Avoiding index-navigation deadlocks

Lock acquisition sequences in queries and updates

Goetz Graefe

Abstract

Query execution plans in relational databases often search a secondary index and then fetch data from a table's primary storage structure. In contrast, database updates usually modify first a table's primary storage structure and then all affected secondary indexes. Deadlocks are likely if queries acquire locks first in the secondary index and then in the primary storage structure while updates acquire locks first in the primary storage structure and then in secondary indexes. In fact, some test teams use this scenario as a reliable means to create deadlocks in order to test deadlock resolution code. Some development teams have known of the problem for over 20 years but so far have failed to solve it, instead recommending that customers and application developers accept a weaker transaction isolation level and thus weaker correctness guarantees. A new technique uses the traditional, opposing navigation directions in queries and updates but the same locking sequence in both queries and updates. This design retains the efficiency of both queries and updates but avoids deadlocks caused by the traditional, opposing locking sequences.

1 Introduction

Even if only few deadlocks occur in most database applications, they lower system throughput and, perhaps more importantly, they reduce predictability of transaction durations and thus robustness of performance. If a traditional rule-of-thumb holds that only 1% of all transactions roll back, most of those due to deadlocks, this can still exceed 1,000 deadlocks and rollbacks per second in a high-performance server. Avoiding many or even most of these deadlocks and rollbacks seems a worthwhile goal if moderate effort and complexity can accomplish it.

Many deadlocks occur due to opposite navigation sequences in queries and updates. Queries usually search secondary indexes first and then fetch additional columns in a table's primary storage structure. Updates, on the other hand, usually modify a table's primary storage structure first and then adjust all affected secondary indexes. Updates here include insertions and deletions.

Updates retain their exclusive locks until end-of-transaction in order to enable transaction rollback without cascading aborts. Queries and searches retain their shared locks depending on the transaction isolation level. In "repeatable read" and "repeatable count" (serializable) transaction isolation, transactions retain their read locks until end-of-transaction. In "read committed" transaction isolation, scans release read locks when moving to the next record. This can have numerous surprising effects. For example, a query might retrieve a row identifier from a secondary index but fail to fetch the row because another transaction has deleted it in the meantime. For another example, a row may no longer satisfy the query predicate when fetched due to an update by another transaction.

These problems are particularly likely in sophisticated query execution plans. For example, intersecting multiple secondary indexes of the same table introduces a delay between index retrieval and final fetch operation, thus increasing the probability of an intervening update. One remedy runs each individual query execution plan in repeatable-read or serializable isolation, i.e., with shared locks held until end-of-statement, even if the overall transaction runs in weaker transaction isolation. Holding a shared lock until end-of-statement is sufficiently long, however, to participate in a deadlock.

If updates navigate from primary storage structure to secondary indexes and acquire locks in this sequence, and if queries navigate from secondary indexes to the primary storage structure and acquire locks in this opposite sequence, then deadlocks can occur whenever queries and updates access the same rows in a database table. In fact, this is one of the few cases when deadlocks occur predictably and, among all deadlocks, relatively frequently. Thus, it seems worthwhile to address this case with a specific and reliable solution. One product team's suggestion, which essentially recommends accepting a lower transaction isolation level and thus a weakened correctness criterion for concurrent transactions, seems like an escape from the problem rather than a solution.

A locking technique that avoids this problem altogether is ARIES/IM [ML 92]. While ARIES/KVL (key-value locking) [M 90] and key-range locking [L 93] acquire locks in each index or b-tree, ARIES/IM ("index management") locks logical rows of a table. A single lock covers a record in the table's primary storage structure, an entry in each secondary index, and (for phantom protection and thus serializability) the gap to the preceding index entry (in each index). ARIES/IM can have surprising effects on concurrency, however. For example, a search in an index on column A locks an entire logical row and thus a seemingly random key range (an index entry and an adjacent gap between index entries) in the index on column B and in the index on column C. This is true even if the search on column A actually produces an empty result. Perhaps less surprising than this behavior, these interactions have contributed to giving both locking and serializability a reputation for low concurrency.

What seems needed is a locking regimen with more precision. In orthogonal key-value locking [GK 15], a lock pertains to a distinct key value within a single index. A lock may cover a distinct key value (including all actual and possible instances), a hash partition of all such instances, a gap between actual distinct key values, a hash partition of possible values within such a gap, or any combination thereof. These lock granularities match accurately the requirements of updates including insertions and deletions via ghost records (pseudo-deleted records in ARIES) as well as the requirements of equality and range queries with and without results. Lock modes include shared and exclusive locks but may also include increment locks (for materialized "group by" views and their indexes) or intention locks (for hierarchical locking).

Even if orthogonal key-value locking solves the problem of excessive lock scopes, it does not solve the problem of opposite lock acquisition sequences in updates and queries and thus the problem of deadlocks. This is the issue addressed here using a simple but novel scheme for deadlock avoidance. The new solution is easy to understand and to implement but it has not been proposed in the past, despite being desirable for decades. The likely reason is that traditional locking methods and their published descriptions require locks not only on affected index entries or key values but also on their neighbors.

Traditional concurrency control locks not only index entries or keys affected by updates but also their neighboring index entries or keys. Locking neighbors forestalls the new solution. Thus, a solution to the problem has been desirable but missing for over 20 years. The crucial enablers of the new solution are ghost records used not only for deletions but also for insertions, as recommended for orthogonal key-value locking. If ghost records are used consistently with the traditional locking methods, then the new solution works in those contexts, too.

The new technique for deadlock avoidance specifically addresses the opposing navigation and lock acquisition sequences in queries and updates. It does not address all deadlocks in a database system and thus cannot replace deadlock detection or an approximation by timeouts on lock requests. Nonetheless, if it avoids a common source of deadlocks, it not only increases system performance but also the robustness of system performance. System performance can often be improved by more, newer, or better hardware, but robustness of performance requires software techniques such as those introduced here.

The remainder of this paper proceeds as follows. The next section reviews related work. The following section introduces the new lock acquisition sequence for insertions, updates, and deletions. The final section concludes with a summary and an outlook on future work.

2 Related prior work

For b-tree indexes in databases, we assume the standard design with user contents only in the leaf nodes. Therefore, all designs for key-value locking and key-range locking apply to key values in the leaf nodes only.

In this paper, the term "bookmark" means row identifier or record identifier in a table's primary storage structure. For example, if the table's primary data structure is a heap file, a bookmark may be a triple of device, page, and slot numbers. If the table's primary storage structure is an index such as b-tree, a bookmark may be a unique search key in the primary index. Such primary b-tree indexes are known as a primary index in Tandem's (eventually HP's) NonStop SQL, as a clustered index in Microsoft's SQL Server, and as an index-organized table in Oracle.

2.1 Ghost records and system transactions

By default, locks are possible on all key values in an index, including those marked as ghost records,

also known as invalid or pseudo-deleted records. Ghost records are usually marked by a bit in the record header. For non-unique secondary indexes with a list of bookmarks attached to each distinct key value, a ghost record is a distinct key value with an empty list or one in which all remaining entries are themselves ghosts.

The initial purpose and application of ghost records was guaranteed rollback of deletions. After a user transaction erases a record from a page, e.g., a b-tree leaf, another transaction may insert a new record; but when the deletion transaction attempts to roll back, it might force a leaf split or other allocation action that might fail. Failing a transaction rollback is, of course, an anathema to reliable transactional database management. The commonly adopted solution splits a deletion in two: the user transaction merely marks a record logically deleted such that rollback merely needs to mark it valid again, and if the user transaction commits, an asynchronous clean-up operation erases the record. For example, any subsequent insertion may invoke removal of any unlocked ghost record.

Record removal is a transaction in its own right, called a system transaction, also known as a top-level action in ARIES. System transactions are very inexpensive because they run in the same thread as the invoking user transaction, a single log record can describe the entire transaction, transaction commit does not require forcing log records to stable storage, and system transactions never acquire locks since they do not modify logical contents. System transactions may, however, inspect the lock manager, e.g., to avoid erasing a ghost record locked by a user transaction.

System transactions may affect physical database representation but not logical database contents. Thus, node splits and other structural modifications in b-trees can run as system transactions that commit ahead of the invoking user transaction. In fact, all space management tasks should be delegated to system transactions: allocation of a new record in preparation of a user transaction's logical insertion, growing a record in preparation of a user transaction's update of a variable-size column to a larger value, shrinking a record after a user transaction's update to a smaller value, free space compaction within a page, and all other clean-up tasks. For example, insertion transactions and reorganization utilities invoke system transactions for space reclamation.

2.2 ARIES/KVL "key-value locking"

ARIES/KVL [M 90] locks distinct key values, even in non-unique indexes. A lock in a secondary index covers all bookmarks associated with a key value as well as the gap (open interval) to the next lower distinct key value present in the index. A lock within a secondary index does not protect any data in another storage structure.

		Next key value	Current key value
Fetch & fetch next			S for commit duration
Insert	Unique index	IX for instant duration	IX for commit duration if next key value *not* previously locked in S, X, or SIX mode X for commit duration if next key value previously locked in S, X, or SIX mode
	Non-unique index	IX for instant duration if *apparently* insert key value *doesn't* already exist No lock if insert key value already exists	IX for commit duration if (1) next key not locked during this call OR (2) next key locked now but next key *not* previously locked in S, X, or SIX mode X for commit duration if next key locked now and it had already been locked in S, X, or SIX mode
Delete	Unique index	X for commit duration	X for instant duration
	Non-unique index	X for commit duration if *apparently* delete key value will no longer exist No lock if value will definitely continue to exist	X for instant duration if delete key value will *not* definitely exist after the delete X for commit duration if delete key value *may* or will still exist after the delete

Figure 1. Summary of locking in ARIES/KVL.

Figure 1, copied verbatim from [M 90], enumerates the cases and conditions required for a correct implementation of ARIES/KVL. At the same time, it illustrates the complexity of the scheme. Note that IX

locks are used for insertions into an existing list of bookmarks, which permits other insertions (also with IX locks) but neither queries nor deletions. In other words, ARIES/KVL is asymmetric as it supports concurrent insertions into a list of bookmarks but not concurrent deletions. Note also the use of locks with instant duration, in violation of traditional two-phase locking. This exemplifies how, from the beginning of record-level locking in b-tree indexes, there has been some creative use of lock modes that ignores the traditional theory of concurrency control but enables higher concurrency without actually permitting wrong database contents or wrong query results. Nonetheless, it substantially expands cost, complexity, and duration of quality assurance and thus of each software release.

Figure 1 gives guidance for insertions and deletions but not for updates. A value change in an index key must run as deletion and insertion, which Figure 1 cover, but an update of a non-key field in an index record may occur in place. Non-key updates were perhaps not considered at the time; in today's systems, non-key updates may apply to columns appended to each index record using, for example, a "create index" statement with an "include" clause, in order to "cover" more queries with "index-only retrieval." More importantly, toggling a record's "ghost" flag is a non-key update, i.e., logical deletion and reinsertion of an index entry.

Clearly, re-insertion by toggling a previously deleted key value requires more than an IX lock; otherwise, multiple transactions, at the same time and without noticing their conflict, may turn a ghost into a valid record. Thus, we conclude that the 1990 design of ARIES/KVL did not yet employ or exploit ghost records or system transactions (called pseudo-deleted records and top-level actions in later ARIES publications).

Due to locking separately in each index, ARIES/KVL suffers opposite locking sequences in queries and updates. Moreover, it prevents a pragmatic solution by requiring locks on the modified key's neighbor.

2.3 ARIES/IM "index management"

ARIES/IM [ML 92] locks logical rows in a table, represented by records in the table's primary storage structure, typically a heap file. With no locks in secondary indexes, its alternative name is "data-only locking." A single lock covers a record in a heap file and a corresponding entry in each secondary index, plus (in each index) the gap (open interval) to the next lower index entry. Compared to ARIES/KVL, this design reduces the number of locks in update transactions. For example, deleting a row in a table requires only a single lock, independent of the number of indexes for the table[1]. The same applies when updating a single row, with some special cases if the update modifies an index key, i.e., the update requires deletion and insertion of index entries with different key values.

	Next key	Current key
Fetch & fetch next		S for commit duration
Insert	X for instant duration	X for commit duration if index-specific locking is used
Delete	X for commit duration	X for instant duration if index-specific locking is used

Figure 2. Summary of locking in ARIES/IM.

Figure 2, copied verbatim from [ML 92], compares favorably in size and complexity with Figure 1, due to fewer cases, conditions, and locks. The rules for index-specific locking apply to the table's primary data structure. Insertion and deletion always require an instant-duration lock and a commit-duration lock on the current or next record.

ARIES/IM, by virtue of locking logical rows including all index entries, does not exhibit the problem of opposite locking sequences in queries and updates. It does, however, exhibit another severe problem in transactions with serializable isolation. When providing phantom protection by locking a gap between index entries in one secondary index, ARIES/IM locks the index entry at the high end of that gap by locking the logical row to which the index entry belongs. This lock freezes not only the gap but also the index entry

[1] This is true only if ghost records are employed. Figure 2 shows the original design of ARIES/IM without ghost records, i.e., immediate removal of each record and index entry, which requires a commit-duration exclusive lock on a neighboring index entry.

itself as well as an index entry in each further secondary index on the same table plus a gap in each index. For example, a query predicate on column A with no satisfying rows needs to lock a gap in the index on A but also locks gaps and index entries in indexes on column B, on column C, etc. This certainly is a surprising and counter-intuitive effect of a query with a predicate on A. Index-specific ARIES/IM eliminates this surprise but it suffers from more locks and lock manager invocations and from the problem of opposite locking sequences in queries and updates. Finally, both forms of ARIES/IM prevent a pragmatic solution by requiring locks not only on the current key but also on a neighboring key, as indicated in Figure 2.

2.4 Orthogonal key-value locking

Orthogonal key-value locking [GK 15] aims to remedy some of the shortcomings of ARIES key-value locking. While both techniques focus on existing distinct key values, there are significant differences between the designs.

First, the gap (open interval) between two distinct key values has a lock mode separate from (and entirely orthogonal to) the concurrency control for the key value and its set of instances. Thus, phantom protection does not need to lock any existing index entries. Instead, it merely requires that a locked key value continue to exist in the index. While one transaction uses a key value for phantom protection, another transaction may lock the key value itself and turn it into a ghost entry. Subsequent transactions may use the ghost for phantom protection and may turn the ghost into a valid index entry again.

Second, the set of all possible instances of a key value (e.g., the domain of bookmarks) is hash-partitioned and each partition can have its own lock mode. The concurrency desired in a system determines the recommended number of partitions. An equality query may lock all partitions at once but an insertion, update, or deletion may lock just one partition such that other insertions, updates, and deletions may concurrently modify other rows with the same key value but a different bookmark. More precisely, a concurrent transaction may update or delete a row with a different hash value and thus belonging to a different partition. Each individual bookmark has its own ghost bit such that two deletions may indeed proceed concurrently and commit (or roll back) independently.

Third, the set of all possible key values in a gap is hash-partitioned and each partition can have its own lock mode. An equality query with an empty result locks merely a single partition within a gap, thus achieving “repeatable count” transaction isolation (serializability) yet permitting other transactions to insert into the same gap. Range queries may lock all partitions within a gap. With this recent refinement not included in earlier descriptions [GK 15], orthogonal key-value locking can lock none, some, or all bookmarks associated with an existing key value plus none, some, or all non-existing key values in a gap between neighboring existing key values.

Index entries and gaps	entry (Gary, 1)	gap (Gary, >1)	gap (>Gary, <Jerry)	gap (Jerry, <3)	entry (Jerry, 3)	gap	entry (Jerry, 6)	gap (Jerry, >6)	gap (>Jerry, <Mary)	gap (Mary, <5)	entry (Mary, 5)
An entire key value				■	■	■	■	■			
A partition thereof					■						
						■					
An entire gap									■		
A partition thereof			▯	▯					▯ ■ ▯		
Maximal combination				■	■	■	■	■	■		
ARIES/KVL			■	■	■	■	■	■			

Figure 3. Lock scopes supported in orthogonal key-value locking.

Figure 3 summarizes the lock scopes that orthogonal key-value locking supports for key value Jerry in a non-unique secondary index sorted on first names and on bookmarks. Orthogonal key-value locking also supports any combination of lock scopes. For example, a serializable transaction with predicate “FirstName in (‘Harry’, ‘Harold’)” may lock two partitions within the same gap between Gary and Jerry. For another example, a range query can, with a single invocation of the lock manager, lock a distinct key value and its adjacent gap. In ARIES/KVL, this is the only available granularity of locking, as shown in the last line of Figure 3.

Further differences include system transactions inserting new key values as ghost records as well as system transactions creating and removing ghost space within records. System transactions perform all allocation and de-allocation operations in short critical sections with inexpensive transaction invocation

(no separate software thread), inexpensive logging (a single log record for the entire transaction), and inexpensive transaction commit (no log flush on commit). User transactions merely modify pre-allocated space, including the ghost space and the ghost bit within an index entry. This greatly simplifies logging and rollback of user transactions as well as free space management within each database page.

2.5 A brief comparison of locking techniques

The following diagrams compare ARIES/KVL, ARIES/IM, KRL, orthogonal key-range locking, and orthogonal key-value locking. Specifically, they compare the lock scopes for phantom protection, for an equality query with multiple result rows, and for a single-row update. Assuming insertion and deletion via ghost records, the lock scope of an insertion or a deletion equals that of a non-key update.

ARIES/KVL [M 90] locks an existing distinct key value in a secondary index, including all existing and non-existing instances of the key value plus the gap to the next-lower existing key value. ARIES/IM [ML 92] locks a logical row including all its index entries and, in each index, the gap to the next-lower index entry. Key-range locking in Microsoft SQL Server [L 93] locks one index entry in one index plus its preceding gap, with some separation between the lock mode for the index entry and lock mode for the gap. Orthogonal key-range locking permits two lock modes with each lock request, one for an index entry and one for the gap to the next-higher index entry. Lock scopes of orthogonal key-value locking are summarized in Figure 3.

Figure 4 illustrates required and actual lock scopes for the example of a query for a non-existing FirstName value, i.e., lock scopes for phantom protection. The column headings indicate ranges in the domain of the index keys. An S in Figure 4 indicates a transaction-duration shared lock to prevent insertion of index value Harry. It is clear that ARIES/KVL locks the largest scope. ARIES/IM appears equal to key-range locking only because Figure 4 does not show the lock scope in the other indexes of this table. Orthogonal key-range locking locks less due to separate lock modes on index entry and gap. Orthogonal key-value locking locks the smallest scope. Without partitioning within the gap, its lock scope matches precisely the locking requirement indicated in the header. Partitioning each gap between distinct key values is useful with this example query and further reduces the lock scope. By partitioning the non-existing key values within the gap between existing key values, the lock scope of phantom protection is as narrow as the FirstName Harry and its hash collisions.

Index entries and gaps	entry Gary, 1	gap (Gary, >1)	gap (>Gary, <Jerry)	gap (Jerry, <3)	entry Jerry, 3	gap	entry Jerry, 6	gap (Jerry, >6)	gap (>Jerry)
ARIES/KVL			S						
ARIES/IM		S							
KRL		S							
Orth. krl		S							
Orth. kvl			S						

Figure 4. Required and actual lock scopes in phantom protection for 'Harry'.

Figure 5 shows lock scopes for an equality query with multiple result rows. Key-value locking requires only a single lock whereas the other techniques require multiple locks (one more than matching instances). Due to separation of gap and key value, orthogonal key-value locking can lock the instance without locking any adjacent gap. Thus, it is the most precise technique (matching the query) and the most efficient technique (with a single lock manager call).

Index entries and gaps	entry Gary, 1	gap (Gary, >1)	gap (>Gary, <Jerry)	gap (Jerry, <3)	entry Jerry, 3	gap	entry Jerry, 6	gap (Jerry, >6)	gap (>Jerry, <Mary)	gap (Mary, <5)	entry Mary, 5
ARIES/KVL			S								
ARIES/IM		S_1			S_2			S_3			
KRL		S_1			S_2			S_3			
Orth. krl		S_1			S_2			S_3			
Orth. kvl				S							

Figure 5. Lock scopes in an equality query for 'Jerry'.

Figure 6 illustrates required and actual lock scopes for a non-key update of a single index entry. ARIES/KVL locks all instances of a distinct key value and ARIES/IM locks an entry and a gap in each index of the table. Key-range locking locks a single entry plus a gap in a single index. Orthogonal key-range locking leaves the gap unlocked and thus locks precisely as much as needed for this update operation. Orthogonal key-value locking locks a partition of index entries, ideally a partition containing only one entry.

Index entries and gaps	entry	gap			entry		entry	gap	
	Gary, 1	(Gary, >1)	(>Gary, <Jerry)	(Jerry, <3)	Jerry, 3	gap	Jerry, 6	(Jerry, >6)	(>Jerry)
ARIES/KVL			X						
ARIES/IM		X							
KRL		X							
Orth. krl					X				
Orth. kvl					X				

Figure 6. Lock scopes in a non-key update for row '3'.

Figure 6 also illustrates the locking patterns of user transactions inserting and deleting index entries via ghost records. Toggling the ghost bit in a record header is a prototypical non-key index update. Without ghost records, older techniques lock more, in particular an adjacent key value or index entry as shown in Figure 1 and Figure 2.

2.6 Summary of related prior work

In summary, there has been plenty of prior work on locking in database indexes. On the other hand, none of it has specifically addressed the problem of deadlocks among queries and updates due to index-to-index navigation in opposite directions. The only method that reliably prevents the problem of opposite lock acquisition sequences in queries and updates, ARIES/IM, has its own problems, namely counter-intuitive and excessive lock scopes in serializable transaction isolation. Both ARIES/IM and ARIES/KVL prevent a pragmatic solution by requiring locks not only on the current key but also on a neighboring key.

3 Recommended locking sequences

The way to avoid deadlocks when queries and updates acquire their locks in primary storage structures and in secondary indexes is to let them acquire their locks in the same sequence. Thus, either queries must acquire locks in a table's primary storage structures before locks in secondary indexes or updates must acquire locks in secondary indexes before locks in a table's primary storage structure.

3.1 New techniques

Queries access secondary indexes and primary storage structures in many ways, e.g., using index intersection or index joins. (An index join is a form of index-only retrieval: it combines two or more secondary indexes of the same table by joining them on their common bookmark field – if the columns in the indexes cover the query and its need for column values, scanning two indexes and their short records plus the join can be faster than scanning the primary storage structure with its large records [GBC 98].) Given the wide variety of possible query execution plans, it seems unreasonable to modify all of them to invert their sequence of lock acquisitions. Moreover, in the case of intersections or joins of multiple indexes on the same table, no single index scan can reliably determine the set of rows that will require locks in the table's primary storage structure.

Therefore, the proposed new lock acquisition sequence pertains to updates and to index maintenance plans. Before modifying a table's primary storage structure, in fact before lock acquisition within the table's primary storage structure, the new lock acquisition sequence acquires all required locks in all affected secondary indexes, with orthogonal key-value locking recommended. In this design, the update and maintenance operations that actually modify the secondary indexes do not acquire any further locks, e.g., key-range locks or key-value locks.

The sequence of database accesses and of database updates remains unchanged – only the lock acquisition sequence changes. Thus, buffer pool management, log record creation, etc. all remain unchanged. Even lock release remains unchanged: early lock release [DKO 84] or controlled lock violation [GLK 13]

remain possible and, in fact, recommended.

Key-range locking and key-value locking must not acquire locks on index keys that do not exist. Thus, in most implementations, a thread retains a latch on an index leaf page while requesting a lock on a key value. (Waiting and queuing require special logic for latch release.) In the new regimen, it is not possible to hold a latch on an index page during lock acquisition for index keys. Thus, it might appear at first as if the new technique could attempt to lock non-existing key values. Fortunately, this is not the case. It is sufficient to hold a latch on the appropriate data page in the table's primary storage structure during acquisition of locks in secondary indexes. This latch guarantees that key values in secondary indexes remain valid during lock acquisition.

Deletions present hardly a problem – the deletion logic while accessing the table's primary storage structure can readily obtain all column values required for the secondary indexes and request the correct locks. This includes a bookmark if it is part of a unique entry in a secondary index and thus part of a lock identifier.

The original descriptions of ARIES locking assumed immediate removal of index entries, which required locks on a neighboring key in addition to the key being deleted [M 90, ML 92] – see Figure 1 and Figure 2. The update of a table's primary storage structure cannot anticipate neighboring key values in all affected secondary indexes. Thus, their deletion logic does not permit lock acquisition while modifying the table's primary storage structure, i.e., before accessing the appropriate leaf page in all secondary indexes. Later ARIES work recommends logical deletion, i.e., turning a valid record into a pseudo-deleted record and relying on subsequent asynchronous clean-up, e.g., as part of a later insertion attempt. Toggling the ghost (pseudo-deleted) bit in an index entry requires a lock only on the index entry or its key value but not its neighbor. Thus, deletion via ghost status, now the standard way of implementing deletion in database systems, enables the change in the sequence of lock acquisitions.

Insertions present a different problem: the required key values might not exist yet in the affected secondary indexes. In general, locks on non-existing key values seem like a bad idea. For example, Tandem's solution to phantom protection inserts a key value into the lock manager but not the index leaf; anecdotal evidence suggests long and onerous testing and debugging. Here, however, the locked key value will soon exist in the secondary index, created by the insertion transaction acquiring the lock. As soon as a system transaction creates the required new ghost index entry, invoked by the insertion for the secondary index, the system reaches a standard state and the user transaction can turn the locked ghost into a valid index entry.

Updates of non-key index fields, i.e., those that do not affect the placement of the entry within the index, permit the new timing of lock acquisition in secondary indexes. There is no issue with non-existing key values. Updates of key fields require deletion and insertion operations and should be locked as described above.

3.2 Examples

The following examples illustrate some query and update operations.

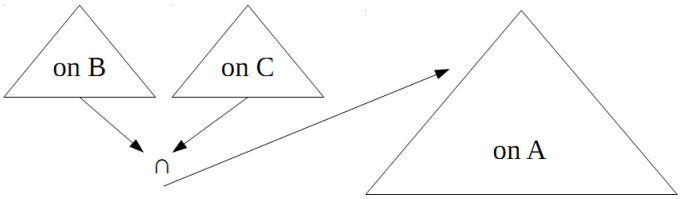

Figure 7. Query execution.

Figure 7 illustrates a query execution plan searching two secondary indexes of the same table, intersecting the obtained sets of bookmarks, and then fetching a few rows from the table's primary index. The primary index here is sorted or keyed on column A, the secondary indexes on columns B and C. Neither scan in a secondary index can reliably predict which records or key values in the primary index need locking. Thus, the recommended lock acquisition sequence is the same as the processing sequence: locks in each secondary index followed by locks in the primary index, but only on those index entries accessed by the query execution plan.

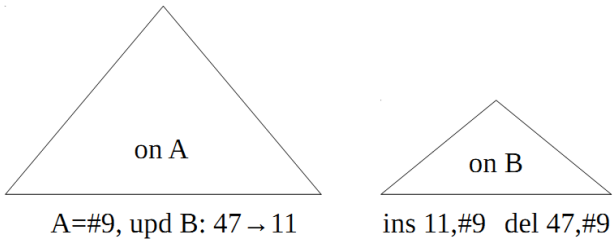

Figure 8. Ordinary update.

Figure 8 illustrates an update to an indexed column other than a bookmark column. In the row with primary key 9, column B is modified from 47 to 11. In the table's primary index, this is a non-key update. In the secondary index on column C, no changes and no locks are required. In the secondary index on column B, this update modifies a search key, which requires a deletion and an insertion. If the insertion merely turns an existing ghost record into a valid record, even if a system transaction just created that ghost record on behalf of the user transaction, then the user transaction can predict reliably and precisely all required locks. – The traditional lock acquisition sequence follows the update logic from the primary index to the secondary index. The recommended sequence first acquires a latch on the affected page in the primary index, then predicts all required locks in all indexes, acquires locks in the secondary indexes (i.e., for insertion and deletion in the index on B), only then acquires a lock in the primary index, applies updates in the primary index, releases the latch, and finally proceeds to the secondary index.

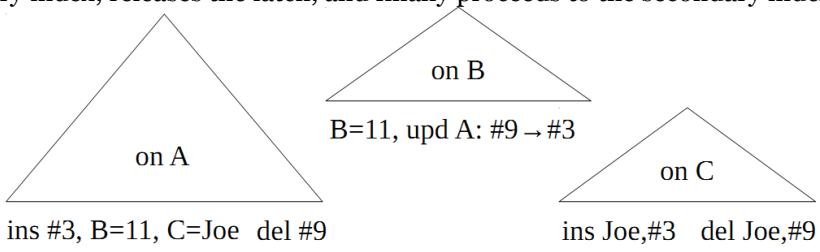

Figure 9. Bookmark update.

Figure 9 illustrates an update of a bookmark column A (from value #9 to value #3), which requires a deletion and an insertion in the table's primary index. A change in a row's bookmark requires an update in each secondary index. If a user-defined key for a secondary index is unique, a change in a bookmark is a non-key update in this secondary index. Otherwise, the bookmark is part of the sort key in the index and a change in a bookmark requires a deletion and an insertion in that secondary index. (In the example, there might be another index entry with C=Joe and A between #3 and #9.) Figure 9 shows both the unique index on column B and the non-unique index on column C. – In all cases, the user transaction can request all required locks before applying its deletions, updates, and insertions in the index storage structures. This is similar to the example of Figure 8 but with some differences. The principal difference is that there are two modification actions in the table's primary storage structure, one deletion and one insertion. Among several possible designs, one is to attach all lock acquisitions for deletions to the deletion in the primary index, and similarly for all insertions.

3.3 Effects

The principal effect of making updates acquire locks in the same sequence as queries is, of course, that it avoids deadlocks due to opposing lock acquisition sequences. The new technique cannot eliminate all deadlocks, only those due to opposing lock acquisition sequences in queries and updates. Thus, deadlock detection, perhaps by timeouts of lock requests, is still required. With fewer deadlocks, however, a transaction processing system will roll back fewer transactions and achieve more consistent transaction durations.

A negative effect is slightly earlier lock acquisition in secondary indexes and thus slightly longer lock retention. Note, however, that this additional lock retention time is always concurrent to locks on the record or key value in the table's primary storage structure. Thus, the actual reduction of concurrency should be minimal.

With respect to the number of locks and of lock manager invocations, the new method is neutral – it

neither adds nor removes any. The same is true for the size or scope of locks. For example, if a traditional update locks an index entry but no adjacent gap, then the new update logic does the same.

4 Conclusions

In summary, deadlocks may affect only 1% of database transactions but that can still mean 1,000s of deadlocks per second. A common cause of deadlocks is the fact that updates and queries navigate indexes in opposite directions and thus traditionally acquire their locks in opposite sequences. Locking logical rows (as in ARIES/IM) avoids this problem but forces large lock scopes and surprising concurrency limitations. In contrast, locking within each index, e.g., orthogonal key-value locking, permits lock scopes that accurately match successful and empty equality and range queries as well as updates including insertions and deletions via ghost records.

Opposite lock acquisition sequences in queries and updates have caused deadlocks for decades. A solution to the problem has been badly wanted for just as long. Recommending weak transaction isolation levels seems more of an escape than a solution. Traditional techniques that lock neighboring index entries or key values (as indicated in Figure 1 and Figure 2) cannot lead to a solution because those values are not known until the update execution plan processes the appropriate index leaf pages. The introductions of the two orthogonal locking methods recommend ghost records not only for deletions but also for insertions. With ghost records, a user transaction only locks the index entries or key values affected, never the neighbors, which finally enables a solution.

In the past, the danger of deadlocks between queries and updates has been an argument in favor of index-only retrieval, often enabled by adding non-key columns to secondary indexes. On the other hand, adding non-key columns to secondary indexes undermines database compression. In-memory databases and non-volatile memory render index-only retrieval less relevant with respect to access latency and query performance; the present work renders it less relevant with respect to concurrency control and deadlocks.

The newly introduced simple change in the update logic reliably avoids deadlocks due to opposite lock acquisition sequences. It works for deletions (with guaranteed index entries in all secondary indexes), insertions (with new key values in secondary indexes), and updates (non-key updates as well as updates of index keys). It does not modify the number of locks required in a query or an update; it only modifies the lock acquisition sequence in index maintenance plans. It also does not modify lock retention or enforcement times, e.g., in controlled lock violation. In other words, it promises to avoid deadlocks without any detrimental change in transaction processing efficiency.

Acknowledgements

Steve Lindell, Mike Zwilling, and Srikumar Rangarajan confirmed the described locking behavior in Microsoft SQL Server including the deadlocks addressed here. Caetano Sauer and Wey Guy helped clarify the problem and its solution. Thanh Do helped tighten the presentation and shorten the paper.

References

[CFL 82] Arvola Chan, Stephen Fox, Wen-Te K. Lin, Anil Nori, Daniel R. Ries: The implementation of an integrated concurrency control and recovery scheme. ACM SIGMOD conf 1982: 184-191.

[CM 86] Michael J. Carey, Waleed A. Muhanna: The performance of multi-version concurrency control algorithms. ACM TODS 4(4): 338-378 (1986).

[DKO 84] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, David A. Wood: Implementation techniques for main memory database systems. ACM SIGMOD conf 1984: 1-8.

[GBC 98] Goetz Graefe, Ross Bunker, Shaun Cooper: Hash joins and hash teams in Microsoft SQL Server. VLDB conf 1998: 86-97.

[GK 15] Goetz Graefe, Hideaki Kimura: Orthogonal key-value locking. BTW conf 2015: 237-256.

[GLK 13] Goetz Graefe, Mark Lillibridge, Harumi A. Kuno, Joseph Tucek, Alistair C. Veitch: Controlled lock violation. ACM SIGMOD conf 2013: 85-96.

[M 90] C. Mohan: ARIES/KVL: a key-value locking method for concurrency control of multiaction trans-

actions operating on b-tree indexes. VLDB conf 1990: 392-405.
[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 E. Levine: ARIES/IM: an efficient and high-concurrency index management method using write-ahead logging. ACM SIGMOD conf 1992: 371-380.
[L 93] David B. Lomet: Key range locking strategies for improved concurrency. VLDB conf 1993: 655-664.