The Log-Structured Merge-Tree (LSM-Tree)

Patrick O'Neil[1], Edward Cheng[2]
Dieter Gawlick[3], Elizabeth O'Neil[1]
To be published: Acta Informatica

ABSTRACT. High-performance transaction system applications typically insert rows in a History table to provide an activity trace; at the same time the transaction system generates log records for purposes of system recovery. Both types of generated information can benefit from efficient indexing. An example in a well-known setting is the TPC-A benchmark application, modified to support efficient queries on the History for account activity for specific accounts. This requires an index by account-id on the fast-growing History table. Unfortunately, standard disk-based index structures such as the B-tree will effectively double the I/O cost of the transaction to maintain an index such as this in real time, increasing the total system cost up to fifty percent. Clearly a method for maintaining a real-time index at low cost is desirable. The Log-Structured Merge-tree (LSM-tree) is a disk-based data structure designed to provide low-cost indexing for a file experiencing a high rate of record inserts (and deletes) over an extended period. The LSM-tree uses an algorithm that defers and batches index changes, cascading the changes from a memory-based component through one or more disk components in an efficient manner reminiscent of merge sort. During this process all index values are continuously accessible to retrievals (aside from very short locking periods), either through the memory component or one of the disk components. The algorithm has greatly reduced disk arm movements compared to a traditional access methods such as B-trees, and will improve cost-performance in domains where disk arm costs for inserts with traditional access methods overwhelm storage media costs. The LSM-tree approach also generalizes to operations other than insert and delete. However, indexed finds requiring immediate response will lose I/O efficiency in some cases, so the LSM-tree is most useful in applications where index inserts are more common than finds that retrieve the entries. This seems to be a common property for History tables and log files, for example. The conclusions of Section 6 compare the hybrid use of memory and disk components in the LSM-tree access method with the commonly understood advantage of the hybrid method to buffer disk pages in memory.

1. Introduction

As long-lived transactions in activity flow management systems become commercially available ([10], [11], [12], [20], [24], [27]), there will be increased need to provide indexed access to transactional log records. Traditionally, transactional logging has focused on aborts and recovery, and has required the system to refer back to a relatively short-term history in normal processing with occasional transaction rollback, while recovery was performed using batched sequential reads. However, as systems take on responsibility for more complex activities, the duration and number of events that make up a single long-lived activity will increase to a point where there is sometimes a need to review past transactional steps in real time to remind users of what has been accomplished. At the same time, the total number of active events known to a system will increase to the point where memory-resident data structures now used to keep track of active logs are no longer feasible, notwithstanding the continuing decrease in memory cost to be expected. The need to answer queries about a vast number of past activity logs implies that indexed log access will become more and more important.

[1]Dept. of Math & C.S, UMass/Boston, Boston, MA 02125-3393, {poneil | eoneil}@cs.umb.edu
[2]Digital Equipment Corporation, Palo Alto, CA 94301, edwardc@pa.dec.com
[3]Oracle Corporation, Redwood Shores, CA, dgawlick@us.oracle.com

Even with current transactional systems there is clear value in providing indexing to support queries on history tables with high insert volume. Networking, electronic mail, and other nearly-transactional systems produce huge logs often to the detriment of their host systems. To start from a concrete and well-known example, we explore a modified TPC-A benchmark in the following Examples 1.1 and 1.2. Note that examples presented in this paper deal with specific numeric parametric values for ease of presentation; it is a simple task to generalize these results. Note too that although both history tables and logs involve time-series data, the index entries of the LSM-Tree are not assumed to have indentical temporal key order. The only assumption for improved efficiency is high update rates compared to retrieval rates.

The Five Minute Rule

The following two examples both depend on the Five Minute Rule [13]. This basic result states that we can reduce system costs by purchasing memory buffer space to keep pages in memory, thus avoiding disk I/O, when page reference frequency exceeds about once every 60 seconds. The time period of 60 seconds is approximate, a ratio between the amortized cost for a disk arm providing one I/O per second and memory cost to buffer a disk page of 4 KBytes amortized over one second. In terms of the notation of section 3, the ratio is $COST_P/COST_m$ divided by the page size in Mbytes. Here we are simply trading off disk accesses for memory buffers while the tradeoff gives economic gain. Note that the 60 second time period is expected to grow over the years as memory prices come down faster than disk arms. The reason it is smaller now in 1995 than when defined in 1987 when it was five minutes, is partly technical (different buffering assumptions) and partly due to the intervening introduction of extremely inexpensive mass-produced disks.

Example 1.1. Consider the multi-user application envisioned by the TPC-A benchmark [26] running 1000 transactions per second (this rate can be scaled, but we will consider only 1000 TPS in what follows). Each transaction updates a column value, withdrawing an amount Delta from a Balance column, in a randomly chosen row containing 100 bytes, from each of three tables: the Branch table, with 1000 rows, the Teller table with 10,000 rows, and the Account table, with 100,000,000 rows; the transaction then writes a 50 byte row to a History table before committing, with columns: Account-ID, Branch-ID, Teller-ID, Delta, and Timestamp.

Accepted calculations projecting disk and memory costs shows that Account table pages will not be memory resident for a number of years to come (see reference [6]), while the Branch and Teller tables should be entirely memory resident now. Under the assumptions given, repeated references to the same disk page of the Accounts table will be about 2,500 seconds apart, well below the frequency needed to justify buffer residence by the Five Minute rule. Now each transaction requires about two disk I/Os, one to read in the desired Account record (we treat the rare case where the page accessed is already in buffer as insignificant), and one to write out a prior dirty Account page to make space in buffers for a read (necessary for steady-state behavior). Thus 1000 TPS will correspond to about 2000 I/Os per second. This requires 80 disk arms (actuators) at the nominal rate of 25 I/Os per disk-arm-second assumed in [13]. In the 8 years since then (1987 to 1995) the rate has climbed by less than 10%/year so that the nominal rate is now about 40 I/Os per second, or 50 disk arms for 2000 I/Os per second. The cost of disk for the TPC application was calculated to be about half the total cost of the system in [6], although it is somewhat less on IBM mainframe systems. However, the cost for supporting I/O is clearly a growing component of the total system cost as the cost of both memory and CPU drop faster than disk.

Example 1.2. Now we consider an index on the high insert volume History table, and demonstrate that such an index essentially *doubles* the disk cost for the TPC application. An

index on "Account-ID concatenated with Timestamp" (Acct-ID||Timestamp) for the History table is crucial to support efficient queries on recent account activity such as:

```
(1.1) Select  *  from History
          where History.Acct-ID = %custacctid
          and History.Timestamp > %custdatetime;
```

If an Acct-ID||Timestamp index is not present, such a query requires a direct search of all rows of the History table, and thus becomes impractical. An index on Acct-ID alone provides most of the benefit, but cost considerations that follow don't change if the Timestamp is left out, so we assume here the more useful concatenated index. What resources are required to maintain such a secondary B-tree index in real time? We see that the entries in the B-tree are generated 1000 per second, and assuming a 20 day period of accumulation, with eight hour days and 16 byte index entries, this implies 576,000,000 entries on 9.2 GBytes of disk, or about 2.3 million pages needed on the index leaf level, even if there is no wasted space. Since transactional Acct-ID values are randomly chosen, each transaction will require at least one page read from this index, and in the steady state a page write as well. By the Five Minute Rule these index pages will not be buffer resident (disk page reads about 2300 seconds apart), so all I/Os are to disk. This addition of 2000 I/Os per second to the 2000 I/Os already needed for updating the Account table, requires a purchase of an additional 50 disk arms, doubling our disk requirements. The figure optimistically assumes that deletes needed to keep the log file index only 20 days in length can be performed as a batch job during slack use times.

We have considered a B-tree for the Acct-ID||Timestamp index on the History file because it is the most common disk-based access method used in commercial systems, and in fact no classical disk indexing structure consistently gives superior I/O cost/performance. We will discuss the considerations that lead us to this conclusion in Section 5.

The LSM-tree access method presented in this paper enables us to perform the frequent index inserts for the Account-ID||Timestamp index with much less disk arm use, therefore at an order of magnitude lower cost. The LSM-tree uses an algorithm that *defers* and *batches* index changes, migrating the changes out to disk in a particularly efficient way reminiscent of merge sort. As we shall see in Section 5, the function of *deferring* index entry placement to an ultimate disk position is of fundamental importance, and in the general LSM-tree case there is a cascaded series of such deferred placements. The LSM-tree structure also supports other operations of indexing such as deletes, updates, and even *long latency* find operations with the same deferred efficiency. Only finds that require immediate response remain relatively costly. A major area of effective use for the LSM-tree is in applications such as Example 1.2 where retrieval is much less frequent than insert (most people don't ask for recent account activity nearly as often as they write a check or make a deposit). In such a situation, reducing the cost of index inserts is of paramount importance; at the same time, find access is frequent enough that an index of some kind must be maintained, because a sequential search through all the records is out of the question.

Here is the plan of the paper. In Section 2, we introduce the two-component LSM-tree algorithm. In Section 3, we analyze the performance of the LSM-tree, and motivate the multi-component LSM-tree. In Section 4 we sketch the concepts of concurrency and recovery for the LSM-tree. In Section 5 we consider competing access methods and their performance for applications of interest. Section 6 contains conclusions, where we evaluate some implications of the LSM-tree, and provide a number of suggestions for extensions.

2. The Two Component LSM-Tree Algorithm

An LSM-tree is composed of two or more tree-like component data structures. We deal in this Section with the simple two component case and assume in what follows that LSM-tree is indexing rows in a History table as in Example 1.2. See Figure 2.1, below.

A two component LSM-tree has a smaller component which is entirely memory resident, known as the C_0 tree (or C_0 component), and a larger component which is resident on disk, known as the C_1 tree (or C_1 component). Although the C_1 component is disk resident, frequently referenced page nodes in C_1 will remain in memory buffers as usual (buffers not shown), so that popular high level directory nodes of C_1 can be counted on to be memory resident.

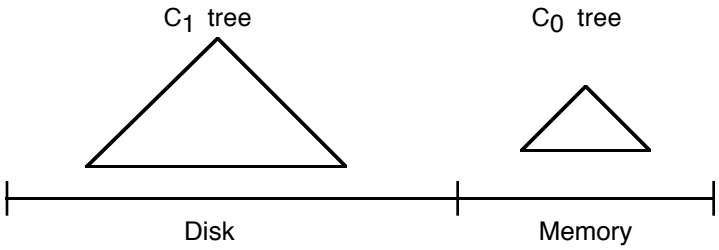

Figure 2.1. Schematic picture of an LSM-tree of two components

As each new History row is generated, a log record to recover this insert is first written to the sequential log file in the usual way. The index entry for the History row is then inserted into the memory resident C_0 tree, after which it will in time migrate out to the C_1 tree on disk; any search for an index entry will look first in C_0 and then in C_1. There is a certain amount of latency (delay) before entries in the C_0 tree migrate out to the disk resident C_1 tree, implying a need for recovery of index entries that don't get out to disk prior to a crash. Recovery is discussed in Section 4, but for now we simply note that the log records that allow us to recover new inserts of History rows can be treated as logical logs; during recovery we can reconstruct the History rows that have been inserted and simultaneously recreate any needed entries to index these rows to recapture the lost content of C_0.

The operation of inserting an index entry into the memory resident C_0 tree has no I/O cost. However, the cost of memory capacity to house the C_0 component is high compared to disk, and this imposes a limit on its size. We need an efficient way to migrate entries out to the C_1 tree that resides on the lower cost disk medium. To achieve this, whenever the C_0 tree as a result of an insert reaches a threshold size near the maximum allotted, an ongoing *rolling merge* process serves to delete some contiguous segment of entries from the C_0 tree and merge it into the C_1 tree on disk. Figure 2.2 depicts a conceptual picture of the rolling merge process.

The C_1 tree has a comparable directory structure to a B-tree, but is optimized for sequential disk access, with nodes 100% full, and sequences of single-page nodes on each level below the root packed together in contiguous multi-page disk *blocks* for efficient arm use; this optimization was also used in the SB-tree [21]. Multi-page block I/O is used during the rolling merge and for long range retrievals, while single-page nodes are used for matching indexed finds to minimize buffering requirements. Multi-page block sizes of 256 KBytes are envisioned to contain nodes below the root; the root node is always a single page by definition.

The rolling merge acts in a series of *merge steps*. A read of a multi-page block containing leaf nodes of the C_1 tree makes a range of entries in C_1 buffer resident. Each merge step then reads a disk page sized leaf node of the C_1 tree buffered in this block, merges entries from the leaf node

with entries taken from the leaf level of the C_0 tree, thus decreasing the size of C_0, and creates a newly merged leaf node of the C_1 tree.

The buffered multi-page block containing old C_1 tree nodes prior to merge is called the *emptying* block, and new leaf nodes are written to a different buffered multi-page block called the *filling block*. When this filling block has been packed full with newly merged leaf nodes of C_1, the block is written to a *new* free area on disk. The new multi-page block containing merged results is pictured in Figure 2.2 as lying on the right of the former nodes. Subsequent merge steps bring together increasing index value segments of the C_0 and C_1 components until the maximum values are reached and the rolling merge starts again from the smallest values.

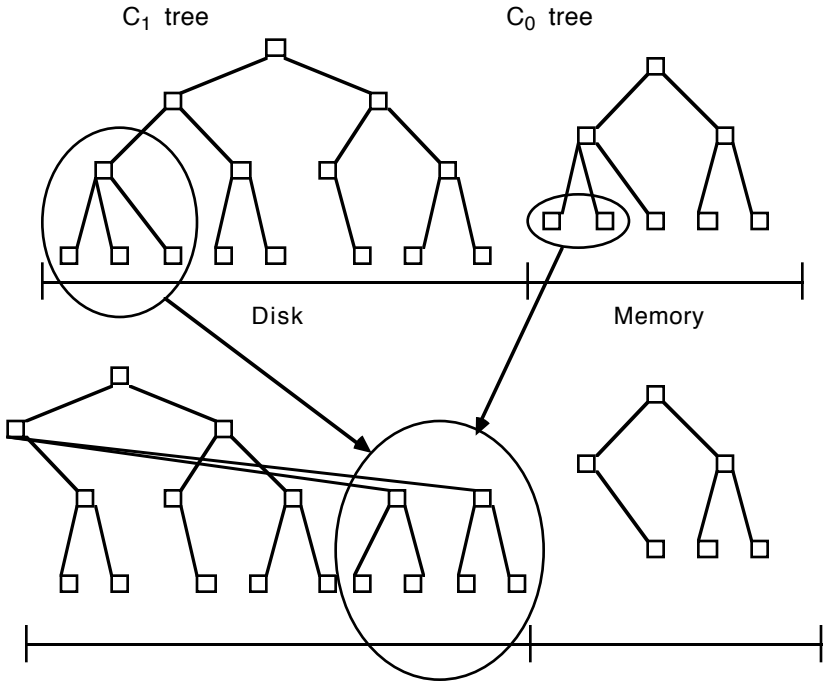

Figure 2.2. Conceptual picture of rolling merge steps, with result written back to disk

Newly merged blocks are written to new disk positions, so that the old blocks will not be overwritten and will be available for recovery in case of a crash. The parent directory nodes in C_1, also buffered in memory, are updated to reflect this new leaf structure, but usually remain in buffer for longer periods to minimize I/O; the old leaf nodes from the C_1 component are invalidated after the merge step is complete and are then deleted from the C_1 directory. In general, there will be leftover leaf-level entries for the merged C_1 component following each merge step, since a merge step is unlikely to result in a new node just as the old leaf node empties. The same consideration holds for multi-page blocks, since in general when the filling block has filled with newly merged nodes, there will be numerous nodes containing entries still in the shrinking block. These leftover entries, as well as updated directory node information, remain in block memory buffers for a time without being written to disk. Techniques to provide concurrency during the merge step and recovery from lost memory during a crash are covered in detail in Section 4. To reduce reconstruction time in recovery, checkpoints of the merge process are taken periodically, forcing all buffered information to disk.

2.1 How a Two Component LSM-tree Grows

To trace the metamorphosis of an LSM-tree from the beginning of its growth, let us begin with a first insertion to the C_0 tree component in memory. Unlike the C_1 tree, the C_0 tree is not expected to have a B-tree-like structure. For one thing, the nodes could be any size: there is no need to insist on disk page size nodes since the C_0 tree never sits on disk, and so we need not sacrifice CPU efficiency to minimize depth. Thus a (2-3) tree or AVL-tree (as explained, for example, in [1]) are possible alternative structures for a C_0 tree. When the growing C_0 tree first reaches its threshold size, a leftmost sequence of entries is deleted from the C_0 tree (this should be done in an efficient batch manner rather than one entry at a time) and reorganized into a C_1 tree leaf node packed 100% full. Successive leaf nodes are placed left-to-right in the initial pages of a buffer resident multi-page block until the block is full; then this block is written out to disk to become the first part of the C_1 tree disk-resident leaf level. A directory node structure for the C_1 tree is created in memory buffers as successive leaf nodes are added, with details explained below.

Successive multi-page blocks of the C_1 tree leaf level in ever increasing key-sequence order are written out to disk to keep the C_0 tree threshold size from exceeding its threshold. Upper level C_1 tree directory nodes are maintained in separate multi-page block buffers, or else in single page buffers, whichever makes more sense from a standpoint of total memory and disk arm cost; entries in these directory nodes contain separators that channel access to individual single-page nodes below, as in a B-tree. The intention is to provide efficient exact-match access along a path of single page index nodes down to the leaf level, avoiding multi-page block reads in such a case to minimize memory buffer requirements. Thus we read and write multi-page blocks for the rolling merge or for long range retrievals, and single-page nodes for indexed find (exact-match) access. A somewhat different architecture that supports such a dichotomy is presented in [21]. Partially full multi-page blocks of C_1 directory nodes are usually allowed to remain in buffer while a sequence of leaf node blocks are written out. C_1 directory nodes are forced to new positions on disk when:

- A multi-page block buffer containing directory nodes becomes full
- The root node splits, increasing the depth of the C_1 tree (to a depth greater than two)
- A checkpoint is performed

In the first case, the single multi-page block which has filled is written out to disk. In the latter two cases, all multi-page block buffers and directory node buffers are flushed to disk.

After the rightmost leaf entry of the C_0 tree is written out to the C_1 tree for the first time, the process starts over on the left end of the two trees, except that now and with successive passes multi-page leaf-level blocks of the C_1 tree must be read into buffer and merged with the entries in the C_0 tree, thus creating new multi-page leaf blocks of C_1 to be written to disk.

Once the merge starts, the situation is more complex. We picture the rolling merge process in a two component LSM-tree as having a conceptual cursor which slowly circulates in quantized steps through equal key values of the C_0 tree and C_1 tree components, drawing indexing data out from the C_0 tree to the C_1 tree on disk. The rolling merge cursor has a position at the leaf level of the C_1 tree and within each higher directory level as well. At each level, all currently merging multi-page blocks of the C_1 tree will in general be split into two blocks: the "emptying" block whose entries have been depleted but which retains information not yet reached by the merge cursor, and the "filling" block which reflects the result of the merge up to this moment. There will be an analogous "filling node" and "emptying node" defining the cursor which will certainly be buffer resident. For concurrent access purposes, both the emptying

block and the filling block on each level contain an integral number of page-sized nodes of the C_1 tree, which simply happen to be buffer resident. (During the merge step that restructures individual nodes, other types of concurrent access to the entries on those nodes are blocked.) Whenever a complete flush of all buffered nodes to disk is required, all buffered information at each level must be written to new positions on disk (with positions reflected in superior directory information, and a sequential log entry for recovery purposes). At a later point, when the filling block in buffer on some level of the C_1 tree fills and must be flushed again, it goes to a new position. Old information that might still be needed during recovery is never overwritten on disk, only invalidated as new writes succeed with more up-to-date information. A somewhat more detailed explanation of the rolling merge process is presented in Section 4, where concurrency and recovery designs are considered.

It is an important efficiency consideration of the LSM-tree that when the rolling merge process on a particular level of the C_1 tree passes through nodes at a relatively high rate, all reads and writes are in multi-page blocks. By eliminating seek time and rotational latency, we expect to gain a large advantage over random page I/O involved in normal B-tree entry insertion. (This advantage is analyzed below, in Section 3.2.) The idea of always writing multi-page blocks to new locations was inspired by the Log-Structured File System devised by Rosenblum and Ousterhout [23], from which the Log-Structured Merge-tree takes its name. Note that the continuous use of new disk space for fresh multi-page block writes implies that the area of disk being written will wrap, and old discarded blocks must be reused. This bookkeeping can be done in a memory table; old multi-page blocks are invalidated and reused as single units, and recovery is guaranteed by the checkpoint. In the Log-Structured File System, the reuse of old blocks involves significant I/O because blocks are typically only partially freed up, so reuse requires a block read and block write. In the LSM-Tree, blocks are totally freed up on the trailing edge of the rolling merge, so no extra I/O is involved.

2.2 Finds in the LSM-tree Index

When an exact-match find or range find requiring immediate response is performed through the LSM-tree index, first the C_0 tree and then the C_1 tree is searched for the value or values desired. This may imply a slight CPU overhead compared to the B-tree case, since two directories may need to be searched. In LSM-trees with more than two components, there may also be an I/O overhead. To anticipate Chapter 3 somewhat, we define a multi component LSM-tree as having components C_0, C_1, C_2, . . ., C_{K-1} and C_K, indexed tree structures of increasing size, where C_0 is memory resident and all other components are disk resident. There are asynchronous rolling merge processes in train between all component pairs (C_{i-1}, C_i) that move entries out from the smaller to the larger component each time the smaller component, C_{i-1}, exceeds its threshold size. As a rule, in order to guarantee that all entries in the LSM-tree have been examined, it is necessary for an exact-match find or range find to access each component C_i through its index structure. However, there are a number of possible optimizations where this search can be limited to an initial subset of the components.

First, where unique index values are guaranteed by the logic of generation, as when timestamps are guaranteed to be distinct, a matching indexed find is complete if it locates the desired value in an early C_i component. As another example, we could limit our search when the find criterion uses recent timestamp values so that the entries sought could not yet have migrated out to the largest components. As the merge cursor circulates through the (C_i, C_{i+1}) pairs, we will often have reason to retain entries in C_i that have been inserted in the recent past (in the last τ_i seconds), allowing only the older entries to go out to C_{i+1}. In cases where the most frequent find references are to recently inserted values, many finds can be completed in the C_0 tree, and so the C_0 tree fulfills a valuable memory buffering function. This point was made also

in [23], and represents an important efficiency consideration. For example, indexes to short-term transaction UNDO logs accessed in the event of an abort will have a large proportion of accesses in a relatively short time-span after creation, and we can expect most of these indexes to remain memory resident. By keeping track of the start-time for each transaction we can guarantee that all logs for a transaction started in the last τ_0 seconds, for example, will be found in component C_0, without recourse to disk components.

2.3 Deletes, Updates and Long-Latency Finds in the LSM-tree

We note that deletes can share with inserts the valuable properties of deferral and batching. When an indexed row is deleted, if a key value entry is not found in the appropriate position in the C_0 tree, a *delete node entry* can be placed in that position, also indexed by the key value, but noting an entry Row ID (RID) to delete. The actual delete can be done at a later time during the rolling merge process, when the actual index entry is encountered: we say the delete node entry *migrates out* to larger components during merge and *annihilates* the associated entry when it is encountered. In the meantime, find requests must be filtered through delete node entries so as to avoid returning references to deleted records. This filtering is easily performed during the search for the relevant keyvalue, since the delete node entry will be located in the appropriate keyvalue position of an earlier component than the entry itself, and in many cases this filter will reduce the overhead of determining an entry is deleted. Updates of records that cause changes to indexed values are unusual in any kind of applications, but such updates can be handled by LSM-trees in a deferred manner if we view an update as a delete followed by an insert.

We sketch another type of operation for efficient index modification. A process known as *predicate deletion* provides a means of performing batch deletes by simply *asserting* a predicate, for example the predicate that all index values with timestamps more than 20 days old are to be deleted. When the affected entries in the oldest (largest) component become resident during the normal course of the rolling merge, this assertion causes them simply to be dropped during the merge process. Yet another type of operation, a long-latency find, provide an efficient means of responding to a query where the results can wait for the circulation period of the slowest cursor. A *find note entry* is inserted in component C_0, and the find is actually performed over an extended period of time as it migrates out to later components. Once the find note entry has circulated out to the appropriate region of the largest relevant component of the LSM-tree, the accumulated list of RIDs for the long-latency find is complete.

3. Cost-Performance and the Multi-Component LSM-Tree

In this section we analyze the cost-performance of an LSM-tree, starting with an LSM-tree of two components. We analyze the LSM-tree by analogy with a B-tree providing the same indexing capabilities, comparing the I/O resources utilized for a high volume of new insertions. As we will argue in Section 5, other disk-based access methods are comparable to the B-tree in I/O cost for inserts of new index entries. The most important reason for the comparison of the LSM-tree and B-tree that we perform here is that these two structures are easily comparable, both containing an entry for each row indexed in collation sequence at a leaf level, with upper level directory information that channels access along a path of page-sized nodes. The analysis of I/O advantage for new entry inserts to the LSM-tree is effectively illustrated by analogy to the less efficient but well understood behavior of the B-tree.

In Section 3.2 following, we compare the I/O insert costs and demonstrate that the small ratio of cost for an LSM-tree of two components to that of a B-tree is a product of two factors. The first factor, $COST_\pi/COST_P$, corresponds to the advantage gained in the LSM-tree by performing all I/O in multi-page blocks, thus utilizing disk arms much more efficiently by saving a great deal

of seek and rotational latency time. The $COST_\pi$ term represents the disk arm cost of reading or writing a page on disk as part of a multi-page block, and $COST_P$ represents the cost of reading or writing a page at random. The second factor that determines I/O cost ratio between the LSM-tree and the B-tree is given as 1/M, representing the batching efficiency to be gained during a merge step. M is the average number of entries merged from C_0 into a page-sized leaf node of C_1. Inserting multiple entries per leaf is an advantage over a (large) B-tree where each entry inserted normally requires two I/Os to read and write the leaf node on which it resides. Because of the Five minute rule, it is unlikely in Example 1.2 that a leaf page read in from a B-tree will be re-referenced for a second insert during the short time it remains in buffer. Thus there is no batching effect in a B-tree index: each leaf node is read in, an insert of a new entry is performed, and it is written out again. In an LSM-tree however, there will be an important batching effect as long as the C_0 component is sufficiently large in comparison to the C_1 component. For example, with 16 byte index entries, we can expect 250 entries in a fully packed 4 KByte node. If the C_0 component is 1/25 the size of the C_1 component, we will expect (about) 10 new entries entering each new C_1 node of 250 entries during a node I/O. It is clear that the LSM-tree has an efficiency advantage over the B-tree because of these two factors, and the "rolling merge" process is fundamental to gaining this advantage.

The factor $COST_\pi/COST_P$ corresponding to the ratio of efficiency of multi-page block over single page I/O is a constant, and we can do nothing with the LSM-tree structure to have any effect on it. However the batching efficiency 1/M of a merge step is proportional to the ratio in size between the C_0 and the C_1 components; the larger the C_0 component in comparison to the C_1 component, the more efficiency is gained in the merge; up to a certain point, this means that we can save additional money on disk arm cost by using a larger C_0 component, but this entails a larger memory cost to contain the C_0 component. There is an optimal mix of sizes to minimize the total cost of disk arms and memory capacity, but the solution can be quite expensive in terms of memory for a large C_0. It is this consideration that motivates the need for a multi-component LSM-tree, which is investigated in Section 3.3. A three component LSM-tree has memory resident component C_0 and disk resident components C_1 and C_2, where the components increase in size with increasing subscript. There is a rolling merge processes in train between C_0 and C_1 as well as a separate rolling merge between C_1 and C_2 that move entries out from the smaller to the larger component each time the smaller component exceeds its threshold size. The advantage of an LSM-tree of three components is that batching efficiency can be geometrically improved by choosing C_1 to optimize the combined ratio of size between C_0 and C_1 and between C_1 and C_2. As a result, the size of the C_0 memory component can be made much smaller in proportion to the total index, with a significant improvement in cost.

Section 3.4 derives a mathematical procedure for arriving at the optimal relative sizes of the different components of a multi-component LSM-tree to minimize total cost for memory and disk.

3.1 The Disk Model

The advantage of the LSM-tree over the B-tree lies mainly in the area of reduced cost for I/O (although disk components that are 100% full offer a capacity cost advantage as well over other known flexible disk structures). Part of this I/O cost advantage for the LSM-tree is the fact that a page I/O can be amortized along with many other pages of a multi-page block.

Definition 3.1.1. I/O Costs and Data Temperature. As we store data of a particular kind on disk, rows in a table or entries in an index, we find that as we increase the amount of data stored, the disk arms see more and more utilization under normal use in a given application environment. We are paying for two things when we buy a disk: first, disk capacity, and sec-

ond, disk I/O rate. Usually one of these two will be a limiting factor in any kind of use. If capacity is the limiting factor, we will fill up the disks and find that the disk arms that provide the I/Os are only fractionally utilized by the application; on the other hand we may find that as we add data the disk arms reach their full utilization rate when the disk is only fractionally full, and this means that the I/O rate is the limiting factor.

A random page I/O during peak use has a *cost*, $COST_P$, which is based on a fair rent for the disk arm, whereas the cost of a disk page I/O as part of a large multi-page block I/O will be represented as $COST_\pi$, and this quantity is a good deal smaller because it amortizes seek time and rotational latency over multiple pages. We adopt the following nomenclature for storage costs:

$COST_d$ = cost of 1 MByte of disk storage
$COST_m$ = cost of 1 MByte of memory storage
$COST_P$ = disk arm cost to provide 1 page/second I/O rate, for random pages
$COST_\pi$ = disk arm cost to provide 1 page/second I/O rate, as part of multi-page block I/O

Given an application referencing a body of data with S MBytes of storage and H random pages per second of I/O transfer (we assume no data is buffered), the rent for disk arms is given by $H \cdot COST_P$ and the rent for disk media is given by $S \cdot COST_d$. Depending on which cost is the limiting factor the other comes for free, so the calculated cost for accessing this disk resident data, COST-D, is given by:

$$\text{COST-D} = \max(S \cdot COST_d, H \cdot COST_P)$$

COST-D will also be the total cost for supporting data access for this application, COST-TOT, under the assumption given that none of the disk pages are buffered in memory. In this case, the total cost increases linearly with the random I/O rate H even while the total storage requirement S remains constant. Now the point of memory buffering is to replace disk I/O with memory buffers at a certain point of increasing I/O rate to the same total storage S. If we assume under these circumstances that memory buffers can be populated in advance to support the random I/O requests, the cost for disk drops to the cost for disk media alone, so the calculated cost of accessing this buffer resident data, COST-B, is simply the cost of memory plus the cost of disk media:

$$\text{COST-B} = S \cdot COST_m + S \cdot COST_d$$

Now the total cost for supporting data access for this application is the minimum of these two calculated costs:

$$\text{COST-TOT} = \min(\max(S \cdot COST_d, H \cdot COST_P), S \cdot COST_m + S \cdot COST_d)$$

There are three cost regimes in the graph of COST-TOT as the page access rate H increases for a given volume of data S. See Figure 3.1, where we graph COST-TOT/MByte vs H/S, or accesses per second per megabyte. If S is small, COST-TOT is limited by the cost of disk medium, $S \cdot COST_d$, a constant for fixed S. As H/S increases, the cost comes to be dominated by disk arm use, $H \cdot COST_P$, and is proportional to increasing H/S for fixed S. Finally, at the point where the Five Minute rule dictates memory residence, the dominant factor becomes $S \cdot COST_m + S \cdot COST_d$, which is dominated by the memory term for present prices, $COST_m >> COST_d$. Following Copeland et al. [6], we define the *temperature* of a body of data as H/S, and we name these three cost regimes *cold*, *warm*, and *hot*. Hot data has a high enough access rate H, and thus temperature H/S, to justify memory buffer residence (see [6]). At the other extreme, cold data is disk

capacity limited: the disk volume that it must occupy comes with enough disk arms to satisfy the I/O rate. In between is warm data, whose access requirements must be met by limiting the data capacity used under each disk arm, so that disk arms are the limit of use. These ranges are divided as follows:

$T_f = COST_d/COST_P$ = temperature division point between cold and warm data ("freezing")
$T_b = COST_m/COST_P$ = temperature division point between warm and hot data ("boiling")

Similarly-defined ranges exist for the multi-page block access case using $COST_\pi$. The division between the warm and hot regions is a generalization of the Five Minute Rule [13].

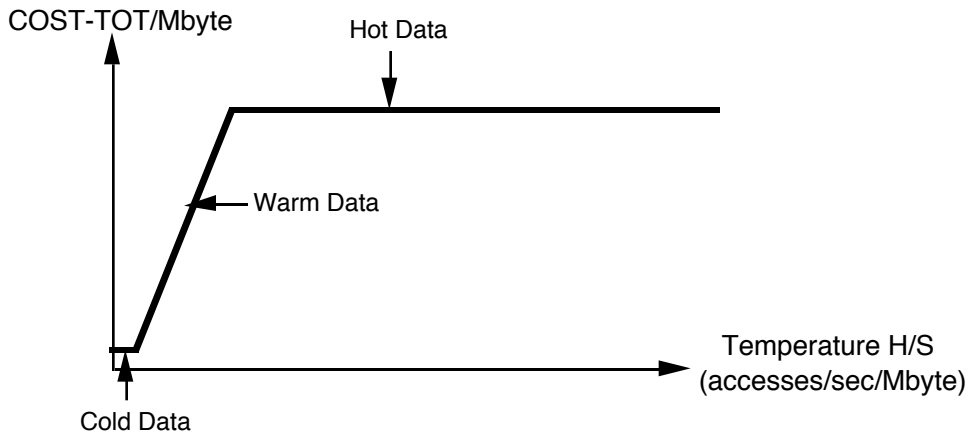

Figure 3.1. Graph of cost of access per MByte vs. Temperature

As stressed in [6], it is straightforward to calculate the temperature of a database table when it is accessed uniformly. However, the relevance of this temperature depends on the access method: the temperature that is relevant involves the actual disk access rate, not the logical insert rate (including batched buffered inserts). One way to express what an LSM-tree achieves is to say that it reduces the actual disk accesses and thus lowers the effective temperature of the indexed data. This idea is revisited in the conclusions of Section 6.

Multi-page block I/O Advantage

The advantage to be gained by multi-page block I/O is central to several earlier access methods, such as Bounded Disorder files [16], SB-trees [21], and Log Structured files [23]. A 1989 IBM publication analyzing DB2 utility performance on IBM 3380 disk [29] gave the following analysis: ". . . The time to complete a [read of a single page] could be estimated to be about 20 ms (assumes 10ms seek, 8.3ms rotational delay, 1.7ms read) . . . The time to perform a sequential prefetch read [of 64 contiguous pages] could be estimated to be about 125ms (assumes 10ms seek, 8.3ms rotational delay, 106.9ms read of 64 records [pages]), or about 2 ms per page." Thus the ratio of 2 ms per page for multi-page block I/O to 20 ms for random I/O implies a ratio of rental costs for the disk arm, $COST_\pi/COST_P$, equal to about 1/10. An analysis of a more recent SCSI-2 disk read of a 4 KByte page gives us a 9 ms seek, 5.5 ms rotational delay, and 1.2 ms read, totalling 16 ms. Reading 64 contiguous 4 KByte pages requires a 9 ms seek, 5.5 ms rotational delay, and 80 ms read for 64 pages, or a total of 95 ms, about 1.5 ms/page. Once again $COST_\pi/COST_P$ is again equal to about 1/10.

We analyze a workstation server system with SCSI-2 disks holding one GByte and costing about $1000, and a peak rate of approximately 60-70 I/Os per second. The nominal usable I/O rate to avoid long I/O queues is lower, about 40 I/Os per second. The multi-block I/O advantage is significant.

Typical Workstation Costs, 1995:

$COST_m$ = $100/MByte
$COST_d$ = $1/MByte
$COST_P$ = $25/(IOs/sec)
$COST_\pi$ = $2.5/(IOs/sec)
$T_f = COST_d/COST_P$ = .04 IOs/(sec·MByte) ("freezing point")
$T_b = COST_m/COST_P$ = 4 IOs/(sec·MByte) ("boiling point")

We use the T_b value to derive the reference interval τ for the Five Minute Rule, which asserts that data sustaining an I/O rate of one page every seconds is incurring the same cost as the memory needed to hold it. That common cost is:

$$(1/\tau)\cdot COST_P = pagesize\cdot COST_m$$

Solving for τ, we see $\tau = (1/pagesize)\cdot(COST_P/COST_m) = 1/(pagesize\cdot T_b)$, and for the values given above, with a page of .004 MBytes, we have $\tau = 1/(.004\cdot 4)$ 62.5 seconds/IO.

Example 3.1. To achieve a rate of 1000 TPS in the TPC-A application of Example 1.1, there will be H = 2000 I/Os per second to the Account table, itself consisting of 100,000,000 rows of 100 bytes, a total of S =10 GBytes. The disk storage cost here is $S\cdot COST_d$= $10,000 whereas the disk I/O cost is $H\cdot COST_P$ = $50,000. The temperature T = H/S = 2000/10,000 = 0.2, well above freezing (a factor of 5), but also well below the boiling point. This warm data uses only 1/5 of its disk capacity for data storage. We are paying for the disk arms and not for the capacity. The situation is similar when we consider the 20 day Acct-ID||Timestamp index to the History table of Example 1.2. Such a B-tree index, as we calculated in Example 1.2, requires about 9.2 GBytes of leaf-level entries. Given that a growing tree is only about 70% full, the entire tree will require 13.8 GBytes, but it has the same I/O rate (for inserts alone) as the Account table, which implies a comparable temperature.

3.2 Comparison of LSM-tree and B-tree I/O costs

We will be considering I/O costs of index operations which we call *mergeable:* inserts, deletes, updates, and long-latency finds. The following discussion presents an analysis to compare an LSM-tree to a B-tree.

B-tree Insert Cost Formula.

Consider the disk arm rental cost of performing a B-tree insert. We must first access the position in the tree where the entry should be placed, and this entails a search down nodes of the tree. We assume that successive inserts to the tree are to *random* positions at the leaf level, so that node pages in the path of access will not be consistently buffer resident because of past inserts. A succession of inserts of ever increasing key-values, an *insert-on-the-right* situation, is a relatively common case that does not obey this assumption. We note that such an insert-on-the-right situation can already be quite efficiently handled by the B-tree data structure, since there is little I/O as the B-tree grows consistently to the right; indeed this is

the basic situation in which a B-tree load takes place. There are a number of other proposed structures to deal with indexing log records by ever-increasing value [8].

In [21], the *effective depth* of a B-tree, symbolized by D_e, was defined to be the average number of pages not found in buffer during a random key-value search down the directory levels of a B-tree. For B-trees of the size used to index Account-ID||Timestamp in Example 1.2, the value for D_e is typically about 2.

To perform an insert to a B-tree, we perform a key-value search to a leaf level page (D_e I/Os), update it, and (in the steady state) write out a corresponding dirty leaf page (1 I/O). We can show that the relatively infrequent node splits have an insignificant effect on our analysis, and therefore ignore them. The pages read and written in this process are all random access, with cost $COST_P$, so the total I/O cost for a B-tree insert, $COST_{B\text{-}ins}$ is given by:

(3.1) $COST_{B\text{-}ins} = COST_P \cdot (D_e + 1)$

LSM-tree Insert Cost Formula.

To evaluate the cost of an insert into the LSM-tree, we need to think in terms of amortization of multiple inserts, since a single insert to the memory component C_0 only occasionally has any I/O effect. As we explained at the beginning of this Section, the performance advantage an LSM-tree has over a B-tree is based on two different *batching* effects. The first is the already mentioned reduced cost of a page I/O, $COST_\pi$. The second is based on the idea that the delay in merging newly inserted entries into the C_1 tree usually allows time for numerous entries to accumulate in C_0; thus several entries will get merged into each C_1 tree leaf page during its trip from disk to memory and back. By contrast, we have been assuming that the B-tree leaf pages are too infrequently referenced in memory for more than one entry insert to take place.

Definition 3.2.1. The Batch-Merge Parameter M. To quantify this multiple-entries-per-leaf batching effect, define the parameter M for a given LSM-tree as the average number of entries in the C_0 tree inserted into *each* single page leaf node of the C_1 tree during the rolling merge. We assert that the parameter M is a relatively stable value characterizing an LSM-tree. In fact, the value for M is determined by index entry size and the ratio in size between the leaf level of the C_1 tree and that of the C_0 tree. We define the following new size parameters:

S_e = entry (index entry) size in bytes
S_p = page size in bytes
S_0 = size in MBytes of C_0 component leaf level
S_1 = size in MBytes of C_1 component leaf level.

Then the number of entries to a page is approximately S_p/S_e, and the fraction of entries of the LSM-tree sitting in component C_0 is $S_0/(S_0 + S_1)$, so the parameter M is given by:

(3.2) $M = (S_p/S_e) \cdot (S_0/(S_0 + S_1))$

Note that the larger the component C_0 in comparison to C_1, the larger will be the parameter M. Typical implementations might have $S_1 = 40 \cdot S_0$ and the number of entries per disk page, S_p/S_e, of 200, so that M = 5. Given the parameter M, we can now give a rough formula for the cost $COST_{LSM\text{-}ins}$ of an entry insert into the LSM-tree. We simply amortize the per-page cost of bringing the C_1 tree leaf node into memory and writing it out again, $2 \cdot COST_\pi$, over the M inserts that are merged into an C_1 tree leaf node during this time.

(3.3) $COST_{LSM\text{-}ins} = 2 \cdot COST_{\pi}/M$

Note that we have ignored the relatively insignificant costs associated with I/Os for index updates in both the LSM-tree and B-tree cases.

A Comparison of LSM-tree and B-tree Insert Costs

If we compare the cost formulas (3.1) and (3.3) for inserts to the two data structures, we see the ratio:

(3.4) $COST_{LSM\text{-}ins}/COST_{B\text{-}ins} = K_1 \cdot (COST_{\pi}/COST_P) \cdot (1/M)$

where K_1 is a (near) constant, $2/(D_e + 1)$, with a value of approximately 0.67 for index sizes we have been considering. This formula shows that the cost ratio of an insert into the LSM-tree to one in the B-tree is directly proportional to each of two batching effects we have discussed: $COST_{\pi}/COST_P$, a small fraction corresponding to the ratio of cost for a page I/O in a multi-page block to a random page I/O, and 1/M, where M is the number of entries batched per page during the rolling merge. Typically the product of the two ratios will give a cost ratio improvement of nearly two orders of magnitude. Naturally, such improvement will only be possible in regimes where the index has a relatively high temperature as a B-tree, so that it is possible to greatly reduce the number of disks when moving to an LSM-tree index.

Example 3.2. If we assume that an index of the kind in Example 1.2 takes up 1 GByte of disk space but is required to sit on 10 GBytes to achieve necessary disk arm access rates, then there is certainly room for improvement in saving money on disk arm costs. If the ratio of insert costs given in Equation (3.4) is 0.02 = 1/50, then the we can shrink the index and disk cost: the LSM-tree will need to take up only 0.7 GBytes on disk because of closely packed entries and reduced disk arm utilization. However, we see that *the more efficient LSM-tree can only reduce cost down to what is needed for disk capacity*. If we had started with a 1 GByte B-tree which was constrained to sit on 35 GBytes to receive needed disk arm service, the ratio of cost improvement of 1/50 could have been fully realized.

3.3 Multi-Component LSM-trees

The parameter M for a given LSM-tree was defined as the average number of entries in the C_0 tree inserted into each single page leaf node of the C_1 tree during the rolling merge. We have been thinking of the quantity M as being greater than 1 because of the delay period during which new entries can accumulate in the C_0 tree before being merged into nodes of the C_1 tree. However, it should be clear from equation (3.2) that if the C_1 tree were extremely large in comparison to the C_0 tree, or entries were extremely large and fit only a small number to a page, the quantity M might be *less than* 1. Such a value for M means that on the average more than one C_1 tree page must be brought in and out of memory for each entry which is merged in from the C_0 tree. In the case where M is extremely small in terms of formula (3.4), specifically if $M < K_1 \cdot COST_{\pi}/COST_P$, this could even cancel the batching effect of multi-page disk reads, so we would do better to use a normal B-tree for inserts in place of an LSM-tree.

To avoid a small value for M the only course with a two-component LSM-tree is to increase the size of the C_0 component relative to that of C_1. Consider a two-component LSM-tree of given total leaf entry size S ($S = S_0 + S_1$, an approximately stable value), and assume we have a constant rate R in bytes per second of new entry inserts into C_0. For simplicity, we assume that

no entries inserted into C_0 are deleted before they get out to component C_1, and therefore entries must migrate out to component C_1 through the rolling merge at the same rate that they are inserted into C_0 to keep the size of C_0 near its threshold size. (Given that the total size S is approximately stable, this also implies that the insertion rate into C_0 must be balanced by a constant deletion rate from C_1, possibly using a succession of *predicate deletes*.) As we vary the size of C_0, we affect the circulation speed of the merge cursor. A constant migration rate out to C_1 in bytes per second requires that the rolling merge cursor move through entries of C_0 at a constant rate in bytes per second, and therefore as the size of C_0 decreases the circulation rate from smallest to largest index values in C_0 will increase; as a result, the I/O rate for multi-page blocks in C_1 to perform the rolling merge must also increase. If a C_0 size of a single entry were possible, at this conceptual extreme point we would require a circulation through all multi-page blocks of C_1 for each newly inserted entry, an immense demand on I/O. The approach of merging C_0 and C_1, rather than accessing relevant nodes of C_1 for each newly inserted entry as is done with the B-tree, would become a millstone around our necks. By comparison, larger size C_0 components will slow down the circulation of the merge cursor and decrease the I/O cost of inserts. However, this will increase the cost of the memory-resident component C_0.

Now there is a canonical size for C_0 determined by the point at which the total cost of the LSM-tree, memory cost for C_0 plus media/disk arm cost for the C_1 component, is minimized. To arrive at this balance, we start with a large C_0 component and pack the C_1 component closely on disk media. If the C_0 component is sufficiently large, we will have a very small I/O rate to C_1. We can now decrease the size of C_0, trading off expensive memory for inexpensive disk space, until the I/O rate to service C_1 increases to a point where the disk arms sitting over the C_1 component media are running at full rate. At this point, further savings in memory cost for C_0 will result in increased media cost, as we are required to spread out the C_1 component over fractionally full disks to reduce the disk arm load, and at some point as we continue to shrink C_0 we will reach a minimum cost point. Now it is common in the two component LSM-tree that the canonical size we determine for C_0 will still be quite expensive in terms of memory use. An alternative is to consider adopting an LSM-tree of three or more components. Conceptually, if the size of the C_0 component is so large that the memory cost is a significant factor, then we consider creating another intermediate size disk based component between the two extremes. This will permit us to limit the cost of disk arms while reducing the size of the C_0 component.

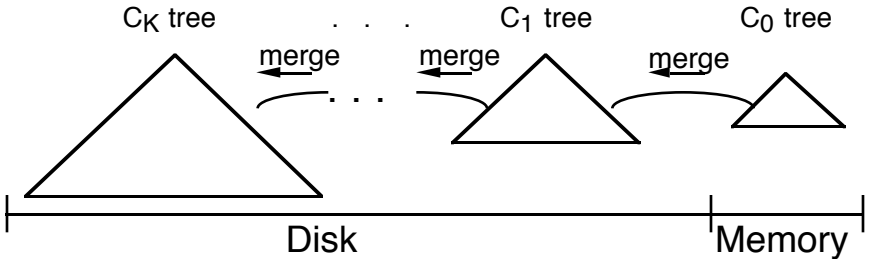

Figure 3.1. An LSM-tree of K+1 components

In general, an LSM-tree of K+1 components has components C_0, C_1, C_2, . . ., C_{K-1} and C_K, which are indexed tree structures of increasing size; the C_0 component tree is memory resident and all other components are disk resident (but with popular pages buffered in memory as with any disk resident access tree). Under pressure from inserts, there are asynchronous rolling merge processes in train between all component pairs (C_{i-1}, C_i), that move entries out from the smaller to the larger component each time the smaller component, C_{i-1}, exceeds its threshold

size. During the life of a long-lived entry inserted in an LSM-tree, it starts in the C_0 tree and eventually migrates out to the C_K, through a series of K asynchronous rolling merge steps.

The spotlight here is on performance under insert traffic because we are assuming that the LSM-Tree exists in an insert-mostly environment. LSM-tree finds of three or more components suffer somewhat in performance, typically by one extra page I/O per disk component.

3.4 LSM-trees: Component Sizes

In the current Section, we derive a formula for the I/O cost for inserts into an LSM-tree of several components and demonstrate mathematically how to choose optimal threshold sizes for the various components. An extended Example 3.3 illustrates the system cost for a B-tree, the improved system cost for an LSM-tree of two components, and the greater savings to be had with an LSM-tree of three components.

We define the *size* of an LSM-tree component, $S(C_i)$, as the number of bytes of entries it contains at the leaf level; the size of component C_i is denoted by S_i, $S(C_i) = S_i$, and S is the total size of all leaf level entries in all components, $S = \sum_i S_i$. We assume there is some relatively steady rate R of insertion, in bytes per second, to component C_0 of the LSM-tree, and for simplicity that all newly inserted entries live to circulate out to component C_K by a succession of rolling merge steps. We also assume that each of the components, $C_0, C_1, \ldots, C_{K-1}$, has a size close to a maximum threshold size to be determined by the current analysis. The component C_K is assumed to have a relatively stable size, because of deletes balancing inserts over some standard time period. The deletes from component C_K can be thought of as taking place without any addition to the rate of insertion R to component C_0.

Given an LSM-tree of K components with a fixed total size S and memory component size S_0, the tree is totally described by the variables r_i, $i = 1, \ldots, K$, representing size ratios between adjacent pairs of components, $r_i = S_i/S_{i-1}$, As detailed below, the total page I/O rate to perform all ongoing merge operations between component pairs (C_{i-1}, C_i) can be expressed as a function of R, the rate of insertions into C_0, and the ratios r_i. We assume that blocks of the different components are striped across different disk arms in a mixed way to achieve a balance in utilization, so that minimizing H is the same as minimizing the total disk arm cost (at least in any range where disk arms rather than media capacity constitute the gating cost). It is a standard calculus minimization problem to find the values for r_i which minimize the total I/O rate H for a given R. It turns out that the assumption that the total size S is fixed leads to a rather difficult problem with a somewhat complex recurrence relation between the r_i values. However, if we make the comparable assumption that the largest component size S_K is fixed (along with the memory size S_0), as we will show in Theorem 3.1, this minimization problem is solved when all of the values r_i are equal to a single constant value r. We show in Theorem 3.2 the slightly more precise solution relating the r_i values where the total size S is held constant, and argue that the constant value r for r_i gives similar results in all areas of real interest. Assuming such a constant value r for all r_i factors, we have $S_i = r^i \cdot S_0$. Thus the total size S is given by the sum of the individual component sizes, $S = S_0 + r \cdot S_0 + r^2 \cdot S_0 + \ldots + r^K \cdot S_0$, and we can solve for r in terms of S and S_0.

Thus in Theorem 3.1 we show that to minimize the total I/O rate H of a multi-component LSM-tree, with fixed S_K, S_0, and insertion rate R, we size intermediate components in a geometric progression between the smallest and largest. We will see, as in the case of a two-component LSM-tree, that if we allow S_0 to vary while R and S_K remain constant, and express H as a

function of S_0, then H increases with decreasing S_0. We can now minimize the total cost of the LSM-tree, memory plus disk arm cost, by varying the size of S_0. The appropriate process to arrive at an optimal total cost for a given number of components is illustrated below in Example 3.3. The only remaining free variable in the total cost is the number of components, K+1. We discuss the tradeoffs for this value at the end of the current Section.

Theorem 3.1. Given an LSM-tree of K+1 components, with a fixed largest-component size S_K, insert rate R, and memory component size S_0, the total page I/O rate H to perform all merges is minimized when the ratios $r_i = S_i/S_{i-1}$ are all equal to a common value r. Thus the total size S is given by the sum of the individual component sizes,

$$(3.5)\quad S = S_0 + r \cdot S_0 + r^2 \cdot S_0 + \ldots + r^K \cdot S_0,$$

and we can solve for r in terms of S and S_0. Similarly, the total page I/O rate H is given by

$$(3.6)\quad H = (2R/S_p) \cdot (K \cdot (1 + r) - 1/2),$$

where S_p is the number of bytes per page.

Proof. Since we have assumed that entries are never deleted until they arrive at component C_K, it is clear in the steady state that the rate R in bytes per second of inserts to C_0 is the same as the rate with which entries migrate by rolling merge out from component C_{i-1} to component C_i, for all i, $0 < i \leq K$. Consider the case where the component C_{i-1} is disk resident. Then the merge from C_{i-1} to C_i entails multi-page block reads from component C_{i-1} at a rate of R/S_p pages per second, where S_p is the number of bytes per page (we derive this from the rate R in bytes per second that entries migrate out from C_{i-1}, assuming that 100% of all entries encountered are deleted from C_{i-1}; other assumptions are possible in a general case). The merge also entails multi-page reads from C_i at a rate $r_i \cdot R/S_p$ pages per second (this follows from the fact that the rolling merge cursor passes over $r_i = S_i/S_{i-1}$ times as many pages belonging to C_i as it does pages of C_{i-1}). Finally, the merge entails multi-page disk writes at a rate of (r_i+1) R/S_p pages per second to write out newly merged data belonging to C_i. Note that here we are taking into account the enlarged size of the C_i component resulting from the merge. Summing over all disk resident components C_i, we have a total rate H of multi-page I/Os in pages per second given by:

$$(3.7)\quad H = (R/S_p)\ ((2 \cdot r_1 + 2) + (2 \cdot r_2 + 2) + \ldots + (2 \cdot r_{K-1} + 2) + (2 \cdot r_K + 1)),$$

where each term of the form $(2 \cdot r_i + k)$ represents all I/O on component C_i: $r_i \cdot R/S_p$ to read in pages in C_i for the merge from C_{i-1} to C_i, $(r_i + 1)$ R/S_p to write out pages in C_i for that same merge, and R/S_p to read in pages in C_i for the merge from C_i to C_{i+1}. Clearly there is no term for C_0 and the term for component C_K does not have this final addition. Equation (3.7) can be rewritten as:

$$(3.8)\quad H = (2R/Sp)\ \left(\sum_{1}^{K} r_i + K - \frac{1}{2}\right)$$

We wish to minimize the value of this function under the condition that: $\prod_{1}^{K} r_i = (S_K/S_0) = C$, a constant. To solve this problem, we minimize $\sum_{1}^{K} r_i$, with the term r_K replaced by $C \cdot \prod_{1}^{K-1} r_i^{-1}$. Taking partial derivatives by each of the free variables r_j, j = 1, . . , K-1, and equating them to zero, we arrive at a set of identical equations of the form: $0 = 1 - \frac{1}{r_j} C \cdot \prod_{1}^{K-1} r_i^{-1}$, which is clearly solved when all r_j (including r_K) are equal to $C \cdot \prod_{1}^{K-1} r_i^{-1}$, or $C^{1/K}$.

Theorem 3.2. We vary the assumptions of Theorem 3.1 to fix the total size S rather than the size S_K of the largest component. This minimization problem is much more difficult, but can be done using Lagrange multipliers. The results are a sequence of formulas for each r_i in terms of higher-indexed r_i:

$$r_{K-1} = r_K + 1$$
$$r_{K-2} = r_{K-1} + 1/r_{K-1}$$
$$r_{K-3} = r_{K-2} + 1/(r_{K-1} \cdot r_{K-2})$$
. . .

We omit the proof.

As we will see, useful values of r_i are fairly large, say 20 or more, so the size of the largest component, S_K, dominates the total size S. Note that in Theorem 3.2 therefore, each r_i normally differs by only a small fraction from its higher neighbor r_{i+1}. In what follows, we base or examples on the approximation of Theorem 3.1.

Minimizing Total Cost

From Theorem 3.1, it can be seen that if we allow S_0 to vary while R and S_K remain constant and express the total I/O rate H as a function of S_0, then since r increases with decreasing S_0 by equation (3.5), and H is proportional to r by equation (3.6), clearly H increases with decreasing S_0. We can now minimize the total cost of the LSM-tree as in the two component case by trading off expensive memory for inexpensive disk. If we calculate the disk media needed to store the LSM-tree and the total I/O rate H that keeps these disk arms fully utilized, this becomes a starting point in our calculation to determine the size for S_0 that minimizes cost. From this point as we further decrease the size of C_0 the cost of disk media goes up in inverse proportion, since we have entered the region where disk arm cost is the limiting factor. Example 3.3, below, is a numerically based illustration of this process for a two and three component LSM-tree. Prior to this example, we offer an analytic derivation for the two component case.

The total cost is the sum of memory cost, $COST_m \cdot S_0$, and disk cost, itself a maximum over disk storage and I/O costs, here based on multi-page block access rate H in pages per second:

$$COST_{tot} = COST_m \cdot S_0 + \max[COST_d \cdot S_1, COST_\pi \cdot H]$$

Consider the case of two components, so that in equation (3.6), K = 1, $r = S_1/S_0$. Let

$s = (COST_m \cdot S_0)/(COST_d \cdot S_1)$ = cost of memory relative to storage cost for S_1 data.

$t = 2\cdot((R/S_p)/S_1)\cdot(COST_\pi/COST_d)(COST_m/COST_d)$

$C = COST_{tot}/(COST_d \cdot S_1)$ = total cost relative to storage cost for S_1 data

then, substituting equation (3.6) and simplifying, assuming S_0/S_1 small, we arrive at a close approximation:

$C \approx s + \max(1, t/s)$

The relative cost C is a function of two variables t and s; the variable t is a kind of normalized temperature measuring the basic multi-page block I/O rate required by the application. The variable s represents how much memory we decide to use to implement the LSM-tree. To decide the size of S_0, the simplest rule would be to follow the line s = t, on which C = s + 1 and the disk storage and I/O capacities are fully utilized. This rule is cost-minimal for t <= 1, but for t > 1, the locus of minimal-C follows the curve $s = t^{1/2}$, on which $C = 2\cdot t^{1/2}$. Putting the result back in dimensional form we obtain, for t >= 1:

$$(3.8)\quad COST_{min} = 2[(COST_m \cdot S_1)(2\cdot COST_\pi \cdot R/S_p)]^{1/2}$$

Thus the total cost of the LSM-tree (for $t \geq 1$) is seen to be twice the geometric mean of the (very high) cost of enough memory to hold all the data in the LSM-tree and the (extremely low) cost of disk required to support the multi-page block I/O needed to write its inserts to disk in the cheapest way. Half of this total cost is used for memory for S_0, the other half for disk for I/O access to S_1. The cost of disk storage does not show up because t >= 1 ensures that the data is warm enough to make disk I/O predominate over disk storage at the minimum point. Note that asymptotically, the cost goes as $R^{1/2}$ compared to R for the B-tree, as $R \to \infty$.

In the case that t <= 1, the cooler case, the minimum cost occurs along s = t, where C = t + 1 < 2. This means that the total cost in this case is always less than twice the basic cost of storing S_1 on disk. In this case we size disk by its storage requirements, and then use all its I/O capacity to minimize memory use.

Example 3.3. We consider the Account-ID||Timestamp index detailed in Example 3.1. The following analysis calculates costs for inserts only, with an insertion rate R of 16,000 bytes per second to the index (1000 16 byte index entries, not counting overhead), resulting in an index of 576 million entries for 20 days of data, or 9.2 GBytes of data.

Using a B-tree to support the index, the disk I/O will be the limiting factor as we saw in Example 3.1 — the leaf-level data is warm. We are required to use enough disk space to provide H = 2,000 random I/Os per second to update random pages at the leaf level (this assumes all directory nodes are memory resident). Using the typical value $COST_P$ = $25 from the table of Section 3.1, we find the cost for I/O is $H\cdot COST_P$ = $50,000. We calculate the cost to buffer upper-level nodes in memory as follows. Assume leaf nodes that are 70% full, $0.7\cdot(4K/16)$ = 180 entries per leaf node, and therefore the level above the leaf contains about 576 million/180 = 3.2 million entries pointing to subordinate leaves. If we grant some prefix compression so that we can fit 200 entries to a node at this level, this implies about 16,000 pages of 4 KBytes each, or 64 MBytes, at a cost for memory, $COST_m$, of $100 per MByte, or $6400. We ignore the relatively insignificant cost of node buffering at levels above this, and say that

the total cost of a B-tree is $50,000 for disk plus $6400 for memory, or a total cost of $56,400.

With an LSM-tree of two components, C_0 and C_1, we need an S_1 of 9.2 GBytes of disk to store the entries, at a cost of $COST_d \cdot S_1$ = $9,200. We pack this data closely on disk and calculate the total I/O rate H supported by an equal cost in disk arms using multi-page block I/O, as H = $9200/COST_\pi$ = 3700 pages per second. Now in Equation (3.6) we solve for r after setting the total I/O rate H as above, the rate R to 16,000 bytes/second, and S_p to 4K. From the resulting ratio $r = S_1/S_0 = 460$ and the fact that S_1 = 9.2 GBytes, we calculate 20 MBytes of memory for C_0, costing $2,000. This is the simple s = t solution, with total cost $11,200 and full utilization of disk capacity and I/O capability. Since t = .22 is less than 1, this is the optimal solution. We add $200 for 2 MBytes of memory to contain merging blocks, and arrive at a total cost of $11,400. This is a significant improvement over the B-tree cost.

Here is a full explanation of the solution. The insert rate of R= 16,000 bytes/second is turned into 4 pages/second that need to be merged from C_0 to C_1. Since C_1 is 460 times larger than C_0, the new entries from C_0 are on the average merged into positions 460 entries apart in C_1. Thus merging a page from C_0 requires reading and writing 460 pages of C_1, a total of 3680 pages per second. But this is exactly what 9.2 disks provide in multiblock I/O capacity, with each providing 400 pages/sec, 10 times the nominal random I/O rate of 40 pages/second.

Since this example shows full utilization of disk resources with two components, we have no reason to explore the three-component LSM-tree here. A more complete analysis would consider how occasional finds must be performed in the index, and would consider utilizing more disk arms. The following example shows a case where three components provide an improved cost for a pure insert workload.

Example 3.4. Consider Example 3.3, with R increased by a factor of 10. Note that the B-tree solution now costs $500,000 for 500 Gbytes of disk to support an I/O rate H = 20,000 I/Os per second; of this 491 Gbytes will be unutilized. But the B-tree is the same size and we still pay $6400 to buffer the directory in memory, for a total cost of $506,400. In the LSM-tree analysis, the increase of R by a factor of 10 means that t increases by the same factor, to 2.2. Since this t is greater than 1, the best 2-component solution will not utililize all the disk capacity. We use equation (3.8) to calculate the minimum cost of $27,000 for a two-component LSM-tree, half of which pays for 13.5 Gbytes of disk and half for 135 Mbytes of memory. Here 4.3 Gbytes of disk are unutilized. With 2Mbytes of memory for buffers, the total cost is $27,200.

Here is a full explanation of the two-component solution. The insert rate R = 160,000 bytes/sec is turned into 40 pages/second that need to be merged from C_0 to C_1. Since C_1 is 68 times larger than C_0, merging a page from C_0 requires 68 page reads and 68 writes to C_1, a total of 5450 pages per second. But this is exactly what 13.5 disks provide in multiblock I/O capacity.

With an LSM-tree of three components for the R = 160,000 bytes/second case, the cost of the largest disk component and a cost-balanced I/O rate are calculated as for two components. With $S_i/S_{i-1} = r$ for i = 1, 2, by Theorem 3.1, we calculate r = 23 and S_0 = 17 MBytes (for memory cost of $1700) for fully occupied disk arms. The smaller disk component costs just 1/23 of the larger. Now increasing the memory size from this point has no good cost effect, and decreasing the memory size will result in a corresponding factor, squared, increase in the cost of disk. Since the cost for disk is currently a good deal higher than the cost of memory, we do not gain cost effectiveness by memory size reduction. Thus we have an analogous s = t solution

in the three-component case. Allowing an additional 4 MBytes of memory for buffering, costing $400, for the two rolling merge operations, the total cost for a 3 component LSM-tree is therefore $9,200 for disk plus $2,100 for memory, or a total cost of $11,300, a further significant improvement over the cost of a 2-component LSM-tree.

Here is a full explanation of the three-component solution. The in-memory component C_0 has 17 Mbytes, the smaller disk component C_1 is 23 times larger, at 400 Mbytes, and C_2 is 23 times larger than C_1, at 9.2 Gbytes. Each page of the 40 pages/second of data that must be merged from C_0 to C_1 entails 23 pages of reading and 23 of writing, or 1840 pages per second. Similarly, 40 pages/second are being merged from C_1 to C_2, each of which requires 23 pages of reads and writes of C_2. The total of the two I/O rates is 3680, exactly the multiblock I/O capacity of the 9.2 G of disk.

An LSM-tree of two or three components will require more I/O for find operations than the simple B-tree. The largest component in either case will look very much like the corresponding simple B-tree, but in the LSM-tree case we have not paid the $6,400 for memory for buffering nodes just above the leaf level in the index. Nodes even higher in the tree are relatively so few as to be negligable, and we can assume they are buffered. Clearly we would be willing to pay for buffering all directory nodes if queries to find entries were sufficiently frequent to justify this cost. In the three-component case, we need to consider the C_1 component as well. Since it is 23 times smaller than the largest component, we can easily afford to buffer all of its non-leaf nodes, and this cost should be added in the analysis. The unbuffered leaf access in C_1 entails another additional read for the find in cases where an entry in C_2 is being sought, and there is a decision to be made whether to buffer the directory of C_2. Thus for the three-component case, there may be a few additional page reads over the two I/Os needed for finds in the simple B-tree (counting one I/O for a page write of a leaf node). For the two-component case, there may be one additional read. If we do buy the memory for the buffering of nodes above leaf level of the LSM-tree components, we can meet the B-tree speed in the two-component case and pay for one extra read only in some cases in the three-component case. The total cost to add buffering in the three-component case would then be $17,700, still far less than the B-tree. But it may well be better to use this money in other ways: a full analysis should minimize total cost over the workload, including both updates and retrievals.

We have minimized the total I/O needed for merge operations with given S_0 by varying the size ratios r_i, with the result of Theorem 3.1, and then minimized the total cost by choosing S_0 to achieve best disk arm and media cost. The only remaining variation possible in the LSM-tree is the total number, K+1, of components provided. It turns out that as we increase the number of components the size of S_0 continues to decrease until the point is reached where the ratio r between component sizes reaches the value e = 2.71. . . , or until we reach the cold-data regime. However, we can see from Example 3.4 that successively smaller S_0 components as the number of components increases make less and less difference to total cost; in an LSM-tree of three components, the memory size S_0 has already been reduced to 17 MBytes. Furthermore, there are costs associated with increasing the number of components: a CPU cost to perform the additional rolling merges and a memory cost to buffer the nodes of those merges (which will actually swamp the memory cost of C_0 in common cost regimes). In addition, indexed finds requiring immediate response will sometimes have to perform retrieval from all component trees. These considerations put a strong constraint on the appropriate number of components, and three components are probably the most that will be seen in practice.

4. Concurrency and Recovery in the LSM-tree

In the current Section we investigate the approaches to be used to provide concurrency and recover for the LSM-tree. To accomplish this, we need to sketch a more detailed level of design for the rolling merge process. We leave a formal demonstration of correctness of the concurrency and recovery algorithms for a later work, and try here simply to motivate the design proposed.

4.1. Concurrency in the LSM-tree

In general, we are given an LSM-tree of K+1 components, C_0, C_1, C_2, . . ., C_{K-1} and C_K, of increasing size, where the C_0 component tree is memory resident and all other components are disk resident. There are asynchronous rolling merge processes in train between all component pairs (C_{i-1}, C_i) that move entries out from the smaller to the larger component each time the smaller component, C_{i-1}, exceeds its threshold size. Each disk resident component is constructed of page-sized nodes in a B-tree type structure, except that multiple nodes in key sequence order at all levels below the root sit on multi-page blocks. Directory information in upper levels of the tree channels access down through single page nodes and also indicates which sequence of nodes sits on a multi-page block, so that a read or write of such a block can be performed all at once. Under most circumstances, each multi-page block is packed full with single page nodes, but as we will see there are a few situations where a smaller number of nodes exist in such a block. In that case, the active nodes of the LSM-tree will fall on a contiguous set of pages of the multi-page block, though not necessarily the initial pages of the block. Apart from the fact that such contiguous pages are not necessarily the initial pages on the multi-page block, the structure of an LSM-tree component is identical to the structure of the SB-tree presented in [21], to which the reader is referred for supporting details.

A node of a disk-based component C_i can be individually resident in a single page memory buffer, as when equal match finds are performed, or it can be memory resident within its containing multi-page block. A multi-page block will be buffered in memory as a result of a long range find or else because the rolling merge cursor is passing through the block in question at a high rate. In any event, all non-locked nodes of the C_i component are accessible to directory lookup at all times, and disk access will perform lookaside to locate any node in memory, even if it is resident as part of a multi-page block taking part in the rolling merge. Given these considerations, a concurrency approach for the LSM-tree must mediate three distinct types of physical conflict.

(i) A find operation should not access a node of a disk-based component at the same time that a different process performing a rolling merge is modifying the contents of the node.

(ii) A find or insert into the C_0 component should not access the same part of the tree that a different process is simultaneously altering to perform a rolling merge out to C_1.

(iii) The cursor for the rolling merge from C_{i-1} out to C_i will sometimes need to *move past* the cursor for the rolling merge from C_i out to C_{i+1}, since the rate of migration out from the component C_{i-1} is always at least as great as the rate of migration out from C_i and this implies a faster rate of circulation of the cursor attached to the smaller component C_{i-1}. Whatever concurrency method is adopted must permit this passage to take place without one process (migration out *to* C_i) being blocked behind the other at the point of intersection (migration out *from* C_i).

Nodes are the unit of locking used in the LSM-tree to avoid physical conflict during concurrent access to disk based components. Nodes being updated because of rolling merge are locked in write mode and nodes being read during a find are locked in read mode; methods of directory

locking to avoid deadlocks are well understood (see, for example, [3]). The locking approach taken in C_0 is dependent on the data structure used. In the case of a (2-3)-tree, for example, we could write lock a subtree falling below a single (2-3)-directory node that contains all entries in the range affected during a merge to a node of C_1; simultaneously, find operations would lock all (2-3)-nodes on their access path in read mode so that one type of access will exclude another. Note that we are only considering concurrency at the lowest physical level of multi-level locking, in the sense of [28]. We leave to others the question of more abstract locks, such as key range locking to preserve transactional isolation, and avoid for now the problem of phantom updates; see [4], [14] for a discussion. Thus read-locks are released as soon as the entries being sought at the leaf level have been scanned. Write locks for (all) nodes under the cursor are released following each node merged from the larger component. This gives an opportunity for a long range find or for a faster cursor to pass a relatively slower cursor position, and thus addresses point (iii) above..

Now assume we are performing a rolling merge between two disk based components, migrating entries from C_{i-1}, which we refer to as the *inner component* of this rolling merge, out to C_i, which we refer to as the *outer component*. The cursor always has a well-defined inner component position within a leaf-level node of C_{i-1}, pointing to the next entry it is about to migrate out to C_i, and simultaneously a position in each of the higher directory levels of C_{i-1} along the path of access to the leaf level node position. The cursor also has an outer component position in C_i, both at the leaf level and at upper levels along the path of access, corresponding to an entry it is about to consider in the merge process. As the merge cursor progresses through successive entries of the inner and outer components, new leaf nodes of C_i created by the merge are immediately placed in left-to-right sequence in a new buffer resident multi-page block. Thus the nodes of the C_i component surrounding the current cursor position will in general be split into two partially full multi-page block buffers in memory: the "emptying" block whose entries have been depleted but which retains information not yet reached by the merge cursor, and the "filling" block which reflects the result of the merge up to this moment but is not yet full enough to write on disk. For concurrent access purposes, both the emptying block and the filling block contain an integral number of page-sized nodes of the C_1 tree which simply happen to be buffer resident. During merge step operations restructuring individual nodes, the nodes involved are locked in write mode, blocking other types of concurrent access to the entries.

In the most general approach to a rolling merge, we may wish to retain certain entries in the component C_{i-1} rather than migrating all entries out to C_i as the cursor passes over them. In this case, the nodes in the C_{i-1} component surrounding the merge cursor will also be split into two buffer resident multi-page blocks, the "emptying" block that contains nodes of C_{i-1} that the merge cursor has not yet reached, and the "filling" block with nodes, placed left-to-right, that contain entries recently passed over by the merge cursor and retained in component C_{i-1}. In this most general case then, the merge cursor position is affecting four different nodes at any one time: the inner and outer component nodes in the emptying blocks where the merge is about to occur and the inner and outer component nodes in the filling blocks where new information is being written as the cursor progresses. Clearly these four nodes may all be less than completely full at any moment, and the same is true of the containing blocks. We take write locks on all four nodes during the time the merge is actually modifying the node structures and release these locks at quantized instants to allow a faster cursor to pass by; we choose to release locks each time a node in the emptying block in the outer component has been completely depleted, but the other three nodes will generally be less than full at that time. This is all right, since we can perform all operations of access on a tree with nodes that are less than completely full as well as blocks that are less than completely full with nodes. The case where one cursor passes another requires particularly careful thought, because in general the cursor position of the rolling merge being bypassed will be invalidated on its inner component, and provision must be

made to reorient the cursor. Note that all of the above considerations also apply at various directory levels of both components where changes occur because of the moving cursor. High level directory nodes will not normally be memory resident in a multi-page block buffer, however, so a somewhat different algorithm must be used, but there will still be a "filling" node and an "emptying" node at every instant. We leave such complex considerations for later work, after an implementation of the LSM-tree has provided additional experience.

Up to now we haven't taken any special account of the situation where the rolling merge under consideration is directed from the inner component C_0 to the outer C_1 component. In fact, this is a relatively simple situation by comparison with a disk-based inner component. As with all such merge steps, one CPU should be totally dedicated to this task so that other accesses are excluded by write locks for a short a time as possible. The range of C_0 entries to be merged should be pre-calculated and a write lock taken on this entry range in advance by the method already explained. Following this, CPU time is saved by deleting entries from the C_0 component in a batch fashion, without attempts to rebalance after each individual entry delete; the C_0 tree can be fully rebalanced after the merge step is complete.

4.2. Recovery in the LSM-tree

As new entries are inserted into the C_0 component of the LSM-tree, and the rolling merge processes migrates entry information out to successively larger components, this work takes place in memory buffered multi-page blocks. As with any such memory buffered changes, the work is not resistant to system failure until it has been written to disk. We are faced with a classical recovery problem: to reconstruct work that has taken place in memory after a crash occurs and memory is lost. As we mentioned at the beginning of Chapter 2, we don't need to create special logs to recover index entries on newly created records: transactional insert logs for these new records are written out to a sequential log file in the normal course of events, and it is a simple matter to treat these insert logs (which normally contain all field values together with the RID where the inserted record has been placed) as a logical base for reconstructing the index entries. This new approach to recover an index must be built into the system recovery algorithm, and may have the effect of extending the time before storage reclamation for such transactional History insert logs can take place, but this is a minor consideration.

To demonstrate recovery of the LSM-tree index, it is important that we carefully define the form of a checkpoint and demonstrate that we know where to start in the sequential log file, and how to apply successive logs, so as to deterministically replicate updates to the index that need to be recovered. The scheme we use is as follows. When a checkpoint is requested at time T_0, we complete all merge steps in operation so that node locks are released, then postpone all new entry inserts to the LSM-tree until the checkpoint completes; at this point we create an LSM-tree checkpoint with the following actions.

- We write the contents of component C_0 to a known disk location; following this, entry inserts to C_0 can begin again, but merge steps continue to be deferred.
- We flush to disk all dirty memory buffered nodes of disk based components.
- We create a special checkpoint log with the following information:
  - The Log Sequence Number, LSN_0, of the last inserted indexed row at time T_0
  - The disk addresses of the roots of all components
  - The location of all merge cursors in the various components
  - The current information for dynamic allocation of new multi-page blocks.

Once this checkpoint information has been placed on disk, we can resume regular operations of the LSM-tree. In the event of a crash and subsequent restart, this checkpoint can be located and

the saved component C_0 loaded back into memory, together with the buffered blocks of other components needed to continue rolling merges. Then logs starting with the first LSN after LSN_0 are read into memory and have their associated index entries entered into the LSM-tree. As of the time of the checkpoint, the positions of all disk-based components containing all indexing information were recorded in component directories starting at the roots, whose locations are known from the checkpoint log. None of this information has been wiped out by later writes of multi-page disk blocks since these writes are always to new locations on disk until subsequent checkpoints make outmoded multi-page blocks unnecessary. As we recover logs of inserts for indexed rows, we place new entries into the C_0 component; now the rolling merge starts again, overwriting any multi-page blocks written since the checkpoint, but recovering all new index entries, until the most recently inserted row has been indexed and recovery is complete.

This recovery approach clearly works, and its only drawback is that there is a possibly large pause while various disk writes take place during the checkpoint process. This pause is not terribly significant, however, since we can write the C_0 component to disk in a short period and then resume inserts to the C_0 component while the rest of the writes to disk complete; this will simply result in a longer than usual latency period during which index entries newly inserted to C_0 are not merged out to larger disk-based components. Once the checkpoint is complete, the rolling merge process can catch up on work it has missed. Note that the last piece of information mentioned in the checkpoint log list above was the current information for dynamic allocation of new multi-page blocks. In the case of a crash, we will need to figure out in recovery what multi-page blocks are available in our dynamic disk storage allocation algorithm. This is clearly not a difficult problem; in fact a more difficult problem of garbage collecting fragmented information within such a block had to be solved in [23].

Another detail of recovery has to do with directory information. Note that as the rolling merge progresses, each time a multi-page block or a higher level directory node is brought in from disk to be emptied it must immediately be assigned a new disk position in case a checkpoint occurs before the emptying is completed and remaining buffered information must be forced out to disk. This means that the directory entries pointing down to the emptying nodes must be immediately corrected to point to the new node locations. Similarly we must immediately assign a disk position for newly created nodes so that directory entries in the tree will be able to point immediately to the appropriate position on disk. At every point we need to take care that directory nodes containing pointers to lower-level nodes buffered by a rolling merge are also buffered; only in this way can we make all necessary modifications quickly so that a checkpoint will not be held up waiting for I/Os to correct directories. Furthermore, after a checkpoint occurs and the multi-page blocks are read back into memory buffers to continue the rolling merge, all the blocks involved must be assigned to a new disk position, and thus all directory pointers to subsidiary nodes must be corrected. If this sounds like a great deal of work the reader should recall that there is no additional I/O necessary and the number of pointers involved is probably only about 64 for each block buffered. Furthermore these changes should be amortized over a large number of merged nodes, assuming that the checkpoints are only taken frequently enough to keep recovery time from growing beyond a few minutes; this implies a few minutes of I/O between checkpoints.

5. Cost-Performance Comparisons with Other Access Methods

In our introductory Example 1.2, we considered a B-tree for the Acct-ID||Timestamp index on the History file because it is the most common disk-based access method used in commercial systems. What we wish to show now is that no other disk indexing structure consistently gives superior I/O performance. To motivate this statement, we argue as follows.

Assume we are dealing with an arbitrary indexing structure. Recall that we calculated the number of entries in the Acct-ID||Timestamp index by assuming they were generating 1000 entries per second over a 20 day period of accumulation with eight hour days. Given index entries 16 bytes in length (4 bytes for the Acct-ID, 8 bytes for the timestamp, and 4 bytes for the History row RID) this implies 9.2 GBytes of entries or about 2.3 million 4 KByte pages of index, even if there is no wasted space. None of these conclusions are subject to change because of the specific choice of index method. A B-tree will have a leaf level with a certain amount of wasted space together with upper level directory nodes, whereas an extendible hash table will have a somewhat different amount of wasted space and no directory nodes, but both structures must contain 9.2 GBytes of entries as calculated above. Now to perform an insert of a new index entry into an index structure, we need to calculate the page on which the entry is to be inserted and make sure that page is memory resident. The question naturally arises: Are newly inserted entries generally placed in an arbitrary position among all 9.2 GBytes of index entries that are already present? The answer, for most classical acccess method structures, is Yes.

Definition 5.1. We say that the index structure of a disk based access method has the property of being a *Continuum Structure* if the indexing scheme provides for immediate placement of a newly inserted index entry in its ultimate collation order, based on key-value, with all other entries already present.

Recall that successive transactions in the TPC benchmark application have Acct-ID values generated at random from each of 100,000,000 possible values. By Definition 1.1, each new entry insert of an Acct-ID||Timestamp index will be placed in a pretty much random position on one of 2.3 million pages of entries that already exist. In a B-tree, for example, the 576,000,000 accumulated entries will contain on the average 5.76 entries for each Acct-ID; presumably each entry with the same Acct-ID has a distinct Timestamp. Each new entry insert will therefore be placed on the right of all entries with the same Acct-ID. But this still leaves 100,000,000 points of insert randomly chosen, which certainly implies that each new insert will be on a random one of the 2.3 million pages of existing entries. In an extendible hashing scheme [9], by contrast, new entries have a collation order calculated as a hash value from the Acct-ID||Timestamp key-value, and clearly any placement of a new entry in sequence with all entries already present is equally likely.

Now 2.3 million pages is the minimum number on which the 9.2 GBytes of entries of a Continuum Structure can sit, and given 1000 inserts per second, each page of such a Structure is accessed for a new insert about once every 2,300 seconds; by the Five Minute Rule it is uneconomical to keep all these pages buffered. If we consider larger nodes to hold the entries as in the Bounded Disorder file [16], this provides no advantage, for although there is a greater frequency of reference, the cost of memory to buffer the node is also greater and the two effects cancel. In general, then, a page is read into memory buffer for an entry insert and must later be dropped from buffer to make room for other pages. In transactional systems that update disk pages in place before dropping them from buffer, this update requires a second I/O for each index insert. Thus we are able to state that a Continuum Structure that does not defer updates will require at least two I/Os for each index insert, approximately the same as a B-tree.

Most existing disk-based access methods are Continuum structures, including B-trees [5] and its large number of variants such as SB-trees [21], Bounded Disorder Files [16], various types of hashing schemes such as extendible hashing [9], and a myriad others. However, there are a few access methods which migrate their entries from one segment to another: MD/OD R-Trees of Kolovson and Stonebraker ([15]) and Time-Split B-trees of Lomet and Salzberg ([17], [18]). The Differential File approach [25] also collects up changes in a small component, later performing updates to the full-sized structure. We will consider these structures in a bit more depth.

First of all we should analyze exactly why the LSM-tree beats the Continuum Structure in terms of I/O performance, reducing the disk arm load as much as two orders of magnitude in certain situations. In its most general formulation the advantage the LSM-tree enjoys results from two factors: (1) the ability to keep component C_0 memory resident, and (2) careful deferred placement. It is crucial that the original insert be made to a memory based component. Inserts of new entries in Continuum Structures require two I/Os for exactly this reason: that the size of the index in which they must be placed cannot economically be buffered in memory. If the assured memory residence of component C_0 in the LSM-tree were not assured, if this were merely a probabilistic concomitant of buffering a relatively small disk resident structure, there would presumably be circumstances where the memory-resident property would deteriorate, and this would lead to serious deterioration in LSM-tree performance as a growing fraction of new entry inserts led to additional I/Os. Given the guarantee that the initial insert will not cause an I/O, the second factor supporting high performance in the LSM-tree, a careful deferred placement in the larger continuum of the index, is important to guarantee that component C_0 won't grow without control in the expensive memory medium. Indeed the multi-component LSM-tree provides for a sequence of deferred placements to minimize our total cost. It will turn out that with the special structures considered that are not Continuum Structures, that while deferred placement in the final position of newly inserted entries is provided for, this is not carefully done to guarantee that the initial component for new inserts remains memory resident. Instead this component is seen as disk resident in the defining papers, although a large proportion may be buffered in memory. But because there is no control of this factor, the component can grow to be predominantly disk resident, so that the I/O performance will degrade to a point where each new insert requires at least two I/Os, just like a B-tree.

Time-Split B-tree

To begin with, we consider the Time-Split B-tree or TSB-tree of Lomet and Salzberg ([17], [18]). The TSB-tree is a two-dimensional search structure to locate records by dimensions of timestamp and keyvalue. It is assumed that each time a record with a given key value is inserted, the old one becomes outmoded; however, a permanent history of all records, outmoded or not, is kept indexed. When a new entry is inserted in a (current) node of a TSB-tree that has no room to accept it, the node can be split either by key-value or by time, depending on circumstance. If a node is split by time, **t**, all entries with timestamp range less than **t** go to the history node of the split, all entries with timestamp range crossing **t** go to the current node. The object is to eventually migrate outmoded records out to a history component of the TSB-tree on inexpensive write-once storage. All current records and current nodes of the tree lie on disk.

We see the model for the TSB-tree is somewhat different from ours. We do not assume our older History row is outmoded in any sense when a new History row with the same Acct-ID has been written. It is indisputable that the *current node* set of the TSB-tree forms a separate component that defers updates to a longer-term component. However, there is no attempt to keep this current tree in memory as with the C_0 component of the LSM-tree. Indeed, the current tree is presented as being disk resident while the history tree is resident on write-once storage. There is no claim that the TSB-tree accelerates insert performance; the intent of the design is rather to provide a history index to all records generated over time. Without a guaranteed memory resident component to which new inserts are performed, we are back to the situation of two I/Os for each entry insert.

MD/OD R-Tree

The MD/OD R-tree of Kolovson and Stonebraker, [15], is comparable to the TSB-tree, in that it uses a two dimensional access method (R-tree) variant to cluster and index historical records by timestamp range and keyvalue. The important R-tree variation introduced in the MD/OD R-

tree is that the structure is meant to span magnetic disk (MD) and optical disk (OD); the ultimate object, as with the TSB-tree, is to eventually migrate outmoded records to an archive R-tree with leaf pages and appropriate directory pages contained on inexpensive write-once optical storage. This migration occurs by means of a Vacuum Cleaner Process (VCP). Whenever the R-tree index on magnetic disk reaches a threshold size, the VCP moves some fraction of the oldest leaf pages to the archive R-tree on optical disk. Two different variations of this process, involving the percentage to be vacuumed and whether the archive and current R-trees are one or two structures, are investigated in the paper (MD/OT-RT-1 and MD/OT-RT-2). As with the TSB-tree, the current (MD R-tree) is represented as being disk resident while the archive tree (OD R-tree) is resident on write-once storage, and there is no claim that the MD/OD R-tree accelerates insert performance. Clearly the OD target precludes the rolling merge technique. Without a guaranteed memory resident component to which new inserts are performed, we return to the situation of two I/Os for each entry insert. Indeed, even with a small number of records used for simulation in [15], Figure 4 shows that the average number of pages read per insert never goes below two for the two variant structures investigated. There is a rough correspondence between the LSM-Tree and the MD/OD R-Tree if the latter is promoted up one level of the memory hierarchy to use memory and disk, but most of the details are not the same because of the differences in the features of the three media.

Differential File

The Differential File approach [25] starts with a main data file which remains unchanged over an extended period, while newly added records are placed into a specific overflow area known as a *Differential File*. At some future point (not carefully specified) it is assumed that the changes will be amalgamated with the main data file, and a new Differential File will be started. Much of the content of the paper has to do with advantages of having a much smaller dynamic area and methods to avoid double-accesses, find operations by unique record identifier which need to look first in the differential file (through some index) and then in the main data file (presumably through a separate index). The concept of a Bloom filter is suggested as the main mechanism to avoid such double accesses. Once again, as with access methods defined above, the Differential File makes no provision to keep the Differential File memory resident. It is suggested in Section 3.4 that while the Differential File is being dumped and later incorporated into the main file, a "differential-differential" file could reasonably be held in memory cache to permit online reorganization. This approach is not analyzed further. It corresponds to the idea of maintaining a C_0 component in memory while C_1 is merged with C_2, but the presentation seems to assume relatively slow insert rates, confirmed by the example given in Section 3.2 of a 10,000,000 record file with 100 changes per hour. It is not suggested that a differential-differential file should be kept memory resident at all times and no mention is made of I/O savings for insert operations.

Selective Deferred Text Index Updates

The text index maintenance method of Dadum, Lum, Praedel, and Schlageter [7] is also designed to improve system performance in index updates by deferring the actual disk writes. Index updates are cached in memory until forced out by conflicts with queries or trickled out by a background task. This being a text system, a conflict here would be between the keywords associated with the document being updated and those associated with the query. After the update, the query runs off of the index on disk. Thus the memory cache is not part of the authoratative index, unlike the LSM-Tree. The deferral method allows some batching of updates in both in the forced and trickled cases. However the pattern of updates still looks like that of a Continuum Structure.

6. Conclusions and Suggested Extensions

A B-tree, because it has popular directory nodes buffered in memory, is really a hybrid data structure which combines the low cost of disk media storage for the majority of the data with the high cost of memory accessibility for the most popular data. The LSM-tree extends this hierarchy to more than one level and incorporates the advantage of merge I/O in performing multi-page disk reads.

In Figure 6.1, we expand on Figure 3.1, graphing "cost of access per MByte" against "rate of access per MByte", i.e., data temperature, for data access through a B-tree and through an LSM-tree of two components, i.e., number of disk components K = 1. Starting at the lowest access rate, "cold" data has a cost proportional to the disk media on which it sits; In terms of the typical cost figures, up to .04 I/Os per second per MByte, the "freezing point", disk access costs $1 per MByte. The "Warm data" region begins at the freezing point, when disk arms become the limiting factor in access and the media is underutilized; In terms of Example 3.3, 1 page I/O per second per MByte would cost $25 per MByte. Finally, we have "Hot data" when the access is so frequent that B-tree-accessed data should remain in memory buffers; at $100 per MByte of memory, the cost of this access rate will be $100 per MByte, and this implies a rate of at least 4 I/Os per second per MByte, the "boiling point".

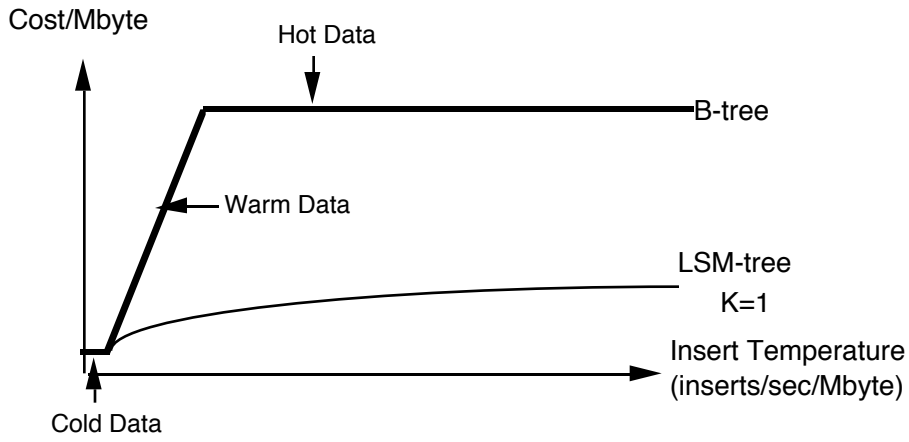

Figure 6.1. Graph of cost of access per MByte vs. Insert Temperature

The effect of buffering on a B-tree is to flatten the graph as the rate of access enters the Hot Data region, so that more frequent access doesn't result in ever higher costs extending the slope of the rising line for Warm Data. With a bit of thought, it can be seen that the effect of the LSM-tree is to reduce the cost of access, for any realistic rate of access for mergeable operations such as insert and delete, strongly towards that of cold data. Further, many cases of access rate that would indicate memory residence of the B-tree, the cases labeled "Hot Data" in Figure 4.1, can be accommodated mostly on disk with the LSM-tree. In these cases, the data is hot in terms of logical access rate (inserts/sec) but only warm in terms of physical disk access rate because of the batching effect of the LSM tree. This is an extremely significant advantage for applications that have a great preponderance of mergeable operations.

6.1 Extensions of LSM-tree Application

To begin with, it should be clear that the LSM-tree entries could themselves contain records rather than RIDs pointing to records elsewhere on disk. This means that the records themselves can be clustered by their keyvalue. The cost for this is larger entries and a concomitant ac-

celeration of the rate of insert R in bytes per second and therefore of cursor movement and total I/O rate H. However, as we saw in Example 3.3 a three component LSM-tree should be able to provide the necessary circulation at a cost of the disk media to store the records and index, and all of this disk media would be needed in any event to store the rows in a non-clustered manner.

Advantages of clustering might have quite important performance implications. For example, consider the Escrow transactional method [20], which serves as a good layer to support work-flow management because of the non-blocking nature of long-lived updates. In the Escrow method, a number of incremental changes to various aggregate Escrow fields can be generated by a long-lived transaction transaction. The approach used is to set aside the incremental amount requested (Escrow quantity) and unlock the aggregate record for concurrent requests. We need to keep logs for these Escrow quantities, and we can think of two possible clustering indexes for these logs: Transaction ID (TID) of the generating transaction, and Field ID (FID) of the field on which the Escrow quantity was taken. We might easily have twenty Escrow logs with a single TID in existence over an extended period (extended enough so that the logs are no longer be memory resident in classical log structures), and clustering by TID would be important up until the time when the transaction performs a commit or abort, which determines the ultimate effect these logs will have. In the event of a commit, the quantity taken out of the field would be permanent and the log can simply be forgotten, but in the event of an abort we would like to return the quantity to the field specified by the log's FID. A certain amount of speed is called for. In processing an abort, the logs of an aborted transaction should be accessed (clustering by TID is an important advantage) and fields with corresponding FID should be corrected. However, if the field is not memory resident, rather than read in the containing record the log can be re-inverted (placed in a different LSM-tree) clustered by its FID. Then when an Escrow field is read back into memory, we will try to access all logs clustered by FID that might have some update to perform; again there might be a large number of logs accessed, and clustering these logs in an LSM-tree is an important savings. Using LSM-trees to cluster Escrow logs first by TID, then by FID when the associated field is not in memory, will save a large number of I/Os where long-lived transactions make large numbers of updates to cold or warm data. This approach is an improvement over the "extended field" concept of [20].

Another possible variation to the LSM-tree algorithm mentioned at the end of Section 2.2 is the possibility of retaining recent entries (generated in the last τ_i seconds) in component C_i rather then letting them migrate out to C_{i+1}. A number of alternatives are suggested by this idea. One variation suggests that during cursor circulation, a time-key index such as that provided by the TSB-tree might be generated. The rolling merge can be used to provide great efficiency for new version inserts, and the multi-component structure suggests a final component migration to write-once storage, with a good deal of control over archival time-key indexing. This approach clearly deserves further study, and has been the subject of a conference paper [22].

Other ideas for further research include the following.

(1) Extend the cost analysis approach of Theorem 3.1 and Example 3.3 to situations where some proportion of find operations must be balanced with the merge for purposes of I/O balancing. Because of tha added load on the disks, it will no longer be possible to assign all of the disk I/O capacity to rolling merge operations and optimize for that case. Some proportion of the disk capacity will have to be set aside for the find operation workload. Other ways to extend the cost analysis are to allow for deletions prior to migration to component C_K and consider retaining some proportion of recent entries in the inner component C_{i-1} during the (C_{i-1}, C_i) merge.

(2) It is clear that we can offload the CPU work to maintain the LSM-tree so that this doesn't have to be done by the CPU that produces the log records. We merely need to communicate the

logs to the other CPU and then communicate later find requests as well. In cases where there is shared memory, it is possible that finds can be done almost without added latency. The design for such distributed work needs to be carefully thought out.

Acknowledgments

The authors would like to acknowledge the assistance of Jim Gray and Dave Lomet, both of whom read an early version of this paper and made valuable suggestions for improvement. In addition, the reviewers for this journal article made many valuable suggestions.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, "The Design and Analysis of Computer Algorithms", Addison-Wesley.

[2] Anon et al., "A Measure of Transaction Processing Power", *Readings in Database Systems*, edited by Michael Stonebraker, pp 300-312, Morgan Kaufmann, 1988.

[3] R. Bayer and M Schkolnick, "Concurrency of Operations on B-Trees", *Readings in Database Systems*, edited by Michael Stonebraker, pp 129-139, Morgan Kaufmann 1988.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman, "Concurrency Control and Recovery in Database Systems", Addison-Wesley, 1987.

[5] D. Comer, "The Ubiquitous B-tree", *Comput. Surv. 11*, (1979), pp 121-137.

[6] George Copeland, Tom Keller, and Marc Smith, "Database Buffer and Disk Configuring and the Battle of the Bottlenecks", *Proc. 4th International Workshop on High Performance Transaction Systems*, September 1991.

[7] P. Dadam, V. Lum, U. Praedel, G. Shlageter, "Selective Deferred Index Maintenance & Concurrency Control in Integrated Information Systems," Proceedings of the Eleventh International VLDB Conference, August 1985, pp. 142-150.

[8] Dean S. Daniels, Alfred Z. Spector and Dean S. Thompson, "Distributed Logging for Transaction Processing", *ACM SIGMOD Transactions*, 1987, pp. 82-96.

[9] R. Fagin, J. Nievergelt, N. Pippenger and H.R. Strong, Extendible Hashing — A Fast Access Method for Dynamic Files, *ACM Trans. on Database Systems*, V 4, N 3 (1979), pp 315-344

[10] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner and K. Salem, "Coordinating Multi-Transactional Activities", *Princeton University Report*, CS-TR-247-90, February 1990.

[11] Hector Garcia-Molina and Kenneth Salem, "Sagas", *ACM SIGMOD Transactions*, May 1987, pp. 249-259.

[12] Hector Garcia-Molina, "Modelling Long-Running Activities as Nested Sagas", *IEEE Data Engineering,* v 14, No 1 (March 1991), pp. 14-18.

[13] Jim Gray and Franco Putzolu, "The Five Minute Rule for Trading Memory for Disk Accesses and The 10 Byte Rule for Trading Memory for CPU Time", *Proceedings of the 1987 ACM SIGMOD Conference*, pp 395-398.

[14] Jim Gray and Andreas Reuter, "Transaction Processing, Concepts and Techniques", Morgan Kaufmann 1992.

[15] Curtis P. Kolovson and Michael Stonebraker, "Indexing Techniques for Historical Databases", *Proceedings of the 1989 IEEE Data Engineering Conference*, pp 138-147.

[16] Lomet, D.B.: A Simple Bounded Disorder File Organization with Good Performance, *ACM Trans. on Database Systems*, V 13, N 4 (1988), pp 525-551

[17] David Lomet and Betty Salzberg, "Access Methods for Multiversion Data", *Proceedings of the 1989 ACM SIGMOD Conference*, pp 315-323.

[18] David Lomet and Betty Salzberg, "The Performance of a Multiversion Access Method", *Proceedings of the 1990 ACM SIGMOD Conference*, pp 353-363.

[19] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil, "The Log-Structured Merge-Tree (LSM-tree)", UMass/Boston Math & CS Dept Technical Report, 91-6, November, 1991.

[20] Patrick E. O'Neil, "The Escrow Transactional Method", *TODS*, v 11, No 4 (December 1986), pp. 405-430.

[21] Patrick E. O'Neil, "The SB-tree: An index-sequential structure for high-performance sequential access", Acta Informatica 29, 241-265 (1992).

[22] Patrick O'Neil and Gerhard Weikum, "A Log-Structured History Data Access Method (LHAM)," Presented at the Fifth International Workshop on High-Performance Transaction Systems, September 1993.

[23] Mendel Rosenblum and John K. Ousterhout, "The Design and Implementation of a Log Structured File System", ACM Trans. on Comp. Sys., v 10, no 1 (February 1992), pp 26-52.

[24] A. Reuter, "Contracts: A Means for Controlling System Activities Beyond Transactional Boundaries", *Proc. 3rd International Workshop on High Performance Transaction Systems*, September 1989.

[25] Dennis G. Severance and Guy M. Lohman, "Differential Files: Their Application to the Maintenance of Large Databases", *ACM Trans. on Database Systems*, V 1, N 3 (Sept. 1976), pp 256-267.

[26] Transaction Processing Performance Council (TPC), "TPC BENCHMARK A Standard Specification", *The Performance Handbook: for Database and Transaction Processing Systems*, Morgan Kauffman 1991.

[27] Helmut Wächter, "ConTracts: A Means for Improving Reliability in Distributed Computing", IEEE Spring CompCon 91.

[28] Gerhard Weikum, "Principles and Realization Strategies for Multilevel Transaction Management", *ACM Trans. on Database Systems*, V 16, N 1 (March 1991), pp 132-180.

[29] Wodnicki, J.M. and Kurtz, S.C.: GPD Performance Evaluation Lab Database 2 Version 2 Utility Analysis, IBM Document Number GG09-1031-0, September 28, 1989.