Efficient verification of B-tree integrity

Goetz Graefe
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94303
Goetz.Graefe@HP.com

Ryan Stonecipher
Microsoft
1 Microsoft Way
Redmond, WA 98052
Ryan.Stonecipher@Microsoft.com

Abstract: The integrity of B-tree structures can become compromised for many reasons. Since these inconsistencies manifest themselves in unpredictable ways, all commercial database management systems include mechanisms to verify the integrity and trustworthiness of an individual index and of a set of related indexes, and all vendors recommend index verification as part of regular database maintenance. This paper introduces algorithms for B-tree validation, reviews the algorithms' strengths and weaknesses, and proposes a simple yet effective improvement for key verification across multiple B-tree levels. The performance is such that B-tree verification can become part of scans or backups. Our experimental comparisons include algorithm performance and scalability measured using a shipping product.

1 Introduction

Even with the most carefully implemented and tested database software, database corruptions can happen at any time. There are many reasons: even shipping software has defects for which vendors eventually offer "fix packs" or "service packs;" there are software defects in operating systems and their device drivers; there is complex hardware and software in today's storage systems, e.g., in network-attached storage; and there are environmental problems such as excessive vibration or overheating in a storage rack. Each of these system layers may have its own verification methods for its data and metadata, but the database management system running on top of them must provide the final verification mechanism that detects failures and corruptions. Therefore, all database vendors provide verification utilities and recommend their use as part of regular maintenance. The purpose of this paper is to lay foundations of reliable and efficient verification algorithms and to introduce a superior new algorithm. This algorithm requires little memory yet is so simple and fast that it even can be integrated into backup and restore operations.

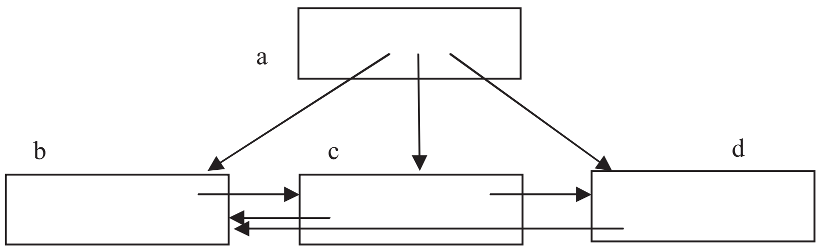

Figure 1. An incomplete leaf split.

For example, Figure 1 shows the result of an incomplete execution, an incomplete recovery, or an incomplete replication of splitting a leaf node. The cause might be a defect in the database software, e.g., in the buffer pool management, or in the storage management software, e.g., in snapshot or version management. When leaf node b was split and leaf node c was created, the backward pointer in successor node d incorrectly remained unchanged. A subsequent (descending) scan of the leaf level will produce a wrong query result, and subsequent split and merge operations will create further havoc.

Different B-tree inconsistencies result if all nodes but c are saved correctly in the database; in that case, a, b, and d point to a page full of garbage. If all nodes but b are saved correctly, the key range in b would not conform to the separator keys in a. If all nodes but a are saved correctly, a search for records in node c will not find them.

There are, of course, many failures and corruptions that may be encountered. A B-tree verification algorithm must be prepared to deal with and identify any of these cases. Our work focuses on the data structures defined and governed by the database management system, ignoring the effects and failure modes of the hardware and software underneath even if the software embedded in the so-called hardware is as large and complex as the database management system itself, e.g., to provide load management and distribution, snapshot backup, geo-replication, or redundancy with online recovery [PGK 88].

In many databases, the most fundamental data structure on disk is a variant of the well-known B-tree index [BM 72]. Therefore, verification of a B-tree's consistency is the fundamental operation we consider in this paper. As database indexes are usually in the form of B*-trees, we focus on those instead of the original B-tree design, and throughout we mean B*-trees when we mention B-trees. In addition, modern database management systems define, explicitly or implicitly, further consistency constraints, e.g., between multiple redundant indexes of a single table or between a materialized view and the base tables. Verification of those relationships is also considered here.

Allocation structures, metadata, and verification of their consistency are omitted here, because these differ too much between products to give such a discussion general value. For the purpose of code reuse alone, including code for consistency checking, B-tree indexes might be used to implement the allocation data structures. In this case, the problem of booting a consistency check arises: if the allocation information is kept in a B-tree, but that B-tree is not verified yet, where should database verification start? Within this paper, we ignore this problem and focus on verification of user tables and their B-tree indexes.

The variety of possible algorithms for B-tree verification is actually surprising. The main issue is verification of the complex web of pointers and keys, and of the many consistency constraints implied in a B-tree's structure. At first sight, nothing but a careful traversal of the tree structure seems safe and complete. If this method were the only possible method, dropping and re-creating indexes could be more efficient than verification.

We have found, however, that orders-of-magnitude faster algorithms can be employed. This can be achieved by dividing the problem into sub-problems, by dividing sub-problems into their contributing facts, and by aggregating such facts and verifying the result. Moreover, relational database systems have built-in mechanisms for implementing such algorithms, namely the query processor. We have found that both query optimization and query execution can contribute to the efficiency of B-tree verification.

Algorithms based on these insights have been shipping and used regularly on millions of production systems for years. Their introduction reduced the times tables had to be read-only or offline and thus increased application availability. Recently, however, we realized that another substantial improvement is still possible. This improvement reduces the required effort to a level that permits B-tree verification as side effects of scans, e.g., during backup and restore operations where B-tree verification would serve to increase the confidence in backup copies and would thus increase their value.

Therefore, the contributions of this research are (i) the introduction, survey and comparison of multiple alternative approaches to verification of structural integrity of entire B-trees, (ii) a description of a fast verification algorithm invented for SQL Server, (iii) the description of actual implementation and usage of multiple algorithms in a commercial product, (iv) identification of opportunities for further improvements of these algorithms, (v) the evaluation of the performance and scalability of alternative algorithms in the context of a shipping product, and (vi) a simpler and faster method for verifying large multi-level B-trees. Automatic repair is beyond the scope of this paper.

After a review of related work, the two main sections of the paper survey alternative algorithmic approaches to B-tree verification, including details on opportunities for improvement, and report on our performance evaluation. We then offer some recommendations and our conclusions from this work.

2 Related work

Several researchers have analyzed frequency and causes of disk errors. For example, Bairavasundaram et al. [BGS 08] found “more than 400,000 instances of checksum mismatches over the 41-month period,” affecting almost 1% of the monitored near-line disks. Even for disk arrays with redundancy, Hafner et al. [HDB 08] observe trends that are “are increasing the likelihood of undetected disk errors”, which “can cause silent data corruption that may go completely undetected (until a system or application malfunction) or may be detected by software in the storage I/O stack.” Our goal is to detect corruptions that manifest themselves as inconsistent B-tree indexes, e.g., those illustrated in Figure 1.

Any paper on B-trees, their structures and their algorithms, owes a debt to early work that defined and popularized B-trees [BM 72, C 79]. Their defining characteristics are uniform root-to-leaf distance, multiple keys and child pointers per interior node, and (in many implementations) sibling pointers among nodes on the same level [GR 93]. Subsequent work has made many important improvements, e.g., compression [BU 77, PP 03].

Unfortunately, verification of physical integrity of databases has not received much attention in database research. Logical verification, both initial verification of new explicitly declared consistency constraints and incremental maintenance of previously declared constraints, has been researched in the past [RSS 96, S 75]. The last one of the techniques described below is related to these prior ones, i.e., referential integrity and foreign key constraints are similar to the relationship between a clustered index and its non-clustered indexes. The other techniques described in the following sections, however, are not related to logical database verification because those prior publications did not consider verification of the tree structure and its many pointers, assuming instead that access methods are unconditionally reliable.

Mohan described the danger of partial writes due to performance optimizations in the SCSI standard [M 95]. His focus was on problem prevention using appropriate page modification, page verification after each read operation, logging, log analysis, recovery logic, etc. The complexity of these techniques, together with the need for ongoing improvements in these performance-critical modules, reinforces our belief that complete, reliable, and efficient verification of B-tree structures is a required defensive measure.

Küspert investigated index verification techniques in the 1980s [K 85], with a focus on local verification as a side effect of ordinary B-tree accesses during query and update processing. For example, page format and contents can be verified before interpreting a database page recently fetched from disk. Page verification may include its relationship to its parent page. Parent and child must belong to successive levels of the same B-tree, and the key values in the child must belong into the key interval defined by the parent. The parent page usually also contains information about a node's neighbors that can be compared to the sibling pointers in a child node during a root-to-leaf B-tree search.

Our techniques are compatible with and complementary to these prior verification techniques. In fact, we present an extension to Küspert's techniques. Our overall goal is different, however, as our techniques permit complete assurance for an entire B-tree or even all B-tree indexes belonging to a table, view, database, or backup.

Prior techniques used in Siemens software reduced verification of B-tree consistency to set comparisons, e.g., the set of B-tree nodes, the set of parent-child pointers, and the sets of left and right sibling pointers [W 81, summarized in K 85]. Expensive set operations were replaced by checksums. Set comparisons are also the conceptual basis of our methods based on aggregation and bitmaps. Our algorithms differ from the earlier ones in two ways, however. First, our algorithms permit gathering much less data in the second pass needed after the first pass has found that a problem exists. Second, our algorithms verify not only the pointer structure but also the ordering relationships of keys among sibling nodes and among ancestor and descendent nodes. Key verification cannot be added to the Siemens algorithms because checksums support only equality comparisons whereas key comparisons also require "less than" comparisons.

Some of the discussion below proposes changes to the traditional page layout for B-trees. Optimization of page layouts, in particular with respect to CPU caches, has been researched in multiple efforts during the last decade. However, these efforts have been focused on CPU caches rather than on the B-tree structure. Prior research has suggested the same change in page layout as is proposed below, but failed to appreciate its importance for B-tree verification despite mentioning the subject [G 04].

SQL Server is used as reference implementation in this paper. Any other product would serve just as well, probably with similar issues, problems, insights, and results. Like other database products, SQL Server stores variable-length records in B-trees with fixed-length nodes or pages. It uses pages of 8 KB, with special techniques for records exceeding this size. An indirection vector within each page indicates the location of records using byte offsets. This indirection vector grows from the high end of the page such that the records may grow from the low end.

Another design decision in SQL Server B-tree pages is that all records in an interior node have the same layout including both a key and a pointer, i.e., the number of keys in an interior node equals the number of pointers. When an interior node is split, entire records are moved from the old node to the new node, such that the new right node contains not only separator keys but also the low boundary key. While this design is common [GR 93], it is not ubiquitous. In SQL Server, it is exploited in a way described later.

Finally, each table may have one clustered index and multiple non-clustered B-tree indexes. The clustered index serves as the primary, non-redundant data store, whereas the non-clustered indexes reference rows in the clustered index physically (using page number and slot number) or logically (using a search key in the clustered index) – SQL Server switched from the former to the latter design choice in release 7.

SQL Server does not employ a technique called B^{link}-trees [LY 81]. For high concurrency while splitting nodes, these trees permit side pointers to new nodes that are not yet referenced in the parent node, with repair at the earliest opportunity. Search is directed to the node at hand or its right neighbor using a separator key associated with the side pointer. This is the same key value eventually posted as separator key in the parent node. Verification of B^{link}-trees is not covered explicitly in this paper, but the algorithms can be adapted with moderate effort and complexity. It is interesting to note, however, that both a low boundary key within each node and a high boundary key within each node are already common or have already been proposed elsewhere. They will be exploited in some of the verification algorithms to be discussed.

3 Verification algorithms

In the algorithms described here, we assume that there are no concurrent transactions updating the indexes and tables being checked. This assumption is true if the verification operation holds a table-level shared lock that covers the metadata, all indexes, etc., or if the verification operation is isolated from concurrent activity by other means, e.g., by copy-on-write semantics implemented within the database or a lower software layer.

We further assume that complete verification is required. All following algorithms are able to verify all constraints in a B-tree index such that successful completion guarantees complete consistency. A database vendor is free to choose among these algorithms based on the desired performance, scalability, engineering effort, etc.

B-tree verification can be divided into multiple aspects. First, each individual page must be consistent, and the first section below describes current and possible techniques that apply to each individual B-tree node. Second, the network of interrelated B-tree nodes must be verified, including child pointers and neighbor pointers as well as key ranges and separator keys. Third, if multiple B-trees store a relational table, these must be verified against each other. For example, a table's clustered and non-clustered indexes must contain the same count of valid records in order to represent the same set of logical rows. Finally, the relationship among tables must be verified, e.g., explicitly declared foreign key constraints, as well as the relationship between tables and materialized and indexed views. The following sections review appropriate algorithms for these problems.

Verification algorithms can produce various forms of output, from a binary decision whether or not a B-tree is currently consistent all the way to a specific identification of each individual problem. Between those extremes fall algorithms that describe the type or approximate location of existing problems. If corruptions are fairly frequent even for small data volumes, specific information is worthwhile even if it is the most expensive to obtain. On the other hand, if corruptions are very rare and a simple binary decision is least expensive to obtain, that type of algorithm has the greatest value, assuming, of course, that identification of specific problems is possible if a corruption exists. The intermediate algorithms trade off performance against output information, such that the output information is less specific but finding the actual corruption, should one be found to exist, is less expensive than the naïve algorithm.

In the following sections, we point out the verification algorithms' most salient aspects, their advantages and disadvantages, their usage of the algorithm in SQL Server, and opportunities for improvements. Discussions of advantages and disadvantages as well as of future opportunities apply to any database management system relying on B-tree indexes. SQL Server is used to provide examples of each algorithm's usage and history rather than convey a limitation to a specific product or release.

3.1 In-page verification

Verification of information within a single B-tree node is fairly straightforward. It is described here in some detail for the sake of completeness. Specific systems and their data structures design require appropriate modifications.

It should start with a physical test to protect against partial or "torn" write or read operations [M 95], i.e., cases in which a B-tree node (e.g., 8 KB) requires multiple disk sectors (e.g., of 512 B) and only some of them are written or read due to a problem in electrical power, vibration, etc. Using a checksum operation of all words in the page or of only a select few ones (say every 512th byte) gives reasonable assurance that the in-memory image read from disk and the one prior to the last write are indeed equal.

Opportunities are plentiful to go beyond the basic algorithm described above. First, torn-page detection can be made sufficiently fast that it is worthwhile to integrate it into all read operations in the database management system. The main expense of torn-page detection is the number of misses in the CPU cache, but even those may be worthwhile if the I/O hardware is feared not to be very reliable. Even further verification steps could be integrated into the I/O operations and might be worthwhile in some systems.

In addition to verification, the same database utility could add general maintenance tasks within each page. If so, database or index verification improves not only confidence in the system's correctness but also performance of subsequent operations. Maintenance tasks can be deeply integrated in the verification task or they can be scheduled to follow the verification tasks, focusing on opportunities identified by the verification utility.

3.2 Index navigation

Having explored in-page verification, we now review alternative algorithmic approaches to verification of a B-tree's structure. For each algorithmic approach, we first focus on verification of a single B-tree and then consider matching multiple B-trees, e.g., a clustered index and its non-clustered indexes. Verification of a B-tree's structure covers the network of pointers among B-tree nodes as well as the ordering of key values among sibling nodes and among parent and child nodes. In database systems using a different variant of B-trees, the following algorithms require adaptation. For example, we assume that leaf nodes as well as interior nodes point to their immediate neighbors by means of page identifiers, with no pointers at the left and right edges of the B-tree. Verification of clustered and non-clustered indexes requires precise one-to-one matches of their records.

The first approach is naïve navigation of the index's structure, in particular of the child pointers and the neighbor pointers. In an ordered B-tree index, a breadth-first traversal is possible but probably not ideal. A depth-first traversal sweeps from the lowest to the highest search key, matching forward and backward pointers as well as key ranges at all B-tree levels. If the B-tree structure is exploited for deep read-ahead for a fast index-order scan, the resulting traversal is really a hybrid of breadth-first and depth-first scans. Of course, a parent node must be verified before its contents may be employed for read-ahead or any other traversal.

The progression through the B-tree's key range is important not only for consistency but also to prevent infinite loops, e.g., among leaf pages if the linkage information is corrupted in an unfortunate way. Massive sequences of duplicate keys in non-unique indexes could foil this safety, but both clustered and non-clustered indexes actually contain only unique entries, if necessary by use of artificial "uniquifier" fields in clustered indexes or by inclusion of the clustered index's search key in non-clustered indexes and their sort order. Such uniqueness is required, for example, for accurate deletion of B-tree entries.

Performance of index navigation will be linear with the number of pages allocated for the B-tree, with very modest demands on the buffer pool. Only single-page read operations can be employed if the "next leaf" pointer is used to scan through the leaf pages, whereas appropriate use of the parent nodes enables large, multi-page read operations with the attendant efficiency improvement. This efficiency improvement can be realized only if the B-tree's layout within the database is not fragmented due to bad algorithms during index creation or due to many insertions and page splits without subsequent B-tree defragmentation. In addition to the large read operations, parent nodes also enable multiple asynchronous read operations, which are helpful in a striped storage organization.

Multiple indexes for a table, say a clustered index and a few non-clustered indexes, must first be verified for internal consistency and then entries in each non-clustered index must be matched against the clustered index. Obviously, the number of records must be the same in each index; in addition, these records must actually match one-to-one. Moreover, an index's tree structure must be verified before the verification algorithm should attempt a key search in the index.

Compared to alternative algorithms, the main advantage of this algorithm is its simplicity and code reuse. Using the same code that ordinary database operations use for index access ensures that the code is tested thoroughly. Given that the main purpose of index verification is certainty that the database is correct, and given that index verification defends against many errors that include software defects in the database management system, the choice of this algorithm is a tradeoff strictly for correctness and dependability. Other advantages include very moderate demands on the buffer pool as well as notification of the error very quickly after the B-tree node containing the error is read.

On the other hand, this algorithm suffers from poor I/O performance including many repeated read operations for each page; thus, its overall performance and scalability are poor unless the entire table fits into the available buffer pool. In other words, the algorithm is still attractive for in-memory databases including caches on mid-tier or client machines, even if a small fraction of the data might have spilled to disk.

SQL Server used to depend on a variant of this algorithm for index verification. The original design target in the mid 1980s had been 16-bit machines, so small memory requirements were very important. The algorithm is still in use today, but only after the current algorithm for matching non-clustered indexes against the clustered index has found the existence of an error but not its precise location, as described later.

Opportunities exist to improve the above basic approach should that be desired. For example, multiple indexes can be verified independently and thus concurrently. Depending on relative CPU and I/O bandwidth, this can speed up the tree verification phase by a small factor, assuming that the buffer hit rate is not materially affected by this concurrency. During the matching phase, read-ahead in the non-clustered index provides a small performance advantage. In the clustered index, B-tree searches can employ asynchronous prefetch by pursuing multiple searches concurrently. On a machine with more disk arms than execution threads, this can improve performance of the match phase also by a small factor. Finally, the matching operation between clustered and non-clustered indexes is rather similar to join operations. The simplest algorithm employs essentially index nested loops join; alternative join algorithms include merge join and hash join.

3.3 Aggregation of facts

An entirely different approach is needed in order to support terabyte databases. The following method gathers the same facts and verifies the same consistency constraints but separates the verification from the read operations. In other words, as data pages are read and information is extracted, it is not verified immediately. Instead, facts are extracted and streamed into a matching algorithm, e.g., "a leaf node on disk page 5 points to a successor leaf node on disk page 92." When a matching fact is found, e.g., "a leaf node on disk page 92 points to a predecessor leaf node on disk page 5," verification for these two facts is successful. At the end of the entire matching operation, the B-tree is consistent if and only if all facts have been matched.

While this approach might seem awkward at first, it offers several advantages. The most significant advantage is that B-tree pages are read only once rather than many times as in the index navigation method above. The I/O reduction may exceed two orders of magnitude. Moreover, B-tree pages may be read in any order and disk-order scans are acceptable. Depending on the fragmentation status of the B-tree, large read operations may contribute another order of magnitude to the speed. The usual optimizations apply just as in ordinary query execution, including asynchronous read-ahead, striping, etc.

The crucial component of this approach is in the selection and representation of facts extracted from B-tree pages and matched in the verification step. For the chain of neighbors, the fact "Page x follows page y" is extracted from both pages x and y. For the parent-child relationship, "Parent x points to child y for key range [a, b)" is extracted from parents and children. These matches also verify the level of the two pages (leaf pages are level 0) and permit the appropriate flexibility in matching separator keys in the parent and actual keys in the child. For the root page, one special fact is generated to provide an artificial parent; alternatively, the appropriate catalog entry and its pointer to the B-tree's root might provide this fact.

For example, consider the B-tree in Figure 2 with root page a and two leaves b and c. Shaded areas represent records and their keys; two keys are different (equal) if their shading is different (equal). Eight facts will be extracted from the B-tree nodes in Figure 2:

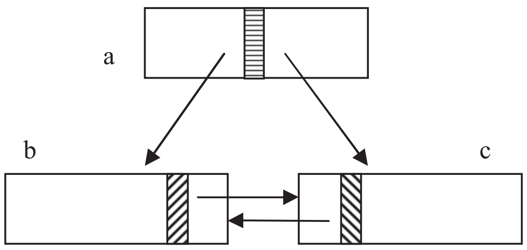

Figure 2. Root and two leaves.

1. From page b, the fact that b is a leaf page, together with its actual key range
2. From page b and its successor pointer, the fact that c follows b
3. From page c, the fact that c is a leaf page, together with is actual key range
4. From page c and its predecessor pointer, the fact that c follows b – this matches with fact 2 above
5. From page a, the fact that b is a leaf page, together with its permissible key range, which is open at the low end – this matches with fact 1 above
6. From page a, the fact that c is a leaf page, together with its permissible key range, which is open at the high end – this matches with fact 3 above
7. From page a, the fact that a is a level-1 node, together with its actual key range (a singleton value)
8. An artificial fact to match fact 7 – this fact could also be derived from the catalogs

With no sibling pointers along the left and right edge of the B-tree, no facts are generated there. The key ranges include information about open and closed ranges, as appropriate. Facts verifying sibling pointers could be augmented with key information, enabling the aggregation to verify that indeed all keys in the predecessor are lower than those in the successor. However, this relationship is already verified transitively via the separator key in the parent. In other words, no additional information or confidence is gained by comparing the two leaf records (shaded diagonally in Figure 2) with each other in addition to comparing each of them with the separator key in the root (shaded horizontally).

Cousin nodes in a B-tree with multiple levels are neighboring nodes with no shared parent but instead a shared grandparent or even higher ancestor. Cousin nodes gives rise to what we call the "cousin problem," illustrated in Figure 3. Verification of keys and pointers among cousin nodes does not have an immediate or obvious solution. Nonetheless, two alternative solutions with different performance characteristics are discussed below, one employed today and a novel one that is both simpler and faster.

The essence of this problem as shown in Figure 3 is that the key separating leaves d and e is not in a shared parent node but in a higher ancestor node, in this case in root a. The potential problem is that there is no easy way to verify that all keys in page d are indeed smaller than the separator key in page a and that all keys in page e are indeed larger than the separator key in page a. Correct key relationships between neighbors (b-c and d-e) and between parents and children (a-b, a-c, b-d, c-e) do not guarantee correct key relationships across skipped levels (a-d, a-e). Two alternative solutions for the cousin problem will be discussed shortly.

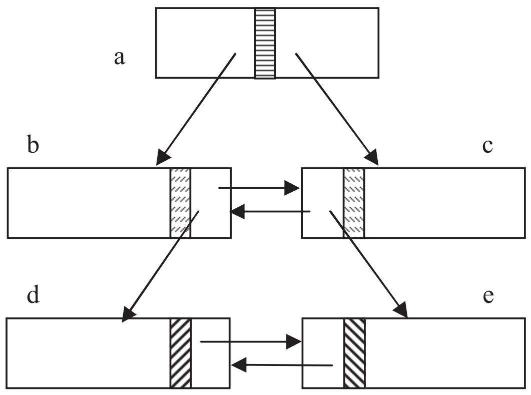

Figure 3. Cousin nodes.

Performance of verification by fact aggregation is proportional to the number of facts derived from a B-tree and thus to the number of pointers or relationships in the B-tree. Each node has a parent, and each node (or only each leaf) has a sibling, with exceptions for the root and for nodes along the left and right edge of the B-tree. Thus, the number of facts is about four times the number of nodes or pages in the B-tree.

Extracting these facts from data pages represents a substantial data reduction. For example, if each fact requires 20-40 bytes of space, 4 facts per B-tree node require 80-160 bytes. Thus, for data pages of 8 KB, the data required for verification of page linkages in a B-tree amounts to 1-2% of the data space.

The data volume of the facts also depends on the method employed to group the individual facts derived from individual pages. Perhaps the most obvious method groups facts by type and the pair of page numbers. For example, in Figure 2, facts 2 and 4 are of the sibling type and may be grouped on the pair (b, c). Similarly, facts 1 and 5 are of the parent-child type and may be grouped on the pair (a, b). This method permits very fine partitioning of all facts as it produces multiple distinct fact records from each page.

An alternative method defines each fact as primarily about one page, not about a pair of pages, and groups facts by this primary page only. For example, in a parent-child relationship, the child may be defined as the primary page, such that facts 2 and 4 above are both considered facts about page c. Similarly, the right sibling in a sibling relationship may be defined as the primary page. In Figure 2, facts 3 and 6 would be considered facts about page c. This technique saves memory by creating fewer facts records. For example, when inspecting page c, facts 3 and 4 above generate only a single aggregation record rather than two. The matching logic must require two matching facts for such a combined fact. Given that most pages in a B-tree are both another node's child and another node's right sibling, this technique immediately reduces the data volume for the aggregation operation, e.g., the size of the hash table for hash aggregation.

For the matching step, all algorithms for "group by" and "distinct" queries can be applied, including sort- and hash-based algorithms [G 93], as the essence of matching is to group and combine all detail facts about each page. The complexity of the entire procedure is *O (N log N)*, i.e., the same as sorting the fact data.

Compared to alternative algorithms, this approach are that it scales reasonably well yet produces precise error messages. Other advantages include sharing code with query execution, in particular code for disk-order scans, sort- and hash-based aggregation, pipelining, parallelism, and memory management.

SQL Server, starting with release 7, verifies the structural integrity of B-trees using a version of this approach with dynamic generation of SQL syntax and using language extensions reserved for such internal queries.

The product also employs some optimizations in the algorithm's implementation, and the information laid out in the explanation of Figure 3 is aggregated using the fewest possible individual fact records. For example, information about both the relationships c-e and d-e in Figure 3 is considered information about node e and aggregated as such.

In combination with the design choice to keep the same number of keys and pointers in each interior node, this optimization also permits solving the cousin problem. Recall that, in a split of an interior node, the new right node retains the low key; in Figure 3, node c contains a copy of the separator key in node a. The cousin problem is solved by aggregating both the low key of node c and the high key of node d into the information about node e, i.e., by ensuring the correct order between nodes c and d. Verification of "second cousins" (also third, etc.) in multi-level B-trees is possible because separators keys are copied during node splits at any level in the B-tree.

If multiple SQL Server indexes are verified in a single pass, their allocation information is aggregated in a single disk-order scan. The facts retained from the allocation information include which page belongs to which B-tree, and verification of a page's information about the B-tree it belongs to is part of the matching process.

Opportunities for improvement are many. For example, the sibling information can be augmented with information about space utilization within pages. If two neighboring pages are found to be nearly empty, an opportunity for merging and thus space reclamation exists. In the opposite case, two pages that are nearly full can be split into three pages with more standard space utilization, and two neighboring pages with a large difference in space utilization may benefit from some movement and load balancing.

Facts can be eliminated as soon as all matching facts are found. In a disk-order scan, success in early aggregation indicates a high quality of clustering, whereas poor success in early aggregation indicates a need for defragmentation for faster index-order scans such as range queries. The opportunity is to focus defragmentation on specifically those pages where it can achieve the largest difference for scan performance.

Fence keys are a technique that can help with the cousin problem illustrated in Figure 3, even if they were originally motivated by write-optimized B-trees [G 04]. The essential difference to traditional B-tree designs is that page splits not only post a separator key to the parent page but also retain copies of this separator key as high and low "fence keys" in the two post-split sibling pages. Note that separators and thus fence keys can be very short due to prefix and suffix truncation [BU 77]. These fence keys take the role of sibling pointers, replacing the traditional page identifiers with search keys.

Figure 4 illustrates the concept of fence keys, and how they replace the sibling pointers of Figure 3. As before, areas with equal shading indicate equal key values. The focus of the discussion is on the one key value with five copies shaded horizontally. Pages d and e are cousins because they belong to the same B-tree level and share a fence key value but not a parent. In order to navigate from page d to its successor, a search from the root page is required. However, such navigation steps will rarely be required because the fence keys are lockable resources; thus, key range locking never needs to navigate to a sibling leaf in search of a lockable key value. Large scans, on the other hand, need to employ parent and grandparent nodes in order to achieve deep, many-page read-ahead.

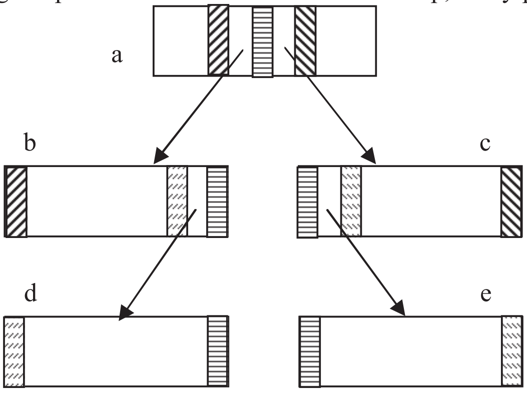

Figure 4. Fence keys.

The important benefit here is that verification is simplified and the cousin problem can readily be solved, including "second cousins", "third cousins", etc. in B-trees with additional levels. In Figure 4, the following four pairs of facts can be derived about the key marked by horizontal shading, each pair derived independently from two pages.

1. From page a, the fact that b is a level-1 page, and its high fence key
2. From page a, the fact that c is a level-1 page, and its low fence key
3. From page b, the fact that b is a level-1 page, and its high fence key – this matches with fact 1 above
4. From page b, the fact that d is a leaf page, and its high fence key
5. From page c, the fact that c is a level-1 page, and its low fence key – this matches with fact 2 above
6. From page c, the fact that e is a leaf page, and its low fence key
7. From page d, the fact that d is a leaf page, and its high fence key – this matches with fact 4 above
8. From page e, the fact that e is a leaf page, and its low fence key – this matches with fact 6 above

The separator key from root a is replicated along the seam of neighboring nodes all the way to the leaf level. Equality and consistency are checked along the entire seam and, by transitivity, across the seam. Thus, fence keys also solve the problem of second and third cousins etc. in B-trees with additional levels.

Beyond solving the cousin problem, i.e., enabling verification of neighbors that do not share a parent, the list above requires only equality comparisons between page identifiers, keys, and level information derived from different pages. In contrast, the verification in Figure 3 requires "less than" comparisons, as does the solution employed in SQL Server.

Complete reliance on equality comparisons enables the next and most efficient algorithmic approach to B-tree verification, discussed shortly.

Finally, the list above indicates that only a single record format and aggregation operation is required to capture the facts required for B-tree verification. This record format includes page identifier, B-tree level, a key value, and an indication whether the key value is a low fence key or a high fence key. Thus, fence keys simplify B-tree verification not only on a conceptual level but also on this detailed implementation level.

Fence keys also extend the local online verification techniques described by Küspert [K 85]. In traditional systems, neighbor pointers can be verified during a root-to-leaf navigation only for siblings but not for cousins, because the verification of a leaf's cousin pointer would require an I/O operation to fetch the cousin's parent node (also its grandparent node for a second cousin, etc.). Thus, Küspert's technique cannot verify all correctness constraints in a B-tree, no matter how many search operations perform verification. Fence keys, on the other hand, are equal along entire B-tree seams, from leaf level to the ancestor node where the key value serves as separator key. A fence key value can be exploited for online verification at each level in a B-tree, and an ordinary root-to-leaf B-tree descent during query and update processing can verify not only siblings with a shared parent but also cousins, second cousins, etc. Two search operations for keys in neighboring leaves verify all B-tree constraints, even for cousin leaves, and search operations touching all leaf nodes verify all correctness constraints in the entire B-tree.

3.4 Bit vector filtering

In addition to these advantages of fence keys over traditional neighbor pointers, fence keys and the complete reliance on equality comparisons during B-tree verification enable a further simplification that reduces the complexity of the entire operation to a single scan. The complexity of this new method has no logarithmic component. Moreover, memory consumption is very moderate and can be fixed independently of the data size.

The basic idea of this approach is quite simple, and readily applies to the pairs of facts derived from Figure 4. Instead of precisely aggregating facts, this algorithmic approach employs a bitmap to verify that facts derived from the on-disk data indeed match up correctly. For each B-tree node and each child pointer in the on-disk data structures, the combination of index identifier, page identifier, B-tree level, low and high fence keys is hashed to a bit position and the bit value currently in that position is reversed. At the end, the entire bitmap should be back in its original state. If it is not, the scanned on-disk data contains an error or corruption. The precise type and location of that error are not known, however. In that sense, it is the first algorithm in this survey that truly only verifies the correctness rather than pinpoints the error or corruption.

There are multiple assumptions underlying this approach. First, on-disk corruptions must be rare; otherwise, this approach is not promising. Second, double-errors also must be rare, and the probability of errors hidden due to hash collisions of double errors must be negligible. Third, it must be more important to speed up those verification runs that find nothing amiss than to ensure that corruptions are pinpointed in the first attempt. Fourth, facts must match in even numbers, typically pairs. Fifth, hashing key values to bit positions supports only equality predicates, not range predicates.

If an on-disk corruption is found to exist, a second pass must find it before it can be repaired. This pass needs to read the same data again, but it only needs to investigate those facts that map to bit positions in the bitmap indicating a failed match. This investigation may employ either of the prior algorithms, i.e., either index navigation starting from each fact that maps to a non-matching bit position or an aggregation of all facts that match to such bit positions.

Compared to other algorithms, the advantage of using bitmaps instead of aggregation is higher performance with less memory, because neither sorting nor a hash table is required. Moreover, the entire process completes in a single scan of the data pages. The main disadvantages are twofold. First, it only works for equality predicates, not range predicates. Thus, it can be applied to B-trees with fence keys but not to traditional B-trees that require "less than" comparisons during verification. Second, the method does not produce precise error information. If an error is found to exist, a second pass over the data must find the precise error and its location. Minor disadvantages are that facts have to match up in even numbers and that there is a theoretical possibility, although with very low probability, of errors hidden due to hash collisions.

SQL Server employs bitmaps for matching non-clustered indexes against clustered indexes. Bit vector filtering is faster with much less memory than aggregation for the same problem. A typical size for the bit vector filter is 32 KB or ¼ M bits. If an error is found, index navigation is used to isolate the fault. Bit vector filtering is not used for verification of B-tree structures, because SQL Server does not employ fence keys in B-tree nodes and leaves. It is not employed for verification of sibling pointers or child pointers because those facts are verified together with the appropriate key relationships.

Opportunities exist for further improvement. For example, a bitmap might not be necessary. Instead, a single integer checksum might suffice. On the other hand, the improvement in performance and scalability due to a checksum rather than a bitmap might not be worth the increased probability of failure to detect a corruption and the increased search effort if an error is found to exist.

Multiple bitmaps permit isolating the type of error, e.g., parent-child relationships versus sibling relationships. Separate bitmaps also permit isolating the location of the corruption, e.g., a bitmap for each B-tree.

The most important opportunities, however, might be in the algorithm's usage rather than in the algorithm itself. Because of its single-scan behavior and its low cost, the algorithm can be integrated into backup and restore operations, database mirroring, and similar database utilities that scan or copy entire tables or even entire databases. Verification of back-up data, both while creating a backup and while restoring it, increase users' confidence in the backup and thus the value of the backup data. Today's standard method is to run a verification step prior to a backup. Integration of verification and backup into a single scan cuts the entire operation's elapsed time in half.

3.5 Query evaluation

The final algorithmic approach relies even more on the database system's query processing engine than the prior ones. The basic idea in this approach is to formulate a query to find violations and to report the query output as database corruptions in need of repair.

This approach is also employed in many data definition statements, e.g., when a new integrity constraint is explicitly defined by a database administrator.

This approach to database verification is most appropriate when verification of the on-disk data structures requires traditional query operations, e.g., when comparing a materialized and indexed view with its base tables, when deriving a computed column using arithmetic and functions, and when extracting index keys from user-defined types.

Compared to the other algorithmic approaches for index verification, this algorithmic approach to database verification can evaluate complex expressions. It reuses existing code including query optimization and parallel query execution. Its main disadvantage is its dependency on a large volume of complex code. In addition, unless the query processing implementation supports optimization and execution of multiple queries together, multiple scans of the same data might be needed. Finally, some query optimizers suffer from somewhat unpredictable plan choices and thus unpredictable performance, which may be considered particularly undesirable for utilities that take entire tables or even an entire database offline.

SQL Server relies on this approach for "not null" constraints and for "check" constraints, exploiting indexes if possible. It also employs this approach for comparing materialized views with query results, and for indexes on user-defined types. The final verification step uses query logic, not bitmaps.

Opportunities for improvement remain, of course. For example, the same code should be employed during verification of data definition statements and during verification on on-disk database contents, even if these two functions have traditionally been implemented by different teams within a large database development team. A second opportunity is that verification of an entire database and all its structures is a complete, well-structured workload that can be planned as a whole, including temporary materialized and indexed views that can be shared for multiple verification steps.

4 Performance evaluation

In order to assess performance and scalability of each algorithm, we ran a number of tests using the algorithms implemented in SQL Server 2005. Figure 6 shows the performance of various index verification tests relative to the scan performance. The data are the "orders" and "line items" tables in a TPC-H database with 10 GB of data plus indexes. The test hardware is a workstation with a 3 GHz dual-core Intel CPU, 2 GB RAM, and a single SATA disk drive. Each test starts with a cold I/O buffer. The tables in this test, orders and line items, each had a clustered index on a date column plus a non-clustered index on the primary key and on each foreign key. Each test includes a variant with parallelism disable and enabled. Each test is shown as a set of 4 bars.

The 1st set of bars in Figure 6 (marked "clustered index scan") shows the performance of "select count (*)" queries, which measure primarily the scan performance of the I/O hardware. Using query optimization hints, we forced a disk-order scan of the entire clustered index. By definition, each bar shows 100%, because we use this query to show the relative performance of more complex operations. Proportional to the data volumes, the actual execution times for operations against the "line item" table are about four times longer than those against the "orders" table.

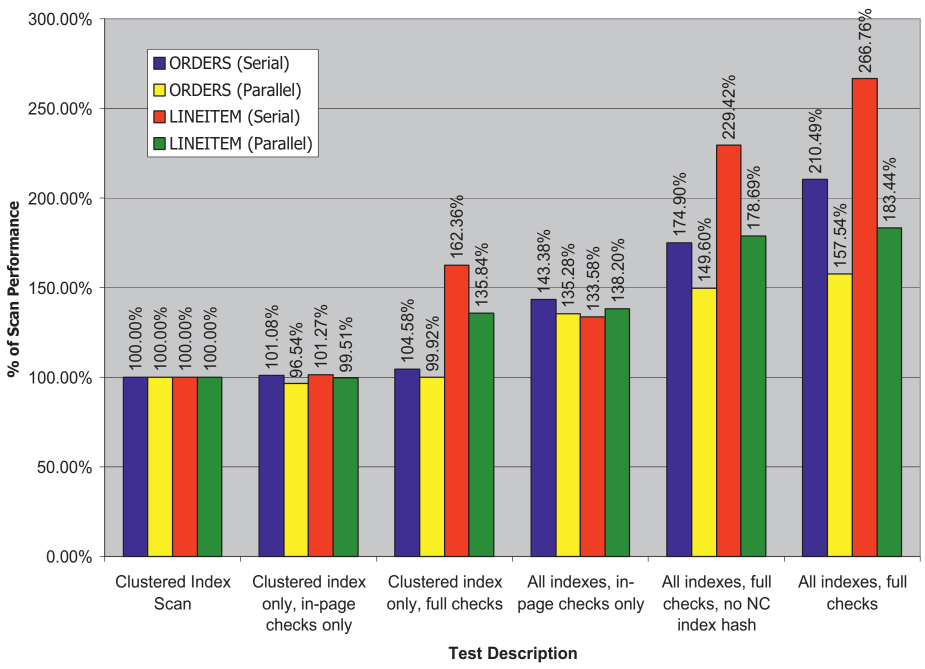

Figure 6. Algorithm performance.

The 2nd set of bars in Figure 6 (marked “clustered index only, in-page checks only”) illustrates the overhead of in-page consistency verification. If B-tree verification is limited to in-page checks, the CPU can readily keep up with the I/O hardware. Thus, the total execution time for in-page-only verification is practically equal to that of the “select count (*)” query in all 4 cases. Thus, the in-page checks outlined in Section 3.1 can be achieved with acceptable effort. This argues for performing such checks not only during offline verification but also online during normal query and update processing, as already proposed by Küspert [K 85] and Mohan [M 95].

The 3rd set of bars in Figure 6 (marked “clustered index only, full checks”) illustrates the cost of full structural verification of a B-tree. For verification of all aspects of the clustered index B-tree fact aggregation as described in Section 3.3 is employed and adds to the execution times. Due to efficient fact extraction and aggregation, this operation can keep up with a disk-order scan, at least for the “orders” table.

The performance for the “orders” table is almost equal to the “select count (*)” query, whereas the performance for the “line items” table is not. The reason is the difference in the table sizes. While the facts extracted from the “orders” table can be sorted in the server’s allocated memory, the “line items” table is four times larger and sorting the extracted facts requires I/O to temporary run files. Without sufficient memory, even moderate sizes of tables and indexes incur the penalty of I/O to temporary files.

As this example illustrates, the performance and scalability of sort operations dominates the complexity, performance, and scalability of B-tree verification by fact aggregation. Parallelism and multiple CPU cores speed up the verification task, because a single CPU core cannot perform in-page verification, fact extraction, and sorting with the same bandwidth as the disk drive. However, parallelism adds its own overhead, too, and using both CPUs fails to cut elapsed times in half.

The 4th set of bars in Figure 6 (marked "all indexes, in-page checks only") illustrates the additional data size due to non-clustered indexes. For in-page verification of both clustered and non-clustered indexes, execution times rise beyond those of the "select count (*)" query and the clustered index in-page verification by a fairly uniform amount. In the experimental database, the size of the non-clustered indexes relative to the clustered index is about 40% for the "orders" table and about 35% for the "line items" table.

The 5th set of bars in Figure 6 (marked "all indexes, full checks, no NC index hash") illustrates the cost of B-tree verification for multiple indexes in a single operation. All a table's B-trees are checked completely but indexes are not matched against one another. Thus, this experiment shows the same task as the 3rd set of bars but for an increased data volume. The performance difference between the 4th and 5th sets of bars is quite similar to the performance difference between the 2nd and 3rd sets of bars. For example, the difference between 162% and 229% is 41% (229/162=1.41), appropriate for the data volume.

The results discussed so far enable a few conclusions. First, verification of a B-tree structure using fact aggregation is many times faster than a traditional method based on index navigation. Second, due to its foundations in query processing and efficient sorting, verification using fact aggregation scales very well. Third, verification of a B-tree's structure is still the most expensive aspect of B-tree verification.

The 6th set of bars in Figure 6 (marked "all indexes, full checks") illustrates the cost of bitmap processing. As readily apparent in a comparison to the 5th set of bars, matching records in clustered indexes with records in non-clustered indexes using bitmaps adds only about 12-20% to the execution times; much less if two CPU cores are employed. This is a low cost for the added value; the best alternative a join for each non-clustered index. It is important to realize that the bitmap operation requires only a small amount of memory, and that this amount is virtually independent of the data size.

This low additional CPU load due to bitmap processing (in this case, for cross-index verification) also points to further possible gains. Both CPU load and overall verification performance would improve if cross-page verification within each B-tree would use bitmaps instead of sorting and aggregation of fact records, i.e., would use the new B-tree verification algorithm proposed in this paper. Instead of adding 35-40% to the cost of the data scan, cross-page verification would add merely about 1-2%. It would add less than the 12-20% observed above for bitmap processing because there are merely 2 facts per B-tree node, not tens or hundreds of keys per leaf. Moreover, verification efforts for larger databases and indexes would increase linearly rather than with the *O (N log N)* complexity of sorting, and it would avoid entirely any I/O to temporary run files.

To summarize the performance observations, B-tree verification can be performed at I/O speeds, including complete verification of all links and relationships within each B-tree index and of the relationships of each non-clustered index and the clustered index. The introduction of fact extraction and aggregation has been a tremendous improvement over the prior verification algorithm based on index navigation. The cost of bitmap processing (as observed for cross-index verification) indicates that reliance on bitmaps also for cross-page verification could further improve performance and scalability. It could enable streaming verification not only during scans but also during backup and restore operations, during database mirroring, and any other database utilities that essentially perform copy operations of databases or tables.

5 Summary and conclusions

In summary, consistency verification within and between B-tree indexes is as important as solid concurrency control and dependable backup. Production users invoke consistency verification weekly or even daily in order to guard against data loss due to problems in their hardware or software, from SAN management software and networking to device drivers and the database management system itself. Moreover, verification of on-disk data structures is invoked even more frequently during development and testing of database software [WY 95], where high performance and good scalability directly contribute to shortening the testing effort and thus the development schedule.

Intra-page verification is fairly straightforward, although further opportunities for innovation have been identified. For inter-page verification, there is a surprising variety of algorithmic approaches. For pragmatic reasons that each seemed decisive at their time, the developers of SQL Server have, over the course of more than 20 years, taken different approaches to different aspects of the overall problem. Other database products have gone through similar histories. In this paper, we have reviewed these approaches, compared their strengths and weaknesses, and evaluated their performance inasmuch as possible with a product that cannot readily be modified for such investigations.

Based on these improvements, database verification proceeds at nearly the speed of scans. Bandwidth above a gigabyte per minute can be observed on ordinary workstations using fact aggregation within B-trees and bit vector filtering across indexes. A verification task that takes minutes today used to take hours or even days using index navigation.

Among the new insights of this research, the most important is the advantage of fence keys for efficient B-tree verification. In addition to their benefits explored in earlier work [G 04], they permit complete and correct verification of all relationships among B-tree nodes and their keys. In particular, as illustrated in Figure 4, they easily solve the cousin problem illustrated in Figure 3, including second cousins, third cousins, etc. Moreover, due to the avoidance of all “less than” comparisons in cross-page checks, fence keys permit replacing the current algorithm based on sorting and aggregation with an algorithm based entirely on bit vector filtering. The resulting algorithm reduces both processing effort and memory requirements for efficient index verification in very large databases. Thus, this improved algorithm promises to keep up with the highly tuned I/O mechanisms used in backup and restore, a capability database owners demand in order to avoid a separate verification step for their backup media.

Many further extensions and improvements of the algorithms described here are possible. Obvious extensions include generalizations for partitioned and shared-nothing architectures, automatic repair of corrupted on-disk data structures, coverage of data duplication and replication including database replication and mirroring, and coverage of other storage formats in addition to traditional B-tree indexes. Such other formats range from unordered heap structures and storage structures for large binary objects to bitmap indexes and column stores. Further possibilities include verification of complex values including XML objects and user-defined types, online verification during active transaction processing, and incremental verification exploiting short periods of low user activity.

6 Acknowledgements

This report would not have been possible without the contributions by the inventors of Microsoft's patents on B-tree and database verification. Mike Zwilling provided helpful suggestions on the presentation of the material. Ahmed Ezzat and Gottfried Vossen suggested excellent improvements in the presentation of our work.

7 References

[BD 83] Dina Bitton, David J. DeWitt: Duplicate Record Elimination in Large Data Files. ACM Trans. Database Syst. 8(2): 255-265 (1983).

[BGS 08] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau: An analysis of data corruption in the storage stack. FAST 2008: 223-238.

[BM 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).

[G 93] Goetz Graefe: Query Evaluation Techniques for Large Databases. ACM Comput. Surv. 25(2): 73-170 (1993).

[G 04] Goetz Graefe: Write-Optimized B-Trees. VLDB Conf. 2004: 672-683.

[GR 93] Jim Gray, Andreas Reuter: Transaction Processing: Concepts and Techniques. Morgan Kaufmann 1993.

[HDB 08] James L. Hafner, Veera Deenadhayalan, Wendy Belluomini, Krishnakumar Rao: Undetected disk errors in RAID arrays. IBM Journal of Research and Development 52(4-5): 413-426 (2008).

[BM 70] Rudolf Bayer, Edward M. McCreight: Organization and Maintenance of Large Ordered Indexes. SIGFIDET Workshop 1970: 107-141.

[K 85] Klaus Küspert: Fehlererkennung und Fehlerbehandlung in Speicherungsstrukturen von Datenbanksystemen. Informatik Fachberichte 99. Springer 1985.

[LY 81] Philip L. Lehman, S. Bing Yao: Efficient locking for concurrent operations on B-trees. ACM Trans. Database Syst. 6(4): 650-670 (1981).

[M 95] C. Mohan: Disk Read-Write Optimizations and Data Integrity in Transaction Systems Using Write-Ahead Logging. ICDE 1995: 324-331.

[MH 94] C. Mohan, Donald J. Haderle: Algorithms for Flexible Space Management in Transaction Systems Supporting Fine-Granularity Locking. EDBT Conf. 1994: 131-144.

[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).

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

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

[RSS 96] Kenneth A. Ross, Divesh Srivastava, S. Sudarshan: Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. SIGMOD Conf. 1996: 447-458.

[S 75] Michael Stonebraker: Implementation of Integrity Constraints and Views by Query Modification. ACM SIGMOD Conf. 1975: 65-78.

[W 81] Christian Weber: Ein Verfahren zur schnellen Konsistenzprüfung von Datenbanken. Angewandte Informatik 23(11): 497-501 (1981).

[WY 95] D. Wildfogel, Ramana Yerneni: Efficient Testing of High Performance Transaction Processing Systems. VLDB 1997: 595-598.