Deferred lock enforcement

Weak exclusive locks for more concurrency and more throughput

Goetz Graefe

Abstract

Within a database transaction, the two longest phases execute the application logic and harden the commit on stable storage. Two existing techniques, early lock release and controlled lock violation, eliminate or weaken concurrency control conflicts during hardening. Two new techniques, deferred lock acquisition and deferred lock enforcement, do the same during execution of the application logic. In combination, they minimize the duration of strict exclusive locks, maximize concurrency and throughput, yet ensure full equivalence to serial execution. The new techniques complement multi-version storage, snapshot isolation, distributed transactions, and two-phase commit.

1 Introduction

Decades of unchanging industrial practice in traditional database systems as well as recent focus on optimistic concurrency control for key-value stores and in-memory databases have encouraged the mistaken impression that further improvements in pessimistic concurrency control and locking will not be possible or worthwhile. The present paper aims to convince even the skeptical reader of the opposite.

In fact, there are multiple avenues towards fewer conflicts, more concurrency, and higher throughput. First, lock *scopes* can be fitted better to queries, updates, and other database operations. For example, while phantom protection with any traditional locking method locks at least an entire gap between two index entries, orthogonal key-value locking locks only a small fraction of such a gap. Second, lock *modes* can match precisely the operation at hand. For example, increment locks are widely ignored but precisely match the needs of incremental maintenance of materialized "group by" and "rollup" views, which are common in data warehousing and web analytics. Third, lock *durations* can be optimized for greater concurrency. For example, early lock release and controlled lock violation let new transactions read and modify database items before their latest versions are hardened, i.e., before their (commit) log records reach stable storage. Two new techniques introduced here, deferred lock acquisition and deferred lock enforcement, complement early lock release and controlled lock violation. As in controlled lock violation [GLK 13] and in orthogonal key-value locking [GK 15], the goal here is to avoid false conflicts.

Transaction execution proceeds in multiple phases and commit processing comprises multiple steps. The longest phase in a transaction is either its read phase, i.e., executing the transaction's application logic, or hardening, i.e., writing its commit log record to the recovery log on stable storage. Traditionally, stable storage has been a pair of rotating magnetic disks; today, it means flash storage; and soon, it will be non-volatile memory. Early lock release and controlled lock violation focus on locks, concurrency, and contention during hardening. They may or may not remain important with recovery logs on non-volatile memory. Deferred lock acquisition and deferred lock enforcement focus on locks, concurrency, and contention during execution of transactions' application logic. Their importance will increase with any decrease in the importance of early lock release and controlled lock violation.

Transaction phases → ↓ Techniques	Read phase = transaction and application logic 10^5 inst = 20 µs	Commit logic			Hardening = force log to stable storage 1 I/O = 200 µs
		Commit preparation	Commit log record 10^3 inst = 0.2 µs	Update propagation	
Traditional locking	■	n/a	■	n/a	■
Research goal	░	■	■		░

Figure 1. Lock enforcement durations in traditional locking.

Figure 1 illustrates the phases of a transaction. The long-running phases are shaded in the header row. Dark shading indicates strictly enforced locks: in traditional locking in all phases, from the first data access until the transaction's commit log record is safe on stable storage. Light shading in the bottom row indicates the desired weak lock enforcement during the two longest phases. Instruction counts and durations in Figure 1 are typical for a small database update on today's hardware. The commit logic includes two auxiliary steps needed in both optimistic concurrency control and the new techniques. This diagram omits distributed transactions and two-phase commit. Variants of Figure 1 illustrate the effects of alternative techniques throughout this paper. For example, Figure 40 in Section 7 summarizes how deferred lock enforcement of exclusive locks during the two long-running transaction phases permits transactions to run almost without conflicts yet with full serializability and equivalence to serial execution.

This paper introduces two new techniques, deferred lock acquisition and deferred lock enforcement. The former employs common update locks whereas the latter uses the semantics of two uncommon lock modes, reserved and pending. When combined with multi-version storage and controlled lock violation, the new techniques ensure full serializability yet enforce exclusive locks for only a short moment during commit processing, as set out as research goal in Figure 1. Traditional weak transaction isolation [GLP 76] is also possible but a separate issue.

The origins of deferred lock acquisition and deferred lock enforcement were the thoughts that

- any method of concurrency control must find all true conflicts and should avoid any false conflicts;
- therefore all methods of concurrency control should permit precisely the same concurrency;
- optimistic concurrency control requires but may also benefit from transaction-private update buffers;
- pessimistic concurrency control might also benefit from transaction-private update buffers;
- a transaction with a transaction-private update buffer has no need for traditional exclusive locks while executing its application logic, i.e., during its read phase; and
- transaction-private update buffers are a form of multi-version storage.

In other words, this study aims to offer an attractive combination or integration of locking and multi-version storage. After Section 2 reviews related prior techniques, Sections 3 and 4 introduce deferred lock acquisition and deferred lock enforcement, respectively. Sections 5 covers integration with a variety of other techniques in concurrency control, logging, and recovery; Section 6 focuses on distributed operations including two-phase commit and log shipping; and Section 7 sums up and concludes.

2 Related prior work

2.1 Multi-version storage, multi-version two-phase locking

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

Transaction mode →	Read-only	Read-write
Concurrency control	Snapshot isolation	Optimistic or pessimistic
Commit point	Start-of-transaction	End-of-transaction
Commit cost	None	Commit log record forced to stable storage
Lock requirements (in pessimistic cc)	None	Before each access, until transaction commit
Distributed transactions	Coordination at start-of-transaction	Two-phase commit at end-of-transaction
Ideal storage organization	Multi-version storage	
Version requirements	Many versions	Two versions

Figure 2. Read-only transactions versus read-write transactions.

With multi-version storage, read-only transactions can run in snapshot isolation without transactional concurrency control, e.g., locks or end-of-transaction validation. Only read-write transactions require concurrency control, for both their read and write actions. Therefore, the remainder of this paper focuses on read-write transactions, ignoring read-only transactions. It mostly ignores weak transaction isolation levels, because those only reduce read-only locks whereas the new techniques focus on exclusive locks.

Figure 2 contrasts read-only transactions and read-write transactions. A read-only transaction in snapshot isolation “sees” the database as it existed, in terms of committed database updates, at start-of-transaction, whereas a read-write transaction “freezes” relevant database contents with shared (read-only) locks, isolates its update targets with exclusive locks that permit committing or aborting the transaction without affecting other transactions, and finally commits its updates. An aborting transaction reverses all its updates and then “commits nothing.”

Multi-version storage can benefit both read-only transactions and read-write transactions, albeit in different ways. An active set of read-only transactions may require multiple versions serving a multitude of start-of-transaction timestamps. In contrast, a set of read-write transactions requires only two versions, namely the most recent committed version for all readers and one uncommitted version for the currently active writer and owner of the exclusive lock. A workload mixing read-only and read-write transactions requires multiple committed versions plus one uncommitted version.

Transaction mode ↓	Lock mode	Version access
Read-only	Snapshot isolation, no locks	Read the most recent version committed before the start-of-transaction timestamp
Read-write	Shared lock	Read the most recent committed version
	Exclusive lock	Create, modify, and read an uncommitted version

Figure 3. Transactional data access in versioned databases.

Figure 3 summarizes the rules for choosing versions, e.g., of rows or of index entries. A read-only transaction in snapshot isolation reads the version appropriate for its start-of-transaction timestamp, a version creator reads its own updates, and all other transactions read the most recent committed version. One transaction’s shared lock does not prevent another transaction from creating of a new, uncommitted version but it does prevent commit of a new version. If multi-version storage is used for both read-only and read-write transactions, all new locking techniques must work in this way.

2.2 Transaction-private update buffers

Any implementation of optimistic concurrency control [KR 81] requires transaction-private update buffers. While executing its application logic, an optimistic transaction prepares end-of-transaction validation by gathering sets of reads, writes (updates), insertions, and deletions. While a read-set merely indicates the database contents read, a write-set also indicates the desired new database contents.

The introduction of optimistic concurrency control [KR 81] assumes a transaction-private update buffer holding uncommitted database pages. At the time, page-level locking was common and page-level optimistic concurrency control seemed appropriate. In the 1990s, record-level locking became common in database storage engines [M 90, ML 92, L 93]. A modern transaction-private update buffer holds individual rows, records, and index entries.

As transactions ought to “see” their own updates, a transaction-private update buffer adds overhead to each database read. Instead of simply fetching data values from the shared database and its buffer pool, each read operation within a read-write transaction must inspect its transaction-private update buffer. In-memory indexes and bit vector filters may reduce the effort but some overhead unavoidably remains.

Transaction-private update buffers also require an additional step during commit processing not required in pessimistic concurrency control. Once a commit log record is formatted and has a log sequence number, buffered updates must be propagated from the transaction-private update buffer to the shared database and its buffer pool. Thereafter, the transaction may free its transaction-private update buffer.

Figure 4 sums up the overheads and costs of transaction-private update buffers. Optimistic transactions with end-of-transaction validation must defer all forms of database updates until validation has succeeded. Propagating a transaction’s updates from its transaction-private update buffer to the shared database, at least to the shared global buffer pool, is a distinct step in commit processing. Nonetheless, numerous systems and their benchmark results have proven that optimistic concurrency control and its required transaction-private update buffers can perform well in terms of both single-transaction latency and transaction processing bandwidth.

Figure 5 shows the most traditional form of optimistic concurrency control. Most notably, there is no concurrency control during a transaction’s read phase. In contrast to traditional locking as illustrated in Figure 1, both commit preparation and update propagation are required. Commit preparation means end-of-transaction validation of the read- and write-sets or of timestamps. Update propagation means draining the transaction-private update buffer and applying its contents to the persistent database or at least its buffer pool. It might also include writing log records for those updates. Moreover, there is another period after hardening that is not present in Figure 1. In backward validation, a committing transaction compares its own read- and write-sets with those of transactions already committed [H 84]; therefore, these sets must remain in place until the last concurrent transaction finishes, i.e., possibly long past commit processing and hardening.

Transaction-private update buffers →	Without	With
Database updates	Directly in the database or its buffer pool	Allocate space, save a version or log record
Database search		First search the update buffer, then the database buffer pool
Commit overhead	Write log records	Write log records, apply updates to the database or its buffer pool
Typical context	Locking = pessimistic concurrency control	End-of-transaction validation = optimistic concurrency control

Figure 4. Costs of transaction-private update buffers.

Sets of insertions and deletions, although required in the original design of optimistic concurrency control [KR 81], can be avoided if system transactions provide all space management. System transactions (also known in ARIES as nested top actions) modify the physical representation of database contents but not the logical database contents as observable with queries. For example, insertion and removal, or presence and absence, of a ghost record (also known as pseudo-deleted record) cannot be discerned with a "count(*)" query or any other type of standard query. Another example is splitting a b-tree node – it changes the physical representation but not the logical database contents. With all space management in system transactions, user transactions only modify allocated space. For example, a logical deletion or insertion merely toggles a record's ghost bit and perhaps modifies other fields. With no physical insertions and deletions, a user transaction can capture all its database changes in a write-set within its transaction-private update buffer.

Transaction phases → ↓ Techniques	Read phase = transaction & application logic	Commit logic			Hardening = force log to stable storage	Beyond transaction completion
		Commit preparation	Commit log record	Update propagation		
Backward validation	no cc	■	■	■	■	■

Figure 5. Exclusion enforcement in optimistic concurrency control.

Härder [H 84] anticipated ghost records but at the time dismissed them as impractical: "insertion of an object means its transition from a special null state to some meaningful state and deletion of an object vice versa […] hardly seem to be achievable in practical applications". He also saw transaction-private update buffers as important in optimistic concurrency control but discounted them for transactions in pessimistic concurrency control: "When locking schemes are used, there is no need of private data buffers." Deferred lock acquisition and deferred lock enforcement, the topics of Sections 3 and 4, combine locking with either transaction-private update buffers or multi-version storage.

2.3 Update locks and intention locks

Update locks are an example of lock *modes* that enable more concurrency, less wasted effort, and thus higher transaction processing rates than systems without update locks. Within Korth's theory of derived lock modes [K 83], Gray and Reuter [GR 93] describe update ("U") locks for upgrading from shared ("S") to exclusive ("X") modes. An update lock is a shared lock with a scheduling component: The transaction holding an update lock is first in line to convert from shared access to exclusive access. Only an exclusive lock permits modifying the shared database, e.g., in the shared buffer pool.

Requested → ↓ Held	S	U	X
S (shared)	√	√	–
U (update)	?	–	–
X (exclusive)	–	–	–

Figure 6. Lock compatibility including update locks.

Figure 6 shows the compatibility of these lock modes. Lightly shaded entries reflect the standard lock compatibility of shared and exclusive locks. The darkly shaded entry reflects that only one transaction can be first in line for an exclusive lock. The question mark indicates that some designs do and some do not permit acquisition of a shared lock while another transaction already holds an update lock.

SQLite [SQL 04] locks database files in shared (S) mode while reading and in exclusive (X) mode while applying updates. While computing database updates, it locks affected files with reserved (R) locks. Commit processing holds pending (P) locks while waiting for active readers to finish. While one transaction holds a reserved lock, other transactions can acquire new shared locks as if the reserved lock were a shared lock; while a transaction holds a pending lock, other transactions cannot acquire new locks as if the pending lock were an exclusive lock; yet the transition from reserved to pending is always conflict-free and instantaneous, because their only difference affects only subsequent lock requests.

Requested → ↓ Held	S	R	P	X
S (shared)	√	√		–
R (reserved)	√	–		–
P (pending)	–			
X (exclusive)	–	–		–

Figure 7. Lock compatibility including reserved and pending modes.

Figure 7 shows the compatibility of these lock modes. The darkly shaded field and its adjoining unshaded fields indicate that reserved and pending locks are forms of update locks. These two lock modes resolve the ambiguity indicated with a question mark in Figure 6.

Also within Korth's theory of derived lock modes [K 83], hierarchical and multi-granularity locks [GLP 76] permit a fine granularity of locking for lookups and updates as well as a coarse granularity of locking in large scans. A prototypical example is locking pages in a file, with the possibility of locking an entire file or locking individual pages. Another standard example is locking an entire index for a large query or individual index entries for a small transaction. Locks on individual granules, e.g., pages or index entries, require an "intention" lock on the coarse granule, e.g., the file or index.

Requested → ↓ Held	S	X	IS	IX
S	√	–	√	–
X	–	–	–	–
IS	√	–	√	
IX	–	–		

Figure 8. Compatibility of traditional and intention lock modes.

Figure 8 shows the compatibility of traditional and intention lock modes. The top-left (shaded) quadrant is the standard lock compatibility matrix. Intention locks are always compatible, indicated in the bottom-right quadrant, because a finer granularity of locking will find any actual conflict. The remaining two quadrants are simply copies of the top-left quadrant. Figure 8 omits the traditional combined lock mode S+IX, which is compatible with any lock mode compatible with both S and IX modes.

2.4 Orthogonal key-value locking

Orthogonal key-value locking [GK 15] is an example of lock *sizes* or *scopes* that enable more concurrency and thus higher transaction rates than traditional designs for record-level locking in b-trees.

While both orthogonal key-value locking and ARIES key-value locking [M 90] focus on existing distinct key values in ordered indexes such as b-trees, there are three significant differences. First, the gap (open interval) between two distinct key values has a lock mode separate from (and entirely orthogonal to) the lock for the key value and its set of instances. With separate locks for key and gap, phantom protection does not need to lock any existing index entries or key values. 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 and logically delete it by turning it into a ghost entry. Subsequent transactions may use the ghost for phantom protection, turn it into a valid index entry again, or reclaim the space for another insertion within the same b-tree node.

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.

Third, the set of all possible key values in a gap is hash-partitioned and each partition can have its own lock mode. An equality query with an empty result locks merely a single partition within a gap. Locking a partition achieves serializability yet permits other transactions to insert into the gap. Range queries may lock all partitions within a gap. With this recent refinement not included in earlier descriptions [GK 15], orthogonal key-value locking can lock none, some, or all rows or row identifiers associated with an existing key value plus none, some, or all non-existing key values in a gap between adjacent existing keys.

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

Figure 9. Lock scopes available in orthogonal key-value locking.

Figure 9 illustrates orthogonal key-value locking for distinct key value ‘Jerry’ in a non-unique secondary index containing first names and row identifiers. The column headings indicate points and ranges in the domain of index keys. In addition to the lock scopes shown, orthogonal key-value locking also supports any combination. For example, a range query can, with a single invocation of the lock manager, lock a key value and its adjacent gap. This lock covers all existing and non-existing instances of the locked key value plus all partitions of non-existing key values in the adjacent gap. For comparison, the last line of Figure 9 shows the one and only lock scope possible in ARIES/KVL [M 90], i.e., a gap between distinct key values plus a distinct key value with all actual and possible instances.

Index entries and gaps	entry (Gary, 1)	gap			entry (Jerry, 3)	gap	entry (Jerry, 6)	gap	
		Gary, >1	>Gary, <Jerry	Jerry, <3				Jerry, >6	>Jerry
ARIES/KVL			▨	▨	▨ S on “Jerry”	▨	▨	▨	
ARIES/IM		▨	▨	▨ S on “3”	▨				
KRL		▨	▨	▨ S on “(Jerry, 3)”	▨				
Orth. krl		▨	▨ NS on “(Gary, 1)”	▨					
Orth. kvl			▨ S						
w/ gap part’g			□□▨□						

Figure 10. Required and actual lock scopes in phantom protection for ‘Harry’.

Figure 10 compares lock scopes in earlier methods and in orthogonal key-value locking. More specifically, it shows required and actual lock scopes for an unsuccessful query, i.e., for phantom protection for non-existing key value ‘Harry’. Shading in the header indicates the gap that needs a lock. An S in Figure 10 indicates that a serializable locking technique acquires a transaction-duration shared lock in order to prevent insertion of index value Harry. It is clear that ARIES/KVL locks the largest scope and thus is most likely to suffer from false conflicts. ARIES/IM shows the same range as key-range locking (KRL) only because Figure 10 does not show its lock scope in other indexes of the same table. Orthogonal key-range locking locks less than key-range locking due to separate lock modes for index entry and gap. Lock mode ‘NS’ (pronounced ‘key free, gap shared’) indicates a shared lock on the gap between index entries but no lock on any index entry. Orthogonal key-value locking locks the smallest scope, in particular if the gap and its non-existing key values are partitioned as shown in the bottom row.

Ancillary differences include that orthogonal key-value locking works uniformly in both primary and secondary indexes, whereas ARIES/KVL applies only to secondary indexes. They further 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 that are fully transactional yet inexpensive (no separate execution thread, 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.

Orthogonal key-value locking complements deferred lock acquisition and deferred lock enforcement. All three techniques aim to avoid false conflicts; but orthogonal key-value locking reduces both count and scope of locks whereas the new techniques reduce the durations of strict exclusive locks.

2.5 Early lock release, Controlled lock violation

Early lock release [DKO 84] and controlled lock violation [GLK 13] improve effective lock *durations*, concurrency, and transaction processing rates. Early lock release and controlled lock violation focus on the time when a transaction becomes durable by writing its commit log record to stable storage. Hardening a transaction may take much longer than executing a small transaction over an in-memory database or buffer pool, as is common now in online transaction processing. Logging on non-volatile memory will eventually reverse this imbalance.

Concurrency control and serializability must ensure that many interleaved transactions are equivalent to some serial execution, ideally matching the sequence of commit log records in the recovery log. Once a transaction has a commit log record with an LSN (log sequence number) in the log buffer in memory, the transaction’s position in the equivalent serial execution is final. The need for further concurrency control seems to end at this point. This is the central insight behind early lock release. Variants of early lock release relinquish either only the read-only locks or all locks at this point.

Controlled lock violation is similar to early lock release but also considers durability and write-ahead logging. It ensures that subsequent transactions cannot publish new database updates before the relevant update transactions are durably committed and logged on stable storage. It retains all locks until a commit log record is on stable storage but locks are weak during hardening. Other transactions may acquire conflicting locks, and query or modify data, but cannot complete until the first transaction is durable. In two-phase commit, locks may be weak during both commit phases [GLK 13].

When a transaction T2 requests a lock that conflicts with a weak exclusive lock held by a committing transaction T1, controlled lock violation grants the request but with a commit dependency. In other words, T2 cannot commit until T1 is completely durable. If T2 is a read-write transaction, this has little practical effect, since T2 eventually needs to write its own commit log record.

There are several interesting variants. First, if committing transaction T1 holds a shared lock and T2 acquires an exclusive lock, T2 incurs merely a completion dependency, i.e., T2 can commit even if T1 aborts. Second, if transaction T2 is a read-only participant within a distributed read-write transaction, a commit dependency causes a delay during the pre-commit phase of the two-phase commit, but this delay in the read-only participant is no longer than the delay in read-write participants writing their pre-commit log records to stable storage. Third, if transaction T1 is a participant in a two-phase commit, another transaction T2 may violate T1’s locks while T1 is waiting for the global commit decision but T2 incurs a commit (or completion) dependency on T1.

Figure 11 contrasts controlled lock violation with two forms of early lock release. Controlled lock violation achieves practically the same performance and scalability as early lock release of both S and X locks but without the danger of leaking updates that are not yet durable. Early lock release of only S locks and controlled lock violation are both correct but with substantial differences in performance.

	Early lock release		Controlled lock violation
	S and X	S only	
Trigger	LSN for commit log record		
Performance	Best	Good	Almost best
Consequence of a write-read conflict	Premature publication	Waiting	Commit dependency

Figure 11. Early lock release versus controlled lock violation.

Controlled lock violation requires management of commit dependencies. Hekaton has proven that this is possible with high performance and scalability [DFI 13]. A read barrier holds back result sets in the server until all commit dependencies have cleared. In the published description, it remains unclear whether transaction hardening, i.e., writing a commit log record to the SQL Server recovery log, is considered such a commit dependency.

Figure 12 compares the effective durations of locks in traditional locking and in controlled lock violation. With weak lock enforcement (light shading) during one of the long-running transaction phases and with a commit dependency as only and almost meaningless penalty for violating another transaction's weak locks, controlled lock violation greatly increases concurrency and system throughput [GLK 13].

Transaction phases → ↓ Techniques	Read phase = transaction & application logic	Commit logic			Hardening = force log to stable storage
		Commit preparation	Commit log record	Update propagation	
Traditional locking	■	n/a	■	n/a	■
Controlled lock violation	■		■		▒

Figure 12. Lock enforcement durations with controlled lock violation.

Early lock release and controlled lock violation are complementary to the new techniques, i.e., deferred lock acquisition and deferred lock enforcement. While these prior techniques reduce lock duration at end-of-transaction, the new techniques reduce durations of exclusive locks during a transaction's read phase.

2.6 Summary of related prior work

Multi-version storage and snapshot isolation free read-only transactions from concurrency control, e.g., locking. Transaction-private update buffers enable optimistic concurrency control and are proven in many implementations but they are new to pessimistic concurrency control. Deferred lock acquisition and deferred lock enforcement require transaction-private update buffers or multi-version storage.

Update locks are used in deferred lock acquisition; their refinement, reserved and pending locks, are not used explicitly in deferred lock enforcement but their semantics are. Finally, controlled lock violation relies on weak enforcement of locks during one transaction phase, as does deferred lock enforcement.

3 Deferred lock acquisition

The goal of deferred lock acquisition is to reduce each transaction's "lock footprint" during execution of the transaction's application logic. By deferring acquisition of all exclusive locks to commit processing, it eliminates their conflicts during execution of the transaction's application logic. In essentials, it is similar to two-version two-phase locking (2V2PL) [BHG 87].

A transaction with deferred lock acquisition requests only shared locks until it commits. Thus, execution of the transaction's application logic can rightfully be called the transaction's read phase. Updates including insertions and deletions are cached in a transaction-private update buffer.

While it is sufficient and correct to acquire only shared locks during a transaction's read phase, update locks are recommended for modified database items, e.g., changing key values in b-tree indexes. Update locks can co-exist with shared locks but not with other update locks (on the same database contents). In this way, deferred lock acquisition detects write-write conflicts immediately when they occur.

Transaction phases → ↓ Techniques	Read phase = transaction & application logic	Commit logic			Hardening = force log to stable storage
		Commit preparation	Commit log record	Update propagation	
Deferred lock acquisition	░	▓	▓	▓	▓

Figure 13. Lock enforcement durations with deferred lock acquisition.

Figure 13 illustrates lock enforcement in a transaction with deferred lock acquisition. During the first long-running transaction phase, all transactions acquire locks only in shared and update modes. Light shading indicates that transactions run freely without lock conflicts except for write-write conflicts.

Permitting only a single uncommitted version for each database item at any time ensures a linear history of versions, including no more than a single uncommitted version. In systems with multi-version storage, the uncommitted version can be stored immediately in its final place (and remain locked until transaction commit). This removes the need for transaction-private update buffers from each transaction's state and the need for the update propagation step from the commit logic.

Deferred lock acquisition upgrades locks in a preparation step during commit processing. If a transaction holds multiple update locks that need upgrading to exclusive mode, and if some of these locks must wait for reader transactions to release their locks, then upgrading and waiting occurs one lock at a time. While avoiding serial waiting seems difficult in deferred lock acquisition, deferred lock enforcement naturally waits for all such locks and reader transactions concurrently, as discussed in Section 4.

3.1 Transaction-private update buffers

In the past, transaction-private update buffers were used exclusively in implementations of optimistic concurrency control. In fact, it seems that transaction-private update buffers have been a distinct difference in the implementations of optimistic and pessimistic concurrency control. The present research started with this observation and the question what advantages transaction-private update buffers could enable in the context of pessimistic concurrency control, i.e., locking.

In spite of their overheads, transaction-private update buffers are a required and proven technology that may be transferred from optimistic to pessimistic concurrency control. In this case, exclusive locks are required only when updating the shared database, e.g., in the shared global buffer pool. Updates in the transaction-private update buffer do not require any locks in the database. Therefore, read-write transactions may defer acquisition of their exclusive locks from their read phase to commit processing, i.e., a step immediately before formatting a commit log record in the log buffer.

3.2 Lock timing and semantics

Nonetheless, a read-write transaction needs to lock during its read phase at least the database items it reads. For updates, it seems prudent to acquire an update lock. An update lock requires the same overhead as a shared lock, i.e., an invocation of the lock manager, and it identifies conflicts that doom a transaction, i.e., two transactions modifying the same database item at the same time. If such a conflict is detected, one of the transactions may abort immediately, which is the only conflict resolution strategy available with end-of-transaction validation, or the requesting transactions may wait, preferably with a lock time-out and with limited wait depth [T 98].

Requested → ↓ Held	S	U	X
S (shared)	√	√	–
U (update)	√	–	–
X (exclusive)	–	–	–

Figure 14. Lock compatibility in deferred lock acquisition.

Figure 14 shows lock compatibility for transactions in their read phases with deferred lock acquisition. Compared to Figure 6, it replaces the question mark with a check mark. As such transactions do not acquire exclusive locks until commit processing, Figure 14 could omit its last row and its last column.

If concurrent read-write transactions hold locks on the same data item, all reader transactions must commit first, i.e., the writer must wait when upgrading its update lock to an exclusive lock during commit processing. Deferred lock acquisition delays read-write conflicts until the updater's commit phase.

If commit preparation upgrades multiple locks from update mode to exclusive mode, sorting these requests might seem to prevent deadlocks. Unfortunately, this is not the case. Concurrent writers in commit preparation will not conflict, because write-write conflicts would have been detected earlier when these transactions acquired their update locks, with no further conflicts during their upgrade to exclusive mode. Regarding read-write conflicts, concurrent transactions in their read phases acquire shared locks as they access database items, i.e., they do not sort their lock requests for deadlock prevention. Hence, sorting locks before upgrading them from update mode to exclusive mode will not guarantee deadlock avoidance.

What does seem helpful is a change in the semantics of update locks according to the state of the transaction. While a transaction is in its read phase, i.e., executing application logic, its update locks should not prevent other transactions from acquiring shared locks. Once a transaction begins commit processing, i.e., lock upgrades to exclusive mode, new requests for shared locks should conflict with update locks. This issue is addressed in deferred lock enforcement.

3.3 Integration with multi-version storage

Transaction-private update buffers are a proven technique but nonetheless introduce overheads, including space management and look-aside searches in a transaction's read phase and update propagation in commit processing. Fortunately, multi-version storage combines elegantly with deferred lock acquisition and its early detection of write-write conflicts. Append-only storage such as log-structured merge trees [OCG 96] and partitioned b-trees [G 03] are necessarily multi-version storage.

Concurrent transactions in their read phases conflict if two transactions attempt to modify the same database item. Their update locks conflict and one transaction must abort or wait. If the granularity of versioning is at least as fine as the granularity of locking, the proposed solution works. For example, if both versioning and locking focus on database pages, there can be only one uncommitted page version at a time. The same is true if both versioning and locking focus on key values. The proposed technique does not apply if the granularity of locking is a distinct key value and the granularity of versioning is a database page, or if the granularity of locking is an individual index entry and the granularity of versioning is a distinct key value and its list of index entries.

With only a single uncommitted version at a time, it can be stored immediately, i.e., while still uncommitted, directly at its final place, i.e., where update propagation would move it when copying it from a transaction-private update buffer during commit processing. In other words, read-write transactions require two versions. Of those, one is uncommitted and locked in update mode and the other one is committed and available to all read-write transactions. Read-only transactions may also require older versions. The oldest active transaction in snapshot isolation governs removal of obsolete versions.

With no transaction-private update buffers, database searches access only one place yet satisfy the semantics of Figure 3. A read-only transaction in snapshot isolation reads the version appropriate for its start-of-transaction timestamp. A read-write transaction reads either the last or second-to-last version: a shared lock gives access to the last committed version and an update lock gives access (both for update and retrieval) to the uncommitted version. Thus, an update transaction naturally reads its own updates.

Transaction phases → ↓ Techniques	Read phase = transaction & application logic	Commit logic			Hardening = force log to stable storage
		Commit preparation	Commit log record	Update propagation	
Dla plus multi-version storage	░	▓	▓	n/a	▓

Figure 15. Lock enforcement durations with multi-version storage.

Figure 15 illustrates the effect of multi-version storage and removal of the update propagation step. This effect is more significant in combination with deferred lock enforcement introduced in Section 4.

While it is desirable that each commit succeeds, it is imperative that transaction abort and rollback must never fail. Fortunately, an update lock is sufficient to remove an uncommitted version. In case of transaction abort, there is no need to upgrade a lock from update to exclusive mode, just as there is no need to force the transaction's final log record to stable storage.

3.4 Summary of deferred lock acquisition

In summary, deferred lock acquisition avoids exclusive locks during a transaction’s read phase, i.e., while executing a transaction’s application logic in the first long-running phase of a transaction. Transactions in their read phase can run freely, without lock conflicts or waiting, until they begin commit processing. Only write-write conflicts are detected immediately when they occur, which prevents conflicting updates that cannot possibly both commit.

The absence of conflicting database updates ensures, for each database item, a linear history of versions as well as a single uncommitted version at any point in time. In multi-version storage, this single uncommitted version can be appended to the version chain in the database (or in the shared global buffer pool) while it remains locked and uncommitted. Transaction-private update buffers are not required. The simple rule for version selection mirrors Figure 3: the transaction holding the update lock reads and modifies the uncommitted version, other read-write transactions (holding shared locks) read the last committed version, and read-only transactions (holding no locks at all) read the version appropriate for their start-of-transaction timestamp.

4 Deferred lock enforcement

The goal of deferred lock enforcement is to improve upon deferred lock acquisition. To that end, it eliminates the conversion of update locks to exclusive locks. More significantly, instead of upgrading lock modes one lock at a time, a transaction’s commit preparation step waits concurrently for exclusive access to all database items modified by the transaction.

With deferred lock acquisition, a read-write transaction in its read phase acquires and holds update locks. In contrast, with deferred lock enforcement, a read-write transaction in its read phase acquires exclusive locks but enforces them only in a weak manner. Deferred lock enforcement interprets exclusive locks as reserved locks during a transaction’s read phase, as pending locks while a transaction is preparing for commit, and as traditional exclusive locks only while a transaction is committing. Importantly, the switch from reserved to pending is accomplished by a simple change in the transaction’s state in the transaction manager. All pending locks drain conflicting reader transactions concurrently. The only serial step is verification that all conflicting reader transactions have indeed released their locks.

4.1 Lock semantics and transitions

The enabling difference to deferred lock acquisition is that deferred lock enforcement has no need for update locks. Instead, it employs only shared and exclusive locks. However, the semantics of exclusive locks depend on the transaction’s execution phase. Whether or not a shared lock and an exclusive lock conflict depends on the state of the transaction holding the exclusive lock.

During a transaction’s read phase, its exclusive locks are interpreted and enforced as if they were reserved locks. As shown in Figure 7, a lock in reserved mode is compatible with existing readers (just like any update lock) and it permits new readers as well. Once a transaction begins commit processing, its exclusive locks are interpreted and enforced as if they were pending locks. With that interpretation, existing readers may finish their work and commit but further readers cannot acquire new shared locks. Eventually, when no reader remains, an exclusive lock attains the traditional exclusive semantics and the transaction holding the lock may modify the shared database, e.g., install and commit a new version.

Instead of waiting for exclusive access, a transaction trying to commit also can force active readers to abort. The policy is not prescribed; for example, this decision can consider transaction ages and sizes.

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

Figure 16. Lock compatibility in deferred lock enforcement.

Figure 16 shows the compatibility of lock modes in deferred lock enforcement. Of course, it resembles Figure 7. While one transaction holds a shared lock, another transaction may transition from working (“X as R”, i.e., exclusive locks interpreted as reserved locks) to attempting to commit (“X as P”, i.e., exclusive locks interpreted as pending locks). Once a transaction is attempting to commit (holding an “X as P” lock), no other transaction can acquire a shared lock (or any other lock) on the same data item.

Transaction phases → ↓ Techniques	Read phase = transaction & application logic	Commit logic			Hardening = force log to stable storage
		Commit preparation	Commit log record	Update propagation	
Deferred lock enforcement	░	▓	▓	▓	▓

Figure 17. Lock durations with deferred lock enforcement.

Figure 17 illustrates lock enforcement in deferred lock enforcement. It is rather similar to Figure 13. During the first long-running transaction phase, all transactions acquire shared and exclusive locks, but the exclusive locks have the semantics of reserved locks until commit processing begins. Transactions run with maximal concurrency until their commit is requested.

A transaction acquires locks only during its read phase, i.e., while executing the transaction’s application logic; it does not request locks while executing its commit logic. Thus, in deferred lock enforcement, a transaction’s lock request for an exclusive lock is always processed as if it were a request for a lock in reserved mode, and only the left half of Figure 16 matters for lock requests under deferred lock enforcement.

The transition from reserved to pending semantics does not require any checks of lock compatibility, because the difference between R and P locks affects only future attempts by other transactions to acquire S locks on the same database item. The transition is accomplished by a change in the transaction’s state and takes effect immediately for all X locks. The transition from pending to exclusive semantics occurs automatically when no concurrent reader remains, i.e., concurrent read-write transactions that have read the database item. This transition requires a check that all concurrent readers have released their locks. In single-threaded transaction execution, these checks happen one lock at a time. However, a transaction modifying multiple database items, thus converting multiple locks from pending mode to exclusive mode, drains all readers concurrently from all its exclusive locks. In contrast, deferred lock acquisition waits to upgrade one lock from U mode to X mode and only then attempts to upgrade the next update lock.

In database systems with fine-granularity locking, transactions locking multiple items are as common as tables with indexes. In almost all database management systems, record-level locking focuses on one index at a time and maintenance of secondary indexes requires locks within each affected index. The only exception is ARIES/IM [ML 92], where the scope of a single lock is a logical row including the row’s entries in all secondary indexes as well as (in each secondary index) the gap to an adjacent index entry.

4.2 Integration with controlled lock violation

The principal idea of deferred lock enforcement is interpretation of exclusive locks as weak locks that permit concurrency in some controlled ways and circumstances. The same idea underlies controlled lock violation (Section 2.5). There are some similarities but also a number of differences.

Both deferred lock enforcement and controlled lock violation support shared and exclusive locks. Both techniques enable hierarchical and multi-granularity locking using intention lock modes, and both techniques set lock strength according to the transaction state. One technique may lead to a transaction abort, e.g., after lock request times out or due to limited wait depth, whereas the other one never leads to a transaction abort, only to a commit dependency. Finally, one technique applies to transactions in their read phase, i.e., before a transaction’s commit point, whereas the other one applies to a transaction while hardening, i.e., after the transaction’s commit point. Deferred lock enforcement and controlled lock violation complement each other and use similar ideas, notably temporarily weak locks.

Transaction phases → ↓ Techniques	Read phase = transaction & application logic	Commit logic			Hardening = force log to stable storage
		Commit preparation	Commit log record	Update propagation	
Both dle and clv	░	▓	▓	▓	░

Figure 18. Lock durations with both deferred lock enforcement and controlled lock violation.

Figure 18 illustrates the combined effect of deferred lock enforcement and controlled lock violation. Exclusive locks are effectively removed from both long-running phases of a transaction, i.e., both from the read phase with the transaction's application logic and from hardening by forcing a commit log record to stable storage, with strict enforcement only during (all steps of) commit processing.

4.3 Integration with multi-version storage

Deferred lock enforcement with single-version storage requires transaction-private update buffers, even if early detection of write-write conflicts ensures that any database item can have only one such version at any time. Multi-version storage complements deferred lock enforcement in much the same way as deferred lock acquisition. Again, all append-only storage is automatically multi-version storage and read-only transactions can run in snapshot isolation without any locks. Early detection of write-write conflicts permits omission of transaction-private update buffers. Uncommitted new versions can be placed immediately directly next to prior committed versions. Thus, there is no overhead for transaction-private update buffers, for look-aside search to ensure that read-write transactions "see" their own updates, or for update propagation during commit processing.

Transaction phases → ↓ Techniques	Read phase = transaction & application logic	Commit logic			Hardening = force log to stable storage
		Commit preparation	Commit log record	Update propagation	
Dle, clv, multi-version storage	R	P	X	MV	CLV

Figure 19. Effective lock modes for exclusive locks in deferred lock enforcement.

Figure 19 shows the effective lock modes with multi-version storage in addition to deferred lock enforcement and controlled lock violation. During its read phase, a transaction holds an exclusive lock like a reserved lock ("R") and tolerates both existing and new readers but no other writers (see also Figure 7 and Figure 16). Only when a transaction prepares to commit, an exclusive lock acts like a pending lock ("P"). The transaction tolerates existing readers but no new readers. An exclusive lock has its traditional strength ("X") only for the short period of formatting the commit log record, allocating space in the recovery log, and thus determining a log sequence number. Update propagation is not required with multi-version storage ("MV"). During transaction hardening with controlled lock violation ("CLV"), all exclusive locks are weak again. The committing transaction tolerates new readers and writers at the cost of a completion dependency. New readers and writers must be compatible with each other, of course; with deferred lock enforcement, they always are.

Transaction phases → ↓ Techniques	Read phase = transaction & application logic	Commit logic			Hardening = force log to stable storage
		Commit preparation	Commit log record	Update propagation	
Traditional locking	■	n/a	■	n/a	■
Dle, clv, multi-version storage	▒	■	■	n/a	▒

Figure 20. Lock enforcement durations with three techniques combined.

Figure 20 compares the remaining enforcement period of exclusive locks with traditional locking as shown in Figure 1. The contrast is as stark as can be. For the instruction counts of Figure 1, the duration of strict exclusive locks and thus their opportunity for conflict with shared locks shrink by three orders of magnitude. While traditional locking holds all locks throughout both long transaction phases, the recommended combination of techniques enforces exclusive locks only during two steps of commit processing. If there is no lock conflict during commit processing, then exclusive locks are enforced only for an instant, independently of the size of the transaction. Concurrency during the two long-running transaction phases is practically unrestrained. Thus, the combination of multi-version storage, deferred lock enforcement, and controlled lock violation achieves the research goal of Figure 1.

4.4 Transaction commit logic and state transitions

With some transactions exploiting deferred lock enforcement and perhaps some not, and with some transactions exploiting controlled lock violation and perhaps some not, the complexity of the commit logic might seem daunting. In fact, it is a fairly simple process.

A read-only transaction in snapshot isolation can finish without commit logic. A read-only transaction not in snapshot isolation may have incurred commit dependencies by violating earlier transactions' weak locks. If so, it must wait for those transactions to become durable, i.e., their commit log records in the recovery log on stable storage. Thereafter, a read-only transaction can finish without any additional commit logic.

In order to enable other transactions to exploit controlled lock violation, a committing read-write transaction must indicate when controlled lock violation becomes acceptable, when completion dependencies are resolved, and when commit dependencies are resolved. These two forms of dependencies are resolved at different times, one at the start and one at the end of the committing transaction's final hardening phase. Finally, if the committing transaction has employed deferred lock enforcement, it must switch from weak enforcement of exclusive locks to traditional enforcement, i.e., it must take its exclusive locks from reserved via pending to traditional exclusive semantics.

Figure 21 shows the commit logic for read-write transactions with and without controlled lock violation as well as with and without deferred lock enforcement. Line 2 terminates the transaction's read phase. If the committing transaction has acquired its exclusive locks with deferred lock enforcement, this change in the transaction status changes their semantics from reserved to pending. Therefore, concurrent transactions can no longer acquire shared locks in conflict with the committing transaction's exclusive locks. Line 3 is required only if the committing transaction has acquired exclusive locks with deferred lock enforcement, i.e., if there could be concurrent transactions holding conflicting shared locks. In response to such a conflict, the committing transaction can wait, abort, or abort the other transaction.

```
1.  void TX::commit () {
2.   status = attempting;
3.   if (DLE) check for conflicting S locks
4.   status = committing;
5.   format a commit log record in the log buffer
6.   propagate the commit LSN to newly created versions
7.   status = hardening;
8.   force the commit log record to stable storage
9.   status = durable;
10.  release all S and X locks
11.  status = done;
12. }
```

Figure 21. Commit logic for read-write transactions.

Line 4 indicates that a commit decision has been made and that any exclusive locks no longer have pending but traditional exclusive semantics. However, as the two bottom lines in Figure 7 equal each other, this change in lock semantics does not change any conflicts or concurrency. Moreover, the committing transaction becomes protected. For example, if a high-priority transaction has a lock conflict with the committing transaction, the higher-priority transaction cannot abort the committing transaction and must either wait or abort itself. Line 5 assigns the committing transaction its place in the global sequence of transaction commits. The only remaining purpose of locks and concurrency control is to prevent premature publication of new database updates. Line 6 then propagates this timestamp (LSN) to all new version records.

Line 7 is required only if other transactions have already exploited controlled lock violation when acquiring locks. It ends completion dependencies, i.e., read-write conflicts between the committing transaction as reader and those other transactions as subsequent writers. Line 8 hardens the committing transaction; this is the last phase shown in Figure 20. Line 9 ends commit dependencies, i.e., write-read conflicts between the committing transaction as writer and other transactions exploiting controlled lock violation as subsequent readers. Line 10 ends the committing transaction's concurrency control and line 11 releases any remaining resources, e.g., the transaction descriptor in the transaction manager.

4.5 Summary of deferred lock enforcement

In summary, deferred lock enforcement complements controlled lock violation, which effectively eliminates lock conflicts during hardening, as well as multi-version storage, which eliminates the update propagation step. Among read-write transactions in their read phase, deferred lock enforcement preserves the high concurrency of deferred lock acquisition. Transactions execute their application logic and access database contents freely, without lock conflicts or waiting, until they begin commit processing. Compared to deferred lock acquisition, deferred lock enforcement simplifies lock management by eliminating the conversion of update locks to exclusive locks. More significantly, if a single transaction has updated and locked multiple database items, e.g., a row and its index entries, and if there are concurrent read-write transactions holding S locks on some of these database items, then a transaction with deferred lock enforcement waits concurrently for all these readers to finish and to release their locks. Figure 22 summarizes locks and their semantics in deferred lock acquisition and in deferred lock enforcement.

	Deferred lock acquisition	Deferred lock enforcement
Locks during execution	S, U	S, X
…and their semantics		S, R
Commit preparation	Upgrade U → X, one lock at a time	Concurrent waiting, check each X lock
Locks during commit	S, X	S, X
…and their semantics		S, P → X

Figure 22. Deferred lock acquisition versus deferred lock enforcement.

5 Deferred lock enforcement and other techniques

This section outlines how deferred lock enforcement works when integrated with other software techniques in database concurrency control, logging, and recovery.

5.1 System transactions

System transactions are a mechanism to let user transactions delegate all space management within a database to short critical sections. For example, system transactions split b-tree nodes and remove ghost records, i.e., logically deleted data retained in the database for easy transaction rollback. System transactions do not modify user-visible contents, only its representation. Protected by latches, they only modify in-memory data structures. With no user-visible semantics, system transactions require no transactional locks and their commit log records can linger in the buffer pool until written with subsequent log records. Figure 23, copied from [G 12], contrasts user and system transactions.

	User transactions	System transactions
Invocation source	User requests	System-internal logic
Database effects	Logical database contents	Physical storage structure
Data location	Database or buffer pool	In-memory page images
Parallelism	Multiple threads possible	Single thread
Invocation overhead	New thread	Same thread
Locks	Acquire and retain	Test for conflicts
Commit overhead	Force log to stable storage	No forcing
Logging	Full “redo” and “undo”	Omit “undo” in most cases
Recovery	Backward	Forward or backward

Figure 23. User transactions versus system transactions.

System transactions are the best mechanism for creation and removal of versions. For example, one system transaction may erase obsolete old versions whereas another system transaction creates a new uncommitted version that is initially a copy of the last committed version locked for the invoking user transaction. With no user-visible change in the database, system transactions never hold locks and therefore interact with deferred lock enforcement no more than with controlled lock violation.

5.2 Weak transaction isolation levels

There are two classes of transaction isolation levels weaker than serializability. First, industry-standard snapshot isolation divides each user transaction into two implementation transactions, one that does all database reads in snapshot isolation (as of transaction start) and one that does all database writes like a traditional transaction (committed at transaction end and visible thereafter). Second, traditional weak transaction isolation levels [GLP 76] reduce the scope or the retention period of read-only locks.

	S	X
S	√	–
X	–	–

	S	X
S	√	√
X	–	–

Figure 24. Standard compatibility of shared and exclusive locks.

Figure 25. Lock compatibility approximating "read committed" transaction isolation.

Figure 24 shows the standard lock compatibility matrix, just for reference and comparison. Figure 25 shows a lock compatibility matrix that approximates the effect of "read committed" transaction isolation. It differs from the standard compatibility matrix only in the case that one transaction holds a shared lock and another transaction requests an exclusive lock, i.e., a read-write conflict. In traditional read-committed transaction isolation, a reader acquires a shared lock but retains it only while actually reading the database item. It does not hold the lock until end-of-transaction as required in repeatable-read and serializable transaction isolation. Figure 25 assumes that a reader retains a shared lock but permits a writer to acquire an exclusive lock. (Figure 25 also shows the lock compatibility matrix for an aborting transaction. While a transaction rolls back its database updates, protected by exclusive locks, the transaction's shared locks may remain in the lock manager but other transactions may ignore them.)

Figure 26 shows a lock compatibility matrix that approximates the effect of "dirty read" transaction isolation. It differs from Figure 25 only in the case that a writer holds an exclusive lock and a reader requests a shared lock. Here, the shared lock is granted such that the reader may read "dirty" database contents, i.e., not yet committed. Therefore, Figure 26 and dirty-read transaction isolation detect write-write conflicts but neither read-write conflicts nor write-read conflicts. Note that a writer transaction may modify an item twice; if a reader transaction reads the database item between two such changes, it reads database contents that was never committed, neither before nor after the writer transaction.

	S	X
S	√	√
X	√	–

	S	X as R
S	√	√
X as R	√	–

Figure 26. Lock compatibility approximating "dirty read" transaction isolation.

Figure 27. Lock compatibility of transactions with deferred lock enforcement.

Figure 27 shows the lock compatibility matrix for deferred lock enforcement, with each transaction in its read or working phase. In a transaction's read phase, an exclusive lock is equivalent to a reserved lock, which is compatible with shared locks. Figure 27, i.e., the top-left quadrant of Figure 16, mirrors Figure 26. Thus, transactions with deferred lock enforcement run about as freely as transactions in traditional dirty-read isolation, but deferred lock enforcement ensures serializability whereas dirty-read isolation is far weaker. Deferred lock enforcement achieves serializability by the commit logic of Figure 21 and by enforcing traditional strict semantics of exclusive locks only briefly (lines 2 to 5 in Figure 21).

Transactions running in industry-standard snapshot isolation do not require locks for their reads, just like read-only transaction in snapshot isolation as discussed earlier. Writes and their locks conflict with other writes, as they do in deferred lock enforcement. Put differently, for committing transactions, deferred lock enforcement detects all conflicts and thus ensures serializability; but for transactions in their application logic, deferred lock enforcement eliminates conflicts between reads and writes. Thus, deferred lock enforcement eliminates most concurrency conflicts and achieves most of the performance advantages of industry-standard snapshot isolation over traditional locking. Of course, since industry-standard snapshot isolation splits reads and writes into separate implementation transactions with different commit points, it violates a core premise of transactions, namely that all actions appear atomic, i.e., indivisible. In contrast, transactions with deferred lock enforcement are truly atomic and serializable.

Traditional weak transaction isolation levels reduce scope or duration of shared locks. The difference between serializable and repeatable-read transaction isolation is phantom protection, i.e., transactional guarantees for continued absence, e.g., by shared locks on gaps between existing key values in an ordered index. The difference between repeatable-read transaction isolation and read-committed transaction isolation is lock retention until transaction commit versus lock release immediately after reading a database value. These differences in scope and duration affect only shared locks, whereas the differences between traditional locking and deferred lock enforcement affect strength and duration only of exclusive locks. The choice among weak or strong transaction isolation levels is orthogonal to the employment of deferred lock enforcement. If weak transaction isolation is indeed desired, deferred lock enforcement is compatible with repeatable-read and read-committed transaction isolation.

5.3 Hierarchical locking

The discussion above focuses on shared and exclusive modes but real systems with mixed transaction sizes rely on hierarchical lock scopes and intention locks. In a way, one can even think of the lock scopes in orthogonal key-value locking as a three-level hierarchy with the combination of key and gap as top level, key and gap separately as middle level, and partitions of key or gap as bottom level.

Just as the update lock mode can be split into reserved and pending modes, the intent-to-update ("IU") mode can be split into intent-to-reserved ("IR") and intent-to-pending ("IP") modes. Lock compatibility follows in a straightforward way, i.e., repeating the construction of Figure 8.

Figure 28 shows the compatibility of lock modes including intention, reserved, and pending modes. Intention locks are always compatible, indicated in the bottom-right quadrant, because a finer granularity of locking will find any actual conflict. The top-left quadrant mirrors Figure 7. The remaining two quadrants are simply copies of the top-left quadrant. Figure 28 omits combination lock modes, e.g., S+IX combining the S and IX modes and compatible with all lock modes compatible with both S and IX locks.

Requested → ↓ Held	S	R	P	X	IS	IR	IP	IX
S	√	√		–	√	√		–
R	√	–		–	√	–		–
P	–				–			
X	–	–		–	–	–		–
IS	√	√		–	√			
IR	√	–		–				
IP	–							
IX	–	–		–				

Figure 28. Compatibility of intention, reserved, pending, and intention locks.

Deferred lock enforcement assigns the semantics of reserved and pending locks to exclusive locks during a transaction's longest phases; in addition, it does the same with the respective intention locks.

5.4 Lock escalation and de-escalation

Section 5.3 introduces mechanisms and lock modes for deferred lock enforcement combined with hierarchical locking. A transaction may, during its execution, change its granularity of locking within the hierarchy. For example, even if a transaction starts with record-level locking, it may switch to page-level locking or even index-level locking, typically in order to reduce the count of its locks, its memory requirements in the lock manager, and its lock release effort at end-of-transaction. Lock escalation of this kind is readily supported with deferred lock enforcement, i.e., X locks interpreted as R and P locks during transaction execution and commit preparation. It works precisely as it does with traditional X and IX locks, except that those are interpreted as R, IR, P, and IP lock modes (see also Figure 28).

Inversely, if a transaction employs a coarse granularity of locking, whether from its start or after lock escalation, it can switch to a fine granularity of locking, e.g., to enable more concurrency, to resolve a conflict, or to avoid a deadlock. Lock de-escalation works precisely as with traditional S and X locks, except that X locks are interpreted as R and P locks during transaction execution and commit preparation.

Enabling future lock de-escalation requires tracking fine-granularity locks even if those are not required for concurrency control due to a coarse-granularity lock. This is quite similar to tracking a transaction's read- or write-set as required in optimistic concurrency control. One possible implementation mechanism inserts these fine-granularity locks into the lock manager. Due to the coarse-granularity lock, these fine-granularity locks cannot have conflicts and thus cannot impose waiting. In fact, they do not even require checking for conflicts.

Techniques	Optimistic concurrency control		Lock de-escalation	
	Backward validation	Forward validation	Transaction-private preparation	Preparation within the lock manager
Lock creation	Upon first access			
Lock sharing	End-of-transaction validation	Immediately	Never, or upon lock de-escalation	Immediately
Conflict management	End-of-transaction validation		Conflict prevention by coarse locks	
Conflict resolution	Transaction abort		Fine-lock acquisition, coarse-lock de-escalation	Coarse-lock de-escalation

Figure 29. Lock and conflict management in optimistic concurrency control and in lock de-escalation.

Figure 29 summarizes lock and conflict management using locks in pessimistic concurrency control and, almost equivalently, read- and write-sets in optimistic concurrency control. Optimistic concurrency control detects conflicts during end-of-transaction validation (also see Section 5.11). Backward validation checks the committing transaction against transactions already committed, but forward validation checks against active transactions, which must therefore share there read- and write-sets immediately as those sets grow. In both forms of optimistic concurrency control, the only choice for conflict resolution after validation failure is transaction abort. Pessimistic concurrency control with lock de-escalation acquires a coarse lock and, only if required for conflict resolution, fine locks. The point of similarity is that both optimistic concurrency control with forward validation and locking with on-demand lock de-escalation insert their fine locks into the server's shared data structures immediately upon data access and without danger of conflict. The difference between the methods is that these insertions are tentative and subject to end-of-transaction validation in optimistic concurrency control, whereas in locking with on-demand lock de-escalation, they are final and guaranteed immediately, even without explicit check for conflicts.

5.5 Increment/decrement locks

Whereas deferred lock enforcement is about lock timing and orthogonal key-value locking is about lock sizes, increment/decrement locks are about lock modes, the third dimension for improving locking and concurrency. Increment/decrement locks are more appropriate than exclusive locks for incremental maintenance of materialized views and their indexes, specifically views with a "group by" clause in their definition. Such a view is also known as a roll-up or a cube. In such views and their indexes, ghost records should be indicated not by a ghost bit but by a count equal to zero [GZ 04]. System transactions without locks can create and drop ghost records with a count (and all sums) equal to zero.

	S	D	X
S	√	–	–
D	–	√	–
X	–	–	–

Figure 30. Lock compatibility including increment/decrement locks.

Figure 30 shows the lock compatibility of shared, increment/decrement, and exclusive locks. Since "I" is commonly used for intention locks, increment/decrement locks are indicated with a "D" here. Increment/decrement locks are compatible with one another (the shaded field in Figure 30) but neither with shared nor with exclusive locks. Therefore, increment/decrement locks permit neither reading a value nor setting a value; they only permit incremental, relative maintenance of counts, sums, and moments. This includes incrementing from zero and decrementing to zero.

Increment/decrement locks require their own weakly enforced forms, equivalent to reserved and pending locks. While one transaction holds an increment/decrement lock with weak enforcement, other transactions may read committed data (the last committed version) under a shared lock. However, an increment/decrement update cannot commit while an active reader may require repeatable reads. An earlier design [GZ 04] enforced this limit to committing one increment/decrement transaction at a time (per data item) using commit-time-only exclusive locks. Using transaction state to determine lock semantics seems a cleaner design with multiple advantages, e.g., concurrent waiting.

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

Figure 31. Lock compatibility including increment/decrement locks and their weak modes.

Figure 31 summarizes lock compatibility for increment/decrement and exclusive locks. Their conflicts with shared locks mirror the conflicts of exclusive locks with shared locks (see the row and the column labeled S). Their conflicts with exclusive locks mirror the conflicts of exclusive locks with other exclusive locks (see the three large blocks marked as conflicts). Conflicts among increment/decrement locks are limited to the commit point (see the shaded area), which makes them much more suitable than exclusive locks for incremental updates of summary rows in materialized “group by” views and their indexes.

5.6 Aggregate lock modes

In a database system with only traditional shared and exclusive locks, a lock manager can easily assign an aggregate lock mode to any database item such that a new lock request needs a compatibility test only with the aggregate lock mode but not with each granted lock. Maintenance of the aggregate lock mode is simple when a new lock is granted or an existing lock is released.

Even update locks, intention locks, and increment/decrement locks permit aggregate lock modes. Incremental maintenance of the aggregate lock mode is simple when a new lock is granted but might require more effort when an existing lock is released. For example, if the current aggregate lock mode is IX and a transaction releases its IX lock, the aggregate lock mode must remain IX if another transaction holds an IX lock but it might drop to IS if all other granted locks are IS locks. Counters might be helpful, e.g., counters of transactions holding IX and IS locks on a database item.

Lock semantics that depend on transaction state, e.g., in deferred lock enforcement and controlled lock violation, might seem to render aggregate lock modes useless. In fact, this is not the case for shared locks (S and IS), because such an aggregate lock mode implies that another transaction can acquire a shared lock immediately without checking each granted lock individually. An aggregate exclusive lock (X or IX) does require individual checks, but there usually is only one such lock (or very few – see Section 5.9).

Inasmuch as orthogonal key-value locking and similar techniques permit lock requests to specific parts or partitions of the lockable database item, an aggregate lock mode should reflect the same parts or partitions. For example, if a lock request can specify a lock mode for a key value or for a gap between key values, an aggregate lock mode is required for the key value and another one for the gap.

5.7 Deadlocks

Unfortunately, deferred lock enforcement does not prevent deadlocks. For example:

1. Transaction T1 holds a shared lock on item I1;
2. Transaction T2 holds reserved locks on items I1 and I2;
3. Transaction T2 attempts to commit and upgrades item I1 to pending and item I2 to exclusive;
4. While transaction T2 is waiting on item I1, transaction T1 requests a shared lock on item I2, and thus also waits.

The usual mechanisms and policies for deadlocks apply, e.g., deadlock prevention by limited wait depth and by abort after a timeout. Either policy prevents the deadlock in the example scenario. Limiting the wait depth to 1 would prevent transaction T2 to wait on transaction T1, since transaction T1 is already waiting; with respect to limited wait depth, it does not matter that T1 is waiting specifically on transaction T2. Transaction abort after a timeout prevents a persistent deadlock, albeit only after a delay.

The two policies differ in the options available for managing a concurrency conflict. With limited wait depth, either transaction may abort. If one transaction is attempting to commit and the other one is still acquiring locks, it seems reasonable to abort the latter transaction. On the other hand, if the committing transaction is small (few scans, few locks) but the other transaction is large (complex query, many locks), then it may be less expensive overall to roll back and repeat the transaction attempting to commit. Approximate deadlock detection by timeout does not identify the set of transactions holding conflicting locks. Thus, the only possible policy aborts the transaction that timed out while requesting a lock.

Deadlock	Detection and resolution		Prevention
Mechanism	Waits-for graph	Timeout	Limited wait depth
Victim transaction	Smallest	Waiting	Smallest

Figure 32. Deadlock mechanisms and policies.

Figure 32 shows mechanisms and policies for deadlocks in the context of deferred lock enforcement. Traditional waits-for graphs have high overhead but permit policy-based victim selection in deadlock resolution. In contrast, a timeout in lock acquisition might not even determine what other transaction(s) own conflicting locks. The only viable policy for deadlock resolution is to abort the waiting transaction. Deadlock prevention by limited wait depth avoids forming a cycle in any waits-for graph. Among the transactions waiting for one another, any can be chosen as deadlock victim.

5.8 Transaction rollback

An aborting transaction immediately weakens (effectively releases) its read-only locks. An X lock, once acquired and held, permits not only an initial update but also subsequent updates of the same data item. This includes in particular updating a database item back to its original value, as required for transaction rollback. With an R lock sufficient to create a new (uncommitted) version of a database item and to modify it again within the same transaction, partial and full transaction rollback works with R locks just as additional updates to database items already modified. Locks in reserved mode suffice for removal of uncommitted versions. An aborting transaction never creates new conflicts with any readers, neither with read-only transactions in snapshot isolation nor with read operations of other read-write transactions. *Mutatis mutandis* the same is true for the D locks of Figure 31.

As in traditional transaction rollback, including partial rollback to a transaction save point, after a database item has been updated back to its original value, the transaction may release its exclusive lock on this database item or downgrade it to a shared lock. In other words, after the rollback logic updates a database item back to its original value (even if not its original representation), it may simply release the pertinent R lock. There is no need for an R lock to become a P or X (or D) lock for transaction rollback. While a transaction rolls back holding an exclusive lock with reserved semantics ("X as R"), new readers can acquire shared locks but any new updater is blocked from acquiring an exclusive lock.

5.9 Controlled lock violation during transaction rollback

Most transactions are short; if such a transaction fails, transaction rollback is fast, too. Some long-running transactions can also roll back fast. For example, if creation of a new secondary (redundant) index fails late in the process, it is not required to erase the new index entries one by one; instead, in most systems and circumstances, it is sufficient to reverse the space allocation actions for the new index. Many long-running transactions can speed up both the original execution as well as their rollback (if required) by writing into newly allocated space; this is the principal argument and use case for partitioned b-trees [G 03], for example. If, however, transaction rollback might take a relatively long time, lock retention during rollback might be an inconvenience or even a problem. Incremental lock release during rollback [MHL 92] can ease but not avoid the pain.

In both deferred lock enforcement and controlled lock violation, the semantics of lock modes, in particular their conflicts with other lock modes, depend on the phase or state of the transaction holding the lock. For example, to an exclusive lock held by a transaction in its read phase, deferred lock enforcement assigns the semantics of a reserved lock (see Section 2.3). Even if a transaction holds an exclusive lock, this lock is weak until commit starts. If commit processing never starts because the transaction initiates abort and rollback before it finishes its read phase, such an exclusive lock never needs to attain the traditional strength and strict semantics of exclusive locks.

One can similarly assign specific lock semantics to exclusive locks held by a transaction in aborting state, e.g., in combination with multi-version storage. For exclusive locks held by aborting transactions, the semantics should be that an uncommitted version might require removal but another transaction may acquire a new exclusive lock and create a new version as a copy of the most recent committed version. After multiple failures, a logical data item might have multiple uncommitted versions and thus multiple exclusive locks but all except one of these uncommitted versions must belong to aborting transactions.

This technique can be useful when an individual large transaction aborts and rolls back but also when system restart or media restore rolls back large loser transactions. In the case of an individual transaction rolling back, all its exclusive locks are effectively released immediately, at least as far as other update transactions and lock conflicts are concerned. In case of system restart and media restore, all loser transactions must re-acquire their exclusive locks but new update transactions may create new uncommitted versions without waiting for the loser transactions to complete their rollback.

5.10 Instant recovery

Instant recovery based on write-ahead logging [GGS 16] assumes "physiological" log records for physical "redo" and logical "undo." As long as concurrency control protects and enables transaction rollback, logging and recovery are not affected by lock durations or scopes. For example, multi-version records or index entries are page contents like any other with respect to logging and recovery. Thus, deferred lock enforcement is entirely orthogonal to all techniques in instant recovery based on log-based single-page repair [GK 12]. On-demand single-page repair and on-demand single-transaction rollback enable instant restart, parallel "redo," parallel "undo," self-repairing indexes, single-phase (single-pass) restore, instant backup, instant restore, instant fail-over, and more.

Logging techniques		ARIES	Instant recovery
Backward pointers to log records	In recovery log	By transaction	By database page
	In transaction manager	Most recent log record	All log records
Contents versus representation	Invalid records	Pseudo-deleted records	Ghost records, ghost space
	…their usage	Record deletion	Deletion, insertion, records growing + shrinking
	Representation changes	Nested top actions	System transactions
	…their completion	Dummy commit log record	Commit log record or no separate log record
Logging transaction rollback	Log record	Compensation log record	Rollback log record
	Contents of log record	Original data values	No values, just a pointer
	Pointer within log record	"Next undo" lsn	Original "do" lsn
Log archiving	Single-pass archive logic	Copy & compress	Run generation & indexing, "net change" aggregation

Figure 33. Logging in ARIES and instant recovery.

Figure 33 and Figure 35 summarize how instant recovery differs from ARIES [MHL 92] in logging, log archiving, backup, recovery, and availability. Figure 33 focuses on contents and format of logs and Figure 35 focuses on techniques and performance of recovery. Light shading indicates the required minor differences in the formats of recovery log and log archive that enable major differences in recovery and availability. The format of the recovery log differs in the backward pointers of ordinary update log records and in rollback log records. The format of the log archive differs in sort order and indexing, yet log archiving remains a single pass over the recovery log. In terms commonly used for external merge sort, log archiving performs run generation and recovery performs the final merge step.

These minor differences enable instant recovery techniques from page repair to system restart, media restore, and node failover; as well as backups without load on the database server or its network, and more. For example, if each root-to-leaf traversal in a self-repairing index [GKS 12] can verify all index invariants along its path, single-page repair of any inconsistent node accesses only the minimal set of log records. For another example, instant restart after a system (software) crash enables new transactions and new checkpoints within seconds of restart (after log analysis and lock re-acquisition) whereas traditional recovery as well as ARIES recovery require minutes for the "redo" phase (due to I/O in the database) and possibly hours for the "undo" phase (in case of long-running transactions that require detailed rollback).

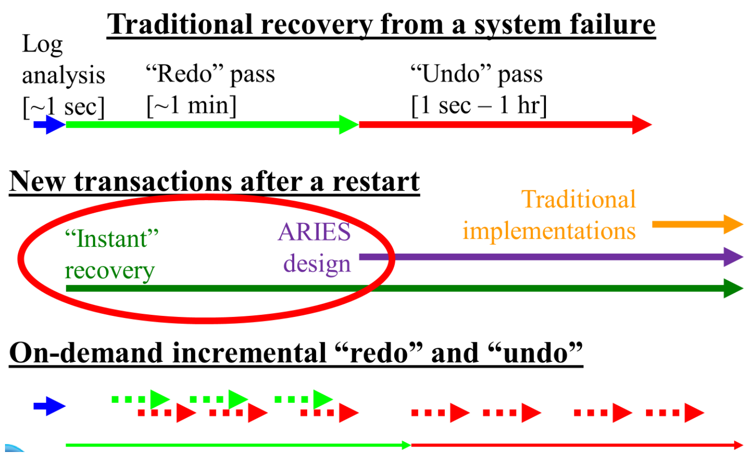
Traditional recovery from a system failure
Log analysis [~1 sec]
"Redo" pass [~1 min]
"Undo" pass [1 sec – 1 hr]
New transactions after a restart
"Instant" recovery
ARIES design
Traditional implementations
On-demand incremental "redo" and "undo"

Figure 34. New transactions and new checkpoints after restart.

Figure 34 compares restart techniques and availability. Traditional recovery permits new transactions and new database checkpoints only after all recovery activities are complete. A variant of ARIES permits new transactions and new checkpoints after the "redo" phase; another variant permits new transactions after log analysis if they do not touch the application's pre-crash working set [M 96]. In contrast, instant restart permits all new transactions and new database checkpoints immediately after log analysis. New transactions trigger incremental page-by-page "redo" of pre-crash database updates as well as incremental transaction-by-transaction "undo" of pre-crash loser transactions. In the background, low-priority threads apply bulk "redo" and "undo," either as in traditional recovery or in parallel by partitioning the sets of in-doubt database pages for "redo" and of loser transactions for "undo." Neither deferred lock enforcement nor controlled lock violation limit or restrict instant recovery.

Recovery techniques		ARIES	Instant recovery
Page failure and repair	Index consistency check	Offline	Continuous, online
	Single-page repair	Backup and log replay	Selective log records
System failure and restart	Log analysis	Forward log scan	Backward log scan
	"Redo" recovery	Forward log scan	Page-by-page, on demand, in parallel
	"Redo" database I/O	Random reads + writes just for recovery	Only for new transactions
	"Undo" recovery	Backward log scan	Transaction-by-transaction, on demand, in parallel
	Lock re-acquisition	During "redo" phase	During log analysis
	First new transaction	After "redo" phase	After log analysis and lock re-acquisition
	First new checkpoint		
Media failure and restore	Backup	Full or incremental copy	Instant, in multiple files; (remote) virtual backup
	Media recovery	Backup copy + log replay	Single-pass restore or instant restore
	First new transaction	After log replay	Instant
	First new checkpoint		
Node failure and failover	Stand-by nodes	Up-to-date database + log	Backup, log archive, log tail
	Failover	Instant	
	Failback, re-integration	Either log replay or full provisioning as new	Instant failback
	First new transaction	After log replay	Instant
	First new checkpoint		

Figure 35. Recovery in ARIES and instant recovery.

During restart, pre-crash loser transactions do not re-acquire read-only locks; thus, there can be no read-write conflict among pre-crash transactions. In the context of deferred lock enforcement and multi-version storage, transaction rollback during restart requires removal of uncommitted new versions. New post-restart transactions may read the last committed version of any database item even before completion of a loser transaction's rollback. Loser transactions merely require R (reserved) locks during restart, or X locks with the semantics of R locks, just as they held before the crash. Note that transaction rollback without a crash similarly proceeds with R locks only, as discussed earlier in Section 5.8.

5.11 Optimistic concurrency control

In optimistic concurrency control, also known as end-of-transaction validation, commit processing requires that each transaction's validation and write phases together are atomic [KR 81]. The write phase must include not only propagation from the transaction-private update buffer to the shared global buffer pool but also hardening the transaction and its commit log record in the recovery log on stable storage. Otherwise, premature publication can occur just as in early lock release [GLK 13].

High-performance database servers cannot tolerate hardening one transaction at a time and therefore require concurrent commit processing for multiple transactions, which in turn requires information sharing among validating transactions and thus a shared data structure rather similar to a traditional lock manager. Lock acquisition, i.e., validation in the parlance of optimistic concurrency control, includes a check for conflicting data accesses by other transactions. Checking mechanisms include version numbers, timestamps, or transactions' read- and write-sets. Lock release occurs after hardening, perhaps with controlled lock violation [GLK 13] during hardening. Many implementations of optimistic concurrency control are equivalent to early lock release (rather than controlled lock violation) without regard to premature publication. If commit processing for optimistic transactions uses a lock manager, it also supports distributed transactions with two-phase commit including controlled lock violation.

The definition of conflicts follows familiar rules: read sets and shared locks do not conflict but all other overlaps do. Put differently, conflicts between optimistic transactions (with read- and write-sets) and pessimistic transaction (with locks, and with or without deferred lock enforcement and controlled lock violation) assume conversion of read- and write-sets into locks and then follow the usual locking rules.

A single system may concurrently execute transactions under optimistic concurrency control and transactions under pessimistic concurrency control, even with deferred lock enforcement. All transactions may employ controlled lock violation. Validation of an optimistic transaction must consider locks in the lock manager, including those held by transactions under pessimistic concurrency control. Commit processing of transactions under pessimistic concurrency control with deferred lock enforcement must consider the read- and write-sets of transactions under optimistic concurrency control.

If indeed end-of-transaction validation in optimistic concurrency control acquires locks and checks whether earlier lock acquisition and retention would have been conflict-free, then optimistic concurrency control with a lock manager for concurrent validation is a form of deferred lock acquisition. If so, how can optimistic concurrency control perform or scale better than pessimistic concurrency control, i.e., locking with deferred lock enforcement and controlled lock violation? It seems that superior performance and scalability require some compromises on correctness, perhaps on phantom protection or on atomic validation and write phases. For example, the atomic phase or critical section starting with the validation phase may fail to include writing the commit log record to stable storage or not even include propagation from the transaction-private update buffer to the persistent database and its shared buffer pool.

Transaction phases → ↓ Techniques	Read phase = transaction & application logic	Commit logic			Hardening = force log to stable storage	Beyond transaction completion
		Commit preparation	Commit log record	Update propagation		
Traditional locking	■	n/a	■	n/a	■	no cc
Backward validation	no cc	■	■	■	■	■
Timestamp validation	▨	■	■	■	▨	no cc
Dle, clv, multi-version storage	▨	■	■	n/a	▨	no cc

Figure 36. Exclusion enforcement in optimistic concurrency control (end-of-transaction validation).

Figure 36 adds two forms of optimistic concurrency control to Figure 20. Periods without concurrency control are marked “no cc.” Backward validation [H 84] is the original design for optimistic concurrency control [KR 81], with each transaction’s read- and write-sets gathered in transaction-private memory and retained beyond transaction commit for the benefit of overlapping transactions validating and committing later. Timestamp validation with concurrent validation also requires locks in a lock manager and can benefit from controlled lock violation during the hardening phase. In other words, timestamp validation can be competitive with the combination of deferred lock enforcement and controlled lock violation.

Three differences remain, however, due to early detection of write-write conflicts in deferred lock enforcement. First, optimistic concurrency control always requires transaction-private update buffers and thus update propagation from those buffers to the shared global buffer pool, whereas deferred lock enforcement can update multi-version storage directly (in the shared buffer pool). Second, optimistic concurrency control wastes effort on doomed transactions due to conflict detection only at end-of-transaction. Third, optimistic concurrency control resolves all conflicts by aborting a transaction that completed its work and is attempting to commit, whereas pessimistic concurrency control permits waiting for a lock, preferably with limited wait depth [T 98] and abort after a timeout.

In some ways, deferred lock enforcement is less restrictive than optimistic concurrency control. In pessimistic concurrency control, lock acquisition occurs while application logic executes and not immediately at start-of-transaction. In contrast, validation of read- and write-sets considers overlap of entire transaction durations, which is equivalent to lock acquisition at start-of-transaction. Timestamp validation is more like lock acquisition because timestamps are inspected and recorded (within a transaction) during data access, not at start-of-transaction.

In other ways, deferred lock enforcement is also more restrictive than optimistic concurrency control. Deferred lock enforcement recognizes write-write conflicts before they occur and it prevents validation failures of reads. Therefore, deferred lock enforcement permits at least as much concurrency as optimistic concurrency control except that it avoids irresolvable conflicts rather than detecting them at end-of-transaction and wasting effort on transactions doomed to fail validation.

5.12 Summary of deferred lock enforcement with other techniques

In summary, deferred lock enforcement readily works with a variety of other techniques for database concurrency control, logging, and recovery. This includes optimistic concurrency control with end-of-transaction lock acquisition as well as recent techniques that optimize lock modes and lock sizes. Deferred lock enforcement is also compatible with traditional recovery techniques such as logged rollback as well as recent recovery techniques such as single-page repair, single-phase restore, and incremental, on-demand, “instant” recovery such as instant restart and instant failover.

6 Distributed operations

Distributed operations mean, most importantly here, multiple local recovery logs, each containing not only log records about local database changes but also transactions’ commit log records.

The discussion below focuses on centralized two-phase commit; it does not consider distributed two-phase commit or non-blocking two-phase commit. Future research may address non-blocking commit protocols, appropriate termination protocols, and their interactions with controlled lock violation.

6.1 Transaction commit logic

During execution of the transaction’s application logic, a local read-write participant in a global transaction may employ R and P locks or an equivalent interpretation of X locks. Locks must become traditional X locks upon the user’s or the application’s commit request, i.e., before the local participant transaction can submit its commit vote to the transaction coordinator.

By default, a transaction must retain its X locks (with full traditional strength) while waiting for the coordinator’s global commit decision. Controlled lock violation weakens lock modes while waiting for the commit decision, with some restrictions and implications [GLK 13]. In general, deferred lock enforcement affects a transaction and its lock modes before its commit process, whereas controlled lock violation affects a transaction during and after the commit logic.

In a distributed system, the commit logic of Figure 21 must integrate two-phase commit. While the local transaction writes its pre-commit log record to stable storage and then communicates with the transaction coordinator, controlled lock violation permits other transactions to conflict with the committing transaction’s locks. If an exclusive lock violates a shared lock, i.e., in case of a read-write conflict, the violating transaction incurs a completion dependency on the committing transaction. If a shared or exclusive lock violates an exclusive lock, i.e., in case of a write-read conflict, the violating transaction incurs a commit dependency on the committing transaction.

```
1.  void TX::commit () {
2.   status = attempting;
3.   if (DLE) check for conflicting S locks
4.   status = committing;
5.   force a pre-commit log record to stable storage
6.   send commit vote to coordinator
7.   receive global commit decision from coordinator
8.   status = decided;
9.   format the final commit log record in the log buffer
10.  propagate the commit LSN to newly created versions
11.  status = hardening;
12.  force the final commit log record to stable storage
13.  status = durable;
14.  release all S and X locks
15.  status = done;
16. }
```

Figure 37. Commit logic for distributed read-write transactions.

Figure 37 shows the commit logic for distributed transactions including the steps required for two-phase commit. It adds the steps in lines 5 to 8 to the commit logic of Figure 21 in Section 4.4.

Line 5 documents in the recovery log that the local transaction will abide by the centralized commit decision. Lines 6 and 7 contribute a vote and await the global commit decision. As both logging and communication can take considerable time, controlled lock violation permits other transactions to violate the committing transaction's locks at the expense of a completion dependency or a commit dependency.

Line 8 indicates that the transaction's final commit decision has been reached and now requires final documentation in the recovery log. The remaining steps are the same as in Figure 21.

Transaction phases → ↓ Techniques	Read phase = transaction & application logic	Commit logic				Hardening = force log to stable storage
		Commit preparation	Two-phase commit	Commit log record	Update propagation	
Traditional locking	▓	n/a	▓	▓	n/a	▓
Controlled lock violation	▓		░	░		░
Deferred lock enforcement	░	▓	▓	▓	▓	▓
Both dle and clv	░	▓	▓	▓	▓	░
Dle, clv, multi-version storage	░	▓	░	░	n/a	░

Figure 38. Lock enforcement durations in distributed transactions.

Figure 38 adds two-phase commit to Figure 20. Lines 5 to 8 of Figure 37 map to the new step within the commit logic, which is shaded in the header as it may take more time than the hardening phase. Traditional locking enforces all locks throughout the transaction's entire duration. Controlled lock violation can become effective as soon as the pre-commit log record has a log sequence number and the transaction has a place in the commit history. Thereafter, other transactions can violate locks while the committing transaction flushes its pre-commit log record to stable storage, votes, waits for a commit decision, and logs it. The combination of deferred lock enforcement and controlled lock violation cannot weaken locks quite so early, because strong exclusive locks are required for update propagation from the transaction-private update buffer to the shared database or its buffer pool. Adding multi-version storage renders transaction-private update buffer and update propagation obsolete; thus, controlled lock violation has full effect and exclusive locks are strictly enforced only for the shortest possible duration, i.e., from receipt of the pre-commit request to assignment of a log sequence number to the pre-commit log record.

The combination of deferred lock enforcement, controlled lock violation, and multi-version storage avoids the transaction-private update buffer and propagation of its contents to the shared global buffer pool. Strict lock enforcement is required only during commit preparation and while formatting a pre-commit log record in the log buffer, i.e., comparable to the strict enforcement of exclusive locks in single-site transactions as shown in Figure 20.

6.2 Log shipping and replication

ARIES-style “physiological” write-ahead logging requires rollback (“compensation”) log records. An aborting transaction updates everything back, logs these updates, and then commits. Since a transaction “commits nothing” after rolling back its updates, i.e., when no logical database changes remain, there is no need to force the commit log record to stable storage. In log shipping from a primary site to a secondary site, the log stream includes either all log records, including rollback log records, or only log records of transactions committed without rollback. The same choice applies to the log archive.

In either design, a primary site executes database transactions, documents their updates in the local recovery log, and ships the log records to each secondary site, which applies them to its copy of the database. A secondary site does not require concurrency control among updates because all required checks already happened on the primary site. Obviously, no concurrency control among read-only queries is required. Ideally, both the primary and the secondary sites employ multi-version storage. With read-only transactions in snapshot isolation never blocking update transactions and read-write transactions never blocking read-only transactions, a secondary site can process its local query workload and its log application workload concurrently without any transactional concurrency control whatsoever.

Transaction mode	Access mode	SI	S	X
Read-only	Snapshot isolation	░		░
Read-write	Reads (S locks)			
	Writes (X locks)	░		▓

Figure 39. Concurrency control conflicts on the recipient site in log shipping.

Figure 39 shows concurrency control within a secondary site or the recipient of log shipping. The fields lightly shaded in Figure 39 cannot lead to conflicts between local queries and log application due to multi-version snapshot isolation of local queries. The field with dark shading could lead to conflicts on a primary site, which would detect them and schedule conflicting transactions as appropriate. The fields without shading do not matter within the recipient of log shipping, because log records received by log shipping, not local database reads, drive all updates in the recipient’s copy of the database.

6.3 Summary of distributed operations

In summary, distributed transactions benefit from the combination of deferred lock enforcement, controlled lock violation, and multi-version storage. Deferred lock enforcement readily integrates with partitioning and log shipping from a primary site (for each partition) to one or multiple secondary sites.

7 Summary and conclusions

With moderate software differences from traditional pessimistic concurrency control, deferred lock acquisition and deferred lock enforcement guarantee serializable transactions but significantly improve concurrency in a transaction’s read phase, i.e., while a transaction executes its application logic. In sharp contrast to traditional locking, the new techniques let concurrent transactions execute their application logic without lock conflicts and thus without waiting for locks, with the exception of write-write conflicts. The traditional strict semantics of exclusive locks are enforced only briefly during commit processing, i.e., for the minimal time that guarantees full equivalence to serial execution in the order of commit log records in the recovery log. While read-only transactions run lock-free in snapshot isolation, the restrictions on read-write transactions are:

- a database item cannot have multiple concurrent updaters (as one of them would need to abort);
- an updater cannot commit with a concurrent active reader (to ensure repeatable-read and -count);
- a new reader must wait (briefly) while an updater commits (to prevent starvation of commits); and

- a reader of updates cannot commit until those updates are durable (to avoid premature publication).

These restrictions apply only in cases with lock conflicts. In a conflict, either transaction can abort or the requester can wait, best with limited wait depth and a timeout. Of course, precise lock modes and scopes avoid many false conflicts. For example, orthogonal key-value locking lets two transactions lock the same key value; one transaction can commit a non-key update of the index entry, e.g., toggling the ghost bit for a logical insertion or deletion, while another transaction ensures phantom protection in an adjacent gap. If truly desired, weak transaction isolation levels reduce shared locks and their conflicts with committing updaters but also weaken correctness guarantees.

Detecting write-write conflicts as early as possible has multiple advantages. First, by finding doomed transactions (that cannot possibly commit) immediately, not at end-of-transaction, it avoids wasting effort on doomed transactions. Second, it enables more conflict resolution options than end-of-transaction validation, e.g., waiting or aborting either transaction in a conflict. Third, it limits each database item to a single uncommitted version at a time. In multi-version storage, which is desirable in any case for read-only transactions in snapshot isolation, this eliminates the overheads of transaction-private update buffers, e.g., look-ups during all reads and update propagation during commit processing.

Deferred lock enforcement complements controlled lock violation: both cut false conflicts to a fraction, both enable more concurrency than traditional locking, and both use weak lock semantics during one of the two long-running phases of read-write transactions; but deferred lock enforcement employs weak locks during a transaction's read phase, i.e., while a transaction executes its application logic, whereas controlled lock violation employs weak locks during hardening, i.e., while a transaction's commit log record goes to the recovery log on stable storage. Deferred lock enforcement also complements multi-version storage: by detecting write-write conflicts as early as possible and ensuring only a single uncommitted version for each database item, deferred lock enforcement permits efficient version storage in shared database structures and avoids the overheads of transaction-private update buffers. Finally, deferred lock enforcement complements non-volatile memory: as low-latency persistent memory replaces external devices for the recovery log, the importance of controlled lock violation decreases and the importance of deferred lock enforcement increases.

Transaction phases → ↓ Techniques	Read phase = transaction & application logic 10^5 inst = 20 µs	Commit logic			Hardening = force log to stable storage 1 I/O = 200 µs
		Commit preparation 10^3 inst = 0.2 µs	Commit log record 10^3 inst = 0.2 µs	Update propagation	
Traditional locking	■	n/a	■	n/a	■
Controlled lock violation	■	n/a	■	n/a	░
Deferred lock enforcement	░	■	■	■	■
Both dle and clv	░	■	■	■	░
Dle, clv, multi-version storage	░	■	■	n/a	░

Figure 40. Lock enforcement durations.

Figure 40 compares techniques from traditional locking to the proposed combination of deferred lock enforcement, controlled lock violation, and multi-version storage. Traditional locking strictly enforces all locks during a transaction's two long-running phases, which indubitably has contributed to giving locking and serializability a reputation for poor concurrency, poor scalability, and poor system performance. In contrast, the combination of three techniques avoids strict locks during both long-running phases. More specifically, controlled lock violation avoids strict locks during hardening, early detection of write-write conflicts and multi-version storage render transaction-private update buffers and update propagation unnecessary, and deferred lock enforcement avoids all conflicts (other than write-write conflicts) among transactions executing their application logic during their read phase. In conclusion, prior techniques plus deferred lock enforcement reduce the effective duration of exclusive locks to the bare minimum: concurrent waiting for truly exclusive access followed by formatting a commit log record.

Acknowledgements

Jeff Naughton suggested multiple improvements in the text. Caetano Sauer gave insightful feedback about deferred lock enforcement and its relationship to optimistic concurrency control.

References

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

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

[CRF 08] Michael J. Cahill, Uwe Röhm, Alan D. Fekete: Serializable isolation for snapshot databases. ACM SIGMOD 2008: 729-738.

[DFI 13] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, Mike Zwilling: Hekaton: SQL Server's memory-optimized OLTP engine. ACM SIGMOD 2013: 1243-1254.

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

[G 03] Goetz Graefe: Sorting and indexing with partitioned b-trees. CIDR 2003.

[G 12] Goetz Graefe: A survey of b-tree logging and recovery techniques. ACM ToDS 37(1): 1:1-1:35 (2012).

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

[G 19a] Goetz Graefe: Avoiding index-navigation deadlocks. In [G 19], 303-319.

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

[GGS 16] Goetz Graefe, Wey Guy, Caetano Sauer: Instant recovery with write-ahead logging: page repair, system restart, media restore, and system failover; second edition. Synthesis Lectures on Data Management, Morgan & Claypool Publishers 2016.

[GK 12] Goetz Graefe, Harumi A. Kuno: Definition, detection, and recovery of single-page failures, a fourth class of database failures. PVLDB 5(7): 646-655 (2012).

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

[GKS 12] Goetz Graefe, Harumi A. Kuno, Bernhard Seeger: Self-diagnosing and self-healing indexes. DBTest 2012: 8.

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

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

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

[GZ 04] Goetz Graefe, Michael J. Zwilling: Transaction support for indexed views. ACM SIGMOD 2004: 323-334. Extended in G. Graefe: Concurrent queries and updates in summary views and their indexes. Hewlett Packard technical report HPL-2011-16.

[H 84] Theo Härder: Observations on optimistic concurrency control schemes. Inf. Syst. 9(2): 111-120 (1984).

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

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

[KR 81] H. T. Kung, John T. Robinson: On optimistic methods for concurrency control. ACM ToDS 6(2), 1981, 221-226.

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

[M 96] C. Mohan: Commit_LSN: a novel and simple method for reducing locking and latching in transaction processing systems. Performance of Concurrency Control Mechanisms in Centralized Database Systems 1996: 307-335.

[MHL 92] C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, Peter M. Schwarz: ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM TODS 17(1): 94-162 (1992).

[ML 92] C. Mohan, Frank E. Levine: ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. ACM SIGMOD 1992: 371-380.
[OCG 96] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, Elizabeth J. O'Neil: The log-structured merge-tree (LSM-tree). Acta Inf. 33(4): 351-385 (1996).
[SQL 04] SQLite: File locking and concurrency in SQLite version 3. 2004. Retrieved August 20, 2019 at sqlite.org/lockingv3.html.
[T 98] Alexander Thomasian: Performance analysis of locking methods with limited wait depth. Perform. Eval. 34(2): 69-89 (1998).

8 Appendix: moving a transaction's commit point

A read-only transaction in snapshot isolation has a start-of-transaction commit point; a read-write transaction with locks has an end-of-transaction commit point. A transaction's commit point can be moved even after the transaction has started and has already completed some work.

8.1 Updates within snapshot isolation transactions

Even if a transaction starts as a read-only transaction in snapshot isolation, the transaction can apply database updates, but only after conversion to a read-write transaction. This conversion shifts the transaction's commit point from start-of-transaction to end-of-transaction. If the conversion is successful, the transaction timestamp assigned at start-of-transaction becomes obsolete. For database reads already completed using that timestamp, deferred lock acquisition is required. Such a conversion requires some preparation as well as multiple steps when the conversion actually occurs.

During the conversion, the transaction must acquire shared locks for database contents read initially in snapshot isolation. Thus, while reading in snapshot isolation, a transaction must track its read set if later it might require a conversion. Tracking may employ the database lock manager and locks in 'snapshot isolation read' (SIread) mode [CRF 08]. Locks in this mode are compatible with all other locks; they are used merely for tracking, not for conflict detection. Figure 41 shows the compatibility matrix.

Requested → ↓ Held	S	X	SI read
S	√	–	√
X	–	–	√
SIread	√	√	√

Figure 41. Lock compatibility with SIread locks.

The conversion logic does not acquire new locks; instead, it upgrades SIread locks to S locks. This upgrade requires absence of incompatible locks, e.g., traditional X locks, but that alone is not sufficient. In addition, this upgrade requires that no update has occurred since the original transaction snapshot (start-of-transaction timestamp). In other words, a version read by the converting transaction must still be the last committed version. In fact, this is the essential requirement for shifting a transaction's commit point from start-of-transaction to end-of-transaction. After all its locks are upgraded, a transaction is successfully converted from a read-only transaction in snapshot isolation to a read-write transaction, and from a start-of-transaction commit point to an end-of-transaction commit point. After a successful conversion, the transaction can acquire additional locks, apply database updates, and eventually commit just as if it had started as a read-write transaction with locking and an end-of-transaction commit point.

If one of its locks fails the upgrade from SIread to S mode, the transaction's conversion fails. In this case, the transaction can abort or continue as a read-only transaction in snapshot isolation. In either case, it releases all its locks.

Note the similarity of the above essential requirement to end-of-transaction validation of a read-only transaction in optimistic concurrency control, in particular end-of-transaction timestamp validation. If multiple transactions perform their validation phase concurrently, they must share information via a shared data structure very similar to a traditional lock manager. This suggests the interpretation of optimistic transaction management as deferred lock acquisition, with immediate commit after success conversion to a lock-based transaction with end-of-transaction commit point.

A few additional points seem worth mentioning. First, SIread locks and upgrades to S locks are required not only after retrieval of existing database contents but also after unsuccessful database searches, i.e., for phantom protection. Thus, the SIread lock mode applies to all lock scopes, e.g., in orthogonal key-value locking. Second, in databases with multi-version storage and deferred lock enforcement, one transaction may convert from snapshot isolation to an end-of-transaction commit point while another transaction is preparing a new version. In other words, a transaction conversion tolerates existing "X as R" locks (see Figure 16). Of course, the converted transaction holding an S lock must finish and release its lock before the updating transaction can commit its new version, as described in Section 4.1. Third, a transaction conversion may acquire an S lock in controlled lock violation, i.e., while a recent update transaction is still hardening its log records. Of course, the converted transaction incurs a commit dependency on the recent update transaction. Fourth, a read-write transaction converted from a read-only transaction in snapshot isolation may employ deferred lock enforcement and permit controlled lock violation just like any other read-write transaction. Fifth, the transaction being converted may be a participant in a distributed transaction with a subsequent two-phase commit. Again, a converted transaction behaves as if it had been a read-write transaction from the start. Finally, transaction conversion applies not only to snapshot isolation transactions but also to "time travel" transactions, i.e., the conversion is oblivious to the age of the transaction's timestamp.

In summary, moving a transaction's commit point is easily possible if and only if all relevant database contents remain unchanged between the two points in time. Specifically, converting a transaction from snapshot isolation to locking requires that for all relevant database contents the last committed version is the same for the transaction's original timestamp and at the time of conversion, lock acquisition, and therefore also at end-of-transaction commit.

8.2 The general case

Converting a transaction from snapshot isolation to locking, and thus its commit point from start-of-transaction to end-of-transaction, is but one example of moving a transaction's commit point. A transaction's commit point can move both forward and backward in time.

In this context, it is important to consider a transaction's commit points with respect to concurrency control and with respect to write-ahead logging. The former is the "high point" in two-phase locking, i.e., after all lock acquisitions and before the first lock release – it determines a transaction's place in the equivalent sequence of transactions in serial execution. The latter is defined by the log sequence number of the transaction's commit log record – it determines a transaction's place in database recovery, e.g., in log replay during media recovery.

In traditional designs, these two commit points are the same or, more importantly, the sequences of transactions' commit points are the same. If the sequences of commit points differ, i.e., if concurrency control defines one commit sequence and write-ahead logging defines a different one, it is impossible to define or verify the semantics of snapshot isolation, of point-in-time recovery, or of "time travel" queries (snapshot isolation with a commit point earlier than start-of-transaction).

Write-ahead logging and recovery have no significance, of course, for read-only transactions. A read-only transaction can shift its commit point from start-of-transaction to end-of-transaction but also from end-of-transaction to start-of-transaction or to any other point in time. As in the conversion of a read-only transaction in snapshot isolation to a read-write transaction with locking, the essential requirement is that all relevant database contents must not have changed in the meantime.

This recommends an alternative perspective on the commit logic in optimistic concurrency control. If concurrent validation of multiple optimistic transactions shares information using a data structure analogous to a traditional lock manager, end-of-transaction validation amounts to deferred lock acquisition. End-of-transaction validation, deferred lock acquisition of shared locks, and transaction conversion from snapshot isolation to locking all require the same tests or conditions. Whereas transaction conversion checks for intermediate updates between start-of-transaction and conversion, deferred lock acquisition in optimistic concurrency control checks for intermediate updates between a transaction's first access to an item in the transaction's read-set and end-of-transaction validation.

8.3 Converting from locking to snapshot isolation

The opposite conversion, from locking with end-of-transaction commit point to read-only in snapshot isolation, is particularly simple if the desired snapshot time equals the time of conversion. This conversion requires, of course, that the transaction does not hold any exclusive locks, only shared locks. After verifying this condition, the transaction can take a snapshot and release its locks. There is no need to retain any information, e.g., a read set, except if a subsequent conversion back to locking might be desired. The time of the snapshot becomes the transaction's new commit point, replacing the original end-of-transaction commit point typical for transactions with locking.

In fact, any desired snapshot time after the most recent lock acquisition is similarly simple. Any earlier timestamp, e.g., start-of-transaction, is more difficult, because another transaction might have modified and committed a relevant database value just before the most recent lock acquisition, even one that the current transaction has not touched or locked yet.

9 Appendix: distributed snapshot isolation transactions

In a distributed read-only transaction, if it is known from the start which nodes in a network are required for the transaction, the equivalent of a two-phase commit at start-of-transaction can establish a shared point in time and an equivalent local timestamp in each node. Neither locking nor end-of-transaction validation nor any other final commit logic is required.

If additional nodes are wanted beyond the initial set, the following options seem available:

1. Prohibit growing the set of participant nodes – either finish with the initial set of participant nodes or abort.
2. Abort and restart the query and the transaction with the required larger set of participant nodes and, of course, a new snapshot timestamp.
3. If supported, map the start-of-transaction timestamp to a wall clock time on the additional nodes, or to very short time intervals. This approach works best if supported by special hardware and software with the precision of atomic clocks.
4. Convert the entire transaction, i.e., each participant transaction, from start-of-transaction commit to end-of-transaction commit, and from snapshot isolation to locking. During and after this conversion, the transaction acquires locks and retains them until two-phase commit at end-of-transaction. Section 8.1 describes the conversion conditions and the required logic including lock acquisition. Lock acquisition might fail and the transaction must abort.
5. Re-run the equivalent of two-phase commit to establish a new timestamp shared by the existing and the additional participant transactions. This step converts the entire transaction, i.e., each existing participant transaction, from the existing start-of-transaction timestamp to the new timestamp just established. The required conditions and logic are a variant of conversion from snapshot isolation to locking. After this conversion, neither locking nor end-of-transaction validation nor any other final commit logic is required.

A variation of technique 5 above performs all checks at end-of-transaction, even for a transaction that grows its set of participants multiple times. Each time, and in particular at the last such time, the transaction runs the equivalent of two-phase commit to establish a new timestamp shared by all participant transactions. Local checks can be omitted at that time but must be invoked at end-of-transaction before the transaction and its results can be considered confirmed and committed. This variation saves repeated checking within a transaction that grows repeatedly. Deferring the equivalent of two-phase commit to the end-of-transaction is, however, not different from a traditional two-phase commit; it therefore seems to defeat the purpose of snapshot isolation with minimal constraints on concurrency.

If most distributed queries (and read-only transactions) can pre-declare their needs, and if technique 5 above is correct and reasonable for the remainder, then there is a new alternative for supporting distributed snapshot isolation without complex and expensive hardware and software.

Note: One way to think about an adjustment of local timestamps in a distributed transaction is local conversion to locking, two-phase commit, and then local conversion back to snapshot isolation. The "equivalent of two-phase commit" follows the same idea and process.

10 Appendix: details of commit on multi-version storage

10.1 End-of-transaction timestamps as version tags

There is a circular dependency between commit LSN and version tags. If each version (record, index entry) is tagged with the timestamp (commit point, commit LSN) of the transaction that created it, and if that timestamp is not known until the commit log record exists in the log buffer, and if the commit log record is a transaction's last log record, and if updates in a version's tag must be logged (for "redo" during recovery from system or media failures), then the only solution is to log all tag updates within a transaction's commit log record, even if a transaction has created new versions for hundreds or thousands of index entries.

An alternative approach modifies version tags lazily, e.g., as side effect of subsequent transactions or of defragmentation and reorganization. This approach requires that such subsequent read-write transactions can determine whether a version is committed. The presence of a newer version or the absence of a lock in the lock manager readily answers this question. Read-only transactions in snapshot isolation, however, also need to determine when a version was committed, i.e., the timestamp (commit LSN) of the transaction that created it. Thus, a mapping is required from transaction identifiers to commit LSNs. Note that this mapping is required only for recently committed transactions but not for those committed before the start-of-transaction timestamp of the oldest active transaction in snapshot isolation.

Instead of inventing a new data structure for tracking the commit times of recently completed transactions, the in-memory b-tree can be used. In this design, newly created versions retain transaction identifiers, even after the transaction has committed. For efficient look-up by subsequent transactions, one part (key range, partition) of the in-memory b-tree maps transaction identifiers to commit LSN values. Each transaction's commit log record includes the update (creation) information for that one data record mapping transaction identifier to commit LSN. Note that a traditional commit log record already carries the essential information, transaction identifier and commit LSN; it merely lacks "redo" information such as a page identifier.

Upon reboot, this mapping information is re-initialized (empty), assuming that no read-only transactions in snapshot isolation survive the reboot. If a snapshot isolation transaction attempts to find a mapping from transaction identifier to commit timestamp yet no such entry is found, it is safe to assume that the transaction committed before the snapshot isolation transaction started. In steady-state operation, entries can be removed for transactions that committed before the oldest active snapshot isolation transaction started.

Either we support time travel even to timestamps before the last reboot, which requires that mapping information survives reboot, **or** we don't, which means that there is no need to log or recover mapping information.

10.2 Reducing the count of version tags

Commit timestamps attached to individual index entries imply a fair bit of effort during commit processing. One mitigation strategy focuses on lock management: if locks are attached to index entries and b-tree nodes and buffer pool pages, then lock release must revisit those places anyway and thus maintain the proper commit timestamps with relatively little additional effort. On the other hand, it may be possible to reduce the commit effort even further.

If a b-tree sorted on one column also contains a correlated column, branch nodes may add minimum and maximum values of that other column to parent-to-child pointers, also known as zone filters. If minimum and maximum values are equal, index entries in leaf nodes may even omit values of that other column. This idea could be applied to commit timestamps, in particular within a partitioned b-tree when a large transaction creates a new partition with many index entries. In other words, there might be a single version tag (or very few version tags) in or near the b-tree's root node, enabling very efficient commit logic.

11 Appendix: details of abort logic and transaction states

A transaction still in “working” or “attempting” state (see Figure 21) can still abort. In such a case, the transaction state changes to “aborting.” Read-only locks become weak immediately such that other transactions can freely violate them (see comments on Figure 25). Early lock release would also be correct for read-only locks (see Section 2.5). For all other (i.e., non-read-only or read-write) locks, transaction rollback can release locks incrementally, but care is required in a transaction that modified a single database item repeatedly, with multiple log records but only a single lock; original log records may indicate whether a lock was acquired during forward processing.

With incremental lock release, controlled violation of exclusive locks is not possible. In the absence of incremental lock release, controlled lock violation could be enabled one lock at a time. However, any complex logic is frowned upon within recovery, where software defects are particularly harmful. Only after formatting the commit log record does controlled lock violation seem perfectly sensible, even if this “commit nothing” log record of an aborted transaction is not forced to stable storage. Instead of “hardening,” “aborted” or “rolled back” seems a more appropriate name for this transaction state.

12 Appendix: early lock acquisition in secondary indexes

The Introduction (Section 1) suggests three avenues towards fewer conflicts, more concurrency, and higher throughput: lock scopes, lock modes, and timing of lock acquisition, enforcement, and release. A related fourth technique focuses on the lock acquisition sequence [G 19a].

Query execution plans in relational databases often search a secondary index and then fetch data from a table's primary storage structure. In contrast, database updates usually modify first a table's primary storage structure and then all affected secondary indexes. Deadlocks are likely if queries acquire locks first in the secondary index and then in the primary storage structure while updates acquire locks first in the primary storage structure and then in secondary indexes. The proposed alternative uses the traditional, opposing navigation directions in queries and updates but the same locking sequence in both queries and updates. This design retains the efficiency of both queries and updates but avoids deadlocks caused by the traditional, opposing locking sequences.

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

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

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

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

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

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

The following examples illustrate some query and update operations.

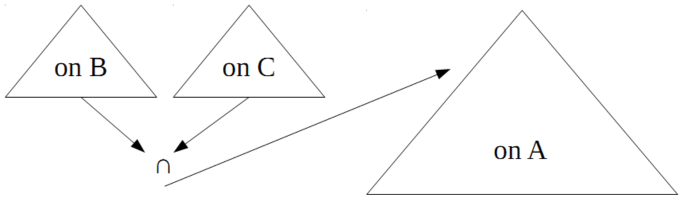

Figure 42. Query execution.

Figure 42 illustrates a query execution plan searching two secondary indexes of the same table, intersecting the obtained sets of row identifiers, and then fetching rows from the table's primary index. The primary index here is sorted or keyed on column A, the secondary indexes on columns B and C. Neither scan in a secondary index can reliably predict which records or key values in the primary index need locking. Thus, the recommended lock acquisition sequence is the same as the processing sequence: locks in each secondary index followed by locks in the primary index, but only on those index entries accessed by the query execution plan.

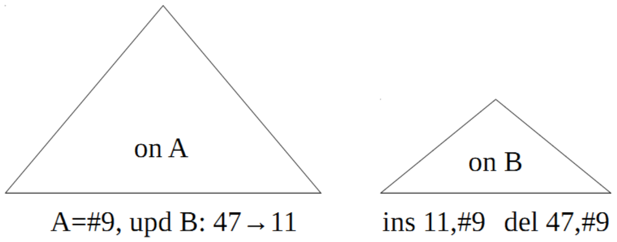

Figure 43. Ordinary update.

Figure 43 illustrates an update to an indexed column other than a row identifier. In the row with primary key A=9, column B is modified from 47 to 11. In the table's primary index, this is a non-key update. In the secondary index on column C, no changes and no locks are required. In the secondary index on column B, this update modifies a search key, which requires a deletion and an insertion. If the insertion merely turns an existing ghost record into a valid record, even if a system transaction just created that ghost record on behalf of the user transaction, then the user transaction can predict reliably and precisely all required locks. – The traditional lock acquisition sequence follows the update logic from the primary index to the secondary index. The recommended sequence first acquires a latch on the affected page in the primary index, then predicts all required locks in all indexes, acquires locks in the secondary indexes (i.e., for insertion and deletion in the index on B), only then acquires a lock in the primary index, applies updates in the primary index, releases the latch, and finally proceeds to the secondary index.

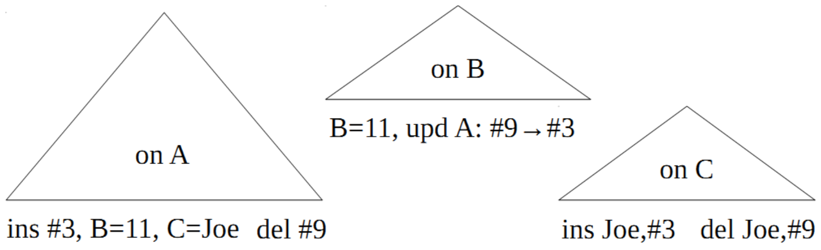

Figure 44. Row identifier update.

Figure 44 illustrates an update of row identifier A (from value #9 to value #3), which requires a deletion and an insertion in the table's primary index. A change in a row's row identifier requires an update in each secondary index. If a user-defined key for a secondary index is unique, a change in a row identifier is a non-key update in this secondary index. Otherwise, the row identifier is part of the sort key in the index and a change in a row identifier requires a deletion and an insertion in that secondary index. (In the example, there might be another index entry with C=Joe and A between #3 and #9.) Figure 44 shows both the unique index on column B and the non-unique index on column C. – In all cases, the user transaction can request all required locks before applying its deletions, updates, and insertions in the index storage structures. This is similar to the example of Figure 43 but with some differences. The principal difference is that there are two modification actions in the table's primary storage structure, one deletion and one insertion. Among several possible designs, one is to attach all lock acquisitions for deletions to the deletion in the primary index, and similarly for all insertions.

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

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

With respect to the number of locks and of lock manager invocations, the proposed method is neutral – it neither adds nor saves any locks. The same is true for the size or scope of locks. For example, if a traditional update locks an index entry but no adjacent gap, then the proposed update logic does the same.

In the context of deferred lock enforcement, it is not clear that the proposed lock acquisition sequence is required or even beneficial. Deferred lock enforcement avoids exclusive locks and deadlocks during transactions' working phase and upgrades all locks instantly and at the same time, both in the upgrade from reserved to pending lock semantics and in the upgrade from pending to exclusive lock semantics. Moreover, a transaction waits for all its strictly exclusive locks concurrently. In other words, it might be that modern high-performance transaction processing servers require either deferred lock enforcement or the proposed lock acquisition sequence, but not both. On the other hand, the two techniques do not interfere with each other.

13 Appendix: auxiliary material

Requested → ↓ Held	S	X	IS	IX	S+IX
S	√	–	√	–	–
X	–	–	–	–	–
IS	√	–	√		√
IX	–	–			–
S+IX	–	–	√	–	–

Figure 45. Compatibility of traditional and intention lock modes.

Figure 45 shows the same lock compatibility matrix as Figure 8 plus the traditional S+IX lock mode. One transaction's lock is compatible with another transaction's S+IX lock if the first transaction's lock is compatible with both an S lock and an IX lock in the other transaction. For example, one transaction may scan an entire index and modify individual keys (S+IX on the entire index plus X on individual keys) and another transaction can read individual keys (IS on the entire index plus S on individual keys).

Employee						SalaryTotalPerDept		
EmpNo	Name	…	Salary	Dept		Dept	Count	SalaryTotal
…								
17	Terry		49	Shoes				
23	Gary		41	Shoes	→	Shoes	3	132
47	Mary		42	Shoes				
…								
34	Jerry		43	Tools	→	Tools	1	43

Figure 46. A base table and a summary view.

Figure 46 shows a trivial example of a table and a "group by" view over the table. If the view is materialized and indexed, insertions, deletions, and updates in the base table must also modify the index(es) of the summary view. Two update transactions may have no concurrency conflict in the base table yet affect the same summary row, e.g., salary updates for two employees in the Shoes department. Increment/decrement locks permit modifying counts and sums in a summary view without conflict. This is what is shown in Figure 30.

Like exclusive locks in deferred lock enforcement, increment/decrement locks have different semantics and conflicts during a transaction's read phase, its commit logic, and its hardening phase. Whereas many transactions in their read phase can increment or decrement the same value, only one such transaction can be in its commit logic and install a new value (or, in multi-version storage, a new version) in the database, i.e., in the shared buffer pool or on persistent storage. Once a transaction finishes its commit logic and enters its hardening phase, the next transaction with an increment/decrement lock (on the same summary row with counts or sums, or index entries of the summary row) can enter its own commit logic. This is what is shown in Figure 31.

The aggregation functions "min" and "max" permit incremental maintenance during base table insertions but not during base table deletions. However, "lower bound" and "upper bound" do not have this limitation, and neither does a reference count for bounding values, e.g., an indication whether a lower bound is currently the minimum existing value. A further refinement also maintains 2nd-to-minimum and 2nd-to-maximum.

Read phase = transaction and application logic 10^5 inst = 20 µs	Commit logic			Hardening = force log to stable storage 1 I/O = 200 µs
	Commit preparation and decision	Commit log record 10^3 inst = 0.2 µs	Write phase = update propagation	

Figure 47. Transaction execution phases and typical durations.

Transaction phases → ↓ Techniques	Read phase = transaction & application logic 10^5 inst = 20 µs	Commit logic			Hardening = force log to stable storage 1 I/O = 200 µs
		Commit preparation 10^3 inst = 0.2 µs	Commit log record 10^3 inst = 0.2 µs	Update propagation	
Traditional locking	■	n/a	■	n/a	■
Dle, clv, multi-version storage	░	■	■	n/a	░

Figure 48. Lock enforcement durations.

Figure 48 compares traditional lock durations with the combination of deferred lock enforcement (during a transaction's read phase), controlled lock violation (during a transaction's hardening phase), and multi-version storage (avoiding transaction-private update buffers and their overheads). Whereas a traditional transaction holds exclusive locks for about 220 µs (in this example, which is deemed typical), this combination of techniques requires exclusive locks with traditional strict enforcement for only 0.4 µs, an improvement by orders of magnitude. Put differently, whereas traditional lock durations detect mostly false conflicts, the new combination of techniques avoids practically all false conflicts and false deadlocks, detects all actual conflicts, and combines strict transactional correctness with maximal concurrency and system throughput. Further improvements can be achieved by combining optimal lock durations as shown in Figure 48 with minimal lock sizes as shown in Figure 9 and Figure 10 (Section 2.4).

Violator's context → ↓ Violator's mode	Local transaction	Participant in a distributed transaction
Read-only in snapshot isolation	No locks, no lock violations, no delay	
Read-only with end-of-transaction commit	Delay up to flushing a commit log record	Delay up to flushing a pre-commit log record
Read-write	No delay beyond flushing a commit log record	No delay beyond flushing a pre-commit log record

Figure 49. Commit delays due to controlled lock violation.

Figure 49 summarizes the delays imposed upon transactions benefiting from controlled lock violation. Note that a read-only transaction in snapshot isolation does not acquire any locks and controlled lock violation does not apply. If the violating transaction is a local read-write transaction, controlled lock violation does not impose any delay; the violating transaction writing its own commit log record enforces the required timing. The same is true if the violating transaction is a read-write participant in a distributed transaction writing a pre-commit log record. Controlled lock violation imposes a delay only if the violating transaction is a read-only transaction with end-of-transaction commit point, i.e., a transaction with local locks yet without local log records.

If the violating transaction is a local transaction, controlled lock violation imposes a delay at most as long as writing a commit log record, because the delay is controlled by the violated transaction writing its commit log record. In other words, in this unusual case of a local read-only transaction not in snapshot isolation, its delay is no longer than the commit process of a local read-write transaction.

If the violating transaction is a read-only participant in a distributed read-write transaction that uses two-phase commit at end-of-transaction, controlled lock violation imposes a delay at most as long as writing a commit log record, because again the delay is controlled by the violated transaction writing its commit log record. Typically, this delay is just as long as other participants need to write their pre-commit log records. A longer delay is possible if the violated transaction is a distributed read-write transaction with its own two-phase commit.

Scale → ↓ History	Focused	Broad, e.g., database-wide
Repeat	▨	“Redo” log scan
Rewind	Transaction rollback	“Undo” log scan

Figure 50. Recovery functionality before single-page repair.

Figure 50 classifies the standard recovery activities in traditional database implementations. With “physiological” logging, “redo” is physical and “undo” is logical. “Redo” guarantees the same contents in the same database pages (even if not equal byte for byte) whereas “undo” effectively “updates back,” which may or may not apply to the same database page and may leave implementation artifacts such as ghost records. Importantly, the traditional standard recovery techniques do not include a focused repeating of history (shaded field).

The need for this functionality is demonstrated by its implementation in successful products. For example, Microsoft SQL Server supports ‘page-level restore,’ which recovers individual database pages from database backup and recovery log. With its full scans of database backup and recovery log (and the implied poor performance and scalability), it is the standard restore logic constrained to a very small subset of database pages. SQL Server ‘page-level restore’ does not exploit the per-page chain of log records in the SQL Server recovery log, which is used only for consistency checking in log shipping and in database restore operations. Incidentally, SQL Server uses focused “undo” in restart after a system failure – instead of the backward log scan of the ARIES design [MHL 92], restart in SQL Server rolls back loser transactions guided by the per-transaction chains of log records within the pre-crash recovery log.

Scale → ↓ History	Focused	Broad, e.g., database-wide
Repeat	Single-page repair	“Redo” log scan
Rewind	Transaction rollback	“Undo” log scan

Figure 51. Recovery functionality including single-page repair.

Figure 51 illustrates how single-page repair fills the gap left by traditional standard recovery techniques. By complementing focused “undo” with focused “redo,” single-page failures and single-page repair enable incremental execution of all database recovery actions and thus a variety of instant recovery techniques (see Section 5.10).