Revisiting optimistic and pessimistic concurrency control

Goetz Graefe

Hewlett Packard Labs
HPE-2016-47

Keyword(s):
Database concurrency control

Abstract:
Optimistic concurrency control relies on end-of-transaction validation rather than lock acquisition prior to data accesses. Optimistic concurrency control is popular and various systems employ it based on a conviction that it increases concurrency, performance, and scalability. In contrast, we have concluded that optimistic concurrency control permits more concurrency than pessimistic concurrency control only if it fails to detect some actual conflicts or if a particular implementation of locking detects false conflicts. An example of the former is a weak transaction isolation level, e.g., "repeatable read" instead of "re-peatable count," i.e., serializability. An example of the latter is an unnecessarily coarse granularity of locking, e.g., traditional key-value locking. Another example of the latter is unnecessarily long lock retention, e.g., while writing a commit log record to stable storage in order to ensure a transaction’s durability. A fair comparison of alternative techniques requires equal criteria (e.g., serializability) and equal opportunities (e.g., in terms of granularity of concurrency control, access modes, etc.). In such a comparison, the crucial differences are the timing of conflict detection and the options for conflict resolution. For example, if a conflict between two transactions is found after one of them is already committed, failing the other transaction seems the only recourse. With these issues properly addressed and exploited, pessimistic concurrency control will always match or out-perform optimistic concurrency control. For the time being, this analysis focuses on single-version storage and ignores distributed transactions.
Revisiting optimistic and pessimistic concurrency control

Goetz Graefe
Hewlett Packard Labs

Abstract

Optimistic concurrency control relies on end-of-transaction validation rather than lock acquisition prior to data accesses. Optimistic concurrency control is popular and various systems employ it based on a conviction that it increases concurrency, performance, and scalability.

In contrast, we have concluded that optimistic concurrency control permits more concurrency than pessimistic concurrency control only if it fails to detect some actual conflicts or if a particular implementation of locking detects false conflicts. An example of the former is a weak transaction isolation level, e.g., “repeatable read” instead of “repeatable count,” i.e., serializability. An example of the latter is an unnecessarily coarse granularity of locking, e.g., traditional key-value locking. Another example of the latter is unnecessarily long lock retention, e.g., while writing a commit log record to stable storage in order to ensure a transaction’s durability.

A fair comparison of alternative techniques requires equal criteria (e.g., serializability) and equal opportunities (e.g., in terms of granularity of concurrency control, access modes, etc.). In such a comparison, the crucial differences are the timing of conflict detection and the options for conflict resolution. For example, if a conflict between two transactions is found after one of them is already committed, failing the other transaction seems the only recourse.

With these issues properly addressed and exploited, pessimistic concurrency control will always match or out-perform optimistic concurrency control. For the time being, this analysis focuses on single-version storage and ignores distributed transactions.

1 Introduction

There is a widely held belief that optimistic concurrency control is better than pessimistic concurrency control, i.e., that end-of-transaction validation permits more concurrent transactions and higher processor utilization than traditional locking. On one hand, this point of view must surely be right: how can concurrency not be higher if transactions can read and write data without any concern for concurrent transactions and without the overhead of lock acquisition? On the other hand, this point of view must surely be wrong: if two transactions truly have a concurrency conflict, any method for concurrency control must detect it; and if two transactions have no true conflict, why would any method for concurrency control hold them back?

These conflicting points of view have long been a troubling aspect in the theory, design, and implementation of database systems and their concurrency control. The same issue bedevils the design and implementation of key-value stores, file systems, streaming engines, graph analytics, and other software systems. Thus, the present paper attempts to gather and present some thoughts that may illuminate the matter. Rather than comparing a small set of specific variants or implementations of the techniques, the analysis and comparison are on a conceptual level, just like the original introduction of optimistic concurrency control [KR 81] as well as subsequent expert reviews of the techniques [H 84, M 94].

The essence of our comparison is that optimistic and pessimistic concurrency control are more similar than different in many respects. For example, both approaches benefit from multi-version storage and from running read-only transactions in snapshot isolation. More importantly, between any pair of read-write transactions, both optimistic and pessimistic concurrency control must detect all actual conflicts.

The difference of optimistic and pessimistic concurrency control is the set of situations erroneously labeled conflicts due to timing, e.g., holding locks longer than truly required, or due to granularity, e.g., locking more than truly required. However, both timing and granularity of concurrency control depend on the specific implementation, not on the choice between optimistic and pessimistic concurrency control.

Further significant differences between optimistic and pessimistic concurrency control are the stages in which transactions detect conflicts and the strategies for conflict resolution viable at that stage. For example, if two transactions conflict but one of them is already committed, aborting the uncommitted transaction seems the only possible remedy.

In the context of database concurrency control, the term “granularity” is usually associated with multi-granularity or hierarchical locking. A typical locking hierarchy may include a database, each table or index, and individual database pages. Modern forms of key-range locking and key-value locking [G 07, GK 15, L 93] are special forms of hierarchical locking. In an ordered index such as b-tree, the gap (open interval) between two key values, one of the adjacent key values, and the half-open interval (gap and key together) form a small hierarchy. The theory of multi-granularity locking [GLP 75] applies directly, while implementations may exploit the limit to two children (a key and a gap) to reduce the number of lock manager invocations [G 07, L 93].
The duration of locks has been considered in the past [DKO 84] but only recent studies have demonstrated the impact on transaction processing throughput [JPS 10, GLK 13]. The original design, early lock release, has flaws that may let uncommitted data values escape. A revised design, controlled lock violation, repairs those flaws without impeding correct concurrency [GLK 13]. In the context of the present discussion, early lock release and controlled lock violation are important because optimistic concurrency control detects conflicts only when transactions execute concurrently, without regard to the duration of their commit activities such as hardening, i.e., installing the write set in the persistent database and writing log records to stable storage. In contrast, traditional implementations of pessimistic concurrency control detect conflicts throughout the lifetime of transactions’ locks, which includes the time to ensure a transaction’s durability. With today’s processors, a small transaction can execute in a time much shorter than it takes to harden the transaction, even if flash technology provides stable storage for the recovery log. Therefore, enforcing locks only during execution of a transaction’s logic but not during its commit activities greatly reduces the potential for lock contention.

Following the distinction central in earlier work [G 10], this paper focuses on concurrency control protecting logical database contents by coordinating user transactions; and it ignores concurrency control protecting in-memory data structures by coordinating software threads. In other words, future work may consider latches, mutexes, spinlocks, critical sections, hardware transactional memory, interlocked instructions, and lock-free data structures. Such work will need to address optimistic and pessimistic techniques as well as cache-conscious optimizations [CHK 01].

The next section reviews related prior work. Section 3 compares one recent technique, orthogonal key-value locking, with prior techniques for row-level locking in ordered indexes. Section 4 gathers observations about optimistic and pessimistic concurrency control, including important implications illuminating their differences and similarities. Section 5 introduces new techniques that combine optimistic and pessimistic techniques for database concurrency control. Section 6 combines the preceding observations, insights, and techniques to describe a new, unified design for concurrency control. Section 7 summarizes and concludes.

2 Related prior work

In order to frame the following discussion, this section reviews transactions, serializability, optimistic and pessimistic concurrency control, row-level locking and key-value locking, snapshot isolation and multi-version storage, early lock release, and controlled lock violation. Readers familiar with those subjects may choose to skip ahead.

Interestingly, neither the authors of [DKO 84, GLK 13, JPS 10] nor those of [GK 15] appreciated that they were addressing crucial weaknesses in the common understanding and in contemporary implementations of locking and thus in the relative merits of optimistic and pessimistic concurrency control. Each of these techniques in isolation is probably not sufficient to equalize optimistic and pessimistic concurrency control. Thus, the firmly yet mistakenly believed advantage of optimistic concurrency control remained unchallenged.

2.1 Transactions

If multiple users or applications query and modify a database concurrently, transactions and their isolation property (synchronization atomicity) provide predictable behavior and thus greatly simplify development and quality assurance of application software. Equivalence to some serial schedule defines full transaction isolation. In practice, the equivalent serial sequence of read-write transactions is equal to the order of their commit log records in the recovery log.

Read-only transactions leave no trace in the recovery log. Their place in the equivalent serial schedule is somewhat arbitrary. While traditional implementations of locking ensure that their end-of-transaction defines this place, new implementations reduce concurrency control conflicts by ensuring equivalence to their start-of-transaction. This is the definition of snapshot isolation. Snapshot isolation and multi-version storage complement each other well.

Serializability can be defined as “repeatable count,” i.e., it requires guaranteed continued presence of data items already seen (also known as “repeatable read”) as well as guaranteed continued absence already sought but not found (also known as the “phantom problem”). The remainder of this paper is about serializability, ignoring weaker transaction isolation levels because they leave the most difficult work to application developers. The common justification for weaker transaction isolation levels has been higher concurrency; perhaps the discussion below on phantoms in various locking models will deflate the argument.

If a system supports snapshot isolation and multi-version storage, all read-only transactions should run in snapshot isolation, which puts their commit points in an equivalent serializable schedule at their start-of-transaction. Read-write transactions, however, should run in serializable transaction isolation with a single, end-of-transaction commit point. In this design, read-only transactions never block read-write transactions and read-write transactions never block read-only transactions. Obviously, read-only transactions never block each other. The remainder of this paper is about read-write transactions in serializable transaction isolation, with end-of-transaction as their commit point.
Figure 1 illustrates two transactions and their commit points. Even if transaction $T_1$ starts before transaction $T_2$, it runs logically later, because transaction $T_2$ runs in snapshot isolation and thus its commit point is before the commit point of transaction $T_1$. In other words, transaction $T_2$ does not “see” database updates of transaction $T_1$.

2.2 Optimistic concurrency control

The essence of optimistic concurrency control is that all conflict detection occurs at end-of-transaction – until then, a transaction executes optimistically assuming that no conflict exists. When a transaction attempts to commit, it compares its read set and its write set to those of other transactions. A transaction without conflicts may indeed commit whereas a transaction with conflicts must roll back. Caching database updates in transaction-private buffers simplifies transaction rollback but requires a separate propagation phase after validation.

Optimistic concurrency control can employ backward validation or forward validation. In backward validation, each committing transaction intersects its read and write sets with those of transactions already committed. Each transaction gathers its read and write sets in a private data structure. Read and write sets of each committed transaction $T$ must persist in a global data structure until no transactions remains active that overlapped with transaction $T$. In forward validation, each committing transaction intersects its read and write sets with those of active transactions (presumably committing in the future). Each active transaction publishes new items of its read and write sets immediately such that committing transactions can correctly intersect their read and write sets with all active transactions. There is no need to retain read and write sets of committed (completed) transactions.

Figure 2 illustrates backward and forward validation. When transaction $T_2$ attempts to commit, backward validation intersects its read and write sets with those of transaction $T_1$ and other committed transactions. The commit logic of transaction $T_3$ will find all conflicts between transactions $T_2$ and $T_3$. In contrast, with forward validation, transaction $T_2$ intersects its read and write sets with those of transaction $T_3$ and other active transactions. The commit logic of transaction $T_1$ already found all conflicts between transactions $T_1$ and $T_2$.

The initial design of optimistic concurrency control specified one transaction at a time in its validation and write phases [KR 81]. In many-core systems, concurrent (multi-threaded) validation of multiple transactions seems desirable. It requires that each transaction intersect its read and write sets not only with committed transactions (in backward validation) or with active transactions (in forward validation) but also with other committing transactions. The required data structure shared among all committing transactions is quite similar to the hash table in a traditional lock manager.

Ever since the introduction of optimistic concurrency control broke the monopoly of locking on concurrency control in databases, it has held a certain fascination that ignored the critiques by experts [H 84, M 94] as well as performance simulations [CS 84, MN 82]. These early simulations indicate that in workloads with few conflicts, all techniques for concurrency control perform well, and that in workloads with many conflicts, effort wasted on doomed transactions (those bound to fail validation) determines overall system throughput.

2.3 Snapshot isolation

Snapshot isolation is a transaction isolation technique typically used with multi-version storage. Each update transaction creates a new version of each modified page, record, index entry, etc. Read operations always choose a committed version. Therefore, read operations do not require locks. This is the attractive quality of snapshot isolation; the undesirable quality is that snapshot isolation fails to ensure serializability and thus fails to completely isolate concurrent transactions.

Update transactions commit at end-of-transaction as usual but read operations “see” the database contents as committed before transaction start. In effect, such a transaction has two commit points: one at start-of-transaction for all read operations and one at end-of-transaction for all updates. Obviously, a read-write transaction operating in this way is not atomic in the sense of indivisible, whereas a read-only transaction is.

There are numerous designs for logic and data structures added to snapshot isolation in order to achieve serializability – see [CRF 09] and its references. These designs proactively avoid write-write conflicts and abort a transaction upon adjacent read-write conflicts, discovered by means of “snapshot isolation read” locks set similarly to traditional shared locks but unable to create transaction conflicts. It is unknown whether adjacent read-write conflicts and immediate transaction abort are more or less efficient and scalable than multi-version two-phase locking, where waiting for a lock acquisition can resolve a read-write conflict.
A well-known alternative focuses on entire transactions rather than individual database accesses. Read-only transactions read databases contents already committed at start-of-transaction and read-write transactions read and modify up-to-date database contents.

This design is quite old. For example, Chan et al. wrote in 1982 [CFL 82] “that update transactions perform two-phase locking. That is, only a single update transaction can be granted the exclusive privilege of creating a new version for any data object at any point in time, and the update transaction must also base its updates on the latest consistent version of the database. … Read-only (retrieval) transactions, on the other hand, do not set locks. …by reading only the first [i.e., newest] 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.”

Nonetheless, today this design is often ignored. For example, even after Carey and Muhanna [CM 86] found multi-version two-phase locking efficient and robust, Cahill et al. [CRF 09] compared multi-version snapshot isolation only with single-version two-phase locking, ignoring multi-version two-phase locking. For a mixed workload with both read-only and read-write transactions, they found (multi-version) snapshot isolation superior to (single-version) two-phase locking without separating the effects of snapshot isolation and of multi-version storage.

Cahill et al. [CRF 09] introduced snapshot isolation read ("SIREAD") locks which track a transaction’s read set within a lock manager but do not cause or detect conflicts, only read-write “anti-dependencies.” Our work below also employs locks in optimistic concurrency control but in different ways.

### 2.4 ARIES row-level locking

In ARIES/IM (“index management”) [ML 92], the finest granularity of locking is a logical row in a table. Locking a row implicitly locks all index entries for that row, one in each index of the table, plus the gap (open interval) to the next-lower entry in each index. Section 3 provides more details in its comparisons of alternative methods for a fine granularity of locking in ordered indexes such as b-trees.

ARIES/IM is great for single-row insertion and deletion, because a single lock acquisition suffices for changes in multiple indexes and thus multiple database pages. It also avoids reading inconsistent pieces of a single row from multiple indexes, e.g., when two secondary indexes provide all columns required in a query, i.e., when “covering” the query by a join on the row identifier. ARIES/IM is not very concise when querying a secondary index, since locking an index entry requires locking the relevant logical row and its index entries in all indexes on the table. Moreover, ARIES/IM locks a gap between two index entries in each one of these indexes.

Phantom protection (locking absence by locking a gap between index entries) is particularly troublesome with ARIES/IM due to its surprising side effects. In order to lock a gap (open interval) between two index entries in a secondary index, ARIES/IM locks the next-higher index entry. It does so by locking its logical row including an index entry and a gap in all secondary indexes. Thus, an insertion in an index on column B might wait for a lock due to an unsuccessful index search on column A. Perhaps needless to say, such effects may create the impression that locking inherently inhibits concurrency. It similarly may create the impression that serializability inherently is a low-performing transaction isolation level.

#### 2.5 ARIES key-value locking

ARIES/KVL [M 90] locks distinct key values in a secondary index. A lock covers a distinct key value, its list of row identifiers (possibly many in a non-unique index), and the gap (open interval) to the next-lower distinct key value present in the index. Section 3 provides more details in its comparisons of alternative methods for a fine granularity of locking in ordered indexes such as b-trees.

ARIES/KVL is great for queries in non-unique indexes: in an exact-match query, a single lock acquisition suffices for many index entries, and in a range query, a lock is required for each distinct key value, not for each individual index entry. Insertion of a new row into a table as well as deletion of a row require a lock in each index. Each lock covers a distinct key value and all its row identifiers, plus the gap to the next-lower distinct key value. Even if a deletion affects only a single row, a lock covers possibly very many index entries (plus a gap). Phantom protection is similarly troublesome because it requires a lock on the next-higher distinct key value and all its row identifiers, perhaps thousands. Again needless to say, such effects may create the same bad impressions about locking and serializability as ARIES/IM.

### 2.6 Key-range locking

The initial design for key-range locking [L 93] aims to increase concurrency among b-tree operations by increasing the number of lock requests compared to both ARIES designs. Locks are specific to a single primary or secondary index and to individual index entries. Thus, a query with an equality predicate may require thousands of locks if there are thousands of duplicate key values. In a non-unique index, the lock identifier comprises both the key value and the row identifier. In addition, a special “insert” lock mode aims to increase concurrency among insertions but it is outside the standard theory of locking. Like ARIES, this design for key-range locking relies on “instant duration” locks, i.e., locks not retained until end-of-transaction and thus outside the established theory of two-phase locking. Finally, the original design for key-range locking separates lock modes for an actual index entry and the gap to the next index entry. For example, a “rangeX-S” mode protects a gap between
A subsequent design for key-range locking makes lock modes for index entry and gap entirely orthogonal [G 10]. The remainder of this paper refers to it as orthogonal key-range locking. It does not require a special “insert” lock mode or “instant duration” locks. Ghost records, i.e., records logically deleted but not physically erased [G 10], also known as “pseudo-deleted” records or as “delete stubs,” simplify and speed up both deletions and insertions.

2.7 Orthogonal key-value locking

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

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

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

Ancillary differences include use of system transactions and of ghost records for insertion of new key values as well as system transactions creating and removing ghost space within individual records. Thus, system transactions perform all allocation and de-allocation operations in short critical sections with inexpensive invocations (no new execution thread) and transaction commits (no log flush on commit). User transactions merely modify pre-allocated space, including the ghost bits in each index entry. This greatly simplifies logging and rollback of user transactions as well as space management.

Incidentally, ghost records permit a useful optimization for a specific but frequent access pattern that combines updates and insertions. In fact, this access pattern is found in some SQL dialects, e.g., as “upsert” command. It is also the default behavior in some key-value stores. If a serializable transaction first checks the existence of a key value and, after finding no instance, inserts a new row with the same key value. In the default method for concurrency control, the initial check locks the key value’s absence by locking an entire gap. Instead, the check can insert a ghost instance of the key value and then only lock that single key value, which the subsequent insertion will turn into a valid record. In this case, neither the check nor the insertion need to lock a gap between actual key values. Insertion of new ghost records for phantom protection is a bad idea in most other cases, in particular for range predicates.

2.8 Early lock release and controlled lock violation

It is well known that read-only participants in a two-phase commit may release their locks during the 1st commit phase. In fact, read-only local transactions need not even participate in the 2nd commit phase. Similarly, a read-write participant in two-phase commit may release its read-only locks in the 1st commit phase but must retain its non-read-only locks to the 2nd commit phase. Retaining these locks ensures efficient and conflict-free implementation of the global commit decision including rollback, if necessary.

In fact, any transaction may release its read-only locks early, i.e., when locks are required only to ensure conflict-free rollback should it become necessary. More specifically, a transaction may release its read-only locks immediately after the user or the application requests transaction commit or abort, i.e., when it becomes certain that the transaction will not repeat any read or count operations. Moreover, a transaction may release its non-read-only locks immediately after allocating a commit log record in the in-memory log buffer, i.e., after assigning a log sequence number to the commit log record and thus giving the transaction a place in the equivalent serial order of transactions.

Once a commit log record is in the in-memory log buffer, only a system failure can invalidate the log buffer (as well as all other memory contents). In that case, recovery during restart may find transactional updates without a commit log record and roll back the transaction, but it will also roll back any subsequent pre-crash transactions. While writing the log buffer to stable storage is required for transactional durability, it is not required for correct concurrency control. With respect to transaction isolation (concurrency control, synchronization atomicity), the transaction is committed when it has a place in the equivalent serial order of transactions.

In a long-running transaction, it makes little difference whether or not the transaction retains its locks while writing the commit log record. For short transactions, however, the difference can be substantial. Moreover, this difference has been growing over time.
Figure 3 illustrates transaction processing decades ago and today. With a small memory and buffer pool (e.g., less than a MB), each transaction might incur multiple page faults. Thus, transaction execution typically takes much longer than hardening the transaction by writing a commit log record to stable storage. Today, most transaction processing systems keep their entire working set or perhaps the entire database in memory. Execution of the transaction logic takes much less time than writing a commit log record to stable storage, even with the recovery log on flash storage. For example, execution of a transaction’s logic may require 100,000 CPU cycles or 0.025 ms at 4 GHz, whereas writing to a flash device may take 0.1 ms or four times longer.

Figure 4 illustrates three transactions that access the same database item. Above the timeline, the transactions follow the traditional scheme and retain their locks until their commit log records are on stable storage. Below the timeline, the transactions employ early lock release or controlled lock violation such that the next transaction can run while the prior one is still becoming durable. It is obvious that early lock release and controlled lock violation reduce concurrency control conflicts and improve transaction throughput in workloads with conflicting database accesses.

Early lock release [DKO 84, JPS 10] may be incorrect if a read-only transaction “publishes” an unlocked value that eventually rolls back during system restart. In distributed transactions, this may even lead to erroneous database contents [GLK 13]. In controlled lock violation, locks remain in place and visible to other transactions but are weak and permit violation, e.g., acquisition of an exclusive lock while another, weak, exclusive lock is still in place. Locks become weak as soon as a transaction has a commit log record in the log buffer; a single flag in the transaction’s state indicates that all its locks are weak. Except for some special cases, early lock release and controlled lock violation are more similar than different in their effects on concurrency, transaction schedules, and transaction processing rates.

2.9 Notable technology variants

While traditional theory suggests deadlock detection by analyzing the global “waits for” graph, assembled from lock managers in all nodes in the case of distributed transactions, a pragmatic solution to deadlock detection relies on lock timeouts. If a lock request waits too long, a deadlock is assumed and the waiting (or waited-for) transaction aborts. Limited wait depth [T 97, T 98] reduces the probability of long wait times and thus contributes to deadlock avoidance. A guaranteed method for deadlock avoidance assigns unique priorities to all active transactions, e.g., using a timestamp at start-of-transaction. High-priority transactions may grant a brief grace period to low-priority transactions such that a low-priority transaction may finish rather than abort just before completion. Priority schemes that are more dynamic may use the sizes of transactions, e.g., the number of log records written prior to the conflict.

Traditional descriptions of optimistic concurrency control have been limited to read and write sets. Recent work proposes optimistic concurrency control adding a set of predicates [NMK 15], with an efficient implementation that compiles predicates to machine code. Thus, each transaction adds a predicate set or replaces its read set with a predicate set. In a technology transfer from optimistic to pessimistic concurrency control, it seems possible to add predicates to a lock manager in the same way as adding it to optimistic concurrency control. Thus, the techniques tracking a predicate set in addition to read and write sets also enables precision locking [JBB 81].

There is also plenty of established technology for distributed transactions. The cornerstone is two-phase commit – in the 1st phase, all participants promise to abort or to commit as the coordinator decides; and in the 2nd phase, the coordinator makes and announces its decision. A traditional optimization of two-phase commit lets read-only participants exit after the 1st phase. In the spirit of early lock release, a little known optimization permits all participants (not only the read-only participants) to release their read-only locks in the 1st phase, with corrections embodied in controlled lock violation [GLK 13]. Read-write
participants retain their non-read-only locks in order to guarantee their ability to implement the coordinator’s decision. Participants may employ controlled lock violation directly after they know the coordinator’s commit decision; they may employ it even during the wait for the decision at the risk of cascading abort.

Proven techniques for pessimistic concurrency control include hierarchical or multi-granularity locking [GLP 75]. Multi-granularity locking permits both coarse and fine granularities of locking for the same data at the same time. Established theory also covers ‘increment’ locks, which have been discussed repeatedly [K 83, G 10] but are rarely implemented in spite of their promise for materialized views defined with “group by” clauses. It is not obvious how to transfer these techniques to optimistic concurrency control.

Most concurrency control techniques support dynamic needs, i.e., the transaction logic determines on the fly whether certain operations are required. For example, a scripted transaction may read a database value and then decide whether to perform an update. Static schemes that require pre-declared concurrency control needs may achieve higher concurrency [FA 15] but require over-reservation or rollback and restart in dynamic applications.

2.10 Summary of related work

In summary of earlier work, optimistic concurrency control might permit more concurrency than traditional row-level locking and key-value locking, but new locking techniques may change the balance of advantages and disadvantages.

3 Case studies on locking in ordered indexes

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

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

<table>
<thead>
<tr>
<th>EmpNo</th>
<th>FirstName</th>
<th>PostalCode</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Gary</td>
<td>10032</td>
<td>1122</td>
</tr>
<tr>
<td>3</td>
<td>Jerry</td>
<td>46045</td>
<td>9999</td>
</tr>
<tr>
<td>5</td>
<td>Mary</td>
<td>53704</td>
<td>5347</td>
</tr>
<tr>
<td>6</td>
<td>Jerry</td>
<td>37745</td>
<td>5432</td>
</tr>
<tr>
<td>9</td>
<td>Terry</td>
<td>60061</td>
<td>8642</td>
</tr>
</tbody>
</table>

Figure 5. An example database table.

Figure 6 illustrates index records in a non-unique secondary index on FirstName. This index format pairs each distinct key value with a list of bookmarks. Unique search keys in the primary index serve as bookmarks; they are also the table’s primary key here.

3.1 Empty queries – phantom protection

The first comparison focuses on searches for non-existing key values. Assuming a serializable transaction, a lock is required for phantom protection until end-of-transaction. In other words, the first comparison focuses on techniques that lock the absence of key values. The example query is “Select... where FirstName = ‘Harry’ ”, which falls into the gap between key values Gary and Jerry. Section 3.3 considers designs like Tandem’s in which an unsuccessful query may introduce a new value into the database or the lock manager – the discussion here considers only locks on pre-existing key values and indexes entries. Figure 7 illustrates the following discussion.

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

Revisiting concurrency control
ARIES/IM also locks the next higher key value, 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, i.e., the index entry in the primary index, as well as the index entry in the secondary index on FirstName, an index entry in each further secondary index, plus (in each index) the gap between those index entries and the next lower key value present in the index. While this lock is in place, no other transaction can insert a new row with FirstName 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. In fact, no other transaction can insert any row with EmpNo 2, because lock includes the gap below EmpNo 3. Finally, assuming the database table of Figure 5 with another secondary index on PostalCode, the lock on EmpNo 3 also stops other transactions from inserting PostalCode values between 37745 and 46045. These certainly are surprising 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 employee (Jerry, 2) or (Gary, 7) is not possible.

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

Finally, orthogonal key-value locking locks the preceding distinct key value, Gary, in a mode that protects the gap (open interval) between Gary and Jerry but imposes no restrictions on those key values or their lists of EmpNo values. For example, another transaction may insert a new row with FirstName Gary or 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 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.

<table>
<thead>
<tr>
<th>Index entries and gaps</th>
<th>Gary, 1</th>
<th>gap (Gary, &gt;1)</th>
<th>gap (Gary, &lt;Jerry)</th>
<th>gap (Jerry, &lt;3)</th>
<th>Jerry, 3</th>
<th>gap (Jerry, &gt;6)</th>
<th>gap (&gt;Jerry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARIES/KVL</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARIES/IM</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KRL</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Orth. krl</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Orth. kvl</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7. Required and actual locking scopes in phantom protection for ‘Harry’.

Figure 7 illustrates required and actual locking scopes for the example of an unsuccessful query, i.e., for phantom protection. The column headings indicate ranges in the domain of the index keys. Note that the gap between index entries (Gary, 1) and (Jerry, 3) comprises three sub-ranges. A serializable transaction with an unsuccessful query for Harry requires concurrency control only for one of them. An S in Figure 7 indicates that a locking technique acquires a transaction-duration shared lock in order to prevent insertion of index value Harry. It is clear that ARIES/KVL has the largest scope and that orthogonal key-value locking has the minimal scope. ARIES/IM appears equal to key-range locking only because Figure 7 does not show the locking scope in the other indexes of this database table.

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, in addition to operations on the key value requiring phantom protection, also restrict operations on pre-existing key values.

3.2 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 8 illustrates the following discussion.

ARIES/KVL requires 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 Harry.

ARIES/IM locks three rows in the table including both rows with FirstName Jerry (rows 3 and 6) and the next higher key value, 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 desired key value with its unique index entry, i.e., (Jerry, 3) and (Jerry, 6), plus the next higher actual key value, 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 values 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 acquires one lock for all actual and possible index entries with FirstName Jerry. Both adjacent key values remain unlocked, i.e., Gary and Mary. Moreover, even the gaps below and above FirstName Jerry remain unlocked, i.e., other transactions are free to insert new index entries with FirstName Harry or Larry.

<table>
<thead>
<tr>
<th>Index entries and gaps</th>
<th>Gary, 1</th>
<th>gap (Gary, &gt;1)</th>
<th>gap (&gt;Gary, &lt;Jerry)</th>
<th>gap (Jerry, &lt;3)</th>
<th>Jerry, 3</th>
<th>gap (Jerry, &gt;6)</th>
<th>gap (&gt;Jerry, &lt;Mary)</th>
<th>gap (Mary, &lt;5)</th>
<th>Mary, 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARIES/KVL</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>ARIES/IM</td>
<td>S1</td>
<td>S2</td>
<td>S3</td>
<td>S3</td>
<td>S3</td>
<td>S3</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>KRL</td>
<td>S1</td>
<td>S2</td>
<td>S2</td>
<td>S2</td>
<td>S2</td>
<td>S2</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>Orth. krl</td>
<td>S1</td>
<td>S2</td>
<td>S2</td>
<td>S2</td>
<td>S2</td>
<td>S2</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>Orth. kvl</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
</tbody>
</table>

Figure 8. Locking scopes in an equality query.

Figure 8 illustrates required and actual locking scopes for the example equality query. Subscripts indicate separate locks and thus lock manager invocations. Whereas the key-range locking methods require multiple locks, the key-value locking methods match the query predicate with a single lock; and whereas the traditional locking methods commingle key values in index entries and gaps between index entries, the orthogonal locking methods do not.

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

3.3 Phantom protection with ghost records

The discussion of locking in successful equality queries also helps understanding locking scopes in phantom protection with a ghost record left behind by a prior deletion. For example, consider the example of Section 3.1 and phantom protection by a ghost index entry with FirstName Harry. In this case, locks for an unsuccessful equality query are similar to the locks for a successful equality query.

<table>
<thead>
<tr>
<th>Index entries and gaps</th>
<th>Gary, 1</th>
<th>gap (Gary, &gt;1)</th>
<th>gap (&gt;Gary, &lt;Harry)</th>
<th>gap (Harry, &lt;47)</th>
<th>Harry, 47</th>
<th>gap (Harry, &gt;47)</th>
<th>gap (&gt;Harry, &lt;Joe)</th>
<th>gap (Joe, &lt;3)</th>
<th>Joe, 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARIES/KVL</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>ARIES/IM</td>
<td>S1</td>
<td>S2</td>
<td>S3</td>
<td>S3</td>
<td>S3</td>
<td>S3</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>KRL</td>
<td>S1</td>
<td>S2</td>
<td>S2</td>
<td>S2</td>
<td>S2</td>
<td>S2</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>Orth. krl</td>
<td>S1</td>
<td>S2</td>
<td>S2</td>
<td>S2</td>
<td>S2</td>
<td>S2</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>Orth. kvl</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
</tbody>
</table>

Figure 9. Phantom protection with a ghost record.

Figure 9 illustrates phantom protection for an unsuccessful selection query using a pre-existing ghost record. The header row indicates the required lock scope and, with the strikethrough font, the ghost record. The two methods locking distinct key
values require a single lock on FirstName Harry. While 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 the value in the ghost record.

If no ghost record matching an unsuccessful selection query exists, the query may invoke a system transaction in order to insert a suitable ghost record. For most methods, careful choice of the EmpNo value in the ghost record improves upon the locks shown in Figure 9. If the chosen EmpNo is $+\infty$, a single lock suffices in ARIES/IM and in key-range locking. Orthogonal key-range locking can use $-\infty$ for the same purpose. 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 $-\infty$, orthogonal key-range locking can also simply use null, assuming it sorts lower than all non-null values.

Index entries and gaps

<table>
<thead>
<tr>
<th></th>
<th>Gary, 1</th>
<th>gap (Gary, &gt;1)</th>
<th>gap (&gt;Gary, &lt;Harry)</th>
<th>gap (Harry, &lt; +\infty)</th>
<th>gap (&gt;Harry, &lt;Joe)</th>
<th>gap (Joe, &lt;3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARIES/KVL</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARIES/IM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KRL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 10. Single-lock phantom protection with next-key locking.

<table>
<thead>
<tr>
<th></th>
<th>Gary, 1</th>
<th>gap (Gary, &gt;1)</th>
<th>gap (&gt;Gary, &lt;Harry)</th>
<th>gap (Harry, null)</th>
<th>gap (&gt;Harry, null)</th>
<th>gap (&gt;Harry, &lt;Joe)</th>
<th>gap (Joe, &lt;3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orth. kr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Orth. kvl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 11. Single-lock phantom protection with prior-key locking.

Figure 10 and Figure 11 show the locks and their scopes for phantom protection using a new ghost record with a carefully chosen key value. ARIES/KVL and orthogonal key-value locking have the same lock counts (one) and lock scopes as in Figure 9. The other three methods require two locks in Figure 9 but only one in Figure 10 and in Figure 11. 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 $-\infty$ and $+\infty$ 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 $+\infty$ in ARIES/IM has surprising consequences. 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 using ARIES/IM.

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

### 3.4 Range queries

The third comparison focuses on range queries. The example query predicate is “…where FirstName between ‘Gary’ and ‘Mary’”. The query result includes all instances of Gary and Mary. Locking for serializability must prevent insertion of additional instances of these key values. The issues to compare among the locking schemes are the number of lock requests and the precision of locking no more than the required key range.

ARIES/KVL needs three locks on key values Gary, Jerry, and Mary. These locks cover the key values and the gaps between them. In addition, the first lock covers the gap below Gary. This is required to prevent insertion of a new row with FirstName Gary and EmpNo 0 but it also prevents new rows with FirstName null, FirstName Aaron, etc.

ARIES/IM acquires five locks on rows 1, 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. These locks also lock index entries in further indexes as well as the gaps (open intervals) below each of the key values.

Key-range locking in SQL Server requires five locks on key values (Gary, 1), (Jerry, 3), (Jerry, 6), (Mary, 5), and (Terry, 9). As in ARIES/IM, the first lock includes the gap below (Gary, 1) 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 (Gary, 1), (Jerry, 3), and (Jerry, 6), all in S mode to cover the index entry and the gap to the next higher key. In addition, in order to prevent insertion of new index entries with FirstName Gary and EmpNo values smaller than 1, a lock on an artificial left bound of the index is required; this can also be FirstName Null. The lock mode for (Null, Null) may be NS (‘key free, gap shared’), leaving the key value free yet protecting the gap to (Gary, 1).
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 Terry. Thus, orthogonal key-range locking seems similar to key-range locking in SQL Server as both lock four key values within the range of the query predicate and one key value 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.

Finally, orthogonal key-value locking requires three locks on the three distinct key values within the range of the query predicate, i.e., FirstName Gary, Jerry, and Mary. For these, it locks all actual and possible EmpNo values. It locks the two gaps between these key values with appropriate lock modes in two of the lock requests. 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, these three locks protect precisely the key range of the query predicate, with other transactions free to insert and delete index entries outside of it.

In summary, orthogonal key-value locking requires the fewest locks yet protects the key range of the query predicate with the best precision.

3.5 Non-key updates

The fourth 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 6 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.

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 above and below it completely unlocked. This is possible with a lock in XN mode (pronounced ‘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, for example, this looks about 1 in 7 or 14% of those instances. For a short list as shown in Figure 6, this is usually a single instance. For a longer list, locking a partition may mean locking multiple instances even if only one instance is needed. Larger choices of k may alleviate hash conflicts and false sharing. The gaps above and below FirstName Jerry remain unlocked.

Figure 12 illustrates required and actual locking scopes for a non-key update of a single index entry. Again, 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.

Figure 12 also illustrates the locking scopes of user transactions inserting and deleting index entries via ghost records, because toggling the ghost bit in a record header is a classic case of a non-key index update. Without ghost records, older techniques lock more, in particular also a neighboring key value or index entry. The following sub-sections provide more details.

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

The fifth comparison focuses on deletion of rows in a table and of index entries in a b-tree. Within an index, a pair of deletion and insertion is required usually for any update of a key column. In other words, the fifth and sixth comparisons also cover updates on key columns in indexes. The example to be discussed here is “delete… where EmpNo = 3”.

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

Non-unique secondary indexes need one bit for each index entry to indicate the ghost status. A list representation needs a bit per element in the list. 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 in system transactions [G 10] or ARIES top-level actions [MHL 92]. Ghost removal does not require transactional locks; latches coordinate threads and protect data structures; log records describe ghost removal if subsequent user transactions log their actions with references to slot numbers within pages rather than key values; and system transactions or top-level actions commit without log flush.

Deletion via a ghost is 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, for example a split of a b-tree node. Deletion via a ghost record requires the same locks as a non-key update, discussed in Section 3.5 and illustrated in Figure 12. 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/KVL, where the commit-duration X lock on the next key indicates the assumption of immediate record removal, i.e., removal without ghost records.

ARIES/KVL distinguishes cases (i) and (ii) above as well as unique and non-unique secondary indexes. In all cases, one distinct key value remains locked in X mode until end-of-transaction in order to protect the 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 above and below the deleted key value in addition to the deleted key value itself.

ARIES/IM needs to lock the affected logical row as well as, in each index, the row with the next higher key value. The lock on the next key value (or its entire logical row) ensures the ability to roll back the deletion. In many ways, this is quite similar to phantom protection discussed above, because the deletion transaction needs to prevent other transactions from inserting a new index entry that might prevent rollback of the deletion. A system using the ARIES/IM techniques definitely benefits from ghost records for deletion.

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.

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 (i) and (ii) above must lock the next lower key value 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 key value itself remains unlocked.

Finally, orthogonal key-value locking also works best with ghost records during both insertion and deletion. When forced to avoid ghost records, cases (i) and (ii) above must acquire and hold locks quite similar to ARIES/KVL, except that the next key value itself remains unlocked in orthogonal key-range locking.

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, orthogonal key-range locking is slightly better than SQL Server key-range locking, orthogonal key-value locking is slightly better than ARIES/KVL, and ARIES/IM requires the fewest locks in tables with multiple indexes but has by far the largest scope in terms of concurrency control and excluded other transactions.

3.7 Insertions

The last comparison focuses on insertion of new rows into a table and thus new index entries in primary and secondary indexes. The example used here is “insert… values (4, ‘Jerry’, 54546, 4499)”. As insertion is the opposite of deletion, there are again three cases to consider. With decreasing complexity, these are (i) insertion of an entirely new distinct key value, (ii) insertion of another instance of an existing key value, and (iii) insertion via ghost status. The ghost record may be a remnant of a prior deletion or the result of a deliberately invoked system transaction or ARIES top-level action. 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 3.5 and illustrated in Figure 12. The remainder of this section focuses on older techniques that do not employ ghost records as preliminary states when inserting new entries into tables and indexes.

ARIES/KVL distinguishes insertions into unique versus non-unique indexes and, within non-unique indexes, insertions of a first instance versus an additional instance of a distinct key value. 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 lock 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 key value in RangeI_N mode to test for conflicting phantom protection and then creates the new key value, retaining a lock in X mode on the new index entry. The lack of a RangeN_X mode forces this X mode lock, i.e., other transactions cannot query or update the gap between the new key value and the next lower key value.

Orthogonal key-range locking works best with a system transaction ensuring that a ghost record exists with the right index key value, whereupon the user transaction merely turns the ghost record into a valid record. The system transaction does not need any locks; it just verifies that no other transaction holds a conflicting lock for phantom protection. The user transaction locks the ghost record for update of non-key attributes as discussed earlier. The XN lock on the ghost-turned-valid record does not restrain other transactions from reading or modifying the gap above or below the newly inserted key value. For insertion without an intermediate ghost record, orthogonal key-range locking first tests that no other transaction ensures phantom protection by holding a lock on the gap above the next lower key value and then locks the newly inserted index entry without the gaps above or below.

Finally, orthogonal key-value locking also works best with a ghost record left by a prior deletion or by a system transaction. Once the ghost record is in place, orthogonal key-value locking locks the appropriate partition within the list 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 below the next higher 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 distinct key value. If the insertion merely adds another instance of an existing key value, an additional lock on the appropriate partition 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 earlier techniques.

3.8 Key updates

Modifying the key value of an index entry requires 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 3.5 and illustrated in Figure 12. Thus, orthogonal key-range locking and orthogonal key-value locking provide the required concurrency control with better precision than all prior methods.

3.9 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 all prior techniques, including orthogonal key-range locking introduced only a few years earlier. In updates including deletion and insertion via ghost status, orthogonal key-range locking is best and orthogonal key-value locking equals orthogonal key-range locking 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 lock scope. Since each inner loop of an index nested loops join acquires locks as illustrated in Figure 7 to Figure 11, join operations multiply the detrimental effects of excessive lock counts and of excessive lock scopes. Thus, precise and efficient locking is important for query processing as much as for data retrieval.

4 Comparing optimistic and pessimistic concurrency control

When comparing optimistic and pessimistic concurrency control as well as designs for their implementation, we observed numerous similarities and differences that seem worth calling out. Two governing principles are the starting point for this investigation. First, any method of concurrency control, including optimistic and pessimistic ones as well as snapshot isolation and multi-version concurrency control, must detect all violations of serializability that truly exist in a schedule of concurrent transactions. Second, all implementations of concurrency control should detect as few false conflicts as possible. Example causes for false conflicts can be a coarse granularity of locking or lock retention longer than truly required.
4.1 Observations

The first few observations pertain to the communication among transactions by recording their data access in a global data structure accessible to all transactions, where the read and write sets of optimistic concurrency control serve the same function as shared and exclusive locks in pessimistic concurrency control.

Both optimistic and pessimistic concurrency control register their concurrency control information in global data structures. In pessimistic concurrency control, the global data structure usually is a hash table within a software module called the lock manager. In optimistic concurrency control, the global data structure contains sets, i.e., a read set and a write set for each transaction.

In optimistic concurrency control with either backward or forward validation, read and write sets remain in the global data structure for a period governed by transaction durations. In backward validation, a single long-running transaction requires that read and write sets of many other transactions remain in the global data structure for a long time. For example, a single long-running transaction prevents dropping read and write sets for many overlapping transactions. In forward validation, retention of a transaction’s read and write sets is governed only by the transaction itself. In other words, when a transaction ends, its read and write are no longer needed independent of the duration of overlapping transactions.

In pessimistic concurrency control, lock acquisition may occur at any time during execution of the transaction logic; on average, it occurs about halfway and lock retention is about half the transaction duration (excluding commit processing). Thus, transactions may sometimes overlap with conflicting locks but without actual conflict. For example, transaction T1 may start before transaction T2, and transaction T1 may acquire an exclusive lock on a data item after transaction T2 acquired an exclusive lock on the same data item, if transaction T2 committed and released its locks before transaction T1’s lock request. In contrast, optimistic concurrency control detects a conflict based on overlapping entire execution times, without regard to timing within a transaction. Figure 13 illustrates the example using dashed lines for lock durations.

![Figure 13. Example transactions T1 and T2.](image)

The following observations focus on algorithms and data structures required to detect conflicts among concurrent transactions. In optimistic concurrency control, end-of-transaction validation requires intersecting sets, i.e., read sets and write sets. Set intersection is equivalent to a join operation in relational query processing, which employs algorithms based on sorting (usually merging), hashing (akin to distribution sort), or indexes. The same variety of algorithms applies to intersection computations for optimistic concurrency control.

Therefore, if each transaction runs concurrently with t other transactions, then each end-of-transaction validation requires intersection computations with t other transactions. If each set contains s items, and if each set is sorted, then intersecting two sets requires $2s-1$ comparisons to advance the merge logic. If all transactions are of the same size, each end-of-transaction validation requires $t(s_1+s_2-1)$ comparisons for set intersection by merging.

If set intersection employs a hash table, then each transaction’s read and write sets requires $s_1+s_2$ insertions and deletions in the hash table. The search key of the hash table is the identifier of the data items read or written.

There can be a local hash table per transaction or a single global hash table holding all transactions’ read and write sets. If the hash tables are local, entire hash tables are deleted; if the hash table is global, it must support efficient deletion of all items belonging to an individual transaction. The global hash table may be distributed, e.g., parts of it attached to database pages resident in the buffer pool.

A local hash table may be temporary for a single validation (comparable to the hash table of a traditional hash join) or permanent (comparable to the index in a traditional index nested loops join). Temporary hash tables require frequent creation and deletion; this approach hardly saves effort compared to frequent set intersection operations by merging sorted lists. A global hash table ought to be permanent (within the server process, not in the database).

If the hash table is global and permanent, then the concurrency control for each data access costs one insertion (during transaction execution or commit) and one deletion (during or after commit). The similarity with traditional locks, lock managers, and their hash tables is striking.

The information in the global hash table deserves more scrutiny, in particular with respect to access modes, with an item in a read set (in optimistic concurrency control) equivalent to a shared lock (in pessimistic concurrency control) and an item in a write set equivalent to an exclusive lock.

4.2 Implications

The observations above suggest multiple implications. First, a comparison of forward and backward validation: Forward and backward validation detect the same set of conflicts among the same pairs of transactions. An exception is the case shown in Figure 13. The principal difference, however, is whether a conflict is found during the first or the second commit within a pair of conflicting transactions. When backward validation detects a conflict between a committing transaction and a committed...
transaction, the only recourse is to abort the transaction attempting to commit. In contrast, when forward validation detects a conflict between a committing transaction and an active transaction, either transaction could abort to resolve the conflict. In other words, the set of conflict resolution strategies available with forward validation is a strict superset of those available with backward validation.

Next, a comparison of forward validation and locking: Both forward validation and locking register their data accesses before they occur. However, when forward validation registers a data access, it does not check for conflicts even if conflicting accesses are readily visible in a permanent, global hash table tracking all transactions’ read and write sets. Forward validation and locking detect the same set of conflicts among the same pairs of transactions. Their difference is whether a conflict is found at the time of the data access or during the first commit within a pair of conflicting transactions. This assumes the same set of access modes considered in concurrency control, i.e., reading and writing but not intention locks, increment locks, key-range locks, key-value locks, etc. When locking detects a conflict between two transactions, either transaction can abort to resolve the conflict, or either transaction can de-escalate a coarse lock in the conflict to finer granularity of locking, or the requesting transaction can wait indefinitely or with a timeout. In other words, the set of conflict resolution strategies available with locking is a strict superset of those available with forward validation.

The overall implication of these observations and their individual implications is the following, which is a main insight and conclusion of this research: If both optimistic and pessimistic concurrency control guarantee serializability (“repeatable count”), and if pessimistic concurrency control locks as little as possible as briefly as possible, then pessimistic concurrency control permits at least as much concurrency as optimistic concurrency control, permits more options for conflict resolution, and wastes less effort on doomed transactions, i.e., those bound to fail validation.

Table 1 summarizes similarities and differences of optimistic concurrency control, including both backward and forward validation, and pessimistic concurrency control, i.e., locking. All methods require that each transaction’s reads and writes be visible to other transactions for a period. A global data structure similar to a lock manager’s hash table is more efficient than sorted sets and transaction-by-transaction merge operations (the lock manager may partition its hash table, e.g., with appropriate partitions attached to database pages in the buffer pool). While forward validation and locking insert their reads and writes into the hash table eagerly, backward validation delays insertions to end-of-transaction. In consequence, forward validation and locking may erase their entries in the hash table at end-of-transaction, while backward validation must wait until the last concurrent transaction ends. Thus, a single long-running transaction may swell the hash table with entries of many small transactions, which cannot occur in forward validation and in locking.

Another peculiarity readily apparent in Table 1 is the split of conflict visibility and conflict detection in forward validation. Two transactions may insert conflicting entries into the hash table, but they remain oblivious to the conflict until the first one attempts to commit. Both backward validation and locking detect conflicts as soon as they become visible, whereas forward validation lets conflicting transactions proceed, only to roll one of them back.

The greatest differences among the techniques are the options for conflict resolution and the set of related techniques developed in research and industrial practice. Waiting for a lock is often the best strategy if a conflict exists, in particular if moderated by a limited wait depth [T 97, T 98] and by a timeout with subsequent transaction abort, which reliably prevents deadlocks. Optimistic concurrency control, on the other hand, can resolve conflicts only by aborting one of the transactions, whereas pessimistic concurrency control permits transaction abort as a possibility but notably not as the most promising one.
Finally, both backward validation and locking benefit from tracking the oldest active (read-write) transaction. Backward validation requires the information for timely deletion of read and write sets in the global data structure. Locking optionally can short-circuit or, in weak isolation levels such as “read committed,” avoid lock acquisition for data on database pages unmodified by all active transactions [M 96].

4.3 Summary of the comparison

In summary, for a fair comparison of optimistic and pessimistic concurrency control, the methods should exploit similar foundations, e.g., a hash table relating transactions and database items, and provide similar guarantees, e.g., serializability or “repeatable count” isolation. For efficiency of query processing and minimal interference, read-only transactions should run in snapshot isolation, i.e., with their commit points at start-of-transaction, supported by multi-version storage. For serializable transaction isolation for all transaction, including transactions with dynamic concurrency control needs as well as distributed transactions, read-write transactions should run with their commit points at end-of-transaction.

With those foundations and assumptions, if traditional locking is modified to lock as little as possible as briefly as possible, then the techniques are very similar in the conflicts they detect and thus in the concurrency they permit; they differ in the timing of conflict detection, in the options for conflict resolution, and in the effort wasted on doomed transactions.

5 Technology transfer and new techniques

The observations above suggest some opportunities for technology transfer. First, hierarchical or multi-granularity concurrency control as known from locking can also work in optimistic concurrency control. It would require intention sets in addition to read and write sets with additional set intersection operations during transaction commit. Second, many descriptions of optimistic concurrency control gloss over creation and removal of data items, e.g., of unique key values in an index. At least in the context of forward validation, ghost records and system transactions [G 10] as well as low-level pessimistic concurrency control, i.e., latching, could help address these problems in a simple and effective way.

Table 1 and its discussion fail to emphasize some aspects of transaction execution and concurrency control of more recent interest. First, early lock release or controlled lock violation reduce lock retention times to a fraction compared to traditional implementations. Figure 3 illustrates this effect in the past and today. Second, orthogonal key-value locking permits efficient locking for equality and range queries, phantom protection without interference with key values, and precise locking for insertions, updates, and deletions. Thus, it eliminates most of the excessive lock footprint that ARIES/IM and ARIES/KVL require for phantom protection and serializability. Third, recent research [NMK 15] into optimistic concurrency control exploits scan predicates, ideally compiled to machine code. Thus, end-of-transaction validation processes a predicate set in addition to read and write sets. Adding predicates to a lock manager and to pessimistic concurrency control is an obvious and promising technology transfer opportunity. It may render one aspect of orthogonal locking obsolete, namely locking gaps (and only gaps) for phantom protection. Together, these techniques permit locking as little as possible as briefly as possible. Table 2 summarizes these technologies and their effects. Together, these techniques reduce the conflict potential of pessimistic concurrency control to that of optimistic concurrency control.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Effect in pessimistic concurrency control</th>
<th>Comparison to optimistic concurrency control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Early lock release, controlled lock violation</td>
<td>Lock retention times reduced to a fraction</td>
<td>More similar conflict potential in locking and validation</td>
</tr>
<tr>
<td>Orthogonal key-value locking</td>
<td>Efficient locks for queries, precise phantom protection, precise locks for updates</td>
<td>Serializability with minimal conflicts</td>
</tr>
<tr>
<td>Predicates in concurrency control</td>
<td>Very efficient locks for queries, very precise phantom protection</td>
<td>Very recent research; technology transfer opportunity</td>
</tr>
</tbody>
</table>

The analysis above suggests a number of new techniques, e.g., the potential technology transfers sketched earlier. The following new techniques go beyond those suggested above.

5.1 Transaction conversions

A single transaction may change its method of concurrency control for a data item during the transaction’s lifetime. This is most useful for a transaction that starts as a read-only transaction, i.e., in snapshot isolation, and needs to convert into a read-write transaction. This technique applies to both multi-version storage systems and single-version storage systems.
Figure 14 illustrates two transactions similarly to Figure 1 plus a third transaction. This transaction T₃ starts in snapshot isolation with a start-of-transaction commit point. Initially, it “sees” the effects neither of transaction T₁ nor of transaction T₂. If and when transaction T₃ attempts a first database update, it needs to become a read-write transaction with an end-of-transaction commit point. The conversion must detect any conflicts between transaction T₃ and either transaction T₁ or transaction T₂.

Conversion of a read-only transaction in snapshot isolation to a read-write transaction with two-phase locking is tantamount to shifting the transaction’s commit point from start-of-transaction to end-of-transaction. This shift is possible if the read-only transaction gathers a read set similar to a transaction in optimistic concurrency control and if, at the time of the conversion, the transaction can acquire the necessary shared locks for the same versions that it might have acquired at start-of-transaction.

In a system with multi-version storage, it is simple to determine whether a converting read-only transaction can acquire shared locks on the correct versions. Lock acquisition succeeds if there is no uncommitted version and the newest committed version is older than the transaction’s start-of-transaction. A system with single-version storage may employ an approximate test similar to the “commit lsn” test [M 96]. Otherwise, this test is difficult or impossible such that lock acquisition and transaction conversion must fail. Fortunately, many systems with snapshot isolation employ multi-version storage.

Lock acquisition during conversion limits concurrency more rigorously than lock acquisition immediately before data access. For example, if two transactions start at the same time in read-write mode, then one of these transactions updates a data item and commits, thereafter the second transaction can acquire a lock on the updated data item. If the second transaction starts in read-only mode in snapshot isolation, however, it cannot convert from a read-only to a read-write transaction. This illustrates a difference between optimistic and pessimistic concurrency control not called out in earlier work, namely that validation considers the entire period from start-of-transaction to commit (validation) whereas locking detects conflicts only from data access (and lock acquisition) to commit.

Today, most database systems run all transactions as read-write transactions unless specific syntax marks a new transaction as a read-only transaction. In the future, it might be best to start all transactions as read-only transactions unless specific syntax marks a transaction a read-write transaction. Note, however, that transaction conversion requires remembering the read set of each read-only transaction, probably in transaction-private space. Note also that this read set must include failed searches such that, during conversion, the transaction can acquire the required locks on gaps within indexes.

5.2 Transaction-private update buffers

Another technology transfer opportunity goes from optimistic to pessimistic concurrency control and focuses on the acquisition time and thus the duration of exclusive locks. Optimistic concurrency control requires transaction-private buffers to hold all uncommitted updates until end-of-transaction validation succeeds. Pessimistic concurrency control usually does not use such buffers. Instead, lock acquisition precedes in-place updates including changes in a record’s ghost bit.

Transaction-private buffers for updates permit deferring the acquisition of exclusive locks, because they break the traditional design relying on in-place updates. If a transaction acquires all its exclusive locks at the same time, it can sort them by object identifier and thus substantially reduce the probability of deadlocks. Moreover, acquisition of exclusive locks after completion of the transaction logic ensures short retention of exclusive locks, and short timeouts in lock requests resolve any deadlocks that might still occur.

The update buffers may be in a temporary memory allocation or in the database. The former case resembles the traditional implementation in optimistic concurrency control; the latter case requires an exclusive lock but only on the transaction’s new, uncommitted records. Similar to version records, a new record may go next to the old record or into a dedicated update area, e.g., Vertica’s write-optimized store [LFV 12] or an in-memory partition in a partitioned b-tree [G 03]. Ideally, uncommitted records exist only while the affected database pages remain in the buffer pool (or equivalently, the buffer pool does not write or evict dirty pages with uncommitted version records). A transaction rollback erases the version record and a transaction commit erases the previous record (or keeps it for read-only transactions in snapshot isolation).

The overhead of a transaction-private update buffer is, of course, the same for optimistic as for pessimistic concurrency control. The effect in optimistic concurrency control is that a transaction requires exclusive ownership of an updated database item only during the validation and write phases; the effect in pessimistic concurrency control is that a transaction requires exclusive locks only during a similarly short period at commit time.

<table>
<thead>
<tr>
<th>Step</th>
<th>Transaction activity</th>
<th>Locks</th>
<th>Transaction state</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>i.</td>
<td>Transaction begin</td>
<td></td>
<td>Active</td>
<td>Empty transaction-private update buffer</td>
</tr>
<tr>
<td>Step</td>
<td>Transaction activity</td>
<td>Locks</td>
<td>Transaction state</td>
<td>Comments</td>
</tr>
<tr>
<td>------</td>
<td>---------------------</td>
<td>-------</td>
<td>-------------------</td>
<td>----------</td>
</tr>
<tr>
<td>ii.</td>
<td>Transaction logic including updates</td>
<td>S</td>
<td></td>
<td>Updates into transaction-private buffer</td>
</tr>
<tr>
<td>iii.</td>
<td>Commit request by user or application</td>
<td></td>
<td>Protected</td>
<td>Similar to phase 1 in two-phase commit</td>
</tr>
<tr>
<td>iv.</td>
<td>Sort required X locks</td>
<td></td>
<td></td>
<td>Reduce probability of deadlock forming</td>
</tr>
<tr>
<td>v.</td>
<td>Lock upgrades</td>
<td>X</td>
<td>Commit decided</td>
<td>Short waits only</td>
</tr>
<tr>
<td>vi.</td>
<td>Early release (weakening) of read-only locks</td>
<td>Weak S</td>
<td>All read-only locks are weak</td>
<td>1st phase of controlled lock violation</td>
</tr>
<tr>
<td>vii.</td>
<td>Update propagation</td>
<td></td>
<td></td>
<td>Also log record creation</td>
</tr>
<tr>
<td>viii.</td>
<td>Commit log record in the log buffer</td>
<td></td>
<td>Position within the equivalent serial schedule</td>
<td></td>
</tr>
<tr>
<td>ix.</td>
<td>Early release (weakening) of all locks</td>
<td>Weak X</td>
<td>All locks are weak</td>
<td>2nd phase of controlled lock violation</td>
</tr>
<tr>
<td>x.</td>
<td>All log records in log on stable storage</td>
<td></td>
<td>Transaction durability</td>
<td></td>
</tr>
<tr>
<td>xi.</td>
<td>Lock release</td>
<td>None</td>
<td></td>
<td>Also release transactions with commit dependencies</td>
</tr>
<tr>
<td>xii.</td>
<td>Transaction end</td>
<td></td>
<td>Complete</td>
<td>Free data structures</td>
</tr>
<tr>
<td>xiii.</td>
<td>Buffer pool writes</td>
<td></td>
<td></td>
<td>Outside of transactions</td>
</tr>
</tbody>
</table>

Figure 15. Commit activities with deferred updates and controlled lock violation.

Figure 15 outlines the steps of a transaction with locking, deferred updates, and controlled lock violation. The end of step v is the high point with respect to two-phase locking. If the transaction reaches this point without conflict, it may commit. Steps ii and x are the only ones with substantial durations. Weakening first the read-only and then all locks (steps vi and ix) is instant, even in transactions with many locks, because it requires merely a simple change in the transaction state, not in each lock instance. Step i allocates an empty update buffer, step ii populates it, and step vii drains it. The transaction holds its exclusive locks strongly only from step v to step ix, with the steps in between very brief. The commit dependencies mentioned in step xi affect read-only transactions that violated a weak X lock. After this technology transfer from optimistic to pessimistic concurrency control, the remaining duration of exclusive locks equals the duration of validation and write phases in optimistic concurrency control.

Implementations may cache intended updates in a transaction-private buffer or create version records in shared persistent storage structures. While transaction-private buffering reduces conflicts with read-only locks of other transactions, it also permits conflicting updates that eventually require transaction rollback. Thus, with respect to concurrency control, transaction-private update buffers are a double-edged sword: less read-write conflicts but more expensive write-write conflicts. However, a transaction-private update buffer can cut the number of log manager requests to a single one.

![Lock compatibility including update locks](image)

Figure 16. Lock compatibility including update locks.

Similar to a transaction using optimistic concurrency control, a transaction using the steps of Figure 15 might expend effort in step ii yet fail to acquire needed exclusive lock in step v. In order to avoid such waste without acquisition of exclusive locks, step ii might acquire “update” locks for its database changes [K 83]. Update locks are compatible with shared locks but not with exclusive locks or other update locks. An alternative, asymmetric definition of update locks permits acquisition of an update lock with concurrent shared locks on the same data item but not of additional shared locks with a concurrent update lock. Figure 16 shows a “?” where these alternatives either permit or deny the lock request.

In fact, this scheme with late lock acquisition is quite similar to optimistic concurrency control with multi-threaded validation using a shared hash table to coordinate concurrently committing transactions. Thus, the proposed scheme using transaction-private update buffers and late lock acquisition reinforces the view that optimistic and pessimistic concurrency control are more similar than different. The remaining differences come from the well-developed theory of hierarchical locks and of update locks, which enable concurrency similar to end-of-transaction validation yet with earlier detection of doomed transactions.
Finally, the commit sequence of Figure 15 is also reminiscent of the ‘verify’ and ‘change’ primitives of IMS/FastPath, which, “publicly introduced in 1976, contain the essential elements of ‘optimistic locking,’ [sic] although the term was then unknown” [GK 85]. The sequence of Figure 15 adds shared locks and upgrade locks during initial execution of the transaction logic as well as lock modes for hierarchical locking, ‘increment’ operations, and orthogonal key-value locking.

5.3 Summary of new techniques

In summary, the similarities of pessimistic concurrency control, optimistic concurrency control with forward validation, and optimistic concurrency control with backward validation suggest or enable a number of technology transfers between them. With all opportunities exploited, the alternative approaches to database concurrency control lose many of their differences and distinctions. In fact, as discussed in the next section, if a database management system supports read-only transactions with version records in the database and read-write transactions with locking and controlled lock violation, then the introduction of delayed lock acquisition converts the system to optimistic concurrency control.

6 Unifying optimistic and pessimistic concurrency control

Careful consideration of Figure 15, i.e., of pessimistic concurrency control with transaction-private update buffers and delayed lock acquisition, reveals a strong similarity to optimistic concurrency control with forward validation. In optimistic concurrency control, if multiple transactions attempt to validate at the same time, they must employ a shared data structure very similar to a lock manager, where they must enter their read and write sets akin to shared and exclusive locks. With multi-version storage, shared locks pertain to the most recent committed version and exclusive locks pertain to the uncommitted version. (Recall that read-only transactions run in snapshot isolation and thus run in serializable transaction isolation without any locks.)

In pessimistic concurrency control as shown in Figure 15, a transaction may release its read-only locks immediately after lock acquisition. A transaction may release its non-read-only locks after (or even incrementally during) update propagation from the transaction-private update buffer to the shared database. If version records in the database (or in page images in the buffer pool) serve the role of the transaction-private update buffer, update propagation is not required and lock release can be very quick.

The rules for optimistic transactions in concurrent forward validation are very similar to those in pessimistic concurrency control. Validation inserts locks into the shared data structure. A transaction may release its read set after validation is complete, i.e., after the transaction has a place in the equivalent serial execution order. A transaction may release its write set immediately after adding a commit log record to the in-memory buffer of the recovery log. It may even release its write set incrementally while propagating updates from the transaction-private update buffer to the database, which includes generation of log records.

As in controlled lock violation, an efficient way to release first the read set and then the write set modifies a transaction status rather than each individual data structure representing entries in the read and write sets. In controlled lock violation, a single status field in the transaction manager indicates whether all locks are strong, all locks are weak, or read-only locks are weak and all other locks are strong. Actual lock release (including space reclamation for lock data structures) occurs, as in traditional implementations, after hardening the transaction by writing a commit log record to stable storage.

6.1 Locks for optimistic concurrency control with forward validation

The above observations suggest an approach for conversion of a database management system using pessimistic concurrency control into one using optimistic concurrency control with forward validation. During execution of a transaction’s logic including its updates, the transaction status is such that lock acquisitions do not check for conflicts and thus cannot fail or delay the transaction. In other words, in spite of using and invoking a lock manager, transactions run as freely as in optimistic concurrency control. Transactions exploit the lock manager to keep their read and write sets but not, at least during execution of the transaction logic, to detect conflicts.

Commit processing strengthens the locks including checks for conflicts with other locks. The step of strengthening locks is equivalent to lock acquisition with conflict detection. Once all locks are strong with no conflicts found, validation is complete. Note that this regimen naturally supports concurrent validation of multiple transactions at the same time, just like a lock manager in pessimistic concurrency control. After validation, transactions first weaken and eventually release their locks as in controlled lock violation for traditional pessimistic concurrency control.

Lock release after concurrent validation in optimistic concurrency control can benefit from the same mechanisms, with the same efficiencies, as controlled lock violation. An optimistic transaction may release its read-only locks immediately after validation and all its locks after its write phase, i.e., after propagating updates from its transaction-private update buffers to database pages or at least their images within the buffer pool. While the transaction hardens by writing its commit log record to stable storage, the transaction’s locks are weak as in controlled lock violation, implemented by advancing the transaction state rather than iterating through all lock data structures.
### Table of Transaction Activities

<table>
<thead>
<tr>
<th>Step</th>
<th>Transaction activity</th>
<th>Locks</th>
<th>Transaction state</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>i.</td>
<td>Transaction begin</td>
<td></td>
<td>Active</td>
<td>Empty transaction-private update buffer</td>
</tr>
<tr>
<td>ii.</td>
<td>Transaction logic including updates</td>
<td>S and X, no checks</td>
<td>Active</td>
<td>Update into transaction-private buffer</td>
</tr>
<tr>
<td>iii.</td>
<td>Commit request by user or application</td>
<td></td>
<td>Protected</td>
<td>Similar to phase 1 in two-phase commit</td>
</tr>
<tr>
<td>iv.</td>
<td>Lock validation</td>
<td>S and X, with checks</td>
<td>Active</td>
<td>Validation as required for optimistic concurrency control</td>
</tr>
<tr>
<td>v.</td>
<td>Early release (weakening) of read-only locks</td>
<td>Weak S</td>
<td>Active</td>
<td>All read-only locks are weak</td>
</tr>
<tr>
<td>vi.</td>
<td>Update propagation</td>
<td></td>
<td>Protected</td>
<td>Also log record creation</td>
</tr>
<tr>
<td>vii.</td>
<td>Commit log record in the log buffer</td>
<td></td>
<td>Protected</td>
<td>Position within the equivalent serial schedule</td>
</tr>
<tr>
<td>viii.</td>
<td>Early release (weakening) of all locks</td>
<td>Weak X</td>
<td>Active</td>
<td>All locks are weak</td>
</tr>
<tr>
<td>ix.</td>
<td>All log records in log on stable storage</td>
<td></td>
<td>Active</td>
<td>Transaction durability</td>
</tr>
<tr>
<td>x.</td>
<td>Lock release</td>
<td>None</td>
<td>Active</td>
<td>Also release transactions with commit dependencies</td>
</tr>
<tr>
<td>xi.</td>
<td>Transaction end</td>
<td></td>
<td>Complete</td>
<td>Free data structures</td>
</tr>
<tr>
<td>xii.</td>
<td>Buffer pool writes</td>
<td></td>
<td>Complete</td>
<td>Outside of transactions</td>
</tr>
</tbody>
</table>

Figure 17. Optimistic forward validation with a lock manager.

Figure 17 summarizes optimistic concurrency control with forward validation using a lock manager, specifically management of read and write sets as locks and end-of-transaction validation by strengthening locks. Not coincidentally and perhaps no longer surprisingly, Figure 17 is quite similar to Figure 15. Step iv is the validation step when a transaction verifies that the locks registered in the lock manager during step ii have no conflict. After the commit is decided, multiple steps end the transaction with respect to isolation and concurrency control, i.e., they weaken and release locks, and with respect to durability, i.e., they write log records and modified database pages.

The similarity to the logic for converting a transaction from optimistic to pessimistic concurrency control (Section 5.1) is striking but not coincidental. The principal difference, of course, is that deferred lock acquisition (or concurrent validation by locking) enables very short lock retention whereas converting a transaction assumes continuation and completion of the transaction in read-write mode and thus holding locks beyond the conversion until transaction commit.

### 6.2 Locks for optimistic concurrency control with backward validation

Optimistic concurrency control with backward validation requires additional changes. As mentioned earlier, e.g., in Table 1 and its discussion, the registration of transactions’ read and write sets must persist long beyond their commits. Thus, a transaction ends (and its data structures cease to exist) after the commit decision of the last concurrent transaction. Put differently, the transaction manager and the lock manager must retain information not only about active and committing transactions but also about fully committed transactions.

An implementation may keep the transaction and its data structures during that time, i.e., past hardening the transaction with a commit log record on stable storage and past acknowledging the successful transaction commit to the user or application. Instead of special data structures for transactions already committed, the standard transaction manager may keep such committed transaction with an appropriate transaction state.

Whenever the oldest active transaction commits, the transaction manager may purge information about committed transactions older than the oldest remaining active transaction, i.e., about transactions committed between the start of newly committed transaction and the start of the oldest remaining active transaction. The required data structures are a timestamp for each transaction, taken at start-of-transaction, and a priority queue tracking the oldest transaction still in its read or validation phase. This priority queue holds one entry for each transaction. Thus, each transaction requires one insertion and one deletion in this priority queue.

The locks acquired and held by committed transactions serve the single purpose of conflict detection during subsequent transaction commits, i.e., validation of read and write sets. Thus, each transactions may register its locks during execution but without conflict tests, retains its locks during validation and other commit logic, and retains them further until the following two events have both occurred. They may occur in either order. First, all old (concurrent) transactions have succeeded or failed their validation. Second, the transaction’s commit log record is safely in the recovery log on stable storage.
During execution of the transaction logic, locks are required but only in a weak mode that does not create conflicts or delays. This weak mode differs from the weak mode in controlled lock violation because conflicting locks do not create commit dependencies. A transaction strengthens its locks and checks for conflicts during its validation phase. The locks remain strong, i.e., able to effect conflicts with other transactions, until the transaction ends.

If, for a given transaction, the last concurrent transaction finishes its validation phase before the given transaction can write its commit log record to stable storage, then standard controlled lock violation applies. Thus, the given transaction’s locks become weak and any other transaction may violate them but incurs a commit dependency. Thus, the other transaction cannot finish its validation phase until the given transaction’s commit log record is safely on stable storage and the committed updates durable even in the case of a system failure including rollback of loser transactions during restart.

<table>
<thead>
<tr>
<th>Step</th>
<th>Transaction activity</th>
<th>Locks</th>
<th>Transaction state</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>i.</td>
<td>Transaction begin</td>
<td>Active</td>
<td>Empty transaction-private update buffer</td>
<td></td>
</tr>
<tr>
<td>ii.</td>
<td>Transaction logic including updates</td>
<td>Registered, no checks</td>
<td>Updates into transaction-private buffer</td>
<td></td>
</tr>
<tr>
<td>iii.</td>
<td>Commit request by user or application</td>
<td>Protected</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iv.</td>
<td>Lock validation</td>
<td>S and X, with checks</td>
<td>Commit decided</td>
<td>Validation as required for optimistic concurrency control</td>
</tr>
<tr>
<td>v.</td>
<td>Update propagation</td>
<td>Also format log records</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vi.</td>
<td>Commit log record in the log buffer</td>
<td>Position within the equivalent serial schedule</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vii.</td>
<td>All concurrent transactions validated or failed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>viii.</td>
<td>Early release (weakening) of all locks</td>
<td>Weak S and X</td>
<td>Controlled lock violation</td>
<td></td>
</tr>
<tr>
<td>ix.</td>
<td>All log records in log on stable storage</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x.</td>
<td>Lock release</td>
<td>None</td>
<td>Also release transactions with commit dependencies</td>
<td></td>
</tr>
<tr>
<td>xi.</td>
<td>Transaction end</td>
<td>Complete</td>
<td>Free data structures</td>
<td></td>
</tr>
<tr>
<td>xii.</td>
<td>Buffer pool writes</td>
<td>Outside of transactions</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 18. Optimistic backward validation with a lock manager.

Figure 18 summarizes optimistic concurrency control with backward validation using a lock manager. In step ii, the transaction registers its read and write sets – the locks do not yet create conflicts or dependencies. From steps iv to viii, the locks are traditional strong locks. The sequence shown in Figure 18 assumes that all concurrent transactions complete their validation before the commit log record reaches the recovery log on stable storage. This is why step viii (controlled lock violation) even exists; if strong locks are required for concurrency control longer than for durability (and dependency tracking), then controlled lock violation does not apply and a transaction ends, including release of all locks and other data structures, when the last concurrent transaction completes its validation or fails.

### 6.3 Summary of unifying optimistic and pessimistic concurrency control

In summary, a moderate software effort converts a traditional implementation of locking into a system that combines optimistic and pessimistic concurrency control. Such a system, with its single and uniformly employed mechanisms, enables a new comparison of optimistic and pessimistic concurrency control that includes the lessons developed in earlier sections of this paper, in particular the importance of locking mechanisms that lock as little as possible and as briefly as possible.

<table>
<thead>
<tr>
<th>Locking Phase</th>
<th>Pessimistic locking (traditional locking)</th>
<th>Optimistic locking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock acquisition</td>
<td>Before each data access</td>
<td>Forward validation</td>
</tr>
<tr>
<td>Lock strengthening</td>
<td>During lock acquisition</td>
<td>Before commit decision</td>
</tr>
<tr>
<td>Conflict detection</td>
<td>During 1st commit</td>
<td>After commit decision</td>
</tr>
<tr>
<td>Lock weakening</td>
<td>After commit request and after allocation of commit LSN</td>
<td>Backward validation</td>
</tr>
</tbody>
</table>

Table 3. Lock management in optimistic and pessimistic concurrency control.
Table 3 summarizes lock management in optimistic and pessimistic concurrency control. In some ways, it mirrors information in Table 1 and its discussion. Background coloring of cells corresponds to that of Table 1. As Gawlick and Kinkade’s expression “optimistic locking” [GK 85] seems to have foreshadowed this unification, the column header uses it rather than the traditional name “optimistic concurrency control.” In the unified implementation, all three concurrency control techniques register their data accesses as they occur. Whereas traditional locks are strong immediately, i.e., bestowing access rights and conflicting with other transactions, optimistic locks strengthen later, either immediately before or immediately after the transaction’s commit decision. Table 1 already covers timing of conflict detection and options for conflict resolution. Locks become weak either after the transaction has a commit LSN (and thus its place in the equivalent serial execution) or after the last concurrent transaction completes its commit decision. As end-of-transaction cannot occur before transaction hardening in all three techniques, Table 3 calls out their differences rather than their commonalities.

7 Conclusions

In summary, traditional row-level locking and key-value locking are not sufficiently precise, most glaringly for phantom protection, which is mandatory for serializability or “repeatable count.” This has created widely held but mistaken impressions about locking and serializability. Alternative locking methods improve the precision and thus may correct these bad impressions. For example, orthogonal key-value locking supports multiple granularities of locking including a distinct key value, a partition of row identifiers (as an approximation of locking a single row identifier), or a gap between key values in an index.

Moreover, optimistic concurrency control detects a conflict only if two transactions overlap with the execution of their transaction logic. Traditional pessimistic concurrency control detects a conflict during a much longer period. More specifically, it retains locks while ensuring transaction durability by writing a commit log record to stable storage. Early lock release and controlled lock violation detect conflicts only if transactions overlap in the execution of their logic, excluding the time to write a commit log record. Even further, optimistic concurrency control checks the overlap of entire execution times, whereas locking considers only the interval from data access to commit, i.e., from lock acquisition to early lock release.

In conclusion, relatively simple improvements in the implementation of locking may render the claimed advantages of optimistic concurrency control over pessimistic concurrency control invalid. With these issues addressed, the advantages of locking over validation, in particular in terms of wasted effort and of conflict resolution options, may render optimistic concurrency control obsolete, at least for concurrency control among database transactions.

The observations above suggest not only some technology transfers (from optimistic to pessimistic concurrency control, and vice versa) but also important new techniques for correct transactional execution of database modifications. Most notable among those are conversion of a read-only transaction in snapshot isolation into a read-write transaction with a traditional, end-of-transaction commit point, local transactions using optimistic concurrency control participating in two-phase commit, and update transactions with deferred updates and thus with extremely short exclusive locks.

Finally, the new unified design for implementing optimistic and pessimistic concurrency control comes back to the starting premises of this research: First, all correct methods for concurrency control must detect the all conflicts and no false conflicts. Second, optimistic and pessimistic concurrency control are more similar than different. Third, their remaining differences are in timing of conflicts and resolution options as well as effort wasted on doomed transactions rather than in the enabled concurrency, performance, and scalability.

References


