

Algorithms for merged indexes

Goetz Graefe

HP Labs ¹
Goetz.Graefe@HP.com

Merged indexes are B-trees that contain multiple traditional indexes and interleave their records based on a common sort order. In relational databases, merged indexes implement “master-detail clustering” of related records, e.g., orders and order details. Thus, merged indexes shift de-normalization from the logical level of tables and rows to the physical level of indexes and records, which is a much more appropriate place for it. For object-oriented applications, clustering can reduce the I/O cost for joining rows in related tables to a fraction compared to traditional indexes, with additional beneficial effects on buffer pool requirements.

Prior research has covered merged indexes without providing much guidance for their implementation. Enabling the design proposed here is a strict separation of B-tree and index into two layers of abstraction. In addition, this paper provides algorithms for (i) concurrency control and recovery including locking of individual keys and of complex objects, for (ii) data definition operations including adding and removing individual indexes in a merged index, for (iii) update operations including bulk insertions and bulk deletions, for (iv) enforcement of relational integrity constraints from uniqueness constraints to foreign key constraints, and for (v) query processing including caching in query execution plans dominated by nested iteration and index navigation.

Within such a merged index, the set of tables, views, and indexes can evolve without restriction. The set of clustering columns can also evolve freely. A relational query processor can search and update index records just as in traditional indexes. With these abilities, the proposed design may finally bring general master-detail clustering and its performance advantages to traditional databases.

1 Introduction

Some database management systems provide master-detail clustering in their on-disk data structures, with the aim of saving I/O operations in online transaction processing, data warehousing, and data mining. Not only can a single I/O operation fetch multiple related records from multiple tables or indexes, but the number of buffer frames required to hold an entire query’s working set also decreases due to master-detail clustering.

1.1 *The value of master-detail clustering*

In general, master-detail clustering should apply not only to pairs of tables or indexes but to any number of them, such that entire complex objects can be stored, fetched, buffered, and saved together. For example, it is beneficial if orders and their order details can be clustered, but it is even better if an entire customer object including orders, shipments, invoices, payments, etc. can be clustered together. In every query that assembles a com-

¹ Palo Alto, CA. Much of this research was performed while employed at Microsoft.

plex object by joining these tables and indexes, I/O cost is reduced not by percentages but by a factor. As others have observed, “optimization techniques that reduce the number of physical I/Os are generally more effective than those that improve the efficiency in performing the I/Os” [HS 04].

As disk performance improves, access latency improves much more slowly than transfer bandwidth. For example, the I/O size for which access time equals transfer time has grown from 10-30 KB in early relational databases to 200-500 KB today. Thus, today’s databases ought to employ large I/Os [GG 97], and techniques are needed that complement large pages, e.g., interpolation search [G 06] and master-detail clustering.

Due to historical trends of disk access times and disk bandwidth, high-end servers often employ “short-stroking,” which is “the practice of formatting a disk drive such that data is written only to the outer sectors of the disk’s platters. In I/O-intensive environments, this increases performance, as it reduces the time spent by the drive actuator seeking sectors on a platter. However, short stroking also wastes a substantial portion of the disk drive’s capacity” [S 05]. Master-detail clustering can cut the number of I/O operations, because related records from multiple indexes and tables can be fetched with a single I/O operation. For example, if many applications access orders and order details almost always together, master-detail clustering can cut I/O in half and only half as many disk arms are needed. Thus, each disk can be loaded with twice as much data and costs for disks, disk controllers, etc. are cut in half. Such cost savings make merged indexes attractive for high-end applications. Cost savings of merged indexes may rival those of compression in data warehouses [PP 03].

1.2 Master-detail clustering in B-trees

In a database management system that relies on B-tree indexes as the main indexing mechanism, as most commercial systems do, an appropriate implementation strategy for master-detail clustering is to map multiple indexes to a single B-tree. If, in the earlier example, each such index has the customer identifier as the leading key column, all records contributing to a customer object are naturally clustered together within the B-tree. As orders and their order details are often accessed by the order number, it seems beneficial to merge the index on orders.orderkey and the index on orderdetails.orderkey into a single B-tree with orderkey as the leading column.

...
Order 4711, Customer “Smith”, ...
Order 4711, Line 1, Quantity 3, ...
Order 4711, Line 2, Quantity 1, ...
Order 4711, Line 3, Quantity 9, ...
Order 4712, Customer “Jones”, ...
Order 4712, Line 1, Quantity 1, ...
...

Figure 1. Record sequence in a merged index.

Figure 1 shows the sort of records within such a B-tree. The sort order alone keeps related records co-located; no additional pointers or references between records are needed.

In addition to performance, B-trees provide effective and efficient space management. If all data about multiple customers fit on a single page, they will share a leaf page. If a

single customer object requires multiple pages, neighboring B-tree leaves will be used. Most B-tree implementations attempt to allocate neighboring leaves contiguously on disk such that all data about a customer can be fetched with a single disk access.

In its most limited form, master-detail clustering combines two non-clustered indexes, e.g., associating two lists of row identifiers with each key value. Alternatively, master-detail clustering may merge two clustered indexes but not admit any non-clustered indexes. The design pursued here accommodates any combination of clustered and non-clustered indexes in a single B-tree, thus enabling clustering of entire complex objects.

In merged indexes, search keys and clustering keys may differ among the participating individual indexes, i.e., participating indexes may differ not only in their overall record structure and contents but also in their keys. The proposed design permits adding and removing individual indexes from a merged B-tree index at any time, with no change in search efficiency. While the performance effects of merged indexes are similar to those of traditional master-detail clustering, this flexibility and thus power for the database administrator distinguishes merged indexes from prior designs for master-detail clustering.

For operations on individual keys and records, including both queries and updates, concurrency control and recovery are no more complex than in traditional indexes, and the different keys do not affect search performance. Even bulk operations such as data load are as fast as in databases and tables using traditional indexes.

1.3 Purpose and scope

The purpose of this research effort is to describe a very flexible yet simple design for merged indexes. The goals outlined above can be achieved with techniques known today, primarily by combining existing data structures and algorithms in new ways. Thus, this research points more often to new combinations of old techniques than to new fundamental techniques, which reduces the implementation effort for merged indexes.

It may be worth pointing out that the purpose here is not to recommend that any and all indexes should participate in merged indexes. If it is important to scan a specific index as fast as possible, interleaving its records with those of another index is counterproductive. Merged indexes are an additional capability available to database administrators during physical database design. Careful index tuning remains a necessity for high performance and high scalability, whether based on manual tuning or a software tool.

The remainder of this paper, after reviewing related work, focuses on data structures, concurrency control and recovery, index creation and maintenance, and query processing techniques, followed by a summary and some conclusions of this research.

2 Related work

Hierarchical and network database management systems have long clustered records of different types by “set membership,” and their storage engines have supported such clustering similarly for a long time. Merged indexes are different as they cluster records by key values rather than record-to-record pointers, and thus merged indexes fit more readily into the architectures of modern database management systems. They are a versatile yet simple way of bringing master-detail clustering to relational database management systems and data warehouses.

Härder’s notion of a “combined image” permits non-clustered indexes with equal search key to share a B-tree [H 78]. Figure 2 illustrates a non-clustered index for three tables. The search key is fixed for the B-tree, as is the set of indexes it contains.

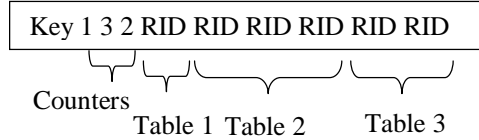


Figure 2. Combined non-clustered index.

In comparison, a merged B-tree permits both clustered and non-clustered indexes. Merged indexes permits the set of indexes to change dynamically, and it permits indexes that agree on only some of their key columns. If they agree on none of their key columns, the merged index degenerates into a “concatenated index.”

Härder and Reinert explored the advantages of such indexes for verification of foreign key constraints in relational databases [HR 96]. These advantages included efficient verification of foreign key constraints as well as efficient join operations, both merge joins for large results and index nested loops joins for small results. In index nested loops join, locality benefits affect disk pages loaded in the buffer pool and, in the ideal case, even the appropriate cache lines in the CPU’s data cache.

Valduriez renewed this idea in join indices, which are “prejoined relations ... stored separately from the operand relations”. It “is a binary relation” that “only contains pairs of surrogates”. Valduriez also proposed a “multi-relation clustering scheme used in combination with join indices” without much elaboration (all quotes from [V 87]).

In comparison, merged indexes support joins and foreign key constraints more generally than combined images and join indices because the set of key columns might vary among the individual indexes. For example, a single merged index may support joins and foreign key verification between customers and orders as well as between orders and order details.

Oracle’s main database product permits multiple tables clustered on the same set of columns, up to 32 tables and up to 16 columns. Records are co-located using a B-tree key or a hash function. Each value of the cluster key is stored only once. Tables can be added and removed, but the cluster key is fixed for all tables and indexes. For example, a single cluster cannot support efficient search of orders by customer identifier and of order details by customer identifier and order number. Thus, Oracle’s design will rarely permit clustering more than two levels of components within a complex object.

In comparison, merged indexes conceptually store each key column in each record, relying on prefix truncation [BU 77] and other compression techniques to save storage space. However, merged indexes support much more flexibility with respect to clustering keys, search keys, complex objects, and future additions to an existing merged index.

There are three dimensions useful to classify merged indexes or equivalent storage structures. First, the contents may be pointers only as in Figure 2, they may be full rows only as in Oracle’s design, or they may be a mixture of clustered and non-clustered indexes. Second, the set of individual indexes within a merged index may be fixed for the entire lifetime of the merged index, it may permit removal only, or it might permit free addition and removal. Third, clustering may require equal keys as in Härder’s and Oracle’s designs, it may support arbitrary keys as long as those are declared when the merged index is first created, or it may permit declaration of additional keys while the merged

index evolves over time. In all three dimensions, the proposed design for merged indexes support the most general choices with minimal impact on efficiency.

Note that merged indexes are very different from “star indexes” in IBM’s Red Brick product [F 94]. A star index is a B-tree index on a single fact table within a relational data warehouse, with uniform record format throughout the index and with some of the foreign key values replaced by record identifiers of the dimension tables.

3 Data structures

In order to simplify design and implementation of merged indexes, a crucial first step is to separate implementation of the B-tree structure from its contents. One technique is to employ normalized keys, such that the B-tree structure manages only binary records and binary keys. Thus, all comparison operations compare keys using binary comparisons using a simple method such as “memcmp()” or even an equivalent hardware instruction. This implementation of B-trees also simplifies implementation of B-tree optimizations such as prefix and suffix truncation [BU 77], order-preserving compression, and many optimizations for effective use of CPU caches [GL 01, L 01].

The remaining crucial issue, therefore, is to map index keys with multiple columns, collation sequences, etc. to binary B-tree keys. For traditional indexes with only a single logical index in each B-tree, fairly simple techniques suffice: unsigned integers are immediately usable (perhaps after reversing the byte order [L 01]), signed integers require toggling the sign bit, a NULL indicator bit must precede each value, variable-length strings require a termination symbol, multiple columns can be simply concatenated, etc.

3.1 Key format

Mapping multi-column keys to binary strings is a bit more complex in merged indexes than in traditional indexes, however, in particular if adding and removing any index at any time is desired and if individual indexes may have different key columns.

FIELD VALUE	FIELD TYPE
“Customer identifier”	Domain tag
123	Data value
“Order number”	Domain tag
4711	Data value
“Index identifier”	Domain tag
“Orders.orderkey”	Identifier value
“2006/12/20”	Data value
“Urgent”	Data value
...	Data values

Figure 3. B-tree record in a merged index.

Merged indexes permit growing and shrinking the set of individual indexes merged into one B-tree. Thus, it is essential to design a flexible mapping from keys in the index to byte strings in the B-tree. A tag that indicates a key column’s domain and precedes the actual key fields, as shown in Figure 3, can easily achieve this.

In practice, different than illustrated in Figure 3, a domain tag will be a small number, not a string. It is possible to coalesce the domain tag with the Null indicator (omitted in

Figure 3) such that the desired sort order is achieved yet actual values are stored on byte boundaries. Similarly, the index identifier will be a number rather than a string.

While it is possible that a tag indicates merely the key's type, e.g., "32-bit unsigned integer," it is desirable to indicate a domain or a distinct type, e.g., "order number" or "invoice number." If tags merely indicate types, orders and invoices may be interleaved in the B-tree pages for this example. If domain tags are used, orders and invoices naturally separate into two groups, yet order details naturally cluster with the appropriate order records based on equal domain tag and order numbers.

Domain tags are not required for all fields in a B-tree record. They are needed only for key columns, and more specifically only for those leading key columns needed for clustering within the merged index. Following these leading key columns is a special tag and the identifier of the individual index to which the record belongs. For example, in Figure 3, there are only 2 domain tags for key values plus the index identifier. If there never will be any need to cluster on the line numbers in order details, only leading key fields up to order number require the domain tag. Thus, the per-record storage overhead for merged indexes is minimal and may indeed be hidden in the alignment of fields to word boundaries for fast in-memory processing.

All fields following the index identifier store only the field but no domain tag. In a clustered index, one may expect that a small minority of fields require a domain tag. The columns following the index identifier may define sort order and thus contribute to efficient searches; they just do not permit finer clustering. In Figure 3, for example, the date value might be part of the sort order and the search key, but it does not require a domain tag because the index identifier precedes it.

Incidentally, if the index identifier is made the first column, records belonging to that individual index will be clustered within the merged index, with no interleaving records from other indexes. The effect is that this index is concatenated with other indexes within the merged index. With respect to storage space, since the index identifier is common to all records in many pages, prefix truncation eliminates it from all records in most pages, both in leaves and in upper B-tree nodes.

3.2 Page format

In general, a traditional page format using an indirection vector to manage variable-length records [L01]. In order to balance the increased key size due to domain tags, key prefixes common across an entire page are truncated and stored only once per page, and that suffixes are truncated from separator keys when they are posted in a parent during a leaf split [BU 77].

For maximal prefix and suffix truncation, split operations do not necessarily split at the center and instead split only near that point, e.g., such that each resulting leaf will be between 40% and 60% full. During index creation and bulk insertion, each leaf might be between 75% and 95% full if the desired fill factor is 85%. Within the range of possible split points, the shortest possible separator is chosen [BU 77].

For some operations, notably initial B-tree creation and bulk load operations, partitioned B-trees offer substantial advantages, as described in detail elsewhere [G 03]. Partitioned B-trees require an artificial leading key column, which indicates the partition in a B-tree index or the run number if a B-tree index is used to store runs in an external merge

sort. The partition number is likely to be constant within all nodes other than the B-tree root and prefix truncation will therefore hide it in practically all B-tree leaves and nodes.

3.3 *Unique indexes*

In order to preempt some possible misconceptions about merged indexes and uniqueness constraints including primary key constraints, it may be useful to point out that these concepts are in fact orthogonal. While it is true that two successive records from the orders table might not be next to each other in a merged index even if order number is the leading key column, insertion of a duplicate key into a unique index will still fail because it will find its intended insertion point already occupied.

During index construction when a uniqueness constraint is first declared, it is desirable to shift verification of the uniqueness constraint from the insertion logic to the sort operation. If no sort operation is required because an appropriate sorted source exists, an operation very similar to stream aggregation, inserted between the sorted source and the index insertion logic, can verify that the future index is indeed free of duplicate keys.

3.4 *Indexes on materialized views*

For the storage layer in a database management system, there is little difference between indexes on standard tables and indexes on materialized views. Thus, it is perfectly possible that a merged index combines individual indexes from multiple tables and multiple views. For example, in order to keep with each order its total value, i.e., the product of quantity and price summed up over all order details, a materialized view with a “group by” clause on order number can be indexed on its primary key and merged with other indexes on order number, notably indexes on the orders and order details tables.

3.5 *Computed columns*

A small difficulty arises if all components of a complex object are to be clustered with the object’s root yet some sub-components are “strong entities” with their own globally unique identifiers. A typical example is a customer object with orders and order details. The B-tree search key that permits clustering all these row and record types is the customer identifier. An appropriate column exists in the orders table as a foreign key, but it does not exist in the table of order details because orders have a globally unique identifier. Thus, it may seem impossible to cluster order details with customers and orders.

COLUMN	DEFINITION
OrderKey	Int foreign key references Orders
LineNo	Int
CustKey	Select CustKey from Orders where ...
Quantity	Numeric
...	...
	Primary key (OrderKey, LineNo)

Figure 4. Table definition for Order Details.

The solution, illustrated in Figure 4, is to propagate the customer identifier to the table of order details using techniques for computed columns and materialized views. A computed column is declared in the table of order details, e.g., as “select custkey from orders

where `orders.orderkey = orderkey`.” This column is indeed stored in indexes for the table. Like other computed columns, this column cannot be updated directly by user commands. Instead, it is recomputed automatically whenever one of the columns `orders.orderkey`, `orders.custkey`, or `order details.orderkey` is modified, quite similar to cascading actions for foreign key constraints. In a sense, the table of order details has become a materialized view joining orders and the traditional table for order details.

This solution hints at further considerations for master-detail clustering and merged indexes. Relational databases would generally benefit from a separation of table definition and storage definition, because redundancy among indexes for tables and for views could be reduced. Among possible approaches, concepts explored for GMAP [TSI 96] could prove useful, in particular if they are expressed using a SQL dialect.

3.6 Hash indexes and hash clustering

A simpler case is a column computed from values within the same row such as a hash function applied to one or more columns. A B-tree index on the result of a hash function is practically equivalent to a hash index, in particular if interior nodes of the B-tree can be as large as the directory of a hash index and if interpolation search is used within each such large node.

Master-detail clustering based on hash functions can be supported more easily within such a B-tree than in traditional hash indexes. For example, index growth or a non-uniform distribution of hash values are accommodated by splitting B-tree leaves.

In such a B-tree, entries with equal hash value can be organized using additional sort columns. For example, customer and order rows may be hashed on `customerkey`, with the result serving as the leading B-tree key. The columns `customerkey`, `orderkey`, etc. may be additional keys in the B-tree. This index is an unsorted hash index as well as a sorted B-tree index, with an overall sort order that the query optimizer can exploit in grouping, merge join and index nested loops join.

Using this design, very little code beyond standard B-tree code is required to augment a database engine with hash indexes as well as master-detail clustering on hash values. For example, existing code can be used for concurrency control, index creation, data load, defragmentation, consistency checking, etc. I/O performance in this design is similar to traditional hash indexes if non-leaf layers can be retained in the buffer pool. CPU performance is similar if B-tree nodes can be very large, if the hash value distribution is approximately uniform, and if interpolation search is used within each node.

3.7 Summary of data structures

In summary, the separation of B-tree implementation and contents permits exploiting performance enhancements such as prefix and suffix truncation as well as adaptation and generalization of master-detail clustering, which previously has been used in pre-relational database management systems and only in limited forms in modern relational database management systems. Relatively simple data structures and a fairly straightforward mapping permit powerful clustering of entire complex objects without violating relational theory, relational languages, or their design philosophy. Expected performance advantages, in particular for online transaction processing applications on high-end servers, can translate into substantial cost advantages.

4 Concurrency control and recovery

Concurrency control and recovery have traditionally been concerns managed in the lower levels of the storage engine, and merged indexes do not require a change of that basic design.

4.1 *Key range locking*

Traditional locking of key values and key ranges [GLP 75, L 93, M 90, ML 92] applies to merged indexes in the same way as to traditional B-tree indexes. The same trade-offs and optimizations apply. Of course, the key values being locked must be the keys as outlined in Figure 3, i.e., they must include domain tags and the index identifier. If locking is implemented in the software layer that interprets B-tree records as binary strings, this requirement is readily met.

In merged indexes, key range locking might lock a record from one individual index in order to insert a record from another individual index. For example, if orders and order details are ordered in the most obvious way, adding a detail record might lock the next order, assuming that “next key locking” is used for range locking [L 93, M 90]. In a design that makes details follow the header, e.g., orders and invoices follow customers and order details follow individual orders, there might be fewer surprising concurrency control conflicts if the prior key is locked for a range rather than the next key.

Maybe more importantly, lock modes should be designed such that a key and a range together can be intention-locked, in the sense of multi-granularity locking [GLP 75], and the gap between two keys and an individual key value each can be locked separately [G 07]. For performance reasons, it makes sense to implement this not as two levels of lockable resources (comparable to the standard example with pages and files) but as a single resource with many possible lock modes [G 07, L 93].

Another interesting case is locking individual keys when dropping an individual index within a merged index, i.e., when bulk-deleting many keys from the B-tree. It is quite possible that many neighboring keys belong to other individual indexes and might be locked by concurrent transactions. If a deletion only updates the record’s ghost bit, marking the record invalid and non-existent for subsequent queries, the deletion operation only requires key value locks that do not interfere with these other transactions. Asynchronous ghost cleanup will eventually erase those records when their space is needed.

4.2 *Locking complex objects*

If locking and concurrency control are left to the lowest levels of the storage engine, i.e., the level that manages B-tree keys and records as binary strings without regard to individual indexes within a merged index, it might seem impossible to lock entire complex objects with a single lock or with only a few locks. Strictly speaking, this is true, but a combination of other design choices effectively results in locks on entire complex objects.

Splitting B-tree leaves not at the center but at a point near the center chosen to maximize the effects of prefix truncation and suffix truncation also promotes leaf boundaries that coincide with boundaries of complex objects. For example, if a leaf split occurs between two records that both belong to an index on order details, the separator key in the

parent must include not only an order number but also a line number. If, on the other hand, the split point is chosen to fall between two orders, the separator key can be shorter. Moreover, if the size of an order and its details roughly coincides with the size of a leaf page, this order like will be stored in a dedicated leaf page, with the obvious beneficial effect on prefix truncation. Optimization of suffix truncation automatically achieves this effect on locking.

In order to achieve object-level locking, the locking strategy should require locks not only in B-tree leaves but also on separator keys in the leaves' parents [G 07]. Some commercial database management systems lock both leaf pages and individual keys within; the proposed method acquires the same number of locks but instead of leaf pages it locks key values and key ranges in the leaves' parent nodes.

If objects are small such that multiple objects fit in a leaf, this design locks multiple objects at a time. If an object is very large, it might require multiple locks at the parent level. While this is not strictly equal to locking complex objects, it achieves its main effects, namely locking related component records with very few locks rather than locking them one at a time, and it achieves this effect with fairly traditional locking mechanisms.

The locking discipline uses standard hierarchical locking with standard intention locks. If most complex objects are very large, locking might start in the leaves' grandparents rather than in the parents. Note also the standard techniques for lock escalation and de-escalation apply [J 91, LC 89], such that a transaction may lock an entire leaf's key range by locking a key in the parent but reducing that lock to an intention lock after acquiring a lock on a key in the leaf in the case of contention. Interestingly, initial acquisition of a large lock and subsequent lock de-escalation can improve performance because the smaller locks can be acquired without concern about conflicts and thus they can be inserted into the lock manager's hash table without preceding search [GL 92].

4.3 Locking summary records

Summary records, e.g., records representing or indexing rows in a materialized view that summarize the total value of each order by summing up order details, can be concurrency bottlenecks and might therefore warrant special lock types. Escrow locks [O 86] can be employed and can be augmented with intent-to-escrow lock modes for use in upper B-tree levels [GZ 04]. Traditional shared and exclusive locks can be combined with escrow locks within the same B-tree, such that rows from tables and materialized and indexed views can be co-located in the same merged index and each locked appropriately for its use. A typical example is a merged index with an index from the table orders and an index on a view that summarizes order details for each order.

If it makes sense for the data in the merged index, locking in escrow mode even at the parent level is possible, in particular if lock de-escalation is available. All intention locks are compatible with all other intention locks [K 83]; therefore, two transactions might concurrently hold, for example, intent-to-share and intent-to-escrow locks for the same complex object.

4.4 Recovery considerations

The concurrency control mechanisms above are very standard with respect to their needs for write-ahead logging and recovery. Escrow locks for summary records are aug-

mented with commit-time-only exclusive locks that follow the normal logging and recovery regimes for traditional updates under traditional exclusive locks [GZ 04]. Thus, advanced optimizations are possible such as lock acquisition during the log analysis phase of crash recovery with transaction processing during redo and undo phases. Other possible optimizations include database mirroring with automatic failover.

The essence of the techniques proposed here is to employ entirely ordinary B-tree techniques for B-trees managing strictly binary keys. The meaning of these binary keys is immaterial, including the mapping from multiple tables, views, indexes, column types, column sequence, sort order, collation sequence, etc. The advanced locking proposals above, e.g., locking key ranges in the parent and grandparent level of a B-tree, only affect the concurrency among multiple transactions; they do not affect their logging or recovery including database mirroring.

One particular logging optimization may be worth calling out. A standard use of system transactions is to remove ghost records left behind by user transactions during deletions. Separation of logical deletion (turning a valid record into a ghost) and physical removal (reclaiming the record's space) serves three purposes, namely simplified roll-back for the user transaction if required, increased concurrency during the user transaction (locking a single key value rather than a range), and reduced overall log volume. If the system transaction performing the ghost clean-up can capture transaction start, record deletion, and transaction commit in a single log record, there is never any need to log undo information, i.e., the deleted record's contents.

For clustered indexes, this may represent a substantial reduction in log volume. In merged indexes, this also applies to records left behind while dropping an individual index. Thus, it is possible to drop an individual index without logging its contents, albeit with many subsequent system transactions and one log record for each of them.

4.5 Summary of concurrency control and recovery

The value of the chosen approach to concurrency control and recovery for merged indexes and complex objects is that it relies entirely on well-understood mechanisms, many of them already implemented in most commercial database systems. The additional facilities, e.g., range locking at the parent level within a B-tree or escrow locks for summary records, are not strictly required to make merged B-trees work. If implemented, they benefit merged indexes and traditional indexes alike. The reliance on traditional B-trees for master-detail clustering and complex-object clustering reduces the variety of on-disk data structures and thus the required effort for development and testing of improvements and of new capabilities.

5 Index creation and maintenance

Similar to concurrency control and recovery for merged indexes, which are characterized primarily by using existing techniques and benefiting from their fortuitous interactions, index creation for merged indexes requires combining mostly traditional techniques with a few recent innovations. This section covers index creation, update operation including bulk updates, and finally algorithms for adding and removing individual indexes in existing, populated merged indexes.

5.1 Index creation

Initial creation of a merged index is fairly simple if all participating indexes are still empty – it is merely a metadata operation to define the appropriate catalog entries. If some or all of the participating tables and views are non-empty, there are multiple algorithms for creating the initial merged index and its B-tree.

The obvious strategy employs a full outer join of all individual indexes. Specifically, after the data for each individual index is scanned and sorted as appropriate for a traditional single-index B-tree, a full outer join using a merge join algorithm combines them such that the new B-tree can be loaded with the desired fill factor, on-disk layout, etc.

A less obvious strategy employs techniques for bulk insertion and incremental index operations, to be discussed shortly.

5.2 Insertion, update, and deletion

Ordinary update operations work on merged indexes just as they do for traditional, single-table indexes. The only difference, of course, is that the search key must be augmented with domain tags as discussed earlier. Duplicate detection in unique indexes relies on finding a key already present.

Large updates can be optimized using index-by-index updates with sorted streams of change items instead of row-by-row updates with random search operations in each non-clustered index of the affected table or view [ABC 01]. Split-sort-collapse sequences for updates of keys in unique non-clustered indexes with the possibility of false uniqueness violations also apply to unique indexes within merged indexes. The only difference, of course, is that the cost function for alternative update strategies must reflect the size of the merged index rather than the individual single-table index.

More interesting is a complex update plan with verification of foreign key constraints and perhaps even cascading. Given that merged indexes are ideal precisely for data with foreign key constraints, optimal performance and locality require that verification of constraints be deeply integrated with index maintenance. In that case, the traditional phase separation between index maintenance and constraint verification is not guaranteed, and interactions are possible similar to the Halloween problem [GLS 93, M 97].

For the execution of such integrated operations, it is an open question whether pipelining change items between update operations is sufficient or a new integrated update operation is required. In either case, if update operations employ read-ahead or prefetch in the indexes they maintain, redundant prefetch requests should be avoided while updating of merged indexes.

5.3 Bulk insertion and bulk deletion

Traditionally, bulk operations such as data load created a dilemma for database administrators. The choice was either to update indexes slowly and incrementally or to drop indexes and recreate them after completion of the load operation. This dilemma applies to merged indexes in an even worse way because dropping an individual index means removing its many records from the merged index, and re-creating an individual index means inserting its many records into the merged index.

There is, however, a third alternative, based on partitioned B-trees [G 03]. Note that traditional partitioning is a metadata operation, such that changes in the partitioning scheme force expensive recompilation of cached query execution plans. Partitioned B-trees define partitions within the B-tree using an artificial leading key column. Partitions appear and disappear by insertion and deletion of records or by updates of values in the artificial leading key column.

A bulk insertion exploiting partitioned B-trees proceeds in two steps. First, new records are inserted into a new partition, i.e., old and new data do not interleave. Second, partitions are merged such that only a single partition remains.

After the first step, data is immediately searchable, as it is correctly present in the B-tree. The B-tree becomes optimized for search performance only during the second step. The algorithm logic employed is very similar to a merge step in an external merge sort. The second step is not a single large transaction, however; instead, many small system transactions ensure high availability of the index data by frequent transaction commits.

Since a merge step is practically equally efficient whether there are only two merge input or many, it is not required that the initial insertion in the first step inserts into only a single partition. Instead, it can use multiple partitions, e.g., using a partition per memory load such that the first step only generates initial sort runs but does not merge any runs. If so, bulk insertion can very efficiently insert new data into multiple indexes, using only in-memory sort for each appropriate index sort order.

The main concern about this procedure is that both steps require logging. If log bandwidth and space are available, the dilemma of choosing between inserting very slowly and wastefully dropping existing indexes is resolved. Very preliminary design work indicates, however, that non-logged merge operations are possible, based on “force” buffer pool management during transaction commit [HR 83] or “careful replacement” of pages in the buffer pool and on disk [GR 93, LT 95].

Bulk deletion proceeds in the opposite order. First, records to be deleted are moved to a separate partition, a few records at a time using a system transaction that can commit without flushing the log. All data remain searchable during this period, even if search performance is not optimal. Second, this entire partition is deleted in a single large and efficient user transaction.

The separation of insertion into a B-tree and optimization of the B-tree structure also applies to bulk insertion into merged indexes. For the B-tree structure and its search efficiency, it does not matter whether a partition contains B-tree entries from one or from multiple individual indexes.

5.4 Adding and removing an index

Adding a new individual index to an existing, populated merged index is very similar to a bulk insertion and can employ the same data movement. After the required catalog updates, the new data is inserted into one or more new partitions. Once the new data is in the B-tree, they are available for queries and updates. Optimization of the B-tree structure by merging the partitions such that complex objects are properly co-located is a second step that can proceed as system load permits using many small system transactions.

The desirable effect of this two-step procedure is that the actual data insertion does not create contention with other indexes and their data. Instead, the existing partitions serve all concurrent user transactions such that user transactions and data definition do

not affect one another. Only the small system transactions that merge data can interfere with user transactions, but the system transactions can commit and release locks at any time because they do not modify the B-tree's contents, only its representation.

Removing an index can use the opposite sequence, very similar to bulk deletion. First, all records belonging to the index are moved to one or more separate partitions; second, this partition is deleted using a large but efficient operation.

It is even possible to add multiple indexes to a merged index in a single operation, or to merge two merged indexes into one. Similarly, it is possible to split an existing merged index into two or to remove multiple individual indexes from a merged index in a single operation. In each case, the generality of the record format is pivotal to making these operations possible, and partitioned B-trees are pivotal to making them efficient.

5.5 Online index operations

An index operation is online if it permits concurrent insertions, deletions, updates in the indexed table or view. Online index creation can be implemented using two alternative approaches [MN 92]. First, the index is created without concern for concurrent updates, which are applied only in a final catch-up step based on a "side file," e.g., the recovery log filtered for updates of the relevant table. Some complexity arises if the index creation operation is part of a large transaction, possibly with multiple online index operations, because all final catch-up must occur at transaction commit or each index must be taken offline from its last catch-up operation until transaction commit.

Second, updates can be applied immediately by the concurrent transactions to the future index; the principal difficulty is dealing with deletions in the relevant table while the index is not yet complete and may not contain the index entries for the affected rows. In this approach, "anti-matter" index entries can be inserted into the future index such that the index build operation recognizes when to suppress index entries. If the new index is for a summary view with "group by" or "distinct" operations, anti-matter takes the form of negative counters rather than a single flag bit [GZ 04].

Merged indexes do not require special considerations for online index operations. Both the side-file and the no-side-file approach are possible, as is online creation of indexes on summary views. The individual indexes within a merged index are not much different from traditional indexes; the main difference is that their keys and information fields are mapped to a binary string in a particular way and that they are then stored not in dedicated B-tree but in a merged index.

5.6 Incremental index operations

Similarly, incremental index creation does not depend on each individual index being stored in a dedicated B-tree. In incremental index creation, a table of contents captures the key ranges in the source, e.g., a table's clustered index, that have already been scanned and appropriately reflected in a new index. This table of contents starts empty, leading to a notion of "instant index creation," and ends up indicating that the entire table is fully indexed. Frequent commit operations by the index builder permit high concurrency with user transactions, which are even permitted to employ the new index if useful while still incomplete [G 03].

In fact, a user transaction can even invoke system transactions that copy appropriate data from the source to the new index, in order to ensure that the entire user transaction can be satisfied by the new index [G 03a]. This variety of incremental index creation could lead to a new, totally demand-driven paradigm for online index tuning.

Merged indexes can participate in incremental index creation as well as in shrinking an index incrementally if desired. A table of contents is required to capture to current state of the partial index. The table of contents can apply to a single one among the individual indexes, to a group of them (e.g., orders and order details but not invoices and invoice details), or to all individual indexes within the merged index. The mechanisms for these operations are fairly straightforward, whereas appropriate tuning policies and software tools remain as research opportunities.

Related to incremental index operations is the concept of in-memory-only indexes, which act effectively as caches. Imagine a buffer manager that, rather than evicting an index page from the buffer pool, invokes appropriate mechanisms that remove the page from the index and ensure appropriate modifications in the index's table of contents. While complex to realize due to the multiple software layers involved, a mechanism of this type might be particularly useful when applied to merged indexes.

5.7 Summary of index maintenance

In summary, index maintenance for merged indexes is not very different from index maintenance for traditional indexes. However, bulk operations take on additional importance because they support not only traditional bulk operations but also changes in the definition of the merged index, i.e., adding and removing individual indexes. Partitioned B-trees may be an important technique, because they not only speed up these operations but also permit changing the set of indexes with minimal concurrency control contention.

6 Query processing

Merged indexes create unique opportunities for efficient query processing, because they store data both sorted and clustered. The strongest effect is, of course, on joins of individual indexes in the same merged index.

What is suffering, however, is scanning just one of the individual indexes. Thus, merged indexes may not work well with some data mining algorithms or with hash join. More generally, if a merged index contains more individual indexes than needed for a given query, scan and join performance suffer. However, there are some reasonably efficient techniques to obtain just the records of one individual index.

Merged indexes can be very useful for nested query execution plans, in particular for caching results of previous invocations. These, too, will be discussed below.

6.1 Relational joins

Merged indexes can improve the performance of both merge join and index nested loops join. In fact, it has been observed previously that merge join and index nested loops join are very similar algorithms in many ways [GLS 93]. Beneficial effects accrue both for disk I/O and for cache faults.

Merge join performance should improve for multiple reasons, including less random I/O and fewer required pages in the buffer pool.

The performance of index nested loops join should improve perhaps even more due to improved locality and the resulting buffer effects. For any one value in the join, e.g., one orderkey in the join of orders and orderdetails, the number of required I/O operations, buffer pool frames, etc. is cut in half. If the merged index combines or replaces more than two traditional indexes, the savings can be even higher.

The greatest difference between merge join and index nested loops join is that the former joins entire streams or indexes whereas the latter is most appropriate when data about a small set of individual objects is needed, e.g., a single invoice or a single customer. Thus, online transaction processing for line-of-business applications will likely experience the most immediate and the most dramatic benefits of merged indexes.

6.2 *Single-index scan*

Scanning an individual index within a merged index is rather like scanning a traditional multi-column B-tree with a predicate on the second column but none on the first column. The required techniques have been described previously as multi-dimensional access method [LJB 95].

Specifically, such a scan interleaves what could be understood as two scans. The first among those enumerates the distinct values of the leading column. If, for example, a traditional B-tree's search columns are named x , y , and z , the first scan answers the query "select distinct x from ... order by x ." The first probe into the B-tree simply follows the B-tree edge from the root to the left-most leaf, equivalent to "select min (x) from ...". Each subsequent probe attempts to find the next-largest value of x , equivalent to adding the clause "where $x > x_0$ " with the most recent result value for column x replacing x_0 . The number of B-tree probes is equal to the number of distinct values in column x plus 1 at the right edge of the B-tree.

The second among these logical scans combines the user's predicate on column y with the term "where $x = x_0$." Thus, the user's predicate is employed and the B-tree is probed once again for each distinct value of column x .

If the leading column x is an integer, an optimization uses "where $x \bullet x_0+1$ " rather than " $x > x_0$." In fact, this predicate can be combined directly with the user predicate on the next column, cutting the total number of B-tree probes to the number of distinct values of column x plus the number of gaps in the series of distinct value in column x .

If there is a user predicate only on the third key column, say column z , distinct pairs of the first two columns must be enumerated using basically the same technique. Alternatively, one can think of it as one scan enumerating distinct values of column x , one scan enumerating distinct values of column y for each value of x , and a third scan evaluating the predicate on column z . The I/O pattern and cost are equal for these two ways of designing and implementing the algorithm.

In a partitioned B-tree, this technique can be used to search all existing partitions. The artificial leading key column that serves as partition identifier takes on the role of column x in the example above, and the number of B-tree probes is multiplied by the number of partitions plus the number of gaps in the sequence of partition identifiers. If the leading user-defined index key is not restricted, the scan must enumerate pairs of values for the partition identifier and leading user-defined column.

When scanning an individual index within a merged index, these techniques apply both to the partition identifier in a partitioned B-tree and to all other columns preceding

the index identifier in the record format. For example, if a merged index contains indexes on the customer, orders, and order details tables, scanning the B-tree for just the index on the orders table needs to enumerate partition identifiers, customer identifiers, and order numbers. Thus, merged indexes prove their advantage in complex object assembly but not for purposes of data mining and statistics or in combination with hash aggregation and hash join, as mentioned in the introduction.

6.3 Caching in nested iteration

Temporary indexes are a powerful capability in database query processing. After all, materialized and indexed views are optional and thus often temporary. Similarly, in-memory hash tables in hash join and in hash aggregation are temporary indexes, albeit in thread-private memory. Sort operations, e.g., for merge join or stream aggregation, are very similar to B-tree index creation [G 03].

Temporary B-tree indexes and merged indexes are particularly interesting in nested iteration and correlated sub-queries. In those query execution plans, an inner plan has formal parameters, also known as correlation columns. The inner plan is invoked repeatedly with actual parameter values taken from the current outer row. If two outer rows provide the same correlation values, execution of the inner query execution plan can be avoided if the result of the first execution has been cached. Thus, an index is needed that maps correlation columns to result columns.

An inner query's result may contain any number of rows, including zero rows. An empty result is important to cache, too, because that information permits avoiding re-execution as much as a non-empty result. In fact, even if the actual result is not cached, its size and cost of re-computation can be quite useful, e.g., to order the outer rows in queries for only a few output rows are desired such as "exists" and "top" queries. The cost of re-computation may vary by correlation value for many reasons, e.g., if there is a predicate in the inner query using a less-than comparison with the correlation value.

In order to represent empty results in the cache, it is expedient to create a "control" index that maps outer correlation columns to information about the result, e.g., its cardinality and size, its cost of re-computation if that depends on specific correlation values, and its usage frequency and most recent usage. In other words, if the main temporary index for the result of the inner query is a cache of fetched data somewhat like a traditional buffer pool, this control index can take the role of the buffer descriptors used to manage the buffer pool contents. The disadvantage of two indexes is, of course, that both indexes may incur I/O during query processing.

Merged indexes can eliminate the overhead of using two separate indexes by ensuring that a lookup in the control index loads the query results into the buffer pool if they are already available. Thus, a merged index provides all the power of representing a query result and its metadata without the need for inappropriate record formats.

6.4 Query optimization

Query optimization considers both logical aspects (tables, rows, columns, cardinality estimation, equivalence transformations) and physical aspects (indexes, records, fields, cost calculation, algorithm choices). Merged indexes offer an additional choice in physical database design and therefore do not affect any of the issues on the logical level.

For cost calculation, merged indexes affect data volume in scans and buffer pool effectiveness in joins. The buffer pool effects are quite similar to those of nested iteration for sub-queries and of index nested loops join, with similar cost formulas.

6.5 Summary of query processing

To recap this section on query processing, merged indexes have their place in join processing, in particular in the context of complex object assembly. They also permit single-index scans but only with limited efficiency. Thus, merged indexes are not a cure-all for performance problems but they are a valuable option during physical database design. In fact, their efficiency during retrieval of complex objects ought to reshape discussions about de-normalization for database performance. Merged indexes also add new power to nested iteration, in particular caching results of inner queries for duplicate values in the outer correlation columns.

7 Summary and conclusions

In summary, merged indexes, an implementation technique at the physical level of indexes and records, achieves the performance advantages that often are pursued with de-normalization at the logical level of tables and rows. These performance advantages are achieved without exposing data and applications to the well-known dangers of non-normalized database schemas.

Many of the beneficial effects in the proposed design are based on four techniques. First, the core B-tree implementation manages only binary records, leaving it to the next layer in the software stack within a database management system to map individual indexes and their key columns to appropriate binary strings. Second, each record includes an index identifier as one of its columns such that records from multiple indexes can be interleaved within a single B-tree yet separated when needed. Third, domain tags precede the leading key columns up to and including the index identifier, which permits adding a new individual index to a merged index at any time with immediate and automatic clustering. Fourth, partitioned B-trees permit efficient bulk operations in any B-tree, which in the case of merged indexes includes not only traditional bulk operations but also changes in the set of individual indexes interleaved to form the merged index.

These techniques enable many benefits. Most importantly, clustering of master-detail records and even of complex objects is almost natural. This is true even in traditional relational databases, with tremendous improvements in join costs and buffer requirements. Moreover, locking of entire complex objects is quite readily possible by replacing locks on leaf pages with locks on pages or on key ranges in the leaves' parent pages.

In relational database management systems, clustering has been neglected to-date due to the perceived difficulties involved. The presented design may change both this perception and the reality. The advantages of merged indexes for performance in online transaction processing and in data warehousing are very compelling. Given that most applications are designed and implemented using object-oriented design, object-oriented programming languages, and object-oriented tools, clusters of related records in merged indexes can match the access behavior of the applications. Each disk read in a merged indexes and each page in the buffer pool can carry more useful information than is possible

with traditional, individual indexes. If the present paper encourages and contributes to the design and to implementations of merged indexes, it has served its purpose.

Acknowledgments

Wey Guy gave very helpful feedback on an earlier draft of this paper.

References

- [ABC 01] Sameet Agarwal, José A. Blakeley, Thomas Casey, Kalen Delaney, César A. Galindo-Legaria, Goetz Graefe, Michael Rys, Michael J. Zwillig: Microsoft SQL Server. Chapter 27, p. 969-1006 in: A. Silberschatz, H. F. Korth, S. Sudarshan: Database System Concepts, 4th Edition. McGraw-Hill 2001.
- [BU 77] Rudolf Bayer, Karl Unterauer: Prefix B-Trees. *ACM TODS* 2(1): 11-26 (1977).
- [F 94] Phillip M. Fernandez: Red Brick Warehouse: A Read-Mostly RDBMS for Open SMP Platforms. *SIGMOD* 1994: 492.
- [G 03] Goetz Graefe: Sorting and Indexing with Partitioned B-Trees. *CIDR* 2003.
- [G 03a] Goetz Graefe: Executing Nested Queries. *BTW Conf.* 2003: 58-77.
- [G 06] Goetz Graefe: B-tree indexes, interpolation search, and skew. *DaMoN* 2006: 5.
- [G 07] Goetz Graefe: Hierarchical Locking in B-tree Indexes. *BTW Conf.* 2007.
- [GG 97] Jim Gray, Goetz Graefe: The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb. *SIGMOD Record* 26(4): 63-68 (1997).
- [GL 92] Vibby Gottemukkala, Tobin J. Lehman: Locking and Latching in a Memory-Resident Database System. *VLDB* 1992: 533-544.
- [GL 01] Goetz Graefe, Per-Åke Larson: B-Tree Indexes and CPU Caches. *ICDE* 2001: 349-358.
- [GLP 75] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, Irving L. Traiger: Granularity of Locks in a Large Shared Data Base. *VLDB* 1975: 428-451.
- [GLS 93] Peter Gassner, Guy M. Lohman, K. Bernhard Schiefer, Yun Wang: Query Optimization in the IBM DB2 Family. *IEEE Data Eng. Bull.* 16(4): 4-18 (1993).
- [GR 93] Jim Gray, Andreas Reuter: *Transaction Processing: Concepts and Techniques*. Morgan-Kaufman, San Mateo, CA, 1993.
- [GZ 04] Goetz Graefe, Michael J. Zwillig: Transaction support for indexed views. *SIGMOD* 2004.
- [H 78] Theo Härder: Implementing a Generalized Access Path Structure for a Relational Database System. *ACM TODS* 3(3): 285-298 (1978).
- [HR 83] Theo Härder, Andreas Reuter: Principles of Transaction-Oriented Database Recovery. *ACM Comput. Surv.* 15(4): 287-317 (1983).
- [HR 96] Theo Härder, Joachim Reinert: Access Path Support for Referential Integrity in SQL2. *VLDB J.* 5(3): 196-214 (1996).
- [HS 04] Windsor W. Hsu, Alan Jay Smith: The performance impact of I/O optimizations and disk improvements. *IBM J. of Research and Development* 48(2): 255-289 (2004).
- [J 91] Ashok M. Joshi: Adaptive Locking Strategies in a Multi-node Data Sharing Environment. *VLDB* 1991: 181-191.
- [K 83] Henry F. Korth: Locking Primitives in a Database System *J. ACM* 30(1): 55-79 (1983).
- [L 93] David B. Lomet: Key Range Locking Strategies for Improved Concurrency. *VLDB* 1993: 655-664.

- [L 01] David B. Lomet: The Evolution of Effective B-tree: Page Organization and Techniques: A Personal Account. SIGMOD Record 30(3): 64-69 (2001).
- [LC 89] Tobin J. Lehman, Michael J. Carey: A Concurrency Control Algorithm for Memory-Resident Database Systems. FODO 1989: 490-504.
- [LJB 95] Harry Leslie, Rohit Jain, Dave Birdsall, Hedieh Yaghmai: Efficient Search of Multi-Dimensional B-Trees. VLDB 1995: 710-719.
- [LT 95] David B. Lomet, Mark R. Tuttle: Redo Recovery after System Crashes. VLDB 1995: 457-468.
- [M 90] C. Mohan: ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multi-action Transactions Operating on B-Tree Indexes. VLDB 1990: 392-405.
- [M 97] Paul R. McJones: The 1995 SQL Reunion: People, Project, and Politics, May 29, 1995. Digital System Research Center Report SRC1997-018: (1997).
- [ML 92] C. Mohan, Frank Levine: ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging. SIGMOD 1992: 371-380.
- [MN 92] C. Mohan, Inderpal Narang: Algorithms for Creating Indexes for Very Large Tables without Quiescing Updates. SIGMOD Conf. 1992: 361-370.
- [O 86] Patrick E. O'Neil: The Escrow Transactional Method. ACM TODS 11(4): 405-430 (1986).
- [PP 03] Meikel Pöss, Dmitry Potapov: Data Compression in Oracle. VLDB 2003: 937-947.
- [S 05] Short-stroking. Storage magazine. <http://storagemagazine.techtarget.com>, July 2005.
- [TSI 96] Odysseas G. Tsatalos, Marvin H. Solomon, Yannis E. Ioannidis: The GMAP: A Versatile Tool for Physical Data Independence VLDB J. 5(2): 101-118 (1996).
- [V 87] Patrick Valduriez: Join Indices. ACM TODS 12(2): 218-246 (1987).