Chapter 10

Implementation and Pragmatic Issues

"All theory, my friend, is grey; but the precious tree of life is green." (Johann Wolfgang von Goethe)

10.1 Goal and Overview of the Chapter

So far we have laid emphasis on a conceptual view of algorithms and techniques so as to separate out the fundamental principles from the implementation issues in a specific system environment. In this chapter we provide an overview of the most important among such additional implementation and other pragmatic techniques that systems builders need to cope with.

We begin with implementation techniques for locking and transient versioning in Sections 10.2, 10.3, and 10.4. Next we consider the special aspect of lock management for multi-threaded transactions (i.e., with parallel threads within a single transaction) in Section 10.5. We discuss a variety of tuning techniques for concurrency control, both at the system and application level, in Section 10.6, and finally dedicate Section 10.7 to the specific issue of overload control.

10.2 Data Structures of a Lock Manager

As pointed out several times already, locking is the concurrency control method of choice for a wide spectrum of workloads and system settings. To implement an industrial-strength lock manager requires careful considerations on the underlying data structures, which are driven by the following two key requirements:

- When a lock is requested, we need an efficient way of checking whether a conflicting lock is already held by another transaction.
- When a lock is released, transactions that earlier requested conflicting locks and have therefore been suspended should now be considered for being resumed.
- When a transaction terminates, all locks (still) held on behalf of the transaction are released at once.

The first requirement dictates that we need an in-memory search structure for locks with associative access on the identifiers of the resources that are locked or requested to be locked. The most efficient choice for this purpose is a *hash table.* Recall that almost all locking protocols that we have covered in this book and that are used in practice are designed so as to map whatever abstract resources they need to lock (e.g., key ranges in an index) to concrete resources with unique identifiers (e.g., individual keys that occur in the index). Thus, it is indeed sufficient to use a single-key hash table which merely supports exact-match search on identifiers. The entries in the hash table then are pointers to some form of *resource control blocks (RCBs)* which are either dynamically allocated, or performance-wise even better, taken from a preallocated pool of such control blocks. Hash conflicts are most often resolved by linking all RCBs with the same value of the hash function into a chain anchored at the corresponding entry of the hash table. With this kind of data structure, checking for a lock conflict merely requires a hash-table lookup.

resource control blocks

With shared locks, multiple locks can be simultaneously held for the same resource, and multiple lock requests may be waiting to be granted for that resource. The bookkeeping for this kind of situation amounts to managing a *queue* of *lock control blocks (LCBs)* which are attached to the same RCB. This queue can in turn be easily implemented as a linked list anchored at the RCB, with the ordering in the list determining the order in which waiting lock requests should be considered for being resumed. So the linked list of LCBs would typically have zero, one, or more LCBs with shared locks at its front, followed by at least one request for an exclusive lock, which could then be followed by an arbitrary number of both shared or exclusive lock requests. Note that, for fairness reasons (i.e., to avoid starvation), a shared lock request should, in most situations, not be allowed to pass a previously issued exclusive request that is already waiting for one or more shared locks to be released.

lock control blocks

Finally, the third requirement stated above makes it necessary to quickly identify all LCBs that belong to the same transaction. This can be implemented in a straightforward manner by maintaining a *transaction control block (TCB)* for each active transaction and including in the TCB the anchor of a linked list of the transaction's LCBs. Then, upon the commit or abort of the transaction, one merely needs to traverse the list of LCBs, which should be removed from the queues of the corresponding RCBs. The successor LCBs in these resource-specific queues are the candidates for being resumed (in queue order).

transaction control blocks

This entire set of fairly simple data structures is illustrated in Figure 10.1. Whenever these structures are manipulated by the lock manager on behalf of a transaction, either upon a lock request or a lock release, some form of semaphore or "latch" (i.e., a flag that can be tested and set by an indivisible instruction) is needed to ensure the lock manager's internal consistency in a multithreaded data server. To avoid that this critical section becomes a performance bottleneck, typically each entry in the hash table has its own latch, often implemented as a bit in the hash-table entry itself.

In conclusion of this section, we point out that the outlined implementation is extremely versatile in the sense that it can be used for whatever locking units are considered appropriate. So RCBs can equally represent pages, records, index entries, or whatever "semantic locks" are needed for advanced object-model locking, as long as the conflict test itself merely requires checking the equality of identifiers (as opposed to checking, for example, the disjointness of predicates or the inclusion of a value in a value interval). In the context of layered (or general object-model) locking, either all locks of all layers could be managed in the same lock table or multiple layer-specific lock tables could be instantiated.

10.3 Multiple-Granularity Locking and Dynamic Escalation

From the previous section, it follows that a lock manager may create some non-negligible overhead in terms of memory consumption. Although the overhead for the various data structures is modest, a heavily loaded data server may end up with hundreds of TCBs and many thousands of RCBs and LCBs simultaneously, thus possibly occupying up to several Megabytes of memory. Although this would still be a small amount relative to the large memory sizes of today's amply equipped servers, the point arises as to whether this memory fraction could and should be better used for other purposes, especially for caching data, rather than mere bookkeeping. To this end, it would be convenient to allow the lock manager to choose dynamically

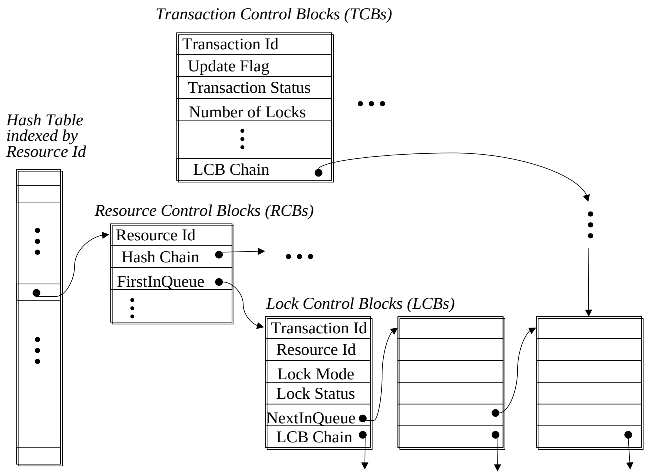

Figure 10.1: Data structures of a lock manager.

between fine-grained and coarse-grained locking, depending on the characteristics of the active transactions. Particularly, for long transactions that would acquire many fine-grained locks such as record and index-key locks, it may be worthwhile to reduce the space overhead of the lock manager at the expense of diminishing the possible concurrency. When concurrency is most critical, however, fine-grained locking would still be the method of choice.

reconciling coarse-grained and fine-grained locking

The tension between fine-grained and coarse-grained locking granularities can be reconciled by supporting both kinds of locks, or possibly an entire hierarchy of locking granularities, in a form of *multiple-granularity locking*. The practically most important case of this approach is to support the co-existence of record or page locks on the one hand, and locks on entire tables or tablespaces on the other. A *tablespace* is a logical unit of storage that can hold several database tables or indexes, and that may be subdevided into smaller units such as *containers* which in turn may be a file-system directory, a physical file, or a device such as a hard disk.

intention locks

The problem that now arises is to test conflicts among such disparate granularities. For example, when a transaction requests a lock on an entire tablespace, how can we quickly find out if this request conflicts with some page locks that are already held? While this is not exactly a deep issue, system architects have developed a specific solution with the benefit of a particularly low overhead, which is always a concern as locking is part of a server's innermost processing loops. The solution is based on so-called *intention locks* to be set on the coarser granularities of the hierarchy by those transactions which prefer fine-grained locking. These locks serve to mark a path in the granularity hierarchy that leads to the items that are actually locked in shared or exclusive mode. This way, other transactions that prefer coarse-grained locking can easily detect conflicts by testing their shared or exclusive locks against the intention locks on the coarser granularities. This consideration leads to the following lock-mode compatibility matrix between shared and exclusive locks, on one hand, and the two corresponding intention lock modes, called IS and IX, on the other hand:

	S	X	IS	IX	SIX
S	+	-	+	-	-
X	-	-	-	-	-
IS	+	-	+	+	+
IX	-	-	+	+	-
SIX	-	-	+	-	-

A common special case that has been added to the above table in the form of another lock mode coined SIX, are table scans that read a large number of pages and may update a very small fraction of these pages. For such transactions, page

locking should be ruled out for the sake of lower overhead, but locking the entire tablespace would require an exclusive lock and thus prevent all other transactions from reading any page of that tablespace. The escape from this dilemma is to have such a transaction acquire both a shared lock and an IX lock on the entire tablespace, so that individual pages can still be locked in the exclusive mode when they turn out to have updates. These two locks can be conveniently combined into a single new lock mode, coined SIX in the above table. Its compatibility properties are derived from the conjunction of an S lock and an IX lock.

Driven by such a lock compatibility table, the protocol for multiple-granularity locking simply boils down to the following two rules (in addition to the usual rules for 2PL or whatever lock protocol is employed):

1. A transaction can lock any granule in S or X mode, depending on whether only reads or writes are intended.
2. Before a granule can be locked in S or X mode, the transaction has to hold an IS or IX lock, respectively, on all coarser granules that contain the one to be locked.

lock escalation

Sometimes it is hard to choose among fine-grained or coarse-grained locking when a transaction starts. It may turn out only in the course of the transaction execution that it acquires a large number of locks and may create too much overhead in the lock manager. In such a situation, we would wish to dynamically convert the fine-grained locks already held into a single (or a few) coarse-grained lock(s). Such a conversion is known as *lock escalation* in the systems literature. It consists of first requesting a coarser-grained lock that implicitly comprises all currently held fine-grained locks, and only then the fine-grained locks can be released in a second step. So, although locking then is no longer two-phase in a narrow technical sense, the resulting behavior is as if the transaction would still hold its prior fine-grained locks thus is equivalent to two-phase locking. In commercial systems, lock escalation is typically triggered by exceptions that are raised when the total amount of memory for RCBs and LCBs or the number of locks of an individual transaction exceed certain thresholds.

10.4 Transient Versioning

As explained in great detail in Chapter 5, transiently keeping around versions of modified pages, records, or index entries provides opportunities for enhanced concurrency by means of a multiversion concurrency control protocol. The most practical among the protocols presented in Chapter 5 is the read-only multiversion protocol

(ROMV) that applies standard two-phase locking to update transactions and assigns versions to the retrieval operations of read-only transactions based on timestamps. More specifically, as you may recall from Chapter 5, new versions are timestamped with the commit time of the transaction that creates them, and read-only transactions read the most recent committed versions as of the transaction's begin.

In this section we sketch some implementation details for this kind of protocol. One of its salient properties is that it can be efficiently implemented at both the access layer, with versions of records and index entries, and the page layer. Here we focus on transient versioning of records, as this is the technique that is actually implemented in several commercial database systems. Other multiversion concurrency control protocols, e.g., the 2V2PL protocol of Chapter 5, would use similar implementation techniques and are not explicitly covered here.

version pool

The key concept for transient record versioning is to keep the most recent version of a record (with a given RID) always in the place where the record would reside if there were no versioning, and to store all older versions in a separate area called the *version pool*, which may reside in memory or on disk. This way the most recent data stays organized as compact as possible so that transactions that access only current versions do not lose efficiency because of versioning. In particular, if the current data is organized by some specific clustering scheme, say with records physically placed into pages in ascending order of a specific sorting key, this organization is in no way affected by the existence of old versions. Another advantage of a separate version pool is that it is much easier to perform garbage collection. In some systems, the version pool is even combined with the organization of recovery-related, temporary information (so-called logging entries, as we will see in Part III of the book) and therefore referred to as "rollback segments".

Each version carries, as two extra fields, its creation timestamp and a pointer to the previous version of the same RID (or the same index key when versioning is applied to index entries). An update operation on a record (or index entry) copies the pre-update version of the record as a new version into the version pool, overwrites the current record, and maintains the linked chain among the record's versions. At commit time of the corresponding update transaction, all modified records are timestamped. This deferred timestamping is feasible as readers would never be directed to an uncommitted version anyway. When a record is deleted, its last state is copied to the version pool, and a specially flagged "marker version" is kept in the place where the record used to reside. Figure 10.2 depicts this organization of the version pool. Here the record with RID 1141 was deleted at time 9:33; so its current version simply is a marker with a deleted flag. Versions with no predecessor, such as the current version of the record with RID 1135 or the version of RID 1143 that was created at time 9:20, were created by the insertion of a new record.

A read-only transaction, with a timestamp that reflects its begin, simply accesses

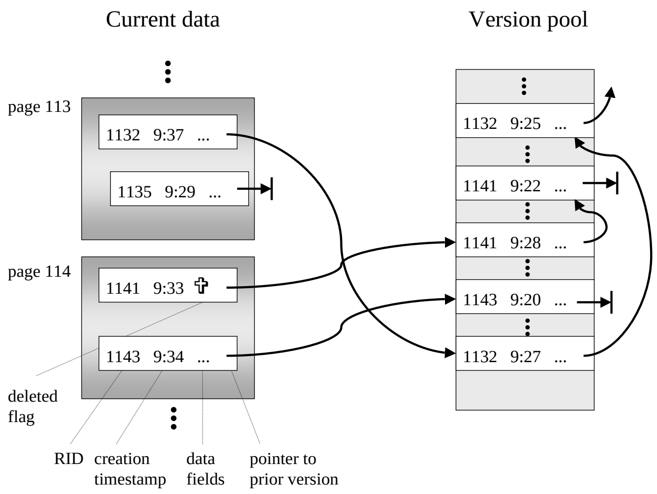

Figure 10.2: Storage organization for transient versioning.

the current versions of records and index entries that it requests to read, and then follows the corresponding linked chain until it finds a version whose timestamp is smaller than the transaction's timestamp. For example, with the data shown in Figure 10.2, a read-only transaction with timestamp 9:26 that requests to access RID 1141 would follow the version chain of the record's current version back to the version with creation timestamp 9:22 and return that version as the result of the read operation.

Instead of a linked chain for the versions of a given record, an alternative organization would be to store within the current version a small *version selection table* that contains the timestamps of and pointers to the older versions. The non-current versions themselves would again be kept in a separate version pool. This organization is particularly attractive when the number of versions for the same record is typically very small, say two or three, or the underlying multiversion concurrency control protocol even enforces such a limit for the number of simultaneously existing versions. In this case it may be even considered to keep the non-current versions of a record or index entry in the same page as the current version, the justification being that this storage overhead is low for a strictly limited number of versions. A typical scenario could then be that out of say 100 records in a page, 90 records have only a current version and 10 have one or at most two additional non-current versions.

garbage collection

Garbage collection is needed to identify and discard old versions that are no longer needed, and to reclaim the storage space used by those versions. With the ROMV protocol the identification of obsolete versions is straightforward: all non-current versions that have a successor (i.e., more recent version) whose timestamp is smaller than the timestamp of the oldest active read-only transaction can be safely discarded.

When versions are kept in the version pool in ascending order of their successor's creation timestamp, garbage collection can be implemented by keeping a roving pointer that marks the logical begin of the version pool as follows. Assume that the oldest active read-only transaction has timestamp TS. Then a non-current version with timestamp less than TS is of interest and must not yet be garbage-collected only if its successor version has timestamp greater than TS (i.e., if it is the youngest version with timestamp less than TS for the given record). Phrased positively, all non-current versions whose successor in the corresponding record's chain has timestamp less than TS can be safely discarded. Thus, by organizing the non-current versions *in ascending order of successor timestamps* and assuming that timestamp TS is itself not assigned to any version (as timestamps are unique among all transactions), the version whose successor has the smallest timestamp larger than TS divides the version pool into the garbage part and the "live" part that must be kept. Note that this ordering of versions in the version pool is, in fact, the most natural order as versions are moved into the version pool at the time when their successor is created.

In the example of Figure 10.2, the five versions shown in the version pool have ascending successor timestamps, namely, 9:27, 9:28, 9:33, 9:34, and 9:37. Thus, assuming that the timestamp of the oldest active read-only transaction is 9:30, the oldest two versions among the five shown versions can be garbage collected (i.e., the ones for RID 1132 with timestamp 9:25 and RID 1141 with timestamp 9:22). The actual storage reclamation is easily implemented by simply moving the logical begin pointer of the version pool to the first version that must be kept (i.e., the one for RID 1141 with timestamp 9:28 if only the five shown version were present).

If storage space is not a bottleneck then the garbage collection should discard only versions which are definitely obsolete in the sense that no transaction would ever have to access them. When storage is scarce, which could especially be the case when the version pool resides in memory or some specifically reserved, limited-size disk area, garbage collection may be forced to drop old versions more eagerly. Then it may happen that a long read-only transaction with a fairly old timestamp would have to access, for consistency, an old version that has been garbage-collected already. In such cases the data server would simply generate an error message to the application program and abort the transaction. In some commercial database systems this event is known as a "snapshot too old" message.

10.5 Nested Transactions for Intra-Transaction Parallelism

When a transaction consists of multiple, parallel threads, locking protocols need to be refined a little bit in order to ensure that different threads of the same transaction are serialized as well. Otherwise, although different transactions could still be isolated, the execution within a transaction might no longer correspond to a serial computation. Note that this was no problem at all from the abstract viewpoint of the correctness criteria for concurrency control, as these simply treated such cases of intra-transaction parallelism by partial orders. The purpose of this section is to add the necessary protocol elements so that two-phase locking can cope with parallel threads inside a transaction.

multi-threaded transactions, aka. nested transactions

In the literature, *multi-threaded transactions* are often referred to as *nested transactions* in that parallel threads form subtrees of a common parent that asynchronously spawned the threads. This consideration naturally leads to a tree whose leaves are the actual read and write data operations; siblings in the tree correspond to parallel threads. This computational model obviously bears resemblence to our notion of object-model transactions, and can indeed be cast into the object model. The difference, however, is that with nested transactions, we can usually not associate a semantically richer operation with an inner node of a transaction tree. So as far as concurrency control is concerned, the leaves of a nested transaction tree is all we need to consider; the inner nodes merely reflect parallel execution of threads.

The additional issue that arises in a nested transaction is that two parallel threads need to be synchronized in order to make sure that their combined execution is equivalent to a serial one. Since we assume a locking protocol anyway, the solution is rather straightforward: each thread, also called a "subtransaction" in this context, acquires locks according to the 2PL protocol. This automatically serializes parallel threads of the same transaction. When a thread terminates, it could allow a sibling, i.e., parallel thread of the same transaction, to proceed even if that sibling needs locks that were held by the terminated thread. After all, all these threads belong to the same transaction; so there is no need for isolating threads beyond their termination, rather threads can be viewed as cooperative light-weight processes. However, the locks of the terminated thread cannot simply be released; they are still needed to prevent other transactions from potentially creating conflict cycles. The technical solution is to keep these locks by having the parent of a thread *"inherit"* them upon the thread's termination. Then, of course, we need to reconsider the rules for lock conflicts: within a transaction, a thread's lock request can be in conflict only with a parallel thread, not with an already terminated one. So if a conflicting lock is held by an ancestor of the requesting thread, the lock can be safely granted.

These considerations lead to the following set of protocol rules:

2PL for nested transactions

- The leaves of a transaction tree acquire locks as needed by the corresponding data operations. These locks are held according to the 2PL for the duration of the transaction.
- Upon terminating a thread, all locks held by the thread are inherited by its parent.
- A lock request by a thread is granted if no conflicting lock on the same data item is currently held or the only conflicting locks are held by ancestors of the thread.

This protocol again resembles the concept of retained locks used in the general object-model 2PL. Note, however, that the retained locks there serve an entirely different purpose, namely, to synchronize different transactions that operate on different abstraction levels. In contrast, inherited locks as we prefer to call them, serve the intra-transaction synchronization among parallel threads of the same transaction. The specific objective of the nested transaction 2PL is to serialize the threads of a transaction tree, as stated more precisely in the following theorem:

Theorem 10.1

> The 2PL protocol for nested transactions generates only schedules which are equivalent to a serial execution of transactions where each transaction executes all its sibling sets serially.

We finally mention that parallel threads in a transaction can, of course, be combined with object-model operation-invocation trees. The protocol for this setting would then need both higher-level, operation-type-specific "semantic" locks on objects, possibly even in the form of retained locks, and lock inheritance of leaf-level, typically page-level, locks. Although this combination sounds complicated, it is fairly straightforward to reconcile the rules of both protocols to implement the combined case.

10.6 Tuning Options

Two-phase locking is widely used for the concurrency control of commercial data servers. Its popularity primarily draws on the simplicity, versatility, and robustness of the protocol. So, unless we have specific information about an application's workload characteristics, 2PL is indeed the method of choice in most cases. In a given application setting, however, it may be desirable to tune the use of 2PL towards specific access patterns and application needs for enhanced concurrency and/or reduced overhead. To this end, most commercial data servers provide the administration staff and sometimes even application developers with a number of "tuning knobs" that refer to the locking protocol. This section serves to point out the most important of these tuning options. It should be stressed, however, that appropriate use of these options may be a delicate matter. When used inappropriately, tuning options may even lead to inconsistent data. So the techniques presented below are for true application and system experts only, and even the experts should use extreme caution when they exert options that deviate from standard protocols like 2PL.

10.6.1 Manual Locking

Many commercial data servers allow application developers to choose among different settings for locking granularity, mode, and duration dynamically when a transaction starts or even upon each client-server interaction (e.g., an SQL command) that initiates locking steps.

locking granularity

Locking granularity can be influenced by choosing one of the granularities supported under a multigranularity locking protocol, for example, table locks versus page or record locks. Typically, a system administrator can specify a preferred locking granularity for each transaction program individually. In addition, the application programmer may sometimes specify the locking granularity dynamically on a per-program-invocation basis. In this case, the command that opens a new transaction would be parameterized to choose either coarse-grained or fine-grained locking. The application developer can, for example, program this decision based on user-provided input parameters to the program invocation, as these may affect the estimated number of locks the transaction would have to acquire.

lock modes

Lock conversions, i.e., "upgrading" a shared lock into an exclusive lock on the same object, are a frequent cause of deadlocks. It may be more desirable to block one transaction earlier rather than risking a deadlock. To this end, some servers provide commands by which a transaction program can advise the server to acquire a stronger exclusive lock earlier. For example, when transactions iterate over query result sets via an SQL command like "open cursor" (and subsequent "fetch" calls), they may have to update a small fraction of the retrieved data records. Usually, the server would first acquire merely shared locks on these records and would convert these into exclusive locks only on those records that need to be updated and only when the update command is issued. To alleviate the deadlock probability for multiple, concurrent instances of this transaction type, the server may consider acquiring exclusive locks on the retrieved records upfront (and possibly "downgrading" them to shared locks as soon as it is clear that a record is not updated). It is, however, very hard for the server to determine when this form of preclaiming (i.e., acquiring stronger locks ahead of time) is beneficial and when it would be better to tolerate the lock conversions. Therefore, programs can provide guidance to the server via special commands or additional parameters to the usual commands. These parameters may be implicit; for example, the SQL command "open cursor" has a variant of the form "open cursor for update", which is interpreted by some servers as a hint to acquire exclusive locks in the first place.

lock durations

The dynamic tuning of lock modes sketched above already influences the duration for which exclusive locks are held. Some servers are even more liberal in this regard in that they allow programs to "manually" acquire locks at specific times and possibly also release locks when the program(mer) considers the situation "safe" enough. So rather than having the server acquiring and releasing all locks automatically on behalf of the transaction program's usual calls (e.g., SQL commands), programs can direct the server to manage locks via explicit "lock" and "unlock" calls. An example for this kind of extremely flexible, but also potentially dangerous tuning capability can be found in the concurrency services of CORBA-style object request brokers. As it is then the program(mer) that determines the actual locking protocol, consistency-preservation would be completely dependent on the overview and care of the application architect or developers. Thus, as far as the comfort and ease of use of transactions is concerned, such tuning options are actually a step backwards,

but they may occasionally be convenient for performance improvements of special applications.

10.6.2 SQL Isolation Levels

The idea of "manually" controlling the lock duration on a per-application or even a per-transaction basis, discussed in the previous section, can be made safer by allowing only a limited number of "locking style" options that the application architect or developer can choose from. These options are known as *isolation levels*, one of them being the usual notion of conflict-serializability, and have even been incorporated into the SQL standard. A specific isolation level is chosen by issuing a corresponding SQL command "set isolation level …".

The isolation levels supported by the SQL standard are defined in terms of controlled deviations from (strongly) strict two-phase locking. They should be understood, however, as schedule classes which, albeit generated by specific locking protocols, can be enforced by whatever appropriate concurrency control algorithm a server chooses to employ. The practically most important isolation levels are:

Definition 10.2 (Isolation levels)

A schedule s is said to run under isolation level *read uncommitted* (also known as "dirty read" or "browse" level) if write locks are acquired and released according to S2PL, i.e., all write locks are held until the end of a transaction.

A schedule s is said to run under isolation level *read committed* (also known as "cursor stability" level) if write locks are acquired and released according to S2PL, and read locks are held (at least) for the duration of each data-server operation issued by a client.

A schedule s is said to run under isolation level *(conflict-) serializability* if it can be generated by the S2PL protocol.

isolation level read uncommitted

The three isolation levels differ in their lock durations for shared and exclusive locks. Essentially, read uncommitted does not require any read locks, and is useful for mere browsing or statistical evaluations (e.g., computing the average price over several thousand books, where the few books whose price is changed during the statistical analysis do not matter much) where a consistent view of the data is not required. The long-duration write locks superficially appear to prevent the persistent data from becoming inconsistent, but this, in fact, cannot be guaranteed, since the data values that are written may now depend on arbitrarily inconsistent, even "dirty" (i.e., uncommitted and later aborted), reads.

The read committed level alleviates such problems to some extent; in particular, it eliminates dirty-read anomalies. It is characterized by short read locks and long write locks. Thus, it reduces the possible data contention especially between long readers and short update transactions. This is a particularly useful and widely used option in practice, but it must be used with extreme caution as it can actually render the persistent data inconsistent. In fact, the read committed isolation level is still susceptible to lost-update anomalies, as the following example shows:

isolation level read committed

$$r_1(x)r_2(x)w_2(x)c_2w_1(x)c1$$

Some commercial data servers and even the SQL standard further distinguish between the level of full (conflict-) serializability and a relaxed, ad-hoc form of serializability that would tolerate phantom problems but "no other" types of inconsistencies. The latter is often referred to as the "repeatable read" level, although this is sometimes also used as a synonym to serializability. Obviously, this ad-hoc notion is not even precisely defined, as the category of "other" inconsistencies is left somewhat vague.

isolation levels serializability vs. repeatable read

As mentioned earlier, the above isolation levels are defined in terms of locking rules, but we can surely design other kinds of concurrency control algorithms that enforce a specific isolation level. If we wanted to employ even multiversion concurrency control protocols, the notion of a conflict that underlies the locking rules does not carry over directly. To this end, some commercial systems have introduced additional isolation levels that assume a multiversion concurrency control algorithm and specify a controlled relaxation from multiversion serializability.

multiversion read committed; snapshot isolation

Definition 10.3 (Multiversion read committed and snapshot isolation levels)

A transaction run under the *multiversion read committed* isolation level reads the most recent versions of the requested data items that were committed at the time when the read operation is issued. All writes that the transaction may invoke are subject to (non-versioned) exclusive locking with locks held until the transaction's termination.

For a transaction run under the *snapshot isolation* level, all operations read the most recent versions as of the time when the transaction began. In addition, the write sets of each pair of concurrent transactions must be disjoint.

Among these two levels, snapshot isolation is the stronger one. While multiversion read committed is still susceptible to lost updates (despite the fact that it has consis-

tent reads), snapshot isolation is fairly close to full serializability. In fact, if it were applied only to read-only transactions, then all transactions would have a consistent view of the data, and the persistent data itself could be guaranteed to remain consistent. This can be easily seen by observing that the read-only multiversion (ROMV) protocol introduced in Chapter 5 generates histories that exactly match the above definition of snapshot isolation. However, the following history (with data-item subscripts denoting versions) shows that snapshot isolation for update transactions can lead to situations that are not multiversion-serializable:

$$r_1(x_0)r_1(y_0)r_2(x_0)r_2(y_0)w_1(x_1)c_1w_2(y_2)c_2$$

As a concrete interpretation, assume that x and y are two numerical data items, both of which have the initial value 5 and are interrelated by a consistency constraint dictating $x + y \geq 0$. Now imagine two concurrent transactions, both of which read x and y and subtract 10 from one of the two data items. Under the read uncommitted level, this could result in the above history, which is not serializable and would result in both data items having a value of -5, thus violating the consistency constraint. A nice property of snapshot isolation, on the other hand, is that it can be formalized and studied in a rigorous manner. Our above definition can be cast into the theory of multiversion concurrency control (see Chapter 5) as follows:

formal definition of snapshot isolation

Definition 10.4 (Formal definition of snapshot isolation)

A multiversion schedule of transactions $T = \{t_1, \ldots, t_n\}$ satisfies the criterion of *snapshot isolation* if the following two conditions hold:

1. The version function maps each read action $r_i(x)$ to the most recent committed write action $w_j(x)$ as of the time of the begin of t_i; that is: $r_i(x)$ is mapped to $w_j(x)$ such that $w_j(x) < c_j < b_i < r_i(x)$ and there are no other actions $w_h(x)$ and c_h ($h \neq j$) with $w_h(x) < b_i$ and $c_j < c_h < b_i$, where b_i denotes the beginning of t_i.
2. The writesets of two concurrent transactions are disjoint; that is, if for two transactions t_i and t_j, either $b_i < b_j < c_i$ or $b_j < b_i < c_j$, then t_i and t_j must not write a common object.

The class of snapshot-isolated schedules and the class MVSR are incomparable. On the one hand, snapshot isolation is a special case of MVSR in that it uses a particular version function, and it is more restrictive because of the write-set disjointness condition. On the other hand, snapshot isolation does not require a reads-from relation that is compatible with that of a serial monoversion schedule; it is more liberal than MVSR in this respect.

The criterion of snapshot isolation can be easily characterized by a graph construction:

Definition 10.5 (Snapshot-isolation serialization graph)

The snapshot-isolation serialization graph of a multiversion history s is a directed graph with the transactions as nodes and the following edges: For each operation $r_j(x_i)$ in the schedule there is an edge $t_i \to t_j$ labeled x. For each pair of operations $r_k(x_j)$ and $w_i(x_i)$ there is an edge

1. $t_i \to t_j$ if $c_i < c_j$ and
2. $t_k \to t_i$ if $c_j < c_i$

labeled x in both cases.

Theorem 10.6

Let s be a multiversion schedule that has a version function according to the snapshot isolation criterion and in which each write step on object x is preceded by a read step on x within the same transaction. Then the following holds:

1. s is MVSR iff its corresponding snapshot-isolation serialization graph is acyclic, and
2. s is snapshot-isolated iff there is no object x such that the corresponding snapshot-isolation graph has a cycle consisting only of edges labeled x.

Snapshot isolation can be implemented using the ROMV protocol introduced in Chapter 5, for all read operations (i.e., even those of update transactions) with an additional mechanism for ensuring the disjointness of the write sets of concurrent transactions. A straightforward approach for the latter would be to remember write sets and perform an explicit disjointness test upon the commit request of a transaction. If the test indicates overlap, the transaction is aborted. A better way, which can detect non-disjointness earlier and thus avoid wasting work in transactions that would eventually be aborted anyway, is to acquire locks for writes as usual and check these locks against the write sets of concurrent transactions that committed earlier. Note that this requires more than just the usual lock conflict test, as the already terminated transactions do no longer hold their locks; rather we need to consider the write locks that they held at their commit time.

10.6.3 Short Transactions

A well-known tuning principle is to strive for short transactions. The extent to which this is feasible depends, of course, on the application. Ideally, knowledge about the application semantics and structure of programs should be fed into automated methods, like “transaction chopping” discussed in Chapter 8, that decompose long transactions into shorter ones. In practice, however, such reasoning is more typically an intellectual task of the application designers. Not unfrequently, the considerations along these lines are fairly pragmatic and exploit the fact that humans are, to a large extent, indulgent creatures and users can tolerate and cope with certain “pathological” situations.

One particularly popular rule of thumb is that a transaction should never span beyond dialog steps with the human user (aka. “user I/O”), to avoid that locks are held during (or the scope of other concurrency control measures extends over) the user’s think time between receiving some output from the computer application and providing the next input. Rather a “well-behaved” transaction program should accept an input message, begin and execute a database transaction, commit this transactions, and deliver a single output message. So applications such as flight reservations are broken down into a sequence of independent transactions. Finding out whether a particular flight has seats available and reporting the result to the user (i.e., the airline customer or the airline agent who speaks to the customer) would be a first transaction; then, once the customer has made up her mind, the actual booking of the flight would be a second transaction. Thus, it is possible that the customer sees available seats, but the attempt to book them a few minutes or even seconds later could possibly fail with a notification that the flight is fully booked. Of course, there is the common practice to overbook flights, but there should be some reasonable limit (say 110 percent of the flight capacity) even with this relaxation. Twenty years ago, such experiences could indeed be made by airline customers, and this was considered acceptable by the airlines as it would happen infrequently enough and annoy only a tiny fraction of unlucky customers. Meanwhile most airlines seem to have fixed such inconveniences by taking care of the problem at the level of the application programs, while still employing the principle of short transactions. One solution could be to make a tentative reservation already when the availability is checked, by explicit update statements on the underlying database. This reservation would be associated with an expiration period, and if the customer does not finalize the booking within this period, the tentative reservation is automatically dropped by a background program.

This example shows that striving for short transactions is not a panacea, and often comes at the expense of additional complexity at the application level, if feasible at all. A related pragmatic approach is to structure application programs such that the more critical write locks are held for the shortest possible time period, by moving update statements towards the end of the transaction and thus acquiring these exclusive locks as late as possible. So in this regard, a "good" transaction would have a read phase for querying information first, defer updates to permanent data by "making notes" about what needs to be done in the program's temporary, private variables and data structures, and finally submitting the corresponding update statements to the data server in a short write phase right before the transaction commit. This approach closely resembles the optimistic concurrency control method discussed in Chapter 4, where writes were deferred by means of a private workspace. In contrast to this method where the workspace resides on the server and the deferral of writes is completely masked to the application programs, the outlined pragmatic approach requires additional programming discipline and efforts by application developers. On the other hand, such an application-level approach would work for all kinds of data servers regardless of whether the server uses locking or optimistic methods or whatever. Recall also that the workspace concept of optimistic concurrency control can be efficiently implemented for page-oriented methods, but may be impractical for record-oriented and index concurrency control. The deferral of update calls in the application program can be seen as a way of simulating optimistic concurrency control for finer granularities.

"optimistic locking"

The technique of postponing updates towards the end of a transaction, for the sake of short lock duration, is, in practice, often combined with choosing relaxed isolation levels for the read phase. Many application developers have adopted the habit of choosing merely "read committed" as isolation level. This does, of course, neither ensure consistent reads nor can it prevent lost updates. A widely used practice that eliminates at least lost updates and thus eliminates the worst possible inconsistencies (but can still not even guarantee truly consistent data under all circumstances) is known under the name "optimistic locking". So transactions read data without transaction-duration locks (hence the name of the technique) but acquire locks for the updates in the application program's write phase. To rule out the possibility of lost updates, the update statements should be modified such that they include a test of whether the objects to be updated have been changed since they were initially read within the same transaction.

This test usually requires specific programming conventions and additional efforts. One popular way of implementing it for embedded SQL programs is to extend each database object with an additional counter field that serves as a timestamp or version number. During the read phase, all queries need to retrieve the corresponding counter value, and the program must remember these values in local variables. In the write phase, all SQL update statements are extended by an extra condition in the WHERE clause that tests the counter field's current value for equality with the previously read and remembered value. If the test fails, the update statement will

not modify any data; the application program would be notified by an appropriate SQL return code and would thus know that another transaction has concurrently modified the database object in question. Then, the application program would abort the transaction, initiating its rollback. So, for example, the code for retrieving and later modifying an account balance, say after computing interests and fees, would have to be modified as follows:

```
SELECT Balance, Counter INTO :b, :c
FROM Accounts
WHERE AccountNo = :x;
...
compute interests and fees
set b to appropriate value
...
UPDATE Accounts
SET Balance = :b, Counter = Counter +1
WHERE AccountNo = :x
AND Counter = :c;
```

Note that the `Counter` field is not part of the application's original database design, but needs to be added to the schema to enable this kind of optimistic locking technique. This approach resembles the notion of snapshot isolation, but it is implemented at the application level rather than inside the server, and it is strictly weaker as it does not even guarantee consistent reads. Needless to say that, despite the popularity of this technique, we would not at all encourage application builders to use it, and we are skeptical that such tricks are practically viable in the long term. The popularity of the approach mostly comes from the fact that it allows smart application builders to develop multi-user applications with decent performance even if the underlying data server uses only page locking and would otherwise be a major bottleneck. The real solution, however, would be to use a better data server that employs high-performance state-of-the-art techniques as presented in the previous chapters of this book.

10.6.4 Limiting the Level of Multiprogramming

Locks, or with respect to more general scheduling algorithms, data items are an abstract resource that concurrent transactions are competing for, in order to ensure consistent views of the data. This competition is known as *data contention.* Like many other forms of resource contention (e.g., memory contention), data contention is susceptible to "performance disasters", also known as "thrashing" phenomena.

data-contention thrashing

The reason for *data-contention thrashing* is that with increasing concurrency the probability for a lock request not being granted and the duration for which a transaction needs to wait upon a lock conflict increase superlinearly. In fact, when too many transactions run concurrently, we may end up with a situation where most transactions are blocked because of lock conflicts and only a few transactions are still running. It can be shown, both mathematically and experimentally, that this problem arises even if no deadlocks occur (e.g., if lock requests were ordered in accordance with data-item numbers). Frequent deadlocks are an additive danger, possibly leading to CPU and disk I/O contention as well when many transactions need to be aborted and restarted for another trial.

To avoid performance disasters of the above kind, virtually all data servers allow administrators to limit the *multiprogramming level (MPL)* (also known as the "degree of multiprogramming"), which is the maximum number of transactions that are allowed to run concurrently. When this limit is reached, additionally arriving transactions are held in a *transaction admission queue*. In this case, one of these transactions is admitted for execution only upon the completion of a running transaction; the admission order typically is FIFO (also known as first-come-first-served or FCFS shortly) for fairness. Since the waiting time that a transaction spends in the transaction admission queue also adds to the response time that is perceived by the client, the MPL should be set as high as possible, but also sufficiently low so as to prevent data-contention thrashing (for the usual workload characteristics of the server). When the MPL is set too high and thrashing occurs, the transaction throughput drops sharply and response times increase drastically ultimately approaching infinity.

This behavior is illustrated in Figure 10.3, which shows typical curves for transaction throughput and mean response time as functions of the number of concurrently active transactions. So without limitation of the MPL, the server would operate in the catastrophic region on the right side of the charts, at least during load peaks. Figure 10.4 illustrates the impact of the MPL limit on a high server load, say with 100 users who continuously submit transactions (with virtually no think time, or equivalently in the order of 1000 users with several seconds think time). For overly conservative MPL, at the left end of the chart, many transactions spend a long time in the transaction admission queue, whereas for an unduely high MPL limit, at the right end of the chart, thrashing sets on.

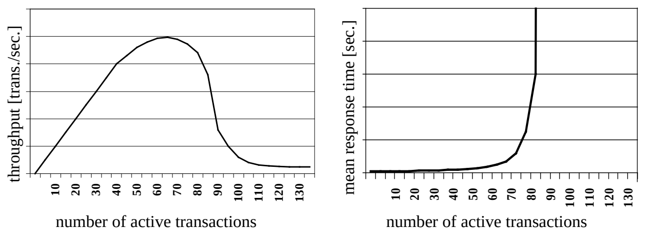

Figure 10.3: Illustration of data-contention thrashing.

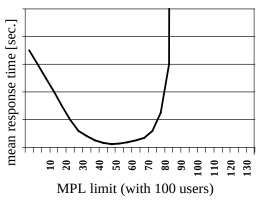

Figure 10.4: Impact of MPL limit.

It is a delicate tuning problem to determine an appropriate MPL setting for a given workload. For workloads with short, frequently arriving transactions that acquire only a few locks most of which are even read locks only, the MPL can and should be set quite high, up to a few hundred, for example. On the other hand, when some of the transactions in the workload are fairly long holding locks for an extended duration, or when their access patterns are highly skewed in that some very frequently updated data items form so-called "hot spots", a much lower MPL is needed, often less than ten. Even worse, the variability of transaction lengths in a mixed workload is an important factor to consider as well. Among two workloads that have the same mean duration of transactions one of which has constant transaction length whereas the other has a high variance, the one with the high variance of the lock duration is much more susceptible to data contention and the danger of thrashing. For this reason, some data servers or TP-monitor-style application servers even support MPL limitations on a per transaction-class basis. For example, in an electronic commerce application, one could specify that up to 10 concurrent order-entry transactions are acceptable, up to 20 concurrent payment transactions, but only up to 3 market-analysis transactions. In addition, one would potentially have to consider subsets of these transaction types as well, for example, by allowing only up to 8 concurrent transactions that are either order-entry or market-analysis transactions, and so on. These kinds of tuning decisions are extremely difficult. In practice they are often based on a mix of "guesswork" by experienced administrators and extensive trial-and-error experimentation before an application system is deployed for real-world service. Obviously, this makes tuning very expensive in terms of human time and cost, but it does correspond to the current state of the art as far as industry practice is concerned. In Section 10.7, we will present approaches towards automating this specific kind of tuning decisions, which will hopefully penetrate into commercial products soon.

10.7 Overload Control

The tuning of the multiprogramming level, as discussed in the previous section has the inherent disadvantage that it relies on human intervention. It requires a highly skilled and expensive administration staff, and reaches manageability limits with increasingly complex workloads. In addition, a fundamental drawback of manual tuning methods is that they do not allow the system to react to the dynamics of workloads. It may be necessary to limit the MPL only during particular peak-load periods with a particular mix of transactions, while the same limitation would be counterproductive during other periods. As these time periods are a priori unknown, one would need a system administrator who monitors the system continuously, sitting in a pilot seat with a control stick for real-time adjustments to the MPL limit. Such a scenario is clearly infeasible, and therefore automatic methods for preventing overload are called for. Appropriate methods, driven by feedback from the server's performance, are presented in this section.

10.7.1 Feedback-driven Method

The basic approach that we are going to present is a overload control method that is driven by feedback from the observed system performance, more specifically a metric coined the *conflict ratio* that reflects the current degree of data contention in the server. This method resembles the "system pilot" analogy given above, except that the administrator is replaced by a piece of software. The basic principle, illustrated in Figure 10.5, is a feedback loop consisting of observation, prediction, and reaction

steps. The method continuously monitors the conflict ratio or another appropriate metric. Based on these observations and possibly statistical knowledge about future transaction behavior (e.g., the expected number of locks yet to be acquired by the currently active transactions of specific types), the method predicts how the degree of data contention will evolve in the near future. When the data-contention indicator is about to exceed a critical threshold, the overload control considers two kinds of appropriate reactions:

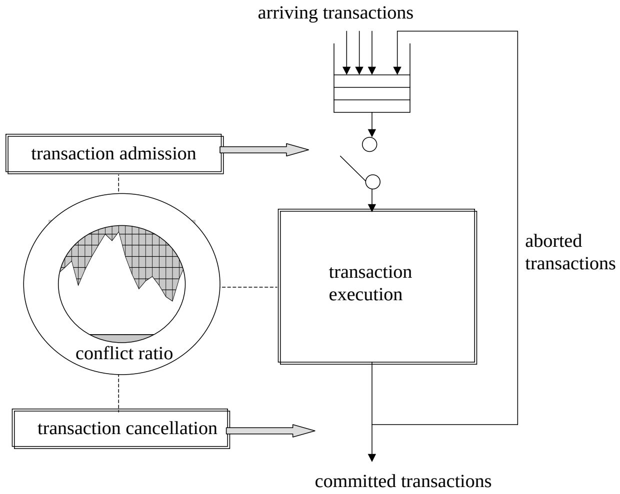

Figure 10.5: Overview of the conflict-ratio-driven load control method.

admission control and cancellation control

1. *Admission control:*
   Newly arriving transactions are no longer admitted for execution; rather they are held in a transaction admission queue. This measure effectively limits the multiprogramming level to its current value.

2. *Cancellation control:*
   In addition, it may be necessary to reduce the current multiprogramming level. This can be achieved by selecting a transaction as a cancellation victim, forcing it to abort and putting it back into the transaction admission queue. This step is repeated until the data-contention indicator drops below the critical threshold.

restart control

Transactions that have become cancellation victims should not be allowed to restart immediately, as this could easily lead to unstable, oscillating behavior with a high probability of immediately re-entering the zone of critical data contention or even thrashing. It has been shown to be beneficial that an aborted transaction, either a deadlock or a cancellation victim, should be restarted only after all transactions by which the victim was blocked are successfully terminated. This restart policy is sometimes referred to as *restart waiting*.

A related issue is the problem of how to select a cancellation victim. Following similar considerations on deadlock resolution (see Chapter 4), a heuristically good choice is the transaction that holds fewest locks, but only such transactions should be eligible which are blocked and do themselves block other transactions. Transactions that are not blocked are making progress and should be exempt from cancellation; likewise, transactions that do not block any other transaction do not cause much harm towards data-contention thrashing. As with deadlock victims, this kind of selection bears the danger of starvation, where the same transaction is chosen for cancellation over and over again (strictly speaking, different "incarnations" of the same transaction). To prevent starvation and ensure a certain fairness, the victim selection policy needs to be rectified: one practically viable possibility is to choose as a cancellation victim the transaction with the smallest product *number of locks currently held* $\times$ *number of previous restarts*.

The key decision in such a feedback-driven approach is the choice of an appropriate metric as a data-contention indicator and robust value for the critical threshold. For a homogeneous workload, where all transactions are of the same type and acquire approximately the same number of locks, it has been shown, both analytically and experimentally, that the *fraction of blocked transactions* is a good indicator for the degree of data contention. As long as the fraction of blocked transactions is below a critical value around 0.25, the server can maintain a high throughput. When this fraction exceeds the critical value, however, the risk of thrashing becomes unbearably high, and with more than 30 percent of the transactions being blocked the system is usually thrashing.

conflict ratio

Unfortunately, this metric is too simple to capture heterogeneous workloads with transactions of highly variable length. Depending on the number of locks that the various transactions hold, different values of the blocked-transaction metric can be tolerated, so that it is impossible to derive a critical threshold that is meaningful across a wide spectrum of workloads. Then, a more appropriate metric is the *conflict ratio*, defined as the following quotient:

$$\frac{total\ number\ of\ locks\ currently\ held\ by\ all\ transactions}{total\ number\ of\ locks\ currently\ held\ by\ all\ non-blocked\ transactions}.$$

Note that this metric implicitly attaches weights to the various, currently active transactions: transactions that hold many locks have a higher weight. When all transactions have the same number of locks, which is the expected steady-state situation under a homogeneous workload, the conflict ratio directly reflects the fraction of blocked transactions. With widely varying transaction lengths, however, it becomes crucial to factor the variability into the data-contention metric. Analytic modeling as well as experimental studies have shown that there exists a value for the conflict ratio, coined the *critical conflict ratio*, that can serve as a reliable indicator for the onset of thrashing, and that value is approximately the same, namely, between about 1.3, for a wide range of workload characteristics. So the critical conflict ratio can be viewed as a constant, and this renders the conflict ratio an appropriate metric for a robust, feedback-driven load control method.

The complete feedback-driven load control method, with the conflict ratio as a data-contention indicator, is described by the following pseudocode.

pseudocode for the conflict-ratio-driven load control method

```
upon the begin request of transaction t:
    if conflict ratio < critical conflict ratio
    then admit t
    else put t in the transaction admission queue
    fi
upon a lock wait of transaction t:
    update conflict ratio
    while not (conflict ratio < critical conflict ratio)
        among the transactions that are blocked and block other transactions
            determine the transaction v with the smallest product
            number of locks held * number of previous restarts
            (with the transaction with the highest product being exempt)
        abort v and put in the transaction admission queue
        if no eligible transaction found then exit loop fi
    od
upon the termination of transaction t:
    if conflict ratio < critical conflict ratio then
        for each transaction q in the transaction admission queue do
            if (q will be started the first time) or
               (q has been a deadlock victim or cancellation victim before and
                all transactions that q was waiting for in its previous execution
                have been successfully terminated)
            then admit q
            fi
        od
    fi
```

10.7.2 Wait-depth Limitation

An alternative approach to overload prevention that is a bit more aggressive in causing more transaction cancellations than the conflict-ratio-driven method is known as *wait-depth limitation*. The wait depth of a transaction is recursively defined as follows:

- a running (i.e., non-blocked) transaction has wait depth 0, and
- a transaction that is blocked by a transaction with wait depth i (and no transaction with higher wait depth) has wait depth $i + 1$.

The wait depth of the currently active transactions can be viewed as an indicator of data contention. In particular, a wait depth of 2 or higher indicates that one or

more blocked transactions do in turn block other transactions. Consequently, it has been suggested to cancel transactions whenever a wait depth of 1 is exceeded. A case analysis of these situations is illustrated in Figure 10.6, with existing lock waits indicated by solid edges and a newly arising lock wait by a dashed edge. In case 1 of the figure, transaction t_k should be canceled (and put into the transaction admission queue as in the conflict-ratio-driven method). In case 2, we can either cancel t_k or t_{i1} to bring the maximum wait depth down to 1. In that case, a subsidiary victim-selection policy should be used by cancelling, for example, the "troublemaker" t_k unless it is the transaction with the largest number of currently held locks in which case it appears more favorable to prioritize t_k and cancel t_{i1}.

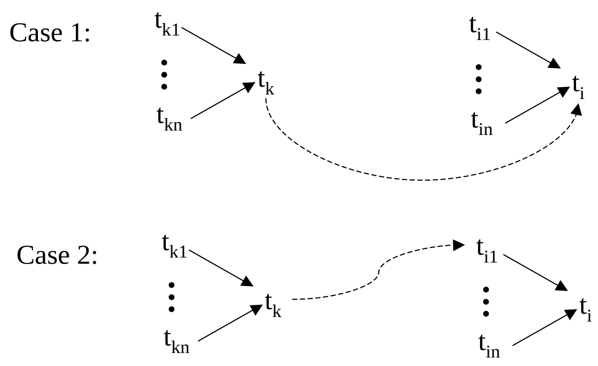

Figure 10.6: Case analysis for the wait-depth limitation method.

Compared to the conflict-ratio-driven load control, wait-depth limitation is a bit simpler to implement. When CPU and I/O resources are ample, its performance is about as good as that of the conflict-ratio-driven method. However, it tends to cancel more transactions under high data contention, and the resulting restarts may lead to a higher utilization of the CPU and the disks so that these resources could become bottlenecks in extreme situations.

10.8 Lessons Learned

Locking, as the concurrency control method used in most commercial systems, can be efficiently implemented with hash-based organization of lock control blocks. To control the overhead, especially the memory consumption, of the lock manager, locking protocols have been extended to cope with multiple granularities so that some loss in concurrency may be dynamically traded for reduced memory usage.

Locking controls can be enhanced by transient versioning and exploiting versions for read-only transactions. We have seen that the storage management for versioning can be implemented with affordable overhead, and that effective and efficient garbage collection is feasible.

Parallel threads within a transaction can be elegantly supported by the model of nested transactions, but there are also other ways of supporting intra-transaction parallelism within a lock manager. Nested transactions should not be confused with object-model transactions. The latter are sometimes referred to as "open nested transactions" because of the early release of low-level locks; nested transactions, which could then consequently be referred to as "closed nested transactions", do not have this salient property of increased concurrency among independent transactions.

For system administrators and, to some extent, also application developers a suite of tuning knobs are available in the form of system-provided options or appropriate structuring of application programs. In particular, the various choices for SQL isolation levels are widely used, especially, the read committed and the snapshot isolation levels. However, we would like to warn once again that these options should be used with extreme caution: if in doubt, do not use them. It is only in conjunction with specific properties of applications that they lead to acceptable behavior; if carelessly employed, they may lead to inconsistent data. So serializability (or multiversion serializability when transparent versioning is supported) remains by far the most appropriate choice for safe and most productive development of multi-user applications.

Finally, we have seen that all concurrency control methods are susceptible to a performance catastrophe known as data-contention "thrashing", during load peaks. To avoid thrashing and ensure that performance would deteriorate gracefully even under sudden load bursts, an explicit form of load control is needed. This is typically implemented by offering system administrators to limit the server's multiprogramming level, but better, "self-tuning", methods have been developed in the literature.

Exercises

1. Discuss to what extent lock escalation may incur deadlocks or livelocks.

2. Discuss if and under which conditions a form of lock de-escalation could make sense. De-escalation would essentially be the opposite of lock escalation, converting one (or a few) coarse-grained lock(s) into several fine-grained locks while a transaction is executing.

3. Give examples for schedules that fall into the following five isolation-level classes (but not into the next larger, surrounding class): not even read uncommitted, read uncommitted, read committed, repeatable read, serializability.

4. Give an example for a snapshot-isolated history violating the consistency of the persistent data. Explain what kinds of inconsistencies may arise.

5. Design a concurrency control protocol that can guarantee snapshot isolation (and can indeed generate all snapshot-isolated histories).

6. Give examples for showing that MVSR and snapshot isolation are incomparable classes of histories, i.e., neither one of them is included in the other.

Bibliographical and Other Remarks

The book by Gray and Reuter (1993) is an excellent source for the kind of pragmatic issues that we discussed in this chapter. Many of these issues, including multiple-granularity locking and isolation levels, can be traced back to the seminal papers by Gray et al. (1975) and Gray (1978). The analog of multi-granularity locks for optimistic and other non-locking protocols has been studied by Carey (1983). Generalizations of multi-granularity locking to low-overhead lock management for complex objects in CAD or CASE applications have been investigated by Herrmann et al. (1990), Cart and Ferrié (1990), and Rezende and Härder (1997).

Implementation techniques for transient versioning have first appeared in the work of Reed and Svobodova (1981) as well as Chan et al. (1982). More recent work that refines the various techniques and includes comparative performance studies has been done by Bober and Carey (1992a, 1992b, 1997). Mohan, Pirahesh, and Lorie

(1992) give a very detailed account of versioning for record- and index-entry-based concurrency control.

Nested transactions have been invented by Moss (1985) and later been studied by many other researchers including, for example, Haerder and Rothermel (1993) and Lynch et al. (1994).

Tuning considerations related to locking are surveyed by Shasha (1992). Good sources on application-level tuning for specific systems are Corrigan and Gurry (1996), Schneider (1999), and Mullins (2000).

Isolation levels, as adopted by the SQL standard, have been critically reviewed by Berenson et al. (1995), who have also pointed out the practical importance of snapshot isolation. Recent studies of snapshot isolation that include precise characterizations have been carried out by Schenkel et al. (2000) as well as Adya et al. (2000); the latter also casts other isolation levels into a formal framework.

Tay et al. (1985) were the first to analytically study and understand the fundamental phenomenon of data-contention thrashing. The specific issue of feedback-driven overload control has been studied by Carey et al. (1990), Heiss et al. (1991), and Moenkeberg et al. (1991, 1992). Our presentation is adopted from Weikum et al. (1994). Important analytic foundations for the conflict-ratio-based method have been laid by Thomasian (1993, 1996). The wait-depth limitation method has been invented by Franaszek et al. (1992); related work on "cautious waiting" has been carried out by Balter et al. (1982) and Hsu and Zhang (1992).