Orthogonal key-value locking

Goetz Graefe, Hideaki Kimura
Hewlett-Packard Laboratories
goetz.graefe@hp.com, hideaki.kimura@hp.com

Abstract: B-trees have been ubiquitous for decades; and over the past 20 years, record-level locking has been ubiquitous in b-tree indexes. There are multiple designs, each a different tradeoff between (i) high concurrency and a fine granularity of locking for updates, (ii) efficient coarse locks for equality and range queries, (iii) run time efficiency with the fewest possible invocations of the lock manager, and (iv) conceptual simplicity for efficient development, maintenance, and testing.

A new design introduced here is efficient and simple yet supports both fine and coarse granularities of locking. A lock request may cover (i) a gap (open interval) between two (actual) key values, (ii) a key value with its entire list of (actual and possible) row identifiers (in a non-unique secondary index), (iii) a specific pair of key value and row identifier, or (iv) a distinct key value and a fraction of all (actual and possible) row identifiers. Using specific examples such as insertions, deletions, equality queries, and phantom protection, case studies compare four prior b-tree locking techniques and the new one. Experiments show that the new technique reduces the number of lock requests yet increases transactional concurrency, improving transaction throughput for both read-only and read-write transactions. The case studies and the experiments suggest that new b-tree implementations as well as existing ones ought to adopt the new locking techniques.

1 Introduction

Many data stores support indexes on multiple attributes of stored data items. In databases, for example, these are known as secondary indexes or as non-clustered indexes. Most database management systems permit dozens of secondary indexes for each table even if fewer secondary indexes per table are more common.

Key values in secondary indexes may or may not be unique. In other words, there may be many, even thousands, of rows with the same value in an indexed attribute. In those cases, compression is useful, e.g., bitmaps instead of lists of row identifiers. Such representation choices, however, are entirely independent from transactional concurrency control, where further improvements are possible despite multiple available techniques [G 10] and decades with little progress.

1.1 Concurrency control and lock scopes

In transactional concurrency control, pessimistic techniques (locking) may focus on rows in a table across all indexes (e.g., ARIES/IM [ML 92]) or on key values in an individual index (e.g., ARIES/KVL [M 90]). When locking entries within a non-unique secondary index, the finest granularity of locking may be individual index entries (representing one row and its value in the indexed attribute) or distinct key values (such that a single lock covers all instances of the same value in the indexed attribute).

Locks on individual index entries are more desirable for insert, update, and delete operations, because they permit that two transactions modify at the same time different rows with the same value in an indexed column. The disadvantage of this approach is that queries may need to acquire a lock on each index entry, possibly thousands of entries for a single key value. Many individual lock acquisitions not only impose overhead but also incur the danger of a lock acquisition failure (e.g., deadlock) late in the process. If that occurs, the entire statement or transaction aborts and all earlier effort is wasted.

Locks on distinct key values are more desirable for search and selection operations, because a query with an equality predicate needs to retrieve (and, for serializability, to lock) all items satisfying the predicate. The disadvantage of this approach is reduced concurrency in case multiple concurrent transactions need to modify separate index entries. A single lock may cover thousands of index entries when only one is needed.

The proposed design combines these benefits, i.e., high concurrency (by using a small granularity of locking) and low overhead (by using a large granularity of locking). It is a variant of multi-granularity locking (hierarchical locking) with some new aspects that lock multiple levels within the hierarchy by a single invocation of the lock manager yet cope with a large or even infinite domain of values in the lower level of the hierarchy.

1.2 Design goals

The obvious primary design goal is correctness, e.g., in two-phase locking. The second design goal is simplicity for easier understanding, development, maintenance, and quality assurance. Thus, the proposed design is simpler than all prior ones, e.g., rendering “instant duration” locks, “insertion” lock modes, and similar creative ideas obsolete.

The third design goal is high concurrency and therefore a fine granularity of locking, wanted in particular for updates and even more particularly for updates guided by searches in other indexes. Accordingly, the proposed design enables locks on individual index entries within lists associated with distinct key values.

The final design goal is run time efficiency and thus a coarse granularity of locking, wanted in particular for large operations such as range queries and equality queries in non-unique indexes, including the read-only search required to determine a set of rows to update or delete as well as index search in index nested loops joins. All transactions benefit from a minimal number of lock manager invocations as well as the fewest and earliest lock acquisition failures in cases of contention. Accordingly, the proposed design enables locks on distinct key values and their entire lists of index entries.

1.3 Preview

The focus here is on the scope of locks in b-tree indexes and on the effect of lock scopes on the number of locks required and on the degree of concurrency enabled. The following section reviews prior techniques for record-level locking in b-tree indexes, i.e., key-value locking and key-range locking. Section 3 introduces a new design and Section 4 compares the techniques using practical example scenarios. Section 5 reports on implementation and performance. Section 6 sums up the techniques and the results.

2 Prior designs

For b-tree indexes in databases, we assume the current standard design that has data records or index entries with user contents only in the leaf nodes, also known as B^+-trees. Therefore, all designs for key-value locking and key-range locking apply to key values in the leaf nodes only.

By default, locks are possible on all key values in the 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.

In this paper, the term "bookmark" means a physical record identifier (device, page, and slot number) if the table's primary data structure is a heap and a unique search key in the primary index if the table's primary data structure is a b-tree. Such primary b-tree indexes are known as a primary key index in Tandem's (now HP's) NonStop SQL, as a clustered index in Microsoft's SQL Server, and as an index-organized table in Oracle.

In most systems, an unsuccessful search in a serializable transaction locks a gap between neighboring existing key values, known as phantom protection. An alternative design inserts an additional key value (for precise phantom protection) either in the index or in some auxiliary data structure. For example, NonStop SQL includes such a key value within the lock manager (but not the data pages in the buffer pool). It seems that quality assurance and code stabilization for this design took substantial amounts of time. All methods discussed below lock the gap by locking an adjacent pre-existing key value.

2.1 ARIES/KVL "key-value locking"

ARIES/KVL [M 90] locks distinct key values, even in non-unique indexes. Each lock on a distinct key value in a secondary index covers all bookmarks associated with that 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 lock any data in the table's primary data structure or in any other secondary index.

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 the test matrix, i.e., cost, complexity, and duration of quality assurance.

Figure 1 provides guidance for insertions and deletions but not for updates. A value change in an index key must run as deletion and insertion, 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, 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 re-insertion of an index entry. Clearly, re-insertion by toggling a previously deleted key value requires more than an IX lock; otherwise, multiple transactions might attempt, at the same time and without noticing their conflict, turning a ghost into a valid record.

		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.

2.2 ARIES/IM "index management"

ARIES/IM [ML 92] locks logical rows in a table, represented by records in the table's primary data structure, a heap file. Thus, 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 key value. Compared to ARIES/KVL, this design reduces the number of locks. For example, updating a row in a table requires only a single lock, independent of the number of indexes for the table (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).

Figure 2, copied verbatim from [ML 92], compares in size and complexity rather favorably with Figure 1, because of much fewer cases and conditions. The conditions for index-specific locking apply to the table's primary data structure. In other words, insertion and deletion always require an instant-duration lock and a commit-duration lock on either the current or the next record.

These conditions apply to secondary indexes and their unique entries (made unique, if necessary, by including row identifiers) if ARIES/IM is applied to each individual index, i.e., if "data only locking" is abandoned. The inventors claim that "ARIES/IM can be easily modified to perform index-specific locking also for slightly more concurrency compared to data-only locking, but with extra locking costs" [ML 92] but no such implementation seems to exist. For the number of lock requests, it would combine the disadvantage of ARIES/KVL (separate locks for each index) with those of ARIES/IM (separate locks for each row in the table).

	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.

2.3 SQL Server key-range locking

Both ARIES/KVL and ARIES/IM reduce the number of lock manager invocations with locks covering multiple records in the database: a lock in ARIES/KVL covers an entire distinct key value and thus multiple index entries in a non-unique index; and a lock in ARIES/IM covers an entire logical row and thus multiple index entries in a table with multiple indexes. ARIES/IM with "index-specific locking" is mentioned in passing, where each lock covers a single index entry in a single index. The next design employs this granularity of locking and further introduces some distinction between a key value and the gap to its neighboring key value.

SQL Server implements Lomet's design for key-range locking [L 93]. Locks pertain to a single index, either a table's clustered index (primary index, index-organized table) or one of its non-clustered indexes (secondary indexes). Each lock covers one distinct key value (made unique, if necessary, by including the bookmark) plus the gap to the next lower key value (phantom protection by next-key locking). There is no provision for locking a distinct key value and all its instances with a single lock request. Instead, page-level locking may be specified instead of key-range locking for any clustered and non-clustered index, with additional complexity when b-tree nodes split and merge.

The set of lock modes follows [L 93]. S, U, and X locks are shared, update, and exclusive locks covering a gap and a distinct value. RS-S, RS-U, RX-S, RX-U, and RX-X locks distinguish the lock mode for the gap between keys (the mode prefixed by "R" for "Range") and for the key value itself. RI-N, RI-S, RI-U, and RI-X are "insertion" locks, all held for instant duration only and used for insertions into gaps between existing key values. The RI-_ lock modes are outside the traditional theory of concurrency control. The design lacks RS-N, RS-X, RX-N, and RU-_ modes.

SQL Server uses ghost records for deletion but not for insertion. It supports indexes on materialized views but not 'increment' locks, e.g., in "group by" views with sums and counts.

2.4 Orthogonal key-range locking

Orthogonal key-range locking is somewhat similar to key-range locking in SQL Server, but with a complete set of lock modes and completely orthogonal locks on key value and gap. Prior-key locking (rather than next-key locking) is recommended such that a lock on a key may include a lock on the gap to the next higher key value [G 07, G 10]. Lock modes (for key value or gap) may include shared, update, exclusive, and increment.

Figure 3 illustrates combined lock modes covering key value and gap derived from traditional lock modes. Concatenation of a lock mode for key values and a lock mode for gaps defines the set of possible lock modes. Additional lock modes are easily possible, e.g., update or increment locks.

Figure 4 shows the compatibility matrix for the lock modes of Figure 3. Two locks are compatible if the two leading parts are compatible and the two trailing parts are compatible. This rule just as easily applies to additional lock modes, e.g., update or increment locks. Some compatibilities may be surprising at first, because exclusive and shared locks show as compatible. For example, XN and NS (pronounced 'key exclusive, gap free' and 'key free, gap shared') are compatible, which means that one transaction may modify non-key attributes of a key value while another transaction freezes a gap. Note that a ghost bit in a record header is a non-key attribute; thus, one transaction may mark an index entry invalid (logically deleting the index entry) while another transaction requires phantom protection for the gap (open interval) between two key values.

Gap → ↓ Key	No lock: _N	Shared: _S	Exclusive: _X
No lock: N_	N	NS	NX
Shared: S_	SN	S	SX
Exclusive: X_	XN	XS	X

Figure 3. Construction of lock modes.

Requested → ↓ Held	S	X	SN	NS	XN	NX	SX	XS
S	ok		ok	ok				
X								
SN	ok		ok	ok		ok	ok	
NS	ok		ok	ok	ok			ok
XN				ok		ok		
NX			ok		ok			
SX			ok					
XS				ok				

Figure 4. Lock compatibility.

2.5 Summary of prior designs

All prior solutions imply hard choices for the finest granularity of locking in a database index: it may be a logical row (including all its index entries – e.g., ARIES/IM) or a distinct key value with all its index entries (e.g., ARIES/KVL) or an individual index entry (requiring many locks if a distinct key value has many occurrences and thus index entries – e.g., Microsoft SQL Server). Each prior solution suffers either from limited concurrency or from excessive overhead, i.e., too many lock manager invocations.

3 Orthogonal key-value locking

What seems needed is a design that permits covering a distinct key value and all its index entries with a single lock acquisition but also, at other times, permits high concurrency among updates and transactions. The proposed design combines elements of orthogonal key-range locking (complete separation of lock modes for key value and gap), of ARIES key-value locking (a single lock for a distinct key value and all its instances), and of ARIES/IM "index-specific locking" and key-range locking (locking individual index entries). Therefore, the new name is orthogonal key-value locking.

3.1 Design goals

As stated earlier in Section 1.2, the overall design goals are correctness, simplicity, concurrency, and efficiency. More specifically, the goal is to combine the advantages of key-value locking and those of orthogonal key-range locking:

- a single lock that covers a key value and all possible instances (e.g., the list of row identifiers);
- concurrent locks on individual instances (e.g., entries in a list of row identifiers); and
- independent locks for key values and the gaps between them.

Orthogonal key-value locking satisfies the first and last of these goals and comes very close to satisfying the remaining one.

3.2 Design

The proposed new design is most easily explained in the context of a non-unique secondary index. Although representation and concurrency control are orthogonal, it might help to imagine that the representation stores, with each distinct key value, a list of bookmarks pointing to individual records in the table's primary data structure.

The proposed solution divides a set of index entries for a specific key value into a fixed number of partitions, say k partitions. Methods for lock acquisition are modified to specify not just one but multiple lock modes in a single lock manager invocation. Some designs for key-range locking, specifically in Lomet's design implemented in Microsoft SQL Server and in orthogonal key-range locking provide a precedent: in those designs, a single lock identified by a key value in an index has two modes, one for the open interval between two existing key values and one for the record with the given key value. Using two lock modes, it is possible to lock the key value without locking the gap between key values and vice versa, with some restrictions in Lomet's design.

The proposed solution extends this idea to k+1 lock modes. One of the lock modes covers the gap between two distinct key values. The other k lock modes pertain to the k partitions in the list of index entries. A lock acquisition may request the mode "no lock" for any partition or for the gap to the next key value.

Figure 5 illustrates an index record within a non-unique secondary index on an imagined employee table. Each index record contains a key value, a count of instances, and the set of instances as a sorted list of bookmarks (here, primary key values). This list is partitioned into k=4 partitions, indicated by bold and italic font choices in Figure 5. In this example, the assignment from bookmark value to partition uses a simple "modulo 4" calculation. Index entries within the same partition (and for the same index key value) are always locked together; index entries in different partitions can be locked independently.

It makes little difference whether the lock acquisition method lists 2 lock modes (in existing designs for key-range locking) or k+1 lock modes (in the proposed design), and whether these lock modes are listed individually (e.g., in an array) or new lock modes are defined as combinations of primitive lock modes (as in Figure 3). For example, orthogonal key-range locking defines lock modes such as "XN" or "NS" (pronounced "key exclusive, gap free" and 'key free, gap shared'). They are formed from exclusive ("X"), shared ("S"), and no-lock ("N") modes on key and gap. Additional primitive lock modes, e.g., update and increment locks, can readily be integrated into orthogonal key-range locking as well as orthogonal key-value locking.

Specifically, each lock request lists k+1 modes. For only the gap between key values for an implementation with k=4 partitions per list, the requested mode might be NNNNS, i.e., no lock on any of the partitions plus a shared lock on the gap to the next key value.

A request may lock any subset of the partitions, typically either one partition or all partitions. For example, a query with an equality predicate on the non-unique index key locks all partitions with a single method invocation, i.e., all actual and possible row identifiers, by requesting a lock in mode SSSSN for k=4 partitions and no lock on the gap following the key value. An insertion or a deletion, on the other hand, locks only one partition for one key value in that index, e.g., NXNNN to lock partition 1 among k=4 partitions. In this way, multiple transactions may proceed concurrently with their insertions and deletions for the same key value, each with its own partition locked, without conflict (occasional hash conflicts are possible, however). An unusual case is a lock request for two partitions, e.g., while moving a row and thus modifying its bookmark.

Individual entries within a list are assigned to specific partitions using a hash function applied to the unique identifier of the index entry, excluding the key value. In a non-unique secondary index, the bookmarks (pointing to records in a primary data structure) serve as input into this hash function. Using locks on individual partitions, concurrent transactions may modify different entries at the same time, yet when a query requires the entire set of entries for a search key, it can lock it in a single lock manager call.

For k=1, the new design is very similar to the earlier design for orthogonal key-range locking. Recommended values are k=1 for unique indexes (but also see Section 3.4 below) and k=3 to k=100 for non-unique indexes. The optimal value of k probably depends on the desired degree of concurrency, e.g., the number of hardware and software threads.

Gender	Count	List of **EmpNo** values
'male'	89	*2*, **3**, 5, ***8***, *10*, ***12***, 13, *14*, **19**, 21, …

Figure 5. A partitioned list of bookmarks.

3.3 Lock manager implementation

Lock requests with k+1 lock modes impose a little extra complexity on the lock manager. Specifically, each request for a lock on a key value in a b-tree index must be tested in each of the k+1 components. On the other hand, in most implementations the costs of hash table lookup and of latching dominates the cost of manipulating locks.

Locks are identified (e.g., in the lock manager's hash table) by the index identifier and a distinct key value, e.g., Gender in Figure 5. This is equal to ARIES/KVL and similar to Microsoft's key-range locking and orthogonal key-range locking. Also equal is the rule that a lock request is granted if the request is compatible with all locks granted already.

The definition of lock compatibility follows the construction of Figure 4 from Figure 3: lock modes are compatible if they are compatible component by component. For example, multiple exclusive locks on the same key value but on different components or partitions are perfectly compatible. Finally, if the lock manager speeds up testing for compatibility by maintaining a "cumulative lock mode" summarizing all granted locks (for a specific index and key value), then this mode is computed and maintained component by component.

For small values of k, e.g., k=3 or even k=1 (e.g., for unique indexes), checking each component is efficient. For large values of k, e.g., k=31 or k=100, and in particular if equality queries frequently lock all partitions at once, it may be more efficient to reserve one component for the entire key value, i.e., all partitions. In a sense, this change reintroduces a lock on the distinct key value, with aspects of both ARIES/KVL (locking all actual and possible instances of a key value) and of orthogonal key-range locking (separate lock modes for key value and gap).

With the additional lock on an entire distinct key value, the number of components in the lock increases from k+1 to 1+k+1 (=k+2) lock modes for each key value in the index. This additional lock component on the entire key value permits absolute lock modes (e.g., S, X) as well as intention lock modes (e.g., IS, IX) and mixed modes (e.g., SIX). A lock request for an individual partition must include an intention lock on the entire key value. A lock request for all partitions at once needs to lock merely the entire key value in an absolute mode, with no locks on individual partitions. A mixed lock mode combines these two aspects: an SIX lock on a key value combines an S lock on all possible instances of that key value and the right to acquire X locks on individual partitions (of bookmarks associated with the key value).

Figure 6 illustrates both the lock modes on all partitions of a distinct key value and on gaps between such key values. Its left-most column lists the set of possible lock modes for the entire key value. The remainder of Figure 6 suggests lock modes formed by pairing the entire key value and the gap to the next key value. With this set of lock modes, the number of components in a lock request shrinks back from k+2 to k+1. More importantly, when a transaction needs to lock an entire key value, e.g., for a query with an equality predicate, lock acquisition and release are as fast as in traditional lock managers.

The top half of Figure 6 is equal to Figure 3; the bottom half adds intention locks for the entire key value. A lock compatibility matrix similar to Figure 4 is easily derived. In fact, derivation of these lock modes and their compatibilities is so straightforward that it is almost faster to write a program than to type them explicitly, which also simplifies adding further basic lock modes such as update and increment locks.

Another obvious optimization pertains to unique indexes locked with k=1, i.e., a single partition. In this case, it is sufficient to rely entirely on the top half of Figure 6, rendering the (one and only) per-partition lock moot.

Gap → ↓ Entire key value	No lock _N	Shared _S	Exclusive _X
No lock: N_	N	NS	NX
Shared: S_	SN	S	SX
Exclusive: X_	XN	XS	X
Intent to share: IS_	ISN	ISS	ISX
Intent to exclude: IX_	IXN	IXS	IXX
Shared with intent to exclude: SIX_	SIXN	SIXS	SIXX

Figure 6. Construction of lock modes.

3.4 Unique secondary indexes

For unique indexes, the obvious default is to use a single partition (k=1) to match the behavior of orthogonal key-range locking. There is, however, another alternative specifically for multi-column unique keys: lock some prefix of the key like the key in a non-unique index, and lock the remaining columns (starting with the suffix of the unique key) like the bookmarks in the design for non-unique indexes.

Perhaps an example serves best to clarify this alternative. Imagine two entity types and their tables, called "student" and "course," plus a many-to-many relationship with its own table, called "enrollment." An index on enrollment may be unique on the combination of student identifier and course number, which suggests orthogonal key-value locking with a single partition (k=1).

The alternative design locks index entries as in a non-unique index on the leading field, say student identifier. In that case, queries may lock the entire enrollment information of one student with a single lock (locking all partitions in the student's list of course numbers); yet updates may lock subsets of course numbers (by locking only a single partition within a list of course numbers). With an appropriate number of partitions, the concurrency among updates is practically the same as with locks on individual course numbers.

Note that this pattern of queries is quite typical for many-to-many relationships: a precise retrieval in one of the entity types (e.g., a specific student), an index that facilitates efficient navigation to the other entity type (e.g., on enrollment, with student identifier as leading key), and retrieval of all instances related to the initial instance. The opposite example would be similarly typical – listing all students for a single course using an index on course number of enrollment – and must be well supported in a database system, its indexes, repertoire of query execution plans, etc. B-tree indexes cluster relevant index entries due to their sort order; the proposed use of key-value locking supports these queries with the minimal number of locks and of lock requests.

3.5 Summary of orthogonal key-value locking

The proposed solution combines all the advantages of locking a distinct key value and all its index entries in a single lock manager invocation, i.e., lock overhead, and most of the advantages of locking individual index entries, i.e., high concurrency. Moreover, the proposed solution opens new opportunities for concurrency control in unique secondary indexes as well as primary indexes.

4 Case studies

In order to clarify the specific behaviors of the various locking schemes, this section illustrates the required locks and the enabled concurrency of each design for all principal types of index accesses. These comparisons are qualitative in nature but nonetheless serve to highlight the differences among the schemes. Differences in performance and scalability depend on the workload; an experimental comparison follows in Section 5.

The comparisons rely on a specific (toy) example table with employee information. This table has a primary index on its primary key (unique, not null) and a non-unique secondary index on one of the non-unique columns.

Figure 7 shows some index records in the primary index. The figure shows the rows sorted and the reader should imagine them in a b-tree data structure. Note the skipped values in the sequence of EmpNo values and the duplicate values in the column FirstName. This toy example has only two duplicates but indexes on real data may have thousands. Figure 8 illustrates index records in a non-unique secondary index on FirstName. This index format pairs each distinct key value with a list of bookmarks. Unique search keys in the primary index serve as bookmarks; they are also the table's primary key here.

EmpNo	**FirstName**	**PostalCode**	**Phone**
1	Gary	10032	1122
3	Joe	46045	9999
5	Larry	53704	5347
6	Joe	37745	5432
9	Terry	60061	8642

Figure 7. An example database table.

FirstName	Count	**EmpNo**s
Gary	1	1
Joe	2	3, 6
Larry	1	5
Terry	1	9

Figure 8. An example non-unique secondary index.

4.1 Empty queries – phantom protection

The first comparison focuses on searches for non-existing key values. Assuming a serializable transaction, a lock is required for phantom protection until end-of-transaction. In other words, the first comparison focuses on techniques that lock the absence of key values. The example query is "Select... where FirstName = 'Hank' ", which falls into the gap between key values Gary and Joe. Recall that we excluded in Section 2 any locking design like Tandem's in which an unsuccessful query may introduce a value into the database or the lock manager.

ARIES/KVL cannot lock the key value Hank so it locks the next higher key value, Joe. This locks all occurrences of the distinct key value without regard to EmpNo values. Thus, no other transaction can insert a new row with FirstName Hank but in addition, no other transaction can modify, insert, or delete any row with FirstName Joe.

ARIES/IM also locks the next higher key value, i.e., it locks the first occurrence of Joe and thus the row with EmpNo 3. A single lock covers the row in the table, i.e., the index entry in the primary index, the index entry in the secondary index (and any further secondary indexes), plus (in each index) the gap between those index entries and the next lower key value present in the index. While this lock is in place, no other transaction can insert a new row with FirstName Hank. In addition, no other transaction can insert new index entries (Gary, 7) or (Joe, 2), for example, because these index entries also belong into the gap locked for phantom protection, whereas a new index entry (Joe, 4) could proceed. In fact, due to the lock's scope in the primary index, no other transaction can insert any row with EmpNo 2.

Key-range locking in Microsoft SQL Server locks the first index entry following the unsuccessful search, i.e., the index entry (Joe, 3). The search in the secondary index does not acquire any locks in the primary index. Insertion of a new row with FirstName Joe is possible if the EmpNo is larger than 3, e.g., 7. Insertion of a new employee (Joe, 2) or (Gary, 7) is not possible.

Orthogonal key-range locking locks the key preceding a gap, i.e., the index entry (Gary, 1) in NS mode (pronounced 'key free, gap shared'). Insertion of new rows with FirstName Gary are prevented if the EmpNo value exceeds 1. On the other hand, non-key fields in the index entry (Gary, 1) remain unlocked and another transaction may modify those, because a lock in NS mode holds no lock on the key value itself, only on the gap (open interval) between index entries. The restriction to non-key fields is less severe than it may seem: recall from Section 2.4 that an index entry's ghost bit is a non-key field, i.e., deletion and insertion by toggling the ghost bit are possible. The lock matrix of SQL Server lacks a RangeS_N mode that would be equivalent to the NS mode in orthogonal key-range locking.

Finally, orthogonal key-value locking locks the preceding distinct key value, Gary, in a mode that protects the gap (open interval) between Gary and Joe but imposes no restrictions on those key values or their lists of EmpNo values. For example, another transaction may insert a new row with FirstName Gary or Joe and with any EmpNo value. Removal of rows with FirstName Joe has no restrictions; deletion of rows with FirstName Gary and removal of their index entries requires that the value Gary remain in the index, at least as a ghost record, until the need for phantom protection ends and the lock on key value Gary is released.

In summary, while all techniques require only a single lock manager invocation, orthogonal key-value locking provides phantom protection with the least restrictive lock scope.

4.2 Successful equality queries

The second comparison focuses on successful index search for a single key value. This case occurs both in selection queries and in index nested loops joins. The example query predicate is "…where FirstName = 'Joe' ", chosen to focus on a key value with multiple instances in the indexed column. While the example shows only two instances, real cases may have thousands. Serializability requires that other transactions must not add or remove instances satisfying this search predicate.

ARIES/KVL requires only one lock for all instances of FirstName Joe. This lock pertains to the secondary index only. Within the secondary index, the lock covers the key value and the gap to the next lower key value, i.e., FirstName Gary. Thus, this lock also prevents insertion of a key value other than Joe, e.g., FirstName Hank.

ARIES/IM locks three rows in the table including both rows with FirstName Joe (rows 3 and 6) and the next higher key value, i.e., row 5 with FirstName Larry. The last lock is required to prevent other transactions from inserting additional instances, e.g., (Joe, 7). These locks include the gap to the next lower key in each index, i.e., both the primary index and the secondary index. Thus, they prevent insertion of new rows with FirstName Joe and EmpNo 2 or 4 as well as rows with FirstName Ken and rows with FirstName Larry and EmpNo smaller than 5.

SQL Server locks each instance of the desired key value with its unique index entry, i.e., (Joe, 3) and (Joe, 6), plus the next higher actual key value, i.e., (Larry, 5). The last lock prevents additional entries with FirstName Joe and EmpNo values greater than 6, but it also prevents insertion of additional entries with FirstName Larry and EmpNo smaller than 5 as well as key values between Joe and Larry, e.g., Ken.

Orthogonal key-range locking is similar to SQL Server locking except it locks the next lower key value from Joe instead of the next higher key value, i.e., Gary instead of Larry, and it leaves the additional record itself unlocked. A lock in NS mode (pronounced 'key free, gap shared') on key value (Gary, 1) leaves the existing index entry unlocked but it prevents insertion of new index entries with FirstName Gary and EmpNo values higher than 1, with FirstName values between Gary and Joe, e.g., Hank, and with FirstName Joe and EmpNo value smaller than 3. Only the last group is truly required to protect the result of the example query. This problem is inherent in all key-range locking schemes.

Finally, orthogonal key-value locking acquires a single lock covering all actual and possible index entries with FirstName Joe. Both adjacent key values remain unlocked, i.e., Gary and Larry. Even the gaps below and above FirstName Joe remain unlocked, i.e., other transaction are free to insert new index entries with FirstName Hank or Ken.

In summary, among all locking schemes for b-tree indexes, orthogonal key-value locking allows repeatable successful equality queries with the fewest locks and the best precision.

4.3 Other cases

A technical report compares the other principal access patterns in similar detail; the following is merely a summary.

In a range query, orthogonal key-value locking can match both open and closed intervals more precisely than all other techniques, in addition to requiring fewer locks than key-range locking and ARIES/IM in cases of duplicate key values.

In a non-key update of a single index entry, orthogonal key-range locking is the most precise method, with orthogonal key-value locking equal assuming an appropriate number of partitions.

In a single-row deletion, toggling the ghost bit in a record header is a non-key update. Thus, orthogonal key-range locking and orthogonal key-value locking are superior to the other techniques. A later system transaction performs ghost removal with latches only.

In a single-row insertion, a system transaction may first create a ghost record using latches but no locks. In the subsequent non-key update including toggling the record's ghost bit, orthogonal key-range locking and orthogonal key-value locking are superior to all earlier techniques.

The preceding cases cover all principal types of index access and compare their locking requirements. In all comparisons, orthogonal key-value locking fares very well. In queries, it is better than all prior techniques, including orthogonal key-range locking. In updates including deletion and insertion via ghost status, orthogonal key-range locking is best. Orthogonal key-value locking equals orthogonal key-range locking except in the case of hash collisions due to an insufficient number of partitions.

5 Experimental evaluation

In this section, we evaluate efficiency and concurrency of orthogonal key-value locking in order to assess the following benefits compared to prior locking techniques:

- The cost of taking orthogonal key-value locks is as low as that of taking coarse grained locks (Section 5.1).
- The concurrency of orthogonal key-value locks is as high as the concurrency with state-of-the-art prior lock modes (Section 0).
- Orthogonal key-value locking provides both of the above benefits at the same time, which no prior locking techniques could do (Section 5.3).

We implemented orthogonal key-value locking in a modified version of the Shore-MT [JPH 09] code base. In addition to orthogonal key-value locking, we applied several modern optimizations for many-core processors. Reducing other efforts and costs clarifies the differences in locking strategies and thus makes our prototype a more appropriate test bed to evaluate orthogonal key-value locking. Put differently, a system without efficient indexing, logging, etc. performs poorly no matter the locking modes and scopes. At the same time, these other optimizations amplify the need for an optimal locking technique. We found the following optimizations highly effective:

- Flush-pipelines and Consolidation Arrays [JPS 10] speed up logging.
- Read-after-write lock management [JHF 13] reduces overheads in the lock manager.
- Foster b-trees [GKK 12] make b-tree operations and latching more efficient and ensure that every node in the b-tree has always a single incoming pointer.
- Pointer swizzling [GVK 15] speeds up b-tree traversals within the buffer pool.

For the first and second optimizations, we thank the inventors for their generous guidance in applying their techniques in our code base. Our measurements confirm their observations, reproducing significant speed-ups in a different code base.

We emphasize that both orthogonal key-value locking and read-after-write lock management are modular improvements in a database code base. We first modified the existing lock modes to orthogonal key-value locking and then modified the existing lock manager in Shore-MT to read-after-write lock management.

In all experiments, we compiled our programs with gcc 4.8.2 with -O2 optimization. Our machine, an HP Z820 with 128 GB of RAM, runs Fedora 20 x86-64 on two Intel Xeon CPUs model E5-2687W v2 with 8 cores at 3.4 GHz.

To focus our measurements on efficiency and concurrency of the lock manager, we run all experiments with data sets that fit within the buffer pool. The buffer pool is pre-loaded with the entire data set (*hot-start*) before we start measuring the throughput. We also we use a RAMDisk (/dev/shm) as the logging device.

All experiments use TPC-C tables with 10 warehouses as well as workloads similar to queries within TPC-C transactions. The transaction isolation level is serializable in all experiments. Error bars in the figures show the 95% confidence intervals.

5.1 Locking overhead

The first experiment compares the overhead of taking locks using a cursor query that frequently appears in TPC-C. In short, the table schema and the query is described as follows.

```
CREATE TABLE CUSTOMER (
INTEGER WID, -- Warehouse ID
INTEGER DID, -- District ID
INTEGER CID, -- Customer ID
STRING LAST_NAME, …)
OPEN CURSOR
SELECT … FROM CUSTOMER WHERE WID=… AND DID=…
ORDER BY LASTNAME
CLOSE CURSOR
```

To process this query, virtually all TPC-C implementations build a secondary index on CUSTOMER’s WID, DID, and name [TPC]. We evaluate the performance of cursor accesses to the secondary index. We compare orthogonal key-value locking that uses the non-unique key prefix (WID, DID) as its lock key to traditional granular locking that takes a lock for each index entry, i.e., ARIES/IM [ML 92] and key-range locking [L 93].

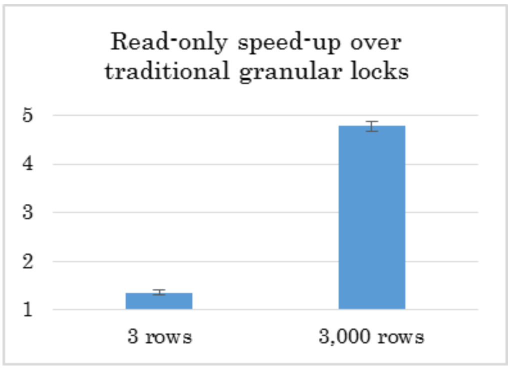

Figure 9. Lock counts and retrieval speed-up.

Figure 9 shows the speed-up achieved by orthogonal key-value locking compared to traditional granular locking in two settings. Note that the diagram shows system throughput, not just the effort spent on concurrency control.

In the first setting, we specify first-name in addition to WID and DID, touching only 3 keys per cursor access. In this case, the dominating cost is the index lookup to identify the page that contains the records. Thus, orthogonal key-value locking reduces lock manager calls by 3× but results in only 35% speed-up.

In the second setting, we do not specify first-name, thus hitting 3,000 keys per cursor access. In this case, the dominating costs are (i) locking overhead and (ii) the cost to read each record. As orthogonal key-value locking requires only one lock request in this case, i.e., one lock covers 3,000 index entries, it effectively eliminates the first cost, resulting in 4.8× better performance.

This experiment verifies the hypothesis that orthogonal key-value locking achieves the low overhead previously achieved only by coarse-grained locking, e.g., locking entire indexes or at least pages with 100s or 1,000s of index entries.

5.2 Concurrency

The experiment above showed that the overhead of orthogonal key-value locking is as low as coarse grained locking. However, traditional coarse-grained locking is known for its low concurrency. The second experiment evaluates the concurrency enabled by orthogonal key-value locking using a write-heavy workload.

We again use the TPC-C Customer table, but this time each transaction updates the table, specifying the primary key (WID, DID, CID). Orthogonal key-value locking uses the prefix (WID, DID) as lock identifier and CID as uniquifier. We compare orthogonal key-value locking with traditional key-value locking on (WID, DID).

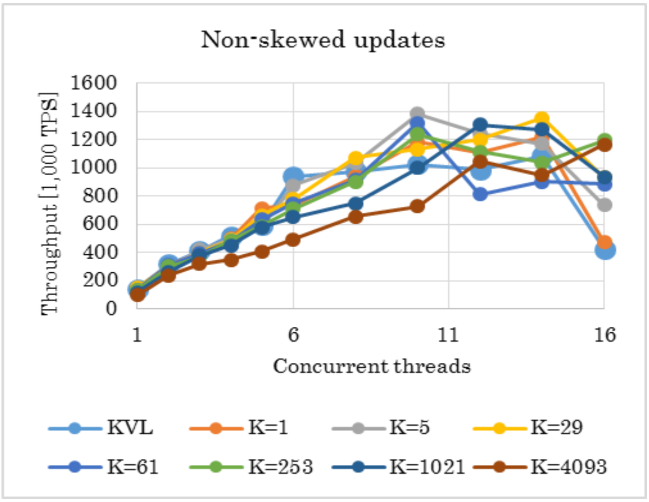

Figure 10. Transaction throughput with no skew in the requests.

Figure 10 shows the average transaction throughput, using uniformly random values for WID and DID in the retrieval requests. We varied the number of partitions in orthogonal key-value locking from k=1 (equivalent to traditional coarse-grained locking) to k=4,093. The values for k are chosen to be efficient in terms of CPU cache lines so that k+2 lock modes fit in 64 bytes or a multiply of 64 bytes.

As there is little skew, all configurations scale well with the number of concurrent threads until the system becomes over-subscribed; background threads conflict with the worker threads, e.g., logging and garbage collection in the lock manager.

One key observation is the high overhead of taking locks with extremely large values for k, e.g., k=4,093. Very large values for k cause many cache misses in the lock manager, making lock acquisition and compatibility tests expensive.

Figure 11 shows the result of the same experiment with skewed values of WID and DID in the updates; 80% of the transactions choose the first WID and 80% choose the first DID. Thus, concurrent transactions attempt to access the same index entries.

In this case, traditional key-value locking as well as orthogonal key-value locking with extremely few partitions (e.g., k=1) suffer from logical lock contention, hitting a performance plateau with only 3 threads. With larger k, orthogonal key-value locking is more concurrent and enables higher transaction throughput, except for an extremely large value (k=4,093), which causes cache-misses and physical contention in the lock queue.

This experiment verifies that, when there are many threads that are concurrently accessing the same data within the database, orthogonal key-value locking with a reasonably number of partitions achieves high concurrency because the lock compatibility of orthogonal key-value locking with many partitions is effectively equivalent to a very fine granularity of locking.

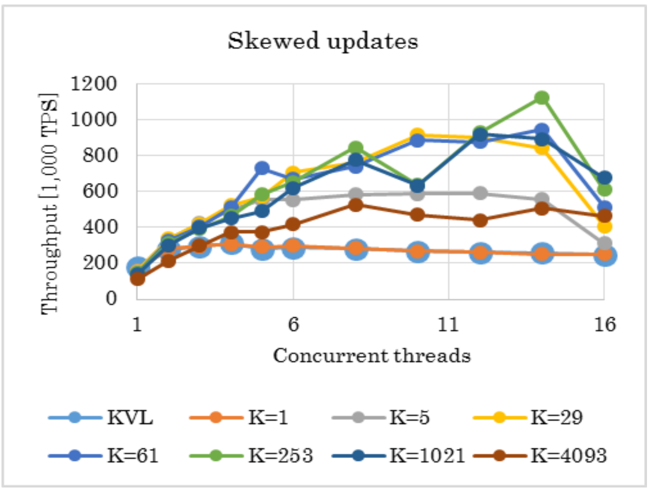

Figure 11. Transaction throughput with skew in the requests.

5.3 Mixed workloads

The experiments above show cases in which orthogonal key-value locking is as good as traditional locking modes, either coarse-grained or fine-grained. In this experiment, we show a more realistic case; a read-write mixed workload. In such a workload, both low overhead and high concurrency are required at the same time.

The workload consists of three transaction types; SELECT, INSERT, and DELETE. The mixture ratio is 40% SELECT, 40% INSERT, and 20% DELETE. Each transaction chooses WID with skew (90% of them chooses the first WID). All transactions use the STOCK table in TPC-C as shown below:

```
CREATE TABLE STOCK(
INTEGER WID, -- Warehouse ID
INTEGER IID, -- Item ID
…)
```

A transaction uniformly picks an item id and selects, inserts, or deletes tuples. STOCK table may or may not have the particular pair of WID and IID. Thus, SELECT and DELETE might hit a non-existing key. We compare four different lock modes to handle these cases:

- ARIES/KVL [M 90] locks unique values of the user-defined index key including a neighboring gap. Our implementation uses a lock on (WID) in this experiment.
- Key-range locking (KRL) [L 93] somewhat separates lock modes for unique index entries and gaps between their key values. As it lacks some lock modes (e.g., 'key free, gap shared'), our implementation takes a lock in a more conservative lock mode (e.g., 'key shared, gap shared', known as RangeS_S) in such cases.
- Orthogonal key-range locking (OKRL) [G 07, G 10, KGK 12] uses orthogonal lock modes for unique index entries and gaps between them.
- Orthogonal key-value locking (OKVL) uses (WID) as the lock key with (IID) as uniquifier. Orthogonal key-value locking acquires only one lock per transaction, but the lock contains partition modes for the accessed IIDs.

The next experiment uses 14 threads and 253 partitions in orthogonal key-value locking.

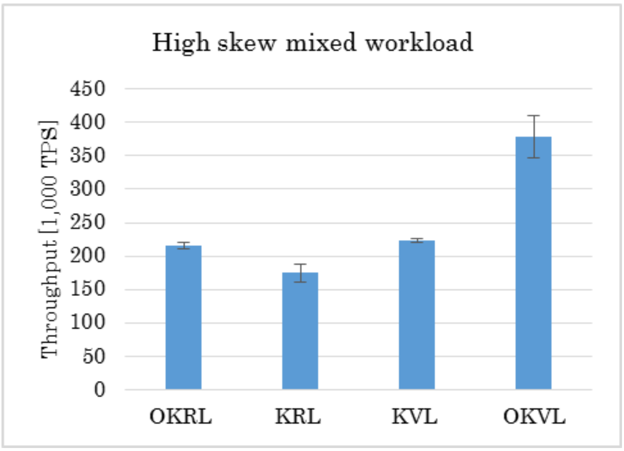

Figure 12. Locking and throughput of read-write transactions.

Figure 12 shows that with a highly skewed workload, ARIES/KVL (coarse-grained lock) fails to enable concurrency among updates. Its degree of parallelism is very low here, even if other experiments (Figure 9) show that is very efficient in single-threaded execution because it takes only one lock per transaction. On the other hand, key-range locking and orthogonal key-range locking enable more concurrent transactions but acquire many locks, which makes these techniques suffer from lock waits and even deadlocks.

In contrast, orthogonal key-value locking achieves both low overhead and high concurrency at the same time. It takes only one lock per transaction yet allows transactions to run concurrently, for 1.7-2.1× better transaction throughput. Thus, this experiment verifies our hypothesis that only orthogonal key-value locking combines the benefit of a coarse and a fine granularity of locking in b-tree indexes.

6 Conclusions

In summary, the new technique called orthogonal key-value locking combines design elements as well as advantages of (i) key-value locking in ARIES (a single lock for an equality query), of (ii) key-range locking (locking individual index entries for high concurrency), and of (iii) orthogonal key-range locking (independent lock modes for gaps between key values). The principal new techniques are (i) partitioning (only for the purpose of locking) each set of bookmarks within a non-unique secondary index and (ii) specifying individual lock modes for each partition and for the gap (open interval) between distinct key values.

A detailed case study compares the locking methods for ordered indexes (in particular b-trees) and demonstrates that orthogonal key-value locking is superior to all prior techniques for queries. For updates, it effectively equals the best prior method, which is orthogonal key-range locking. It performs sub-optimally only if ghost records must not be used for some reason or if the number of partitions is chosen so small that hash collisions within a list of bookmarks lead to false sharing and thus to lock conflicts.

Our prototype validates the anticipated simplicity of implementation in any database management system that already uses similar, traditional techniques, namely key-range locking or key-value locking. Like orthogonal key-range locking, and unlike prior lock techniques for b-tree indexes, orthogonal key-value locking permits automatic derivation of combined lock modes (e.g., for entire key value and gap) and automatic derivation of the lock compatibility matrix. It seems possible to automate even the derivation of test cases including expected test outcomes.

An experimental evaluation validates the insights gained from the case study: in situations with high contention, orthogonal key-value locking combines the principal advantages of key-value locking and (orthogonal) key-range locking. A read-only experiment shows retrieval throughput increase by 4.8 times and a mixed read-write workload shows a transaction throughput 1.7-2.1 times better than the prior techniques. Thus, we expect to find the new techniques in new implementations of b-trees and hope that they will replace the locking schemes in existing implementations. We hope that this study will affect design and implementation of databases, key-value stores, and (modern, b-tree-based) file systems alike.

Acknowledgments: Wey Guy, Ryan Johnson, Ilia Petrov, and the reviewers gave excellent feedback on earlier versions of this paper.

[BHG 87] Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman: Concurrency control and recovery in database systems. Addison-Wesley 1987.

[BS 77] Rudolf Bayer, Mario Schkolnick: Concurrency of operations on b-trees. Acta Inf. 9: 1-21 (1977).

[CAB 81] Donald D. Chamberlin, Morton M. Astrahan, Mike W. Blasgen, Jim Gray, W. Frank King III, Bruce G. Lindsay, Raymond A. Lorie, James W. Mehl, Thomas G. Price, Gianfranco R. Putzolu, Patricia G. Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, Robert A. Yost: A history and evaluation of System R. Comm. ACM 24(10): 632-646 (1981).

[G 78] Jim Gray: Notes on data base operating systems. Advanced course: operating systems. Springer 1978: 393-481.

[G 07] Goetz Graefe: Hierarchical locking in b-tree indexes. BTW 2007: 18-42.

[G 10] Goetz Graefe: A survey of b-tree locking techniques. ACM TODS 35(3) (2010).

[GKK 12] Goetz Graefe, Hideaki Kimura, Harumi Kuno: Foster b-trees. ACM TODS 37(3) (2012).

[GMB 81] Jim Gray, Paul R. McJones, Mike W. Blasgen, Bruce G. Lindsay, Raymond A. Lorie, Thomas G. Price, Gianfranco R. Putzolu, Irving L. Traiger: The recovery manager of the System R database manager. ACM Comput. Surv. 13(2): 223-243 (1981).

[GVK 15] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi Kuno, Joseph Tucek, Mark Lillibridge, Alistair Veitch: In-memory performance for big data. Accepted for publication, VLDB 2015.

[JHF 13] Hyungsoo Jung, Hyuck Han, Alan David Fekete, Gernot Heiser, Heon Young Yeom: A scalable lock manager for multicores. ACM SIGMOD 2013: 73-84.

[JPH 09] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, Babak Falsafi: Shore-MT: a scalable storage manager for the multicore era. EDBT 2009: 24-35.

[JPS 10] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, Anastasia Ailamaki: Aether: a scalable approach to logging. PVLDB 3(1): 681-692 (2010).

[KGK 12] Hideaki Kimura, Goetz Graefe, Harumi A. Kuno: Efficient locking techniques for databases on modern hardware. ADMS@VLDB 2012: 1-12.

[L 93] David B. Lomet: Key range locking strategies for improved concurrency. VLDB 1993: 655-664.

[LY 81] Philip L. Lehman, S. Bing Yao: Efficient locking for concurrent operations on b-trees. ACM TODS 6(4): 650-670 (1981).

[M 90] C. Mohan: ARIES/KVL: a key-value locking method for concurrency control of multiaction transactions operating on b-tree indexes. VLDB 1990: 392-405.

[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 1992: 371-380.

[TPC] http://www.tpc.org/tpcc/results/tpcc_perf_results.asp.