Chapter 4

Concurrency control for main memory databases

A database system optimized for in-memory storage can support much higher transaction rates than current systems. However, standard concurrency control methods used today do not scale to the high transaction rates achievable by such systems. In this chapter we introduce two efficient concurrency control methods specifically designed for main-memory databases. Both use multiversioning to isolate read-only transactions from updates but differ in how atomicity is ensured: one is optimistic and one is pessimistic. To avoid expensive context switching, transactions never block during normal processing but they may have to wait before commit to ensure correct serialization ordering. We also implemented a main-memory optimized version of single-version locking. Experimental results show that while single-version locking works well when transactions are short and contention is low performance degrades under more demanding conditions. The multiversion schemes have higher overhead but are much less sensitive to hotspots and the presence of long-running transactions.

4.1 Introduction

Current database management systems were designed assuming that data would reside on disk. However, memory prices continue to decline; over the last 30 years they have been dropping by a factor of 10 every 5 years. The latest Oracle Exadata X2-8 system ships with 2TB of main memory and it is likely that we will see commodity servers with multiple terabytes of main memory within a few years. On such systems the majority of OLTP databases will fit entirely in memory, and even the largest OLTP databases will keep the active working set in memory, leaving only cold, infrequently accessed data on external storage.

A DBMS optimized for in-memory storage and running on a many-core processor can support very high transaction rates. Efficiently ensuring isolation between concurrently executing transactions becomes challenging in such an environment. Current DBMSs typically rely on locking but in a traditional implementation with a separate lock manager the lock manager becomes a bottleneck at high transaction rates as shown in experiments by Johnson et al. [52]. Long read-only transactions are also problematic as readers may block writers.

In this chapter, we investigate what are the appropriate high-performance concurrency control mechanisms for memory-resident OLTP workloads. We found that traditional single-version locking is "fragile": It works well when all transactions are short and there are no hotspots but performance degrades rapidly under high contention or when the workload includes even a single long transaction.

Decades of research has shown that multiversion concurrency control (MVCC) methods are more robust and perform well for a broad range of workloads. This led us to investigate how to construct MVCC mechanisms optimized for main memory settings. We designed two MVCC mechanisms: the first is optimistic and relies on validation, while the second one is pessimistic and relies on locking. The two schemes are mutually compatible in the sense that optimistic and pessimistic transactions can be mixed and access the same database concurrently. We systematically explored and evaluated these methods, providing an extensive experimental evaluation of the pros and cons of each approach. The experiments confirmed that MVCC methods are indeed more robust than single-version locking.

This chapter makes three contributions. First, we propose an optimistic MVCC method designed specifically for memory resident data. Second, we redesign two locking-based concurrency control methods, one single-version and one multiversion, to fully exploit a main-memory setting. Third, we evaluate the effectiveness of these three different concurrency control methods for different workloads. The insights from this study are directly applicable to high-performance main memory databases: single-version locking performs well only when transactions are short and contention is low; higher contention or workloads including some long transactions favor the multiversion methods; and the optimistic method performs better than the pessimistic method.

The rest of this chapter is organized as follows. Section 4.2 gives a brief overview of related work. Section 4.3 covers preliminaries of multiversioning and describes how version visibility and updatability are determined based on version timestamps. The optimistic scheme and the pessimistic scheme are described in Sections 4.4 and 4.5, respectively. Section 4.6 reports performance results, and Section 4.7 offers concluding remarks.

4.2 Related work

Concurrency control has a long and rich history going back to the beginning of database systems. Several excellent surveys and books on concurrency control are available [13, 40, 55, 77].

Multiversion concurrency control methods also have a long history. Chapter 5 in [13] describes three multiversioning methods: multiversion timestamp ordering (MVTO), two-version two-phase locking (2V2PL), and a multiversion mixed method. 2V2PL uses at most two versions: last committed and updated uncommitted. They also sketch a generalization that allows multiple uncommitted versions and readers are allowed to read uncommitted versions. The mixed method uses MVTO for read-only transactions and Strict 2PL for update transactions.

The optimistic approach to concurrency control originated with Kung and Robinson [56], but they only considered single-version databases. Many multiversion concurrency control schemes have been proposed [2, 14, 15, 18, 19, 41, 57, 67], but we are aware of only two that take an optimistic approach: Multiversion Serial Validation (MVSV) by Carey [20, 21] and Multiversion Parallel Validation (MVPV) by Agrawal et al. [1]. While the two schemes are optimistic and multiversion, they differ significant from our scheme. Their isolation level is repeatable read; other isolation levels are not discussed. MVSV does validation serially so validation quickly becomes a bottleneck. MVPV does validation in parallel but installing updates after validation is done serially. In comparison, the only critical section in our method is acquiring timestamps; everything else is done in parallel. Acquiring a timestamp is a single instruction (an atomic increment) so the critical section is extremely short.

Snapshot isolation (SI) [12] is a multiversioning scheme used by many database systems. Several database management systems support snapshot isolation to isolate read-only transactions from updaters: Oracle, PostgreSQL and SQL Server [61] and possibly others. However, SI is not serializable and many papers have considered under what circumstances SI is serializable or how to make it serializable. Cahill et al. [19] published a complete and practical solution in 2009. Their technique requires that transactions check for read-write dependencies. Their implementation uses a standard lock manager and transactions acquire “locks” and check for read-write dependencies on every read and write. The “locks” are nonblocking and used only to detect read-write dependencies. Whether their approach can be implemented efficiently for a main-memory DBMS is an open question. Techniques such as validating by checking repeatability of reads and predicates have already been used in the past [17].

Oracle TimesTen [64], IBM's solidDB [47] and SAP HANA [31] are three commercially available main-memory DBMSs. TimesTen uses single-version locking with multiple lock types (shared, exclusive, update) and multiple granularities (row, table, database). For main-memory tables, solidDB also uses single-version locking with multiple lock types (shared, exclusive, update) and two granularities (row, table). For disk-based tables, solidDB supports both optimistic and pessimistic concurrency control. HANA has a transaction manager that supports ACID transactions and is based on multi-version concurrency control. We are not aware of any published information on how this multi-version concurrency control method is implemented, and whether it is based on locking or validation.

Main-memory concurrency control kernels frequently implement a superset of the functionality offered by software transactional memory [73] implementations, as they add support for durability and application-specific isolation requirements for each transaction. Hardware transactional memory [45] has been proposed for better performance, and Tran et al. [78] have explored how to use hardware transactional memory to execute entire transactions. The emergance of hardware transactional memory support in mainstream Intel and IBM processors allows designers to simplify the logic and the underlying data structures in light of concurrent modifications, and is an interesting area for future work.

4.3 Multi-version storage engine

A transaction is by definition serializable if its reads and writes logically occur as of the same time. The simplest and most widely used MVCC method is snapshot isolation (SI). Snapshot isolation does not guarantee serializability because reads and writes logically occur at different times: reads occur at the beginning of the transaction and writes at the end. However, a transaction is serializable if we can guarantee that it would see exactly the same data if all its reads were repeated at the end of the transaction.

To ensure that a transaction T is serializable we must guarantee that the following two properties hold:

Read stability: If T reads some version V1 of a record during its processing, we must guarantee that V1 is still the version visible to T as of the end of the transaction, that is, V1 has not been replaced by another committed version V2. This can be implemented either by read locking V1 to prevent updates or by validating that V1 has not been updated before commit. This ensures that nothing has disappeared from the view.

Phantom avoidance: We must also guarantee that the transaction's scans would not return additional new versions. This can be implemented in two ways: by locking the scanned part of an index/table or by rescanning to check for new versions before commit. This ensures that nothing has been added to the view.

Lower isolation levels are easier to support:

1. For repeatable read, we only need to guarantee read stability.

2. For read committed, no locking or validation is required; always read the latest committed version.

3. For snapshot isolation, no locking or validation is required; always read as of the beginning of the transaction.

We have implemented a prototype main-memory storage engine. We begin with a high-level overview of how data is stored, how reads and updates are handled, and how it is determined what versions are visible to a reader, and that a version can be updated.

4.3.1 Version format

In a multi-version scheme, a record version is visible only to transactions whose logical read time is within a particular timestamp interval. Storing the timestamp range of this interval is sufficient when there is no update activity in the system. However, when a transaction is performing an update, the precise visibility interval is not known. In those cases, we need to transiently store a unique transaction identifier that corresponds to the transaction that modified the visibility interval.

In our prototype, we keep track of two additional fields in each database record to determine whether a version is visible or not to transactions. In addition to the user-defined columns, each record contains a `Begin` and an `End` field:

Begin A field that stores the earliest logical read time this version is visible to transactions. This is a 64-bit field and consists of:

1. A flag (1 bit) indicating whether a transaction is currently changing the beginning of the visibility interval.
2. If the 1-bit flag is set, the remaining 63 bits contain a unique *transaction identifier* that corresponds to the transaction changing the visibility of this version. If the 1-bit flag is unset, the remaining 63 bits store a *timestamp*, which indicates the earliest logical read time this version is visible to transactions.

End A field that stores the latest logical read time this version is visible to transactions. This is also a 64-bit field and consists of:

1. A flag (1 bit) indicating whether a transaction is currently changing the end of the visibility interval.
2. If the 1-bit flag is set, the remaining 63 bits contain a unique *transaction identifier* that corresponds to the transaction changing the visibility of this version. If the 1-bit flag is unset, the remaining 63 bits store a *timestamp*, which indicates the latest logical read time this version is visible to transactions.

Both fields are 64 bits wide so as to be changed atomically with the atomic compare-and-swap instruction provided by the underlying hardware. This eliminates the need to acquire latches to read or change the visibility of a version and greatly improves concurrency.

4.3.2 Storage and indexing

Our prototype currently supports only hash indexes which are implemented using lock-free hash tables. A table can have many indexes, and records are always accessed via an index lookup; there is no direct access to a record without going through an index. (To scan a table, one simply scans all buckets of any index on the table.) The techniques presented here are general and can be applied to ordered indexes implemented by trees or skip lists. Figure 4.1 shows a simple bank account table containing six versions. Ignore the numbers (100) in red for now. The table has two (user) columns: Name and Amount. Each version has a valid time defined by timestamps stored in the `Begin` and `End` fields. A table can have several indexes, each one with a separate (hash) pointer field. The example table has one index on the `Name` column. For simplicity we assume that the hash function just returns the first letter of the name. Versions that hash to the same bucket are linked together using the `HashPtr` field.

Hash bucket J contains four records: three versions for John and one version for Jane. The order in which the records appear is immaterial.

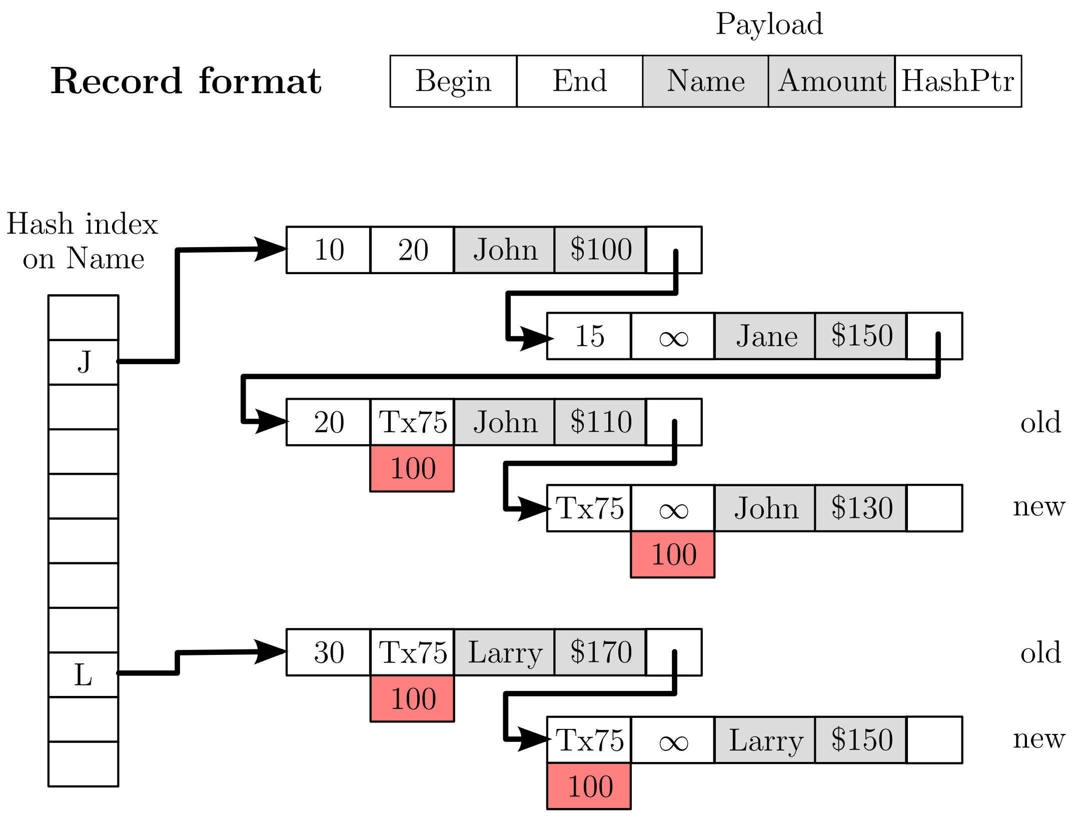

Figure 4.1: Example account table with one hash index. Transaction 75 has transferred $20 from Larry's account to John's account but has not yet committed.

Jane's single version (Jane, 150) has a valid time from 15 to infinity meaning that it was created by a transaction that committed at time 15 and it is still valid. John's oldest version (John, 100) was valid from time 10 to time 20 when it was updated. The update created a new version (John, 110) that initially had a valid time of 20 to infinity. We will discuss John's last version (John, 130) in a moment.

4.3.3 Reads

Every read specifies a logical (as-of) read time and only versions whose valid time overlaps the read time are visible to the read; all other versions are ignored. Different versions of the same record have non-overlapping valid times so at most one version of a record is visible to a read. A lookup for John, for example, would be handled by a scan of bucket J that checks every version in the bucket but returns only the one whose valid time overlaps the read time.

4.3.4 Updates

Bucket L contains two records which both belong to Larry. Transaction 75 is in the process of transferring $20 from Larry's account to John's account. It has created the new versions for Larry (Larry, 150) and for John (John, 130) and inserted them into the appropriate buckets in the index.

Note that transaction 75 has stored its transaction identifier in the `Begin` and `End` fields of the new and old versions, respectively. A transaction identifier stored in the `End` field serves as a write lock and prevents other transactions from updating the same version and it identifies which transaction has updated it. A transaction identifier stored in the `Begin` field informs readers that the version may not yet be committed and it identifies which transaction owns the version.

Now suppose transaction 75 commits with validation timestamp 100. It then returns to the old and new versions and sets the `Begin` and `End` fields, respectively, to 100. The final values are shown in red below the old and new versions. The old version (John, 110) now has the valid time 20 to 100 and the new version (John, 130) has a valid time from 100 to infinity.

Every update creates a new version so we need to discard old versions that are no longer needed to avoid filling up memory. A version can be discarded when it is no longer visible to any transaction. In our prototype, once a version has been identified as garbage, collection is handled cooperatively by all threads. Although garbage collection is efficient and fully parallelizable, keeping track of the old versions does consume some processor resources.

4.3.5 Transaction phases

A transaction can be in one of four states: Active, Preparing, Committed, or Aborted. Figure 4.2 shows the possible transitions between these states.

A transaction has two unique timestamps: (1) a *start timestamp* reflects the logical time the transaction started executing, and (2) a *validation timestamp* reflects the final serialization order of this transaction with respect to all other transactions in the system. The timestamp is acquired by atomically reading and incrementing a single, global, and monotonically increasing counter.

Property 4.1. *All timestamps are assigned by atomically reading and incrementing a global counter.*

Property 4.2. *Every transaction has a unique start timestamp ST and a unique validating timestamp VT, with $ST < VT$.*

A transaction goes through three different phases. We outline the processing in each phase only briefly here; it is fleshed out in more detail in connection with each concurrency control method in Sections 4.4 and 4.5.

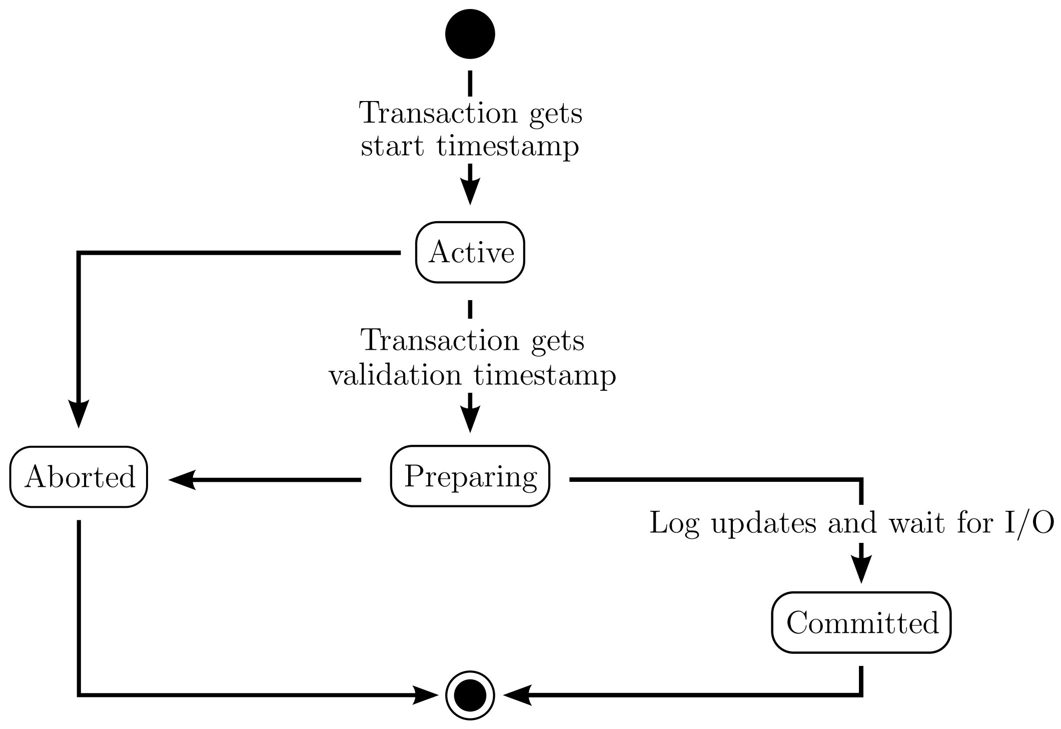

Figure 4.2: State transitions for each transaction.

1. The transaction is created; it acquires a unique *start timestamp* and sets its state to Active.

2. **Normal processing phase.** The transaction does all its normal processing during this phase. A transaction never blocks during this phase. For update operations, the transaction copies its transaction identifier into the `Begin` field of the new versions and into the `End` field of the old or deleted versions. If it aborts, it changes its state to Aborted and skips directly to step 4. When the transaction has completed its normal processing and requests to commit, it acquires a unique *validation timestamp* and switches to the Preparing state.

3. **Preparation phase.** During this phase the transaction determines whether it can commit or is forced to abort. If it has to abort, it switches its state to Aborted and continues to the next phase. If it is ready to commit, it writes information about all its new versions and deleted versions to a redo log record and waits for the log record to reach non-volatile storage. The transaction then switches its state to Committed.

4. **Postprocessing phase.** If the transaction has committed, it proceeds to replace its transaction identifier with its validation timestamp from both the `Begin` field of the new versions and from the `End` field of the old or deleted versions. If the transaction has aborted, it marks all its new versions as garbage and discards them immediately.

5. The transaction is terminated. The old versions are assigned to the garbage collector, which is responsible for discarding them when they are no longer needed.

4.3.6 Version visibility

A read must specify a logical read time under multiversioning. Only versions whose valid time overlaps the logical read time are visible to the read. The read time can be any value between the transaction's start timestamp and the current time. Which read time is chosen depends on the concurrency control method used and the transaction's isolation level; more about this in Sections 4.4 and 4.5.

While determining the visibility of a version is straightforward in principle, it is more complicated in practice as we do not want a transaction to block (wait) at any time during normal processing. Recall that a version's `Begin` or `End` fields can temporarily store a transaction identifier, if the version is being updated. If a reader encounters such a version, determining visibility without blocking requires checking another transaction's state and validation timestamp and potentially even restricting the serialization order of transactions. (We discuss this mechanism in detail in Section 4.3.8.) Table 4.1 summarizes the four possible cases.

We now examine each case in turn, beginning from the easiest and most common case where both fields contain a timestamp. We then discuss the cases where the `Begin` or `End` fields contain transaction identifiers.

Both `Begin` and `End` fields contain timestamps

In the absence of concurrent updates, both the `Begin` and `End` fields of a version contain timestamps. In this common case, a simple comparison suffices to determine whether the version is visible. Let RT denote the logical read time being used by transaction T. To determine whether version V is visible to T, we check V's `Begin` and `End` fields. If both fields contain timestamps, V is visible to T if and only if RT falls between the two timestamps.

`Begin` field contains a transaction identifier

When reading a version shortly after its creation, its `Begin` field may transiently contain a transaction identifier and not a timestamp. Suppose transaction T reads version V and finds that V's `Begin` field contains the identifier of a transaction TB. Whether version V is visible depends on transaction TB's state and TB's validation timestamp. For example, TB may have committed already but not yet finalized the `Begin` fields of its new versions. If so, V is a committed version with a well-defined `Begin` timestamp. Table 4.2 summarizes the cases that may occur and the action to take depending on the state of the transaction TB.

If transaction TB is still in the Active state, the version is uncommitted and thus is not visible to any other transaction. (If T = TB, T can still see its own updates.) If TB has updated a record multiple times, only the latest version is visible to it.

V's `Begin` field is	V's `End` field is	Outcome
Timestamp	Timestamp	V is visible if and only if `Begin` < RT < `End`.
Transaction identifier	Timestamp	Visibility depends on the state of the transaction whose identifier is stored in the `Begin` field. (See Table 4.2 for the analysis.) If needed, proceed to check whether RT < `End` to determine visibility.
Timestamp	Transaction identifier	If RT < `Begin`, V is not visible. Otherwise, visibility depends on the state of the transaction whose identifier is stored in the `End` field. (See Table 4.3 for the analysis.)
Transaction identifier	Transaction identifier	Visibility depends on the state of the transactions whose identifiers are stored in the `Begin` and `End` fields. First, follow the case analysis shown in Table 4.2. Continue, if needed, with the analysis shown in Table 4.3 to determine visibility.

Table 4.1: Case analysis of action to take when transaction T checks visibility of version V as of logical read time RT.

If TB's state is	Action to take when transaction T checks visibility of version V.
Active	If T = TB, treat V's `Begin` field as if it contains a timestamp of zero when testing for visibility. Otherwise, V is not visible.
Preparing	T *speculatively reads* V using TB's validation timestamp as V's `Begin` field when testing for visibility.
Committed	Use TB's validation timestamp as V's `Begin` field to test visibility.
Aborted	Ignore V.
Terminated or TB not found	Check visibility of V again.

Table 4.2: Case analysis of action to take when transaction T checks visibility of version V, and the `Begin` field of version V contains the identifier of transaction TB.

If TB is in the Preparing state, is is unknown whether TB will eventually commit or abort. A safe approach in this situation would be to have transaction T wait until transaction TB either commits or aborts. However, we want to avoid all blocking during normal processing, so instead T *speculates* that TB will commit. If TB commits, V's `Begin` timestamp field will contain TB's validation timestamp. T can therefore speculatively use TB's validation timestamp as V's `Begin` timestamp and continue with the visibility test. To guarantee serializability, transaction T acquires a *commit dependency* on TB, restricting the serialization order of the two transactions. (That is, once T takes a commit dependency on TB, T is allowed to commit only if TB commits.) Commit dependencies are discussed in more detail in Section 4.3.8.

If TB is in the Committed state, V's `Begin` timestamp field will soon be changed to TB's validation timestamp. T can therefore proceed and use TB's validation timestamp as V's `Begin` timestamp to check for visibility. There is no need to constrain the serialization order by acquiring a commit dependency in this case, as it is certain that TB has already committed.

If TB is Aborted, this means that transaction T read a version V that was created by TB, but will soon be discarded as part of TB's post-processing. T ignores V in this case.

Finally, there is the rare case where TB's state is Terminated or TB can not be found in the system. This occurs only when TB completes all post-processing in the very short time interval between when T checked V's `Begin` field and when T checked TB's state. (Although this is a very short window, the two checks are not atomic.) In this rare case, T simply re-reads version V's `Begin` field and repeats the visibility check.

Property 4.3. *A transaction T with a validation timestamp VT has created a new version V. The new version V is invisible to transactions reading earlier than VT.*

`End` field contains a transaction identifier

Once it has been determined that version V's valid time begins before transaction T's read time RT, we proceed to check V's `End` field. If it contains a timestamp, determining visibility is straightforward: V is visible to T if and only if RT is less than V's `End` timestamp. However, if version V is concurrently being deleted or updated by another transaction TE, the `End` field contains the identifier of transaction TE. In this case we have to check the state and validation timestamp of TE to determine whether V is visible. Table 4.3 summarizes the various cases and the actions to take, assuming that we have already determined that V's begin timestamp is, or will be, less than RT.

If transaction TE is Active, the version V is uncommitted and not visible to any other transaction but TE.

If TE's state is	Action to take when transaction T checks visibility of a version V.
Active	V is visible only if T $\neq$ TE.
Preparing	Let RT be transaction T's logical read time, and TS be transaction TE's validating timestamp. If RT $<$ TS, V is visible. Otherwise, T *speculatively ignores* V.
Committed	Use TE's validation timestamp as V's `End` field when testing for visibility.
Aborted	V is visible.
Terminated or TE not found	Reread V's `End` field.

Table 4.3: Case analysis of action to take when transaction T checks visibility of version V, and the `End` field of version V contains the identifier of transaction TE.

If TE's state is Preparing, it has a validation timestamp TS that will become the `End` timestamp of V if TE commits. If the logical read time RT is before TE's validation timestamp TS, V will be visible regardless whether TE commits or aborts: If TE commits, it will store its validation timestamp TS in the `End` field of V during post-processing. If TE aborts, V will still be visible, because any transaction that updates V after TE has aborted will obtain a validation timestamp greater than TS. If the logical read time RT is after TE's validation timestamp TS, we have a more complicated situation. If TE commits, V will not be visible to T; but if TE aborts, V will be visible. We could handle this by forcing T to wait until TE commits or aborts but we want to avoid all blocking during normal processing. Instead we allow T to *speculatively ignore* V and proceed with its processing: Transaction T acquires a commit dependency on TE, that is, T is allowed to commit only if TE commits. (See Section 4.3.8 for details on the commit dependency mechanism.)

The case when TE's state is Committed is obvious, but the Aborted case warrants some explanation. If TE has aborted, some other transaction TO may have sneaked in after T read V's `End` field, discovered that TE has aborted and updated V. However, TO must have updated V's `End` field after T read V, and TO must have been in the Active state. TO's validation timestamp would be assigned when switching to the Preparing state. Thus, TO's validation timestamp must be later than T's logical read time. It follows that it doesn't matter if a transaction TO "sneaked in" and updated the version; if TE is in the Aborted state, V is always visible to T.

If TE has terminated or is not found, TE must have finalized V's validation timestamp since we read the field. So we read the `End` field again and try again.

Property 4.4. *An uncommitted version is never visible.*

4.3.7 Updating a version

Suppose transaction T wants to update a version V. The version V is updatable only if it is the latest version, that is, it has an `End` timestamp field equal to infinity, or its `End` field contains the identifier of a transaction TE that is in the Aborted state. If the state of the transaction TE is Active, V is the latest committed version but there is a later uncommitted version. This is a write-write conflict. We follow the first-writer-wins rule and force transaction T to abort.

Property 4.5. *An update or delete to a version is always preceded by a read to the same version. If transaction T creates new version VN, T has first read old version V.*

Suppose transaction T finds that version V is updatable. It will create a new version and proceeds to install it in the database. The first step is to atomically compare-and-swap T's transaction identifier in V's `End` field to prevent other transactions from updating V. If the swap fails because the `End` field has changed, T must abort because some other transaction has sneaked in and updated V before T managed to install its update. If the swap succeeds, T then connects the new version into all indexes it participates in. T also saves pointers to the old and new versions; they will be needed during post-processing.

Property 4.6. *A transaction will abort if it attempts to update or delete a historic version of a record. A historic version has an* `End` *field that is either a timestamp not equal to infinity, or it contains a transaction identifier and the referenced transaction has committed.*

4.3.8 Commit dependencies

When a transaction T1 speculates on the outcome of another transaction T2, it is necessary to restrict the serialization order to guarantee correctness. We achieve this by keeping track of the commit dependencies between transactions.

A transaction T1 has a commit dependency on another transaction T2, if T1 is allowed to commit only if T2 commits. If T2 aborts, T1 must also abort, so cascading aborts are possible. T1 acquires a commit dependency either by speculatively reading or speculatively ignoring a version, instead of waiting for T2 to commit. We implement commit dependencies by a register-and-report approach: T1 registers its dependency with T2 and T2 informs T1 when it has committed or aborted. We track commit dependencies with three additional variables for each transaction:

1. A counter, `CommitDepCounter`, that counts how many unresolved commit dependencies a transaction still has. A transaction cannot commit until this counter is zero.

2. A boolean variable, `AbortNow`, that other transactions can set to force this transaction to abort.

3. Each transaction T also has a set, `CommitDepSet`, that stores the identifiers of the transactions that depend on T.

To take a commit dependency on a transaction T2, T1 increments its `CommitDepCounter` and adds its transaction identifier to T2's `CommitDepSet`. If T2 commits, it locates each transaction in its `CommitDepSet` and decrements their `CommitDepCounter`. If T2 aborts, it signals the dependent transactions to also abort by setting their `AbortNow` flags. T2 takes no action if a dependent transaction is not found, as this implies that the dependent has already aborted.

Note that transaction T1 with a commit dependency on T2 may not have to wait at all — T2 may have committed and disappeared before T1 is ready to commit. In essence, commit dependencies consolidate all waits into a single wait and postpone the wait to just before commit.

With the commit dependency mechanism, some transactions may have to wait until the `CommitDepCounter` is zero before they commit. Waiting raises a concern of deadlocks. However, deadlocks cannot occur because a commit dependency is registered only from an Active transaction to a Preparing transaction. Therefore, it follows that an older transaction (lower validation timestamp) will never wait on a younger transaction (higher validation timestamp). If one were to construct a wait-for graph, the direction of edges would always be from a younger transaction to an older transaction, so cycles are impossible.

4.4 Optimistic transactions

This section describes in more detail the processing performed in the different phases for optimistic transactions. We first consider serializable transactions and then discuss lower isolation levels. We then prove that the concurrency control scheme described here admits only 1SR multi-version histories.

The original paper by Kung and Robinson [56] introduced two validation methods: backward validation and forward validation. We use backward validation but optimize it for in-memory storage. Instead of validating a read set against the write sets of all other transactions, we simply check whether a version that was read is still visible as of the end of the transaction. A separate write phase is not needed; a transaction's updates become visible to other transactions when the transaction changes its state to Committed.

A serializable optimistic transaction keeps track of its reads, scans and writes. To this end, a transaction object contains three sets:

1. The **ReadSet** contains pointers to every version read.
2. The **WriteSet** contains pointers to versions updated (old and new), versions deleted (old) and versions inserted (new).
3. The **ScanSet** stores information needed to repeat every scan.

4.4.1 Normal Processing Phase

Normal processing consists of scanning indexes (see Section 4.3.2) to locate versions to read, update, or delete. Insertion of an entirely new record or updating an existing record creates a new version that is added to all indexes for records of that type.

To do an index scan, a transaction T specifies an index, a predicate P, and a logical read time RT. The predicate is a conjunction $P = P_S \wedge P_R$ where P_S is a search predicate that determines what part of the index to scan and P_R is an optional residual predicate. For a hash index, P_S is an equality predicate on columns of the hash key. For an ordered index, P_S is a range predicate on the ordering key of the index.

We now outline the processing during a scan when transaction T runs at the serializable isolation level. All reads specify the start timestamp of T as the logical read time.

Start scan When a scan starts, it is registered in T's ScanSet so T can check for phantoms during validation. Sufficient information must be recorded so that the scan can be repeated. In our implementation, we record the index the scan operates on, and the predicate P.

Check visibility Next we check whether version V is visible to transaction T as of time RT (see Section 4.3.6). The result of the test may be conditional on another transaction T2 committing. If so, T registers a commit dependency with T2 (see Section 4.3.8). If the visibility test returns false, the scan proceeds to the next version.

Check predicate If a version V doesn't satisfy P, it is ignored and the scan proceeds. If the scan is a range scan and the index key exceeds the upper bound of the range, the scan is terminated. If the scan is an equality predicate for a hash index, the scan is terminated when the end of the hash chain is encountered.

Read version Transaction T records the read by adding a pointer to version V to its ReadSet. The pointer will be used during validation. T can now read V without any further checks.

Check updatability If transaction T intends to update or delete V, we must check whether the version is updatable. A visible version is updatable if its `End` field equals infinity or it contains a transaction identifier and the referenced transaction has aborted. Note that speculative updates are allowed, that is, an uncommitted version can be updated but the transaction that created it must have completed normal processing.

Update version To update version V, transaction T first creates a new version VN and then atomically sets V's `End` field to T's transaction identifier. This fails if some other transaction T2 has already set V's `End` field. This is a write-write conflict and T must abort.

In the most likely outcome, T succeeds in setting V's `End` field. This serves as an exclusive write lock on V because it prevents further updates of V. Transaction T records the update by adding two pointers to its WriteSet: a pointer to V (old version) and a pointer to VN (new version). These pointers are used later for three purposes: (1) for logging new versions during commit, (2) for post-processing after commit or abort, and (3) for locating old versions when they are no longer needed and can be garbage collected.

The new version VN is not visible to any other transaction until T switches to the Preparing state, therefore T can proceed to include VN in all indexes that the table participates in.

Delete version A delete is an update of V that doesn't create a new version. The `End` timestamp of V is first checked and then set in the same way as for updates. If this succeeds, a pointer to V (old version) is added to the write set and the delete is complete.

When transaction T reaches the end of normal processing, it *pre-commits* and enters its preparation phase. Pre-commit simply consists of acquiring the transaction's unique validation timestamp and setting the transaction state to Preparing.

4.4.2 Preparation phase

The preparation phase of an optimistic transaction consists of three steps: read validation, waiting for commit dependencies, and logging. We discuss each step in turn.

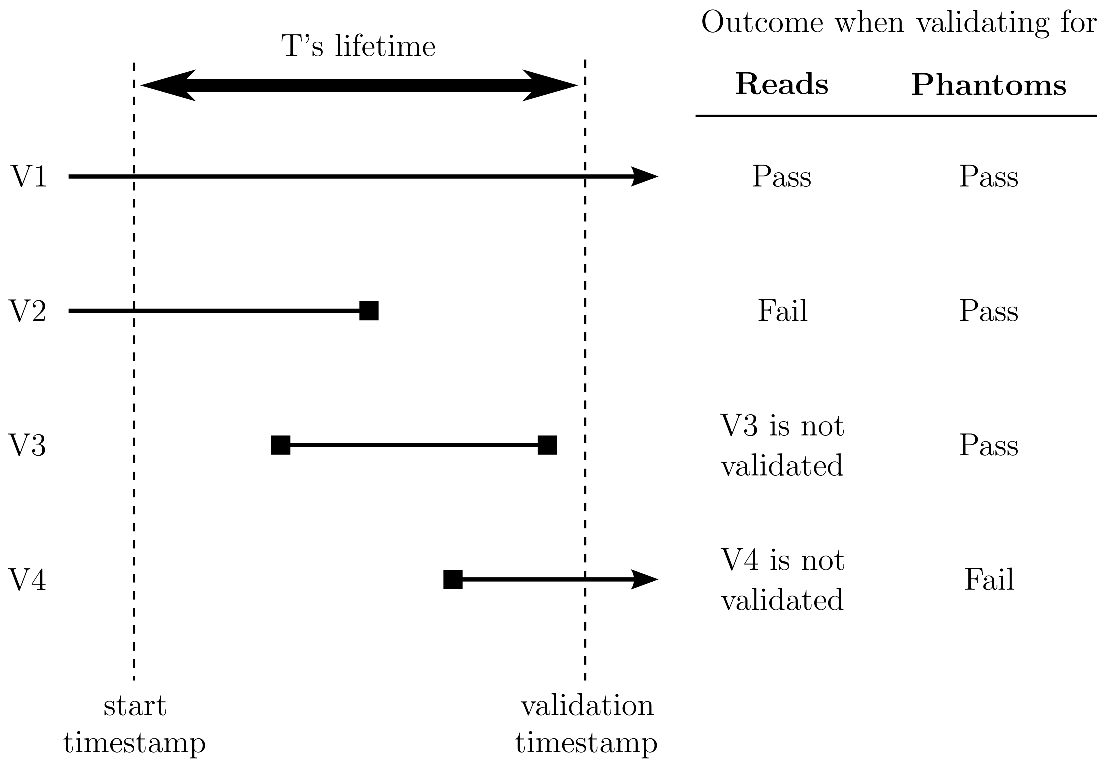

Figure 4.3: Possible transaction validation outcomes.

Validation consists of two steps: checking visibility of the versions read and checking for phantoms. To check visibility, transaction T scans its ReadSet. For each version read, T checks whether the version is still visible as of the end of the transaction. To check for phantoms, T walks its ScanSet and repeats each scan looking for versions that came into existence during T's lifetime and are visible as of the end of the transaction. (T may acquire additional commit dependencies during validation but only if it speculatively ignores a version.)

Figure 4.3 illustrates the different cases that can occur. It shows the lifetime of a transaction T, the valid times of versions V1 to V4 of four different records, and the expected outcome of read validation and phantom detection. We assume that all four versions satisfy the search predicate used by T and that they were all created and terminated by transactions other than T.

Version V1 is visible to T both at its start and validation timestamp. If V1 is included in T's ReadSet, it passes read validation and also phantom detection.

Version V2 is visible to T as of its start timestamp but not at the end of the transaction (that is, as of the validation timestamp). If V2 is included in T's ReadSet, it fails read validation. V2 is not a phantom.

Version V3 both began and ended during T's lifetime, so it is not visible to T at the start nor at the end of the transaction. It is not included in T's ReadSet so it won't be subject to read validation. V3 is not visible at the end of T, so V3 is not a phantom.

Version V4 was created during T's lifetime and is visible at the end of T, so V4 is a phantom. It is not included in T's ReadSet because it was not visible as of T's start time.

If T fails validation, it is not serializable and must abort. If T passes validation, it must wait for outstanding commit dependencies to be resolved, if it has any. More specifically, T can proceed to the post-processing phase if either its CommitDepCounter is zero or its AbortNow flag is set.

Property 4.7. *Let transaction T have a start timestamp ST and a validating timestamp VT. Every read that transaction T performed during the Active phase is associated with some logical read time RT, such that* $ST < RT < VT$*. During the Preperation phase, every read will be repeated as of logical time VT. Transaction T aborts if the read as of logical time RT and the read as of logical time VT return different versions.*

To complete the commit, T scans its WriteSet and writes the new versions it created to a persistent log. Commit ordering is determined by transaction validation timestamps, which are included in the log records so that multiple log streams on different devices can be used to ameliorate bottlenecks due to logging [53]. Deletes are logged by writing a unique key or, in the worst case, all columns of the deleted version. After the log writes have been completed and the log record is stored in non-volatile storage, T sets its transaction state to Committed, thereby signaling the end of this phase.

4.4.3 Postprocessing

During this phase a committed transaction TC propagates its validating timestamp to the `Begin` and `End` fields of new and old versions, respectively, listed in its WriteSet. An aborted transaction TA can discard all the new versions it created immediately. In addition, transaction TA attempts to reset the `End` fields of any old versions to infinity. However, another transaction may already have detected that TA aborted, created another new version and reset the `End` field of the old version. If so, TA leaves the `End` field unchanged.

The transaction then processes all outgoing commit dependencies listed in its `CommitDepSet`. If it aborted, it forces the dependent transactions to also abort by setting their `AbortNow` flags. If it committed, it decrements the target transaction's `CommitDepCounter` and wakes up the dependent transaction if the count becomes zero.

Once post-processing is done, other transactions no longer need to refer to the transaction object. It can be removed from the transaction table but it will not be discarded entirely; the pointers to old versions in its WriteSet are needed for garbage collection.

4.4.4 Lower isolation levels

Enforcing lower isolation levels requires less bookkeeping and fewer instructions. A transaction requiring a higher isolation level bears the full cost of enforcing stronger isolation and does not burden transactions running in lower isolation levels.

Repeatable read Repeatable read is required to enforce read stability but not to prevent phantoms. We implement repeatable read simply by validating a transaction's ReadSet before commit. As phantom detection is not required, a transaction's scans are not recorded. The transaction's start timestamp is used as the logical read time.

Read committed Read committed guarantees that only committed versions are read. We implement read committed by always using the current time as the logical read time. No validation is required, as uncommitted versions will never be visible (see Section 4.3.6). A transaction's reads and scans are not recorded at this isolation level.

Snapshot isolation Implementing snapshot isolation with a multi-version storage engine is straightforward: always read as of the start timestamp of the transaction. No validation is needed, so scans and reads are not tracked.

Read-only transactions If a transaction is known to be read-only, the best performance is obtained by running it under snapshot isolation or read committed depending on whether it needs a transaction-consistent view or not.

4.4.5 Correctness proof

We now prove that the multi-version optimistic scheduler only admits 1SR multi-version histories. We use the notation from Section 5.2 of [13].

The multi-vesrion serialization graph $MVSG(H, \ll)$ is a graph defined on a multi-version history H and a version total order $\ll$. Each node in the

$MVSG$ is a committed transaction in H. By definition, the $MVSG$ has an edge from transaction T_i to transaction T_j $(i \neq j \neq k)$ if and only if:

1. T_i writes version V_i and T_j reads version V_i, or

2. T_i writes version V_i and T_k reads version V_j, where $V_i \ll V_j$, or

3. T_i reads vesrion V_k and T_j writes version V_j, where $V_k \ll V_j$.

We will now prove that there are no cycles in $MVSG$ because every edge in $MVSG$ is ordered with respect to the validating timestamp order of the transactions involved. Let $E(T)$ denote the validating timestamp of transaction T. We prove that an edge $T_i \rightarrow T_j$ exists in $MVSG$ if and only if $E(T_i) < E(T_j)$.

1. We start with the first case, where an edge is in $MVSG$ because T_i writes version V_i and T_j reads version V_i. Suppose T_j read V_i at timestamp $R(T_j)$. From Property 4.7, it follows that $R(T_j) < E(T_j)$. If $R(T_j) < E(T_i)$, it would have been impossible for T_j to read V_i, as it is uncommitted (Properties 4.3 and 4.4). Therefore $E(T_i) \leq R(T_j)$, so $E(T_i) < E(T_j)$.

2. The second case occurs when T_i writes version V_i and T_k reads version V_j, where $V_i \ll V_j$. Suppose T_k read V_j at timestamp $R(T_k)$. As T_k reads V_j and uncommitted versions are invisible (Property 4.4), it follows that some transaction T_j created V_j and committed successfully. From Property 4.7, $E(T_j) \leq R(T_k)$, otherwise V_j would have not been visible to T_k as of time $R(T_k)$. Furthermore, from Property 4.7, it follows that $R(T_j) < E(T_j)$. There exists no version V_k such that $V_i \ll V_k \ll V_j$, otherwise T_j would have aborted during the update (Property 4.6). Since T_j wrote V_j, that implies that T_j read V_i (Property 4.5), hence $E(T_i) \leq R(T_j)$. Therefore, $E(T_i) < E(T_j)$.

3. The third case occurs when T_i reads vesrion V_k and T_j writes version V_j, where $V_k \ll V_j$. Suppose T_i read V_k at $R(T_i)$. From Property 4.7, $R(T_i) < E(T_i)$. Suppose that $E(T_j) < E(T_i)$. This ordering is impossible if T_j and T_i are committed transactions from Property 4.7: Transaction T_i would validate whether V_k is still visible as of $E(T_i)$. T_j has committed and $V_k \ll V_j$, therefore V_k is no longer visible as of $E(T_i)$ and validation would fail. T_i would abort and would never be a node in $MVSG$. Therefore, $E(T_i) < E(T_j)$.

We have shown that an edge $T_i \rightarrow T_j$ exists in $MVSG$ if and only if $E(T_i) < E(T_j)$, where $E(T)$ is the validating timestamp of committed transaction T. As validating timestamps are integers that are assigned from a monotonically increasing counter (Property 4.1), edges in $MSVG$ cannot be involved in a cycle. Hence, the multi-version histories accepted by our multi-version optimistic concurrency control scheduler are 1SR.

4.5 Pessimistic transactions

An optimistic transaction running at a higher isolation level may find its reads being invalidated repeatedly from short transactions. This can cause large read-mostly transactions to starve. Instead of going through a validation phase at the end, a transaction can choose to be *pessimistic* and acquire read locks to prevent its transactional reads from being invalidated. This section describes our design for multiversion locking optimized for main-memory databases.

We first describe the additional data structures and lock types needed to efficiently implement multiversion locking. We then summarize the processing phases a pessimistic transaction goes through, and highlight the differences compared to the optimistic scheme.

A serializable pessimistic transaction must keep track of which versions it read, which hash buckets it scanned, and its new and old versions. To this end, the transaction maintains three sets:

1. The **ReadSet** contains pointers to versions read-locked by the transaction.

2. The **BucketLockSet** contains pointers to hash buckets visited and locked by the transaction.

3. The **WriteSet** contains references to versions updated (old and new), versions deleted (old) and versions inserted (new).

4.5.1 Lock types

We use two types of locks: record locks and bucket locks. Record locks are placed on versions to ensure read stability. Bucket locks are placed on (hash) buckets to prevent phantoms. The name reflects their use for hash indexes in our prototype but range locks for ordered indexes can be implemented in the same way.

Record locks

Updates or deletes can only be applied to the latest version of a record; older versions cannot be further updated. Thus, locks are required only for the latest version of a record; never for older versions. So what's needed is an efficient many-readers-single-writer lock for this case.

We do not want to store record locks in a separate table — it's too slow. Instead we embed record locks in the `End` field of versions so no extra space is required. In our prototype, the `End` field of a version is 64 bits. As described earlier, this field stores either a timestamp or a transaction identifier with one bit indicating what the field contains. We change how we use this field to make room for a record lock.

1. ContentType (1 bit): indicates the content type of the remaining 63 bits.

2. Timestamp (63 bits): when ContentType is zero.

3. RecordLock (63 bits): when ContentType is one.

    3.1. NoMoreReadLocks (1 bit): a flag set when no further read locks are accepted. Used to prevent starvation.

    3.2. ReadLockCount (8 bits): number of read locks.

    3.3. WriteLock (54 bits): identifier of the transaction holding a write lock on this version or infinity (max value).

We do not explicitly keep track of which transactions have a version read locked. Each transaction records its ReadSet so we can find out by checking the ReadSets of all current transactions. This is only needed for deadlock detection which occurs infrequently.

A transaction acquires a read lock on a version V by atomically incrementing V's ReadLockCount. No further read locks can be acquired if the counter has reached its max value (255) or the NoMoreReadlocks flag is set. If so, the transaction aborts.

A transaction write locks a version V by atomically copying its transaction identifier into the WriteLock field. This action both write locks the version and identifies who holds the write lock.

Bucket Locks (Range Locks)

Bucket locks are used only by serializable transactions to prevent phantoms. When a transaction TS begins a scan of a hash bucket, it locks the bucket. Multiple transactions can have a bucket locked. A bucket lock consists of the following two fields:

1. LockCount: number of locks on this bucket.

2. LockList: list of (serializable) transactions holding a lock on this bucket.

The current implementation stores the LockCount in the hash bucket to be able to check quickly whether the bucket is locked. LockLists are implemented as arrays stored in a separate hash table with the bucket address as the key.

To acquire a lock on a bucket B, a transaction TS increments B's Lock-Count, locates B's LockList, and adds its transaction Id to the list. To release the lock it deletes its transaction identifier from B's LockList and decrements the LockCount.

Range locks in an ordered index can be implemented in the same way. If the index is implemented by a tree structure, a lock on a node locks the subtree rooted at that node. If the index is implemented by skip lists, locking a tower locks the range from that tower to the next tower of the same height.

4.5.2 Eager Updates, Wait-For Dependencies

In a traditional implementation of multiversion locking, an update transaction TU would block if it attempts to update or delete a read locked version or attempts to insert or update a version in a locked bucket. This may lead to frequent blocking and thread switching. A thread switch is expensive, costing several thousand instructions. In a main-memory system, just a few thread switches can add significantly to the cost of executing a transaction.

To avoid blocking we allow a transaction TU to eagerly update or delete a read locked version V but, to ensure correction serialization order, TU cannot precommit until all read locks on V have been released. Similarly, a transaction TR can acquire a read lock on a version that is already write locked by another transaction TU. If so, TU cannot precommit until TR has released its lock.

Note that an eager update or delete is not speculative because it doesn't matter whether TR commits or aborts; it just has to complete and release its read lock. The same applies to locked buckets. Suppose a bucket B is locked by two (serializable) transactions TS1 and TS2. An update transaction TU is allowed to insert a new version into B but it is not allowed to precommit before TS1 and TS2 have completed and released their bucket locks.

We enforce correct serialization order by wait-for dependencies. A wait-for dependency forces an update transaction TU to wait before it can acquire a validation timestamp and begin commit processing. There are two flavors of wait-for dependencies, read lock dependencies and bucket lock dependencies that differ in what event they wait on.

A transaction T needs to keep track of both incoming and outgoing wait-for dependencies. T has an incoming dependency if it waits on some other transaction and an outgoing dependency if some other transaction waits on it. To track wait-for dependencies, the following fields are included in each transaction object.

1. WaitForCounter: indicates how many incoming dependencies the transaction is waiting for.

2. NoMoreWaitFors: when set the transaction does not allow additional incoming dependencies. Used to prevent starvation by incoming dependencies continuously being added.

3. WaitingTxnList: Tracks outgoing wait-for dependencies by storing the identifiers of transactions waiting on this transaction to complete.

Read lock dependencies

A transaction TU that updated or deleted a version V has a wait-for dependency on V as long as V is read locked. TU is not allowed to acquire a validation timestamp and begin commit processing unless V's ReadLockCount is zero.

When a transaction TU updates or deletes a version V, it acquires a write lock on V by copying its transaction identifier into the WriteLock field.

If V's ReadLockCount is greater than zero, TU takes a wait-for dependency on V simply by incrementing its WaitForCounter.

TU may also acquire a wait-for dependency on V by another transaction TR taking a read lock on V. A transaction TR that wants to read a version V must first acquire a read lock on V by incrementing V's ReadLockCount. If V's NoMoreReadLocks flag is set or ReadLockCount is at max already, lock acquisition fails and TR aborts. Otherwise, if V is not write locked or V's ReadLockCount is greater than zero, TR increments V's ReadLockCount and proceeds. However, if V is write locked by a transaction TU and this is the first read lock on V (V's ReadLockCount is zero), TR must force TU to wait on V. TR checks TU's NoMoreWaitFors flag. If it is set, TU cannot install the wait-for dependency and aborts. Otherwise everything is in order and TR acquires the read lock by incrementing Vs' ReadLockCounter and installs the wait-for dependency by incrementing TU's WaitForCounter.

When a transaction TR releases a read lock on a version V, it may also need to release a wait-for dependency. If V is not write locked, TR simply decrements V's ReadLockCounter and proceeds. The same applies if V is write locked and V's ReadLockCounter is greater than one. However, if V is write locked by a transaction TU and V's ReadLockCounter is one, TR is about to release the last read lock on V and therefore must also release TU's wait-for dependency on V. TR atomically sets V's ReadLockCounter to zero and V's NoMoreReadLocks to true. If this succeeds, TR locates TU and decrements TU's WaitForCounter.

Setting the NoMoreReadLocks flag before releasing the wait-for dependency is necessary because this may be TU's last wait-for dependency. If so, TU is free to acquire a validation timestamp and being its commit processing. In that case, TU's commit cannot be further postponed by taking out a read lock on V. In other words, further read locks on V would have no effect.

Bucket lock dependencies

A serializable transaction TS acquires a lock on a bucket B by incrementing B's LockCounter and adding its transaction identifier to B's LockList. The purpose of TR's bucket lock is not to disallow new versions from being added to B but to prevent them from becoming visible to TR. That is, another transaction TU can add a version to B but, if it does, then TU cannot precommit until TS has completed its processing and released its lock on B. This is enforced by TU obtaining a wait-for dependency on TS.

TU can acquire this type of dependency either by acquiring one itself or by having one imposed by TS. We discuss each case.

Suppose that, as a result of an update or insert, TU is about to add a new version V to a bucket B. TU checks whether B has any bucket locks. If it does, TU takes out a wait-for dependency on every transaction TS in B's LockList by adding its own transaction identifier to TS's WaitForList and incrementing its own WaitForCounter. If TU's NoMoreWaitFors flag is set, TU can't take out the dependency and aborts.

Suppose a serializable transaction TS is scanning a bucket B and encounters a version V that satisfies TS's search predicate but the version is not visible to TS, that is, V is write locked by a transaction TU that is still active. If TU commits before TS, V becomes a phantom to TS. To prevent this from happening, TS registers a wait-for dependency on TU's behalf by adding TU's transaction identifier to its own WaitingTxnList and incrementing TU's WaitForCounter. If TU's NoMoreWaitFors flag is set, TS can't impose the dependency and aborts.

When a serializable transaction TS has precommitted and acquired its validation timestamp, it releases its outgoing wait-for dependencies. It scans its WatingTxnList and, for each transaction T found, decrements T's WaitForCounter.

4.5.3 Processing phases

This section describes how locking affects the processing done in the different phases of a transaction.

Normal processing phase

Recall that normal processing consists of scanning indexes to select record versions to read, update, or delete. An insertion or update creates a new version that has to be added to all indexes for records of that type.

We now outline what a pessimistic transaction T does differently than an optimistic transaction during a scan and how this depends on T's isolation level. For snapshot isolation, the logical read time is always the transaction start timestamp. For all other isolation levels, it is the current time which has the effect that the read sees the latest version of a record.

Start scan If T is a serializable transaction, it takes out a bucket lock on B to prevent phantoms and records the lock in its BucketLockSet. Other isolation levels do not take out a bucket lock.

Check predicate This phase is the same as for optimistic transactions. (Refer to Section 4.4.1 for details.)

Check visibility This is done in the same way as for optimistic transaction, including taking out commit dependencies as needed. If a version V is not visible, it is ignored and the scan continues for all isolations levels except serializable. If T is serializable and V is write locked by a transaction TU that is still active, V is a potential phantom so T forces TU to delay its precommit by imposing a wait-for dependency on TU.

Read version If T runs under serializable or repeatable read and V is a latest version, T attempts to acquire a read lock on V. If T can't acquire the read lock, it aborts. If T runs under a lower isolation level or V is not a latest version, no read lock is required.

Check updatability This phase is the same as for optimistic transactions. (Refer to Section 4.4.1 for details.)

Update version As for optimistic transactions, T creates a new version N, sets V's WriteLock and, if V was read locked, takes out a wait-for dependency on V by incrementing its own WaitForCounter. T then proceeds to add N to all indexes it participates in. If T adds N to a locked index bucket B, it takes out wait-for dependencies on all (serializable) transactions holding locks on B.

Delete version A delete is essentially an update of V that doesn't create a new version. T sets V's WriteLock and if V was read locked, takes out a wait-for dependency on V by incrementing its own WaitForCounter.

When transaction T reaches the end of normal processing, it releases its read locks and its bucket locks, if any. If it has outstanding wait-for dependencies (its WaitForCounter is greater than zero), it waits. Once its WaitForCounter is zero, T precommits, that is, acquires a validation timestamp and sets its state to Validating.

Preparation phase

Pessimistic transactions require no validation that's taken care of by locks. However, a pessimistic transaction T may still have outstanding commit dependencies when reaching this point. If so, T waits until they have been resolved and then proceeds to write to the log and commit. If a commit dependency fails, T aborts.

Postprocessing phase

Postprocessing is the same as for optimistic transactions. Note that there is no need to explicitly release write locks; this is automatically done when the transaction updates `Begin` and `End` fields.

4.5.4 Deadlock detection

Commit dependencies are only taken on transactions that have already precommitted and are completing validation. As discussed earlier Section 4.3.8 commit dependencies cannot cause or be involved in a deadlock.

Wait-for dependencies, however, can cause deadlocks. To detect deadlocks we build a standard wait-for graph by analyzing the wait-for dependencies of all transactions that are currently blocked. Once the wait-for graph has been built, any algorithm for finding cycles in graphs can be used. Our prototype uses Tarjan's algorithm [75] for finding strongly connected components.

A wait-for graph is a directed graph with transactions as nodes and waits-for relationships as edges. There is an edge from transaction T2 to transaction T1 if T2 is waiting for T1 to complete. The graph is constructed in three steps.

Create nodes Scan the transaction table and for each transaction T found, create a node N(T) if T has completed its normal processing and is blocked because of wait-for dependencies.

Create egdes from explicit dependencies Wait-for dependencies caused by bucket locks are represented explicitly in WaitingTxnLists. For each transaction T1 in the graph and each transaction T2 in T1's WaitingTxnList, add an edge from T2 to T1.

Create edges from implicit dependencies A wait-for dependency on a read-locked version V is an implicit representation of wait-for dependencies on all transaction holding read locks on V. We can find out which transactions hold read locks on V by checking transaction read sets. Edges from implicit dependencies can be constructed as follows. For each transaction T1 in the graph and each version V in T1's ReadLockSet: if V is write locked by a transaction T2 and T2 is in the graph, add an edge from T2 to T1.

While the graph is being constructed normal processing continues so wait-for dependencies may be created and resolved and transactions may become blocked or unblocked. Hence, the final graph obtained may be imprecise, that is, it may differ from the graph that would be obtained if normal processing stopped. But this doesn't matter because if there truly is a deadlock neither the nodes nor the edges involved in the deadlock can disappear while the graph is being constructed. There is a small chance of detecting a false deadlock but this is handled by verifying that the transactions participating in the deadlock are still blocked and the edges are still covered by unresolved wait-for dependencies.

4.5.5 Correctness proof

The multi-version locking scheme is a MV2PL scheduler. Section 5.5.2 of [79] describes MV2PL in detail and proves it only admits 1SR multi-version histories. In our implementation, having the NoMoreReadLocks bit set on the latest version of a record (see Section 4.5.2) is equivalent to holding a certify (commit) lock on a data item in MV2PL.

4.5.6 Peaceful coexistence

An interesting property of our design is that optimistic and pessimistic transactions can be mixed. The change required to allow optimistic transactions to coexist with pessimistic transactions is straightforward: optimistic update transactions must honor read locks and bucket locks. Making an optimistic transaction T honor read locks and bucket locks requires the following changes:

1. When T write locks a version V, it uses only a 54-bit transaction identifier and doesn't overwrite read locks.

2. When T updates or deletes a version V that is read locked, it takes a wait-for dependency on V.

3. When T inserts a new version into a bucket B, it checks for bucket locks and takes out wait-for dependencies as needed.

4.6 Experimental results

Our prototype storage engine implements both the optimistic and the pessimistic scheme. We also implemented a single-version storage engine with locking for concurrency control. The implementation is optimized for main-memory databases and does not use a central lock manager, as this can become a bottleneck [52]. Instead, we embed a lock table in every index and assign each hash key to a lock in this partitioned lock table. A lock covers all records with the same hash key which automatically protects against phantoms. We use timeouts to detect and break deadlocks.

The experiments were run on a two-socket Intel Xeon X5650 @ 2.67 GHz (Nehalem) that has six cores per socket. HyperThreading was enabled. The system has 48 GB of memory, 12 MB L3 cache per socket, 256 KB L2 cache per core, and two separate 32 KB L1-I and L1-D caches per core. This is a NUMA system and memory latency is asymmetric: accessing memory on the remote socket is approximately 30% slower than accessing local memory. We size hash tables appropriately so there are no collisions. The operating system is Windows Server 2008 R2.

The I/O bandwidth required for logging is moderate: Each update produces a log record that stores the difference between the old and new versions, plus 8 bytes of metadata. Even with millions of updates per second, the I/O bandwidth required is within what even a single desktop hard drive can deliver. However, logging each transaction individually involves a very high number of I/O operations. For our experiments, each transaction generates log records but these are asynchronously written to durable storage;

transactions do not wait for log I/O to complete. This choice allows us to focus on the effect of concurrency control. Asynchronous logging allows us to submit log records in batches (group commit), greatly reducing the number of I/O operations for our experiments. The emergence of fast non-volatile storage that can sustain a very high number of I/O operations promises to ameliorate this problem in the future.

Traditional disk-based transaction processing systems require hundreds of concurrently active transactions to achieve maximum throughput. This is to give the system useful work to do while waiting for disk I/O. Our main-memory engine does not wait for disk I/O, so there is no need to over-provision threads. We observed that the CPU is fully utilized when the multi-programming level equals the number of hardware threads; allowing more concurrent transactions causes throughput to drop. We therefore limited the number of concurrently active transactions to be at most 24, which matches the number of threads our machine supports.

We experiment with three CC schemes: the single-version locking engine (labeled "1V" the multi-version engine when transactions run optimistically ("MV/O") and the multi-version engine where transactions run pessimistically ("MV/L").

4.6.1 Homogeneous workload

We first show results from a parameterized artificial workload. By varying the workload parameters we can systematically explore how sensitive the different schemes are to workload characteristics. We focus on short update transactions which are common for OLTP workloads. The workload consists of a single transaction type that performs R reads and W writes against a table of N records with a unique key. Each row is 24 bytes, and reads and writes are uniformly and randomly scattered over the N records.

The memory footprint of the database differs for each concurrency control scheme. In 1V, each row consumes 24 bytes (payload) plus 8 bytes for

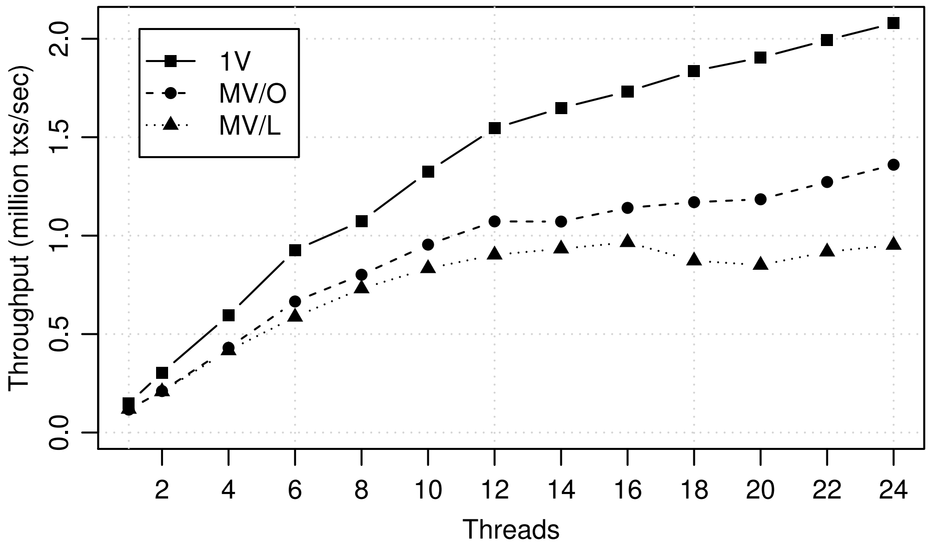

Figure 4.4: Scalability under low contention.

the "next" pointer of the hash table. Both MV schemes additionally use 16 bytes to store the `Begin` and `End` fields (cf. Figure 1), but the total consumption depends on the average number of versions required by the workload. Comparing the two MV schemes, MV/L has the biggest memory footprint, due to the additional overhead of maintaining a bucket lock table.

Scalability (Read Committed)

We first show how transaction throughput scales with increasing multiprogramming level. For this experiment each transaction performs 10 reads and 2 writes (R=10 and W=2) against a table with N=10,000,000 rows. The isolation level is Read Committed.

Figure 4.4 plots transaction throughput (y-axis) as the multiprogramming level increases (x-axis). Under low contention, throughput for all three schemes scales linearly up to six threads. After six threads, we see the effect of the higher access latency as the data is spread among two NUMA nodes, and beyond twelve threads we see the effect of HyperThreading.

For the 1V scheme, HyperThreading causes the speed-up rate to drop but the system still continues to scale linearly, reaching a maximum of over 2M transactions/sec. The multiversion schemes have lower throughput because of the overhead of version management and garbage collection. Creating a new version for every update and cleaning out old versions that are no longer needed is obviously more expensive than updating in place.

Comparing the two multiversion schemes, MV/L has 30% lower performance than MV/O. This is caused by extra writes for tracking dependencies and locks, which cause increased memory traffic. It takes MV/L 20% more cycles to execute the same number of instructions and the additional control logic translates into 10% more instructions per transaction.

Scaling under contention (Read Committed)

Records that are updated very frequently (hotspots) pose a problem for all CC schemes. In locking schemes, high contention causes transactions to wait because of lock conflicts and deadlocks. In optimistic schemes, hotspots result in validation failures and write-write conflicts, causing high abort rates and wasted work. At the hardware level, some data items are accessed so frequently that they practically reside in the private L1 or L2 caches of each core. This stresses the hardware to the limits, as it triggers very high core-to-core traffic to keep the caches coherent.

We simulate a hotspot by running the same R=10 and W=2 transaction workload from Section 5.1.1 on a very small table with just N=1,000 rows. Transactions run under Read Committed. Figure 4.5 shows the throughput under high contention. Even in this admittedly extreme scenario, all schemes achieve a throughput of over a million transactions per second, with MV/O slightly ahead of both locking schemes. 1V achieves its highest throughput at six threads, then drops and stays flat after 8 threads.

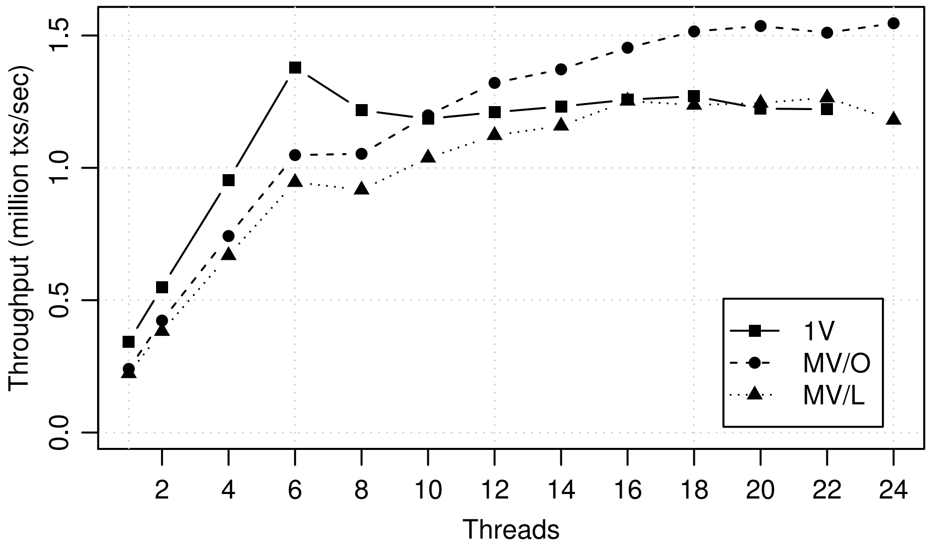

Figure 4.5: Scalability under high contention.

Higher isolation levels

The experiments in the previous section ran under Read Committed isolation level, which is the default isolation level in many commercial database engines [46, 61], as it prevents dirty reads and offers high concurrency. Higher isolation levels prevent more anomalies but reduce throughput. In this experiment, we use the same workload from Section 4.6.1, we fix the multiprogramming level to 24 and we change the isolation level.

In Table 4.4, we report the transaction throughput from each scheme and isolation level. We also report the throughput drop as a percentage of the throughput when running under the Read Committed isolation level.

The overhead for Repeatable Read for both locking schemes is very small, less than 2%. MV/O needs to repeat the reads at the end of the transaction, and this causes an 8% drop in throughput. For Serializable, the 1V scheme protects the hash key with a lock, and this guarantees phan-

	Read Committed	Repeatable Read		Serializable	
	Throughput (tx/sec)	Throughput (tx/sec)	% drop vs RC	Throughput (tx/sec)	% drop vs RC
1V	2,080,492	2,042,540	1.8%	2,042,571	1.8%
MV/L	974,512	963,042	1.2%	877,338	10.0%
MV/O	1,387,140	1,272,289	8.3%	1,120,722	19.2%

Table 4.4: Throughput at higher isolation levels, and percentage drop compared to Read Committed (RC).

tom protection with very low overhead (2%). Both MV schemes achieve serializability at a cost of 10%- 19% lower throughput: MV/L acquires read locks and bucket locks, while MV/O has to repeat each scan during validation. Under MV/O, however, a transaction requesting a higher isolation level bears the full cost of enforcing the higher isolation. This is not the case for locking schemes.

4.6.2 Heterogeneous workload

The workload used in the previous section represents an extreme update-heavy scenario. In this section we fix the multiprogramming level to 24 and we explore the impact of adding read-only transactions in the workload mix.

Impact of short read transactions

In this experiment we change the ratio of read and update transactions. There are two transaction types running under Read Committed isolation: the update transaction performs 10 reads and 2 writes (R=10 and W=2), while the read-only transaction performs 10 reads (R=10 and W=0).

Figure 4.6 shows throughput (y-axis) as the ratio of read-only transactions varies in the workload (x-axis) in a table with 10,000,000 rows. The

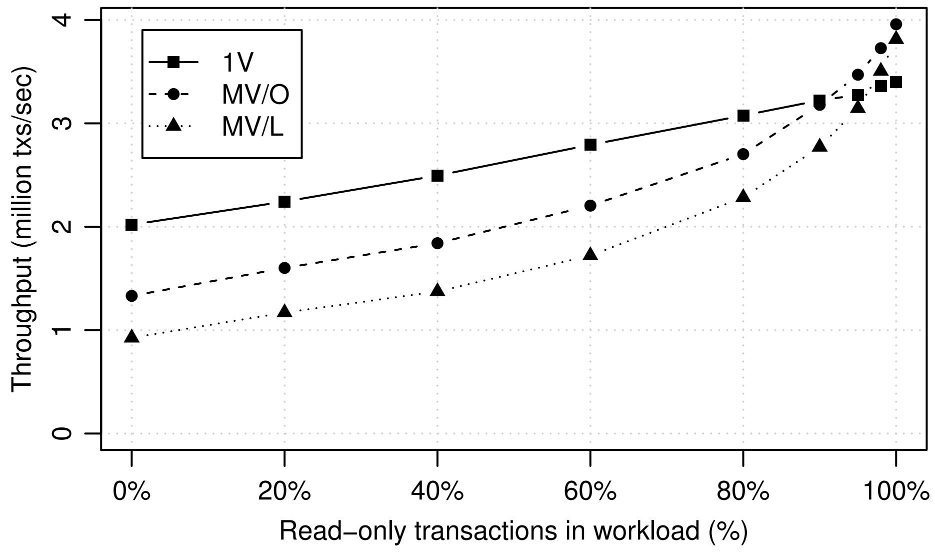

Figure 4.6: Impact of read-only transactions on throughput (low contention).

leftmost point (x=0%) reflects the performance of the update-only workload of Section 4.6.1 at 24 threads. As we add read-only transactions to the mix, the performance gap between all schemes closes. This is primarily because the update activity is reduced, reducing the overhead of garbage collection.

The MV schemes outperform 1V when most transactions are read-only. When a transaction is read-only, the two MV schemes behave identically: transactions read a consistent snapshot and do not need to do any locking or validation. In comparison, 1V has to acquire and release short read locks for cursor stability even for read-only transactions which impacts performance.

In Figure 4.7 we repeat the same experiment but simulate a hotspot by using a table of 1,000 rows. The leftmost point (x=0%) again reflects the performance of the update-only workload of Section 5.1.12 under high contention at 24 threads. The MVCC schemes have a clear advantage at

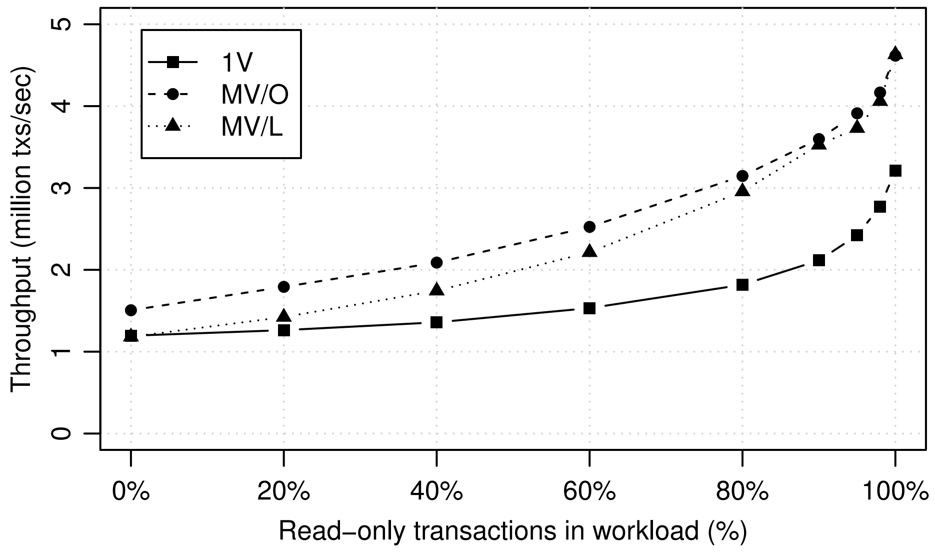

Figure 4.7: Impact of read-only transactions on throughput (high contention).

high contention, as snapshot isolation prevents read-only transactions from interfering with writers. When 80% of the transactions are read-only, the MVCC schemes achieve 63% and 73% higher throughput than 1V.

Impact of long read transactions

Not all transactions are short in OLTP systems. Users often need to run operational reporting queries on the live system. These are long read-only transactions that may touch a substantial part of the database. The presence of a few long-running queries should not severely affect the throughput of "normal" OLTP transactions.

This experiment investigates how the three concurrency control methods perform in this scenario. We use a table with 10,000,000 rows and fix the number of concurrently active transactions to be 24. The workload consists of two transaction types: (a) Long, transactionally consistent (Serializable),

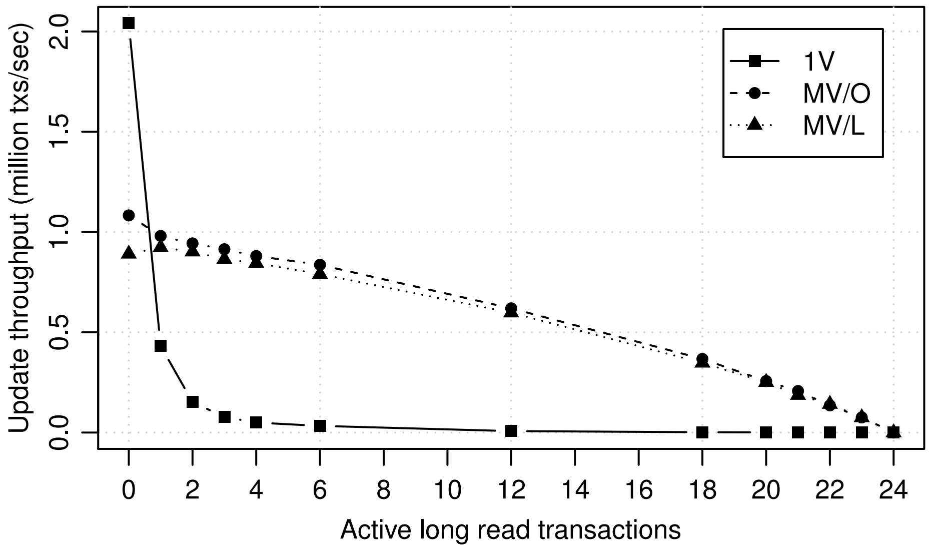

Figure 4.8: Update throughput with long read transactions. Each update transaction performs ten reads and two updates.

read-only queries that touch 10% of the table (R=1,000,000 and W=0) and (b) Short update transactions with R=10 and W=2.

Figures 4.8 and 4.9 show update and read throughput (y-axis) as we vary the number of concurrent long read-only transactions in the system (x-axis). At the leftmost point (x=0) all transactions are short update transactions, while at the rightmost point (x=24) all transactions are read-only. At x=6, for example, there are 6 readonly and 24-6=18 short update transactions running concurrently.

Looking at the update throughput in Figure 7, we can see that 1V is twice as fast as the MV schemes when all transactions are short update transactions, at x=0. (This is consistent with our findings from the experiments with the homogeneous workload in Section 4.6.1.) However the picture changes completely once a single long read-only transaction is present in the system. At x=1, update throughput drops 75% for the single

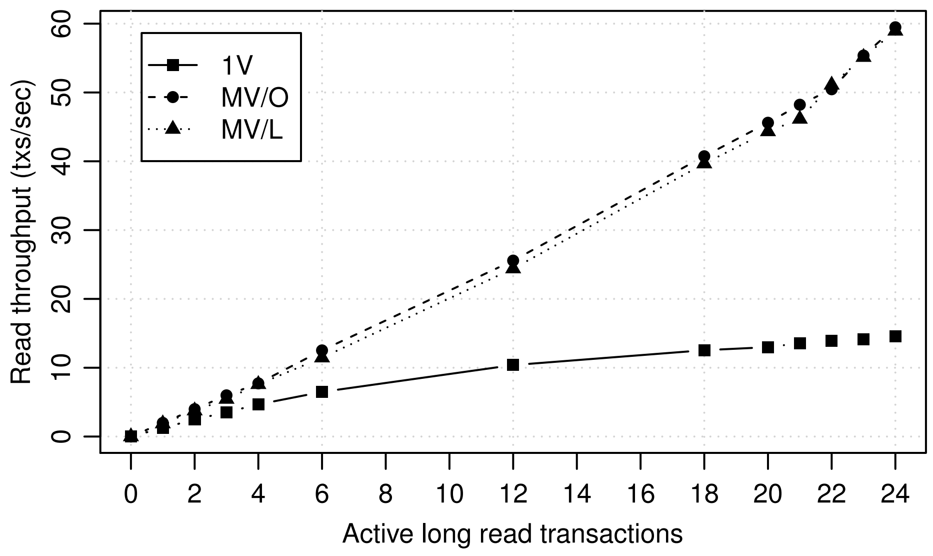

Figure 4.9: Read throughput with long read transactions. Each long read transaction performs one million reads (10% of the database).

version engine. In contrast, update throughput drops only 5% for the MV schemes, making MV twice as fast as 1V. The performance gap grows as we allow more read-only transactions. When 50% of the active transactions are long readers, at x=12, MV has 80X higher update throughput than 1V. In terms of read throughput (Figure 4.9), both MV schemes consistently outperform 1V.

4.6.3 TATP results

The workloads used in the previous sections allowed us to highlight fundamental differences between the three concurrency control mechanisms. However, real applications have higher demands than randomly reading and updating values in a single index. We conclude our experimental evaluation by running a benchmark that models a simple but realistic OLTP

	Transactions per second
1V	4,220,119
MV/L	3,129,816
MV/O	3,121,494

Table 4.5: TATP results.

application.

The TATP benchmark [76] simulates a telecommunications application. The database consists of four tables with two indexes on each table to speed up lookups. The benchmark runs a random mix of seven short transactions; each transaction performs less than 5 operations on average. 80% of the transactions executed only query the database, while 16% update, 2% insert, and 2% delete items. We sized the database for 20 million subscribers and generated keys using the non-uniform distribution that is specified in the benchmark. All transactions run under Read Committed.

Table 4.5 shows the number of committed transactions per second for each scheme. Our concurrency control mechanisms can sustain a throughput of several millions of transaction per second on a low-end server machine. This is an order of magnitude higher than previously published TATP numbers for disk-based systems [52] or main memory systems [50].

4.7 Concluding remarks

In this chapter we investigated concurrency control mechanisms optimized for main memory databases. The known shortcomings of traditional locking led us to consider solutions based on multiversioning. We designed and implemented two multi-version concurrency control methods, one optimistic using validation and one pessimistic using locking. For comparison purposes we also implemented a variant of single-version locking optimized for main memory databases. We then experimentally evaluated the performance of the three methods. Several conclusions can be drawn from the experimental results.

- Single-version locking can be implemented efficiently and without lock acquisition becoming a bottleneck.
- Single-version locking is fragile; it performs well when transactions are short and contention is low but suffers under more demanding conditions.
- The multi-version concurrency control schemes have higher overhead but are more resilient, retaining good throughput even in the presence of hotspots and long read-only transactions.
- The optimistic multi-versioning concurrency control scheme consistently achieves higher throughput than the pessimistic scheme.

Bibliography

[1] Divyakant Agrawal, Arthur J. Bernstein, Pankaj Gupta, and Soumitra Sengupta. Distributed optimistic concurrency control with reduced rollback. *Distributed Computing*, 2(1):45–59, 1987.

[2] Divyakant Agrawal and Soumitra Sengupta. Modular synchronization in multiversion databases: Version control and concurrency control. In *SIGMOD Conference*, pages 408–417, 1989.

[3] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving relations for cache performance. In *VLDB*, pages 169–180, 2001.

[4] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a modern processor: Where does time go? In *VLDB*, pages 266–277, 1999.

[5] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *PVLDB*, 5(10):1064–1075, 2012.

[6] Arvind Arasu, Spyros Blanas, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, Prasang Upadhyaya, and Ramarathnam Venkatesan. Engineering security and performance with cipherbase. *IEEE Data Eng. Bull.*, 35(4):65–72, 2012.

[7] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, and Ramaratnam Venkatesan. Orthogonal Security with Cipherbase. In *CIDR*, 2013.

[8] Subi Arumugam, Alin Dobra, Christopher M. Jermaine, Niketan Pansare, and Luis Leopoldo Perez. The DataPath system: a data-centric analytic processing engine for large data warehouses. In *SIGMOD*, pages 519–530, 2010.

[9] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*, 2013.

[10] Ronald Barber, Peter Bendel, Marco Czech, Oliver Draese, Frederick Ho, Namik Hrle, Stratos Idreos, Min-Soo Kim, Oliver Koeth, Jae-Gil Lee, Tianchao Tim Li, Guy M. Lohman, Konstantinos Morfonios, René Müller, Keshava Murthy, Ippokratis Pandis, Lin Qiao, Vijayshankar Raman, Richard Sidle, Knut Stolze, and Sandor Szabo. Business analytics in (a) blink. *IEEE Data Eng. Bull.*, 35(1):9–14, 2012.

[11] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. In *ISCA*, pages 237–248, 2013.

[12] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD Conference*, pages 1–10, 1995.

[13] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. ISBN 0-201-10715-5.

[14] Paul M. Bober and Michael J. Carey. Multiversion query locking. In *VLDB*, pages 497–510, 1992.

[15] Paul M. Bober and Michael J. Carey. On mixing queries and transactions via multiversion locking. In *ICDE*, pages 535–545, 1992.

[16] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, pages 54–65, 1999.

[17] Mihaela A. Bornea, Orion Hodson, Sameh Elnikety, and Alan Fekete. One-copy serializability with snapshot isolation under the hood. In *ICDE*, pages 625–636, 2011.

[18] Albert Burger, Vijay Kumar, and Mary Lou Hines. Performance of multiversion and distributed two-phase locking concurrency control mechanisms in distributed databases. *Information Sciences*, 96(1&2):129–152, 1997.

[19] Michael J. Cahill, Uwe Röhm, and Alan David Fekete. Serializable isolation for snapshot databases. *ACM Transactions on Database Systems*, 34(4), 2009.

[20] Michael J. Carey. Multiple versions and the performance of optimistic concurrency control. Technical report, #517, Computer Sciences Department, University of Wisconsin–Madison, October 1983.

[21] Michael J. Carey and Waleed A. Muhanna. The performance of multiversion concurrency control algorithms. *ACM Transactions on Computer Systems*, 4(4):338–378, 1986.

[22] John Cieslewicz, William Mee, and Kenneth A. Ross. Cache-conscious buffering for database operators with state. In *DaMoN*, pages 43–51, 2009.

[23] John Cieslewicz and Kenneth A. Ross. Data partitioning on chip multiprocessors. In *DaMoN*, pages 25–34, 2008.

[24] John Cieslewicz, Kenneth A. Ross, and Ioannis Giannakakis. Parallel buffers for chip multiprocessors. In *DaMoN*, 2007.

[25] David J. DeWitt and Jim Gray. Parallel database systems: The future of database processing or a passing fad? *SIGMOD Record*, 19(4):104–112, 1990.

[26] David J. DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6), 1992.

[27] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David A. Wood. Implementation techniques for main memory database systems. In *SIGMOD Conference*, pages 1–8, 1984.

[28] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *PDIS*, pages 280–291, 1991.

[29] David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *VLDB*, pages 27–40, 1992.

[30] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and concurrent memcache with dumber caching nd smarter hashing. In *Proceedings of the 10th USENIX NSDI*, Lombard, IL, April 2013.

[31] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA database - Data management for modern business applications. *SIGMOD Record*, 40(4):45–51, 2011.

[32] Glenn Fowler, Landon Curt Noll, and Phong Vo. FNV hash. `http://www.isthe.com/chongo/tech/comp/fnv/`. [Online reference; accessed June 11, 2013].

[33] Philip Garcia and Henry F. Korth. Database hash-join algorithms on multithreaded computer architectures. In *Conf. Computing Frontiers*, pages 241–252, 2006.

[34] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. SharedDB: Killing one thousand queries with one stone. *PVLDB*, 5(6):526–537, 2012.

[35] Gaston H. Gonnet. Expected length of the longest probe sequence in hash code searching. *J. ACM*, 28:289–304, April 1981.

[36] Goetz Graefe. Encapsulation of parallelism in the Volcano query processing system. In *SIGMOD*, pages 102–111, 1990.

[37] Goetz Graefe. Sort-merge-join: An idea whose time Has(h) passed? In *ICDE*, pages 406–417, 1994.

[38] Goetz Graefe. Implementing sorting in database systems. *ACM Comput. Surv.*, 38(3), 2006.

[39] Jim Gray. A "measure of transaction processing" 20 years later. *IEEE Data Engineering Bulletin*, 28(2):3–4, 2005.

[40] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993. ISBN 1-55860-190-2.

[41] Theo Härder and Erwin Petry. Evaluation of a multiple version cheme for concurrency control. *Information Systems*, 12(1):83–98, 1987.

[42] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, pages 981–992, 2008.

[43] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. QPipe: A simultaneously pipelined relational query engine. In *SIGMOD*, pages 383–394, 2005.

[44] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*, chapter 1, pages 3, 24. Morgan Kaufmann, 5th edition, 2012. ISBN 978-0-12-383872-8.

[45] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[46] *IBM DB2 10.1 for Linux, UNIX, and Windows: Troubleshooting and Tuning Database Performance*, chapter 3, page 153. January 2013. Reference number: SC27-3889-01.

[47] *IBM solidDB: In-Memory Database User Guide*, March 2013. Reference number: SC27-3845-04.

[48] "Compare Intel(r) Products" website. `http://ark.intel.com/compare/47922,37149`. [Online reference; accessed June 11, 2013].

[49] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 1, chapter 14. Reference number: 253665-047US.

[50] *Intel and IBM Collaborate to Double In-Memory Database Performance.* Whitepaper reference number: IMW14204-USEN-00.

[51] *Intel Xeon Processor 7500 Series Uncore Programming Guide*, March 2010. Reference number: 323535-001.

[52] Ryan Johnson, Ippokratis Pandis, and Anastasia Ailamaki. Improving OLTP scalability using speculative lock inheritance. *PVLDB*, 2(1):479–489, 2009.

[53] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. Aether: A scalable approach to logging. *PVLDB*, 3(1):681–692, 2010.

[54] Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *PVLDB*, 2(2):1378–1389, 2009.

[55] Vijay Kumar, editor. *Performance of Concurrency Control Mechanisms in Centralized Database Systems.* Prentice-Hall, 1996. ISBN 0-13-065442-6.

[56] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.

[57] Sanjay Kumar Madria, Mohammed Baseer, Vijay Kumar, and Sourav S. Bhowmick. A transaction model and multiversion concurrency control for mobile database systems. *Distributed and Parallel Databases*, 22(2-3):165–196, 2007.

[58] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. What happens during a join? Dissecting CPU and memory optimization effects. In *VLDB*, pages 339–350, 2000.

[59] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing main-memory join on modern hardware. *IEEE Trans. Knowl. Data Eng.*, 14(4):709–730, 2002.

[60] John C. McCallum. Memory prices (1957-2013). `http://www.jcmit.com/memoryprice.htm`. [Online reference; accessed June 11, 2013].

[61] *Microsoft SQL Server 2008 R2 Books Online: Isolation Levels in the Database Engine.* Available at http://msdn.microsoft.com/en-us/library/ms189122.aspx [Online reference; accessed June 11, 2013].

[62] David R. Musser. Introspective sorting and selection algorithms. *Softw., Pract. Exper.*, 27(8):983–993, 1997.

[63] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[64] *Extreme Performance Using Oracle TimesTen In-Memory Database*, July 2009. An Oracle technical whitepaper.

[65] *Oracle Exalytics In-Memory Machine: A Brief Introduction*, October 2011.

[66] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.

[67] Christos H. Papadimitriou and Paris C. Kanellakis. On concurrency control by multiple versions. *ACM Transactions on Database Systems*, 9(1):89–99, 1984.

[68] Andrew Pavlo, Carlo Curino, and Stanley B. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD*, pages 61–72, 2012.

[69] Jun Rao and Kenneth A. Ross. Making B$^+$-trees cache conscious in main memory. In *SIGMOD Conference*, pages 475–486, 2000.

[70] Kenneth A. Ross. Efficient hash probes on modern processors. In *ICDE*, pages 1297–1301, 2007.

[71] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *SIGMOD*, 2010.

[72] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. Cache conscious algorithms for relational query processing. In *VLDB*, pages 510–521, 1994.

[73] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.

[74] Michael Stonebraker. The case for shared nothing. In *HPTS*, 1985.

[75] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[76] *Telecommunication Application Transaction Processing (TATP) Benchmark Description.* Available at `http://tatpbenchmark.sourceforge.net` [Online reference; accessed June 11, 2013].

[77] Alexander Thomasian. Concurrency control: Methods, performance, and analysis. *ACM Computing Surveys*, 30(1):70–119, 1998.

[78] Khai Q. Tran, Spyros Blanas, and Jeffrey F. Naughton. On Hardware Transactional Memory, spinlocks, and database transactions. In *ADMS*, 2010.

[79] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann, 2002. ISBN 1-55860-508-8.

[80] Marcin Zukowski and Peter A. Boncz. Vectorwise: Beyond column stores. *IEEE Data Engineering Bulletin*, 35(1):21–27, 2012.