Orthogonal key-value locking

Precise concurrency control in ordered indexes such as b-trees

Goetz Graefe

Abstract

B-tree indexes have been ubiquitous for decades in databases, file systems, key-value stores, information retrieval, and internet search. Over the past 25 years, record-level locking has also become ubiquitous in database storage structures. Multiple designs for fine-granularity locking in b-tree indexes exist, each a different tradeoff between (i) high concurrency and a fine granularity of locking during updates, (ii) efficient coarse locks for range queries and equality queries in non-unique indexes, (iii) run-time efficiency with the fewest possible invocations of the lock manager, and (iv) conceptual simplicity for efficient development, maintenance, and quality assurance as well as for users' ease of understanding and predicting system behavior.

Unnecessarily coarse lock scopes have contributed to giving locking and serializability a reputation for poor concurrency, poor scalability, and poor system performance. The diagram below illustrates the extent of the problem and of the opportunity, using as example phantom protection for non-existing key value 'Harry' in an ordered index such as a b-tree that maps people's first names to row identifiers. Shading indicates the required lock scope and, for two traditional and two novel locking techniques, the actual lock scopes. Other access patterns common in queries and updates show similarly big differences between traditional and novel techniques. Locking more than required is a principal cause of false conflicts among concurrent transactions.

Index entries and gaps → ↓ Techniques	entry (Gary, 1)	gap			entry (Jerry, 3)	gap	entry (Jerry, 6)	gap	
		Gary, >1	>Gary, <Jerry	Jerry, <3				Jerry, >6	>Jerry
Traditional b-tree locking									
Orthogonal key-value locking									

A new technique, orthogonal key-value locking, is simple and efficient yet supports both fine and coarse granularities of locking. A lock request may cover (i) a distinct key value with all existing and possible rows or row identifiers, (ii) a subset of these rows or row identifiers, (iii) a gap between adjacent existing key values, or (iv) a subset of key values within such a gap. Thus, each lock or each lock request covers none, some, or all of the rows associated with an existing key value plus none, some, or all of the non-existing key values in a gap between adjacent existing key values.

Using specific examples such as insertions, deletions, equality queries, range queries, and phantom protection, case studies with diagrams like the one above compare the new design with four prior ones. For queries, the new design dominates all prior ones including the designs in industrial use today. For updates, the new design is practically equal to the most recent prior design, which dominates all earlier ones. Experiments demonstrate that the new technique reduces the number of lock requests yet improves concurrency and throughput for read-only queries and for read-write transactions.

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 software permits dozens or even hundreds of secondary indexes for each table. In common practice, each database table has a few secondary indexes, e.g., on all foreign keys and on non-key columns frequently searched by applications and users.

Key values in secondary indexes may or may not be unique. There may be many, even thousands, of rows in a database table with the same value in an indexed attribute. In those cases, compression is useful, e.g., a bitmap instead of a list of row identifiers. Such representation choices, however, are entirely independent of choices for transactional concurrency control, where further improvements are possible despite multiple existing techniques [G 10] and decades with only moderate progress.

1.1 *Motivation*

Many transaction processing applications require record-level locking in their databases. Moreover, reliably correct applications require the simplest and most robust programming model with respect to concurrency and recovery, which only 'serializable' transactions can offer. Thus, the focus here is on concurrency control beyond 'read committed' and 'repeatable read' transaction isolation. Unfortunately, many systems developers disparage both serializability and locking due to their effects on concurrency, perceiving them as unreasonably limiting. Their remedy often are system designs and implementations that fail to preserve serializability among concurrent queries and updates, thus leaving substantial work to application developers and users to prevent or detect and repair cases in which insufficient transaction isolation leads to erroneous database output or even erroneous database updates. The present work aims to weaken arguments against serializability. Its new mechanisms for fine-granularity locking restrict concurrent transactions where truly required but otherwise enable maximal concurrency.

The present research offers a new design with a new tradeoff between

- high concurrency and a fine granularity of locking during updates,
- efficient coarse locks for equality and range queries,
- run-time efficiency with the fewest possible invocations of the lock manager, and
- conceptual simplicity for efficient development, maintenance, and testing.

Invocations of the lock manager imply substantial run-time costs beyond a mere function invocation. Each invocation searches the lock manager's hash table for pre-existing locks and for lock requests waiting for the same lockable resource. Beyond the cost of the search, a lock manager shared among many threads requires low-level concurrency control, which is relatively expensive in many-core processors and multi-socket servers.

Among all components of a database management system, lock managers most urgently need simplicity in design, implementation, maintenance, test design, and test execution. "Stress-testing into submission" is not an efficient or pleasant way of releasing new software versions. Instead, a good theory can be immensely practical [Kurt Lewin]. Creative techniques outside of established theory, e.g., instant-duration locks and insertion locks, can wreak havoc with understanding, correctness, code maintenance, quality assurance, and release schedules.

In pursuit of a new tradeoff, the new design employs locks that may cover any combination of

- a existing key value with all (existing and possible) row identifiers,
- specific pairs of distinct key value and row identifier,
- a gap (open interval) between two (existing) distinct key values,
- specific (non-existing) key values within such a gap.

Using specific examples such as insertions, deletions, equality queries, and phantom protection, case studies compare the new design for locks in b-tree indexes with four prior ones. Experiments demonstrate that the new technique reduces the number of lock requests yet increases transactional concurrency, improving transaction throughput for read-only queries, for read-write transactions, and for mixed workloads.

1.2 *Concurrency control and lock scopes*

In relational databases, pessimistic concurrency control (i.e., 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 an individual index entry (representing one row and its value in the indexed attribute) or a distinct key value (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 insertions, updates, and deletions, because they permit concurrent transactions to modify 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 for a single key value. Many individual lock acquisitions not only incur overhead but also the danger of a failure late in the process. In a deadlock, the entire statement or transaction aborts and all earlier efforts are 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 when multiple concurrent

transactions attempt to modify separate index entries. A single lock may cover thousands of index entries among which only a single one is modified.

The proposed design combines these benefits, i.e., high concurrency (by using a fine granularity of locking) and low overhead (by using a coarse granularity of locking). It is a variant of multi-granularity locking (hierarchical locking) [GLP 75, GLP 76], with a new technique that locks multiple levels of the hierarchy by a single invocation of the lock manager yet copes with large or even infinite domains of values in the lower level of the hierarchy.

1.3 *Design goals*

The primary design goal is correctness, e.g., two-phase locking and serializable transactions. The second design goal is simplicity for easier design, understanding, development, maintenance, and quality assurance. Accordingly, the proposed design is simpler than all prior ones, e.g., obviating "instant duration" locks, "insertion" lock modes, and similar ideas that are creative but outside the traditional concise theory for transactions and concurrency control.

The third design goal is high concurrency and therefore a fine granularity of locking, wanted for updates and in particular for insertions as well as for deletions and updates guided by searches in other indexes. Accordingly, the proposed design enables locks on individual index entries within lists associated with distinct key values as well as individual non-existing key values within a gap between two adjacent existing key values.

The final design goal is run-time efficiency and therefore a coarse granularity of locking, wanted in particular for large operations such as range queries and equality queries in non-unique indexes, including index search in nested loops join operations as well as the read-only search required to determine a set of rows to update or to delete. 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.

1.4 *Outline*

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 the new design and Section 4 compares the techniques using practical example scenarios. Section 5 outlines some future opportunities. Section 6 sums up the techniques and the results. An Appendix reports on our implementation experience and performance observations.

2 Prior designs

For b-tree indexes in databases, we assume the current standard design with user contents (data records or index entries) only in leaf nodes, also known as B$^+$-trees. Therefore, this paper focuses on key-value locking and key-range locking applied to key values in leaf nodes.

In this paper, the term "bookmark" means a physical record identifier (device, page, and slot numbers) if the table's primary data structure is a heap. If the table's primary data structure is an index such as a b-tree, a bookmark is a unique search key in the primary index. Such primary indexes are known as index-organized tables in Oracle, as clustered indexes in Microsoft's SQL Server, and as primary key indexes in Tandem's NonStop SQL. Other systems may use yet other names.

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. In a non-unique secondary index with a list of bookmarks for 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 a materialized summary ("group by") view and its indexes, a ghost record is a row summarizing zero base rows, i.e., sums and count equal to zero [GZ 04].

In most systems, an unsuccessful search within a serializable transaction locks a gap between adjacent 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 in data pages. It seems that code stabilization for this design was difficult and tedious. All methods discussed below lock a gap by locking an adjacent existing key value.

Locking gaps between key values can use next-key locking or prior-key locking. At leaf boundaries, next-key locking benefits from pointers to the next leaf page [LY 81]. Other systems use fence keys, i.e., ghost keys in the leaf nodes equal to the branch keys in their parent nodes [GKK 12].

2.1 *History of locking in b-trees*

Early work on locking in b-tree data structures focuses on protecting the physical data structure rather than the logical contents. Bayer and Schkolnick [BS 77] introduce a notion of safety that requires a unique path from the root to each leaf node; Lehman and Yao [LY 81] introduce temporary overflow nodes that violate the original b-tree design and its strictly uniform and logarithmic path length from root to any leaf node; and Foster b-trees [GKK 12] bridge that divide by letting an overflowing node serve as temporary parent ("foster parent") for its overflow node, with no permanent pointers among nodes at the same b-tree level.

Recent designs for lock-free (non-blocking) b-trees optimistically avoid blocking: updates prepare alternative nodes and then attempt to install them using an atomic compare-and-swap instruction applied to a single child pointer. In times of low contention among updaters, this approach promises better performance; in cases or places of high contention among updates (also known as hot spots), this approach promises a lot of wasted work because concurrent updates prevent installation of prepared alternative nodes. For example, a sequence of ever-increasing insertions creates a hot spot at the right edge of a b-tree. Lock-free designs perform poorly for this update pattern. Another weakness of lock-free data structures is the lack of a shared or read-only access mode.

A notable compromise between traditional pessimistic latching and lock-free data structures is partially optimistic, e.g., in the form of read-only shared latches [CHK 01]. They address the issue that a traditional shared latch counts the number of concurrent readers and thus twice updates the cache line containing the latch. In systems with multiple caches and CPU cores, exclusive access to a cache line incurs bus traffic, invalidations, etc. Read-only shared latches have each exclusive operation (writer) increment a version counter twice, once before and once after an update. When a reader starts, it can verify, without modifying the state of the latch, that the version counter is even and therefore no writer is active. When a reader ends and perhaps repeatedly while it progresses, it can verify that the version counter is unchanged and therefore no writer has been active in the meantime. Read-only shared latches are particularly valuable for non-leaf index nodes, where updates are infrequent and can often be deferred, e.g., in Foster b-trees [GKK 12].

	Locks	Latches
Coordinate …	User transactions	Threads
Protect …	Database contents	In-memory data structures
During …	Entire transactions	Critical sections
Modes …	Shared, exclusive, update, intention, escrow, schema, etc.	Read, write, (perhaps) update
Deadlock …	Detection & resolution	Avoidance
… by …	Analysis of the waits-for graph, timeout, transaction abort, partial rollback, lock de-escalation	Coding discipline, "lock leveling"
Kept in …	Lock manager's hash table	Protected data structure

Figure 1. Locks and latches.

Protection of the physical data structure is not the topic of this paper, however. Instead, following [G 10], the proposed design assumes a strict division of low- and high-level concurrency control. The former, also known as latching, coordinates threads in order to ensure the physical integrity of in-memory data structures, including those of lock manager, transaction manager, and buffer pool, including in-memory images of database pages. The latter, transactional locking, coordinates transactions in order to protect the logical contents of database tables and indexes. Figure 1, copied verbatim from [G 10], summarizes most differences between locking and latching.

The proposed design assumes the need for a granularity of locking finer than pages or entire b-tree nodes. ARIES/KVL ("key-value locking") [M 90] and ARIES/IM ("index management") [ML 92] are early

designs that break through the preceding restriction to page-level locking. This restriction had been imposed on System R [CAB 81], for example, by its recovery techniques [GMB 81] that rely on idempotent "redo" and "undo" actions applied to page images [G 78]. The alternative is "exactly once" application of log records, including compensation log records, based on page version information or PageLSN values [MHL 92]. These log records are also known as "rollback" log records or "update back" log records.

Product developers and researchers at DEC [L 93] and later Microsoft designed "key-range locking," which partially separates locking index entries from locking the ranges (gaps, open intervals) between index entries. A later design [G 07] separates those lock scopes completely. This design exploits ghost records for logical deletion and insertion. These records are called "pseudo-deleted records" in ARIES and are used there only for deletion but not for insertion of new index entries. Asynchronous system transactions, akin to "nested top actions" in ARIES, clean up superfluous ghosts. Based on these separations of index entry versus gap, logical versus physical insertion and deletion, and user transaction versus system transaction, let us call this last design "orthogonal key-range locking."

The most recent developments apply the concepts of orthogonality and hierarchy to key-value locking. A distinct key value is locked separately from a gap between existing key values, and partitioning is applied to both the set of bookmarks associated with an existing key value and the set of non-existing key values in a gap. This design is the focus of Section 3.

Design name	Origin	Granularity of locking	Comments
System R	IBM 1981	Page	No record-level locking due to recovery logic
ARIES/KVL	IBM 1990	Distinct key value	All possible instances plus gap to next lower "Instant duration" IX locks for insertions
ARIES/IM "data-only locking"	IBM 1992	Logical row	Heap record + all index entries + gaps to next lower index entries
ARIES/IM "index-specific locking"	IBM 1992	Index entry	Including gap to next lower index entry
Key-range locking	DEC 1993	Existing index entry Half-open interval	Partial separation of index entry vs gap Combined lock modes "Insert" lock mode
Orthogonal key-range locking	Microsoft 2007	Existing index entry Open interval between existing index entries	Cartesian product – simple derivation of combined lock modes and their compatibility
Orthogonal key-value locking	HP 2014	Open interval between distinct key values Distinct key value Partitions of bookmarks	All possible instances Hierarchy of key value and partitions
	Google 2016	Added: partitions of possible gap values	Hierarchy of gap and partitions

Figure 2. Record-level locking in b-tree indexes.

Figure 2 summarizes the history of fine-granularity locking in ordered indexes such as b-trees. It ignores latching, i.e., coordination of threads to protect in-memory data structures, as well as snapshot isolation and multi-version concurrency control, which seem to be the next frontier [BHE 11, CRF 09, LFW 12, PG 12]. Perhaps the ideal is a hybrid model [BHG 87] using snapshot isolation supported by multi-version storage for read-only transactions and two-phase locking for read-write transactions [CFL 82]. In this hybrid, the commit point (serialization order) of a read-only transaction is equivalent to the transaction's start time whereas the commit point (serialization order) of a read-write transaction is equivalent to the transaction's end time.

Chan et al. [CFL 82] wrote about this hybrid: "Read-only (retrieval) transactions, on the other hand, do not set locks. Instead, each of these transactions can obtain a consistent (but not necessarily most up-to-date) view of the database by following the version chain for any given logical page, and by reading only the first version encountered that has been created by an update transaction with a completion time earlier than the time of its own initiation. Thus, read-only transactions do not cause synchronization delays on update transactions, and vice versa."

The remainder of this paper focuses on serializable read-write transactions, ignoring read-only transactions, snapshot isolation, and multi-version concurrency control.

2.2 *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 3, copied verbatim from [M 90] with a new column order, 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.

		Current key value	Next key value
Fetch & fetch next		S for commit duration	
Insert	Unique index	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	IX for instant duration
	Non-unique index	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	IX for instant duration if *apparently* insert key value *doesn't* already exist No lock if insert key value already exists
Delete	Unique index	X for instant duration	X for commit duration
	Non-unique index	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	X for commit duration if *apparently* delete key value will no longer exist No lock if value will definitely continue to exist

Figure 3. Summary of locking in ARIES/KVL.

Figure 3 provides guidance for insertions and deletions but not for updates. A value change in an index key must run as deletion and insertion, but a change in 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." Moreover, in transaction processing databases, a table's primary storage structure may be a b-tree, e.g., as a "clustered index" or as "index-organized table." Most importantly, toggling a record's "ghost" flag (logical deletion and re-insertion of an index entry) is a non-key update. 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 try to turn the same ghost into a valid record.

2.3 *ARIES/IM "index management"*

ARIES/IM [ML 92] locks logical rows in a table, represented by records in the table's primary data structure, which the design assumes to be 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 key value. Compared to ARIES/KVL, this design reduces the number of locks in update transactions. For example, deleting a row requires only a single lock, independent of the number of indexes for the table. 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.

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

Figure 4. Summary of locking in ARIES/IM.

Figure 4, copied verbatim from [ML 92] with a new column order, compares in size and complexity rather favorably with Figure 3, because of much fewer cases, conditions, and locks. 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. Again note the omission of updates in non-key fields.

These conditions apply to secondary indexes and their unique entries (perhaps including row identifiers) if ARIES/IM is applied to each individual index. 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]. No such implementation seems to exist but key-range locking in SQL Server comes close.

2.4 *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 an index entry and the gap to the adjacent index entry.

SQL Server implements Lomet's description of key-range locking [L 93] quite faithfully. Locks pertain to a single index, either a table's clustered index (elsewhere known as primary index or index-organized table) or one of its non-clustered indexes (secondary indexes). Each lock covers one index entry (made unique, if necessary, by including the table's bookmark) plus the gap to the next lower index entry (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.

The set of lock modes follows [L 93]. S, U, and X locks are shared, update, and exclusive locks for a gap and an index entry; N stands for no lock. RSS, RSU, RXS, RXU, and RXX locks distinguish the lock mode for the gap between index entries (the mode prefixed by "R" for "Range") and for the index entry itself. RIN, RIS, RIU, and RIX are "insertion" locks held for instant duration only and used for insertions into gaps between existing index entries. RI_ lock modes are outside traditional theory of concurrency control. Both the design and the implementation lack RSN, RSX, RXN, and all possible RU_ modes.

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

2.5 *Orthogonal key-range locking*

Orthogonal key-range locking is somewhat similar to key-range locking in SQL Server, but with completely orthogonal lock modes for index entry and gap. Prior-key locking (rather than next-key locking) is recommended such that a lock on an index entry can include a lock on the gap to the next higher index entry [G 07, G 10]. For example, if many transactions append new key values to an index on order number, one transaction can hold a lock on its recent insertion, the next transaction can check the gap for phantom protection before inserting its new order number, and orthogonal key-range locking prevents false conflicts.

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

Figure 5. Construction of lock modes in orthogonal key-range locking.

Figure 5 illustrates combined lock modes covering index entry and gap derived from traditional lock modes. Concatenation of a lock mode for index entries 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. The names of the locks mention key and gap; for example, the XN lock is pronounced "key exclusive, gap free."

If index entry and gap are locked in the same mode, Figure 5 uses a single lock mode for the combination of index entry and gap. In this way of thinking, all other (two-letter) lock modes imply an intention lock for the combination of index entry and gap. For example, SX ("key shared, gap exclusive") implies an IX lock on the combination of index entry and gap. Alternatively, one can stay with separate (two-letter) lock modes and derive lock compatibility strictly component by component. There is no practical difference between these two ways of thinking, e.g., in the enabled concurrency or in the number of lock manager invocations.

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 6. Lock compatibility in orthogonal key-range locking.

Figure 6 shows the compatibility matrix for the lock modes of Figure 5. 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 an index entry 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 ensures phantom protection for the gap (open interval) between two index entries.

2.6 *A note on b-tree compression by suffix truncation*

Bayer and Unterauer [BU 77] recommend compressing b-tree keys by truncating redundant prefixes and suffixes. Any prefix shared by all keys in a b-tree node is stored only once. Not surprisingly, prefix truncation is most effective in leaf nodes. Suffix truncation reduces branch keys in branch nodes, i.e., parent nodes and further ancestor nodes, to the minimum required to guide future b-tree searches.

Suffix truncation is applied when splitting leaf nodes; it must not be applied again when splitting branch nodes. For maximum effectiveness, leaf splitting might forgo perfect balance between the two resulting leaf nodes and choose the shortest key value or key prefix that can guide future b-tree searches. Thus, it is possible that leaf splitting creates new artificial key values.

If b-trees forgo neighbor-to-neighbor pointers and instead use fence keys, i.e., copies of branch keys, and if fence keys participate in record-level locking, then leaf splitting and creation of new artificial key values may require duplication of existing locks, e.g., in key-range locking.

Key values in the overflowing leaf node
… “Gary”, “Harry”, “Jerry”, “Larry”, “Mary”, “Terry” …

Key values in the left node
… “Gary”, “Harry”, “Jerry”, “L”

Key values in the right node
“L”, “Larry”, “Mary”, “Terry” …

Figure 7. Key values before and after a leaf split.

Figure 7 illustrates this need. The shortest key string that splits the original overflowing leaf node near its center is “L”. This artificial key value is used as branch key in the parent node and as fence key in the leaf nodes after the split. Even if it is a ghost record that does not contribute to query results, it participates in key-range locking. Thus, if a user transaction requires phantom protection for non-existing key value “Kerri” and holds a lock on the gap between “Jerry” and “Larry” during the leaf split, this lock requires a duplicate attached to the new key value “L” such that the user transaction enjoys phantom protection for the gap between “Jerry” and “L” as well as the gap between “L” and “Larry”. It should also hold a shared lock on the new key value “L”.

2.7 *Summary of prior designs*

The present work assumes latches for low-level concurrency control, i.e., protecting in-memory data structures by coordinating threads, and focuses on locks for high-level concurrency control, i.e., protecting logical database contents by coordinating transactions. Moreover, it assumes earlier designs for hybrid concurrency control, i.e., snapshot isolation for read-only transactions and locking for all data accesses within read-write transactions. Thus, the remainder of this paper focuses on shared and exclusive locks that coordinate read-write transactions.

All prior solutions imply hard choices for the finest granularity of locking in a database index: it may be a logical row with all its index entries (e.g., ARIES/IM), 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, i.e., a coarse granularity of locking, or from excessive overhead, i.e., too many lock manager invocations. In contrast, orthogonal key-value locking, introduced next in Section 3, offers multiple lock scopes focused on a distinct key value within an index. Section 4 expands upon these differences among the techniques including a finer semantics of gaps between key values. It also illustrates these differences in Figure 22 to Figure 28.

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 update transactions. The proposed design combines principal elements of orthogonal key-range locking (complete separation of lock modes for key value and gap) and of ARIES key-value locking (a single lock for a distinct key value and all its instances). Therefore, its name is orthogonal key-value locking.

3.1 *Design goals*

As stated earlier in Section 1.3, the overall design goals are correctness, simplicity, concurrency, and efficiency. More specifically, the goal is to combine the advantages of key-value locking and of orthogonal key-range locking:

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

Orthogonal key-value locking satisfies two of these goals and comes close to satisfying the remaining one. In addition, it introduces

- phantom protection with better precision than prior methods for database concurrency control; and
- serializable insertion into gaps with concurrent phantom protection.

3.2 *Orthogonal locks on key value and gap*

In orthogonal key-value locking, the focus of each lock, and in fact its identifying "name" in the lock manager, is one distinct key value within one index. For each lock request, there are two lock modes, one for the key value and one for an adjacent gap, i.e., the gap to the next higher key value. Thus, the technique permits orthogonal lock modes for data and gaps. In contrast to prior orthogonal locking techniques, the data here are a distinct key value. A lock for a key value covers all its instances, including both the ones currently present in the index and all non-existing, possible instances.

Index entries and gaps	entry (Gary, 1)	gap			entry (Jerry, 3)	gap	entry (Jerry, 6)	gap			entry (Mary, 5)
		Gary, >1	>Gary, <Jerry	Jerry, <3				Jerry, >6	>Jerry, <Mary	Mary, <5	
A key value											
A gap											
Their combination											

Figure 8. Orthogonal locks for key values and gaps.

Figure 8 illustrates how in orthogonal key-value locking a lock can cover a distinct key value (including all existing and non-existing row identifiers), a gap between distinct key values, or the combination of a distinct key value plus the succeeding gap. The case studies in Section 4 and in particular Figure 20 and Figure 21 provide more detail on the underlying table and its index.

In system designs with neither ghost records nor system transactions, i.e., if user transactions create and erase key values, orthogonal locking of key and gap creates a subtle issue. An insertion may check for phantom protection, create a new key value, and then retain a lock only on the new key value. While the insertion remains uncommitted, a second transaction may use that key value for phantom protection in the adjacent gap. This lock is compatible with the existing exclusive lock as locks on keys and gaps are orthogonal. However, a transaction failure and rollback of the insertion may erase the new key value; the phantom protection lock should not impede the insertion transaction's ability to roll back. In that case, a third transaction could insert precisely the value for which the second transaction requires phantom protection; this third transaction would lock an adjacent key value but not the transient key value locked by the second transaction. Therefore, the obvious conflict would not be detected.

The required invariant is that locks can exist only on existing key values. If a key value is not yet permanent, it must not be locked except by the inserting transaction – and similarly for a key value about to be deleted. Thus, in this example, the second transaction must check whether another transaction holds the key value in exclusive mode. In fact, any transaction locking a gap for phantom protection must perform this check. This is similar to the "instant duration" locks in ARIES/KVL (Section 2.2) and ARIES/IM (Section 2.3), except that here the lock mode in the instant-duration lock request is shared to test for other transactions' exclusive locks.

A design that fully exploits ghost records and system transactions avoids this subtlety. Only system transactions create and erase key values, system transactions hold only latches but not locks, and user transactions only toggle ghost bits for logical insertions and deletions including their rollback. Note that each system transaction logs only a single log record, and that only completely unlocked ghosts are removed.

3.3 *Partitioned lists*

The second innovation of the 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 a list of bookmarks with each distinct key value, each bookmark pointing to an individual record in the table's primary storage structure.

The proposed technique divides a set of bookmarks for a specific existing key value into a fixed number of partitions, say k partitions. Methods for lock acquisition specify not just one but k lock modes in a single lock manager invocation. Thus, the proposed design extends the orthogonal locks of Section 3.2 and Figure 8 from 2 to k+1 lock modes for locking a distinct key value. One of the lock modes covers the gap to an adjacent distinct key value. The other k lock modes pertain to the k partitions of index entries. A lock acquisition may request the mode "no lock" for any partition or for the gap to the next key value.

Gender	Count	List of **EmpNo** values
'male'	173	2, 3, 5, 8, 10, 12, 13, 14, 19, 21, ...

Figure 9. A list of bookmarks as stored.

Figure 9 shows 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 a sorted list of bookmarks.

Gender	Partitions of **EmpNo** values			
'male'	8, 12...	5, 13, 21...	2, 10, 14...	3, 19...

Figure 10. Partitions of bookmarks for orthogonal key-value locking.

Figure 10 shows this list partitioned into k=4 partitions for orthogonal key-value locking. In this example, the assignment from bookmark value to partition uses a simple "modulo 4" calculation. This partitioning is only conceptual; the storage representation is shown in Figure 9. Index entries within the same partition (and for the same index key value) are always locked together, whereas index entries in different partitions can be locked independently.

Index entries and gaps	entry (Gary, 1)	gap			entry (Jerry, 3)	gap	entry (Jerry, 6)	gap			entry (Mary, 5)
		Gary, >1	>Gary, <Jerry	Jerry, <3				Jerry, >6	>Jerry, <Mary	Mary, <5	
An entire key value											
Partitions thereof											

Figure 11. Partitions within a list of bookmarks.

Figure 11 illustrates lock scopes possible due to partitioning within a list of bookmarks, using the format and index entries of Figure 8. Again using k=4 partitions and a "modulo 4" calculation as a hash function, partitions 3 and 1 are shown as locked (shaded).

It makes little difference whether the lock acquisition method lists 2 lock modes (as 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 5). 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 in orthogonal key-value locking lists k+1 modes. In order to lock only the gap between key values for an implementation with k=4 partitions per list, phantom protection may need a lock in mode 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 these 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 existing 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 collisions are possible, however). An unusual case is a lock request for two partitions, e.g., while moving a row within the table's primary store and thus modifying its bookmark without modifying the indexed key value.

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 the 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 example, orthogonal key-value locking lets two concurrent transactions delete bookmarks 8 and 13 in Figure 9, which is not possible with traditional key-value locking, i.e., ARIES/KVL.

For k=1, the new design is similar to the earlier design of Section 3.2. Recommended values are k=1 for unique indexes (but also see Section 3.6) and 3-100 for non-unique indexes. The best value of k depends on the desired degree of concurrency, e.g., the number of active hardware and software threads.

3.4 *Partitioned gaps*

A serializable transaction requires phantom protection for an unsuccessful query, i.e., a query with an empty result set. Without phantom protection, there is no guarantee for a repeatable count within the transaction.

Traditional locking techniques lock an entire gap. For example, ARIES/IM and key-range locking lock gaps between adjacent existing index entries and ARIES/KVL locks the gap between two distinct key values. However, if the predicate in an unsuccessful query is an equality predicate, locking an entire gap freezes more than is truly required. In other words, locking an entire gap prevents insertions that do not truly conflict with the unsuccessful query.

For example, if key values 80 and 90 are adjacent entries in an ordered index, a search for key value 84 comes up empty. Ensuring "repeatable count" transaction isolation (serializability) means preventing insertion of an index entry with key value 84. However, insertions of key values 81, 82, 83, 85, 86, 87, 88, or 89 could proceed without an actual conflict with the required protection for key value 84.

For those cases, Tandem introduced special-case logic in which an unsuccessful query may introduce a new value into the database or only into the lock manager. Anecdotal evidence suggests that this design required many test cases, many test runs, and much debugging effort. For example, multiple transactions may query and insert many key values within the same gap, some transactions may use range predicates that may end within gaps and on key values, some transactions may roll back to a savepoint or abort altogether, etc.

In contrast, orthogonal key-value locking can partition not only the list of bookmarks associated with an existing key value but also the set of possible, non-existing key values in a gap between two existing distinct key values. A lock request specifies a lock mode for each individual partition of possible key values. Note that in Section 3.3 partitions and their lock modes pertain to existing and non-existing bookmarks associated with an existing key value, whereas here partitions and their lock modes pertain to non-existing key values. It is not required that the two counts of partitions be equal but the discussion below assumes so for simplicity.

Index entries and gaps	entry (Gary, 1)	gap			entry (Jerry, 3)	gap	entry (Jerry, 6)	gap			entry (Mary, 5)
		Gary, >1	>Gary, <Jerry	Jerry, <3				Jerry, >6	>Jerry, <Mary	Mary, <5	
An entire gap											
A partition thereof											

Figure 12. Partitions within a gap.

Figure 12 illustrates lock scopes possible due to partitioning within a gap, using the format and index entries of Figure 8. In contrast to all prior methods of concurrency control in ordered indexes such as b-trees, phantom protection for a missing value, e.g., Larry, freezes less than a gap between adjacent index entries or key values.

Continuing the earlier example, if hash partitioning forms a few disjoint subsets of possible key values between 80 and 90, an unsuccessful query for key value 84 locks only one of these partitions while the other key values remain unlocked. Thus, insertions of values between 80 and 90 but different from 84 remain possible.

Ideally, a locked partition contains only a single key value, e.g., 84 in the example. It is possible that it contains multiple values due to hash collisions. For example, if the partitioning function is a modulo calculation for k=4 partitions, then a lock to protect key value 84 implicitly but unnecessarily prevents insertion of key value 88. The desired degree of concurrency governs the number of partitions. Using k=4 partitions and modulo as hash function, Figure 13 shows the lock footprint of phantom protection for non-existent key value 84.

Key	Gap				Key
80	84, 88	81, 85, 89	82, 86	83, 87	90

Figure 13. Phantom protection for key value 84.

If another transaction needs to insert a new distinct key value into a partition remaining unlocked in a phantom protection lock, it invokes a system transaction that inserts a ghost and augments the lock held for phantom protection. Specifically, the system transaction creating a ghost record with the new key value needs to create a lock on the new key value and assign it to the reader transaction. Thus, when the system transaction ends, the reader transaction has phantom protection on the two gaps below and above the new key value. If there are multiple transactions with locks on partitions of this gap, all of them require additional locks in this way. If the system supports incremental lock release during rollback to a savepoint or during transaction abort, each new lock must be next to its original lock.

In the example, if a user transaction holds a lock for phantom protection for key value 84 and another user transaction needs to insert key value 87, it invokes a system transaction to create a ghost with that key value. This system transaction duplicates locks such that the same partitions are locked for phantom protection in the gaps between 80 and 87 and between 87 and 90. Put differently, the system transaction copies all locks on the gap between 80 and 90 from the pre-existing key value 80 to the new key value 87. This is required because the original lock on the partition containing key value 84 does not indicate whether it is only for key value 84, only for key value 88, or for both, i.e., all key values in the partition. Note the similarity to lock duplication discussed earlier (Section 2.6).

The inserting user transaction requires an exclusive lock on the newly existing key value 87 before changing the ghost into a valid index entry. The insertion transaction does not require any shared locks on partitions within the original gap between key values 80 and 90 or in the final gaps below and above key value 87. A third concurrent transaction could insert key value 83 in the same way, also without a lock conflict. Note that key values 83 and 87 belong to the same partition in Figure 13.

Key	Gap				Key	Gap				Key
80	84	81, 85	82, 86	83	~~87~~	88	89			90

Figure 14, Locks after insertion of a ghost with key value 87.

Figure 14 illustrates locks after the insertion of ghost key 87. The first user transaction holds locks on key values 80 and 87, in each case a shared lock on one partition within the gaps above those key values. Thus, this transaction continues to have phantom protection for key value 84 and its hash collisions in the original gap between 80 and 90. The second user transaction holds an exclusive lock on key value 87 but not on any of the non-existing key values in either adjacent gap.

Key	Gap			Key	Gap				Key	Gap			Key
80		81	82	~~83~~	84	85	86		~~87~~	88	89		90

Figure 15. Locks after insertion of a ghost with key value 83.

Figure 15 shows the locks after a third user transaction inserts ghost key 83. The first user transaction continues to have phantom protection for key value 84 and its hash collisions in the original gap between 80 and 90. The second and third user transactions hold exclusive locks on the ghosts with key values 83 and 87, respectively. They can turn these ghosts into valid index entries and thus complete their logical insertions. There are no lock conflicts among these three user transactions.

The complexity of copying locks in a system transaction inserting a ghost record (key values 83 and 87 in the example) is the price for the additional concurrency enabled by partitioning a gap between existing key values. All prior serializable concurrency control techniques falsely find conflicts between the example transactions and their accesses to the three distinct key values 83, 84, and 87 – and none permits inserting key values 83 or 87 while another user transaction requires phantom protection for key value 84 within the gap between key values 80 and 90.

If the ghost insertion requires reorganization of the database page (e.g., ghost removal and free-space compaction) or even splitting an index leaf, locks on key values are not affected. Any such space management belongs into separate system transactions coded and invoked for these representation changes. The need for these system transactions is independent of orthogonal key-value locking. Creation

of a new key value as branch key in the parent node, e.g., due to suffix truncation [BU 77], may imply creation of a new fence key in the leaf nodes [GKK 12] and thus copying of phantom protection locks similar to the insertion of ghost key 87 in the example.

Insertion into a gap with a locked partition has no effect on the adjacent key values and on locks for these key values (as opposed to locks for gaps). In the example, it is not required to copy locks on the bookmarks of key value 80, and user transactions as well as the system transactions can ignore key value 90 and its locks. Note the contrast to ARIES/KVL (see Section 2.2 and Figure 3).

In a query with a range predicate, all hash partitions between two adjacent key values must be locked with the same lock mode for all partitions. An alternative design dedicates one lock mode to the entire gap in a special application of hierarchical locking. With this dedicated lock mode for all partitions, a query with an equality predicate lock a single partition together with an intention lock on the entire gap.

Performance studies in the Appendix do not include partitioning of gaps between existing key values.

3.5 *Lock manager implementation*

Lock requests specify k lock modes for partitions of bookmarks and another k lock modes for partitions of non-existing key values. Lock requests with 2k 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 2k partitions. On the other hand, in most implementations the costs of searching in a hash table and of concurrency control (latching) in the lock manager dominate the cost of manipulating locks. Locks are identified in the lock manager's hash table by the index identifier and a distinct key value, e.g., Gender in Figure 9, just as in ARIES/KVL.

The definition of lock compatibility follows the construction of Figure 6 from Figure 5: lock modes are compatible if they are compatible partition by partition. For example, multiple exclusive locks on the same key value but on different 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 partition by partition.

Releasing a lock usually requires re-computing the cumulative lock mode by scanning the list of all granted locks. The alternative is maintenance of multiple reference counts per lock. Releasing a lock on a single partition, however, might be simpler, because such a lock will often be an exclusive lock. In such cases, the cumulative lock mode may simply set the appropriate partition to "no lock."

While one transaction holds a shared lock for phantom protection on a partition of a gap, another transaction may acquire an exclusive lock on a different partition for insertion of a new key value. As part of the insertion, the query transaction needs to acquire an additional lock such that it achieves phantom protection on the same partition within the gaps on both sides of the new key value. In other words, both the insertion transaction and the query transaction hold locks on the new key value: the insertion on the key value and the query on the adjacent gap. A system transaction is the right implementation mechanism for creation of a new ghost record and for duplicating the existing lock. Should the user transaction and its insertion roll back, the deletion should be merely logical and retain the new key value as a ghost record. Eventually, another user transaction will turn the ghost into a valid record or a system transaction can remove the ghost record when it is unlocked.

This design follows the rule that user transactions inspect and modify logical database contents and system transactions modify its representation. Ghost removal, like all system transactions, requires no transactional locks. Instead, latches coordinate threads and protect data structures. Log records describe ghost removal because subsequent user transactions may log their actions with references to slot numbers within pages rather than key values. System transactions commit without log flush.

For small values of k, e.g., k=3 or even k=1 (e.g., for unique indexes), checking each partition is efficient. For large values of k, e.g., k=31 or k=127, and in particular if queries frequently lock all partitions at once, it may be more efficient to reserve one lock mode for the entire key value, i.e., all partitions of bookmarks, and another component for the entire gap, i.e., all partitions of non-existing key values. These additional entries in each lock request should also be reflected in the cumulative lock mode in the lock manager's internal state. In a sense, this change re-introduces a lock on the distinct key value and on the gap, combining aspects and advantages of both ARIES/KVL (locking all existing and possible instances of a key value with a single lock request and a uniform lock mode) and of orthogonal key-range locking (two separate lock modes for key value and gap).

With the additional lock on an entire distinct key value and on an entire gap, the number of components in the lock increases from 2k to 2k+2 lock modes for each key value in the index. Each of

these two additional lock components permit 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 or gap. 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 mode combines these two aspects. For example, 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.

3.6 *Unique secondary indexes*

Another optimization pertains to unique indexes. A natural inclination might be to employ orthogonal key-value locking with k=1, i.e., a single partition, at least for the list of bookmarks. An alternative specifically for multi-column unique keys locks some prefix of the key like the key in a non-unique index and the remaining columns (starting with the suffix of the unique key) like the bookmarks in the design for non-unique indexes.

Perhaps an example clarifies this case. Imagine two tables, e.g., "course" and "student," plus a many-to-many relationship in its own table, e.g., "enrollment." An index on enrollment may be unique on the combination of course number and student identifier, 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, e.g., on course number. In that case, queries may lock the entire enrollment information of one course (a class roster) with a single lock (locking all partitions in the list of student identifiers); yet updates may lock subsets of student identifiers (by locking only a single partition within a roster). With an appropriate number of partitions, the concurrency among updates is practically the same as with locks on individual pairs of course number and student identifier.

CourseNo	Count	List of (**StudentId**, **Grade**) pairs
CompSci 101	23	(**00**7, A), (*13*, F), …

Figure 16. A partitioned list of index entries.

Figure 16 illustrates the idea, even adding a further column into a secondary index. For each distinct key value, here the course number, there is a list of entries, each containing all remaining index columns. Note that the physical organization of the data structure and the techniques for concurrency control are independent of each other. The entries in the list are hash partitioned on the field that completes the unique index key, here the student identifier. As in Figure 9, hash partitions are computed "modulo 4" and indicated using **bold** and *italic* font choices.

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 course), an index that facilitates efficient navigation to the other entity type (e.g., on enrollment, with course number as leading key), and retrieval of all instances related to the initial instance. The opposite example is similarly typical, i.e., listing a single student's enrollment information (a student transcript) using an enrollment index with student identifier as leading column. It must be equally well supported in a database system, its indexes, its repertoire of query execution plans, its concurrency control, etc. B-tree indexes cluster relevant index entries due to their sort order; the proposed use of orthogonal key-value locking supports these queries with the minimal number of locks and the minimal number of lock requests.

Splitting unique keys into prefix and suffix and applying orthogonal key-value locking to prefixes affect phantom protection. For example, in an index with course number as prefix and student identifier as suffix, phantom protection for a specific missing student identifier within an existing course must lock a partition of suffixes, i.e., similar to a partition of bookmarks, whereas phantom protection for a non-existing course must lock a partition of course numbers, i.e., similar to a partition of non-existing key values. A range predicate on student identifiers within an existing course number must lock the entire course number, i.e., similar to locking an entire key value with all its partitions of bookmarks. Perhaps these locking pattern for course number and student identifier, or for prefix and suffix within a unique index, reflect the behavior of applications more naturally than traditional locking techniques. The traditional real-world terms roster and transcript suggest that these access patterns are indeed common.

3.7 *Primary indexes*

In many cases, a table's primary key is also the search key in the table's primary index. In other cases, in particular if the user-defined search key in the primary index is not unique, the table's bookmark adds a system-generated value to the user-defined search key. This value can be unique by itself within the table or even the entire database. For example, some systems attach a unique, high-resolution time stamp to each row upon creation, often called "database key" or a similar name. In some systems, the system-generated value is unique only for a specific table and for a specific value of the user-defined search key. If a specific value of the user-defined search key happens to be unique, this system-generated "uniquifier" might be null or even entirely absent.

If the user-defined search key for a primary index is unique, orthogonal key-value locking in the primary index does not benefit from partitioning. In those cases, orthogonal key-value locking with k=1 works like orthogonal key-range locking, except that gaps may still be partitioned. For primary indexes with non-unique keys, partitioning is appropriate for the system-generated key suffix. In other words, the user-defined key value is locked like a distinct key value in a non-unique secondary index and system-defined suffix values are partitioned like bookmarks.

With this design, when a unique bookmark value (including the suffix) is used to search the primary index, e.g., when fetching additional columns after searching a secondary index, a single partition is the appropriate granularity of locking in the primary index. When a specific value only for the user-defined search key is used to search the primary index, e.g., based on an equality predicate in a query, orthogonal key-value locking can, with a single lock request, lock all rows with that search key value.

When a search is not successful in a serializable transaction, phantom protection is needed in primary indexes just as much as in secondary indexes. Orthogonal key-value locking, using a lock on the gap between distinct key values or on partitions thereof, protects such a transaction against insertion of phantom records without blocking any operations on the adjacent key values. In other words, insertion and deletion of rows with one of the adjacent user-defined key values remain unrestricted by the lock required for phantom protection. With partitioning of the gap between distinct values of the user-defined key, phantom protection locks only a subset of possible future values.

3.8 *Master-detail clustering*

Merged indexes [G 11] are b-tree structures with entries from multiple indexes, typically on multiple tables, with key values constructed in such a way that related data is co-located by the sort order in the b-tree. For example, to cluster information on orders and order details, all records and their sort keys start with an order number. The index identifier (and thus the indication of a record's internal format) is a minor sort key. As another example, of a table representing a many-to-many relationship such as enrollment of students in courses, two indexes may be merged with indexes on two tables representing the connected entity types, i.e., students and courses. Note that such b-trees and indexes can still support uniqueness constraints, e.g., on student identifier for the student table or on student identifier plus course number for the enrollment table.

Extreme forms of this idea are domain indexes, i.e., a single b-tree combining all single-table indexes on the same key column, e.g., customer identifier. Key-value locking enables a form of object locking in merged indexes. For example, if each lock covers all index entries with a given order number, it covers an order and its order details. This lock scope is often desirable as it satisfies a query's concurrency control needs with a single invocation of the lock manager. Similarly, a single key-value lock on a student identifier may cover a student record plus all index entries for the student's transcript.

Orthogonal key-value locking can satisfy these situations but also other ones. Specifically, by partitioning the records, it is possible to lock only a single order detail. In more complex object types and in merged indexes, a partition may cover a specific group. For example, within a customer object and index on customer identifier, one partition might cover all orders or all invoices of a specific customer.

gap	key	gap	key	gap	key	gap
	4711, …		4711,1, …		4711,2, …	

Figure 17. Master and detail records as individual key values.

Figure 17 illustrates a specific representation of a master record, e.g., order 4711, and its detail records, e.g., line items. This representation could guide a traditional assignment of index entries and gaps to locks, where each master record and each detail record are locked separately.

gap	key			gap
	4711, …	4711,1, …	4711,2, …	

Figure 18. Master and detail records as instances of a single key value.

Figure 18 illustrates an alternative representation that would guide an alternative assignment to locks. The entire complex object including master and detail records can be locked as a unit or the individual records can be locked as partitions. If indeed many transactions access master and detail records together, then locking entire complex object seems advantageous. While these representations may suggest alternative lock assignments, representation and lock assignment are orthogonal and either lock assignment could be used with either representation.

As the choices of data representation and of the granularity of locking are independent, these techniques also work in traditional separate indexes. For example, if orders and order details are indexed in separate b-trees, locks on complex objects may still be used. A referential integrity constraint between order details and orders enables locking in one index without inspecting the other index.

3.9 *Summary of orthogonal key-value locking*

A new technique, to be called orthogonal key-value locking, combines all the advantages of locking a distinct key value and all its index entries in a single lock manager invocation, i.e., high efficiency and low overhead for locking, and most of the advantages of locking individual index entries, i.e., high concurrency and low lock contention. Moreover, for unsuccessful equality queries, partitioning within a gap enables phantom protection with better precision and with less impact on concurrent insertions than prior techniques for concurrency control in databases, their tables, and their indexes. Specific optimizations apply to unique secondary indexes, primary indexes with compound keys, and merged indexes for master-detail clustering.

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

Figure 19. Lock scopes in orthogonal key-value locking.

Figure 19 summarizes the lock scopes that orthogonal key-value locking supports for key value Jerry, based on the database table shown in Figure 20 and its index entries shown in Figure 21. Note that locking a partition of bookmarks, e.g., for EmpNo 3 or 5, always includes partitions within the intervals at the ends of a key value, e.g., EmpNo < 3 and EmpNo > 6. Orthogonal key-value locking also supports any combination of lock scopes. For example, a serializable transaction searching the index with the predicate "FirstName in ('Harold', 'Harry')" may lock two partitions within the gap between Gary and Jerry. For another example, a search for "FirstName in ('Gerald', 'Jerry')" may lock all partitions of existing key value Jerry plus one partition in a gap. For a final example, a range query can, with a single invocation of the lock manager, lock a key value and its adjacent gap. This lock covers all existing and non-existing instances of the locked key value as well as all non-existing key values in the adjacent gap. For comparison, the bottom of Figure 19 shows the one and only lock scope possible in ARIES/KVL, i.e., a gap and an adjacent distinct key value with all instances.

4 Case studies

In order to clarify and compare the specific behaviors of the various locking schemes, this section illustrates the required locks in each design for all principal kinds of index accesses. These comparisons are qualitative in nature but nonetheless serve to highlight the differences between the schemes. Quantitative differences in performance and scalability depend on the workload; an experimental comparison can be found in the Appendix.

EmpNo	FirstName	PostalCode	Phone	HireYear
1	Gary	10032	1122	2014
3	Jerry	46045	9999	2015
5	Mary	53704	5347	2015
6	Jerry	37745	5432	2015
9	Terry	60654	8642	2016

Figure 20. An example database table.

Figure 20 shows a specific example table with employee information. The table has a primary key, which is, of course, unique and not null. Figure 20 shows index records in the primary index, which is on the table's primary key in this example. 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 small example has only two duplicate instances but a column value in a real index may have hundreds or thousands of instances.

FirstName	Count	EmpNos
Gary	1	1
Jerry	2	3, 6
Mary	1	5
Terry	1	9

Figure 21. Records in an example non-unique secondary index.

Figure 21 shows index records in a non-unique secondary index on FirstName. Unique search keys in the primary index serve as bookmarks. The index format in Figure 21 pairs each distinct key value with a list of bookmarks. The table could have additional secondary indexes, e.g., on PostalCode.

4.1 *Empty queries – phantom protection*

The first comparison focuses on searches for non-existing key values. A serializable transaction requires a lock for phantom protection until end-of-transaction. In other words, like the example in the abstract, this first comparison focuses on techniques that preserve the absence of key values. The example predicate is "…where FirstName = 'Harry' ", which searches the gap between key values Gary and Jerry. Section 4.3 considers designs like Tandem's in which an empty query may introduce a new value into the database or the lock manager. The discussion here considers locks only on the pre-existing key values and index entries shown in Figure 21. Figure 22 illustrates the following discussion.

ARIES/KVL cannot lock the key value Harry so it locks the next higher key value. This lock freezes all occurrences of key value Jerry. No other transaction can insert a new row with FirstName Harry but, in addition, no other transaction can modify, insert, or delete any index entry with FirstName Jerry. While the example shows only two instances of FirstName Jerry, there may be thousands in other examples. ARIES/KVL would lock all of them, giving an unsuccessful search a surprisingly large lock footprint.

ARIES/IM locks the next higher index entry, i.e., it locks the first occurrence of Jerry and thus the row with EmpNo 3. A single lock covers the row in the table and its entire representation, i.e., the index entry in the primary index, the index entry in the secondary index on FirstName, an index entry in each further secondary index, and (in each index) the gap between those index entries and the next lower index entry. While this lock is in place, no other transaction can insert a new row with FirstName Harry. In addition, no other transaction can insert new index entries (Gary, 7) or (Jerry, 2), for example, because these index entries also belong into the gap locked for phantom protection, whereas new index entries (Gary, 0) and (Jerry, 4) could proceed. Moreover, no other transaction can insert any row with EmpNo 2, because the

lock includes in the primary index the gap below EmpNo 3. Finally, if the database table of Figure 20 has another secondary index on PostalCode, the lock on EmpNo 3 also stops other transactions from inserting any PostalCode values between 37745 and 46045. These are rather counter-intuitive effects of a query with a predicate on FirstName.

Key-range locking in Microsoft SQL Server locks the first index entry following the unsuccessful search, i.e., the index entry (Jerry, 3). The unsuccessful search in the secondary index does not acquire any locks in the primary index. Insertion of a new row with FirstName Jerry is possible if the EmpNo is larger than 3, e.g., 7. Insertion of a new employees (Gary, 7), (Harumi, 11), or (Jerry, 2) is not possible until the transaction searching for Harry releases its locks.

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 index entry itself, only on the gap (open interval) between index entries. This restriction to updating non-key fields is less severe than it may seem: recall from Section 2.5 that an index entry's ghost bit is a non-key field, i.e., logical deletion and insertion by toggling a ghost bit are possible. Key-range locking in Microsoft SQL Server lacks a RangeS_N mode that would be equivalent to the NS mode in orthogonal key-range locking.

Index entries and gaps	entry (Gary, 1)	gap			entry (Jerry, 3)	gap	entry (Jerry, 6)	gap	
		Gary, >1	>Gary, <Jerry	Jerry, <3				Jerry, >6	>Jerry
ARIES/KVL			S on "Jerry"						
ARIES/IM		S on "3"							
KRL		S on "Jerry, 3"							
Orth. krl		NS on "Gary, 1"							
Orth. kvl						← N+S on "Gary"			
w/ gap part'g						← N+NNSN on "Gary"			

Figure 22. Required and actual lock scopes in phantom protection for 'Harry'.

Finally, orthogonal key-value locking locks the preceding distinct key value, Gary, in a mode that protects the gap (open interval) between Gary and Jerry but imposes no restrictions on this key value or its lists of EmpNo values. For example, another transaction may insert a new row with FirstName Gary or Jerry and with any EmpNo value. Removal of rows with FirstName Jerry has no restrictions; deletion of rows with FirstName Gary and removal of their index entries requires that the key 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.

Figure 22 illustrates required and actual lock scopes for an example unsuccessful query, i.e., for phantom protection. The column headings indicate ranges in the domain of index keys. Shading in the header row indicates the gap that needs phantom protection for key value Harry. An S in Figure 22 indicates that a serializable locking technique acquires a transaction-duration shared lock in order to prevent insertion of index value Harry. In orthogonal key-value locking without gap partitioning, the entire gap is locked using a lock on key value Gary; with gap partitioning, the S lock pertains only to one of the partitions within the gap. It is obvious that ARIES/KVL locks the largest scope. ARIES/IM shows the same range as key-range locking only because Figure 22 does not show the lock scope in other indexes of the same table. Orthogonal key-range locking locks less than key-range locking due to separate lock modes for index entry and gap. Orthogonal key-value locking locks the smallest scope, even without the gap partitioning techniques of Section 3.4.

In an equality query on the index key, partitioning within a gap as introduced in Section 3.4 can reduce the lock scope even further, i.e., to a fraction as illustrated at the bottom of Figure 22. By hash partitioning the possible key values within the gap between existing key values, the footprint of the lock and thus of phantom protection is as narrow as the single FirstName Harry and its hash collisions. People with names such as Gerhard or Hank will appreciate the increased concurrency in this gap.

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. All other techniques restrict concurrent operations not only on the non-existing key value Harry but also on pre-existing adjacent key values Gary and Jerry.

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 = 'Jerry' ", 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. Figure 23 illustrates the following discussion, with subscripts indicating separate locks and thus multiple lock manager invocations.

Index entries and gaps	entry (Gary, 1)	gap			entry (Jerry, 3)	gap	entry (Jerry, 6)	gap			entry (Mary, 5)
		Gary, >1	>Gary, <Jerry	Jerry, <3				Jerry, >6	>Jerry, <Mary	Mary, <5	
ARIES/KVL			S on "Jerry"								
ARIES/IM		S on "3"				S on "6"		S on "5"			
KRL		S on "Jerry, 3"				S		S on "Mary, 5"			
Orth. krl		NS on "Gary, 1"			S		S on "Jerry, 6"				
Orth. kvl				S+N on "Jerry"							

Figure 23. Lock scopes in an equality query for 'Jerry'.

ARIES/KVL uses a single lock for all instances of FirstName Jerry. This lock pertains to the secondary index only, with no effect on the primary index. It includes phantom protection, i.e., it prevents insertion of additional index entries with FirstName Jerry. The lock also covers the gap to the next lower key value, i.e., FirstName Gary. Thus, this lock also prevents insertion of a key value other than Jerry, e.g., FirstName Gerold or Harold or Jarold.

ARIES/IM locks three rows in the table, one more than the number of rows matching the query. These locks include the rows with FirstName Jerry (rows 3 and 6) and the next higher index entry, i.e., row 5 with FirstName Mary. The last lock is required to prevent other transactions from inserting additional instances, e.g., (Jerry, 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 Jerry and EmpNo 2 or 4 as well as rows with FirstName Larry and rows with FirstName Mary and EmpNo smaller than 5.

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

Orthogonal key-range locking is similar to SQL Server locking except it locks the next lower key value from Jerry instead of the next higher key value, i.e., Gary instead of Mary, and it leaves the additional record itself unlocked. A lock in NS mode (pronounced "key free, gap shared") on index entry (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 Jerry, e.g., Harry, and with FirstName Jerry and EmpNo value smaller than 3. Only the second group is truly required to protect the result of the example query. This problem is inherent in all locking schemes focused on index entries rather than distinct key values.

Finally, orthogonal key-value locking needs only one lock for all existing and possible index entries with FirstName Jerry. Both adjacent key values remain unlocked, i.e., Gary and Mary. Even the gaps below and above FirstName Jerry remain unlocked, i.e., other transaction can insert new index entries with FirstName Harry or Larry.

In summary, among all locking schemes for b-tree indexes, only orthogonal key-value locking allows repeatable successful equality queries with perfect precision and with a single lock, even if other methods employ multiple locks. Someone named James or Jim would appreciate the precise lock scope of orthogonal key-value locking.

4.3 *Phantom protection with ghost records*

The discussion of locking in successful equality queries also helps understanding lock scopes in phantom protection with a ghost record left behind by a prior deletion. For example, consider the example

of Section 4.1 and a pre-existing ghost with FirstName Harry. In this case, locks for an unsuccessful equality query are equal to the locks for a successful equality query for a key value with only a single instance or bookmark.

Index entries and gaps	entry (Gary, 1)	gap			ghost	gap			entry (Jerry, 3)
		Gary, >1	>Gary, <Harry	Harry, <47	(~~Harry, 47~~)	Harry, >47	>Harry <Jerry	Jerry, <3	
ARIES/KVL			S						
ARIES/IM		S_1				S_2			
KRL		S_1				S_2			
Orth. krl		S_1			S_2				
Orth. kvl				S					

Figure 24. Phantom protection with a pre-existing ghost record.

Figure 24 illustrates phantom protection for an empty selection query based on a pre-existing ghost record. The header row indicates the required lock scope and, with the strikethrough font, the pre-existing ghost record. The two methods locking distinct key values require a single lock on FirstName Harry. Whereas orthogonal key-value locking locks only FirstName Harry, ARIES/KVL also locks the gap between Gary and Harry. The other three techniques require two locks in order to cover the ranges of EmpNo values below and above 47.

Figure 24 shows a case with a single ghost record. If there were multiple ghost records matching the query predicate, then Figure 24 would resemble Figure 23 even more closely.

If no ghost record exists that matches an unsuccessful selection query, a query could invoke a system transaction to insert a suitable ghost record. For the methods focused on index entries rather than distinct key values, a careful choice of the EmpNo value in the ghost record improves upon the locks shown in Figure 24. If the chosen EmpNo is +∞, a single lock suffices in ARIES/IM and in key-range locking. Orthogonal key-range locking can use −∞ in the same way. These values differ because the traditional methods use next-key locking in order to cover a gap whereas the orthogonal methods use prior-key locking. Instead of −∞, orthogonal key-range locking can also simply use *null*, assuming it sorts lower than all non-*null* values.

Index entries and gaps	entry (Gary, 1)	gap			ghost	gap		entry (Jerry, 3)
		Gary, >1	>Gary, <Harry	Harry, < +∞	(~~Harry, +∞~~)	>Harry <Jerry	Jerry, <3	
ARIES/KVL			S					
ARIES/IM		S						
KRL		S						

Figure 25. Single-lock phantom protection with next-key locking.

Index entries and gaps	entry (Gary, 1)	gap		ghost	gap			entry (Jerry, 3)
		Gary, >1	>Gary, <Harry	(~~Harry, *null*~~)	Harry, > *null*	>Harry <Jerry	Jerry, <3	
Orth. krl				S				
Orth. kvl				S				

Figure 26. Single-lock phantom protection with prior-key locking.

Figure 25 and Figure 26 show the locks and their scopes for phantom protection using a new ghost record with a carefully chosen index entry. ARIES/KVL and orthogonal key-value locking have the same lock counts (one) and lock scopes as in Figure 24. The other three methods require two locks in Figure 24 but only one in Figure 25 and in Figure 26. ARIES/IM and key-range locking no longer lock FirstName values above Harry and orthogonal key-range locking no longer locks FirstName values below Harry. Conceivably, two ghost entries with FirstName Harry and EmpNo values −∞ and +∞ could further reduce

over-locking. In effect, such a pair of ghost entries permits locking a distinct key value using mechanisms focused on individual index entries.

Locking the EmpNo value +∞ in ARIES/IM has surprising consequences, however. In an ordered index on EmpNo, this value protects the high end of the key domain. If new EmpNo values, e.g., for newly hired employees, are chosen in an increasing sequence, unsuccessful selection queries with predicates on the FirstName column may prevent insertions of newly hired employees even with different FirstName values. Therefore, this optimization for phantom protection has surprising and thus undesirable consequences in systems and tables locked with ARIES/IM.

In summary, both with a pre-existing ghost record and with a new ghost record, none of the prior locking methods can match the perfect precision of orthogonal key-value locking.

4.4 *Range queries*

The fourth comparison focuses on range queries. The example query predicate is "…where FirstName between 'Jerry' and 'Mary' ". The query result includes all instances of Jerry and Mary. Locking for serializability must prevent insertion of additional instances of these key values and of key values in between. Figure 27 illustrates the following discussion.

ARIES/KVL needs two locks on key values Jerry and Mary. These locks cover the key values and the gaps between them. In addition, the first lock covers the gap below Jerry. This is required to prevent insertion of a new row with FirstName Jerry and EmpNo 2 but it also prevents new rows with FirstName Harry, Howard, Jack, etc.

ARIES/IM acquires four locks on rows with EmpNo 3, 6, 5, and 9. The lock on row with EmpNo 9 is required to stop other transactions from inserting new index entries with FirstName Mary and EmpNo values greater than 5, but it also prevents new rows with FirstName Mason, Taylor, etc. Moreover, these locks also lock index entries in further indexes as well as the gaps (open intervals) below each of those index entries.

Key-range locking in SQL Server requires four locks on index entries (Jerry, 3), (Jerry, 6), (Mary, 5), and (Terry, 9). As in ARIES/IM, the first lock includes the gap below (Jerry, 3) and the last lock protects the gap above (Mary, 5) from new entries with FirstName Mary and EmpNo values greater than 5.

Orthogonal key-range locking requires locks on (Jerry, 3) and (Jerry, 6) in S mode to cover the index entries and the gaps to the next higher key. Moreover, a lock on (Gary, 1) in NS ("key free, gap shared") prevents insertion of rows with FirstName Jerry and EmpNo less than 3 but also rows with FirstName Harry etc. Finally, a lock on (Mary, 5) must be in S mode in order to cover both the key value and the gap above. Locking the gap prevents new index entries with FirstName Mary and EmpNo values greater than 5, although it also prevents new entries with FirstName greater than Mary, e.g., FirstName Mason or Terrence. Thus, orthogonal key-range locking seems similar to key-range locking in SQL Server as both lock three index entries within the range of the query predicate and one index entry outside. Orthogonal key-range locking is slightly more precise than SQL Server as it locks the outside key value in a less restrictive mode not available in SQL Server's lock matrix.

Index entries and gaps	entry (Gary, 1)	gap			entry (Jerry, 3)	gap	entry (Jerry, 6)	gap			entry (Mary, 5)	gap			entry (Terry, 9)
		Gary, >1	>Gary, <Jerry	Jerry, <3				Jerry, >6	>Jerry, <Mary	Mary, <5		Mary, >5	>Mary, <Terry	Terry, <9	
ARIES/KVL			S_1						S_2						
ARIES/IM		S_1				S_2		S_3				S_4			
KRL		S_1				S_2		S_3				S_4			
Orth. krl		S_1			S_2		S_3				S_4				
Orth. kvl				S_1						S_2					

Figure 27. Lock scopes in a range query.

Finally, orthogonal key-value locking requires locks on the two distinct key values within the range of the query predicate, i.e., FirstName Jerry and Mary. For these, it locks all existing and possible EmpNo values. A range query in a serializable transaction locks gaps between key values and all their partitions. For the highest key value within the range of the query predicate, i.e., FirstName Mary, the lock request leaves the gap to the next higher key value free (no lock). Thus, two locks protect precisely the key range of the query predicate, with other transactions free to insert index entries outside the predicate range.

Figure 27 illustrates lock scopes for the example range query. ARIES/KVL and orthogonal key-value locking require only two locks because they lock distinct key values, not individual index entries. The other techniques require four locks and thus four lock manager invocations. All prior methods lock more than required, whereas orthogonal key-value locking locks precisely the required part of the key domain. People with names such as Irv, Mike, or Mohan will appreciate the improved concurrency and system performance.

In summary, orthogonal key-value locking requires the fewest locks yet it is the only technique that protects a key range with perfect precision. This also applies to key ranges that exclude one or both end points.

4.5 *Non-key updates*

The fifth comparison focuses on updates that modify non-key columns. In the primary index of the running example, such an update modifies any column other than the primary key. The secondary index of the running example requires a small definition change for such an update to be possible, namely a column added to each index entry, e.g., using the "include" clause supported by some systems, which puts the included column behind the bookmark in each index record. It is therefore irrelevant for the sort order and the index organization. A non-key update of this extended secondary index modifies such an included column. In the list representation of non-unique secondary indexes, each element in the list carries an instance of the included columns. This is the case discussed below. Imagine the secondary index of Figure 21 extended by "include PostalCode" and an update "update… set PostalCode = … where EmpNo = 3".

ARIES/KVL acquires locks in each index; in the non-unique secondary index of the extended example, it locks all instances of FirstName Jerry as well as the gap between FirstName Gary and FirstName Jerry.

ARIES/IM locks the affected logical row including all its index entries and the gaps below those index entries. With only a single lock request required, this case seems to be the principal design point for ARIES/IM.

Key-range locking in SQL Server locks only the index entry (Jerry, 3) and the gap below it. All other index entries, including all other index entries with FirstName Jerry, remain unlocked.

Orthogonal key-range locking locks only the index entry (Jerry, 3) but leaves the gaps below and above it completely unlocked. This is possible with a lock in XN mode ("key exclusive, gap free").

Finally, orthogonal key-value locking locks the distinct key value Jerry but only a single partition within the list of instances. For k=7 partitions, for example, this locks about 1 in 7 or 14% of those instances. For a short list as shown in Figure 21, this is usually a single instance. For a longer list, locking a partition may mean locking multiple instances even if only one instance needs locking. Larger choices of k may alleviate hash collisions and false sharing. The gaps below and above FirstName Jerry remain unlocked.

Figure 28 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 (Gary, 1)	gap			entry	gap	entry	gap	
		Gary, >1	>Gary, <Jerry	Jerry, <3	(Jerry, 3)		(Jerry, 6)	Jerry, >6	>Jerry
ARIES/KVL			X on "Jerry"						
ARIES/IM		X on "3"							
KRL		X on "Jerry, 3"							
Orth. krl	XN on "Jerry, 3" →								
Orth. kvl	NNNX+N on "Jerry" →								

Figure 28. Lock scopes in a non-key update.

Figure 28 also illustrates the lock scopes of user transactions inserting and deleting index entries via ghost records, because toggling the ghost bit in a record header is a prototypical case of a non-key index update. The XN lock in orthogonal key-range locking is on the entry "Jerry, 3"; in orthogonal key-value locking, the X lock pertains only to partition 3 within the list of row identifiers for key value "Jerry." Without ghost records, older techniques lock more, in particular also an adjacent key value or index entry as detailed in Figure 3 and Figure 4. The following sub-sections provide more details.

In summary, all schemes support an individual non-key update with a single lock. Orthogonal key-range locking is most precise, with orthogonal key-value locking equally precise given an appropriate number of partitions.

4.6 *Deletions*

The sixth comparison focuses on deletion of rows in a table and of index entries in a b-tree. Within an index, any update of a key column usually requires a pair of deletion and insertion. Thus, the sixth and seventh comparisons also cover updates on key columns in indexes. An example here is "delete… where EmpNo = 3".

There are three cases to consider. With increasing complexity, these are (i) turning an existing, valid index entry into a ghost, (ii) deletion of a single index entry, with other index entries remaining for the same key value, and (iii) removal of a distinct key value as part of deleting its last remaining index entry.

Non-unique secondary indexes need one bit for each index entry to indicate the ghost status. A key value becomes a ghost when its list is empty or when all remaining entries are ghosts. Ghost removal happens outside of user transactions, i.e., in system transactions [G 10] or ARIES nested top actions [MHL 92].

Deletion via a ghost is now a standard technique. Among other advantages, it ensures simple and reliable transaction rollback if a user transaction aborts or fails. Specifically, rollback of a user transaction never requires space allocation, e.g., splitting a b-tree node. Deletion via a ghost record, i.e., by toggling a ghost bit, requires the same locks as a non-key update, discussed in Section 4.5 and illustrated in Figure 28.

The remainder of this sub-section pertains to deletion without ghosts, even if it is not the recommended implementation technique. If nothing else, it offers a level comparison with ARIES, e.g., Figure 4, where the commit-duration X lock on the next key indicates the assumption of immediate record removal, i.e., removal without ghost records. Given that the purpose of commit-duration locks after a deletion is to ensure conflict-free re-insertion in case of a transaction rollback, in most cases the lock scopes equal those required for phantom protection.

ARIES/KVL distinguishes cases (ii) and (iii) above as well as unique and non-unique secondary indexes, as summarized in Figure 3. In all cases, one distinct key value remains locked in X mode until end-of-transaction in order to protect the transaction's ability to roll back. Even if the transaction removes only a single entry in a list of thousands, the entire list remains locked together with the key value and the gap to the next lower distinct key value. Worse yet, upon removal of a key value and its empty list, the next key remains locked, which also covers the gaps below and above the deleted key value in addition to the deleted key value itself.

ARIES/IM needs to lock the affected logical row including, in each unique secondary index, the row with the next higher index entry. The latter aspect prevents concurrent insertion of other logical rows with conflicting unique index keys. A system using the ARIES/IM techniques definitely benefits from deletion via ghost records.

SQL Server deletes via ghost records, which system transactions eventually remove. In order to adapt SQL Server key-range locking to deletion without ghost records, locks on the deleted index entry as well as the next higher index entry are required. The lock manager might as well remove the lock on the deleted (and removed) index entry as soon as the entry disappears in the page image in the buffer pool, quite comparable to the instant locks of the ARIES techniques (see Figure 3 and Figure 4).

The design of orthogonal key-range locking calls for ghost records during both insertion and deletion. If ghost records must be avoided for some reason, cases (ii) and (iii) above must lock the next lower index entry in the index in NX mode ("key free, gap exclusive") until end-of-transaction. The effect is quite similar to key-range locking in SQL Server, except that the next lower index entry itself remains unlocked.

Finally, orthogonal key-value locking also works best with ghost records during both insertion and deletion. When forced to run without ghost records, case (ii) above requires a lock on one of the partitions

within the newly created gap and case (iii) above requires a lock on one of the partitions of bookmarks for the existing and remaining key value. In a unique index, there is only a single set of partitions.

In summary, for deletion via ghost status, which has long been the preferred implementation technique, locking follows the rules for non-key updates, where orthogonal key-range locking and orthogonal key-value locking are best. For deletion without ghosts, ARIES/KVL and ARIES/IM lock much more than truly required, orthogonal key-range locking is slightly better than SQL Server key-range locking, and orthogonal key-value locking benefits from partitioning bookmarks and gaps. ARIES/IM requires the fewest locks in tables with multiple non-unique indexes but has the largest cost in terms of concurrency control scope, false sharing, and excluded concurrent transactions.

4.7 *Insertions*

The seventh comparison focuses on insertion of new rows into a table and thus new index entries in primary and secondary indexes. An example here is "insert… values (4, 'Jerry', 54546, 4499)".

As insertion is the opposite of deletion, there are again three cases to consider. With increasing complexity, these are (i) insertion via ghost status, (ii) insertion of another instance of an existing key value, and (iii) insertion of an entirely new distinct key value. The ghost record may be a remnant of a prior deletion or the product of a deliberately invoked system transaction. When a user transaction turns a ghost record into a valid record, which is the recommended implementation at least for the orthogonal locking techniques, it acquires locks as discussed in Section 4.5 and illustrated in Figure 28. The remainder of this sub-section focuses on traditional techniques that do not employ ghost records as preliminary states when inserting new entries into tables and indexes.

ARIES/KVL distinguishes cases as shown in Figure 3. In all cases, one of the distinct key values in the index remains locked in X mode until end-of-transaction, together with the gap to the next lower key value as well as possibly thousands of index entries.

ARIES/IM locks, with a single request, the new row in the table including all its new index entries as well as the gaps to the next lower key in each index. As in all techniques, a short test (called an instant-duration lock in ARIES) ensures that no other transaction holds a conflicting lock for phantom protection.

Key-range locking in SQL Server briefly locks the next index entry in RangeI_N mode (discussed as RI-N in Section 2.4) to test for conflicting phantom protection and then creates the new index entry, retaining an X lock on the new index entry. The lack of a RangeN_X mode forces this X lock, i.e., other transactions cannot query or update the gap between the new index entry and the next lower index entry.

Orthogonal key-range locking works best with a system transaction ensuring that a suitable index entry exists as a ghost, whereupon the user transaction merely turns the ghost into a valid index entry as a non-key update. For insertion without a ghost, orthogonal key-range locking first tests that no other transaction ensures phantom protection by holding a lock on the gap above the next lower index entry and then locks the new index entry, not including the gaps around it.

Finally, orthogonal key-value locking also works best with a ghost left behind by a prior deletion or by a system transaction. Once the ghost is in place, orthogonal key-value locking locks the appropriate partition of bookmarks associated with the distinct key value, exactly as described earlier for updates on non-key attributes. If insertion via a ghost record is undesirable for some reason, insertion of a new distinct key value requires a lock on the gap above the next lower distinct key value. This lock is for phantom protection and may be held only briefly. Thereafter, a lock on the new distinct key value and the appropriate partition are required, with no lock on the other partitions or on the gaps above or below the new key value. If the insertion merely adds another instance of an existing key value, a lock on the appropriate partition of bookmarks suffices.

In summary, for insertion via ghost status, locking follows the rules for non-key updates, where orthogonal key-range locking and orthogonal key-value locking are best. For insertion without ghosts, ARIES/IM requires the fewest locks but holds the largest concurrency control scope, whereas orthogonal locking techniques merely hold a lock on the newly inserted index entry after testing for conflicting locks retained for phantom protection. Insertion via ghost status is the recommended implementation technique for the two orthogonal techniques but it also offers advantages for the traditional techniques.

4.8 *Key updates*

Modifying the key value of an index entry requires (in practically all cases) deletion of the old index entry and insertion of a new index entry at a location appropriate for the new key value. Both the deletion

and the insertion benefit from ghost records as discussed above. If ghost records are used, both the deletion and the insertion require locks as discussed in Section 4.5 and illustrated in Figure 28. Thus, orthogonal key-range locking and orthogonal key-value locking provide the required concurrency control with better precision than all prior methods.

4.9 *Non-unique primary indexes*

The final case study assumes an alternative physical database design, namely a primary index on a non-unique attribute such as the HireYear column in the database table of Figure 20. (In a large and growing company with only weekly or monthly starting dates and orientation sessions for new employees, there may be hundreds of new employees on a specific day and thousands within a year.) In database tables with high insertion rates, a primary storage structure is desirable such that updates can be appended with high bandwidth; thus, a time-ordered primary storage structure is not uncommon. Data warehouse tables often benefit from a time-ordered storage structure because most queries focus on specific time periods.

If the primary storage structure has a non-unique search key such as HireYear, then index entries in secondary indexes refer to specific index entries in the primary storage structure using a search key plus an artificial additional value that uniquely identifies a row in the table. For example, in the database table of Figure 20, the database might silently append an integer value to the HireYear. Since this internal database value counts or numbers, in effect, the new employees within each calendar year, we will call it the HireNo below. A more generic but less pleasing name might be "uniquifier."

Locking in a non-unique primary storage structure may focus on individual rows, e.g., employees in Figure 20, or on distinct key values, e.g., years of hire. The access pattern of interest here is fetching rows from the primary index after a search in one or more secondary indexes. For example, index intersection in a data warehouse query may produce a set of pairs of HireYear and HireNo. The question then is the look footprint of each technique in the table's primary index upon a lookup with a pair of HireYear and HireNo. Let the example intersect index searches on "FirstName = 'Mary' " and "PostalCode = 53704" and then retrieve the row with HireYear 2015 and HireNo 2.

ARIES/KVL, which IBM never applied to primary indexes because primary storage structures in IBM's DB2 database systems are always heaps but not search structures such as b-trees, would lock an entire HireYear, i.e., all employees hired in 2015 plus the gap after 2014. In this particular example, the gap happens to be empty or non-existing because there is no year between 2014 and 2015 or because 2014 and 2015 are consecutive integer values. In a read-only data warehouse, locking all employees hired in 2015 seems perfectly acceptable, but in other environments (or during concurrent load operations in a data warehouse), locking an entire year's worth of employees when fetching column values for a single employee seems excessive.

ARIES/IM would not acquire any locks when accessing the primary storage structure; all required locks would have been acquired during the scans in the secondary indexes. Note, however, that index intersection typically eliminates most index entries found in the individual index scans. Retaining locks on all these logical rows, i.e., the union of two predicate clauses, seems excessive for freezing merely their intersection (even if the example query and database do not illustrate the difference between union and intersection). The obvious remedy releases locks on rows eliminated in an intersection operation, but this approach requires special logic for rows satisfying multiple individual clauses in queries with multiple clauses linked by both "and" and "or."

SQL Server locks individual index entries or rows in a non-unique primary storage structure such as a clustered index on HireYear. Thus, fetching a single row locks only the record identified by a pair of HireYear and HireNo plus a gap to the prior index entry or row. If the query returns multiple employees hired in the same year, each row requires its own lock and lock manager invocation, even if there are hundreds or thousands of them. Thus, key-range locking fails to take advantage of a time-ordered storage structure for time-focused data warehouse queries.

Orthogonal key-range locking locks substantially less – only the index entry or row but not an adjacent gap to the neighboring index entry or row. Note that this gap can be substantial and locking it can be surprising if a query fetches the first hire of a year, i.e., a row with HireNo 1, because it affects the preceding year. This example illustrates clearly the advantage of orthogonality and the difference between orthogonal key-range locking and traditional key-range locking.

Orthogonal key-value locking offers two relevant granularities of locking. First, it can lock a distinct key value and thus all employees hired in 2015 – this might be suitable for queries fetching many rows or

as a lock escalation option for queries with large result sets. Second, it can lock merely a partition among the employees hired in 2015 – this seems more suitable for selective queries and as a starting policy that permits subsequent lock escalation when warranted. In fact, simply fetching multiple rows with the same distinct key value will automatically achieve the effect of lock escalation when all partitions become locked. The number of fetched rows required for this effect depends on the number of partitions selected for the instances of the key value.

Index entries and gaps	entry	gap	entries			gap	entry
	(2014,1)	2014,>1	(2015,1)	(2015,2)	(2015,3)	2015,>3	(2016,1)
ARIES/KVL							
ARIES/IM							
KRL							
Orth. krl							
Orth. kvl							

Figure 29. Lock scopes when fetching from a non-unique primary index.

Figure 29 illustrates the preceding discussion. Due to the dense sequence of calendar years and thus of HireYear values, the difference between ARIES/KVL and orthogonal key-value locking is not immediately apparent. The same is true for the differences between ARIES/IM, key-range locking, and orthogonal key-range locking. What is immediately apparent, however, is that orthogonal key-value locking offers two granularities of locking and thus the advantages both of ARIES/KVL, i.e., few locks and few lock manager invocations, and of key-range locking, i.e., precise locks and high concurrency.

4.10 *Summary of the case studies*

The preceding cases cover all principal types of index accesses and compare their locking requirements. The comparison criteria include both lock scope, i.e., locked database contents beyond the truly required scope, and overhead, i.e., the number of locks and thus of lock manager invocations.

In all comparisons, orthogonal key-value locking fares very well. In queries, it is better than the prior techniques, including orthogonal key-range locking introduced only a few years earlier. The partitioning technique of Section 3.4 increases the advantage further. In updates including deletion and insertion via ghost status, orthogonal key-range locking is best and orthogonal key-value locking performs equally well except in the case of hash collisions due to an insufficient number of partitions.

While the case studies above focus on selection predicates, both empty queries and successful index searches also occur in join operations, in particular during index nested loops joins. Index nested loops join can be superior to merge join and hash join not only in terms of I/O and CPU effort but also in terms of concurrency control. Since each inner loop of an index nested loops join acquires locks as illustrated in Figure 22 to Figure 26 (and in Figure 27 for the case of non-equality join predicates), join operations multiply the detrimental effects of excessive lock counts and of excessive lock scopes.

5 Future opportunities

Possible future research might apply orthogonality and lock partitioning in other contexts. The following is an initial outline for some of these directions.

5.1 *Column-level locking*

The essence of orthogonal key-value locking is a single distinct key value with multiple locks and lock modes, plus partitioning bookmarks and gaps. Section 3.6 extends partitioning of bookmarks to horizontal partitioning of index entries associated with a distinct key prefix. An alternative design considers vertical partitioning, i.e., partitioning the set of columns in a table or an index. Note that there are two independent choices between vertical and horizontal partitioning, one for the representation or data structure and one for locks and concurrency control.

In this approach, each column in a table or index is assigned to one of the partitions and thus covered by one of the lock modes in a lock request. One of the lock modes may be reserved for the entire set of columns. Thus, fewer partitions may be sufficient even for a table with tens or hundreds of columns.

Lock requests may be for individual index entries or for distinct key values. In the former case, each lock request covers columns in a single row or record. In the latter case, each lock request covers the same columns or fields in a set of index entries or records, namely those with the same distinct key value.

With a large set of choices, the application and expected access patterns must guide the final design.

5.2 *Orthogonal row locking*

While SQL Server key-range locking locks individual index entries in individual indexes, both ARIES methods aim to reduce the number of locks required in a query or an update. ARIES/KVL optimizes queries, in particular retrieval from non-unique indexes, whereas ARIES/IM optimizes single-row updates and single-row index-to-index navigation. Both approaches have their merits. Orthogonal key-value locking can be viewed as a refinement of ARIES/KVL and it seems worthwhile to consider an equivalent refinement of ARIES/IM.

Figure 30 summarizes locking techniques by their granularity of locking. The rows in Figure 30 indicate the granularity of locking. For example, in index-specific ARIES/IM, a lock covers a single row's representation in a single index. The granularity of locking also implies the search key in the lock manager's hash table. For example, in ARIES/IM, locks are identified by the record identifier in the table's primary storage structure. The columns in Figure 30 indicate the scope of each lock. For example, all ARIES techniques lock a gap between index entries together with an adjacent index entry, whereas key-range locking was the first to separate gap and key, albeit not fully and orthogonally.

Granularity of locking	Example lock identifier	Traditional locking: key and gap as unit	Some separation	Orthogonal locking
Individual index entries	"Jerry, 3"	Index-specific ARIES/IM	Key-range locking	Section 2.5
Distinct key values	"Jerry"	ARIES/KVL		Section 3
Logical rows	"3"	ARIES/IM		Section 5.2

Figure 30. Locking techniques by granularity of locking.

The foundation of ARIES/IM is to lock logical rows of a table: by locking a bookmark, a transaction locks a record in the table's primary storage structure, one index entry in each secondary index, plus a gap (to the next lower index entry) in each secondary index. While good for single-row updates, the design suffers from too small a scope and too many locks in non-unique secondary indexes (compared to ARIES/KVL, where each lock covers a distinct key value with all its instances) and from too large a scope in particular in index-only retrieval. For example, for phantom protection, one might want to lock a gap between two index entries in one secondary index, but the only lock scope available also locks the index entry above the gap as well as the logical row it pertains to, plus index entries in all other indexes as well as gaps between those index entries.

Improvements in orthogonal key-value locking over ARIES/KVL include (i) the separation of key value and gap, (ii) partitioning within the list of bookmarks, (iii) partitioning of non-existing key values in a gap between existing distinct key values, and (iv) a hierarchy of lock scopes in order to reduce the number of lock manager invocations. These ideas could also improve ARIES/IM, one improvement at a time, yielding what might be called "orthogonal row locking." As in orthogonal key-value locking, the core idea is to keep the lockable resources and thus the number of lock manager invocations but to introduce finer granularities of locking, possibly encoding those using combined lock modes.

First, one could split a lock into two such that one lock applies only to the table's primary storage structure and the second one applies to any and all secondary indexes. This would allow one transaction to search an index or even multiple indexes while another transaction updates non-indexed columns in the table.

Second, one could separate the index entries from their adjacent gaps as well as the logical row from its adjacent gap. In other words, a lock on a logical row would have two modes, one for the row one for the adjacent gap, and the lock on all index entries also would also have two modes, one for the index entries and another one for the gaps in all the indexes. This would permit, for example, one transaction locking a row merely for phantom protection in one of the indexes while another transaction modifies the row including some of its index entries. This update may modify non-key fields including the ghost bits, thus

logically deleting the row and all its index entries. Combined lock modes, constructed in the spirit of Figure 5 with lock compatibility derived as in Figure 6, reduce the number of lock manager invocations.

Third, one could apply partitioning to the index entries belonging to the same logical row, possibly including the record in the table's primary storage structure. For that, a lock for a logical row is augmented with a list of indexes in which index entries are to be locked. If the set of possible indexes is very large, one could limit the size of this list by partitioning the set of possible indexes such that locks cover a partition rather than a specific index. Thus, one transaction might update the row in the primary storage structure as well as some affected indexes while another transaction searches other indexes of the same table, including index entries for the row being updated (in indexes not affected by the update).

Fourth, one could partition the indexes and in particular their gaps, i.e., a lock request for a logical row would specify the indexes in which gaps ought to be locked. This would permit, for example, phantom protection with a locked gap in a single index only. With hash collisions, it would lock gaps in multiple indexes.

Fifth, one could define a hierarchy of lock scopes – the finest granularity of locking would be the set of partitions, the coarsest granularity of locking would be the row and all its index entries and all gaps, i.e., all partitions. Combined lock modes, constructed in the spirit of Figure 5 and Figure 19, enable lock acquisition with the scope and speed of the original ARIES/IM design as well as locks with much reduced scope with little additional overhead. For example, for phantom protection, a single lock request for a logical row can lock merely the gap between two index entries within a single index.

Technique	Modes per lock
Split primary storage structure vs secondary indexes	2
Split row vs gap in each secondary index	3
Split index entry vs gap in the primary storage structure	4
Partitioning the set of index entries	k
Partitioning the set of gaps	k
Hierarchical locking	~3+2k

Figure 31. Lock counts in orthogonal row locking.

Figure 31 summarizes these ideas applying the techniques of orthogonal key-value locking to locking logical rows, i.e., to refining ARIES/IM in ways similar to orthogonal key-value locking refining ARIES/KVL. For each of the extensions listed above, Figure 31 indicates the number of lock modes required in each lock manager invocation.

Index entries and gaps	Index entry in primary index (on EmpNo)	Gap in EmpNo	Index entry in secondary index on FirstName	Gap in FirstName	Index entry in secondary index on PostalCode	Gap in PostalCode
ARIES/IM						
Index-specific ARIES/IM						
Orthogonal row locking						

Figure 32. Lock scopes in alternative row locking techniques.

Figure 32 illustrates possible lock scopes in alternative locking techniques applied to the database table of Figure 20 and its indexes. Locking focuses on logical rows and row identifiers rather than

individual indexes and their key values. ARIES/IM "data-only locking" has only a single granularity of locking. Index-specific ARIES/IM locks an entry and an adjacent gap one index at a time. The three rows in Figure 32 for ARIES/IM index-specific locking represent three different locks and three separate lock manager invocations. Orthogonal row locking permits locking records without any gaps, whether in the primary index, a single secondary index, or multiple indexes; an open interval (gap only) or a half-open interval (index entry and gap) in just one index; any combination of the above; or all of the above thus mirroring the lock scope of ARIES/IM. The six rows in Figure 32 for orthogonal row locking are examples of what a single lock manager invocation can request.

Inasmuch as orthogonal key-value locking is designed for queries and data analysis, e.g., index intersection for efficient evaluation of conjunction predicates, orthogonal row locking is designed for single-row updates and for look-up queries with single-row index-to-index navigation. Thus, the choice between these two alternative methods for precise yet efficient database concurrency control depends on the application context. The two methods can co-exist within a single server, a single database, or even a single table, e.g., if usage patterns change over time such as business hours versus night hours. Using both methods for a single table at the same time introduces complexity and invites solutions similar to general multi-granularity locking when compared to strict hierarchical locking [GLP 75].

5.3 *Orthogonal locking in tables and indexes*

Systems with page-, row-, or key-locking typically employ hierarchical locking with an intention lock for each table or index followed by another lock request, e.g., for key-range locking or key-value locking. A new alternative using only table- or index-level locking reduces lock acquisitions per index by half, i.e., from two to one.

If the entire database or at least the working set fit in memory and if interactive transactions with multiple messages between database process and application process are rare, then there is little advantage in using more software threads than hardware threads (CPU cores). With only dozens of hardware threads, the number of partitions in a locking scheme and in each lock request may exceed the desired degree of concurrency. In such a case, it may be sufficient to lock only tables or indexes rather than pages, key values, or index entries.

For example, if the hardware in a system, e.g., a single node in a cluster, supports 16 hardware threads, then 32 software threads may be sufficient to keep all hardware busy all the time. A partitioning scheme with 128 partitions may be sufficient to avoid most false conflicts due to hash collisions, independent of the number of rows in the table. Thus, table- or index-level locking may be sufficient, due to partitioning and a lock mode per partition.

Hash partitioning works well, but only for equality queries. If query predicates specify key ranges, the outlined design often requires locks on entire tables or indexes. Thus, a design based on hash partitioning is only of limited use and requires a complementary locking technique that is effective and efficient for key ranges.

This suggests a combination of hash partitioning and range partitioning. This combination employs m hash partitions and n range partitions for each file, table, or index. Each data access requires absolute locks (e.g., S or X) on one kind of partition and intention locks (e.g., IS or IX) on the other. For example, a range query may take absolute locks on the appropriate set of range partitions and intention locks on all hash partitions.

	Hash partitions	Range partitions
Equality query	One S	One IS
Range query	All IS	Some S
Non-key update, Insertion, Deletion	One X	One IX
	One IX	One X
Key update	Two X	Two IX
Orthogonal key-value locking	Some IS or IX	Some IS or IX

Figure 33. Locking in combined hash and range partitions.

Figure 33 lists access patterns and their locking requirements. An equality query benefits from the precision of hash partitioning whereas a range query benefits from the efficiency of locking key ranges. A non-key update takes exclusive locks with the same scope as an equality query. Insertion and deletion via ghost records are non-key updates. A key update requires one deletion and one insertion, each with the appropriate locks. The last line of Figure 33 anticipates a consideration below.

Relying on range partitioning might be sub-optimal if accesses are not balanced across key ranges. If the storage structure is an ordered hierarchical index such as a b-tree, the key range boundaries may be selected from the separator keys in a structure's root page. Updates of the root page should affect these key range boundaries only while it is entirely unlocked.

For even higher concurrency, a fine granularity of locking may be desired that cannot be achieved by hash and range partitioning with the table or index lock. For example, by acquiring IS locks on all hash partitions and all range partitions, a single range query plus a single equality query may prevent an update on even a single row.

Such applications and database accesses may require locking individual data items in the index leaves. As indicated in the last line of Figure 33, those index accesses must acquire intention locks on both kinds of partitions together with the intention lock on the table or index as a whole, whereupon they may use record-level locking within index leaves, e.g., orthogonal key-value locking. By doing so, they permit other applications and accesses to rely entirely on table or index locks with partitions. Actual conflicts, if and when they exist, can be found reliably and efficiently. This design adds to the theory of multi-granularity locking (beyond hierarchical locking) [GLP 75, GLP 76] multiple lock modes in a single lock manager invocation as well as hash and range partitioning.

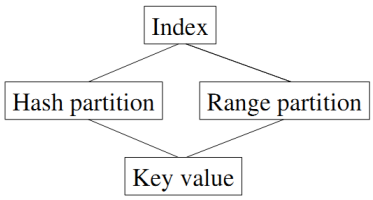

Figure 34. A hierarchy of lock scopes.

Figure 34 illustrates this hierarchy of lock scopes. Index and key value, are traditional granularities of locking. The intermediate lock scopes, i.e., hash partitions and range partitions, are the new level. Folding them into the index lock ensures in the worst case the same efficiency in lock acquisition and release as the traditional design, yet with the opportunity of reasonable concurrency with a single lock only on the index and its partitions.

5.4 *Orthogonal schema locking*

Another possible research direction applies combined lock modes (in the spirit of Figure 5) and partitioning to schemas and catalogs, yielding what might be called "orthogonal schema locking." Most database systems support an interpreted data definition language (DDL) and thus require type information stored as data in catalogs rather than compiled into all programs and scripts written in the system's data manipulation language (DML). Microsoft SQL Server, for example, has special lock modes for this schema information. The "schema stability" lock is used, among other times, during query optimization – without any locks on data such indexes, partitions, pages, records, or keys. The "schema modification" lock is used, for example, while dropping an index – ensuring exclusive use of both schema and all data structures. All other lock modes permissible for a table (e.g., S, X, IS, IX) imply a schema stability lock. They are implied by a schema modification lock and incompatible with another transaction's schema modification lock. Other database management products with dynamic schema modifications employ similar systems for concurrency control on metadata.

In order to simplify schema management and schema locking in databases, it seems possible and desirable to avoid these special lock modes, to map the concurrency control requirements to combined locks, and to avoid any increase in lock manager invocations. In fact, it seems possible and useful to divide either the schema or the data and to achieve a finer granularity of locking using combined locks.

In a design with combined lock modes, query optimization might hold an SN lock (pronounced 'schema shared, data free' here), whereas dropping an index requires an XX lock (pronounced 'schema

exclusive, data exclusive' here). Intention lock modes must be supported, at least for the data (somewhat similarly to Section 2.5). Thus, achieving parity with the existing design seems quite possible.

A further optimization divides the data lock into individual data structures, e.g., a table's set of indexes. In other words, rather than locking a table and all its data structures, a single lock request (with a list of lock modes) may lock individual indexes. Typically, a table has only a few indexes, but it can have dozens. With only a few indexes, each index may have its own entry in a list of lock modes provided with each lock request for a table. With a multitude of indexes, the set of indexes might be partitioned using a hash function to map index identifiers to positions in the lock list. The most immediate benefit may be to reduce the number of lock requests: where a small transaction today may lock first a table in IX mode and then two indexes each in IX mode, in the future a transaction may lock the table in IX mode and, within the same lock manager request, acquire IX locks on two indexes (or actually, on two partitions within the set of indexes, with each partition probably containing only a single index).

Another optimization divides the schema lock into individual components of a schema, e.g., column set or individual columns, index set or individual indexes, integrity constraints, partitions (of table or indexes), histograms, etc. Sets of schema entries with individual locks may improve concurrency among changes in the data definition or, probably more importantly, concurrency of schema changes (DDL) with query and transaction processing (DML). For example, one transaction's query execution may join two secondary indexes while another transaction adds a new index. – Thus, online index creation no longer requires a "schema modify" lock for the entire table when it first registers a new index and when it releases the completed index for unrestricted use. The ideal outcome of this research direction would improve the online behavior of DDL with minimal impact on active users and applications.

Without doubt, more research is required to validate these possible research directions, to work out details of alternative designs and their limitations, and to assess with a prototype whether simplicity, robustness, performance, scalability, concurrency, or online behavior can be improved in future database systems.

5.5 *Summary of future opportunities*

Multiple research directions, albeit outlined above only vaguely, may improve locking precision and locking overheads using a common technique. The ideas common to all are orthogonality and partitioning. Orthogonality separates locks, e.g., on existing and non-existing data. Partitioning maps possibly infinite sets, e.g., of index entries, to a fixed set of disjoint partitions. A single lock manager invocation can lock any subset of partitions by indicating a lock mode for each partition as well as for the entire set as a whole.

6 Conclusions

In summary, sharing structured data is the principal purpose of databases and of database management software. Effective sharing requires explicit schema information including integrity constraints, security (at least authorization if not also authentication), logical and physical data independence including automatic mapping of queries and updates to storage structures and to execution plans, and atomic transactions including serializability and durability. Integrity constraints and concurrency control interact in ways easy to overlook. For example, query optimization may remove a semi-join from a query expression if the schema contains an equivalent foreign key integrity constraint – but only in serializable transaction isolation, because weak transaction isolation may permit concurrent transactions with temporary integrity constraint violations. More specifically, one transaction may read a foreign key value that it cannot find as primary key value due to a deletion by another transaction. With weak transaction isolation, the same effect can occur during navigation from a table's secondary index to the same table's primary storage structure. These and other forms of havoc cannot occur in serializable transaction isolation, i.e., full equivalence to serial execution. Thus, the focus of this research is on full isolation of serializable transactions, with particular attention on phantom protection and locking non-existing key values in gaps between existing key values.

A new technique, orthogonal key-value locking, combines design elements and advantages of (i) key-value locking in ARIES, i.e., a single lock for an equality query, of (ii) key-range locking, i.e., locking individual index entries for highly concurrent updates, and of (iii) orthogonal key-range locking, i.e., independent lock modes for key values and gaps between key values. In addition, orthogonal key-value locking enables phantom protection with better precision than these prior methods. The principal new

techniques are (i) partitioning (only with respect to concurrency control and locking) the set of bookmarks or rows associated with a non-unique index key or key prefix, (ii) partitioning the set of possible key values in a gap between adjacent existing key values, and (iii) specifying a lock mode for each partition. With these innovations, orthogonal key-value locking can lock none, some, or all of the row identifiers or rows associated with an existing key value plus none, some, or all of the non-existing key values in a gap between adjacent existing key values.

Partitioning a set of bookmarks is superior to key-range locking (locking individual index entries) because key-value locking requires only a single lock request per distinct key value and because the principal lock scope matches that of query predicates, in particular equality predicates as well as end points of range predicates. Partitioning the set of non-existing key values in a gap cuts the concurrency control footprint of phantom protection to a fraction compared to the footprint of prior methods. Thus, orthogonal key-value locking promises truly serializable transaction isolation without the traditional high cost of false conflicts among transactions.

A detailed case study compares prior and new locking methods for ordered indexes, e.g., b-trees. For both successful and unsuccessful (empty) queries, the case study demonstrates that orthogonal key-value locking is superior to all prior techniques. For updates, it effectively equals orthogonal key-range locking, which is superior to all prior techniques. Orthogonal key-value locking is sub-optimal 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 false lock conflicts.

A prototype (see the Appendix) validates the anticipated simplicity of an 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 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 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 of (orthogonal) key-range locking. A read-only experiment shows an increase in retrieval throughput by 4.8 times and a mixed read-write workload shows a transaction throughput 1.7–2.1 times better than with prior locking techniques. Thus, the experiments confirm our expectations from the case studies. We hope that this study will affect design and implementation of databases, key-value stores, information retrieval systems, and (modern, b-tree-based) file systems alike.

Even if research into b-tree locking may seem a quaint relic of bygone decades (or even of a bygone millennium), there continue to exist good reasons to pursue it. B-trees and their variants have been ubiquitous for a long time and seem hard to displace, in particular if carefully constructed keys enable partitioning within an index (and thus read- and write-optimized operations), versioning (e.g., for multi-version concurrency control), master-detail clustering (for application objects and for graph data), spatial indexes (including moving objects), compression (of keys as well as child pointers, row identifiers, and other index contents), and more [G 11]. Designs and implementations of locking must ensure optimal lock modes, e.g., including update locks and intention locks [K 83] as well as increment locks [GZ 04], minimal lock duration, e.g., by early lock release or controlled lock violation [GLK 13], and the optimal granularity of locking, e.g., by orthogonal key-value locking. Recent work demonstrates that locking is categorically superior to optimistic concurrency control [G 19a] due to fewer conflicts, more concurrency, and fewer rollbacks.

References

[BHE 11] Mihaela A. Bornea, Orion Hodson, Sameh Elnikety, Alan Fekete: One-copy serializability with snapshot isolation under the hood. ICDE 2011: 625-636.

[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).

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

[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 1982: 184-191.

[CHK 01] Sang Kyun Cha, Sangyong Hwang, Kihong Kim, Keunjoo Kwon: Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. VLDB 2001: 181-190.

[CRF 09] Michael J. Cahill, Uwe Röhm, Alan David Fekete: Serializable isolation for snapshot databases. ACM TODS 34(4) (2009).

[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).

[G 11] Goetz Graefe: Modern b-tree techniques. Foundations and Trends in Databases 3(4): 203-402 (2011).

[G 19] Goetz Graefe: On transactional concurrency control. Synthesis Lectures on Data Management, Morgan & Claypool Publishers 2019.

[G 19a] Goetz Graefe: Deferred lock enforcement. In [G 19], 327-365. The author can provide an extended version.

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

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

[GLP 75] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, Irving L. Traiger: Granularity of locks in a large shared data base. VLDB 1975: 428-451.

[GLP 76] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, Irving L. Traiger: Granularity of locks and degrees of consistency in a shared data base. IFIP Working Conf on Modelling in Data Base Mgmt Systems 1976: 365-394.

[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 14] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi Kuno, Joseph Tucek, Mark Lillibridge, Alistair Veitch: In-memory performance for big data. PVLDB 8(1): 37-48 (2014).

[GZ 04] Goetz Graefe, Michael J. Zwilling: Transaction support for indexed views. ACM SIGMOD 2004: 323-334.

[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.

[JHF 14] Hyungsoo Jung, Hyuck Han, Alan David Fekete, Gernot Heiser, Heon Young Yeom: A scalable lock manager for multicores. ACM TODS 29:1-29:29 (2014).

[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).

[K 83] Henry F. Korth: Locking primitives in a database system. J. ACM 30(1): 55-79 (1983).

[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.

[LFW 12] David B. Lomet, Alan Fekete, Rui Wang, Peter Ward: Multi-version concurrency via timestamp range conflict management. ICDE 2012: 714-725.
[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.
[PG 12] Dan R. K. Ports, Kevin Grittner: Serializable snapshot isolation in PostgreSQL. PVLDB 5(12): 1850-1861 (2012).
[RTA 12] Kun Ren, Alexander Thomson, Daniel J. Abadi: Lightweight locking for main memory database systems. PVLDB 6(2): 145-156 (2012).
[TPC] http://www.tpc.org/tpcc/results/tpcc_perf_results.asp.

7 Appendix: 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 7.1).
- The concurrency with orthogonal key-value locks is as high as the concurrency with state-of-the-art prior lock modes (Section 7.2).
- Orthogonal key-value locking provides both of the above benefits at the same time, which no prior locking techniques could do (Section 7.3).

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 observed that the following optimizations were highly effective:

- Flush-pipelines and consolidation arrays [JPS 10] speed up logging.
- Read-after-write lock management [JHF 13, JHF 14] 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 14] 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. 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. Neither modification required rewriting other modules.

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 (2 standard deviations).

7.1 *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 LAST_NAME
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].

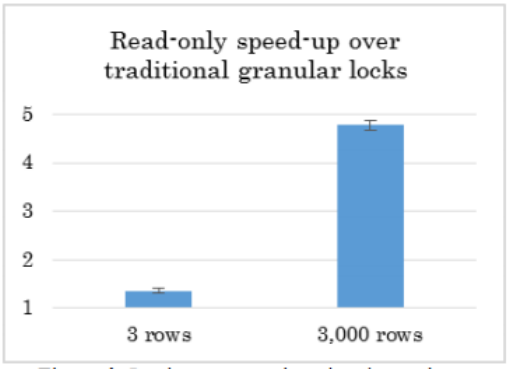

Figure 35. Locking costs and retrieval speed-up.

Figure 35 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 for 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 search 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 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.

7.2 *Enabled 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).

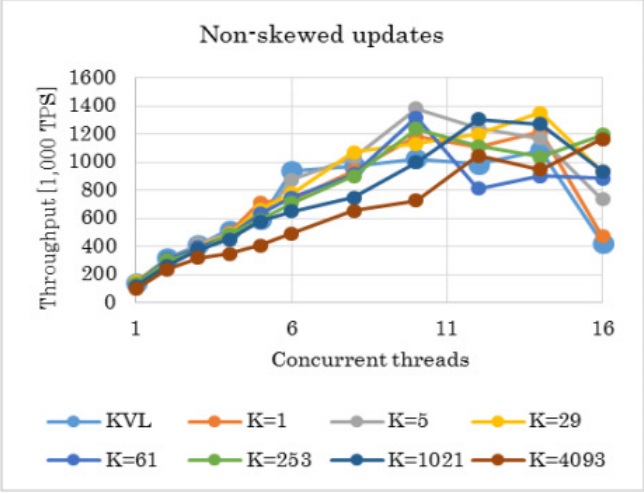

Figure 36. Transaction throughput with little contention.

Figure 36 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 multiple 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.

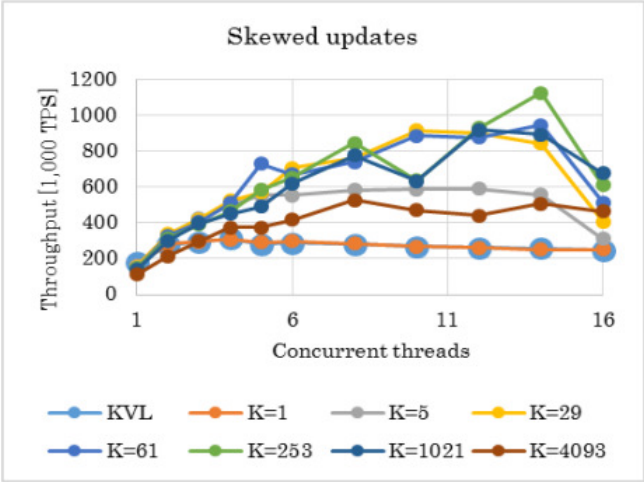

Figure 37. Transaction throughput with heavy contention.

Figure 37 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 as few as 3 threads. With larger values of 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.

One key observation is the high overhead of extremely large values partition counts, 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.

7.3 *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] (marked “KVL" in Figure 38) locks unique values of the user-defined index key including an adjacent gap. Our implementation uses a lock on (WID) in this experiment.

- Key-range locking (marked "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. We fix the number of threads to 14 and the number of partitions in orthogonal key-value locking to k=253.

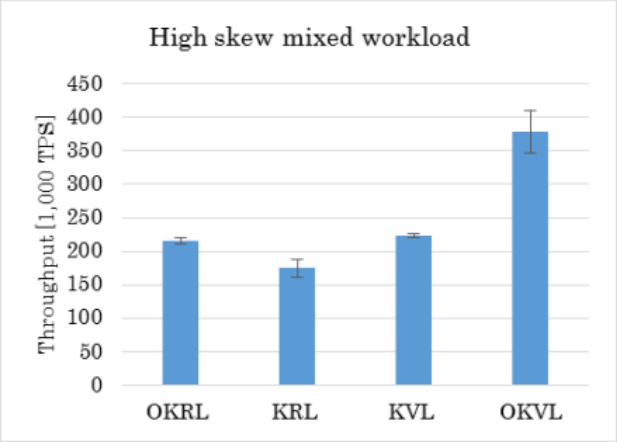

Figure 38. Locking and throughput of read-write transactions.

Figure 38 shows the results. As this workload is highly skewed, ARIES/KVL (coarse-grained lock) fails to enable concurrency among updates. Its degree of parallelism was very low here, even if other experiments (Figure 35) 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 can process more concurrent transactions but have to 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 throughputs. 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.

7.4 *Summary of experimental evaluation*

In summary, the implementation demonstrate the feasibility and the simplicity of the method – definitely at least on par with earlier techniques. The execution time measurements demonstrate that the cost of locking remains measurable or even substantial, in particular after better ones have replaced unnecessarily expensive techniques, e.g., pointer swizzling in the buffer pool.

In particular, the first experiment demonstrates how key-value locking, e.g., orthogonal key-value locking, reduces the costs and overheads of locking compared to key-range locking, i.e., locking each individual index entry.

The second experiment demonstrates that, as expected, orthogonal key-value locking with very few partitions, e.g., a single one, performs just like traditional key-value locking, and that orthogonal key-value locking with very many partitions, e.g., hundreds, introduces overheads first reducing and eventually negating the performance benefits. More informatively, the experiment demonstrates that under contention, orthogonal key-value locking out-performs traditional key-value locking by a substantial margin.

The third experiments demonstrates that the alternative locking strategies affect not only locking costs and concurrency but also the overall transaction throughput – the experiment found almost a factor of two in transaction throughput between the worst and the best locking strategies.

Appendix: auxiliary diagrams

Partitions of possible key values			
84, 88	81, 85, 89	82, 86	83, 87

Figure 39. Partitions of possible key values between 80 and 90.

Key	Gap				Key
80	84, 88	81, 85, 89	82, 86	83, 87	90

Figure 40. Partitions of non-existing key values between existing key values 80 and 90.

Figure 39 and Figure 40 show a possible partitioning of non-existing key values in an index.

HireYear	Count	**EmpNo**s
2014	1	1
2015	3	3, 5, 6
2016	1	9

Figure 41. Another secondary index.

PostalCode	Count	**EmpNo**s
10032	1	1
37745	1	6
46045	1	3
53704	1	5
60654	1	9

Figure 42. Yet another secondary index.

Figure 41 and Figure 42 show two additional indexes for the example table.

FirstName	**EmpNo**	**PostalCode**
Gary	1	10032
Jerry	3	46045
Jerry	6	37745
Mary	5	53704
Terry	9	60654

Figure 43. Secondary index with an additional column.

FirstName	Count	**EmpNo**+**PostalCode** pairs
Gary	1	(1, 10032)
Jerry	2	(3, 46045), (6, 37745)
Mary	1	(5, 53704)
Terry	1	(9, 60654)

Figure 44. Secondary index with lists of pairs of values.

Figure 43 and Figure 44 show the same information. The difference in data or storage structure makes no difference on the mechanisms for lock scopes and transactional locking. Such indexes permit "index-only retrieval" during index-to-index navigation, e.g., from employee names to their postal areas.

Project	**EmpNo**	**Effort**
624	1	100%
624	3	50%
4711	3	50%
4711	5	100%
4096	6	100%
…	…	…

Figure 45. A many-to-many relationship table.

Project	Count	(**EmpNo**, **Effort**) pairs
624	2	(1, 100%), (3, 50%)
4096	1	(6, 100%)
4711	2	(3, 50%), (5, 100%)
…		…

Figure 46. An index on the relationship table.

EmpNo	Count	(**Project**, **Effort**) pairs
1	1	(624, 100%)
3	2	(624, 50%), (4711, 50%)
5	1	(4711, 100%)
6	1	(4096, 100%)
…	…	…

Figure 47. Another index on the relationship table.

Figure 45, Figure 46, and Figure 47 show a table for a many-to-many relationship and two indexes that permit navigating this relationship efficiently in both directions.

gap		entry	gap			entry		entry	gap			entry
<Gary	Gary, <1	(Gary, 1)	Gary, >1	>Gary, <Jerry	Jerry, <3	(Jerry, 3)	gap	(Jerry, 6)	Jerry, >6	>Jerry, <Mary	Mary, <5	(Mary, 5)

Figure 48. Dividing the key ranges between index entries.

Figure 48 shows (most of) the index entries of Figure 21; the index entry for key value 'Terry' is missing, as are the gaps around it. The index entries are stored in the database index; the gaps are not. Figure 48 illustrates how the gaps between index entries with different key values can be divided as shown into a range of further, non-existing key values and two ranges of existing key values but non-existing row identifiers.

Index entries and gaps → ↓ Techniques	entry (Gary, 1)	gap			entry (Jerry, 3)	gap	entry (Jerry, 6)	gap	
		Gary, >1	>Gary, <Jerry	Jerry, <3				Jerry, >6	>Jerry
Traditional b-tree locking									
Research goal									

Figure 49. Research goal compared to traditional lock scopes.

The research goal is to avoid false conflicts between transactions. Excessively coarse locks have contributed to giving locking and serializability a reputation for poor concurrency, poor scalability, and poor system performance.

Index entries and gaps	entry (Gary, 1)	gap			entry (Jerry, 3)	gap	entry (Jerry, 6)	gap			entry (Mary, 5)
		Gary, >1	>Gary, <Jerry	Jerry, <3				Jerry, >6	>Jerry, <Mary	Mary, <5	
A gap + a key value											

Figure 50. A lock scope in ARIES/KVL.

Index entries and gaps	gap			entry (2015, 3)	gap	entry (2015, 5)	gap	entry (2015, 6)	gap		
	2014, >1	>2014, <2015	2015, <3						2015, >6	>2015, <2016	2016, <9
A gap + a key value											

Figure 51. An ARIES/KVL lock in another secondary index.

A lock in ARIES/KVL covers a distinct key value plus the gap to the preceding key value in the index.

Index entries and gaps	entry (Gary, 1)	gap			entry (Jerry, 3)	gap	entry (Jerry, 6)	gap			entry (Mary, 5)
		Gary, >1	>Gary, <Jerry	Jerry, <3				Jerry, >6	>Jerry, <Mary	Mary, <5	
A logical row											

Figure 52. A lock scope in ARIES/IM.

A lock in ARIES/IM covers a logical row, all its index entries (including the primary storage structure), plus in each index the gap to the preceding index entry. Figure 52 shows the scope of a lock on row identifier '3' – note that this lock also affects all other indexes of the same database table.

Index entries and gaps	entry (37745, 6)	gap			entry (46045, 3)	gap			entry (53704, 5)
		37745, >6	>37745, <46045	46045, <3		46045, >6	>46045, <53704	53704, <5	
A logical row									

Figure 53. Implied lock in another secondary index.

Index entries and gaps	entry (2014, 1)	gap			entry (2015, 3)	gap	entry (2015, 5)	gap
		2014, >1	>2014, <2015	2015, <3				
A logical row								

Figure 54. Implied lock in yet another secondary index.

Index entries and gaps	entry (Jerry, 6)	gap			entry (Mary, 5)	gap			entry (Terry, 9)
		Jerry, >6	>Jerry, <Mary	Mary, <5		Mary, >5	>Mary <Terry	Terry, <9	
ARIES/KVL			S on "Mary"						
ARIES/IM		S on "5"				S on "9"			
KRL		S on "Mary, 5"				S on "Terry, 9"			
Orth. krl		NS on "Jerry, 6"			S on "Mary, 5"				
Orth. kvl				S+N on "Mary"					

Figure 55. Locking unique key value 'Mary' including phantom protection.

		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 56. Summary of locking in ARIES/KVL (original column sequence).

Figure 56 equals Figure 3 except that it shows the original sequence of columns [M 90].

	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 57. Summary of locking in ARIES/IM (original column sequence).

Figure 57 equals Figure 4 except that it shows the original sequence of columns [ML 92].

	Prior key	Current key	Next key
Fetch non-existing	N+NNSN		
Fetch existing		NSNN+N	
Fetch next		…+S	SNNN+N
Update		NNNX+N	
Insert (via ghost)			
Delete (via ghost)			
Erase (a ghost)			
Create (a ghost)			

	Prior key	Current key	Next key	Comments
Fetch non-existing	N+NNSN			also see Section 3.4
Fetch existing		NSNN+N		also see Section 3.3
Fetch next		…+S	SNNN+N	lock gap and new result
Update		NNNX+N		update of non-key fields
Insert (via ghost)				from host to valid
Delete (via ghost)				from valid to ghost
Erase (a ghost)				erase ghost under latch, with no conflicting lock
Create (a ghost)				create ghost under latch, with no conflicting lock (phantom protection); also see Section 3.4

Figure 58. Summary of orthogonal key-value locking.

Figure 58 summarizes the locks required in orthogonal key-value locking. All locks are held until transaction commit; latches are released after the critical section. Partitioning of lists and of gaps is assumed; for example, lock mode “N+NNSN” means no lock on a key value and a shared lock on the appropriate partition within the gap. – A lock on a prior key value is required only for phantom protection; a lock on the next key only in a scan when that next key is the new scan result. Insertion and deletion are precisely like non-key updates because they are, in fact, non-key updates on ghost bits. Creation and removal of ghosts require no locks, only latches, plus checks in the lock manager for conflicting locks.

	NL	SCH-S	SCH-M	S	U	X	IS	IU	IX	SIU	SIX	UIX	BU	RS-S	RS-U	RI-N	RI-S	RI-U	RI-X	RX-S	RX-U	RX-X
NL	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
SCH-S	N	N	C	N	N	N	N	N	N	N	N	N	N	I	I	I	I	I	I	I	I	I
SCH-M	N	C	C	C	C	C	C	C	C	C	C	C	C	I	I	I	I	I	I	I	I	I
S	N	N	C	N	N	C	N	N	C	N	C	C	C	N	N	N	N	N	C	N	N	C
U	N	N	C	N	C	C	N	C	C	C	C	C	C	N	C	N	N	C	C	N	C	C
X	N	N	C	C	C	C	C	C	C	C	C	C	C	C	C	N	C	C	C	C	C	C
IS	N	N	C	N	N	C	N	N	N	N	N	N	C	I	I	I	I	I	I	I	I	I
IU	N	N	C	N	C	C	N	N	N	N	N	C	C	I	I	I	I	I	I	I	I	I
IX	N	N	C	C	C	C	N	N	N	C	C	C	C	I	I	I	I	I	I	I	I	I
SIU	N	N	C	N	C	C	N	N	C	N	C	C	C	I	I	I	I	I	I	I	I	I
SIX	N	N	C	C	C	C	N	N	C	C	C	C	C	I	I	I	I	I	I	I	I	I
UIX	N	N	C	C	C	C	N	C	C	C	C	C	C	I	I	I	I	I	I	I	I	I
BU	N	N	C	C	C	C	C	C	C	C	C	C	N	I	I	I	I	I	I	I	I	I
RS-S	N	I	I	N	N	C	I	I	I	I	I	I	I	N	N	C	C	C	C	C	C	C
RS-U	N	I	I	N	C	C	I	I	I	I	I	I	I	N	C	C	C	C	C	C	C	C
RI-N	N	I	I	N	N	N	I	I	I	I	I	I	I	C	C	N	N	N	N	C	C	C
RI-S	N	I	I	N	N	C	I	I	I	I	I	I	I	C	C	N	N	N	C	C	C	C
RI-U	N	I	I	N	C	C	I	I	I	I	I	I	I	C	C	N	N	C	C	C	C	C
RI-X	N	I	I	C	C	C	I	I	I	I	I	I	I	C	C	N	C	C	C	C	C	C
RX-S	N	I	I	N	N	C	I	I	I	I	I	I	I	C	C	C	C	C	C	C	C	C
RX-U	N	I	I	N	C	C	I	I	I	I	I	I	I	C	C	C	C	C	C	C	C	C
RX-X	N	I	I	C	C	C	I	I	I	I	I	I	I	C	C	C	C	C	C	C	C	C

Key

N	No Conflict	SIU	Share with Intent Update
I	Illegal	SIX	Shared with Intent Exclusive
C	Conflict	UIX	Update with Intent Exclusive
		BU	Bulk Update
NL	No Lock	RS-S	Shared Range-Shared
SCH-S	Schema Stability Locks	RS-U	Shared Range-Update
SCH-M	Schema Modification Locks	RI-N	Insert Range-Null
S	Shared	RI-S	Insert Range-Shared
U	Update	RI-U	Insert Range-Update
X	Exclusive	RI-X	Insert Range-Exclusive
IS	Intent Shared	RX-S	Exclusive Range-Shared
IU	Intent Update	RX-U	Exclusive Range-Update
IX	Intent Exclusive	RX-X	Exclusive Range-Exclusive

Figure 59. SQL Server lock compatibility.

Figure 59, copied from Microsoft's online documentation, lists all lock modes employed in SQL Server including schema locks, traditional locks, intention locks, bulk update locks, and special lock modes for keys and key ranges in b-tree indexes. The compatibility matrix is large and complex; it requires careful study for full understanding and verification.

Requested → ↓ Held	S	X	D
S	ok		
X			
D			ok

Figure 60. Lock compatibility with increment/decrement locks.

Figure 60 shows the compatibility of shared ("S") and exclusive ("X") locks with "D" locks that permit incrementing or decrementing a count or aggregate. These locks are particularly useful in materialized and indexed views defined with a "group by" query. An increment/decrement ("D") lock does not permit reading the current value or setting a specific value (except during commit processing). If a transaction needs to increment a value as well as read it, it must acquire an exclusive lock.

Requested → ↓ Held	S	X	D	IS	IX	ID
S	ok			ok		
X						
D			ok			ok
IS	ok			ok		
IX						
ID			ok			

Figure 61. Lock compatibility with increment/decrement intention locks.

Figure 61 shows the locks of Figure 60 plus appropriate intention locks. The top-left quarter of Figure 61 equals Figure 60; the bottom-right quarter indicates that intention locks are always compatible because actual conflicts will be found at a finer granularity of locking; and the other two quarters are copies of the top-left quarter.

Requested → ↓ Held	S	X as R	X as P	X	D as R	D as P	D
S	√	√		–	√		–
X as R	√	–			–		
X as P	–						
X							
D as R	√	–			√		√
D as P	–						–
D							

Figure 62. Lock compatibility including increment/decrement locks and their weak modes.

Figure 62 summarizes lock compatibility for increment/decrement and exclusive locks. Their conflicts with shared locks mirror the conflicts of exclusive locks with shared locks (see the row and the column labeled S). Their conflicts with exclusive locks mirror the conflicts of exclusive locks with other exclusive locks (see the three large blocks marked as conflicts). Conflicts among increment/decrement locks are limited to the commit point (see the shaded area), which makes them much more suitable than exclusive locks for incremental updates of summary rows in materialized "group by" views and their indexes.