Online update of B-trees.

ABSTRACT

Many scenarios impose a heavy update load on B-tree indexes in modern databases. A typical case is when B-trees are used for indexing all the keywords of a text field. For example upon the insertion of a new text record (e.g. a new document arrives), a barrage of new keywords has to be inserted into the index causing many random disk I/Os and interrupting the normal operation of the database. The common approach has been to collect the updates in a separate structure and then perform a batch update of the index. This update "freezes" the database. Many applications, however, require the immediate availability of the new updates without any interruption of the normal database operation. In this paper we present a novel online B-tree update method based on a new buffering data structure we introduce - Dynamic Bucket Tree (DBT). The DBT-buffer serves as a differential index for new updates. The grouping of keys in DBT-buffer is based on the longest common prefixes (LCP) of their binary representations. The LCP is used as a measure of the locality of keys to be transferred to the main B-tree. Our online update system does not slow down concurrent user transactions or lead to degradation of search performance. Experiments confirm that our DBT buffer can be efficiently used for online updates of text fields. As such it represents an effective solution to the notorious problem of handling updates to an Inverted Index.

1. INTRODUCTION

The amount of data stored in modern databases is constantly increasing. This is especially true when the database continually draws data from different sources, and is updated by multiple users simultaneously. A modern database system has to be able to deal with very high update rates. At the same time, many applications require that new updates be available for search immediately, not deferred.

Updates change not only the underlying tables but also their indexes. The performance of disk-based indexes is dominated by the number of disk I/Os. The focus of this paper is the efficient update of B-tree indexes.

Each insertion or deletion in a B-tree is performed with at most a logarithmic number of random disk I/Os. In practice, B-trees in most systems tend to favor the non-leaf nodes to be cached in the memory pool while the leaf pages tend to be swapped out. Assuming that most leaf pages are on the disk, each new update requires only one disk access per key. However, if we want to update a B-tree index with, say 300 keys simultaneously, even one disk I/O per key is an impractically high price to pay. The update of the index will take very long.

A commonly used solution is to defer the update and first collect and sort the (update) keys in a memory buffer and then transfer them to the disk-based B-tree in one bulk update. This will decrease the amortized number of disk I/Os per key, since several keys in the buffer will fall in the same B-tree leaf, and thus can all be inserted with one disk I/O. The search now needs to be performed simultaneously on the buffer and B-tree. Clearly, this is not a problem since the buffer is in main memory. However, when the buffer is full, its contents have to be transferred to the B-tree. This buffer-to-B-tree transfer takes a considerable time, during which all the database activities are "frozen," and therefore such a transfer is normally performed *off-line* (blocking search functionality completely). In summary, the efficiency of B-tree updates can be improved, though significantly harming search performance [7], or completely blocking it.

The problem, therefore, can be formulated as following:

> [**Online Update of B-trees**] How to improve the performance of B-tree updates avoiding periods of operation degradation or blockage.

This has a great practical importance for systems handling massive dynamic data collections. Consider as an example the scenario when a B-tree index stores all the keywords of a large unstructured text field. In this case, an update of a single text record triggers thousands of updates to the B-tree index. At the same time, if the update rate is constantly high, there is no system idle time for synchronization and maintenance operations.

Indexing the keywords of a text field cell corresponds to the task of indexing the keywords of a document. In such a case, the B-tree index is in essence used to store keywords of an inverted index. The online update of inverted indexes was identified as one of the main bottlenecks of a Web information retrieval system (cf. [11]).

The problem of online update of B-trees is not easy. All proposed systems collect the updates in a main-memory buffer, and once it is full, perform a bulk load from the buffer to the B-tree [2, 9, 14, 15, 19, 20] (see also [7] for a review). Also, the techniques for updating inverted indexes for large text collections, whether they use B-trees or other data structures, are essentially the same: collect new keywords in main memory, then merge them with the main on-disk index (see for example [4, 5, 6, 10, 13, 18, 21]. As explained above, such buffer-now-transfer all-later approaches suffer from significant performance penalties.

Our approach. The main idea of our approach is as follows. As in previous approaches, we also collect the update keys in a main memory buffer. However, what is different is the transfer of keys from the buffer to the index on disk. Intuitively, once the buffer is full, we can transfer to the index only a part (slice) of the buffered keys, thus avoiding interruptions of the database activity and freeing some space in the buffer for new updates. The question now becomes: what part of the buffered keys should we choose for such buffer-to-index transfer, in order to free up a fair amount of the buffer and still be unnoticeable with respect to the database performance.

For this, we designed a new data structure–*Dynamic Bucket Tree (DBT)*–which we use to organize the main memory buffer. The new keys pass through the main-memory DBT, and once we need to free up some part of the buffer, the structure locates the group (bucket) of keys that would lead to a high disk locality during transfer (touching only a small number of B-tree leaves).

The selection of the bucket to transfer is based on the longest common prefix (LCP) of the bit sequence representation of the keys in the bucket. Notably, the LCP of the keys is a good measure of their locality. Experiments comparing DBT with other buffering data structures (such as main memory B-trees) show that DBT performs better by an order of magnitude. Due to our short locality–aware transfers, our update system avoids degradation of the database performance during heavy of update load.

In addition, the search in the DBT buffer is very efficient because of the keyword-tree-like nature of our DBT structure.

To summarize, the main contribution of this paper is an index update system which is able to perform online update of B-trees. Specifically we propose:

1. A new data structure for buffering differential updates–Dynamic Bucket Tree (DBT).

2. A new measure of key locality, based on the longest common prefix (LCP) of the binary representation of keys.

3. A new slice-wise transfer strategy, which works well for high update loads, and does not cause a degradation of the database performance.

The rest of the paper is organized as follows. In Section 2 we review the related work. In Section 3 we describe the DBT buffer and the update strategy in detail. In Section 4 we present extensive experimental results of the update performance for various data sets.

2. RELATED WORK

In order to bring down the amortized number of disk I/Os per key insertion/deletion, the commonly used strategy is to buffer and group new updates in main memory and transfer them into the B-tree at once, accessing leaf nodes less often.

Several works explored the possibility of creating a main memory buffer for each internal node of B-tree [1, 2, 20]. A new key is first inserted into the root buffer. When a buffer is full its keys are distributed to the buffers of the next level (as determined by the intervals of the corresponding internal B-tree nodes). If there is no next level of internal nodes (i.e. they are the parents of the leaves), there are no main memory buffers where to distribute the keys. In this case, the keys of the entire internal buffer are transferred to the leaves of the B-tree. This transfer may touch many leaves, as many as the branching degree of the internal node whose associated buffer was full. As such this method was proposed for more efficient bulk update, but not online B-tree updates, which we consider in this work.

An alternative to buffering updates in buffers attached to tree nodes is to create a separate data structure for buffering new insertions (c.f. [9, 14, 15]). This data structure can be another B-tree or it can be a different type of in-memory data structure, e.g. hash table. In fact, it can also be a collection of data structures, forming a cascade of staging areas, similar to the organization of generational garbage collection [19]. The buffered keys are kept sorted. Once the buffer is full, a batch insert of the sorted buffered keys is performed [16]. A review of these buffering techniques can be found in [7].

Summarizing the above approaches, the amortized raw performance of B-tree updates is generally improved by using buffered updates. However, these approaches are “fill-in and then empty-all” buffering strategies, and as such cannot be used for online updates. This is because during the bulk transfer many leaves in distant parts of the B-tree may need to be accessed. This essentially “freezes” the normal operation of the database.

The most striking example of a heavy update load is when we want to index the keywords of a new document coming to be stored in a database as unstructured text. Indexing the keywords of text fields is equivalent to building an *inverted index*. An inverted index ([21]) stores mappings from each keyword to the IDs of documents (or records) containing the keyword. An inverted index stored as a flat file is very difficult to update online. The update of a single document requires multiple insertions and/or deletions of keywords throughout the entire file, and thus requires shifting disk-resident data. For this reason the problem of updating an inverted index was identified as one of the major bottlenecks in the Google’s seminal paper [3].

The possibility of dynamization of inverted indexes was studied in [6], based on merging an existing index with an in-memory buffer. The method uses the same buffering and merging strategy as for batch updates of a B-tree, and as such it belongs to the “fill-in and then empty-all” family of methods.

The use of B-trees for storage and dynamic updates was studied in [5]. Here authors propose to use a B-tree memory pool as a buffer for new keywords. The buffer is merged with the on-disk B-tree when full. The use of B-trees does not require shifting of the entire disk data as in the case of flat files. However, again, the insertions of new keywords from a buffer can spread out to the entire interval of B-tree leaves, and thus, the merge of the buffer with B-tree can take a long time, making the method infeasible for an online setting.

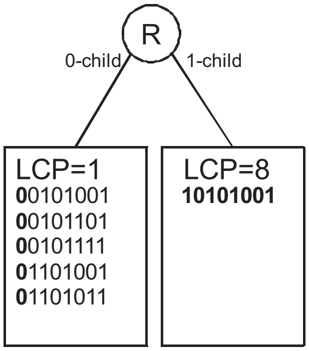

Figure 1: Initial state of the DBT-buffer. The maximum size of a bucket in this example is $k = 5$.

In contrast we propose a truly online method for handling heavy update loads without degrading or freezing the database.

3. OUR SOLUTION.

In this section we present our new buffer data structure, the *Dynamic Bucket Tree Buffer–DBT-buffer–*and show how it is used for online updates of B-tree indexes.

The DBT-buffer resides in main memory and consists of an array of B buckets, each holding up to k key-rowid pairs. These buckets are the leaves of a suffix-tree like structure. This tree contains at most B internal nodes and B leaves, the latter being the buckets themselves.

Each key is considered to be a sequence of bits (its binary representation). Each internal node of the tree has two children – the "0-bit" child and the "1-bit" child. The children of a (internal) node can be of two types: internal nodes or the buckets (leaves). Only the latter contain keys.

For simplicity we will explain here only the case of insertions.[1] When the system starts, all the B buckets are empty. The tree has initially the root and two buckets which are children of the root. The first keys are distributed among these two buckets. The keys whose first bit is 0 are inserted into the first bucket, and those with the first bit 1 are added to the second bucket. An example is shown in Figure 1.

The keys in the buckets are kept sorted. In addition, each bucket contains the bit length of the *longest common prefix* (LCP) of all the keys in the bucket. We refer to LCP as the length of the longest common prefix, not the prefix itself.

After adding a new key to a bucket, the LCP of the bucket might need to be updated. For this, we only need to find the LCP of the new key and a neighboring key in the sorted list of keys in the bucket. If this LCP is shorter than the bucket LCP, the latter is updated accordingly.

When a bucket becomes full, we split it, using the next available bucket in the bucket array. We distribute the keys into two buckets based on the bit at position $LCP + 1$ (See Figure 2 for an example). The LCPs of these two buckets are at least 1 bit longer than the LCP of the original bucket.

An internal node is created each time such a bucket, say

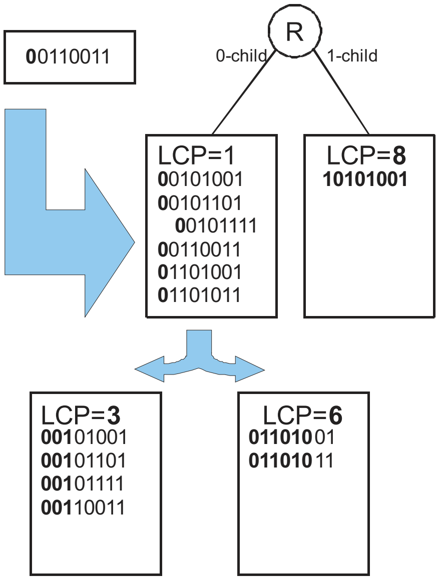

Figure 2: The first bucket split. Bits at position 2 will be used for the split.

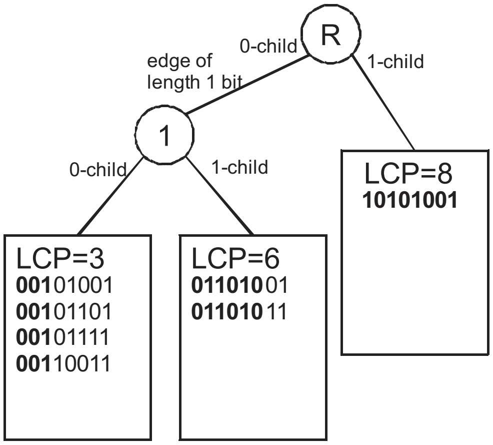

Figure 3: DBT tree update after the first bucket split.

[1] An update process can be regarded as a combination of insert and delete operations. Extending the buffer with the deleted keys can be done in a manner similar to adding the new keys, with an additional 1-bit flag, indicating deletion.

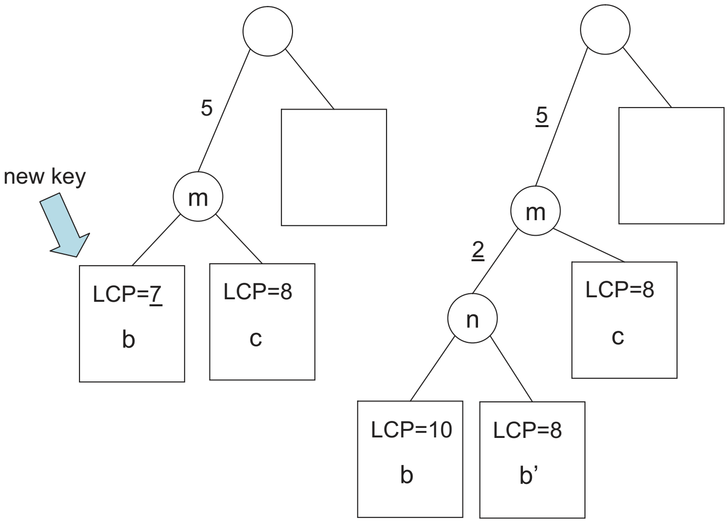

Figure 4: The depth of a new internal node n increases compared to its parent, and as such the LCP of each bucket increases with each split.

b, is split. Let m (internal node) be the parent of b before the split, and n be the new internal node which will be the parent of b and its new sibling b' after the split. n will be a child of m.

Each incoming edge to an internal node has a *length*. The length of the edge between m and n is equal to the increase in LCP of the keys in b and b' compared to the LCP of the keys in b and its former sibling before the split.

The *depth* of an internal node is the sum of edge lengths on the path from the root to this internal node. The *depth* of n equals the LCP of union of the keys in both the resulting buckets.

As an example consider Figure 4. Before the split (left), node m has two child buckets, b and c, with LCPs 7 and 8, respectively. The depth 5 of m indicates that the LCP of the union of keys in b and c is 5. The LCP of b is 7, and this says that the keys there share their first 7 bits, but there is at least a pair of keys differing on the 8th bit. After the split (right), the new internal node, n, has depth 7, which means the keys in b and b' share the first 7 bits. The length of the edge between m and n equals 2 which is the increase in the LCP for the union of keys in b and b'. Note that, the LCP of the resulting buckets is always greater than the LCP of the original bucket by at least one.

We use the LCP value of a given bucket as a measure of locality for all the keys inside it. Indeed, the longer the LCP of the bucket, the higher the chance that its keys will be inserted into the same (or neighboring) leaves of the B-tree. This will maximize the locality of disk I/Os when the bucket content is transferred to disk. As explained above, during a bucket split, the LCP of the two resulting buckets is 1 longer than the LCP of the original bucket, thus the locality of the keys inside the buckets increases with each split.

In the rare cases when all the keys in a full bucket are equal, we cannot split, and thus, we transfer all these keys to the B-tree, incurring essentially the same number of *random* I/Os as for the insertion of a single key (the least possible price for the insertion of multiple keys).

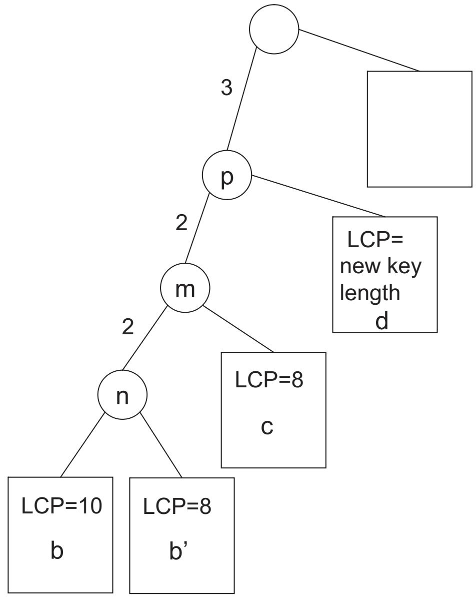

Figure 5: A new internal node p and a new bucket d are created for the new key. This is because only the three first bits of the new key match the incoming label of node m.

The insertion of keys continues causing new bucket splits as needed. If a new key does not share a prefix matching one of the paths to the existing buckets, then we say that this key does not belong to any such bucket, and thus, a separate bucket is created. Also, a new internal node is created as a parent of this newly created bucket. This internal node will split an edge at the proper depth. As an example consider the tree in Figure 4 (right). Suppose the edge connecting the root to node m represents the bit sequence 00001. Now suppose the 5-bit prefix of a new key is 00010, i.e. only the first three bits agree with the above edge. Clearly, this key does not belong to any existing bucket, and we need to introduce a new bucket d for it (see Figure 5). The new internal node, p, splits the edge from the root to m at depth 3.

At some point we will need a new bucket (to realize a split or accommodate a new key), while there is none left in the bucket array. In such a case, we need to free up at least one slot in the bucket array. The content of some bucket is to be transferred to the B-tree on disk.

We choose to transfer (to B-tree) a bucket with the longest LCP among all the buckets currently in the DBT-buffer. This is easily accomplished by a depth-first traversal of internal nodes of the tree selecting the internal node with the greatest depth. Next, the child bucket of this node with the largest number of keys is detached from the DBT-buffer, and its keys are transferred during a short batch insert into the B-tree. The internal node is removed, and one slot in the bucket array is freed up. The removal of such a deepest bucket is illustrated in Figure 6.

Note that the bucket whose keys are transferred to the B-tree (with the longest LCP), always contains at least $k/2$ keys, where k is the maximum capacity of a bucket. This is

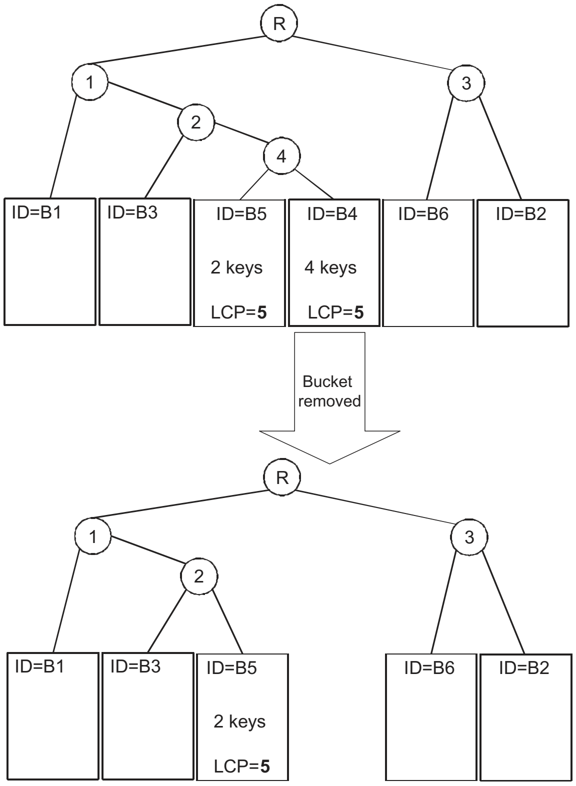

Figure 6: Removal of a deepest bucket.

true since the bucket with the largest LCP is guaranteed to be a child of some internal node, and as such is a result of a split of a previous bucket. After the split, one of two new child buckets contains at least k/2 keys, and it is exactly this bucket that is chosen for the transfer. In practice, the average number of keys in each transferred bucket is about 90% of its maximum capacity.

Summarizing, all the update keys pass through the DBT-buffer, where they are grouped into small chunks (buckets) of locality-related data, and eventually written to disk in such small local groups.

The DBT-buffer represents an additional in-memory index for searching the keys. Note that this additional search is very efficient with only $|Q|$ bit comparisons, where $|Q|$ is the query length, and does not depend on the total size of the buffer.

Regarding database consistency, similar latching and locking structures can be employed for the DBT buffer as for the B-tree. When the buckets are being flushed to the B-tree, these are done in such a way that the locks can be transferred to the B-tree entries.

Recovery from failures

Recall, that the changes (updates) that need to be reflected in the B-tree index are first reflected only in the in-memory DBT-buffer. The regime of transferring records from the DBT-buffer to the B-tree index is based on the value of their key, not the arrival time in the system. For example if we want to insert the keywords of a document, they will be first inserted into the DBT-buffer and the timing of when a keyword reaches the B-tree on disk depends only on the depth of the bucket that the keyword ends-up to be placed based on its binary value.

Now, the question is: what happens if the system crashes?

Our solution is a redo logging variant with checkpointing. In redo logging, the changes asked by a transaction T, before being reflected to the database, are first recorded in a log file, followed by a <*COMMIT(T)*> entry, and then a log flush. Only then, the buffered changes start to be transferred to the on-disk database.

If the system crashes, the recovery manager reads the log file from the beginning identifying all the transactions marked as committed, and re-executes these transactions. If a transaction is not marked as committed, then this transaction has not yet been reflected to disk, and thus it is safe to ignore.

We propose the following order when adding new keys to the DBT-buffer. Any change made in memory for a record which needs to be reflected to the B-tree index is marked with a special character $*$, e.g. $< T, A, 5, * >$. The key-pointer pair $(5, A)$ is *not* added to the DBT-buffer, at this point. After <*COMMIT(T)*> is written to the log, the log is scanned backwards to the <*START(T)*> point, and all the key-pointer pairs marked by $*$ are added to the DBT-buffer. If the system crashes, all the committed transactions are executed again. In this case, only the key updates that are part of committed transactions are transferred to the DBT-buffer; if the crash occurs during this process, some of the keys may be already in the B-tree. These would be re-updated anyway during the recovery.

Checkpointing is used to avoid scanning the log file from the beginning. We write *START CKPT(T, ...)* to express that a checkpointing started and record in this log entry all the uncommitted transactions. We then write *END CKPT*, only after all the committed transactions are guaranteed to be transferred to disk. Recall, that only the keys of the committed transactions are added to the DBT-buffer. Thus, between the start and the end of the checkpoint, the DBT-buffer is also completely emptied into the on-disk B-tree. Now, suppose that a crash occurs after *END CKPT*. Then, all the transactions committed before the *START CKPT(T, ...)* are guaranteed to be reflected to disk, including the B-tree updates. On the other hand, if a crash occurs after *START CKPT(T, ...)*, but before *END CKPT*, then we need to consider the log file starting from the previous *START CKPT*. Again, we only care for the committed transactions.

To summarize, the solution to the problem of recovery can be easily incorporated into an existing recovery scheme: redo logging with checkpoints.

4. EXPERIMENTAL EVALUATION

4.1 Setup

In order to evaluate the performance gain provided by the DBT buffer, we have designed four different update systems. For each type of the input data we consider, a large B-tree (2 GB) was generated in advance. These B-trees were then updated using one of the update strategies described in the following.

1. **No buffer.** Each new key is inserted into the B-tree immediately.

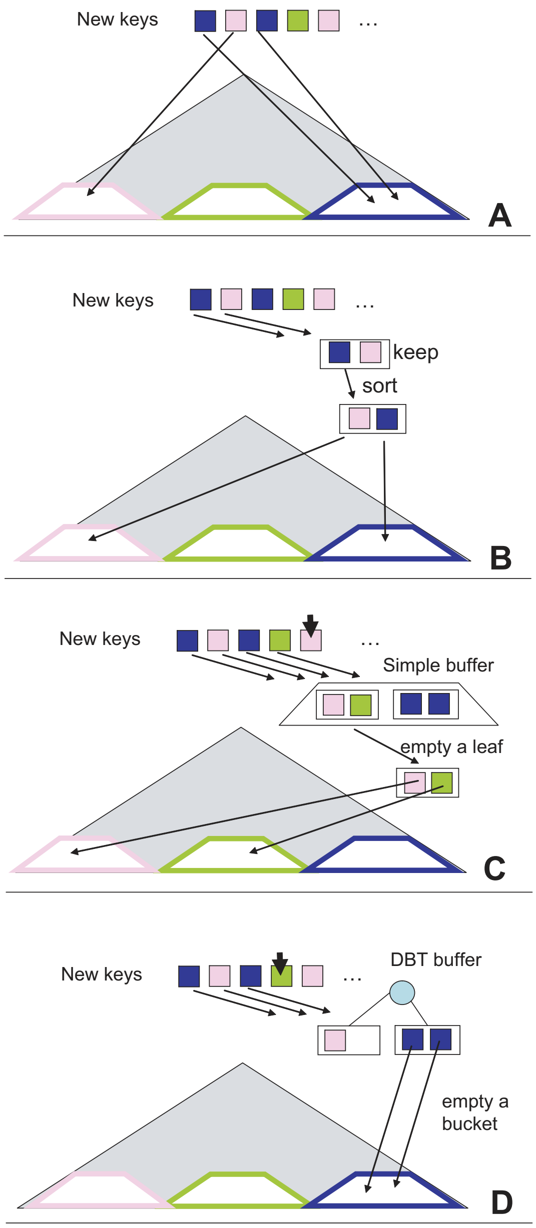

Figure 7: High-level representation of the possible buffering strategies for the online update of B-trees. A. No buffer: each key is inserted directly to the B-tree. B. Sequential buffer: keys are collected, sorted, and then transferred to the B-tree. C. Simple buffer: in-memory B-tree. D. DBT-buffer.

2. **Sequential buffer**. A small amount (350 in our experiments) of sequentially arriving keys is buffered in main memory, *sorted*, and then transferred to the B-tree in one batch insert of sorted keys. Note that even though the keys in such an insertion batch are sorted, in the typical case they are scattered over multiple B-tree leaves.

   The reason for considering 350 keys for the sequential buffer is that, in our experiment, this is the average number of keys we need to insert when updating (adding) a document. Considering more than this number of keys for the sequential buffer would give all the drawbacks of the batch update which we explained in detail earlier in this paper.

3. **Simple buffer**. The buffer represents an array of leaf buckets managed by a root node. This system is a two-level in-memory B-tree. The root contains a set of key intervals each associated with a pointer to a corresponding leaf bucket. The total size of the buffer in our experiments was 16 MB, thus, a two-level B-tree with 8000 keys in the root is sufficient. Based on our experiments, we set the size of the leaf buckets to be 128 keys. This size gave us the best locality during the update of the main index.

4. **DBT-buffer**. The keys are initially inserted into the DBT-buffer. When a new bucket is required and there are no free slots in the array of buckets, the bucket with the longest *LCP* is transferred to the B-tree. This system is described in detail in Section 3. The size of a bucket is the same as for the simple buffer, 128 keys.

The high-level graphical representation of the four update systems is depicted in Figure 7. The goal of the implemented systems is to keep the index up-to-date, *online*, i.e. the new keys are immediately available for search, and there is no big bulk transfer from the main memory buffer to disk.

In all the experiments, we have considered a continuous heavy workload and have measured the time required to keep the index updated when about $1,000,000$ new keys are added (one after another). Of course, for massive insertions of keys which are all sorted, no buffer is required, since they access the B-tree leaves sequentially. In practice, the updates to a B-tree index arrive in random order, i.e. they do not come as a stream of sorted keys, unless the column is of type DATE or TIMESTAMP and transactions come in time order. Therefore, we assume in our experiments that the keys are originally not sorted.

For the fairness of comparison, we compare the time and number of disk I/Os only for the last $350,000$ keys indexed during the insertion of $1,000,000$ total keys. This ensures that both a Simple buffer and a DBT-buffer are in their general working state, i.e. there are no empty slots in the bucket array. It turns out that when all slots are occupied, the insertion of keys using any of the buffering systems causes the same total number of keys to be transferred to the B-tree. However, the order and the grouping of transferred keys is different which accounts for better locality of disk accesses in our new DBT-buffer based system.

As a proof-of-concept we have implemented only the *insertion* of new keys. An update process can be regarded as a combination of insert and delete operations. Extending the buffer with the deleted keys can be done in a manner

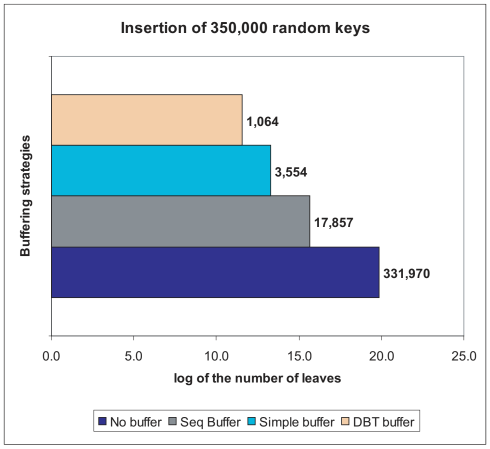

Figure 8: RANDOM KEYS. The figure shows the number of affected leaves for the same update with different buffering strategies. The numbers indicate the absolute number of leaves. The bars are in the logarithmic scale.

similar to adding the new keys, with an additional 1-bit flag, indicating deletion. This extension to the full update functionality would not influence the comparative performance results presented here.

The update systems were all implemented in C, and compiled using the GCC compiler version 4.3.3. The experiments were performed using the Ubuntu 9.04 operating system on a 3GHz Intel Pentium 4 machine with 2 GB of RAM.

We used three data sets (types): Randomly generated keys, string keys from MusicBrainz ([12]), and keywords from a large document collection ([8]).

4.2 Random keys.

As the first step in performance evaluation, we tested the efficiency of the above systems using randomly generated integer keys of 32 bits. The disk I/Os generally dominate the running time of the entire update process. Assuming that all but leaf nodes are kept in the main-memory pool, we have chosen as a measure of performance the number of leaves touched during the insertion of new keys. The leaf touch count is incremented only if the following leaf is not identical to the previous one. Thus, this count corresponds to the number of disk I/Os for loading the corresponding B-tree leaves from disk.

The results of this experiment for the insertion of $350,000$ last keys out of total inserted $1,000,000$ random numeric keys are shown in Figure 8. The integer keys are treated as binary sequences of 32 bits.

These results show that direct insertion of random keys is very costly, and the number of different leaves accessed is almost equal to the total number of new keys. Obviously, it is not feasible to directly update the B-tree with each new key in a heavy update load scenario. The buffering strategies (update systems 2,3, and 4) improve the performance tremendously. System 4 with the DBT-buffer performs considerably better than the Sequential buffer or the Simple buffer. For example, the DBT-buffer is 3 times better than the simple buffer.

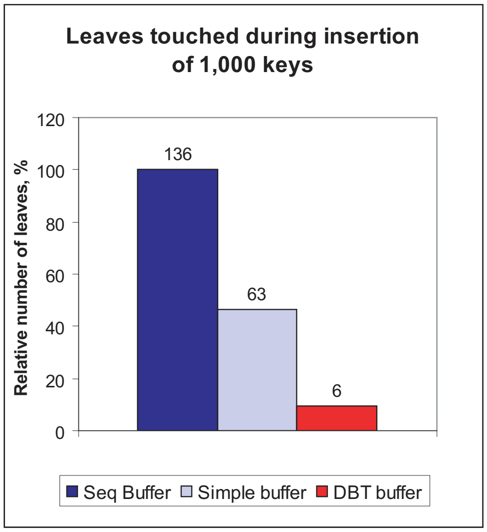

Figure 9: STRING KEYS FROM THE MUSIC DATABASE.The number of affected leaves for the same update with different buffering strategies for inserting 1000 new keys. The numbers indicate the absolute number of leaves. The bars show the percentage of systems 3 and 4 versus system 2 (simplest).

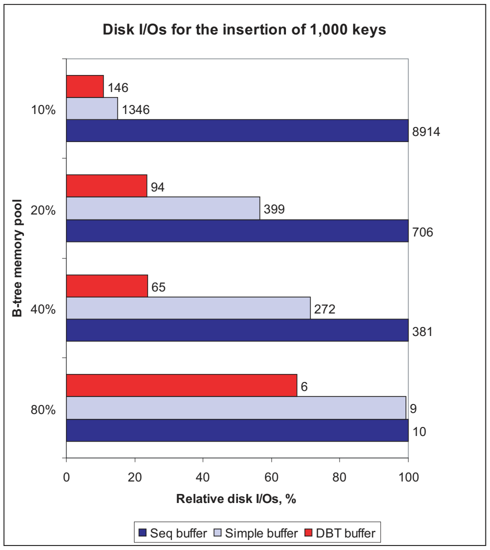

Figure 10: STRING KEYS FROM THE MUSIC DATABASE. The number of disk I/Os for the transfer of the 1000 keys to the B-tree. The results are for four different sizes of the memory pool for caching nodes of the main B-tree. The numbers on the bars indicate the actual number of disk I/Os. The bars show the relative number of disk I/Os as a percentage of the buffering system 2 (sorting 350 consecutive keys in main memory before the transfer).

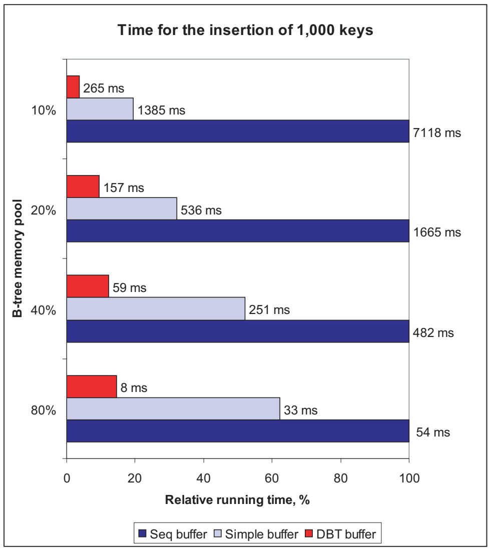

Figure 11: STRING KEYS FROM THE MUSIC DATABASE. The total running time for the transfer of 1000 keys to the B-tree. The results are for four different sizes of the memory pool for caching nodes of the main B-tree. The numbers on the bars indicate absolute time. The bars present the time as a percentage of the simplest buffering strategy, update system 2.

4.3 Performance evaluation for real data.

In order to model the performance for large disk-based B-trees, when only a small portion of B-tree is cached in the main memory pool, we turned-off the system cache, opening in our code the B-tree disk-resident file with O_DIRECT flag. This ensures that when we read the B-tree nodes into the memory pool, we always read from the physical disk, bypassing the system cache, avoiding the read-ahead buffering strategy and system cache optimizations. This provides much more reliable results and allows to model a situation when the B-tree file is much larger than the memory available to cache its nodes. Replacing the operating system cache with a controllable programmed memory pool is a common strategy in implementing large database systems.

We have experimented with indexing of large-scale data from the music track/CD collection provided in [12]. The Tracks table contains about $9,000,000$ records, and the Title column of this table represents a title of a music track (VARCHAR data type). As before, the large B-tree on the Title field was built in advance, and the $1,000,000$ last values were used for testing the online update.

The results presented in Figures 10, 11 and 9 do not include update system 1 (direct key insertion), since this system works many orders of magnitude slower and cannot be used for the heavy update load which we study in this paper. As for system 2, 350 consecutive keys were buffered, sorted and then inserted into B-tree.

We have measured the number of disk accesses for the insertion of the 1000 last keys. Since parts of the main B-tree are buffered in the memory pool, the number of actual disk I/Os (and the running time) depends on the size of the memory pool for caching B-tree nodes. The experimental results for different sizes of the B-tree memory pool are shown in Figures 10, and 11.

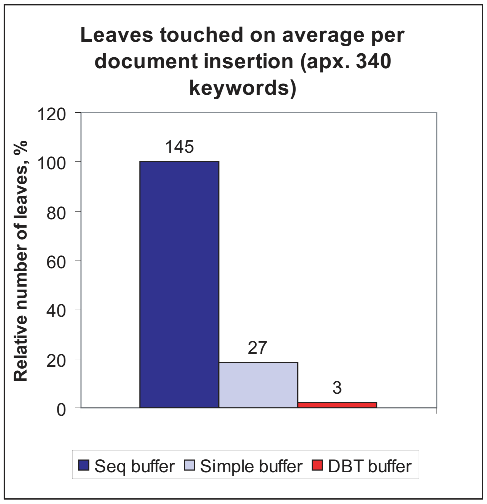

Figure 12: DOCUMENT KEYWORDS. The average number of different B-tree leaves affected by the insertion of a new document with 300–400 keywords each. The values on the bars indicate the absolute number of affected leaves. The bars give a percentage of the number of leaves comparing to system 2.

The results indicate that update system 4 (DBT-buffer) performs one to two orders of magnitude faster than the other systems. This becomes more apparent for large B-trees that cannot be efficiently cached in memory. As such, the proposed system (DBT) can be used during heavy update loads for updating a B-tree index without interrupting (freezing) the database operations. The benefits of the proposed system can be increased if we need to update multiple B-tree indexes for a single table. In this case, one DBT-buffer per each B-tree will solve the problem.

4.4 Update of B-tree with document keywords (Inverted index)

Further, we experimented with online insertion of document keywords. This represents a very high update load. As before, a large B-tree was created in advance which stores all keyword-document id pairs of a large document collection. This B-tree was then updated with the same set of new keywords, using one of the three update strategies, described above. In particular, for each document its words were parsed, sorted, duplicates removed, and then they were inserted: (for system 2) directly to B-tree in one batch insertion, (for system 3) directly to the Simple buffer with occasional transfers from buffer to disk and (for system 4) directly to the DBT-buffer, also with occasional transfers to disk.

The input was generated from the Gutenberg text collection [8], each text being split into a set of smaller texts of size up to 4 KB. The B-tree index had size 1.3 GB (before the updates) and contained the keywords of $300,000$ such documents (about 500 MB of input).

The set of documents for update was different from the

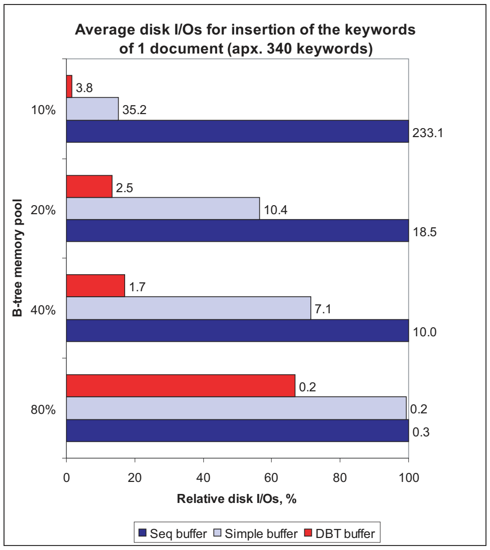

Figure 13: DOCUMENT KEYWORDS. The average number of disk I/Os for the insertion of new documents with 300–400 keywords each. The numbers on the bars indicate the average number of disk I/Os. The bars show the number of I/Os as a percentage of the update with the sorted keys of each new document - system 2.

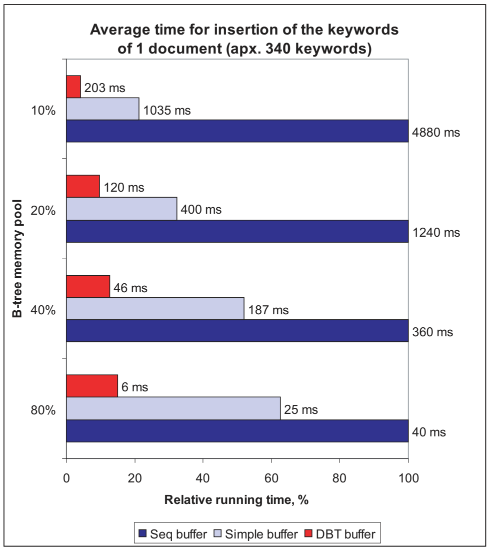

Figure 14: DOCUMENT KEYWORDS. The average running time for the insertion of new documents with 300–400 keywords each. The numbers on the bars indicate the average time in ms. The bars show the same results as a percentage of the time of the update with the sorted keys of each new document - system 2..

one used to build the index. This set was extracted from the same collection, and contained $48,607$ files of a total size of 191 MB. The total number of keywords to be inserted was about 1,000,000.

The results are presented in Figures 13, 14, and 12. From these results we can see that even in the very unfavorable situation, when 90% of B-tree nodes are accessed directly on disk, the DBT-buffer performs on average no more than 4 disk I/Os when inserting the keywords of a document (a document has on average 340 keywords). This is in stark contrast with the 233 I/Os for the online insertion of keys directly to the B-tree, or with the 35 disk I/Os when using the Simple Buffer. This relative disk I/Os pattern is also reflected in the running time. The insertion of the same amount of approximately 340,000 keys (1000 documents) into the main B-tree is orders of magnitude faster when using the DBT-buffer than when using direct updates.

The average number of B-tree leaves accessed (touched) during the insertion of each new document is depicted in Figure 12. This is a more objective measure of the efficiency of grouping keys before transferring them to disk, since it counts all accesses to the different leaves of the main B-tree, independently of whether they happened to be in the memory pool or are read from disk. Using the DBT-buffer is 30 times more efficient than insertion of sorted keys of one document and (more than) 10 times more efficient than using the Simple Buffer.

We also studied the dynamics of the key transfer from a buffer to a B-tree for the last 20 documents of each insert process. Figures 15 and 16 show the number of disk I/Os and the number of leaves touched for the insertion of each document. Approximately the same total number of keys is transferred to B-tree by all three update systems. The grouping of keys by locality, as DBT does, leads to much more efficient update, even though the number of inserted keys for some transfers may be significantly larger than when using the B-tree directly.

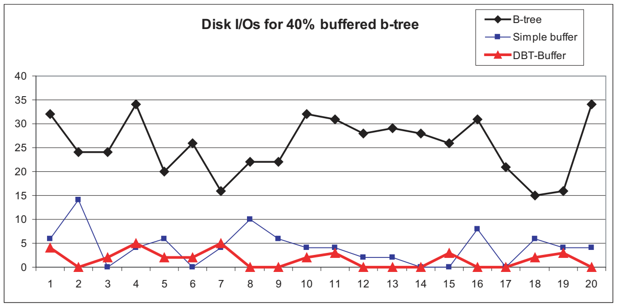

Figure 15: The absolute number of disk I/Os per document for a system with 40% of the B-tree cached in the main memory pool. The DBT-buffer shows the best grouping of the keys by locality so that the minimum number of disk I/Os is performed per document update.

If we collect the keywords of each document and immediately add them to the B-tree, they fall into numerous intervals represented by the leaves of the main B-tree. This causes multiple disk I/Os and is quite inefficient for large trees. Now consider the grouping of keys in the simple buffer or any variation of it. The distribution of the ranges of keys in the main B-tree and in the simple buffer is similar. However the buffer has a smaller number of ranges, which implies

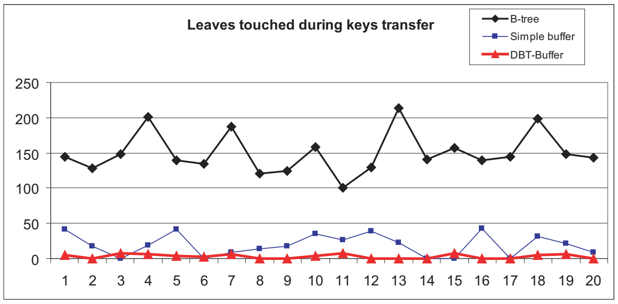

Figure 16: The absolute number of B-tree leaf nodes affected during each document insertion. The DBT-buffer shows the best grouping of the keys by locality so that the minimum number of leaves is accessed per document update.

that the keys from the simple–buffer range correspond to a much wider range of keys in the main B-tree. This explains the inefficiency of using the simple buffer strategy for online update.

The main efficiency gain of the DBT-buffer comes from the fact that it groups the new keywords always trying to narrow the range of those grouped together. The narrower the range, the less leaves of the B-tree index would be affected, thus causing less disk I/Os. Experiments confirm that when inserting the keys after grouping them with the DBT-buffer, the range of affected B-tree leaves is several times smaller than for the two other examined systems (see Figures 9,12, and 16).

5. CONCLUSION

We presented a novel system for managing B-tree updates. In contrast with the previous approaches, our system achieves online updates, and does not cause freezing of the database which is the main drawback of methods performing large bulk transfers from buffer to disk. Our new buffering method significantly outperforms other methods in terms of the total number of disk I/Os and absolute time required for transferring an equal number of keys to the leaves of a B-tree index. In addition, each of our short transfers is unnoticeable by the user – for example, the average time for indexing about 350 different keywords of a new document never exceeds 0.3 seconds in all our experiments. In comparison, the immediate online update of the B-tree with the sorted keys of each document calls for about 1 minute per document. By using our superior grouping of update keys we are able to handle heavy update loads without sacrificing the normal search operations. Our buffer is very efficient to search, with a time complexity that depends on query size only. This provides truly effective and immediate data availability. Our method can be used not only for online updates of B-tree indexes in relational databases, but also for online updates of inverted indexes for massive document collections.

6. REFERENCES

[1] L. Arge Efficient External-Memory Data Structures and Applications. *BRICS Dissertation Series*,DS-96-3: 1996.

[2] L. Arge, K. Hinrichs, J. Vahrenhold, and J. S. Vitter Efficient Bulk Operations on Dynamic R-Trees. *Algorithmica*, 33(1): 104–128, 2002.

[3] L. Brin, and L. Page The anatomy of a large-scale hypertextual Web search engine. *Computer networks and ISDN systems*, 30(1–7): 107–117, 1998.

[4] S. Buttcher, C. Clarke, and B. Lushman Hybrid Index Maintenance for Growing Text Collections. *SIGIR*: 2006.

[5] D. Cutting and J. Pedersen Optimizations for dynamic inverted index maintenance. *Proceedings of SIGIR90*: 405-111, 1990.

[6] L. Galambos Dynamization in IR Systems. *Intelligent Information Systems*:297–310, 2004.

[7] Goetz Graefe B-tree indexes for high update rates. *SIGMOD Record*, 35(1):39–44, 2006.

[8] Official project Gutenberg WEB site. *http://www.gutenberg.org/wiki/Main_Page*

[9] Leslie, H., Jain, R., Birdsall, D., and Yaghmai, H. Efficient Search of Multi-Dimensional B-trees. The VLDB conference: 710–719, 1995.

[10] L. Lim, M. Wang, S. Padmanabhan, J.S. Vitter, R. Agarwal Dynamic Maintenance of Web Indexes Using Landmarks. *The W3 Conference*: 2003.

[11] C. D. Manning, P. Raghavan, and H. Schütze Introduction to Information Retrieval *Cambridge University Press*: 2008.

[12] MusicBrainz, a user maintained community music metadatabase. *http://metabrainz.org/*

[13] A. Moffat, and J. Zobel Self-Indexing Inverted Files for Fast Text Retrieval. *ACM TIS*, 14(4): 349-379, 1996.

[14] Muth, P., O'Neil, P., Pick, A., and Weikum, G. The LHAM log-structured history data access method. *The VLDB Journal*, 8 (3–4): 199-221, 2000.

[15] O'Neil, P., Cheng, E., Gawlick, D., and O'Neil, E. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4): 351-385, 1996.

[16] Pollari-Malmi, K., Soisalon-Soininen, E., and Ylönen,T. Concurrency Control in B-trees with Batch Updates. *IEEE Trans. Knowl. Data Eng.*, 8(6):975–984, 1996.

[17] Project Gutenberg Selections. *http:/nltk.googlecode.com/svn/trunk/nltk_data*

[18] A. Tomasic, H. Garcia-Molina, and K. Shoens Incremental Updates of Inverted Lists for Text Document Retrieval. *the SIGMOD Conference*: 289-300, 1994.

[19] D. Ungar Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. *Software Development Environments (SDE)*: 157–167, 1984.

[20] J. Van den Bercken, B. Seeger, and P. Widmayer A Generic Approach to Bulk Loading Multidimensional Index Structures. *The VLDB conference*: 406-415, 1997.

[21] J. Zobel and A. Moffat Inverted Files for Text Search Engines. *ACM Computing Surveys* 38 (2): 6.