Sorting And Indexing With Partitioned B-Trees

Goetz Graefe
Microsoft Corporation
Redmond, WA 98052-6399
USA
GoetzG@Microsoft.com

Abstract

Partitioning within a B-tree, based on an artificial leading key column and combined with online reorganization, can be exploited during external merge sort for accurate deep read-ahead and dynamic resource allocation, during index creation for a reduced delay until the first query can search the new index, during data loading for streaming integration of new data into a fully indexed database, and for miscellaneous other operations. Despite improving multiple fundamental database operations using a single basic mechanism, the proposal offers these benefits without requiring data structures or algorithms not yet supported in modern relational database management systems. While some of the ideas discussed here have been touched upon elsewhere, the focus here is on re-thinking the relationship between sorting and B-trees more thoroughly, on exploiting this relationship to simplify and unify data structures and algorithms, and on gathering comprehensive lists of issues and benefits.

Introduction

Even the most advanced data models rely on very traditional data structures and algorithms for storing and managing records, including efficient query and update processing. Thus, there is a continuous stream of research into improvements to these data structures, these algorithms, and their usage. Among the perpetually interesting data structures in database systems is the B-tree [BM 72] and its many variants, and among the perpetually interesting algorithms is external merge sort. Sorting is used to build B-tree indexes efficiently, and B-trees are used to avoid the expense of sorting and to reduce the expense of searching during query processing – however, the mutually beneficial relationship between sorting and B-trees can go substantially further than that.

The present paper proposes not a new data structure or a new search algorithm but an adaptation of well-known algorithms and of a well-known data structure. The essence of the proposal is to *add an artificial leading key column to a B-tree index*. If only a single value for this leading B-tree column is present, which is the usual and most desirable state, the B-tree index is rather like a traditional index. If multiple values are present at any one point in time, which usually is only a transient state, the set of index entries is effectively partitioned. It is rather surprising how many problems this one simple technique can help address in a database management product and its real-world usage.

Let us briefly consider some example benefits, which will be explained and discussed in more detail in later sections of this paper.

First, it permits putting all runs in an external merge sort into a single B-tree (with the run number as artificial leading key column), which in turn permits improvements to asynchronous read-ahead and to adaptive memory usage. Given the trend to remote disks, e.g., in SAN and NAS environments, hiding latency by exploiting asynchronous read-ahead is important, and given the continued trend to striped disks, forecasting multiple I/O operations is gaining importance. Similarly, given the trend to extremely large online databases, the ability to dynamically grow and shrink resources dedicated to a single operation is very important, and the proposed changes permit doing so even to the extremes of pausing an operation altogether and of letting a single operation use a machine's entire memory and entire set of processors during an otherwise idle batch window.

Second, it substantially reduces by at least a factor of two the wait time until a newly created index is available for query answering. While the initial form of the index does not perform as well as the final, fully optimized index or a traditional index, at least it is usable by queries and permits replacing table scans with index searches. Moreover, the index can be improved incrementally from its initial form to its final and fully optimized form, which is very similar to the final form after traditional index creation. Thus, the final indexes are extremely similar in performance to indexes created offline or with traditional online methods; the main difference is cutting in half (or better) the delay between a decision to create a new index and its first beneficial impact on query processing.

Third, adding a large amount of data to a very large, fully indexed data warehouse so far has created a dilemma between dropping and rebuilding all indexes or updating

Proceedings of the 2003 CIDR Conference

all indexes one record at a time, implying random insertions, poor performance, a large log volume, and a large incremental backup. The present proposal resolves this dilemma in most cases. Note that it does so without special new data structures. Recently proposed approaches to this problem have relied on adding a special separate lookup structure in main memory, or on retaining records waiting to be pushed down within an index tree by dividing each B-tree node into a segment with traditional key-pointer pairs and another segment with waiting records. Special or novel data structures and algorithms can have enormous costs for real-world database systems, first in development and testing, then when installing the new release and reformatting large production databases, and finally for training staff in application development and in operations; all this not only for the core database system but also for relevant third-party add-on products for capacity planning, tuning, operations, disaster preparedness and recovery, monitoring, etc.

After a brief summary of related research, the remainder of this paper first describes precisely how to manage partitions within B-trees and then discusses how this technique assists in the three situations outlined above, plus a few other ones.

Related work

The present proposal is orthogonal to research into alternative layouts of data within B-tree pages, e.g., in [BU 77, DR 01, GL 01, H 81]. Similarly, it is orthogonal to the data collection being indexed or the attribute being indexed, which could be a column in a traditional relational table, a hash value, a location in multi-dimensional space mapped to a single dimension [RMF 00], or any other (deterministic) function.

Prior research and development into partitioning in parallel and distributed database systems are closely related, including [AON 96, HD 91, CAB 88]. However, none of the prior work specifically considers online index operations such as index creation, schema changes, etc., and how to exploit partitioning for those purposes. Online index construction has been considered in the past [MN 92], but not in the contexts of partitioning or of querying an index still in its construction, as proposed here. Mohan and Nareng [MN 92] also mention in a footnote that an index could be made available incrementally, but their description implies waiting until the complete sort operation starts to emit output, and they do not consider how a query processor could exploit indexes coming online incrementally, as the present paper does.

Another related research direction has considered fast insertion into novel data structures derived from B-trees, both small insertions in OLTP environments and large insertions in bulk loading, e.g., in [JDO 99, JNS 97, JOY 02, MOP 98, OCG 96]. Other research has considered fast bulk deletions, either in response to user requests [GKK 01] or as part of data migration in partitioned data stores [LKO 00]. The value of the present proposal, from a database implementer's point of view, is that no new data structures, algorithms, or quality assurance tests are required, except of course tests of truly new functionality, e.g., pausing and resuming a sort operation or querying an index still being built. Moreover, the present proposal provides improvements concurrently in three main areas – external sorting, index creation, and bulk loading – plus a few additional ones.

There is, of course, a vast amount of research on sorting. The most relevant work is on external merge sort with dynamic memory management [PCL 93, ZL 97]. These prior algorithms adjusted the merge fan-in between merge steps, which might imply a long delay; the contribution here is the ability to vary merge fan-in and memory usage dramatically and quickly at any point during a merge step without wasting or repeating any work.

Artificial leading key columns

The essence of the present proposal is to *maintain partitions within a single B-tree, by means of an artificial leading key column, and to reorganize and optimize such a B-tree online* using, effectively, the merge step well known from external merge sort. This key column probably should be an integer of 2 or 4 bytes. By default, the same single value appears in all records in a B-tree, and most of the techniques described later rely on carefully exploiting multiple alternative values, temporarily in most cases and permanently for some few techniques. If a table or view in a relational database (or any equivalent concept in another data model) has multiple indexes, each index has its own artificial leading key column. The values in these columns are not coordinated or propagated among the indexes. In other words, each artificial leading key column is internal to a single B-tree, such that each B-tree can be reorganized and optimized independently of all others. If a table or index is horizontally partitioned and represented in multiple B-trees, the artificial leading key column can be defined separately for each partition or once for all partitions – the present paper does not consider this issue further.

In fact, the leading artificial key column effectively defines partitions within a single B-tree. The proposal differs from traditional horizontal partitioning using a separate B-tree for each partition in an important way. Most advantages of the present proposal depend on partitions (or distinct values in the leading artificial key column) being created and removed quite dynamically. In a traditional implementation of partitioning, each creation or removal of a partition is a change of the table's schema and catalog entries, which requires locks on the table's schema or catalog entries and thus excludes concurrent or long-running user accesses to the table, as well as forcing recompilation of cached query and update plans. If, as proposed, partitions are created and removed as easily as inserting and

deleting rows, smooth continuous operation is relatively easy to achieve.

Adding an artificial leading key column to every B-tree raises some obvious concerns, which will now be discussed in turn – potential benefits will be discussed in subsequent sections. First, the artificial leading key column increases record lengths and therefore total disk usage as well as required disk bandwidth while reading or writing the entire B-tree. However, if prefix truncation is used [BU 77], almost all B-tree pages, both leaves and internal nodes, will store only a single copy of this key column, since its value will be constant for all records in almost all pages. Note that implementations that exploit prefix truncation do not necessarily split pages in the middle upon page overflow, instead favoring a split point near the middle that permits truncating the longest possible prefix in both pages after the split. Thus, this artificial key imposes negligible new disk space and bandwidth requirements.

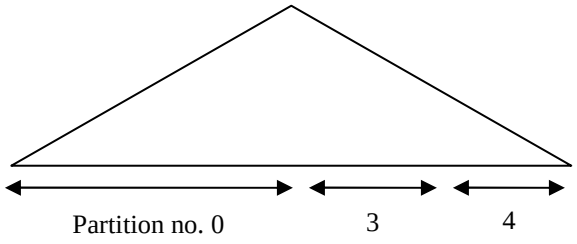

Figure 1. B-tree with partitions

Second, searches within a page are more expensive, because each comparison must compare the entire key, starting with the artificial leading key column. However, if prefix truncation is used, the key component that has been truncated because it is constant for all records in a page actually does not participate in comparisons; thus, only comparisons within pages with multiple values of the artificial key column within the page incur some cost, meaning hardly any pages and thus hardly any comparisons. Note that prefix truncation is not really required to reduce the comparison cost; "dynamic prefix truncation" requires that comparison operations indicate where in the comparison arguments the first difference was found, and permits comparisons to skip over those leading parts in which lower and upper bound of the remaining search interval coincide [L 98].

Third, searches in the B-tree are more complex and more expensive than in traditional B-tree indexes, in particular if multiple partitions exist. The situation is, of course, very similar to other B-tree indexes with low-cardinality leading columns. Each searching probe into the B-tree must first determine the lowest actual value for the artificial leading key, then search for the actual parameter of the probe, then search whether there is another value for the leading artificial key column, etc. [L 95]. The probe pattern effectively interleaves two sequences: enumerating distinct values in the leading column (as might be useful in a "select distinct …" query) and searching for index entries matching the current query.

Presume, for example, that the B-tree in Figure 1 is an index on column x, and that a user query requests items with x = 19. The first probe into the B-tree inspects the left edge of the B-tree and determines that the lowest value for the artificial leading key column is 0; the second probe finds index entries within partition 0 with x = 19. The third probe finds the first item beyond partition 0 and thus determines that the next value in the artificial leading key column is 3, etc., for a total of 7 probes including the left and right edges of the B-tree.

Fortunately, this search can be limited at both ends by the use of integrity constraints, either traditional "hard" constraints or "soft" constraints that are observed automatically by the database system and invalidated automatically when a violating record is inserted into the database [GSZ 01]. In Figure 1, if a constraint limits the partition number to 4 or less, the probe at the right edge can be omitted. If there is only one value for the artificial leading key column in the B-tree, and if integrity constraints for both ends of the B-tree exist, a probe into the proposed B-tree is as efficient as a probe into a traditional B-tree.

Fourth, B-tree indexes deliver sorted data streams as query output or as intermediate query result. In order to obtain the same sorted output stream, records from multiple partitions of the B-tree must be merged on the fly. If the number of partitions is moderate, this can be achieved very efficiently, using well known algorithms and data structures used in external merge sort.

Fifth, B-tree indexes are often used to efficiently enforce uniqueness constraints, and the proposed B-trees with the artificial leading key column substantially increase the expense of checking for a duplicate key value. This check disregards, of course, the artificial leading key column, and therefore must probe into the B-tree index for each actual value of the artificial leading key column. Again, when multiple values for this column are present in the B-tree, this concern is valid; however, in most cases and at most times, there should be only one value present and this fact should be known due to hard or soft integrity constraints.

Sixth, selectivity estimation, which is crucial for effective query optimization, could be hampered because the histogram associated with an index describes primarily or even exclusively the distribution of the leading key column, i.e., the artificial leading key column rather than the first user-chosen key column. Fortunately, most modern database systems separate the notions of histograms and indexes. While it used to make sense to link the two because both needed full data scans and sorting for efficient construction, modern database systems build histograms from sampled data and refresh them much more often than they rebuild B-tree indexes. Typically, a sufficient sample

easily fits into main memory and thus can be sorted efficiently. Due to this efficiency, most database systems and installations support statistics for columns that are not indexed at all or are not leading columns in indexes, which is precisely the type of statistics needed here.

Finally, a few more observations that likely are obvious and thus are mentioned only briefly. The proposed use of B-trees is entirely orthogonal to the data collection to be indexed. The proposed technique applies to relational databases as well as data models and other storage techniques that support associative search, it applies to both primary (clustered) and secondary (non-clustered) indexes, and 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 well as to indexes on traditional tables.

To summarize, adding an artificial column to each B-tree index raises several obvious possible concerns, but all of them can be mitigated to a negligible level. Having considered these concerns, let us now discuss the benefits.

Sorting

Virtually all database systems use external merge sort for large inputs, with a variety of algorithms used for internal sorting and run generation. One important design issue is how to store intermediate runs on disk such that can be read efficiently in sort order. Many database servers use roughly ten times more disk drives than CPUs; in some case, however, the number of disk arms is effectively unknown to the database management system since an entire disk farm or network attached storage is shared by many users and even multiple servers, including multiple database servers. In order to keep all disk arms usefully busy and in order to hide all I/O latencies, asynchronous I/O is needed while writing initial runs and while reading and writing runs during merge steps. Asynchronous writing is relatively easy since it is always clear which pages should be written and since the CPU process does not need to wait for completion of the I/O. Asynchronous reading in merge steps requires more attention for two reasons. First, if a required page is not yet in memory, the sorting program must wait, thus relinquishing not only the CPU but also the CPU cache. Second, the very nature of merging implies that many inputs are read, and it is necessary to determine which of the inputs must be read next, commonly known as forecasting [K 73].

Note that double buffering [S 89a] for all input runs does not truly solve the problem. On one hand, it reduces the merge fan-in to half, whereas good forecasting reduces the fan-in only by a relatively small fixed number. Useful values are the number of disk drives if known or simply ten, based on the rule of thumb that there are roughly ten times more disks than CPUs in a balanced server. On the other hand, when merging runs of very different sizes, substantially more read operations will pertain to the large input runs – a typical situation occurs when merging some initial runs (which are about the size of memory) and some intermediate merge results (which are larger than the initial runs by a factor equal to the merge fan-in, e.g., 100). Moreover, if the key distribution in the input is skewed, i.e., if there is any form of correlation between input order and output order, even input runs of similar sizes might require different amounts of read-ahead at different times during a merge step.

In both cases, deep forecasting is required, i.e., forecasting that reaches beyond one asynchronous read operation and beyond finding the lowest one among the highest keys on each page currently consumed by the merge logic [K 73]. Other researchers have considered technique for planning the "page consumption sequence" ahead of a merge step [ZL 98] or as the merge progresses [S 94]. In both efforts, a separate data structure was designed to retain the highest keys in each data page. In commercial reality, however, every new data structure requires new development and, maybe more importantly and more expensively, testing, which is why neither of these designs has been transferred into real products.

Retaining all runs in a single B-tree, using the run number as the artificial leading key column, addresses several issues without introducing the need for a new data structure. Most immediately, the parent level about the B-tree's leaves is a natural storage container for precisely the keys needed for accurate deep forecasting. In fact, it is possible to forecast arbitrarily deeply, and to do so dynamically while merging progresses, i.e., adapt the forecasting depth to the current I/O delay as well as add or drop runs from the forecasting logic. Moreover, a scan over the leaves' immediate parent nodes is already implemented in some database systems because it is also required for multi-page read-ahead in an ordered key range retrieval, e.g., a large "between" predicate.

The space and I/O overhead for using a B-tree for runs is negligible: internal B-tree nodes of 8 KB have a fan-out of at least 100, meaning that about 99% of all pages in the B-tree are leaves. A B-tree fan-out of 100 is very conservative if prefix and suffix truncation are used and if the space utilization is 100%, which is possible because the B-tree is loaded sequentially by the merge step. Thus, a B-tree fan-out of 400 seems realistic in many cases, meaning about 0.25% of all pages are not leaves. Actually, since the leaves' immediate parents are equivalent to any other data structure that captures the consumption sequence of pages in merge input runs, only 1% of 1% of all pages (or 0.25% of 0.25%) in the B-tree is overhead due to using a B-tree to store all runs.

Another benefit of using a B-tree to store all runs is that parallel threads can be added or removed from the sort effort at any time. A new thread can be put to good use

simply by choosing and assigning a set of runs to merge and a key range within those runs. Even in an external sort with a single merge step, the final merge can be parallel. Inversely, a thread can stop its work at any time – the remaining B-tree is still a valid and consistent collection of runs. No work already performed is wasted and no work is performed twice. The operation to delete an entire key range within a merge input run is precisely the same one that deletes an entire run, and is already implemented in B-tree implementations used in data warehousing, where entire date ranges are regularly added and removed. Similarly, memory can be added and removed from a sort operation at any time, without loss in I/O efficiency, i.e., without the need to shrink the units of data transfer. The merge process can add or drop runs at any time. In the extreme case, a merge process can drop all its runs, meaning that the entire sort operation is paused. With appropriate transaction support, sort operations can be resumed even after server restart. Note that it is quite straightforward to drop runs from the current merge step; adding a run requires finding in an existing run precisely the right key that matches the current merge progress. This search is obvious and easy with runs in a B-tree, due to B-trees' inherent support for "between" predicates, whereas it requires expensive searching in traditional "flat" run files.

The resulting runs with partial key ranges enable optimizations traditionally conceived for partially pre-sorted inputs [H 77]. Two runs with disjoint key ranges can be thought of as a single run, and can together, one after another, serve as a single input in a future merge step, a technique called "virtual concatenation" here. In addition to the traditional use of this technique, a B-tree even permits to rearrange key ranges within runs. Instead of merging or concatenating entire runs, fractions of runs defined by key ranges could be merged or concatenated. When reaching a given pre-planned key, one or multiple merge inputs are removed from the merge logic and other runs added. For example, consider an external merge sort with memory for a fan-in of 10, and 18 runs remaining to be merged with 1,000 records each. The keys are strings starting with a character in the range 'a' to 'z'. Presume both these keys occur in all runs, so traditional virtual concatenation does not apply. However, presume that in 9 of these 18 runs, the key 'm' appears in the 100th record; while in the other runs, it appears in the 900th record. The final merge step in all merge strategies will process all 18,000 records, with no savings possible. The required intermediate merge step in the standard merge strategy first chooses the smallest 9 runs (or 9 random runs, since they all contain 1,000 records), and merge those at a cost of 9,000 records read, merged, and written. The total merge effort is 9,000 + 18,000 = 27,000 records. The alternative strategy proposed here merges key ranges. In the first merge step, 9 times 100 records with keys 'a' to 'm' are merged followed by 9 times 100 records with keys 'm' to 'z'; all 1,800 records into a single output run. The final merge step merges these 1,800 records with 9 times 900 records with keys 'a' to 'm' followed by another 9 times 900 records with keys 'm' to 'z'. Thus, the total merge effort is 19,800 records – a savings of about 25% in this (artificial) example. Starting a merge at such a "given" key within a run on disk is very inefficient with traditional runs, but is no problem if runs are stored in a B-tree.

Given all these adaptive mechanisms[1], one important design issue is management of information about runs, how to determine efficiently and at any time which runs currently exist, their sizes and their key ranges. In an index with only a few partitions, it is possible to enumerate the partitions at the expense of one root-to-leaf probe per partition, possibly saving the first and the last probe through constraints on the artificial leading key column. In a large external merge sort, one can determine the set of runs in the same way. Even some of the interesting properties of runs, e.g., run sizes, can be estimated quite accurately because all leaves and all interior nodes of the B-tree are filled 100%. It is probably more efficient, however, to employ a separate table with information about runs. Depending on the detail captured, e.g., information about entire runs only or information about key distributions within runs for virtual concatenation of key ranges, this table might need to be stored on disk. One design allocates a small amount of memory to run management, e.g., two pages, and overflows all further run descriptors to disk. Merge planning is based on those two pages, and only when the number of runs has shrunk such that their descriptors fit on one page, another page of run descriptors is loaded. In an alternative design also using a small amount of memory, intermediate merge steps are forced when the number of runs exceeds a given threshold. Note that for such forced intermediate merge steps, merge planning should attempt to merge runs with the most similar sizes rather than the smallest runs, which is the usual optimization heuristic for merge planning.

To summarize this section on sorting, capturing runs in an external merge sort opens new opportunities, principally in two directions. First, it enables more efficient sorting due to accurate deep forecasting and to virtual concatenation of key ranges. Second, it enables mechanisms that enable large sort operations to adapt to the current system load quickly and over a wide range of resource levels. In other words, it enables mechanisms required in self-tuning database management systems.

[1] For memory adjustment during run generation, Zhang and Larson proposed a method that is both adaptive and cache-efficient [ZL 97]: Sort each incoming data page into a mini-run, and merge mini-runs (and remove records from memory) as required to free space for incoming data pages or competing memory users. Techniques from [LG 98] can be adapted to manage space for individual records, including variable-length records.

Index operations

Database system use sorting for many purposes, not the least among them is efficient construction of B-tree indexes. All the sorting techniques discussed above apply to index creation operations, including pause and resume without loosing or wasting work, e.g., after a load spike or server shutdown. In addition, online index creation can exploit B-tree indexes with an artificial leading key column in an interesting way, as follows. At the end of the run generation phase, a single B-tree contains all future index records, albeit not yet in the final order. Nonetheless, the records are already sufficiently organized to permit reasonably efficient searches. Thus, concurrent queries may start exploiting the new index after only a single pass over the data, even before the start of the merge phase. If the index creation requires only a single merge step (the usual case nowadays), this means that the index is available for querying in half the time of traditional index creation. For a very large index, the reduction in latency might even be a factor 3 (for a two-level merge sort).

When searching indexes that are not fully merged and optimized yet, there is a compromise in search efficiency but, unless the initial runs are very small, it is faster to probe into each run using a root-to-leaf B-tree traversal than to scan the new B-tree in its entirety. For example, presume that all nodes above the leaves or at least above the leaves' immediate parent nodes will fit in the buffer and therefore will not incur I/O during a probe. Thus, each root-to-leaf traversal will incur at most two random I/Os, which takes about 12 ms using contemporary fast disks. Recall that two root-to-leaf passes may be required for each run or partition within the B-tree, or about 24 ms per distinct value in the artificial leading key column. During that time, today's fast disk drives can deliver about 8 MB at their nominal (ideal) speed of 320 MB/s. Thus, if the average initial run is longer than 8 MB, queries will perform better by probing the new index than by scanning the old storage structures. Note that a file scan at full speed puts much more load on the system's CPUs, memory, and bus than repeated index probes; thus, index probes are even more preferable in a multi-user environment.

For correct transactional behavior, the transaction creating the index should commit after the initial runs are complete. Concurrent transactions should not query the new index when its creation might still roll back; in fact, the query optimizer should not create execution plans that search an index whose existence is not yet committed. After the initial index is committed, subsequent merge steps may be part of the original statement execution but should not be part of the original transaction. Instead, since they only modify the internal index structure but not database or index contents, they can be system transactions that commit independently, rather similar to system transactions used routinely today, for example during a node split in a B-tree index. While concurrent queries search and update the index, concurrent merge steps should have excellent online behavior. Specifically, when conflicting with a lock held or requested by a concurrent user transaction, the merge step should let the concurrent transaction proceed. Fortunately, as discussed in the section above on sorting, small ranges can be merged individually, even concurrently by multiple independent threads, and a merge step can commit and terminate at any time, and resume later without any work being wasted.

For correct durability after a new index has been committed in its initial format, all further modifications of the index must be fully logged, including the merge actions. Changes may be logged per page in order to avoid per-record overheads, and it may be possible to combine log records for page deletion (in the merge input) and page creation (in the merge output). The initial data transfer (prior to committing the initial index) may omit data logging, similar to today's techniques that require flushing the new index to disk and capturing the index contents when backing up the log, as optional in [MS 98]. Further reductions in logging volume may be possible but require further research, and they may introduce new tradeoffs and compromises, e.g., retaining rather than reclaiming data pages of merge input runs.

If concurrent transactions update the indexed table (view, etc.) while the initial runs are created, these updates must be applied to the future index before it may be queried or updated. There are two well-known methods to do so [MN 92]: either a log-driven "catch up" phase applies these updates to the index after the index builder completes, or the concurrent transactions apply their updates immediately to the index, which in the present proposal consists of initial runs. Given that the recovery log typically cannot be searched by key value, the latter technique is more interesting here. Thus, a new index must be tagged "in construction" such that updates but not queries consider the index. Deletions of records not yet in the index insert some special markers or "anti-matter" that will be applied and cleared out later by the index builder. Transactions searching the index before the merge phase completes must search not only for valid records but also for anti-matter, very similar to searching in a differential file [SL 76]. Fortunately, all insertions, deletions, and anti-matter insertions by concurrent transactions can be collected in a single partition, i.e., a single, constant, well-known value for the artificial leading key column. Assuming record-level or key value locking, the level of lock contention among concurrent transactions should not be greater than lock contention will be in the final index, and thus may be presumed to be acceptable. In order to reduce lock and latch contention between concurrent transactions and run generation within the index builder, it is advantageous to separate this B-tree partition from the runs, e.g., use value 0 in the artificial leading key column for inser-

tions and deletions by concurrent transactions and start run generation with run number 1.

The possibility of creating a new index in a single pass over the data, to the point of making the index usable to retrieval queries even if it is not immediately optimal, can be extended even further. If the data source during the index creation is ordered, e.g., if it is a primary index, key ranges in the source will approximately correspond to initial runs in the index being built. Specifically, if run generation repeats read-sort-write cycles (as many sort implementations based on quicksort do), initial runs in the new index will precisely correspond to key ranges in the old index. If run generation streams data from the input to the initial runs (as many sort implementation based on replacement selection do), the steady pipeline can be interrupted and flushed every now and then, or at least the assignment of run numbers to records entering the priority heap in replacement selection can be modified to flush input records into the new index. After all records with key values in the old index within a certain range have been flushed into the new index (albeit in multiple runs), the set of records already captured in the new index can be described with simple range predicates in both the data source and the new index being built. In the scanned data source, the predicate uses the search key of that index, and in the new index, the predicate uses the run number, i.e., the artificial leading key column. Such simple range predicates are, of course, fully supported in all implementations of indexed (materialized) views; thus, even such a partial index [S 89b] can be made available to the optimizer, very similar to an index on a (materialized) selection view.

For views of this type, optimizers can construct dynamic execution plans with two branches for each table access, one branch exploiting the new index for a query predicate subsumed by the predicate describing the range of rows already indexed, and one branch to process the query without the new index. Note that some query executions may employ both the old and the new indexes, relying on two mutually exclusive range predicates for the two indexes to find each qualifying row exactly once. Thus, a new B-tree index can be considered by the query optimizer immediately after index creation begins, and it becomes more and more useful for query processing as index constructions proceeds, both during initial run generation (range by range) and during the subsequent merge phase (fewer and fewer partitions or runs within the new B-tree). In the extreme case, a new index can be committed instantly without moving any data at all and independently of the size of the data source and of the new index. In other words, the initial user transaction verifies schema and permissions, reserves sufficient disk space for the entire future index, creates initial catalog entries including a boundary predicate and a boundary not satisfied by any data currently in the database, and then commits and reports success to the user. Thus, it leaves all data movement and sort work to asynchronous system transactions that will run later. The price of this flexibility is that even the initial data insertions must be fully logged, like all data movement after the existence of the index has been committed, although further research may be able to reduce the logging volume.

An index can be populated not only for one continuous range, as proposed above, but for multiple ranges that may or may not be contiguous, called “inclusion ranges” elsewhere [SS 95]. These ranges are maintained in a control table very similar to control tables (also called “tables of contents”) commonly used today for selective replication or caching. Despite describing the contents of an index, these control tables are data, not meta data. Therefore, a change in a control table does not trigger plan invalidation or query recompilation. Note that a single control table may suffice for an entire database with all its tables and indexes if normalized keys are used, i.e., in some sense all indexes only have one search column, which is a binary string. While an additional range is being populated, concurrent updates (by concurrent transactions) must be applied to the new index, including anti-matter, as discussed above for online index creation without ranges. Thus, individual ranges must be tagged as “fully operational” or “in construction,” and branch selection in dynamic execution plans for updating a table are controlled slightly differently than in dynamic plans for selecting from a table.

Partial indexes also open a door for another promising technique. If the query optimizer cannot find a suitable index for a query and must thus plan a table scan, it can determine the most useful possible index and then prepare a dynamic plan with two alternative branches. One branch exploits this index if it already exists; the other branch performs the table scan but leaves behind a B-tree that contains the initial runs for this index, leaving it to an asynchronous utility operation to optimize this index by merging those runs into a single traditional B-tree. Actually, there must be three branches in the query plan, the third one simply performing the table scan without leaving anything behind, to be used if the empty initial index cannot be created when the query plan needs to start, e.g., due to space constraints or due to locks held by concurrent transactions. Note that run generation using a replacement selection does not substantially alter the flow behavior of the file scan. One of the issues that need to be resolved is how the file scan produces records for two different transaction contexts, the user query and the index builder. Fortunately, using “insert” and “delete” tags familiar from maintenance plans for indexed (materialized) views can readily be adapted for this purpose.

Unique indexes, or indexes built for efficient maintenance of uniqueness constraints, pose an additional challenge for online index creation. The traditional approach has been to fail concurrent transactions or the index

builder when a uniqueness violation is detected [MN 92]. Thus, index creation may be aborted hours or even days after it starts and minutes before it completes due to a single committed insertion by a concurrent transaction, even if another concurrent transaction is about to delete one of the duplicate keys and commit it before the index builder will complete. Fortunately, a useful technique exists that also extends "soft constraints" [GSZ 01] from single-row "check" constraints to uniqueness and key constraints. Its essence is to maintain a counter of uniqueness violations for each possibly desirable uniqueness constraint, in the minimal case only one but in the maximal case for all prefixes of the B-tree's search key. Thus, whenever a B-tree entry is inserted or deleted, it must be compared with one or both of its neighbors, and counters must be incremented or decremented appropriately. It is important to maintain these counters accurately and with correct transaction semantics. While escrow locks [O 86] might prove helpful for such counters, some systems maintain such counters apparently without them [CAB 93], possibly by using transaction-internal counters made globally visible only during commit processing. When a counter for a specific prefix is zero, this set of leading columns is unique and a uniqueness constraint can be enabled instantly without further validation. If such counters are maintained during index construction, neither concurrent transactions nor the index builder must be aborted due to uniqueness violations, and the index structure may be retained even if the uniqueness constraint is not satisfied at the time the index builder completes. Note that this separation of indexes and uniqueness is entirely consistent with the physical nature of indexes (they are an issue of database representation, not of database contents) and the logical nature of constraints (which limit and describe the database contents, not the data representation), and is also reflected appropriately in the SQL standards, which include syntax and semantics for constraints but not for indexes.

It might seem that an artificial leading key column inhibits using this technique. Indeed, while there are multiple distinct values for this column, it is impossible to activate a uniqueness constraint instantly. Instead, the existing partitions of the index B-tree must be merged to verify uniqueness. The result of this merge step can be materialized, in which case this merge step is precisely the final step of sorting or of index creation. Alternatively, the merge result can be ignored, leaving the final optimization of the index to a subsequent online reorganization. Notice, however, as mentioned earlier, that enforcing a uniqueness constraint using an index with multiple partitions is more expensive than using one with only a single partition. Thus, finalizing the B-tree index before activating a uniqueness constraint based on the new index seems like the right approach in general. Nonetheless, there are also situations in which multiple partitions in persistent indexes are extremely useful, e.g., when importing large amounts of data into a pre-existing large and fully indexed database.

To summarize this section, a new index can be available for queries substantially earlier than in traditional methods if initial sort runs are collected in a single B-tree, and online index construction can even be incremental. Moreover, self-tuning query plans as briefly outlined here would represent a great step forward compared to the tuning capabilities found in most database systems today; maybe B-tree indexes with an artificial leading key column will turn out an important step in this direction due to the single-pass construction of the initial index. Rather than relying on "wizards" or "advisors" that run outside the query processor [ACN 00, VZZ 00], creating and populating useful indexes can become a native and integral part of query optimization and query execution. Incidentally, query optimizers routinely decide on index creation and have the query execution plan populate such indexes; the main difference is that those temporary indexes today are created, populated, used, and destroyed within a single query plan and transaction context rather than maintained during database updates and then amortized over multiple invocations of the same (possibly parameterized) query.

B-tree loading

While we may hope that indexed (materialized) views substantially alleviate the response time in relational OLAP (online analytical processing), one issue that will remain is importing new data into existing large, populated, and indexed data warehouses. In a very common scenario, at the end of every month, another month's worth of data is added to the detail data. The key difficulty is that the largest table typically has multiple indexes, all on different columns, and only one index on time. Traditional solutions have been to keep separate tables and their indexes for each time period, equivalent solutions using partitioning and "local" indexes (i.e., secondary indexes are partitioned precisely like the primary index or the base table), dropping all indexes during data import and rebuilding them afterwards, or special "update execution plans" that merge an ordered scan of the old index with a sorted set of index insertions into an entirely new index. It appears that partitioning with local indexes has shown the most desirable properties, namely fast data import and short delay until queries can exploit new indexes on the new data. The unfortunate aspect of partitioning is that many individual partitions must be managed for each index, with additional catalog tables, catalog entries, catalog look-ups, etc.; the other very unfortunate aspect is, of course, that each partition must be searched when the query predicate does not restrict the query to a single time period.

B-tree indexes with an artificial leading key column offer an attractive combination of features in these situa-

tions. In effect, the artificial leading key column defines partitions; however, it does so within the structure of a single index, single B-tree, and single partition as far as the catalogs and the query optimizer are concerned. Moreover, the partitions within the B-tree are temporary, to be removed by online reorganization at the earliest convenient time. This perspective, using the artificial leading key column as a form of partitioning, immediately leads to a very efficient bulk insert strategy: let each large insert define a new partition, and then let incremental index reorganization re-optimize the B-tree structure at a convenient time. In the best case, this reorganization is incremental, online, and responsive to the current system load. Note that multiple batches of bulk insertions can be inserted into a B-tree before reorganization takes place, that reorganization may proceed incrementally range by range, that a reorganization step does not necessarily affect all existing partitions, and that reorganization can proceed even while another batch is being imported. Note also that multiple batches can be inserted concurrently, even by multiple users; contention for locks and latches should be negligible if page splits optimize the key distribution for maximal prefix truncation [BU 77] rather than assign precisely half the data to each resulting page.

Letting each large insert operation create a single new partition implies that the insert operation pre-sorts the new data appropriately for each index. Following the discussion earlier in the paper, this sort operation might employ a B-tree to hold intermediate runs. Rather than using a separate, temporary B-tree, however, the bulk import operation can immediately use the import target. Thus, when importing into a B-tree index, the incoming data is processed into runs, and runs are added immediately as individual partitions to the permanent B-tree. When importing data into a table with multiple B-tree indexes, each of those forms its own new partitions, which are independent of the new partitions in the other B-trees. Thus, runs for all B-trees can be formed concurrently, such that incoming data is never written to any temporary structures, meaning that it is processed in memory only once before it can become available for retrieval queries. Both load-sort-store and replacement selection can be used for run generation. The expected size of the runs relative to the memory size depends on the choice of algorithms, but runs should be at least as large as the allocated memory in all cases. As discussed for index creation, parts of the data can be committed and made available for queries at any time; of course, in this case, all participating indexes must be flushed in order to ensure transactional consistency among them.

It might be interesting to compare the effort in a traditional bulk insert with the effort in the scheme proposed here, i.e., the combined effort of appending partitions and merging them into the main partition. The traditional bulk insert strategy for a single large batch sorts its entire insert set and then merges the sorted set into the B-tree [MS 98, GKK 01]. Sorting the insert batch typically relies on an external merge sort. If there are multiple batches, each of them is sorted and each of them requires updating many or even all leaf pages in the index. The proposed method, on the other hand, works efficiently independent of the number and size of batches, unless batches are much smaller than the memory that can be dedicated to run generation. Runs are formed and merged using the same amount of memory and effort in the traditional and in the proposed strategies; the main difference is that the merge strategy can be optimized ignoring which batch generated which runs.

The proposed bulk insert strategy offers further benefits. Maybe most importantly, newly imported data are available for queries much faster than in traditional strategies, even if multiple indexes need to be maintained. Thus, the proposed strategy is suitable for indexing and querying continuous data streams. Moreover, the proposed algorithm reduces the number of index pages that are modified and thus will participate in a transaction rollback (should that be necessary) or in an incremental backup immediately following the data import. In addition, this strategy effectively eliminates lock conflicts between the import transaction and any concurrent transactions. Finally, many fewer log records are required during bulk insertion because each actual log record can describe an entire new page, rather than an individual record insertion. Note that index entries in secondary indexes are often much smaller than the fixed space overhead for log records, including previous LSN, next undo LSN, etc. [MHL 92]. An index entry of 20 bytes might result in a log record of 80 bytes – thus, paying the overhead of a log record once per page rather than per index entry substantially reduces the log volume during the insert operation. If, on the other hand, bulk insertions are not logged but only flushed to disk at the end of the insert operation, to be made truly durable only by a backup, partitioning within a B-tree substantially reduces both the flush effort and the backup volume.

Unfortunately, there are also some concerns during data loading. If enforcement of a uniqueness constraint relies on a B-tree index, duplicate search keys may be located not only in immediately neighboring B-tree entries but also in other partitions. In other words, if it is truly imperative that the B-tree at no time and under no circumstances contain duplicate entries, bulk loading has to search in all partitions for all search keys. Note that each such search can be leveraged for multiple new records; the resulting algorithm resembles a merge join with each prior partition (actually a merge-based anti-semi-join). If, on the other hand, it is sufficient that the uniqueness constraint holds only for the B-tree entries in the default partition (say value 0 for the artificial leading key column), bulk insert into other partitions can proceed at full speed, and verification of the uniqueness constraint can be left to the B-tree reorganization performed later. For example, the

reorganization might simply skip over duplicate keys; when reorganization is complete, only duplicate keys are left in those partitions. In fact, most implementers and administrators of data warehouses prefer a tolerant data loading process, because typically only a small minority of records violates any constraints and it is not worthwhile to disrupt a high-speed loading process for a few violations that can be identified and resolved later.

A related issue is the generation of "uniquifiers" in primary indexes. One design lets entries in secondary indexes "point" to entries in the primary index by means of search keys – the advantage of this design is that page splits in the primary index do not affect the secondary indexes [O 93]. If the search keys in the primary index are not unique, an artificial uniquifier column is added as a trailing column to each clustering key. (In efficient implementations, one instance per unique search key may have a *NULL* uniquifier value, which like other *NULL* values is stored very compressed in the primary index and in any secondary index.) If multiple partitions may hide actual duplicate search keys in the primary index, either the assignment of uniquifiers must search all partitions or the "pointer" from a secondary index into the primary index must include the value of the artificial leading key column in the primary index. Moreover, any reorganization of the primary index may need to assign new uniquifier values and thus require expensive updates in all secondary indexes.

Perhaps a better design that requires less reorganization of secondary indexes adopts an additional artificial trailing key column in the primary index, and a slowly increasing boundary value indicating which values for the artificial leading key column have already been used and reorganized into the main part of the primary index. If a value for the artificial leading column is found in an entry in a secondary index that is higher than this boundary value, it is interpreted as the artificial leading key column in the primary index as described above. If, however, a value is found that is lower than the boundary value, the pointer into the secondary index is dereferenced using the default value for the artificial leading key column in the primary index and the old low partition number is interpreted as the value for the new trailing key column. Thus, a table's row and all its representations in the primary and all secondary indexes will retain the initial partition number forever, but the interpretation of that number changes over time when the primary index is reorganized.

To summarize this section, B-tree indexes with partitions defined by an artificial leading key column transfer the advantages of partitioning without some of the disadvantages. In particular, a large data insert operation or bulk insert can append runs or partitions to all B-tree indexes in a table, whether or not the load file and the indexes are sorted on the same columns, and it does so without lock conflicts and with minimal log volume. It is even possible to append runs to multiple indexes, which is particularly attractive for capturing and indexing non-repeatable data streams.

Other applications

B-tree indexes with artificial leading key columns can improve not only large inserts but also small ones. Other researchers have proposed constructing multiple coordinated structures, e.g., the log-structured adaptations of B-trees [MOP 98, OCG 96], or employing new structures with new algorithms, tuning parameters, etc., e.g., buffer trees [A 95, V 01] or the Y-tree [JDO 99]. Instead, a single traditional B-tree can be used, with multiple partitions based on an artificial leading key column, with one partition small enough to fit in memory. Inserts are directed to this partition, and online reorganization in the background merges its records into the main partition of the B-tree. Note that this idea works for both updates and retrievals. If certain values are searched more often than others, those can be gathered into one small partition, such that those searches can be performed always entirely within the buffer pool. In a way, this design creates a self-organizing search structure.

Large deletions, on the other hand, can greatly benefit from a preparatory online reorganization. First, all index entries to be deleted are assigned to a single B-tree partition, i.e., are assigned a new value of the artificial leading key column. When this is complete, a large and efficient range deletion can remove all those entries from the B-trees very efficiently. The reorganization is about as fast as the bulk deletion techniques described in [GKK 01], whereas the actual deletion should be an order of magnitude faster. A special application of this technique is data migration in a parallel database or any other partitioned data store, e.g., when adjusting the boundary between two partitions: first prepare all source indexes for a large deletion using small online steps, then move data using the bulk delete and bulk insertion strategies proposed in this paper, and finally optimize the destination indexes in small online steps. Note that the transactions performing the initial and final reorganizations are local transactions; therefore, multi-node commit processing is needed only for the actual data movement.

Another promising application combining insertions and deletions is capturing and holding a window within a continuous stream of incoming data, e.g., sensor data. The insertions may be grouped, sorted, and inserted as a batch similar to traditional bulk data import; or random insertions can always focus on the latest, smallest, most active, and therefore memory-resident partition. If only a window of recent data is to be retained and items older than a preset threshold are to be deleted, e.g., in order to analyze auto-correlations or periodic phenomena within a stream, deletions can similarly either be batched or focused within a single, small and thus memory-resident partition.

Incremental index maintenance over continuous input streams also enables a symmetric dataflow join that mirrors the benefits of earlier proposals [DST 02, WA 91] – two indexes are built on the two join inputs, input from either join input is accepted at any time and matched against the currently existing index on the other input. This strategy closely mirrors earlier prototypes of symmetric hash join; the difference is that the in-memory hash table and hash table overflow are replaced by a B-tree and the standard buffer manager support for B-tree indexes.

Presuming incremental online index reorganization is available, the techniques discussed above for creating B-tree indexes can be extended to other index operations, e.g., changing the schema of the records stored and indexed. A typical example is changing an existing column's type, length, or precision, e.g., from an integer or a decimal numeric to a floating point type. If all records in the index are modified immediately as part of the change statement, such a simple statement may run a long time, typically with an exclusive lock on the entire index or even the entire table. Incremental online reorganization is an attractive alternative, although it implies that the index contains both old and new records for a while, and that the index must support both queries and updates for this mixture of record formats, which introduces new complexity. If records and their keys are not normalized, records of old and new formats can be compared correctly, albeit quite expensively – every single comparison must consider the formats of the two records currently being compared. If records and keys are normalized, and in particular if the normalized form is compressed, normalization of the old and new record formats might not permit correct comparisons. In that case, defining two partitions within a B-tree index is a simple and effective solution, with different normalizations in the separate partitions. It permits instant completion of the user's request as well as efficient normalization and incremental online reorganization.

Not only different schemas and their normalization but also different validity status can be assigned to different partitions. For example, online index creation "without side file" [MN 92] requires that concurrent transactions insert a deletion marker ("anti matter") into an index if they cannot delete an entry because the index builder has not inserted it yet. If, for some reason, it is desirable to keep a large part of an index stable, e.g., in order to limit the size of incremental backups, insertions and deletions may all be inserted into a separate partition, using anti matter to represent deletions. Of course, this is very similar to differential files [SL 76], but applied specifically within B-tree indexes in database systems. In a variation of this technique, if insertions and deletions are marked with time stamps, multiple partitions can serve as main B-tree and its associated version store in multi-version concurrency control and snapshot isolation. The required query execution techniques are very similar to those required in some execution plans for large updates, namely those that merge a pre-existing index with the delta stream into an entirely new index. The only difference is that the merge result is not stored in a new index structure but pipelined to the next operation in the query evaluation plan.

To summarize, there seems to be a large variety of situations in which partitioned B-tree indexes using an artificial leading key column can enable or at least simplify implementation of online changes of schema and data in large databases.

Implementation notes

B-tree indexes with artificial leading key columns can be implemented at two levels. If B-tree indexes with artificial leading key columns are not a native feature in a database management system, a database administrator can create those columns, one per index, and adjust various commands to take advantage of the resulting flexibility. For example, bulk insert commands must assign suitable values to these columns, constraints must restrict these columns to a single constant value at most times, and histograms must exist for the columns beyond the artificial leading key column. Online index reorganization can be achieved using scripts that search for indexes with multiple values in the artificial leading key column and, when found, assign new values to some rows selected by ranges of the trailing index columns. While this method is more cumbersome and less efficient than a native implementation, it achieves many of the desired benefits.

If, on the other hand, the vendor deems the advantages discussed so far important enough, the artificial leading key column can be a hidden implementation feature of B-tree indexes. Whether a customer wants them or not, they are always there. Prefix truncation ensures that their space usage and their overhead in individual comparisons are negligible, soft constraints ensure that they do not introduce additional root-to-leaf B-tree searches (i.e., in most cases the optimizer can exploit that there is only a single value), and a suitable histogram implementation ensures that the artificial leading key column does not confuse selectivity estimation in the query optimizer. Online index reorganization is a great advantage in this implementation strategy.

In order to ensure correct transaction semantics, traditional locking mechanisms suffice. For example, index reorganization today employs many small system transactions – these transactions do not change database contents, only data representation, and therefore can commit even if the invoking transaction might roll back, and they can commit without forcing the commit record to stable storage. If system transactions are small, e.g., B-tree page splits or small reorganizations, a "no steal" buffer policy permits omitting "undo" log records [HR 83]. Commercial database management systems already employ various

techniques to avoid “redo” log records for index creation. During run generation in online index creation, each transaction scans a range of input data, produces as many runs as necessary, inserts run descriptors into the auxiliary table, updates the boundary value in the predicate describing the coverage of the new index, and then commits. Concurrent user transactions retain ordinary locks in the auxiliary table and thus prevent runs from vanishing during a user transaction; this is the reason why concurrent user transactions should be small and short, and one of the ways in which large concurrent transactions reduce the efficiency of online index creation.

Summary and conclusions

The purpose of this paper has been to re-think techniques that might have seemed completely understood. It turns out that a fairly simple and maybe surprising technique can substantially increase the performance and capabilities of sorting and indexing, in particular in the increasingly important aspects of self-tuning and self-management as well as online operations for continuous availability. Levels of resource allocation in a multi-user server must adapt fast to be truly useful, and online index operations are an important feature for both low-end and high-end database installations: At the high end, online operations improve service availability, and at the low end, they are a crucial enabler of automatic index tuning, because automatically dropping and creating an index is only acceptable and is only invisible if all applications and all data remain available at all times without “random” interruptions of service due to automatically initiated tuning or maintenance.

The central idea of this paper is to employ ordinary B-trees in a slightly unusual way, namely by introducing an artificial leading key column and thus, effectively, partitioning within a single B-tree. Earlier sections reviewed possible concerns and overheads, most of which can be overcome or reduced to a truly negligible level, as well as the many benefits of the proposed change. Supporting multiple partitions in a single B-tree index is an extraordinarily powerful concept, in particular if combined with incremental online reorganization using the merge logic well known from external sorting. It permits using a single B-tree to store all runs in an external merge sort, which in turn enables relatively straightforward implementations of important dynamic sorting techniques, including deep forecasting when merging runs from many disk drives as well as dynamic memory management even to the extremes of running multiple merge steps for separate ranges concurrently and of pausing a merge step at any point to resume it later without wasting any work. Partitioning within a single B-tree also enables practically useful advances in online index operations, e.g., index creation or schema modification. The most exciting among those is that a new index can be made available to queries in half the time of the the time of the traditional method, and even less if partial indexes are exploited by the query optimizer. Finally, partitioning within a single B-tree can be exploited for speedier updates and retrievals, most importantly bulk insertion, which can proceed with the speed of index creation even when adding new records within the preexisting key ranges of multiple populated B-tree indexes.

Acknowledgements

Several friends and colleagues have made a number of interesting and helpful suggestions on earlier drafts of this paper or its contents, including David Campbell, John Carlis, César Galindo Legaria, Jim Gray, James Hamilton, Joe Hellerstein, Tengiz Kharatishvili, Per-Åke Larson, Steve Lindell, and Florian Waas.

References

[A 95] Lars Arge: The Buffer Tree: A New Technique for Optimal I/O-Algorithms. Workshop on Algorithms and Data Structures, Lecture Notes in Computer Science 955, Springer Verlag, 1995: 334-345.

[ACN 00] Sanjay Agrawal, Surajit Chaudhuri, Vivek R. Narasayya: Automated Selection of Materialized Views and Indexes in SQL Databases. VLDB Conf. 2000: 496-505.

[AON 96] Kiran J. Achyutuni, Edward Omiecinski, Shamkant B. Navathe: Two Techniques for On-line Index Modification in Shared Nothing Parallel Databases. SIGMOD Conf. 1996: 125-136.

[CAB 88] George P. Copeland, William Alexander, Ellen E. Boughter, Tom W. Keller: Data Placement In Bubba. SIGMOD Conf. 1988: 99-108.

[CAB 93] Richard L. Cole, Mark J. Anderson, Robert J. Bestgen: Query Processing in the IBM Application System/400. IEEE Data Eng. Bull. 16(4): 19-28 (1993).

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

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

[DR 01] Kurt W. Deschler, Elke A. Rundersteiner: B+ Retake: Sustaining High Volume Inserts into Large Data Pages, ACM Fourth Int’l Workshop on Data Warehousing and OLAP, Atlanta, GA, 2001.

[DST 02] Jens-Peter Dittrich, Bernhard Seeger, David Scot Taylor, Peter Widmayer: Progressive Merge Join: A Generic and Non-blocking Sort-based Join Algorithm. VLDB Conf. 2002.

[GKK 01] Andreas Gärtner, Alfons Kemper, Donald Kossmann, Bernhard Zeller: Efficient Bulk Deletes in Relational Databases. ICDE 2001: 183-192.

[GL 01] Goetz Graefe, Per-Åke Larson: B-Tree Indexes and CPU Caches. ICDE 2001: 349-358.

[GSZ 01] Jarek Gryz, K. Bernhard Schiefer, Jian Zheng, Calisto Zuzarte: Discovery and Application of Check Constraints in DB2. ICDE 2001: 551-556.
[H 77] Theo Härder: A Scan-Driven Sort Facility for a Relational Database System. VLDB Conf. 1977: 236-244.
[H 81] Wilfred J. Hansen: A Cost Model for the Internal Organization of B+-Tree Nodes. ACM TOPLAS 3(4): 508-532 (1981).
[HD 91] Hui-I Hsiao, David J. DeWitt: A Performance Study of Three High Availability Data Replication Strategies. PDIS 1991: 18-28.
[HR 83] Theo Härder, Andreas Reuter: Principles of Transaction-Oriented Database Recovery. ACM Computing Surveys 15(4): 287-317 (1983).
[JDO 99] Chris Jermaine, Anindya Datta, Edward Omiecinski: A Novel Index Supporting High Volume Data Warehouse Insertion. VLDB Conf. 1999: 235-246.
[JNS 97] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, Rama Kanneganti: Incremental Organization for Data Recording and Warehousing. VLDB Conf. 1997: 16-25.
[JOY 02] Chris Jermaine, Edward Omiecinski, Wai-Gen Yee: Out From Under the Trees. Technical Report, Georgia Inst. of Technology, 2002.
[K 73] Donald E. Knuth: The Art of Computer Programming, Volume III: Sorting and Searching. Addison-Wesley 1973.
[L 95] David Lomet, personal communication, 1995.
[L 98] Per-Åke Larson, personal communication, 1998.
[LG 98] Per-Åke Larson, Goetz Graefe: Memory Management during Run Generation in External Sorting. SIGMOD Conf. 1998: 472-483.
[LKO 00] Mong-Li Lee, Masaru Kitsuregawa, Beng Chin Ooi, Kian-Lee Tan, Anirban Mondal: Towards Self-Tuning Data Placement in Parallel Database Systems. SIGMOD Conf. 2000: 225-236.
[MHL 92] C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, Peter Schwarz: ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. ACM TODS 17(1): 94-162 (1992).
[MN 92] C. Mohan, Inderpal Narang: Algorithms for Creating Indexes for Very Large Tables Without Quiescing Updates. SIGMOD Conf. 1992: 361-370.
[MOP 98] Peter Muth, Patrick E. O'Neil, Achim Pick, Gerhard Weikum: Design, Implementation, and Performance of the LHAM Log-Structured History Data Access Method. VLDB Conf. 1998: 452-463.
[MS 98] Microsoft Corp., SQL Server 7.0, product documentation, 1998.
[O 86] Patrick E. O'Neil: The Escrow Transactional Method. ACM TODS 11(4): 405-430 (1986).
[O 93] Edward Omiecinski: An analytical comparison of two secondary index schemes: physical versus logical addresses. Inform. Sys. 18(5): 319-328 (1993).
[OCG 96] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, Elizabeth J. O'Neil: The Log-Structured Merge-Tree (LSM-Tree). Acta Informatica 33(4): 351-385 (1996).
[PCL 93] HweeHwa Pang, Michael J. Carey, Miron Livny: Memory-Adaptive External Sorting. VLDB Conf. 1993: 618-629.
[RMF 00] Frank Ramsak, Volker Markl, Robert Fenk, Martin Zirkel, Klaus Elhardt, Rudolf Bayer: Integrating the UB-Tree into a Database System Kernel. VLDB Conf. 2000: 263-272.
[S 89a] Betty Salzberg: Merging Sorted Runs Using Large Main Memory. Acta Informatica 27(3): 195-215 (1989).
[S 89b] Michael Stonebraker: The Case for Partial Indexes. SIGMOD Record 18(4): 4-11 (1989).
[S 94] Leonard D. Shapiro, personal communication, 1994.
[SC 92] V. Srinivasan, Michael J. Carey: Performance of On-Line Index Construction Algorithms. EDBT Conf. 1992: 293-309.
[SL 76] Dennis G. Severance, Guy M. Lohman: Differential Files: Their Application to the Maintenance of Large Databases. ACM TODS 1(3): 256-267 (1976).
[SS 95] Praveen Seshadri, Arun N. Swami: Generalized Partial Indexes. ICDE 1995: 420-427.
[V 01] Jeffrey Scott Vitter: External memory algorithms and data structures. ACM Computing Surveys 33(2): 209-271 (2001).
[VZZ 00] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, Alan Skelley: DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. ICDE 2000: 101-110.
[WA 91] Annita N. Wilschut, Peter M. G. Apers: Dataflow Query Execution in a Parallel Main-Memory Environment. PDIS 1991: 68-77.
[ZL 97] Weiye Zhang, Per-Åke Larson: Dynamic Memory Adjustment for External Mergesort. VLDB Conf. 1997: 376-385.
[ZL 98] Weiye Zhang, Per-Åke Larson: Buffering and Read-Ahead Strategies for External Mergesort. VLDB Conf. 1998: 523-533.