# Write-Optimized B-Trees

Goetz Graefe
Microsoft

## Abstract

Large writes are beneficial both on individual disks and on disk arrays, e.g., RAID-5. The presented design enables large writes of internal B-tree nodes and leaves. It supports both in-place updates and large append-only ("log-structured") write operations within the same storage volume, within the same B-tree, and even at the same time. The essence of the proposal is to make page migration inexpensive, to migrate pages while writing them, and to make such migration optional rather than mandatory as in log-structured file systems. The inexpensive page migration also aids traditional defragmentation as well as consolidation of free space needed for future large writes. These advantages are achieved with a very limited modification to conventional B-trees that also simplifies other B-tree operations, e.g., key range locking and compression.

Prior proposals and prototypes implemented transacted B-tree on top of log-structured file systems and added transaction support to log-structured file systems. Instead, the presented design adds techniques and performance characteristics of log-structured file systems to traditional B-trees and their standard transaction support, notably without adding a layer of indirection for locating B-tree nodes on disk. The result retains fine-granularity locking, full transactional ACID guarantees, fast search performance, etc. expected of a modern B-tree implementation, yet adds efficient transacted page relocation and large, high-bandwidth writes.

## 1 Introduction

In a typical transaction-processing environment, the dominant I/O patterns are reads of individual pages based on index look-ups and writes of updated versions of those pages. As memory sizes grow ever larger, the fraction of write operations among all I/O operations increases. While "90% reads, 10% writes" was a reasonable rule of thumb 15 or 20 years ago, "33% writes" is more realistic today once a database server and its applications have reached steady state production. In a future with 64-bit addressing in practically all servers and even most workstations, we may expect ever larger fractions of write operations among all I/O. In some scenarios, writes already dominate reads. For example, in a recent result of the SAP SD benchmark (designed for performance analysis and capacity planning of sales and distribution applications), simulating 47,528 users required 75 MB disk reads per second and 8,300 MB disk writes per second [LM 03]. In other words, in this environment with ample main memory, write volume exceeded read volume by a factor of more than 100.

In write-intensive environments, improving the performance of write operations is very important. Both on single disks and in disk arrays, large write operations provide much higher bandwidth than small ones, often by an order or magnitude or even more. In RAID-5 and similar disk arrays, large writes avoid the "small write penalty," which is due to maintenance of parity information. Log-structured file systems have been invented to enable and exploit large writes, but have not caught on in transaction processing and in database management systems. We believe this failed to happen for two principal reasons. First, log-structured file systems introduce overhead for finding the current physical location of a logical page, i.e., a mapping layer that maps a page identifier to the page's current location in the log-structured file system. Typically, this overhead implies additional I/O, locking, latching, search, etc., even if a very efficient mapping mechanism is employed. Second, log-structured file systems optimize write performance to the detriment of scan performance, which is also important in many databases, at least for some tables and indexes. Therefore, even if optimizing write performance is highly desirable for some tables in a database, it might not improve overall system performance if it applies indiscriminately to all data in the database.

The techniques proposed here are designed to overcome these concerns. First, the overhead of finding a single page is equal to that in a traditional B-tree index; retrieving a B-tree node does not require a layer of indirection for locating a page on disk. Second, if scan performance is important for some tables or indexes within a database, our design permits that those can be updated in-place, i.e., without any adverse effect on scan perform-

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004.

ance. Specifically, any individual write operation can be in-place ("read-optimized") or part of a large write ("write-optimized"), and the choice can be independent of the choices taken for other pages. In other words, our design provides the mechanisms for write-optimized operation, but it does not imply or prescribe policies and it does not force a single policy for all data and for all time.

Many policies are possible. For example, "hot" tables and indexes may be permanently present in the I/O buffer, which suggests write-optimized I/O when required, e.g., during checkpoints. Alternatively, B-tree leaf pages may be updated in-place (read-optimized) whereas upper index layers are presumed permanently buffered, and any required write operations bundled into large, efficient writes. Another possible policy writes in-place during ordinary buffer replacement but minimizes checkpoint duration by using write-optimized I/O.

The two extreme policies are updating everything in-place, which is equivalent to a traditional (read-optimized) database, or bundling all write operations into large, append-only writes, which is equivalent to a log-structured (write-optimized) file system. The value of the proposed design is that it permits many mixed policies, and that it applies specifically to B-tree indexes and thus database management systems rather than file systems. Therefore, if policies are set appropriately, our mechanisms will perform as well as or better than a traditional file system for applications in which a traditional file system out-performs a log-structured file system, and they will perform as well as or better than a log-structured file system for applications in which a log-structured file system out-performs a traditional file system.

In the following sections, we review related work including prior efforts to employ log-structured file systems for transaction processing, introduce our data structures and algorithms, consider defragmentation and the space reclamation effort required in a log-structured file system, describe the mechanisms that enable write-optimized B-tree indexes, review the performance of our mechanisms, and finally offer our conclusions from this research.

## 2 Related work

Our design requires limited modifications to traditional B-trees, and many of the techniques used here have already been employed elsewhere. In this section, we review B-trees, multi-level transactions, log-structured file systems, and prior attempts to use log-structured file systems in transaction processing.

Mentioned here briefly for the sake completeness, the proposed use of B-trees is entirely orthogonal to the data collection being indexed. The proposed technique applies to relational databases as well as other data models and other storage techniques that support associative search, both primary (clustered) and secondary (non-clustered) indexes. Moreover, it applies to indexes on traditional columns as well as on computed columns, including B-trees on hash values, Z-values (as in "universal B-trees" [RMF 00]), and on user-defined functions. Similarly, it applies to indexes on views (materialized and maintained results of queries) just as much as to indexes on traditional tables.

### *2.1 B-tree indexes*

B-tree indexes are, of course, well known [BC 72, C 79], so we review only a few relevant topics. Following common practice, we assume here that traditional B-tree implementations are actually $B^+$-trees, i.e., they keep all records in the leaf nodes and they chain nodes at the leaf level or at each level using "sibling" pointers. These are used for a variety of purposes, e.g., ascending and descending cursors.

For high concurrency, key range locking and equivalent techniques [L 93, M 90] are used in commercial database systems. Unfortunately, when inserting a new key larger than any existing key in a given leaf, the next-larger key must be located on the next B-tree leaf, which is an expensive operation even if all B-tree leaves are chained together. Such "crawling" can be particularly expensive (and complex to code correctly, and even more complex to test reliably as the software evolves) if B-tree leaves can be empty, depending on the policy when to merge and deallocate empty or near-empty leaf pages. Our B-tree modifications avoid all crawling for key range locking as a desirable-but-not-essential by-product.

A common B-tree technique is the use of "pseudo-deleted" or "ghost" records [JS 89, M90b]. Rather than erasing a record from a leaf page, a user's delete operation simply marks a record as invalid and leaves the actual removal to a future insert operation or to an asynchronous clean-up activity. Such ghost records simplify locking, transaction rollback, and cursor navigation after an update through the cursor. Ghost records can be locked and indeed the deleting user transaction retains a lock until it commits or aborts. Subsequent transactions also need to respect the ghost record and its key as defining a range in key range locking, until the ghost record is truly erased from the leaf page. Alternatively, a ghost record can turn into a valid record due to a user inserting a new row with the same index key. Interestingly, an insert operation realized by a conversion from a ghost record into a valid record does not require a key range lock; a key value lock is sufficient.

In most B-tree indexes, internal nodes have hundreds of child pointers, in particular if prefix and suffix truncation [BU 77] are employed. Thus, 99% and more of a B-tree's pages are leaf pages, making it realistic that all or most internal nodes remain in the I/O buffer at nearly all times. This is valuable both for random probes (e.g., driven by an index nested loops join) and for large scans, because efficient large scans on modern disk systems and disk arrays require tens or hundreds of concurrent read-ahead hints, which can only be supplied by scanning the "parent" and "grandparent" level, not by relying on the chain of B-tree leaves.

### 2.2 Multi-level transactions and system transactions

Modern transaction processing systems separate a database's logical contents from the database's physical representation. This is well known as physical data independence when designing tables, views, and constraints versus indexes and storage spaces. However, this distinction is also found in the implementation of query optimization, where logical query expressions with abstract operations such as join are mapped to physical query evaluation plans with concrete algorithms and access paths such as index nested loops join, and in the implementation of transaction semantics. Modification of physical representation, e.g., splitting a B-tree node or removing a ghost record, is often executed separately as a "nested top-level action" [MHL 92] or as a "system transaction." System transactions may change physical structures but never database contents, and thus differ from user transaction in a fundamental way. System transactions may commit and release their locks independently of the invoking user transaction, yet they may be lock-compatible with the invoking user transaction if that transaction pauses until the system transaction completes. Moreover, system transactions can be committed very inexpensively, i.e., without forcing the recovery log to stable storage, because durability of their effects is needed only if and when a subsequent user transaction and its log records rely on the system transaction's effects. If a user relies on the effects of a committed user transaction, that user transaction will have forced the log, which of course also forces any prior log records to stable storage, including those of any prior system transaction.

### 2.3 Log-structured file systems

The purpose of log-structured file systems is to increase write performance by replacing multiple small writes with a single large write [RO 92]. Reducing the number of seek operations is the principal gain; in disk arrays with redundancy, writing an entire "array page" at a time also eliminates the "small write penalty," which is due to adjusting parity pages after updates. While the actual parity calculations may be simple and inexpensive "exclusive or" computations, the more important cost is the need to fetch and then overwrite the parity page within an array page each time one of the data pages is updated. Thus, writing a single page may cost as much as 4 I/O operations in a RAID-4 or RAID-5 array, and even more in a RAID-6 or RAID-15 array.

Turning multiple small writes into a much more efficient single large write requires the flexibility to write dirty pages to entire new locations, which entails two new costs. First, there is a distinction between page identifier and page location – most of the file system links pages by page identifier, and page identifiers must be mapped to their current locations on disk. Updates to the structure that maintains this mapping must be logged carefully yet efficiently, quite comparable to the locking, latching, and logging required when splitting a B-tree page in a traditional multi-user multi-threaded database system. The main difference is that updates to the mapping information are initiated when the buffer manager evicts a dirty page, i.e., during write operations, rather than in the usual course of database updates.

Second, as pages are updated and their new images are written to new locations, the old images become obsolete and their disk space should be reclaimed. Unfortunately, disk pages will be freed in individual pages, not in entire array pages at a time, whereas only entire free array pages lend themselves to future fast write operations. The simple solution is to keep track of array pages with few remaining valid pages, and reclaim those disk pages by artificially updating them to their current contents – the update operation forces a future write operation, which of course will migrate the page contents to a new location convenient for the current large write operation at that time. Depending on the overall disk utilization, a noticeable fraction of disk activity might need to be dedicated to space reclamation. Fortunately, disk space is relatively inexpensive and many database servers run with less-than-full disks, because this is the only way to achieve the desired I/O rates. In fact, recent and current trends in disk technology increase storage capacity must faster than bandwidth, which motivates our research into bandwidth improvements through large write operations as well as justifies our belief that disks typically will be less than full and thus permit efficient reclamation and defragmentation of free space.

### 2.4 Transaction processing and log-structured file systems

A tempting but erroneous interpretation of the term "log-structured" assumes that a log-structured file system can support transactions without a recovery log. This is not the case, however. If a database system supports a locking granularity smaller than pages, concurrent transactions might update a single page; yet if one of the transactions commits and the other one rolls back, no page image reflects the correct outcome. In other words, it is important to realize that log-structured file systems are a software technique that enables fast writes; it is not an appropriate technique to implement atomicity or durability. Interestingly, techniques using shadow pages, which are similar to log-structured file systems as they also allocate new on-disk locations as part of write operations, have been found to suffer from a very similar restriction [CAB 81]. Consequently, shadow page techniques have been abandoned because they do not truly assist in the implementation of ACID transaction semantics, i.e., atomicity, consistency, isolation, and durability [G 81].

Seltzer's attempts of integrating transaction support into log-structured file systems [S 92, S 93, SS 90] did not materialize the expected gains in performance and simplicity, and apparently were abandoned. Rather than integrating transaction support into a file system, whether read-optimized or write-optimized, our approach is to integrate log-structured write operation into a traditional database management system with B-tree indexes, multi-level transactions, etc. It turns out that rather simple mechanisms suffice to achieve this purpose, and that these mechanisms largely exist but are not exploited for write-optimized database operation.

Lomet observed that the mapping information can be considered a database in its own right, and should be maintained using storage and transaction techniques similar to database systems [L 95], as in the Spiralog file system [WBW 96]. Our design follows this direction and keeps track of B-tree nodes and their current on-disk locations using traditional B-trees and database transactions, but it does not force all updates and all writes to migrate as log-structured file systems do.

If the mapping information can be searched efficiently as well as maintained efficiently and reliably, it is even conceivable to build a log-structured storage system that writes and logs not pages but individual records and other small objects, as in the Vagabond system [NB 97]. In contrast, our design leaves it to traditional mechanisms to manage records and objects in B-tree indexes and instead focuses on B-tree nodes stored as disk pages.

## 3 Proposed data structures and algorithms

In this section, we introduce our proposed changes to B-tree pages on disk and consider some of the effects of these changes. Further new opportunities enabled by these changes are discussed in detail in the subsequent sections.

Our proposed change is designed to solve the following problem. When a leaf page migrates to a new location, three pointers to that page (parent and two siblings) require updating. If a leaf page moves as part of a write operation, which is the essential mechanism of log-structured file systems whose advantageous effects we aim to replicate, not only its parent but also both of its siblings are updated and thus remain as dirty pages in the buffer pool. When those dirty pages are written, they too will migrate, and then force updates, writes, and migration of their respective siblings. In other words, updates and write operations ripple forward, backward, and back among the leaf pages.

### *3.1 Data structures*

Our proposed change in data structures is very limited. It affects the forward and backward pointers that make up the chain of $B^+$-tree leaves (and may also exist in higher levels of a $B^+$-tree). Instead of pointing to neighboring pages using page identifiers, we propose to retain in each page a lower and upper "fence" key that define the range of keys that may be inserted in the future into that page. One of the fences is an inclusive bound, the other an exclusive bound, depending on the decision to be taken when a separator key in a parent node is precisely equal to a search key.

In the initial, empty B-tree with one node that is both root and leaf, negative and positive infinity are represented with special fence values. If the B-tree is a partitioned B-tree [G 03], special values in the partition identifier (the artificial leading key column) can represent these two fence values. In principle, the fences are exact copies of separator keys in the parent page. When a B-tree node (a leaf or an internal node) overflows and is split, the key that is installed in the parent node is also retained in the two pages resulting from the split as upper and lower fences.

A fence may be a valid B-tree record but it does not have to be. Specifically, the fence key that is an inclusive bound can be a valid data record at times, but the other fence key (the exclusive bound) is always invalid. If a valid record serving as a fence is deleted, its key must be retained as ghost record in that leaf page. In fact, ghost records are the implementation technique of choice for fences except that, unlike traditional ghost records, fences cannot be removed by a record insertion requiring free space within a leaf or by an asynchronous clean-up utility. A ghost record serving as inclusive fence can, however, be turned into a valid record again when a new record is inserted with precisely equal key.

The desirable effect of the proposed change is that splitting a node into two or merging two nodes into one is simpler and faster with fences than with physical pointers, because there is no need to update the nodes neighboring the node being split or merged. In fact, there is only a single physical pointer (with page identifier, etc.) to each node in a B-tree, which is the traditional, essential parent-to-child pointer. The lack of a physical page chain differs from traditional B-tree implementations and thus raises some concerns, which we address next. The benefits of this change will be considered in subsequent sections.

### *3.2 Concerns and issues*

Before considering the effects of having only a single pointer to a B-tree node, from its parent, the most obvious issue to consider is the additional space requirement due to the fences. After all, the fences are keys, and keys can be lengthy strings values. Fortunately, however, these effects can be alleviated by suffix truncation [BU 77]. Rather than propagating an entire key to the parent node during a leaf split, only the minimal prefix of the key is propagated. Note that it is not required to split a full leaf precisely in the middle; it is possible to split near the middle if that increases the effectiveness of suffix truncation, and it is reasonable to do so because the shorter separator key in the parent will make future B-tree searches a little bit faster. Since the fences are literal copies of the separator key, truncating the separator immediately reduces not only the space required in the parent node but also the overhead due to fences.

While suffix truncation aids compressing the fences, the fences aid compressing B-tree entries because they simplify prefix truncation. The fences define the absolutely lowest and highest keys that might ever be in a page (until a future node split or merge); thus, if prefix truncation within each page is guided by the fences, there is no danger that a newly inserted key reduces the length of the prefix common to all keys in a page and requires reformatting all records within that page. Note that prefix truncation thus simplified can be employed both in leaves and in all internal B-tree nodes. If both prefix and suffix truncation is applied, then the remaining fences retained in a page may not be much larger than the traditional forward and backward pointers (page identifiers) they replace.

The exclusive fence record can simplify implementation of database compression in yet another way. Specifically, this record could store in each non-key field the most frequent value within its B-tree leaf (or the largest duplicate value), such that all data records with duplicate values can avoid storing copies of those values. This is a further simplification of the compression technique implemented in Oracle's database management system [PP 03].

Maybe the lack of forward pointers and its effect on cursors and on large (range or index-order) scans are a more substantial concern. Row-by-row cursors, upon reaching the low or high edge of a leaf node, must extract the fence key and search the B-tree from root to leaf with an appropriate "<", "≤", "≥", or ">" predicate, and the B-tree code must guide this search to the appropriate node, just as it does today when it processes "<" and ">" predicates.

For large scans, note that disk striping and disk arrays require deep read-ahead of more than one page. In a modern data warehouse server with 1 GB/s read bandwidth, 8 KB B-tree nodes, and 8 ms I/O time, 1,000 pages must be read concurrently (1 GB/s × 8 ms / 8 KB/page = 1,000 pages). Thus, a truly efficient range scan in today's multi-disk server architectures must be guided by the B-tree's interior nodes rather than based on the forward pointers, and in fact the page chain is useless today already for high-performance query processing.

Another important use of the page chain today is consistency checking – the ability of commercial database management systems to verify that the on-disk database has not been corrupted by hardware or software errors. In fact, write-optimized B-trees can be implemented without fence keys, but the reduced on-disk redundancy might substantially increase the effort required for detection of hardware and software errors. Thus, write-optimized B-trees without fence keys might not be viable for commercial database management systems. Fortunately, because the fences are precise copies of each other as well as the separator key in the parent node, they can serve the same purpose as the traditional page chain represented by page identifiers. Thus, our proposed change imposes no differences in functionality, performance, or reliability of consistency checks.

Key range locking, on the other hand, is affected by our change. Specifically, a key value captured in the fences is a resource that can be locked. Note that it is the key value (and a gap below or above that key) that is locked, not a specific copy of that key, and that it is therefore meaningless to distinguish between locking the upper fence of a leaf or the lower fence of that leaf's successor page. Because any leaf contains at least two fences, there never is a truly empty leaf page, and crawling through an empty leaf page to the next key is never required. More fundamentally, because a gap between existing keys never goes beyond a fence value (as the fence value separates ranges for the purpose of key range locking), crawling from one leaf to another in order to find the right key to lock is eliminated entirely. Thus, key range locking is substantially simplified by the presence of fences, eliminating both some complex code (that requires complex regression tests) and a run-time cost that occurs at unpredictable times. In fact, this benefit has been observed previously [ELS 97] but not, as in our design, exploited for additional purposes such as defragmentation, free space reclamation, and write-optimized B-trees.

## 4 Defragmentation and space reclamation

Large range queries as well as order-dependent query execution algorithms such as merge join require efficient index-order scans. Index updates, specifically split and merge operations on B-tree nodes, may damage contiguity on disk and thus reduce scan efficiency. Therefore, many vendors of database management systems recommend periodic defragmentation of B-tree indexes used in decision support.

During index defragmentation, the essential basic operation is to move individual or multiple pages allocated to the index. Pages are usually moved in index order and the move target is chosen in close proximity to the preceding correctly placed index node.

Reclaiming and consolidating free space as needed in log-structured file systems is quite similar. Again, the essential basic operation is to move pages with valid data to a new location. Pages to move are chosen based on their current location, and the move target is either a gap in the current allocation map or an area to which many such pages are moved. Not surprisingly, defragmentation utilities attempt to combine these two purposes, i.e., they attempt to defragment one or more indexes and concurrently consolidate free space in a single pass over the database.

### *4.1 B-tree maintenance during page migration*

Moving a node in a traditional B-tree structure is quite expensive, for several reasons. First, the page contents might be copied from one page frame within the buffer pool to another. While the cost of doing so is moderate, it is probably faster to "rename" a buffer page, i.e., to allocate and latch buffer descriptors for both the old and new locations and then to transfer the page frame from one descriptor to the other. Thus, the page should migrate within the buffer pool "by reference" rather than "by value." If each page contains its intended disk location to aid database consistency checks, this field must be updated at this point. If it is possible that a deallocated page lingers in the buffer pool, e.g., after a temporary table has been created, written, read, and dropped, this optimized buffer operation must first remove from the buffer's hash table any prior page with the new page identifier. Alternatively, the two buffer descriptors can simply swap their two page frames.

Second, moving a page can be expensive because each B-tree node participates in a web of pointers. When moving a leaf page, the parent as well as both the preceding leaf and the succeeding leaf must be updated. Thus, all three surrounding pages must be present in the buffer pool, their changes recorded in the recovery log, and the modified pages written to disk before or during the next checkpoint. It is often advantageous to move multiple leaf pages at the same time, such that each leaf is read and written only once. Nonetheless, each single-page move operation can be a single system transaction, such that locks can be released frequently both for the allocation information (e.g., an allocation bitmap) and for the index being reorganized.

If B-tree nodes within each level form a chain not by physical page identifiers but instead by lower and upper fences, page migration and therefore defragmentation are considerably less expensive. Specifically, only the parent of a B-tree node requires updating when a page moves. Neither its siblings nor its children are affected; they are not required in memory during a page migration, they do not require I/O or changes or log records, etc. In fact, this is the motivation of our proposed change in the representation of B-tree nodes.

### *4.2 Logging and recovery of page migrations*

The third reason why page migration can be quite expensive is logging, i.e., the amount of information written to the recovery log. The standard, "fully logged" method to log a page migration during defragmentation is to log the page contents as part of allocating and formatting a new page. Recovery from a system crash or from media failure unconditionally copies the page contents from the log record to the page on disk, as it does for all other page allocations.

Logging the entire page contents is only one of several means to make the migration durable, however. A second, "forced write" approach is to log the migration itself with a small log record that contains the old and new page locations but not the page contents, and to force the data page to disk at the new location prior committing the page migration. Forcing updated data pages to disk prior to transaction commit is well established in the theory and practice of logging and recovery [HR 83]. A recovery from a system crash can safely assume that a committed migration is reflected on disk. Media recovery, on the other hand, must repeat the page migration, and is able to do so because the old page location still contains the correct contents at this point during log-driven redo. The same applies to log shipping and database mirroring, i.e., techniques to keep a second (often remote) database ready for instant failover by continuously shipping the recovery log from the primary site and running continuous *redo* recovery on the secondary site.

A unique aspect of writing the page contents to its new location is that write-ahead logging is not required, i.e., the migration transaction may write the data page to the new location prior to writing any of its log records to stable storage. This is not true for the changes in the global allocation information; it only applies to the newly allocated location. The reason is that any recovery considers the new location random disk contents until the allocation is committed and the commit record is captured in the log. Two practically important implications are that a migration transaction with forced data write does not require any synchronous log writes, and that a single log record can capture the entire migration transaction, including transaction begin, allocation changes, page migration, and transaction commit. Thus, logging overhead for a forced-write page migration is truly minimal, at the expense of forcing the page contents to the new location before the page migration can commit. Note, however, that the page at the new location must include a log sequence number (LSN), requiring careful sequencing of the individual actions that make up the migration transaction if a single log record captures the entire transaction. The forced-write migration transaction will be the most important one in subsequent sections.

The most ambitious and efficient defragmentation method neither logs the page contents nor forces it to disk at the new location. Instead, this "non-logged" page migration relies on the old page location to preserve a page image upon which recovery can be based. During system recovery, the old page location is inspected. If it contains a log sequence number lower than the migration log record, the migration must be repeated, i.e., after the old page has been recovered to the time of the migration, the page must again be renamed in the buffer pool, and then additional log records can be applied to the new page. To guarantee the ability to recover from a failure, it is necessary to preserve the old page image at the old location until a new image is written to the new location. Even if, after the migration transaction commits, a separate transaction allocates the old location for a new purpose, the old location must not be overwritten on disk until the migrated page has been written successfully to the new location. Thus, if system recovery finds a newer log sequence number in the old page location, it may safely assume that the migrated page contents are available at the new location, and no further recovery action is required.

Some methods for recoverable B-tree maintenance already employ this kind of write dependency between data pages in the buffer pool, in addition to the well-known write dependency of write-ahead logging. To implement this dependency using the standard technique, both the old and new page must be represented in the buffer manager. Differently than in the usual cases of write dependencies, the old location may be marked clean by the migration transaction, i.e., it is not required to write anything back to the old location on disk. Note that redo recovery of a migration transaction must re-create this write dependency, e.g., in media recovery and in log shipping.

The potential weakness of this third method are backup and restore operations, specifically if the backup is "online," i.e., taken while the system is actively processing user transactions, and the backup contains not the entire database but only pages currently allocated to some table or index. Moreover, the detail actions of backup process and page migration must interleave in a particularly unfortunate way. In this case, a backup might not include the page image at the old location, because it is already deallocated. Thus, when backing up the log to complement the online database backup, migration transactions must be complemented by the new page image. In effect, in an online database backup and its corresponding restore operation, the logging and recovery behavior is changed in effect from a non-logged page migration to a fully logged page migration. Applying this log during a restore operation must retrieve the page contents added to the migration log record and write it to its new location. If the page also reflects subsequent changes that happened after the page migration, recovery will process those changes correctly due to the log sequence number on the page. Again, this is quite similar to existing mechanisms, in this case the backup and recovery of "non-logged" index creation supported by some commercial database management systems.

While a migration transaction needs to lock a page and its old and new locations, it is acceptable for a user transaction to hold a lock on a key with the B-tree node. It is necessary, however, that any such user transaction must search for the B-tree node again, with a new search pass from B-tree root to leaf, in order to obtain the new page identifier and to log further contents changes, if any, correctly. This is very similar to split and merge operations of B-tree nodes, which also invalidate knowledge of page identifiers that user transactions may temporarily retain. Finally, if a user transaction must roll back, it must compensate its actions at the new location, again very similarly to compensating a user transaction after a different transaction has split or merged B-tree nodes.

### 4.3 System transactions for page migration

While one may assume that database management systems already include defragmentation and a system transaction to migrate a page, our design is substantially more efficient than prior designs yet ensures the ability of media and system recovery. The most important advantage of the presented design over traditional page migration are the minimal log volume and the avoidance of ripple effects along the page chain. To summarize details of the redesigned page migration, as they may be helpful in later discussions:

- Since page migration does not modify database contents but only its representation on disk, it can be implemented as a system transaction.
- A system transaction can be committed very inexpensively without writing the commit record to stable storage.
- A page migration changes only one value in one B-tree node, i.e., the pointer from a parent node to one of its children, plus global allocation information.
- A migration transaction can force the page contents to its new location, log the page contents, or log only the migration without flushing.
- For system or media recovery after minimal logging, the page contents must be preserved in the old location, i.e., the old page location must not be overwritten, until the first write to the new location.
- The page migration operation must accept as parameters both the old and the new locations.
- When a B-tree node migrates from one disk location to another, it is required that the page itself is in memory in order to write the contents to the new location, and that its parent node is in memory and available for update in order to keep the B-tree structure consistent and up-to-date.
- The buffer pool manager can contribute to the efficiency of page migration by providing mechanisms to rename a page frame in the buffer pool.

We now employ this system transaction in our design for write-optimized B-trees.

## 5 Write-optimized B-trees

Assuming an efficient implementation of a system transaction to migrate a page from one location to another, the essence of our design is to invoke this system transaction in preparation of a write operation from the buffer pool to the disk. If the buffer pool needs to write multiple dirty pages to disk that do not require update-in-place for efficient large scans in the future, the buffer manager invokes the system transaction for page migration for each of these pages and then writes them to their new location in a single large write. In other words, the unusual and novel aspect of our design is that the buffer manager initiates and invokes a system transaction, in this case a page migration for each page chosen to participate in a large write.

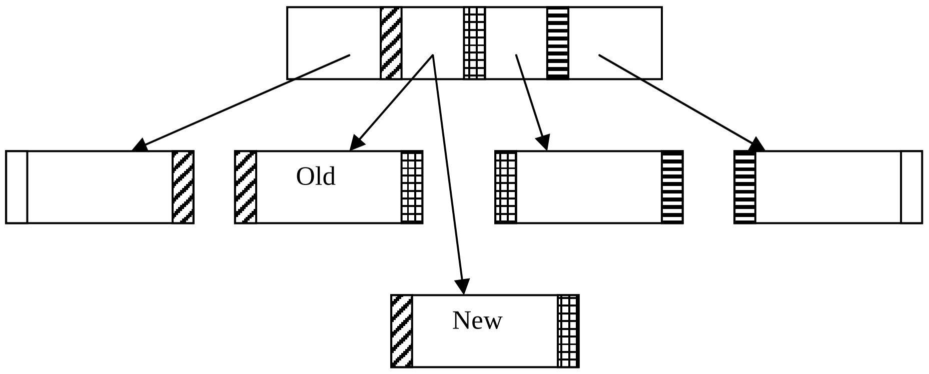

Figure 1. Page migration in a B-tree with fence keys.

Figure 1 illustrates the main concept enabling write-optimized B-trees, and also demonstrates the difference from B-trees implemented on top of log-structured file systems. When a page migrates to a new location as part of large write operation, its current location and thus the migration are tracked not in a separate indirection layer but within the B-tree itself. There is no need to adjust sibling pointers because those have become logical pointers, i.e., when a leaf is split, the separator key propagated to the parent node is retained in both leaves as lower and upper fence keys.

In many ways, recording a page's new location in a parent node is very comparable to recording the new location of a page in a log-structured file system. In fact, all the operations required in our system transaction are also required in a log-structured file system. The main difference is that our design keeps track of page migrations within the B-tree structures already present in practically all database management systems rather than imposing a separate mapping from logical page identifier to physical page location.

## *5.1 Accessing the parent node*

It is essential for efficient page migration that access to the parent node is very inexpensive. We offer three approaches to this concern, with the third approach representing the preferred solution.

First, it is possible to search the B-tree from the root and simply abandon the page migration if the parent node cannot be found without I/O – recall that our design does not require page migration as part of every write as a traditional log-structured file system does.

Second, given that a B-tree node can only be located from its parent node, it is extremely probable that the parent is still available in the buffer pool, suggesting that it is reasonable to require that for each B-tree node in the buffer pool, its parent (and transitively the entire path to the root) be present in the buffer pool. Incidentally, cursor operations can also benefit from the parent's guaranteed presence in the buffer pool. This requirement can be implemented efficiently by linking the buffer descriptor of any B-tree node to the buffer descriptor of its parent node. Since multiple children can link to a single parent, reference counting is required. The most complex and expensive operation is splitting a parent node, since this requires probing the buffer pool for each of the child nodes that, if present, must link to the newly allocated parent node. Note that this operation requires no I/O; only the buffer pool's internal hash tables are probed. To assess the overhead, it may be useful to consider that some commercial database management systems today approximate the effect of write-only disk caches [SO 90] by probing prior to each I/O the buffer manager's hash table for neighboring disk pages that are dirty and could be written without an additional disk seek.

Third, in order to avoid a hard requirement that the parent node be in the buffer for each B-tree node in the buffer, the buffer manager simply avoids page migrations for pages without a link to a parent node. Thus, when evicting an internal B-tree node, all links from child nodes also in the buffer must be removed first, which requires multiple probes into the buffer pool's hash tables but no I/O. If a parent is reloaded into the buffer pool, the buffer manager may again search whether any child nodes are in the buffer, or a child-parent link may be re-established the next time a B-tree search navigates from the parent to a particular child node.

## *5.2 B-tree root nodes*

B-tree root nodes have no parent node, of course, and their locations are recorded in the database catalogs. For root nodes, two alternatives suggest themselves.

First, given that page migration must be possible for defragmentation, there probably exists a system transaction to migrate a root page and correctly update the database catalogs. If root pages are appropriately marked in their buffer descriptors, this system transaction could be invoked by the buffer manager.

Second, B-tree root pages are always updated in place, i.e., they do not migrate as part of large write operations. Either the root pages are specially marked in their buffer descriptors or the absence of a link to the buffer descriptor of the parent page is interpreted precisely as for other B-tree nodes whose parent nodes have been evicted from the buffer pool, as discussed above.

### *5.3 Storage structures other than B-trees*

If the database contains data structures other than B-trees, those structures can be treated similar to B-tree root nodes. In other words, they can be updated in place or specialize migration transactions could be invoked by the buffer manager. However, since the focus of this research is on write-optimized B-trees, we do not pursue the topic further. It may be worth to point out, however, that prior research has suggested employing B-tree structures even for somewhat surprising purposes, e.g., for run files in external merge sort [G 03].

### *5.4 Allocation and deallocation of disk pages*

Keeping track of free space is a concern common to all log-structured file systems. Typically, a bitmap with a bit per page on the disk is divided into page-sized sections, these pages kept in the buffer pool for fast access, and dirty pages written to disk during database checkpoints. Some database systems, however, also maintain a bitmap per index. These bitmaps can guide fast disk-order index scans, provide added redundancy during consistency checks, and speed the search for a "lost" page identified in a consistency check. In a write-optimized environment, however, redundancy and update costs should be kept to a minimum, i.e., per-index bitmaps should be avoided. Instead, consistency checks and large scans should exploit the upper B-tree levels. Given that file systems rely entirely on tree structures for both purposes, and given that database management systems often use files in a file system to store data and logs, it is reasonable to conclude that database management systems also do not need this extra form of redundancy.

If a page is newly allocated for an index, e.g., due to a node split, it does not seem optimal to allocate a disk location for the new node if it will migrate as part of writing it to disk for the first time. For those cases, we suggest simulating a virtual disk device. Its main purpose is to dispense unique page identifiers that are used only while a newly allocated page remains in the buffer pool. In fact, the location of the buffer frame within the buffer pool could serve this purpose. When a new page is required, a page identifier on this virtual device is allocated and recorded in the node's parent. More importantly, this page identifier is used in log records whenever a page identifier is required. When the page is written to disk, it migrates from its virtual disk location to a genuine disk location, using the system transaction for page migration defined earlier. This technique avoids the cost of allocating a free disk page when splitting a B-tree node. Its expense, however, is additional complexity should the buffer manager attempt to evict the parent node prior to writing such a newly allocated page.

A very similar technique also applies to deallocation of pages. While multiple newly allocated pages require different virtual page identifiers, deallocated pages can probably all migrate to a single "trash bin" location.

### *5.5 Benefits*

Having considered our design for write-optimized B-trees in some details, let us now review some benefits and advantages of the design, comparing it both to traditional read-optimized B-trees and to log-structured file systems.

An important benefit relative to log-structured file systems is that page migration is tracked and recorded within the B-tree structure. Thus, probing a B-tree for individual nodes, e.g., in an index nested loops join operation, is just as efficient as in read-optimized B-trees, without the complexity and run-time overhead associated with a log-structured file system. Thus, we believe that this design is attractive for online transaction processing environments, whereas prior designs based on log-structured file systems were not.

An important benefit relative to read-optimized B-trees is that write operations can be much larger than individual B-tree nodes. It is well known that disk access time is largely seek and rotation time except for very large transfers, and that random disk writes are not as fast as strictly sequential log writes. In fact, our design enables enormously flexible write logic. Dirty pages can be written in-place as in traditional database management systems, they can use the append-only logic of log-structured file systems in order to make previously random data writes as fast as sequential log writes, or they can be written very opportunistically at a location that is currently particularly convenient. For example, the NetApp file system [HM 00] uses "write anywhere" capabilities to write in any free location near the current location of the disk access mechanism. Using the same rationale, a database management system can write a dirty page to any free location near a currently active read request, as an alternative to write-only disk caches [SO 90].

In disk arrays, the ability to convert multiple small write requests into a single large write operation provides continuous load balancing and it circumvents the "small write penalty" [PGK 88]. In RAID-4, -5, -6, and -15 arrays [CLG 94], modifying a single data page requires reading, modifying, and updating one or more pages with parity data, and possibly even logging them for recovery purposes. Write-optimized B-trees and their large write operations are therefore a perfect complement to such disk arrays.

Finally, B-trees can benefit from a particularly simple and efficient form of compression. Recall that B-tree pages are utilized only about 70 % in most realistic scenarios [JS 89]. Thus, if multiple B-tree pages are written sequentially, multiple B-tree nodes can be compressed without any encoding effort. Unfortunately, data from an individual B-tree node may straddle multiple pages, and whether or not this form of compaction is an overall performance gain remains a topic for future research.

### *5.6 Space reclamation overhead*

Write-optimized B-trees migrate individual pages from their current on-disk location to a new location, very similar to log-structured file systems, and must reclaim the fragmented free space left behind by page migrations. The required mechanisms must identify which areas of disk space to reclaim and then initiate a page migration of the valid pages not yet migrated from the area being reclaimed. It might very well be advantageous to distinguish multiple target areas depending on the predicted future lifetime of the data, e.g., using generation scavenging [OF 89, U 84] or a scheme based on segments like Sprite LFS [RO 92]. Our design makes no novel contributions for space reclamation policies, and we propose to adopt mechanisms developed for log-structured file systems, including space reclamation that also achieves defragmentation within each file or B-tree as a side benefit.

There is, however, an additional technique that is compatible with write-optimized B-trees but has not been employed in log-structured file systems. If disk utilization is very high and space reclamation is urgent, frequent, and thus expensive, the techniques explored in this research permit switching to read-optimized operation at any time. Thus, write-optimized B-trees can gracefully degrade to traditional read-optimized operation, with performance no worse than today's high performance database management systems. Moreover, as space contention eases and free space is readily available again, write-optimized B-trees can switch back to large, high-bandwidth writes at any time.

## 6 Performance

In migration transactions, each page write requires an update in the page's parent page as well as a log record due to that update. In this section, we analyze how these increases in write volume affect overall performance.

Large write operations increase the write bandwidth of a single disk or of a disk array by an order of magnitude or more. If the increase in write volume is substantially lower than the increase in write bandwidth, the increased write volume will diminish but not negate the I/O advantage of write-optimized B-trees.

If migration transactions happen frequently, it seems worthwhile to optimize their logging behavior. We expect the log volume due to a migration transaction to be between 160 and 400 bytes. If a data page and therefore a B-tree node are as large as 8 KB, and if every single write operation initiates a migration transaction, the logging overhead will remain at 2-5%. Assuming the log writes are always sequential and always fast, the additional logging volume should be small compared to the time savings in data writes.

More importantly, writing a page might dirty a parent page that had been previously clean. If so, this parent page must also be written before or during the next checkpoint. If the parent migrates at that time, the grandparent needs to be written in the subsequent checkpoint, etc., all the way to the B-tree root. Thus, write-optimized B-trees increase the volume of write operations in a database.

Clearly, the B-tree root should be written only once during each checkpoint, no matter how many of its child nodes, leaf pages, and pages in intermediate B-tree layers have been migrated during the last checkpoint period. Thus, in order to estimate the increase in write volume, it is important to estimate at which level sharing begins on a path from a leaf to the root.

Assuming that each B-tree node has 100 children (a conservative value for nodes of 8 KB, in particular if prefix and suffix truncation are employed) and assuming that updates and write operations are distributed uniformly over all leaves, sharing can be estimated from the fraction of updated leaves during each interval between two checkpoints. If 1% of all leaves are updated, each parent node will see one migrated leaf per checkpoint interval, whereas grandparent nodes will see many migrations of parent nodes during each checkpoint interval, i.e., no effective sharing at the parent level but lots of sharing at the level of grandparent nodes. Thus, the volume of write operations is increased by a factor marginally larger than 2. If the fan-out of B-tree nodes is 400 instead of 100, for example because nodes are larger or because prefix and suffix truncation are employed, sharing happens after 2 levels if as little as 0.25% of leaves are updated in each checkpoint interval. If 1% of 1% of all leaf pages (or 1 page in 10,000; or 1 in 160,000 assuming the larger fan-out of 400) are updated during each interval between checkpoints, sharing occurs after two levels. In those cases, the write volume is increased by as much as a factor of 3. If write bandwidth due to large writes increases by a factor of 10, the increased write volume diminishes but does not erase the advantage of large writes.

The situation changes dramatically if updates are not distributed uniformly across all leaves, but instead concentrated in a small section of the B-tree. For example, if a B-tree is partitioned, e.g., using an artificial leading key column [G 03], the most active keys and records can be assigned to a single "hot" partition. Leaf pages in that partition will be updated frequently, whereas all other leaf pages will be very stable. For a data collection where 80% of all updates affect 20% of rows, this design can be quite attractive, not only but in particular when the storage is organized as a partitioned and write-optimized B-tree. Alternatively, a pair of partitions can operate similar to a differential file [SL 76], i.e., one partition is not updated at all and the other one contains all recent changes.

## 7 Summary, future work, and conclusions

In summary, the design presented here advances database index management in two ways: it improves the performance of B-tree defragmentation and reorganization, and it can be used to implement write-optimized B-trees.

For defragmentation, it substantially reduces the logging effort and the log volume without much added complexity in buffer management or in the recovery from system and media failures. In fact, the reduction in log volume may reverse today's advantage of rebuilding an entire index over defragmentation of the existing index. Incremental online defragmentation, one page and one page migration transactions at a time, is preferable due to better database and application availability, and can now be achieved with competitive logging volume and effort.

Incidentally, efficient primitives for page movement within a B-tree also enable a promising optimization that seems to have been largely overlooked. O'Neil's SB-trees are ordinary B-tree indexes that allocate disk space in moderately large contiguous regions [O 92]. A slight modification of that proposal is a B-tree of super-nodes, each consisting of multiple traditional single-page B-tree nodes (this is reminiscent of proposals to interpret a single-page B-tree node as a B-tree of cache lines, e.g., [CGM 02]). When a super-node fills up, it is split and half its pages moved to a newly allocated super-node. The implied page movement is very similar to that in B-tree defragmentation, and it could be implemented very efficiently using our techniques for defragmentation.

For write-optimized B-trees, the design overcomes the two obstacles that have prevented success in prior efforts to combine ideas from log-structured file systems with online transaction processing. First, page access performance is equal to that of traditional (read-optimized, update-in-place) B-trees, with no additional overhead due to write-optimized operation and page migration. Second, the presented design permits an arbitrary mixture of read-optimized and write-optimized operation, allowing a wide variety of policies that can range from traditional update-in-place to a pure log-structured file system.

Alternatively, the presented design for write-optimized B-trees could be employed in a traditional log-structured file system to manage and maintain the mapping from logical page identifiers to their physical locations. Database researchers have recommended maintaining this mapping and its underlying index structure with strict and reliable transaction techniques, including shared and exclusive locks, transaction commits, checkpoints, durability through log-based recovery, etc. [L 95]. However, to the best of our knowledge, this recommendation has not yet been pursued by operating system or file system researchers.

The essential insights that enable the presented design are that the pointers inherent in B-trees can keep track of a node's current location on disk, and that page migrations in log-structured file systems are quite similar to defragmentation. Exploiting the pointers inherent in B-trees eliminates the indirection layer of log-structured file systems. The similarity to defragmentation permits exploiting traditional techniques for concurrency control, recovery, checkpoints, etc. Thus, the principal remaining problem was equivalent to making defragmentation very efficient. This problem was solved by representing the chain of neighboring B-tree nodes not with physical pointers as in traditional $B^+$-trees but with fence keys, which are copies of the separator key posted to the parent node when a B-tree node is split. Migrating a page from one location to another, both during defragmentation or while assembling multiple dirty buffer pages into a large write operation, requires only a single update in the node's parent. This change can be implemented reliably and efficiently using a system transaction that does not require forcing its commit record to stable storage and also does not require logging or writing the page contents.

In addition to enabling fast defragmentation and write-optimized operation, the design also simplifies splitting and merging nodes as well as prefix truncation within a node. It even substantially simplifies key range locking, because it entirely eliminate the code complexity and run-time overhead of crawling to neighboring pages in search of a key to lock. Thus, lower and upper fence keys instead of sibling pointers may be a worthwhile modification of traditional $B^+$-trees even disregarding defragmentation and write-optimized larger I/O.

As the required mechanisms are simple, robust, and quite similar to existing data structures and algorithms, we expect that they can be implemented with moderate development and test effort. A thorough and truly meaningful performance analysis of alternative policies for page migration and space reclamation will be possible only with a working prototype implementation within a complete database management system supporting real applications. This future investigation must consider policies for choosing between in-place updates and append-only writes, for logging during page migration, for buffer management, for space reclamation, and for incremental defragmentation using the mechanisms described earlier.

## Acknowledgements

Discussions with Phil Bernstein, David Campbell, Jim Gray, David Lomet, Steve Lindell, Paul Randal, Leonard

Shapiro, and Mike Zwilling have been stimulating, helpful and highly appreciated. Barb Peters' suggestions have improved the presentation of the material.

## References

[BC 72] Rudolf Bayer, Edward M. McCreight: Organization and Maintenance of Large Ordered Indices. Acta Inf. 1: 173-189 (1972).

[BU 77] Rudolf Bayer, Karl Unterauer: Prefix B-Trees. ACM Trans. Database Syst. 2(1): 11-26 (1977).

[C 79] Douglas Comer: The Ubiquitous B-Tree. ACM Comput. Surv. 11(2): 121-137 (1979).

[CAB 81] Donald D. Chamberlin, Morton M. Astrahan, Mike W. Blasgen, Jim Gray, W. Frank King III, Bruce G. Lindsay, Raymond A. Lorie, James W. Mehl, Thomas G. Price, Gianfranco R. Putzolu, Patricia G. Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, Robert A. Yost: A History and Evaluation of System R. Commun. ACM 24(10): 632-646 (1981).

[CGM 02] Shimin Chen, Phillip B. Gibbons, Todd C. Mowry, Gary Valentin: Fractal prefetching $B^+$-Trees: optimizing both cache and disk performance. SIGMOD Conf. 2002: 157-168.

[CLG 94] Peter M. Chen, Edward L. Lee, Garth A. Gibson, Randy H. Katz, David A. Patterson: RAID: High-Performance, Reliable Secondary Storage. ACM Comput. Surv. 26(2): 145-185 (1994).

[ELS 97] Georgios Evangelidis, David B. Lomet, Betty Salzberg: The hB-Pi-Tree: A Multi-Attribute Index Supporting Concurrency, Recovery and Node Consolidation. VLDB J. 6(1): 1-25 (1997).

[G 81] Jim Gray: The Transaction Concept: Virtues and Limitations (Invited Paper). VLDB Conf. 1981: 144-154.

[G 03] Goetz Graefe: Sorting and indexing with partitioned B-trees. Conf. on Innovative Data Systems Research, Asilomar, CA, January 2003.

[HM 00] Dave Hitz, Michael Marchi: A Storage Networking Appliance. Network Appliance, Inc., TR3001, updated 10/2000, http://www.netapp.com/tech_library/3001.html.

[HR 83] Theo Härder, Andreas Reuter: Principles of Transaction-Oriented Database Recovery. ACM Comput. Surv. 15(4): 287-317 (1983).

[JS 89] Theodore Johnson, Dennis Shasha: Utilization of B-trees with Inserts, Deletes and Modifies. PODS Conf. 1989: 235-246.

[L 93] David B. Lomet: Key Range Locking Strategies for Improved Concurrency. VLDB Conf. 1993: 655-664.

[L 95] David B. Lomet: The Case for Log Structuring in Database Systems. HPTS, October 1995. Also at http://www.research.microsoft.com/~lomet.

[LM 03] Bernd Lober, Ulrich Marquard: Anwendungs- und Datenbank-Benchmarking im Hochleistungsbereich von ERP-Systemen and Beispiel von SAP. Datenbank-Spektrum 7: 6-12 (2003). See also http://www.sap.com/benchmark.

[M 90] C. Mohan: ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. VLDB Conf. 1990: 392-405.

[MHL 92] C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, Peter M. Schwarz: ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. ACM Trans. Database Syst. 17(1): 94-162 (1992).

[NB 97] Kjetil Nørvåg, Kjell Bratbergsengen: Write Optimized Object-Oriented Database Systems. Conf. of the Chilean Computer Science Society, Valparaiso, Chile, November 1997: 164-173.

[O 92] Patrick E. O'Neil: The SB-Tree: An Index-Sequential Structure for High-Performance Sequential Access. Acta Inf. 29(3): 241-265 (1992).

[OF 89] John K. Ousterhout, Fred Douglis: Beating the I/O Bottleneck: A Case for Log-Structured File Systems. Operating Systems Review 23(1): 11-28 (1989).

[PGK 88] David A. Patterson, Garth A. Gibson, Randy H. Katz: A Case for Redundant Arrays of Inexpensive Disks (RAID). SIGMOD Conf. 1988: 109-116.

[PP 03] Meikel Pöss, Dmitry Potapov: Data Compression in Oracle. VLDB Conf. 2003: 937-947.

[RO 92] Mendel Rosenblum, John K. Ousterhout: The Design and Implementation of a Log-Structured File System. ACM Trans. Computer Syst. 10(1): 26-52 (1992).

[S 92] Margo I. Seltzer: File System Performance and Transaction Support. Ph.D. thesis, Univ. of California, Berkeley, 1992.

[S 93] Margo I. Seltzer: Transaction Support in a Log-Structured File System. ICDE 1993: 503-510.

[SL 76] Dennis G. Severance, Guy M. Lohman: Differential Files: Their Application to the Maintenance of Large Databases. ACM Trans. Database Syst. 1(3): 256-267 (1976).

[SO 90] Jon A. Solworth, Cyril U. Orji: Write-Only Disk Caches. SIGMOD Conf. 1990: 123-132.

[SS 90] Margo I. Seltzer, Michael Stonebraker: Transaction Support in Read Optimizied and Write Optimized File Systems. VLDB Conf. 1990: 174-185.

[U 84] D. Unger: Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. ACM SIGSOFT/SIGPLAN Software Eng. Symp. on Practical Software Development Environments, Pittsburgh, April 1984.

[WBW 96] Christopher Whitaker, J. Stuart Bayley, Rod D. W. Widdowson: Design of the Server for the Spiralog File System. Digital Technical Journal 8(2): 15-31 (1996).