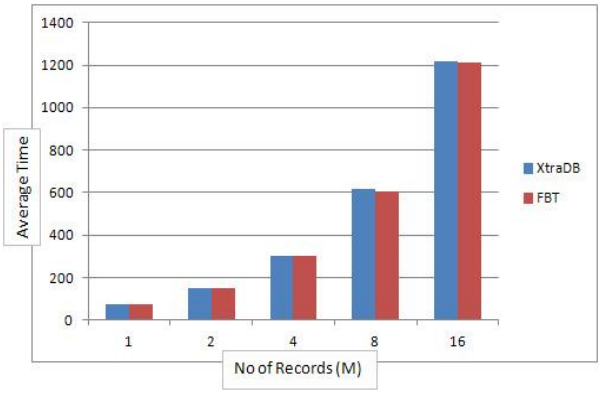


Figure 8

To evaluate the performance improvement in throughput, we ran the same application with 16, 32 and 64 concurrent connections. Each worker thread inserted 1 million random records. As shown in Table 4 and Figure 9, Foster B-trees perform better with more number of concurrent connections. The performance improves linearly with increasing the number of threads and with 64 concurrent connections, it performs 20% better than default XtraDB.

Table 4

No of Connections	Average Time in seconds (XtraDB)	Average Time in seconds (FBT)	Improvement in %
16	435	433	0.461894
32	441	406	8.62069
64	461	385	19.74026

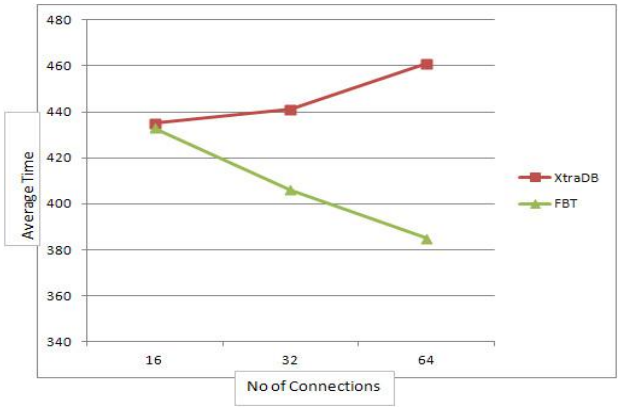


Figure 9

5. Conclusions

We have successfully demonstrated the benefits of Foster B-tree using MariaDB/XtraDB DMBS. We have integrated Foster B-tree with XtraDB and redesigned its algorithms to reap the benefits of Foster B-tree to improve the performance. We have redesigned the XtraDB page layout to have only one incoming pointer and removed the next and previous pointer. We have eliminated the pessimistic-inserts and implemented new algorithms to support “optimistic insert” to handle page splits. We have implemented an advanced approach to adopt foster children using U-latch. We have instrumented the XtraDB storage engine to monitor the number of latches acquired during database operations. We have evaluated the performance on an environment where the entire data set fit into memory and have recorded a significant reduction

in latch contention on B-tree data structures.

The results show the Foster B-tree is an ideal Access method for transactional storage engines designed for a NVM that supports byte addressing.

References

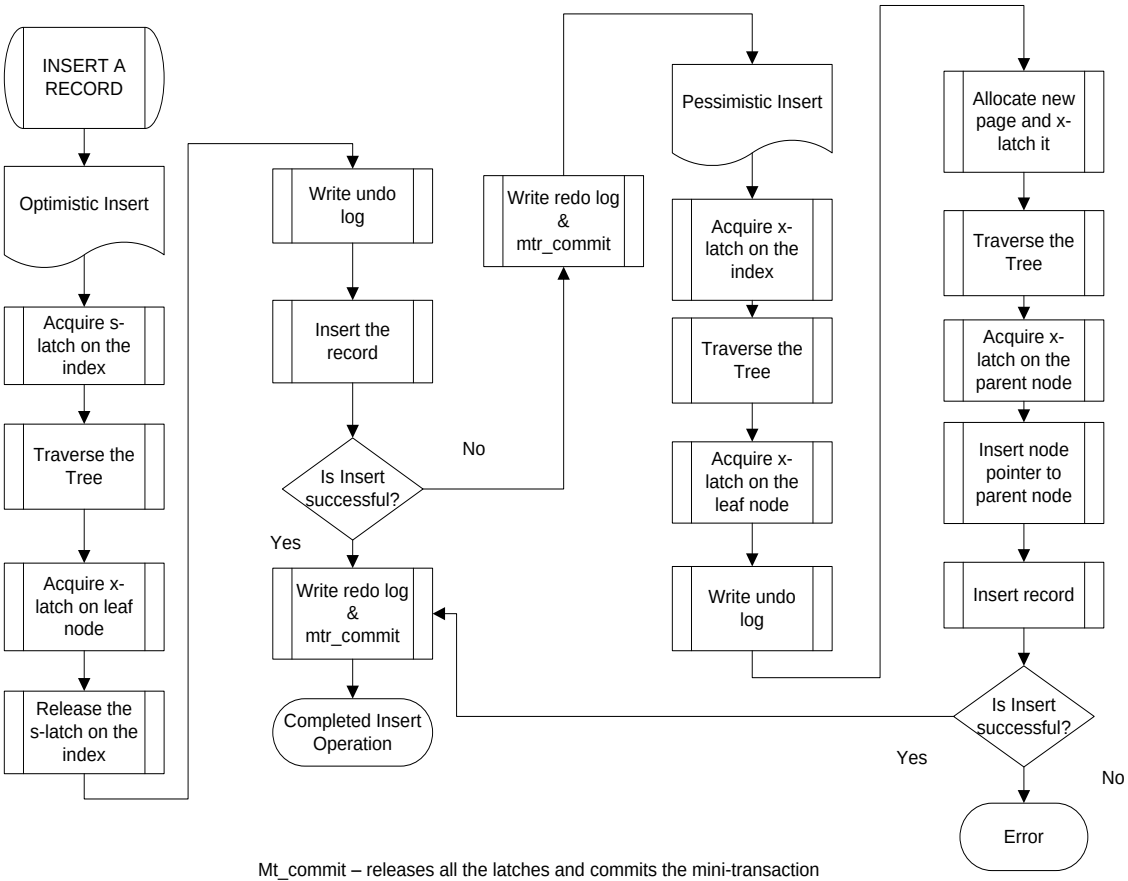
[HAMS08] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, Michael Stonebraker "OLTP Through the Looking Glass, and What We Found There" SIGMOD 2008

[GKK12] Goetz Graefe, Hideaki Kimura, Harumi A. Kuno: Foster b-trees. ACM Trans. Database Syst. 37(3): 17 (2012)

[HSH07] Joseph M. Hellerstein, Michael Stonebraker, James Hamilton: Architecture of a Database System. Foundations and Trends® in Databases Vol. 1, No. 2 (2007) 141–259

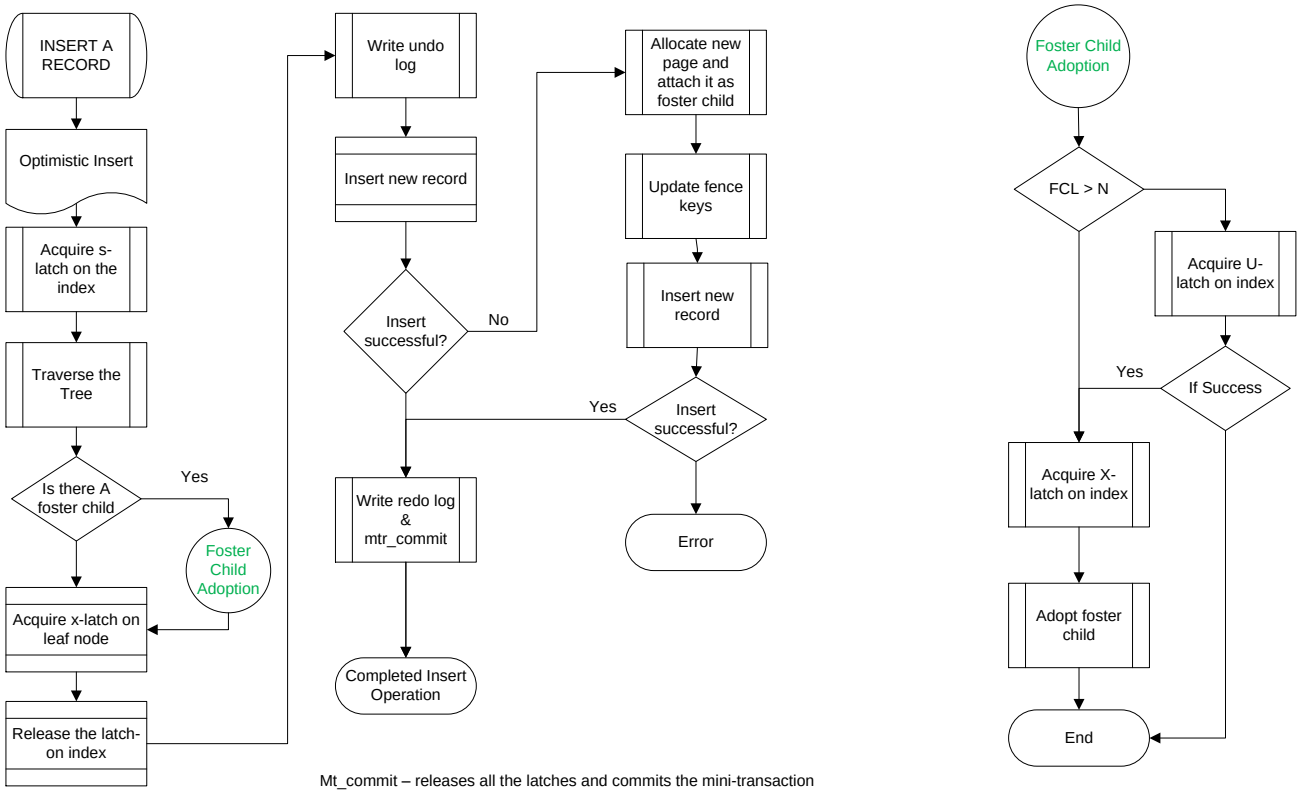
[LY81] P. L. Lehman, S. B. Yao: Efficient locking for concurrent operations on B-trees. ACM Trans. Database Syst., vol. 6, pp. 650–670, December 1981.

Appendix



Mt_commit – releases all the latches and commits the mini-transaction

Flowchart 1



Mt_commit – releases all the latches and commits the mini-transaction

Flowchart 2