Hierarchical locking in B-tree indexes

Goetz Graefe

HP Labs [1]
Goetz.Graefe@HP.com

Three designs of hierarchical locking suitable for B-tree indexes are explored in detail and their advantages and disadvantages compared.

Traditional hierarchies include index, leaf page, and key range or key value. Alternatively, locks on separator keys in interior B-tree pages can protect key ranges of different sizes. Finally, for keys consisting of multiple columns, key prefixes of different sizes permit a third form of hierarchical locking.

Each of these approaches requires appropriate implementation techniques. The techniques explored here include node splitting and merging, lock escalation and lock de-escalation, and online changes in the granularity of locking. Those techniques are the first designs permitting introduction and removal of levels in a lock hierarchy on demand and without disrupting transaction or query processing.

In addition, a simplification of traditional key range locking is introduced that applies principled hierarchical locking to keys in B-tree leaves. This new method of key range locking avoids counter-intuitive lock modes used in today's high-performance database systems. Nonetheless, it increases concurrency among operations on individual keys and records beyond that enabled by traditional lock modes.

1 Introduction

Since the 1970s, hierarchical locking has served its intended purpose: letting large transactions take large (and thus few) locks and letting many small transactions proceed concurrently by taking small locks. Multi-granularity locking was one of the great database inventions of that decade.

Hierarchical locking is widely used for database indexes. The standard lock hierarchy for B-tree indexes starts by locking the table or view, then locks the index or an index partition, and finally locks a page or individual key. This design is implemented in many database systems where it works reliably and efficiently.

1.1 Problem overview

With disk sizes approaching 1 TB and with very large databases and indexes, this hierarchy of locking begins to show a flaw. Specifically, the currently common step from locking an index to locking its individual pages or keys might prove too large. There may be millions of pages and billions of keys in an index. Thus, if a lock on an entire index is too large and too restrictive for other transactions, thousands and maybe millions of individual locks are required. If a transaction touches multiple indexes and if there are many concurrent transactions, the count of locks multiplies accordingly.

[1] Palo Alto, CA. Much of this research was performed while employed by Microsoft.

Each lock structure takes 64-128 bytes in most implementations. This includes a doubly linked list for the transaction and one for the lock manager's hash table, where each pointer nowadays requires 8 bytes. In addition, there is a list for transactions waiting for a lock conversion (e.g., shared to exclusive) and a list of transactions waiting to acquire an initial lock on the resource. Thus, millions of locks take a substantial amount of memory from the buffer pool, from sort operations for index creation and index maintenance, and from query processing.

Acquiring and releasing a single lock costs thousands of CPU cycles, not only for memory allocation but also for searching the lock manager's hash table for conflicting locks, and not only for instructions but also for cache faults and pipeline stalls in the CPU. Thus, millions of locks require billions of CPU cycles, which is why hierarchical locking and automatic lock escalation are very important for commercial workloads.

A single ill-advised lock escalation suffices, however, to reduce the multi-programming level to one. Consider, for example, a transaction that reads some fraction of a B-tree index and updates some selected entries. Depending on the fraction read, tens or hundreds of such transactions operate concurrently on a single B-tree index. If a single one of them exceeds its lock escalation threshold and indeed escalates to an index lock, no other transaction can update even a single B-tree entry, the multi-programming level collapses and server throughput drops accordingly.

1.2 Solution overview

Neither abandoning hierarchical locking nor partitioning large indexes into small partitions solves this problem satisfactorily. The solution advanced in this research redefines the hierarchy of resources locked. It introduces two new designs with additional lock granularities between an entire index and an individual page or key, and compares strengths and weaknesses of traditional locking hierarchies with the two new designs.

All three designs let large scans lock entire indexes such that no conflict can invalidate a scan when it is almost complete. They also let transaction processing lock individual records or keys. In addition, the two new designs let queries lock index fractions of various sizes. Thus, the original spirit of hierarchical locking is preserved and indeed strengthened by eliminating the dilemma between locking an entire index and acquiring thousands or millions of individual locks.

In contrast to prior lock hierarchies, the two new designs support online changes to the granularity of resources available for locking. Thus, they permit to start with traditional locks on indexes and keys only and to introduce or remove larger locking granules if, when, where, while, and as much as warranted by the workload.

In addition, a new design for key range locking is introduced. Compared to existing methods, it relies on hierarchical locking in a principled way. It is simpler than traditional designs for key range locking as it avoids counter-intuitive lock modes such as "RangeI-N" (discussed below). Nonetheless, it permits higher concurrency in B-tree leaves. It requires new lock modes, but eliminates as many, and the new lock modes are based on established sound theory. Due to these advantages, the new design for key range locking has value independent of the proposed techniques for locking large sections of B-trees.

The following sections first review related work and some preliminaries, then describe each of the three alternative designs for hierarchical locking in B-tree indexes and, and finally offer some conclusions from this effort.

2 Related work

Multiple prior efforts have investigated and refined multi-granularity locking as well as key range locking in B-tree indexes. The techniques proposed in subsequent sections do not require understanding of the related work reviewed in this section, and some readers may wish to skip ahead. The details given here are provided for those readers who wish to compare and contrast prior techniques with the present proposals.

2.1 Related work on multi-granularity locking

Multi-granularity locking has been a standard implementation technique for commercial database system for more than a quarter century [GLP 75]. Other hierarchical concurrency control methods [C 83] have not been widely adopted.

Multi-granularity locking is often restricted to hierarchies. The hierarchy levels usually include table, index, partition, page, record, key, or a subset of these. The choice of lock level can be based on explicit directives by the database administrator or by the query developer or it can be based on cardinality estimates derived during query optimization. The theory of multi-granularity locking is not restricted to these traditional hierarchies, however. A premise of this research is that alternative hierarchies deserve more attention than they have received in the past.

A radically different approach has also been advocated in the past. In predicate locking, locks contain query predicates, and conflict detection relies on intersection of predicates [EGL 76]. In precision locking [JBB 81], read locks contain predicates whereas write locks contain specific values. Conflict detection merely applies predicates to specific values. Precision locking may be seen as predicate locking without the cost and complexity of predicate intersection.

These techniques target the same problem as hierarchical locking, i.e., locking appropriate large data volumes with little overhead, but seem to have never been implemented and evaluated in commercial systems. The key-prefix locks in Tandem's NonStop SQL product [GR 93] might be closest to an implementation of precision locks. Note that Gray and Reuter explain key-range locking as locking a key prefix, not necessarily entire keys, and that "granular locks are actually predicate locks" [GR 93].

"In addition to the ability to lock a row or a table's partition, NonStop SQL/MP supports the notion of generic locks for key-sequenced tables. Generic locks typically affect multiple rows within a certain key range. The number of affected rows might be less than, equal to, or more than a single page. When creating a table, a database designer can specify a "lock length" parameter to be applied to the primary key. This parameter determines the table's finest level of lock granularity. Imagine an insurance policy table with a 10-character ID column as its primary key. If a value of "3" was set for the lock length parameter, the system would lock all rows whose first three bytes of the ID column matched the user-defined search argument in the query." [from SB 97; see also BS 95]

Gray [G 06] recalls a design in which locking a parent or grandparent node in S mode protects the entire range of all leaf nodes underneath. A lock on the root page locks the entire index. IS locks are required on the entire path from the root to a leaf. A leaf-to-leaf scan locks appropriate parent and ancestor nodes. Exclusive locks work similarly.

At first sight, these locks may seem equivalent to key range locks on separator keys. Both methods adapt gracefully to very large tables and indexes. However, there are important differences. For example, before a leaf page can be locked in S mode, its parent page needs to be locked in IS mode. Both locks are retained until transaction commit. The IS lock prohibits any X lock on that parent node and thus prevents any update there, e.g., splitting a different leaf page under the same parent or splitting the parent itself. While holding IS locks on the root page, user transactions prevent any kind of update of the root node, e.g., splitting a child node or load balancing among the root's child nodes.

In addition, System R had "a graph for physical page locks" [GPL 75] but no further public documentation seems to exist [G 06].

2.2 Related work on key range locking

In the final System R design, a lock on a key implicitly locks the gap to the next key value; locking a key and locking the gap to the next key is the same [BG 06]. The conceptual hierarchy applies to fields within a record but does not consider the gap between keys a lockable resource separate from a key.

The Aries family of techniques includes two methods for concurrency control for B-tree indexes, key value locking [M 90] and index management [ML 92]. The former is quite subtle as it relies not only on locks but also on latches and on instant locks. The latter establishes that a lock on a row in a table can cover the index entries for that row in all the indexes for the table.

Lomet's design for key range locking [L 93] builds on Aries [M 90, ML 92] but leaves multiple opportunities for further simplification and improvement. The present work addresses those pointed out here.

- Most importantly, multi-granularity locking is considered for general ranges and keys, but is described in detail only for keys in leaf pages.
- Lock manager invocations are minimized by identifying multiple granules with the same key value and by introducing additional lock modes, but the B-tree level is omitted in the lock identifier which implies that locking key ranges in interior B-tree nodes is not possible.
- Intention locks (e.g., IS, IX) on the half-open interval (K_{i-1}, K_i] permit absolute locks on the key value K_i, but no provision is made to lock the open interval (K_{i-1}, K_i) separately from the key value, and appropriate concurrency can be achieved only by using unconventional and counter-intuitive lock modes.
- Protecting a gap between two keys by locking the prior or next key is discussed and next-key locking is chosen, but it seems that prior-key locking is more suitable for B-trees with suffix truncation for separator keys [BU 77], i.e., half-open intervals of the form [K_i, K_{i+1}).
- Application of key range locking to heap files is mentioned, but not generalized to ranges larger than immediately successive record identifiers.
- Locking complex objects with sets of components by means of range locks is considered, but neither developed in detail nor related to B-tree indexes.
- Locking individual pairs of key and record identifier even if the representation stores each distinct key value only once is discussed, but locking such a key value with all its record identifiers is neither appreciated for its beneficial interaction with query predicates nor generalized to locking arbitrary key prefixes in a hierarchy.

- Finally, deletion using “ghost” records is mentioned, i.e., separating row deletion in a user transaction and key removal in a subsequent clean-up action, but instead of applying this idea symmetrically to insertions, “instant” insertion locks are employed.

Microsoft SQL Server is a sample commercial database system using key range locking [M 06]. Many aspects of the design and its implementation match Lomet’s description [L 93]. It locks the open interval between two keys together with the interval’s high boundary key. This is evident in the “RangeI-N” mode, which is an insertion lock on the open interval with no lock on the boundary.

Each key value identifies locks on both the key value and the gap to the neighboring key such that a single invocation of the lock manager may obtain a combined lock. These savings are enabled by introduction of additional lock modes. This technique is employed in multiple ways, including “schema stability” and “schema modification” locks on tables. Schema stability read-locks a table’s catalog information,[2] e.g., during query optimization. In order to save lock manager invocations during query execution, both S and X locks on a table imply a schema stability lock. Schema modification write-locks both the catalogs and the data, e.g., during schema changes. Schema, data, and their combination could be modeled as a hierarchy in the locking protocol. However, while locking the schema without locking the data is useful, e.g., during query optimization, it is not immediately obvious what it means to lock the data without protecting the schema against modifications by other transactions.[3]

In SQL Server’s implementation of key range locking, the set of combined lock modes is not equivalent to multi-granularity locking. This is evident in several missing lock combinations, e.g., an exclusive lock on the gap between keys with no lock on the key value, i.e., there is no “RangeX-N” mode. This mode would permit one transaction to delete a key within an interval while another transaction modifies (the non-key portions of) the record at the interval’s high boundary. Similarly, a query with an empty result cannot lock the absence of a key without locking at least one existing key value, i.e., there is no “RangeS-N” mode. Finally, due to the missing “RangeS-X” and “RangeS-N” modes, it is not possible that one query inspects a key range and then updates the boundary key while another query merely inspects the key range.

Despite these missing lock modes, SQL Server uses as many as 6 lock modes of the form “RangeP-Q”, where P can be S, X, or I, i.e., shared, exclusive, or insertion lock modes, and Q can be S, X, or N, i.e., shared, exclusive, or no lock. In addition, SQL Server employs update locks [GR 93, K 83] on key values but not on ranges. Nonetheless, update locks add another 3 lock modes of the form “RangeP-U”, where P again can be S, X, or I.

SQL Server relies on short-term locks to guard key ranges against insertion of phantom records. Ghost records and system transactions (see below) are used for deletion but not for insertion. SQL Server employs lock escalation from keys or pages to an entire

[2] In a way, schema stability protects an entire complex object with rows in multiple tables and records in multiple indexes. We plan on generalizing this idea to user-defined complex objects in future work.

[3] There are exceptions to this general statement. For example, one transaction’s query may continue while another transaction adds a new constraint to a table. We plan on investigating concurrency among schema operations and data operations in future work.

index, based on an initial threshold and an incremental threshold if earlier lock escalation attempts failed. SQL Server does not support lock de-escalation. Rollback to a savepoint does not reverse lock escalation or lock conversion.

IBM's multi-dimensional clustering [BPM 03, PBM 03] is a special implementation of partitioning that includes the ability to lock an entire "block" (partition) or individual records within a block, albeit without lock escalation from record to block. The only commercial database system that supports lock de-escalation seems to be Oracle's (formerly Digital's) RDB product [J 91], although only in the special case of "lock caching" within nodes of a shared-disk cluster.

3 Assumptions

Assumptions about the database environment are designed to be very traditional.

B-tree indexes map search keys to information. This information might represent a row in a table or a pointer to such a row, i.e., B-trees may be clustered or non-clustered indexes. Each B-tree entry fits on a page; for larger objects, B-tree variants with a byte count rather than a search key added to each child pointer [CDR 89, SL 88] may be an attractive storage format but are not explored further here.

B-tree pages range from 4 KB to 64 KB, although the discussion applies to pages of any size. The number of records per page may be in the tens, most likely is in the hundreds, and in extreme cases is in the thousands.

Prefix and suffix truncation [BU 77] might be applied to leaf entries and to separator keys, but hardly affect the discussion at hand. Other compression methods, e.g., order-preserving run-length encoding or dictionary compression [ALM 96], affect the representation of keys and records but do not change the substance of the discussion, neither the techniques nor the conclusions.

	Latches	**Locks**
Separate …	Threads	Transactions
Protect …	In-memory data structures	Database contents
Modes	Shared, exclusive	Shared, exclusive, update, intention, escrow, etc.
Duration	Critical section	Transaction
Deadlock …	Avoidance	Detection and resolution
… by …	Coding discipline, "lock leveling"	Lock manager, graph traversal, transaction abort, partial rollback, lock de-escalation

Table 1. Latches and locks.

The database implementation observes the separation of latches and locks summarized in Table 1. This separation lets a single thread serve multiple connections and transactions and it lets a single transaction use multiple threads in a parallel query or index operation. In-memory data structures include the in-memory images of database pages.

The database contents include only those characteristics observable with ordinary "select" queries. This excludes database pages, distribution of records within pages, existence and contents of non-leaf pages in B-trees, etc. Thus, non-leaf pages do not require locks and are protected by latches only. The remainder of this paper focuses on locks.

In contrast to user transactions, "system transactions" modify only the database representation but not its contents and therefore permit certain optimizations such as commit without forcing the transaction log. Logging and recovery rely on standard write-ahead logging [G 78, GR 93, MHL 92].

A common use of system transactions is to remove ghost records left behind after deletions by user transactions. Separation of logical deletion (turning a valid record into a ghost) and physical removal (reclaiming the record's space) serves three purposes, namely simplified rollback 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 ghost removal can capture record deletion and transaction commit in a single log record (perhaps even including transaction start), there is never any need to log undo information, i.e., the deleted record's non-key contents.

Lock escalation and de-escalation are required or very desirable due to unpredictable concurrency contention and due to inaccuracy of cardinality estimation during query optimization. Lock escalation, e.g., from locking individual pages to locking an entire index, reduces overhead for queries with unexpectedly large results. It saves both invocations of the lock manager and memory for managing the locks.

Lock de-escalation reduces contention by relaxing locks held by active transactions. It requires that each transaction retain in transaction-private memory the information required to obtain the appropriate fine-grain locks. For example, even if a transaction holds an S lock on an entire index, it must retain information about the leaf pages read as long as de-escalation to page locking might become desirable or required.

Initial large locks combined with on-demand lock de-escalation can improve performance, because detail locks can be acquired without fear of conflict and thus without search in the lock manager's hash table [GL 92]. Also note that management of such locks in transaction-private memory is very similar to management of locks in a shared lock manager's hash table. Therefore, propagation of locks from private memory to the shared lock manager is quite similar to bulk updates of indexes, which has been found to improve fault rates in CPU caches (and in the buffer pool for on-disk indexes) and thus performance and scalability.

For large index-order B-tree scans, interior B-tree nodes guide deep read-ahead. Therefore, accessing those nodes and their separator keys do not incur any extra I/O, and it is conceivable to lock those nodes or their separator keys if desired.

Some operations benefit from fence keys, i.e., when a node is split, copies of the separator key posted in the parent are retained in the two sibling nodes. Fence keys aid B-tree operations in multiple operations, including key range locking and defragmentation [G 04]. Prefix and suffix truncation minimize fence key sizes [BU 77].

All B-tree entries are unique. It is not required that the declared search keys are unique; however, the entries must have identifying information such that a row deletion leads to deletion of the correct B-tree entry. Standard solutions add the row pointer to the sort order of non-clustered indexes or a "uniquifier" number to keys in clustered indexes.

Finally, locks on the low boundary of a gap protect the gap, e.g., against insertion of phantoms [GLP 76]. In other words, the description below employs "prior-key locking" rather than "next-key locking."

4 Traditional locking hierarchies

Traditional locking in B-tree indexes is the basis with which the two new designs of hierarchical locking will be compared. Locking of units larger than individual B-trees, e.g., databases and tables, is ignored in this discussion, because locking those does not affect hierarchical locking within B-trees. Partitioning is also ignored; if a table or an index is partitioned, the focus is on the B-tree that represents a single partition of a single index. If such a B-tree cannot be locked because it can only be locked implicitly by locking a larger unit such as a table or an index with all its partitions, the other database contents covered by those locks is ignored and the lock is called an index lock nonetheless.

The standard techniques lock the index and then leaf pages, keys, or both. The choice among page and key locks can be built into the software or might be available to database administrators. In the latter case, a change requires a short quiescence of all query and update activity for the affected table or index.

4.1 Locks on keys and ranges

Both proposed methods for hierarchical locking in B-trees depend on key range locking. Therefore, it may be useful to first introduce a simplification for traditional key range locking that relies more directly on traditional multi-granularity locking [GLP 75, K 83], avoids irregular complexities such as instant locks and insert locks [L 93], and increases concurrency when applied to keys in B-tree leaves. While helpful for hierarchical locking in B-trees discussed in later sections, it also is a simplification and improvement for traditional key range locking applied solely to keys in leaves.

For maximal concurrency and for maximal simplicity, hierarchical lock modes on keys should be designed so that a key and a key range together can be intention-locked and such that the gap between two keys and an individual key value can each be locked separately. In order to optimize performance of lock acquisition and release, it makes sense to implement not two levels of lockable resources (comparable to the standard example with pages and files) but as a single resource with many possible lock modes [L 93]. This technique should be an implementation optimization only rather than lead to the introduction of new locking semantics such as range insert locks.

	S	**X**	**IS**	**IX**
S	Yes	No	Yes	No
X	No	No	No	No
IS	Yes	No	Yes	Yes
IX	No	No	Yes	Yes

Figure 1. Traditional lock compatibility matrix.

Figure 1 shows a traditional lock compatibility matrix. For this discussion, the intention locks apply only to the combination of key value and gap between keys. The absolute locks can apply to the key value, the gap between keys, or their combination.

Using the locks in this lock compatibility matrix, the key value, the gap between keys, and the combination of key value and gap are three separate resources that must be locked using different locks and thus separate invocations of the lock manager code. The optimized design exploits the fact that the key value serves as the identifier for each of the three resources in the lock manager's hash table, which permits employing only one resource and one invocation of the lock manager at the expense of additional lock modes.

In the optimized design, which is strictly equivalent to traditional locking for separate resources, an S or X lock covers the combination of key value and gap between keys. The new lock modes in Figure 2 lock a key and the gap between two keys separately.

In order to avoid any possible confusion with combination lock modes such as SIX, which really represents two locks on the same resource such as a file, these new lock modes should be thought of and pronounced with the words "key" and "gap" added. For example, SØ should be pronounced "key shared" and SX should be pronounced "key shared, gap exclusive." Intention locks on the combination of key and gap, though implied by these new lock modes in the obvious way, are not part of the name.

The new lock mode SØ locks the combination in IS mode, the key in S mode, and the gap not at all. A ØS lock on key K_i locks the open interval (K_i, K_{i+1}) yet keeps the key value K_i unlocked. SX locks the combination in IX mode, the key in S mode, and the gap in X mode. A lock held in this mode is most likely the result of a lock conversion rather than a single request; as is the XS mode. All other lock modes are defined similarly.

Figure 2 shows the lock compatibility matrix for the new lock modes. It is important to stress that these lock modes are merely a straightforward application of the traditional theory of hierarchical locking [GLP 75, K 83], and thus less complex and less prone to mistakes by developers during implementation and maintenance than traditional key range locking [L 93, M 90].

	S	**X**	**SØ**	**ØS**	**XØ**	**ØX**	**SX**	**XS**
S	Yes	No	Yes	Yes	No	No	No	No
X	No	No	No	No	No	No	No	No
SØ	Yes	No	Yes	Yes	No	Yes	Yes	No
ØS	Yes	No	Yes	Yes	Yes	No	No	Yes
XØ	No	No	No	Yes	No	Yes	No	No
ØX	No	No	Yes	No	Yes	No	No	No
SX	No	No	Yes	No	No	No	No	No
XS	No	No	No	Yes	No	No	No	No

Figure 2. Lock modes for key values and gaps.

These new lock modes are identified by a key value but they really represent traditional locks on the key value itself, the gap between key values, and the combination of key value and gap. IS and IX modes are not shown because they do not apply to key values in this scheme other than implied by the new lock modes. The derivation of the values in the compatibility matrix follows directly from the construction of the lock modes.

For example, SØ and ØX are compatible because both take intention locks on the combination of key value and gap, and one transaction locks the key value in S mode whereas the other transaction locks the gap in X mode. Similarly, XØ and ØX are compatible, as are SØ and SX. A specific use of the ØS lock is to ensure absence of a key without inhibiting updates on the locked key. An alternative design inserts a ghost key for the absent key and retains a lock only on the specific key value.

New lock modes based on other basic modes, e.g., update locks [GR 93, K 83] or escrow locks [GZ 04, O 86], can be added in a similar way. In fact, it seems promising to integrate automatic derivation of lock modes such as SØ from a basic set of operational lock modes (e.g., S, X) and the set of components identified by the same database item (e.g., key value, gap between keys, and their combination) in a way similar to Korth's lock derivation techniques [K 83]. Automatic derivation seems far more promising for the lock modes in Figure 2 than for the lock modes of traditional key-range locking.

The strict separation of locks on keys and gaps enables concurrency not supported by traditional key range locking. The earlier discussion of lock modes omitted in SQL Server [M 06] includes several examples of concurrent operations that are correct but are not enabled. The new design enables all of them. The essential reason is that the presented design follows the traditional theory for hierarchical locking and thus provides fully independent locks on keys and gaps. Nonetheless, SQL Server and the proposed design require the same number of new lock modes (6) and of lock manager invocations.

4.2 Key insertion using ghost records

Insertion of a new key and record can be implemented as a system transaction. This is analogous to key deletion implemented by turning a valid B-tree record into a ghost record with asynchronous ghost removal using a system transaction. The system transaction for key insertion leaves behind a ghost record that the user transaction can lock in key value mode and then turn into a valid record to effect the logical insertion [GZ 04].

The value of using a system transaction for key insertion is that it releases its lock on the gap between pre-existing keys when it commits. The user transaction holds a lock only on the new key value, not on the gap. Holding this XØ lock until it commits, the user transaction can modify the record from a ghost to a valid record. In the case of a rollback, the ghost record remains and can be removed like any ghost record, e.g., upon request by a future user transaction that requires more space in the B-tree leaf.[4]

In the meantime, another user transaction or a system transaction may lock the gap defined by the new key. Using a ØX lock on the new key (really locking the gap only), the other transaction may insert yet another key, for example. Thus, this design supports high insertion rates within a page and an index, without any need to introduce irregular complexities such as instant locks and insert locks [L 93] that do not fit the traditional theories of two-phase locking and of hierarchical locking [GLP 75, K 83].

Moreover, the system transaction locks only the gap into which it inserts the new key; it has no need to lock the pre-existing key value that identifies the gap. Thus, another transaction can concurrently read the record with the pre-existing key. It can update and even delete the record if deletion is implemented by turning the valid record into a ghost record. Neither concurrent readers nor concurrent updates are possible in locking schemes without locks on only the open interval between keys, e.g., traditional key range locking [L 93, M 90, M 06]. The system transaction inserting a new key into a gap is compatible with any of these actions on the pre-existing key as long as the key is not erased from the B-tree. Thus, interference among neighboring keys is minimized by use of ghost records and system transactions.

[4] In fact, since a rollback operation never needs to roll back itself, the ghost can be removed as part of rolling back the user transaction, if desired.

This system transaction can be very efficient. Forcing the transaction log on commit is not required [GZ 04], and a single log record can cover transaction start, key insertion, and transaction commit. In workloads that append many B-tree keys with predictable keys, e.g., order numbers in an order-entry application, a single system transaction can create multiple ghost records such that multiple user transactions each can find an appropriate ghost record that merely needs updating, without creating a new record or a new lockable resource. Note that this technique applies not only to unique indexes on order numbers but also to indexes with multiple records for each order number, e.g., an index on order details or a merged index holding both orders and order details [G 07].

4.3 Splitting and merging key ranges

Consider a user transaction that first ensures the absence of a key value and then attempts to insert a new record with that key. In the first step, the user transaction obtains a shared lock on a key range. The second step requires an exclusive lock on that range, albeit only briefly. If the second step employs a system transaction, the two transactions and their locks conflict with each other.

One possible solution is to ignore lock conflicts between a system transaction and the transaction that invoked it – this can lead to complex conditions if one or both of the transactions roll back. Another possible solution is to avoid the system transaction in this special case, letting the user transaction perform the entire insertion – this requires complex code to recognize the correct cases.

The preferred solution is to enable inserting a new ghost key into an existing key range even while it is locked. In order to ensure that existing transactions are not materially affected, their locks on the key range must be duplicated such that they cover the two ranges resulting from the split as well as the newly inserted key. In the specific example above, a new ghost record and its key are inserted and all locks on the original key range are duplicated for the newly created key range including the new key itself.

The newly created locks are inserted into the lock manager and into the appropriate transactions' data structures, with a few differences to the standard procedures. First, lock conflicts with concurrent transactions are not possible, so there is no need to search the lock manager's hash table for conflicting locks. Second, these locks cannot be released during a partial transaction rollback to a savepoint. Instead, these locks must be retained during rollback, unless the lock on the original range is also released. Third, if locks are recorded in the transaction log for re-acquisition during log analysis after a crash, these newly created locks appear at an appropriate point in the recovery log with respect to the locks they protect in the next-lower B-tree level. Thus, the recovery procedure must be augmented to ensure that the lock hierarchy is respected during log analysis and recovery.

While perhaps tempting, the reverse operation is not always correct. A ghost record must not be erased while it is locked, because the lock might indicate that deletion of the key is not yet committed. If, however, only key ranges are locked but the ghost record's key value is not, and if the locks on the ranges below and above the ghost key do not conflict, then the ghost record can be erased and sets of locks combined.

4.4 Summary

The presented design is, in a way, "radically old." The contribution is identifying appropriate existing techniques and combining them into a simple and sound design that permits more concurrency than traditional key range locking.

The proposed design applies the theory of multi-granularity and hierarchical locking to keys and gaps in B-tree leaves. By doing so, it permits concurrency not supported by prior designs, yet it eliminates multiple counter-intuitive lock modes. It then applies a second existing technique to reduce the count of lock manager invocations by introducing derived lock modes. The proposed design benefits from this technique as much as traditional designs for key range locking. Finally, introduction of new B-tree keys using system transactions and ghost records simplifies the implementation of highly concurrent insertions and mirrors a standard implementation technique for key deletion.

5 Locks on separator keys

Key range locking can be applied more broadly than it is today. Specifically, in addition to its use in B-tree leaves, it can be used in interior B-tree pages. A lock on a separator key represents the entire key range from one separator key to its next neighbor (at the same B-tree level). Thus, a key range lock at the level of the leaves' parents is similar in some ways with a traditional lock on an entire leaf page. Both intention locks and absolute locks can be employed at that level, again similar to intention locks and absolute locks on B-tree leaf pages in the traditional hierarchy.

Like key range locks on keys in B-tree leaves, key range locks in non-leaf B-tree nodes are special forms of predicate locks [EGL 76]. Arguments and proofs for the correctness of key range locking in non-leaf B-tree nodes are based on established techniques for predicate locks. For example, splitting and merging of nodes and of ranges must ensure that appropriate predicate locks are preserved and protected.

Locking key ranges at the parent level results in locking overhead and concurrency behavior similar to traditional locking of leaf pages. A key range lock at the grandparent level, however, covers a much larger key range. Depending on a node's fan-out, a range lock at the grandparent level may save tens or hundreds of locks at the parent level, just like a key range lock in a parent node covers tens or hundreds of individual keys. For even larger B-tree indexes and queries, locking can start even higher in the tree.

Each level adds some overhead for acquisition and release of intention locks, even for transactions that actually touch only a single key. Fortunately, the B-tree level at which locking starts can be tuned according to the data and the application at hand, from locking keys in the leaves only to locking a key range at every level from the root to the leaves. Mechanisms for changing this level dynamically and online are discussed below.

5.1 Techniques

In contrast to locking in B-tree leaves, it is less useful in interior B-tree nodes to separate locks on key values and locks on gaps between keys. Thus, only standard lock modes (Figure 1) apply to those keys, and cover the half-open range from the locked separator key (inclusively) to the next separator key (exclusively).

A “high fence key” in each B-tree node, i.e., a copy of the separator key in the node’s parent, may delimit the range protected by a lock on the highest key value in a node. Some B-tree implementations already retain such a fence key, e.g., B^{link}-trees [LS 97]. Similarly, a “low fence key” may readily provide the key value to lock in order to protect the key range of a node’s left-most child. Thus, each node should also retain a “low fence key.” Many systems already retain this key [GR 93] even if they do not employ fence keys for multiple purposes [G 04].

Locks on separator keys in parents are different from locks on keys in leaves, even if the key values are the same. Thus, their identifier in the lock manager includes the index identifier, the key value, and the node level within the B-tree. Node levels are commonly stored in each node’s page header, with level 0 assigned to leaves, if for no other reason than for use during physical consistency check of the tree structure. Note that this identification of locks does not refer to physical location, e.g., a page identifier, such that locks remain valid if a separator key migrates when a node splits or when two nodes merge.

5.1.1 Splitting and merging nodes

When a node splits, a new separator key is inserted in the node’s parent. The required steps with respect to existing locks are precisely those necessary to insert a new ghost key into a key range even while it is locked. This procedure permits page splits at all B-tree levels without interrupting the flow of transactions and ensures that no transaction’s locked range is modified due to a change in the physical B-tree structure.

When two nodes merge, the opposite procedure applies. As two key ranges merge in the parent node, transactions end up holding locks on the combined range, as if they had locked both original ranges equally. It is required to verify, therefore, that the combined set of locks does not include conflicts, e.g., an S lock on one original range and an IX lock on the other original range[5]. In such cases, the merge operation must be delayed.

Load balancing between two neighboring nodes can be accurate in its locking but fairly complex due to the shift in key ranges, or it can be made fairly simple. The accurate method determines the locks needed on the parents from the locks held on the child entries that migrate. The simple method models load balancing as a merge operation followed by a split. The actual implementation and manipulation of data structures is, of course, not tied to these steps. The likely result is that some transactions hold more locks than they need. However, given that load balancing operations are rare (if implemented at all) and that the alternative procedure is complex, the simple procedure is preferable.

5.1.2 Lock escalation and de-escalation

While mechanisms to split and merge key ranges are required, lock escalation and de-escalation are optional, although they add significant value to locks on separator keys.

By default, locking may start with key range locks in the leaves’ parents, which is most similar to traditional page locking. Alternatively, it may start at the grandparent

[5] In the instant of merging the two ranges, transactions do not conflict even if their lock modes do. Thus, it might seem safe to permit merge operations even if locks conflict. However, transactions might perform additional operations based on locks on the combined range, and conflicts among these additional operations would remain undetected.

level in order to cover larger key ranges and require fewer locks in large range scans. For the discussion here, assume that initially locking starts with intention locks on separator keys at the grandparent level (e.g., IS locks) and continues with absolute locks on separator keys at the parent level (e.g., S locks). In the B-tree sketched in Figure 3, this means intention locks (IS or IX) on key ranges in grandparent node a, absolute locks (S or X) on key ranges in parent node c, and no locks in leaf nodes e, f, and g.

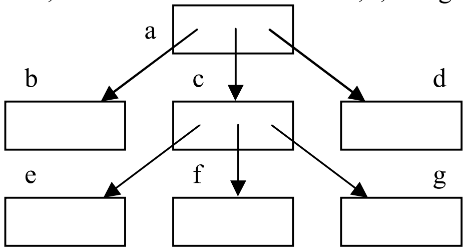

Figure 3. Lock escalation and de-escalation.

If so, either lock escalation (to absolute locks in the grandparents) or de-escalation (to intention locks in the parents and absolute locks in the leaves) may be required. Which one is required at what time depends on the workload, and in fact may differ within a single B-tree, e.g., an index on some time attribute in a data warehouse.

For lock escalation, the intention lock at the grandparent must become an absolute lock. In Figure 3, the only remaining locks will be absolute locks (S or X) on key ranges in grandparent node a. The locks at the parent level, now obsolete due to the absolute lock at the grandparent level, may be erased. Alternatively, they may be retained in the lock manager or in transaction-private memory for future lock de-escalation, which may be desired due to increasing contention or due to a partial transaction rollback.

For lock de-escalation, the absolute locks at the parent level must become intention locks. In Figure 3, intention locks will be used in grandparent node a as well as parent node c, and absolute locks will be used only in the leaves e, f, and g. This requires the opposite procedure and relies on proactive gathering of leaf-level key locks omitted during initial processing. After these leaf-level locks have been propagated from transaction-private memory to the global lock manager's hash table, the lock on the separator key in the parent can be relaxed to an intention lock.

Partial transaction rollback probably should not reverse lock de-escalation because reversal requires upgrading a lock (e.g., from IS to S in node c in Figure 3), which might fail due to locks held by concurrent transactions. In fact, after lock de-escalation due to contention by concurrent transactions, it is likely that such a lock upgrade during partial transaction rollback will fail.

5.1.3 Granularity changes

An entirely different kind of locking adjustment is a change in the lock hierarchy. In the case of locking separator keys in a B-tree index, this means changes in the set of B-tree nodes where key ranges must be locked. For example, the discussion above assumed locks on separator keys in the leaves' grandparents and parents but not the great-grandparents. However, this can be adjusted dynamically and online, i.e., without disruption of transaction processing.

Techniques for granularity changes may be useful beyond locking in B-trees. They apply whenever a specific granularity of locking is to be activated or deactivated online. Nonetheless, modifying the granularity of locking is a new notion not used or described elsewhere, as are techniques for doing so online.

Like lock escalation and de-escalation, these techniques are not mandatory for a functional implementation of locks on separator keys. However, these techniques may be required to minimize locking overhead for both online transaction processing and large range queries, in particular for workloads that change over time.

It seems most convenient to augment each interior B-tree node with a label to indicate whether or not keys in that node must be locked. The label applies to each node individually, not to entire B-tree levels. In fact, it even seems possible to label each individual separator key, an idea to be explored in the future.

In Figure 3, for example, assume that grandparent node a is not labeled and no locks are taken there, and that parent node c is labeled such that key range locks are taken on separator keys there.

Setting this label on a node, e.g., on grandparent node a in Figure 3, can be either immediate or delayed. The advantage of the immediate method is that a transaction may request lock escalation even to a node in which no transaction has acquired locks yet; the advantage of the delayed method is less search in the lock manager's hash table.

The immediate method searches in the lock manager's hash table for active transactions that ought to hold intention locks. These transactions can be found by searching for locks on keys in the node's children (e.g., nodes b, c, and d in Figure 3). For all such transactions, appropriate intention locks for the appropriate separator keys in the present node are inserted into the lock manager. While the search for such transactions is expensive, acquisition of the new intention locks is very fast as there is no need to search for conflicting locks (intention locks never conflict with other intention locks). The correctness of the immediate method relies on its correct acquisition of all intention locks that would have been held already if the label had been set on the node before any of the active transaction began.

The delayed method forces all new transactions to acquire intention locks and prevents acquisition of absolute locks in the node until all older transactions have completed. To do so, it employs two system transactions. The first one labels the node, thus forcing all future transactions to acquire intention locks while descending the B-tree. A second system transaction obtains IX locks on all keys in the node, observes the set of currently active transactions, waits until the last one of those has finished, and then commits. For the duration of these IX locks, no transaction may acquire an absolute lock. The correctness of the delayed method relies on a system transaction holding IX locks on all keys on behalf of all active transactions, whether those transactions require them or not.

When a non-leaf node is split, e.g., grandparent node a or parent node c in Figure 3, both resulting nodes inherit the label from the original node. A split operation can proceed even while a granularity change is on-going, whether the immediate or delayed method is employed. For two nodes to merge, they must be labeled equally first, i.e., a modification in the granularity of locking might be needed prior to a merge operation.

Removing a node's label, e.g., on parent node c in Figure 3, also employs a system transaction that acquires IX locks on all keys in the nodes, erases the node's label, and commits. If necessary, it waits for other transactions to release their absolute locks, if any, on those keys. Requesting lock de-escalation by those other transactions permits faster completion; once their absolute locks are relaxed to intention locks, there is no conflict with the desired IX locks.

Both setting and removing a node's label are very local operations that hardly affect data availability. Note that new user transactions can acquire intention locks within a node while a system transaction holds IX locks, because IX locks do not conflict with other intention locks, only with absolute locks.

The set of nodes labeled for locks on separator keys has few restrictions. For example, it is possible that a node is labeled but some of its siblings are not, or that a leaf's parent is not labeled but its grandparent is (skip-level locking). These examples seem counter-intuitive, but they might be appropriate incremental states while an entire B-tree is converted to a different granularity of locking, or they might be permanent states tuned for skew in the data distribution or in the access pattern. Nonetheless, skip-level locking might not be a good idea because it raises the complexity of lock escalation, de-escalation, and granularity changes.

While powerful, these mechanisms require governing policies. For transaction processing systems, an initial policy might start with locking keys in the leaves only and locking separator keys in interior B-tree nodes only inasmuch as necessary for lock escalation. In other words, non-leaf nodes are labeled as described above only on demand. Similarly, labels are removed after their benefit ceases, i.e., nodes remain labeled only during demand. For relational data warehousing, locks in parent and grandparent nodes seem like a reasonable default.

5.2 Advantages

The most salient advantage of locking separator keys in non-leaf B-tree nodes is that the method scales with the size and the height of the index tree. The stepping factor is fairly uniform across all levels of the index, even in an index with a non-uniform key value distribution. Typical values for the stepping factor may be 100 to 1,000; the latter value requiring large pages or very short keys, e.g., due to aggressive prefix and suffix truncation. Thus, the database administrator or the automatic tuning component can adjust the granularity of locking very accurately, much better than in the rigid traditional locking hierarchy of index, page, and key.

In addition, lock escalation and de-escalation permit graceful adjustments to precision and overhead of locking during query execution, even in cases of inaccurate cardinality estimation during query optimization. For unpredictable or time-variant workloads, the nodes at which key range locks are taken can be adjusted with very little disruption of online transaction processing. Thus, there is a real promise that the method can be made automatic, incremental, online, yet robust.

Even if multi-level hierarchies are not exploited and ranges are locked only in leaf nodes and their parent nodes, the method has some advantages over traditional locking of leaf pages and leaf keys. Specifically, there never is any doubt about which lock covers which other lock. Thus, there is no need for lock migration during leaf splits and merges, with the attendant code simplification and performance improvements.

5.3 Disadvantages

Despite the advantages over traditional locking for B-tree indexes, some queries and update still suffer with this strategy for hierarchical locking. While simple range queries will likely need only tens or maybe hundreds of locks, or orders of magnitude less than with traditional hierarchical locking, more complex yet common query clauses may still require a thousand locks or more. Consider, for example, an index on columns (a, b) and the query predicate "where a in (3, 9, 15, 21)". There are eight range boundaries to consider in this simple query, and each range boundary requires a choice whether to lock a larger range using few locks or to lock precisely using many fine-grain locks.

5.4 Opportunities

While the mechanisms described above are reasonably straightforward, their usage is an opportunity for further research, e.g., the number of locks required in average and worst cases. This analysis should cover not only selections but also joins, e.g., not only "between" queries but also "in" queries with lists of values or with nested queries. This evaluation ought to consider page splits near (rather than: at) a full node's center, as already recommended for effective prefix truncation [BU 77].

Research into policies and automatic self-tuning – starting level, performance metrics to observe, thresholds for lock escalation, lock de-escalation, and changes in the granularity of locking – could speed adoption of the proposed mechanisms in commercial systems. Nonetheless, if there is a global switch to force locking of keys or pages only and to prevent changes in the granularity of locking, the proposed mechanisms can always be restricted to a behavior very similar to current systems, thus providing a safe fall-back if a novel automatic policy fails.

Finally, one might design an alternative concurrency control method in which transactions lock large key ranges by locking interior B-tree pages, i.e., physical pages instead of logical key ranges. During searches and updates of existing pages, this approach is similar to locking separator keys. It is quite different, however, with respect to lock migration during page split and merge operations, during defragmentation, and during write-optimized operation [G 04].

5.5 Summary

The proposed design for hierarchical locking in indexes exploits the tree structure of B-trees. Traditional absolute locks and intention locks on separator keys in interior B-tree nodes protect all entries in their key ranges. Data skew is considered automatically, just as it is in the insertion, deletion, and search logic of B-trees with respect to concurrency control. Workload skew can be accommodated by lock escalation and lock de-escalation.

In order to minimize locking overhead during transaction processing, locking can be limited to keys in a B-tree's leaf level. Mixed operation, i.e., concurrent large queries and small updates in a real-time data warehouse, can be accommodated by adjusting the locking granularity dynamically. Modifying a database's lock hierarchy online is a novel technique that may seem particularly well suited to key range locking in B-tree indexes but is applicable to any form of hierarchical or multi-granularity locking.

6 Locks on key prefixes

The prior method exploited the tree structure for multi-level hierarchical locking; the second alternative for hierarchical locking in B-tree indexes ignores the tree structure and instead exploits the key structure for the same purpose. Consider, for example, a B-tree index for a multi-column compound key with columns (a, b, c, d) and with specific values (a_0, b_0, c_0, d_0). Any leading prefix such as (a_0, b_0) can serve as a resource that permits locking all index entries with keys starting with these specific values.

The promise of this method is that it matches precisely with query predicates, e.g., a query clause “where a = a_0 and b = b_0”. More complex predicates map to only a few very precise locks. For example, a query with the clause “where a in (a_0, a_1, a_2) and b in (b_0, b_1, b_2, b_3)” requires only 12 locks, independent of the sizes of the table and its index. Thus, locking key prefixes may be competitive with predicate locking and precision locking but without any need for predicate evaluation for concurrency control.

At the same time, this method promises efficient locking for single-row updates, because it is possible to lock a single B-tree entry (recall that all B-tree entries are unique in order to permit accurate deletion and maintenance). Thus, hierarchical locking based on key prefixes promises to match very well with both queries and updates in both transaction processing and data warehousing.

Locking key prefixes matches so well with query predicates because it is, in fact, a special form of predicate locking. Thus, arguments and proofs for the correctness of locking key prefixes rely on established techniques for predicate locks [EGL 76].

6.1 Techniques

A lock on a specific value, say the two-column prefix (a_0, b_0), covers all B-tree keys starting with these values. In order to prevent phantom records in serializable transaction isolation, however, non-existent key values also need locking.

In order to solve this problem, locks can cover only a specific prefix, the gap between two actual prefix values, or both. As in traditional key range locking in leaf pages, this can be modeled using an additional level in the locking hierarchy, e.g., those shown in Figure 1. For a smaller number of lock manager invocations, the actual implementation may use the artificial lock modes and their compatibility matrix shown in Figure 2.

Existence of specific key values in a B-tree can be decided only after a B-tree search has found the right index leaf. Thus, key locking starts only after the navigation from the B-tree root to the leaf. As discussed earlier, this navigation is protected by latches, not locks, including the inspection of the current leaf contents.

6.1.1 Insertion and deletion of values

While exact-match lookup in a B-tree, e.g., during an index nested loops join, benefits from precise locking and thus from key prefixes, insertion and deletion of keys might require key range locking. Specifically, during insertion, if a key prefix needs to be locked, the lock mode depends on the existence of a prior B-tree entry with the same prefix value as the new record. If such a prior B-tree entry exists, a key value lock suffices. If no such entry exists, a range lock on the prior key is needed. Note that this lock only needs to cover the open interval between the two pre-existing actual prefix values; there is no need to lock either of these values.

The initial insertion can be a system transaction that only inserts a ghost record, leaving it to the user transaction to turn that ghost record into a valid record. While the system transaction requires a range lock, the user transaction needs to lock only the new key value, not the gap between keys. Thus, the key range is locked only for a very short time.

While this description is very similar to the earlier discussion of key range locking of unique keys in B-tree leaves, it applies here to key ranges defined by key prefixes. For example, if a locked key prefix matches the definition of complex objects clustered in the B-tree by common key prefixes [G 07], a new complex object can be inserted without ever locking its neighboring objects or their components.

Deletion follows a similar pattern. A user transaction might simply mark a B-tree entry a ghost, leaving it to a system transaction to erase the record from the B-tree, or a user transaction might immediately erase the record and the key. Erasing a record requires a lock on the record's key in exclusive mode, thus ensuring that no other transaction holds any kind of lock on it. Moreover, the gap between two neighboring keys must remain locked in order to ensure successful transaction rollback if required. Gaps must be locked at all levels in the hierarchy for which a distinct value disappears.

6.1.2 Lock escalation and de-escalation

Whereas insertion and deletion of values are required, lock escalation and de-escalation are optional, as are granularity changes to be discussed shortly.

While key insertion and deletion are slightly subtle, lock escalation and de-escalation are particularly simple in this design, for two reasons. First, the granularity of locking and the currently active components of the lock hierarchy are uniform across the entire index. Second, structural modifications such as page splits of the B-tree do not affect the set of locks required. Both reasons, of course, are based on the complete separation of physical B-tree structure and locking hierarchy.

The sequence of locking granularities can be defined in very general ways. For example, it is not required that each column in the key defines a lock granularity. Skipping a column might be advisable if the column has only very few distinct values. A column with an extremely large number of distinct values could conceivably be modeled using multiple lock levels, e.g., the first three digits in a zip code or in a social security number can be modeled as a separate level in the locking hierarchy by treating them as a column in their own right, even if only for the purposes of locking.

Given these considerations, lock escalation and de-escalation follow the fairly obvious pattern. For lock escalation, an intention lock is upgraded to an absolute lock for the appropriate short key prefix, and the prior absolute locks can be forgotten or retained for possible subsequent lock de-escalation. For de-escalation, appropriate information must be retained during the transaction's prior activities such that appropriate locks on longer prefixes can be inserted into the lock manager's hash table, and the absolute lock on the short prefix can be downgraded to an intention lock.

6.1.3 Granularity changes

When locking separator keys in interior B-tree nodes, modifying the granularity of locking affects one node at a time. When locking key prefixes, the granularity of locking must be modified uniformly for the entire index. Thus, changes in the granularity of locking affect the entire B-tree, even for very large indexes.

With this exception, the procedure for changing the granularity of locking is quite similar for the two new strategies for hierarchical locking in B-tree indexes. When adding a new granularity of locking, both immediate and delayed methods can be designed.

In the delayed method, a system transaction locks all distinct key values at the new granularity of locking across an entire index. In contrast, hierarchical locking based on separator keys can modify the granularity of locking one B-tree node at a time.

Enumeration of all distinct values for a specific key prefix is a standard problem in relational query processing, e.g., for a query such as "select distinct a from …" based on an index on columns (a, b). While standard solutions exist under various names, e.g., as multi-dimensional B-tree access [LJB 95], the run-time expense seems impractical for adding a granularity of locking to a B-tree index.

The immediate method analyzes existing locks in order to acquire only those locks that are truly needed for specific active transactions. If hierarchical locking based on key prefixes is more precise than alternative locking methods and thus there are fewer locks to consider, this analysis can be faster than in hierarchical locking based on separator keys. An important difference, however, is that granularity changes apply to locking on separator keys one node at a time, whereas they apply to locking key prefixes for an entire index all at once.

Therefore, the delayed method does not seem promising for hierarchical locking based on key prefixes, and only the immediate method seems feasible, albeit not very practical. This is in contrast to hierarchical locking of separator keys, where both methods seem practical and appropriate in different circumstances.

Removal of an existing granularity of locking needs to wait for the release of all absolute locks at that granularity. A possible implementation technique is to wait for an IX lock competing with each such absolute lock held by a user transaction. Identifying all such locks requires search either in the lock manager's hash table or in the index.

To summarize, while it is possible to adjust the granularity of locking when locking key prefixes and when locking separator keys, the implementation mechanisms and tradeoffs are quite different. Whereas locking separator keys only on demand and only during demand appears to be a promising approach, a similarly promising approach is not apparent for locking key prefixes.

6.2 Advantages

Hierarchical locking of key prefixes has several very attractive characteristics. All of them reflect the match between query predicates and key prefixes. One overall advantage is that all of these characteristics are achieved with a single mechanism. This mechanism permits locking large ranges or individual B-tree records, and thus supports equally well decision support, transaction processing, and applications with similar access patterns.

First, this design matches well with equality predicates and "in" predicates, which are very common both in single-table selections and in joins. A single lock can often cover precisely the index entries needed, such that consistency and serializability among such a set of index entries are guaranteed, even in transaction isolation levels weaker than strict serializability. This aspect makes the locking strategy ideal for non-unique indexes including indexes on foreign key columns. If the leading key column is not specified and multiple index probes are required [LJB 95], a single lock per probe will suffice.

Second, if master-detail clustering is supported based on equality of search columns in a B-tree [G 07], locking key prefixes permits locking complex objects. For example, if orders and order details are co-located within a B-tree index based on equal values in the common column of order numbers, locking an order number locks an entire order object and all its component records. A single lock can also cover a large object, e.g., a customer with multiple orders, shipments, invoices, and payments. Insertion and deletion of one complex object does not require any lock on its neighbors.

Finally, locking key prefixes is a good match for partitioned B-trees [G 03b], i.e., B-trees with an artificial leading key column that indicates partitions in indexes and run numbers during sorting. The first granularity of locking is the partition identifier, which permits efficient operations on partitions such as merging partitions, creating a partition during data import, or dropping a partition while purging data from a data warehouse.

6.3 Disadvantages

This approach to hierarchical locking in B-tree indexes also has some disadvantages. Their common theme is the difficulty to set good defaults for the granularity of locking.

First, there is the danger of needing thousands of individual locks for thousands of distinct values, e.g., in index nested loops join or other forms of nested iteration. Thus, this locking strategy seems practical only if lock escalation is supported. Even then, a leading column with millions of unique values may be troublesome without dynamic changes in the hierarchy of locks.

Second (and amplifying the first point), the number of unique values may not be known due to old statistics or due to a missing range predicate on the leading index column. The number of unique values may be surprisingly large, leading to many individual locks. It may also be surprisingly small, leading to coarse locks that might restrict concurrency more than anticipated or desired.

Third, the effect of data skew are unclear, yet skew seems ubiquitous in real data.

Finally, there does not seem to be an opportunity to adjust the locking strategy locally to match any skew in the actual data values or in the workload. In order to achieve local adjustments in the granularity of locking similar to those described for key range locking on separator keys, an additional control table is needed that indicates which specific key prefixes should be locked with a granularity different than the default. Introduction of a control table for concurrency control seems like an unacceptable increase in the complexity of implementation, testing, maintenance, tuning, user education, and post-deployment vendor support.

6.4 Opportunities

Some of the disadvantages above could possibly be addressed by locking not only prefix columns but parts thereof. For example, if a column is a four-byte integer, each individual byte could be interpreted as a column of its own for the purpose of locking.

This idea is reminiscent of bit-sliced indexes [OQ 97] and, like those, can interact beneficially with range predicates.

This idea might alleviate the problem of key columns with a very large number of unique values, but it does not address data skew. Keys could be compressed, however, e.g., using order-preserving Huffman coding, in order to maximize the entropy of each stored bit. In that case, data skew might be less of a problem, although skew in the access pattern remains an unsolved issue.

Another opportunity based on disassembling a column is to apply hierarchical locking based on key prefixes not only to indexes but also to heap files. Specifically, components of a record identifier including file identifier, page identifier, slot number, and their individual bytes are treated as if they were individual columns in a compound B-tree index. Locks may cover predefined static ranges or dynamic ranges. In the former case, even ranges as yet unused require locks, i.e., files and pages ranges with no pages yet allocated for the relevant heap. In the latter case, the information about actual ranges covered by each lock would need to be inferred from the data structures used for space allocation. While very preliminary, this might be the first proposal for scalable hierarchical locking in heaps.

6.5 Summary

Hierarchical locking based on key prefixes is promising because locks match query predicates as well as complex objects in master-detail clustering. Insertion and deletion of unique values can benefit from the lock modes of Figure 2. Lock escalation and de-escalation can readily be supported. Adjusting the granularity of locking is not particularly complex if it is applied uniformly to an entire index, which unfortunately seems expensive and thus impractical. Nonetheless, if adjusting the granularity is not required and if lock escalation and de-escalation are sufficient dynamic mechanisms, locking key prefixes seems a viable alternative or complement to other locking strategies.

7 Summary and conclusions

Traditional hierarchical locking is starting to fail for very large indexes – the stepping from a single page to an entire index is too large. The dilemma between locking an entire index and locking millions of individual pages or keys demands a resolution.

The two alternative solutions introduced in this research have their distinct advantages and disadvantages when compared with the traditional design or with each other. Both new designs scale more gracefully than the traditional design due to their intermediate incremental stepping between index lock and page or key lock.

Key range locking in interior B-tree nodes scales with additional B-tree levels and adapts to skewed or unpredictable key distributions. The additional stepping levels between index and key eliminate the dilemma between massive overhead due to many individual locks and massive contention due to a single index lock. Moreover, splitting pages at keys that minimize the size of truncated separator keys [BU 77] matches key ranges in parent and grandparent nodes to clusters of records and complex objects.

Locking prefix values of search keys more obviously matches query predicates and complex objects in master-detail clustering, in many cases comparably with predicate locks and precision locks. However, it suffers in cases of very few or very many distinct values and in cases of data skew. Moreover, it does not permit efficient local adjustment of the granularity of locking. Thus, key range locking in parent and grandparent nodes is more practical as a general solution than locking prefix values of search keys.

In addition, this research is the first to consider dynamic changes in the lock hierarchy in response to skew in the data or in the workload, and to propose specific algorithms for doing so. In fact, these algorithms are online and incremental such that they minimize disruption to ongoing transaction processing.

These dynamic changes can be exploited in the many ways. For example, to maximize performance and scalability of transaction processing, only keys in leaf nodes are locked by default, with overhead, performance, and scalability comparable to traditional key value locking. If an actual transaction would benefit from a larger granularity of locking, e.g., locking the key range of an entire leaf page, key range locking can be introduced specifically in those parents and grandparents that benefit an actual transaction. When no longer advantageous, that granularity of locking can be abandoned to maximize transaction processing performance again.

For some applications and their indexes, the ideal locking strategy might be a combination of these techniques. Multi-granularity locking permits each element to participate in multiple hierarchies. For example, some readers may lock only keys and key ranges based on leaf records and separator keys, while other readers lock key prefixes. This is permissible if writers acquire appropriate locks in all hierarchies. Such locking strategies might be particularly useful in real-time data warehousing with concurrency among large queries and small incremental "trickle" updates, as well as in partitioned B-trees and their use cases [G 03b]. Even other hierarchies not considered in this research might be included, e.g., key ranges based on key ranges with fixed boundary values [L 93] or even multi-dimensional ranges in temporal or spatial data and applications.

Finally, this research simplifies and improves key range locking. The presented proposal simply applies the traditional sound theory for hierarchical locking to keys, gaps between keys, and the combination of key and gap. It avoids unconventional lock modes that violate traditional two-phase locking and traditional hierarchical locking. The new lock modes in Figure 2 are performance techniques with no new semantics. They ensure the same number of lock manager invocations as used in traditional key range locking. For high concurrency during both key insertions and removals, the proposed design employs system transactions. These transactions can use ordinary lock modes and lock retention to transaction commit. In combination, these techniques simplify key range locking without introducing new overhead. Equally important, they correctly admit more concurrency than traditional key range locking and may thus be useful in all database management systems.

Acknowledgments

Jim Gray has been very helpful and encouraging. Phil Bernstein, Barb Peters, and the reviewers contributed helpful suggestions that improved the presentation of the material.

References

[ALM 96] Gennady Antoshenkov, David B. Lomet, James Murray: Order Preserving Compression. ICDE 1996: 655-663.

[BG 06] Michael Blasgen, Jim Gray: personal communication. March 2006.

[BPM 03] Bishwaranjan Bhattacharjee, Sriram Padmanabhan, Timothy Malkemus, Tony Lai, Leslie Cranston, Matthew Huras: Efficient Query Processing for Multi-Dimensionally Clustered Tables in DB2. VLDB 2003: 963-974.

[BS 95] Charles J. Bontempo, C. M. Saracco. Database Management: Principles and Products. Prentice Hall (1995).

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

[C 83] Michael J. Carey: Granularity Hierarchies in Concurrency Control. PODS 1983: 156-165.

[CDR 89] Michael J. Carey, David J. DeWitt, Joel E. Richardson, Eugene J. Shekita: Storage Management in EXODUS. Object-Oriented Concepts, Databases, and Applications: 341-369.

[EGL 76] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, Irving L. Traiger: The Notions of Consistency and Predicate Locks in a Database System. Comm. ACM 19(11): 624-633 (1976).

[G 78] Jim Gray: Notes on Data Base Operating Systems. Advanced Course: Operating Systems 1978: 393-481.

[G 03] Goetz Graefe: Executing Nested Queries. BTW Conf. 2003: 58-77.

[G 03b] Goetz Graefe: Sorting and Indexing with Partitioned B-Trees. CIDR 2003.

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

[G 06] Jim Gray: personal communication. March 2006.

[G 07] Goetz Graefe: Algorithms for merged indexes. BTW Conf. 2007.

[GL 92] Vibby Gottemukkala, Tobin J. Lehman: Locking and Latching in a Memory-Resident Database System. VLDB 1992: 533-544.

[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.

[GLP 76] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, Irving L. Traiger: Granularity of Locks and Degrees of Consistency in a Shared Data Base. IFIP Working Conference on Modelling in Data Base Management Systems 1976: 365-394.

[GR 93] Jim Gray, Andreas Reuter: Transaction processing: concepts and techniques. Morgan-Kaufman 1993.

[GZ 04] Goetz Graefe, Michael J. Zwilling: Transaction Support for Indexed Views. SIGMOD 2004: 323-334.

[J 91] Ashok M. Joshi: Adaptive Locking Strategies in a Multi-node Data Sharing Environment. VLDB 1991: 181-191.

[JBB 81] J. R. Jordan, J. Banerjee, R. B. Batman: Precision Locks. SIGMOD 1981: 143-147.

[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.

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

[LS 97] David B. Lomet, Betty Salzberg: Concurrency and Recovery for Index Trees. VLDB J. 6(3): 224-240 (1997).

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

[M 06] Microsoft MSDN on SQL Server key-range locking, http://msdn2.microsoft.com/en-us/library/ms191272.aspx, retrieved 9/16/2006.

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

[ML 92] C. Mohan, Frank Levine: ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging. SIGMOD 1992: 371-380.

[O 86] Patrick E. O'Neil: The Escrow Transactional Method. ACM TODS 11(4): 405-430 (1986).

[OQ 97] Patrick E. O'Neil, D. Quass: Improved Query Performance with Variant Indexes. SIGMOD 1997: 38-49.

[PBM 03] Sriram Padmanabhan, Bishwaranjan Bhattacharjee, Timothy Malkemus, Leslie Cranston, Matthew Huras: Multi-Dimensional Clustering: A New Data Layout Scheme in DB2. SIGMOD 2003: 637-641.

[SB 97] C. M. Saracco and Charles J. Bontempo. Getting a Lock on Integrity and Concurrency. Database Programming & Design 1997.

[SL 88] J. Srivastava and V. Lum: A Tree Based Access Method (tbsam) for Fast Processing of Aggregate Queries. ICDE 1988: 504-510.