B-tree indexes for high update rates

Goetz Graefe

Abstract

In some applications, data capture dominates query processing. For example, monitoring moving objects often requires more insertions and updates than queries. Data gathering using automated sensors often exhibits this imbalance. More generally, indexing streams apparently is considered an unsolved problem.

For those applications, B-tree indexes are reasonable choices if some trade-off decisions are tilted towards optimization of updates rather than of queries. This paper surveys techniques that let B-trees sustain very high update rates, up to multiple orders of magnitude higher than traditional B-trees, at the expense of query processing performance. Perhaps not surprisingly, some of these techniques are reminiscent of those employed during index creation, index rebuild, etc., while others are derived from other well known technologies such as differential files and log-structured file systems.

1 Introduction

Some applications record more data or record data more often than they query them. For example, a fleet management system for a trucking or taxicab company might record each vehicle's latest position more often than the vehicles' positions are queried by a fleet supervisor. In those cases, index and B-tree organization should be optimized for insertion and update performance rather than for query performance, as has been the traditional objective.

Another application domain for the techniques discussed in this survey is indexing of continuous data streams. Filtering streams on the fly is reasonably well understood, but streams that contain identifiers of some real-world objects often need to be matched by identifier and descriptive attribute against static data as well as other streams. Thus, it is imperative that streams can be captured, typically in the order of data arrival, as well as indexed by attributes other than arrival time, sometimes in multiple indexes with multiple orders. For example, a stream of credit card transactions might need, for efficient and near-instantaneous fraud detection, indexing by card number, customer identity or household (a customer might have lost multiple credit cards at the same time), and merchant (a single dishonest employee might fraudulently charge credit cards from multiple customers).

In the following, we assume that update and insertion performance is more important than query performance. If the reader is not concerned about such applications, traditional B-tree optimizations should be applied rather than the techniques surveyed here. Moreover, we assume that any throttling of the workload, e.g., "best effort" recording of current vehicle locations, has already been applied, such that the remaining update requests must indeed be applied to all indexes under consideration. Finally, we assume that hardware assistance has been considered and exploited to the extent possible and appropriate, e.g., disk striping and solid-state disks or disk buffers.

2 I/O optimizations

As with most database operations, focusing on the efficiency of disk I/O is an effective means for improving performance and scalability. However, one must separate between improvements

Dagstuhl Seminar Proceedings 05421
Data Always and Everywhere - Management of Mobile, Ubiquitous, Pervasive, and Sensor Data
http://drops.dagstuhl.de/opus/volltexte/2006/763

to the overall system throughput and improvements to the response time of individual transactions, which may or may not be tremendously interesting here.

There are several very generic performance improvement technologies, e.g., data compression [WKH 00]. In update-intensive workloads, relevant compression applies not only to the data but also to the transaction log. Some compression techniques are surprisingly simple, e.g., truncating leading and trailing zeroes or blanks, or aggregating multiple log records from the same transaction into one in order to save many record headers in the transaction log.

2.1 Prefetch, read-ahead, and write-behind

Write-behind of log pages and of data pages are well known techniques. By itself, is does not improve system throughput, because the amount of writing does not decrease. However, write-behind often enables large writes, which is even more efficient that queued I/O. Moreover, they are helpful in the case of spikes in the workload and they permit additional optimizations. For example, modern disk drives support native command queuing and thus perform better if there are 10s of I/O operations pending at all times [ADR 03].

Read-ahead (as commonly used in large scans) does not apply to large append operations, but it applies to merging an entire batch of modifications into an existing B-tree. When merging multiple B-tree partitions (discussed below) into one, read-ahead with forecasting can improve performance, since merging such partitions is essentially the same problem as merging runs in an external merge sort [G 03a].

Prefetch operations based on individual keys apply not only to retrieval operations, e.g., navigation from a non-clustered index to a clustered index, but also to update operations. However, similar to read-ahead and write-behind, prefetch also does not directly improve system throughput or bandwidth, only response time or latency of individual operations, which might improve system throughput indirectly by reducing concurrency control contention.

2.2 Write-optimized B-trees

In addition to asynchronous I/O, dynamic placement of contents on disk can improve write performance [G 04]. This effect is well known and has been extensively studied for log-structured file systems [OD 89], in particular in the context of RAID storage [PGK 88]. The principal idea behind write-optimized B-trees is to allocate a new location on disk each time a page is written to disk, and to do so as part of the write operation, i.e., subsequent to the buffer manager's replacement decision, and to allocate a page's new location in such a way that multiple concurrent write operations all target the same area on disk.

In order to avoid subsequent updates of neighboring pages, the traditional page chain using physical page identifiers is replaced by a logical page chain using separator keys, i.e., each page carries as lower and upper fences the separator key propagated to the page's parent node when the page was split from its neighbors. In addition to supporting the same consistency checks and other maintenance operations supported by traditional physical page chains, fence keys simplify and improve key range locking, because it is never required to navigate to a neighboring leaf page in order to find the right key to lock. After physical page chains have been replaced by logical fence keys, the only role for physical page identifiers is in child pointers, and only those have to be updated when a node moves to a new location on disk.

In traditional B-tree algorithms, a new location is allocated as part of the B-tree manager's decision to split a node, such that subsequent log records can refer to the page identifier. In write-optimized B-trees, a new page is given a temporary identifier that log records may refer to, and the page is moved as part of the write operation in a way very similar to a page move during B-tree defragmentation. Thus, proven concurrency control and recovery mechanisms apply.

The performance effect of write-optimized B-trees is such that random write operations are converted to large sequential write operations, with a bandwidth advantage of factor 10 or more, at the expense of added maintenance of each node's parent each time a node is written to a new location on disk.

3 Buffering insertions

There are multiple ways to buffer and group new insertions in order to modify each B-tree node less often, with the advantage of fewer disk I/Os, fewer faults in the CPU cache, etc. Query operations either need to search the buffer structure in addition to the B-tree index or they force some or all buffered records into the B-tree index.

For correct transactional execution, it seems that both insertion and deletion in the buffer must be logged; thus the log volume in these methods may exceed the traditional log volume by a factor of three or more. However, only the initial insertion into the first buffer is a user transaction, whereas all subsequent movements of a record can be system transactions that can commit inexpensively without forcing the tail of the transaction log to stable storage.

3.1 Buffering within tree nodes

Several researchers have explored data structures and algorithms that add a large buffer to each interior tree node [A 96, AHV 02, VSW 97]. Often the size of this buffer exceeds the size of the area dedicated to traditional key-pointer pairs, not only because each buffered new record is larger than a key-pointer pair but also because the number of retained records should be larger than the number of key-pointer pairs. When the buffer fills up, appropriate records are pushed down to the child with the most retained records.

It seems that records should be retained only for those children not immediately available in the I/O buffer. Given that most B-trees have a fan-out of 100 or more, and given that in most database servers the RAM size exceeds 1% of the disk size, and given that the B-trees discussed here are among the most active and performance-critical indexes, one may infer that such buffering applies only at the nodes immediately above the leaves. In other words, there may be additional improvement possible beyond published methods that permit buffering in nodes of all B-tree levels.

3.2 Buffering in separate structures

An alternative to buffering insertions in tree nodes is to create a separate data structure to buffer new insertions [LJB 95, MOP 00, OCG 96]. This data structure can be another B-tree or it can be a different type of in-memory data structure, e.g., a hash table. In fact, it can also be a collection of data structures, forming a hierarchy or cascade of staging areas. Interestingly, this organization is reminiscent both of generational garbage collection [U 84] and of the tree structure of the main B-tree index, such that different parts of this data structure buffer insertions pending at different B-tree levels.

Separate structures imply, unfortunately, additional mechanisms for concurrency control and recovery. Thus, a standard index structure that is already implemented might be the preferred mechanism. Otherwise, not locking modes or protocols require correctness arguments, implementation, testing, etc. Perhaps the most desirable implementation avoids both separate structures and modifications of existing structures, and instead only uses existing mechanisms in different ways.

3.3 Buffering in B-tree partitions

One design motivated by the desire to avoid special-case code employs the main B-tree as its own buffer data structure by introducing partitions within each B-tree [G 03a]. By introducing an artificial leading key column, the traditional B-tree structure is retained. The “main B-tree” is defined by a common value for the artificial leading key column, say 0 or null, and one or more “buffers” are defined by different values in that column, say 1, 2, etc.

Traditional buffer management together with a size limit on newly added partitions can ensure that data insertions by user transactions can be processed entirely in memory. In the extreme case, partitions of new insertions can be as small as a single record, i.e., each new insertion defines a new partition and can thus proceed practically without any search or page reorganization within the B-tree. Thus, insertion rates and throughput by user transactions are maximized, at the expense of more merge effort during B-tree optimization and reorganization.

Queries obviously have to search in each partition, using traditional methods for queries that restrict some index columns but not the leading one [LJB 95]. Alternatively, query activities may force some merge activities, executed prior to actual data retrieval and implemented using system transactions. Thus, B-tree maintenance work that traditionally is part of update operations is shifted to query operations. The main differences are when the work is done and how it is synchronized without transactions; the difference in how it is done seems minor in comparison. In the extreme case, a query may force complete merging and optimization of all partitions, maybe excepting one partition targeted by current insertions.

Some interesting aspects of such B-trees are (i) that the reorganization operation that combines multiple partitions into one is very similar to a merge step in a traditional external merge sort, (ii) that such merge operations can execute as system transactions and commit very small key range at a time, (iii) that merge and reorganization operations can pause and resume at any time in response to load spikes etc., and (iv) the same technique can aid bulk deletions, i.e., B-tree entries to be deleted are moved by small system transactions into one dedicated partition and then deleted in one fast user transaction that cuts multiple full pages from the B-tree.

3.4 Graceful degradation

In addition to raw performance improvements, buffering insertions also enables graceful degradation after errors in cardinality estimation during query optimization. Today, query optimization can choose between row-by-row update processing and index-by-index update processing. Updating row-by-row implies maintenance of all appropriate indexes immediately for each row. Updating index-by-index means that all changes are applied to one index at a time, possibly after splitting each update into a deletion and an insertion, sorting the changes, and re-combining changes if appropriate; a generalized version of techniques described in [GKK 01] implemented in Microsoft SQL Server since 1998. Row-by-row updates are most appropriate for small changes, whereas index-by-index updates are more efficient for large updates, in particular if there are more individual changes that leaf pages in an index.

Buffering insertions as described here opens up another option, namely row-by-row processing with the I/O pattern and efficiency better than index-by-index processing, albeit with the disadvantage of non-optimal indexes left behind. Thus, a query execution plan may prescribe row-by-row update processing due to an anticipated small update set, yet the actual execution may determine that the update set is rather large. In such a case, an update plan may apply updates row-by-row until the actual size of the update set becomes apparent and then switch to buffered or partitioned updates. While it is possible to implement graceful degradation from row-by-row to index-by-index updates using conditional execution in a traditional query execution plan, assigning a new partition identifier (artificial leading key column) to index changes is much simpler and it achieves even faster update performance.

4 Differential files and indexes

While the designs discussed in the prior section are able to buffer insertions, they cannot buffer other update operations, i.e., modifications or deletions. However, they can be extended to do so, by adapting ideas from differential files [SL 76] and specializing them to B-tree indexes. Interestingly, some B-tree adaptations for multi-version concurrency control and for historical indexes are very similar, including the logic required during query processing.

The basic approach is to append records that invalidate prior records without actually modifying those prior records. In an update, a new record supersedes the prior B-tree entry with the same key. In a deletion, the newly appended record simply indicates the end of the history for a particular key.

Query evaluation needs to search the history for each particular key, either for the most current state (for traditional query semantics) or for the state at a particular time (for point-in-time historical queries). Merge operations may condense the history of keys depending on the desired future query capabilities.

In other words, like buffering insertions, buffering updates and deletions in differential B-trees trades query performance in favor of update performance. Turning random single-record insertions, deletions, and updates into append operations with large sequential write operations can improve the sustained update throughput by two orders of magnitude.

5 Transaction guarantees

Another opportunity for performance improvement may be to weaken transactional guarantees for some indexes, in particular for redundant non-clustered indexes. We consider three techniques that do so, one that dilutes the separation of individual transactions by batching, one that weakens guarantees in case of system failures, and one that records changes only in the transaction log without even attempting to apply them to the index, with the implicit danger that the attempt to apply such changes later might fail. Obviously, these techniques only apply if the remaining transactional guarantees are still strong enough for the application at hand.

5.1 Log-only operations

If the index maintenance cannot keep up with the update stream, maybe at least the transaction log can. In that case, one could write logical *redo* records to the transaction log and apply them later, essentially using *redo* recovery. Of course, this process violates multiple traditional assumptions about logging, e.g., that *redo* operations are always physical operations that already happened, that *redo* operations cannot fail, etc. However, depending on the application, such failures might not be total disasters and could be ignored, for example, when some individual location reports in a vehicle tracking application cannot be recorded in the historical index. Clearly, this idea might apply, but details need to be worked out, e.g., what transaction commit truly promises and what it guarantees, how checkpoints work and what they guarantee, etc.

5.2 Non-logged B-trees

Some database systems employ special techniques during index creation such that the contents of the new index do not appear in the transaction log. Instead, only page allocation and catalog changes are logged. Failure during index creation results in deallocation of those pages and erasure of the new index in the catalogs. Index creation ends with flushing all newly allocated and filled pages to disk, and subsequent backup operations of the database or even the transaction log capture those new pages. Subsequent user transactions log their changes to the new index in the usual way.

This idea can be extended in the following way. If an index is truly redundant similar to a traditional cache, and if erasing the index during media or system recovery is acceptable, then all operations on this index may be non-logged, i.e., only space allocation is logged. This specifically includes user transactions running after index creation is complete. Rollback of a user transaction is driven by virtual log records attached to the transaction descriptor in memory, similar to virtual log records used in other transaction processing designs [G 04]. Details of this technique have not been worked out or published at this point, but the technique seems promising for some applications, in particular for temporary caches and indexes that exist only in memory.

5.3 Batching updates

Finally, one may group multiple update operations and transactions into a single transaction. However, it seems important to separate the transaction semantics from the data structure. For example, many small user transactions may all insert into a single buffer as described above, leaving it to a subsequent system transaction (or series of small system transactions) to merge such insertions into the main B-tree. In other words, it might not be necessary or advantageous to modify or weaken the boundaries and semantics of user transactions in order to achieve the desired advantages in performance and scalability.

6 Summary and conclusions

In summary, if one is willing to accept deterioration of query performance by an order of magnitude, e.g., due to searching multiple partitions, update and insertion performance can be improved by two orders of magnitude or more, e.g., by turning insertions to append operations and by avoiding random in-place writes into large sequential writes to newly allocated disk space. Less dramatic tradeoffs also exist. While most applications issue more queries than update requests and thus demand a query-optimized database organization, some applications (e.g., tracking moving objects) record more data changes than they answer queries (e.g., about current object location). For those applications, numerous techniques are readily available for implementation by database vendors. Some are even available to database users, e.g., by introducing an artificial leading key column in the visible database schema and exploiting it for index creation and possibly for index maintenance during bulk operations [G 03b].

This survey is an attempt to list a variety of possible techniques. Among them, the most promising might be write-optimized B-trees, partitioned B-trees using partitions to buffer insertions or all modifications in the manner of differential files, and non-logged B-trees. However, this intuitive appraisal requires validation using prototyping or even product implementations.

Numerous open research questions present themselves, including the question for additional or better trade-offs between update and query performance, a comparative performance evaluation of the methods described above based on an appropriate benchmark, adaptation of some of the techniques discussed above to other index structures, in particular to multi-dimensional indexes such as UB-trees and R-trees and to materialized and indexed views, and integration of query and update processing with database maintenance operation such as consistency checks, defragmentation, and statistics refresh for query optimization. Maybe the present survey will stimulate and structure such research.

Acknowledgments

Theo Härder and Bernhard Mitschang read earlier incomplete drafts and contributed multiple very helpful suggestions.

References

[A 96] Lars Arge: Efficient External-Memory Data Structures and Applications. University of Aarhus (Denmark), 1996.

[ADR 03] Dave Anderson, Jim Dykes, Erik Riedel: More Than an Interface - SCSI vs. ATA. Conference on File and Storage Technology (FAST), March 2003.

[AHV 02] Lars Arge, Klaus Hinrichs, Jan Vahrenhold, Jeffrey Scott Vitter: Efficient Bulk Operations on Dynamic R-Trees. Algorithmica 33(1): 104-128 (2002).

[G 03a] Goetz Graefe: Sorting and Indexing with Partitioned B-Trees. Conference on Innovative Data Systems Research, 2003.

[G 03b] Goetz Graefe: Partitioned B-trees - a user's guide. Datenbanksysteme für Business, Technologie und Web (BTW) 2003: 668-671.

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

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

[JNS 97] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, Rama Kanneganti: Incremental Organization for Data Recording and Warehousing. VLDB Conference 1997: 16-25

[LJB 95] Harry Leslie, Rohit Jain, Dave Birdsall, Hedieh Yaghmai: Efficient Search of Multi-Dimensional B-Trees. VLDB Conference 1995: 710-719.

[MOP 00] Peter Muth, Patrick E. O'Neil, Achim Pick, Gerhard Weikum: The LHAM Log-Structured History Data Access Method. VLDB J. 8(3-4): 199-221 (2000).

[OCG 96] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, Elizabeth J. O'Neil: The Log-Structured Merge-Tree (LSM-Tree). Acta Inf. 33(4): 351-385 (1996).

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

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

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

[U 84] David Ungar: Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. Software Development Environments (SDE), ACM SIGPLAN Notices 19(5): 157-167 (1984).

[VSW 97] Jochen Van den Bercken, Bernhard Seeger, Peter Widmayer: A Generic Approach to Bulk Loading Multidimensional Index Structures. VLDB Conference 1997: 406-415.

[WKH 00] Till Westmann, Donald Kossmann, Sven Helmer, Guido Moerkotte: The Implementation and Performance of Compressed Databases. ACM SIGMOD Record 29(3): 55-67 (2000).