2 TUNING THE GUTS

2.1 Goal of Chapter

This chapter discusses tuning considerations having to do with the underlying components common to most database management systems. Each component carries its own tuning considerations.

- Concurrency control—how to minimize lock contention.
- Recovery and logging—how to minimize logging and dumping overhead.
- Operating system—how to optimize buffer size, process scheduling, and so on.
- Hardware—how to allocate disks, random access memory, and processors.

Figure 2.1 shows the common underlying components of all database systems.

➤WARNING ABOUT THIS CHAPTER This is the most difficult chapter of the book because we have written it under the assumption that you have only a passing understanding of concurrent systems. We want to lead you to a level where you can make subtle trade-offs between speed and concurrent correctness, between speed and fault tolerance, and between speed and hardware costs.

So, it may be tough going in spots. If you have trouble, take a break and read another chapter.

2.2 Locking and Concurrency Control

Database applications divide their work into *transactions*. When a transaction executes, it accesses the database and performs some local computation. The strongest assumption that an application programmer can make is that each transaction will

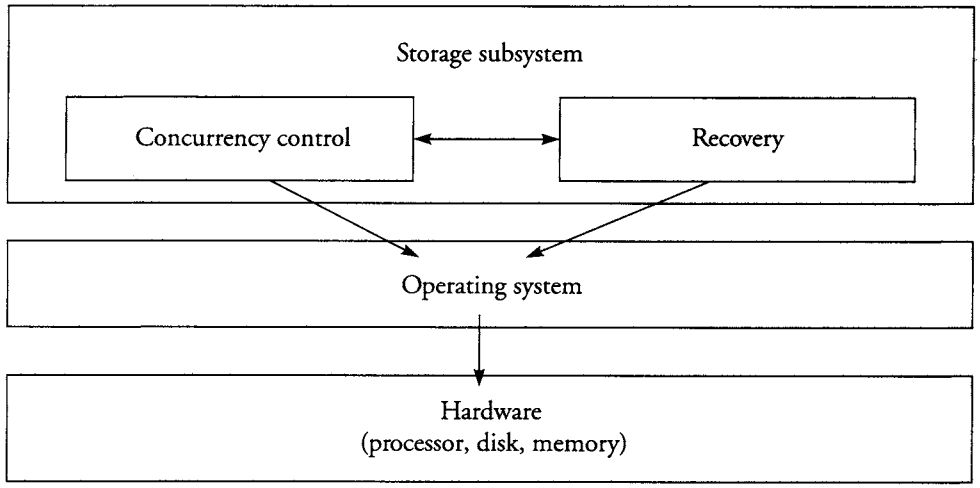

FIGURE 2.1 Underlying components of a database system.

appear to execute in isolation—without concurrent activity. Because this notion of isolation suggests indivisibility, transactional guarantees are sometimes called *atomicity guarantees.*[1] Database researchers, you see, didn't bother to let 20th-century physics interfere with their jargon.

The sequence of transactions within an application program, taken as a whole, enjoys no such guarantee, however. Between, say, the first two transactions executed by an application program, other application programs may have executed transactions that modified data items accessed by one or both of these first two transactions. For this reason, the length of a transaction can have important correctness implications.

---

1. Three excellent references on the theory of this subject are the following:

Phil Bernstein, Vassos Hadzilacos, and Nat Goodman, *Concurrency Control and Recovery in Database Systems.* Reading, Mass.: Addison-Wesley, 1987. Now in the public domain at: *http://research.microsoft.com/pubs/ccontrol/.*

Jim Gray and Andreas Reuter, *Transaction Processing: Concepts and Techniques.* San Francisco: Morgan Kaufmann, 1993.

Gerhard Weikum and Gottfried Vossen, *Transactional Information Systems: Theory, Algorithms, and Practice of Concurrency Control and Recovery.* San Francisco: Morgan Kaufmann, 2001.

EXAMPLE: THE LENGTH OF A TRANSACTION

Suppose that an application program processes a purchase by adding the value of the item to inventory and subtracting the money paid from cash. The application specification requires that cash never be made negative, so the transaction will roll back (undo its effects) if subtracting the money from cash will cause the cash balance to become negative.

To reduce the time locks are held, the application designers divide these two steps into two transactions.

1. The first transaction checks to see whether there is enough cash to pay for the item. If so, the first transaction adds the value of the item to inventory. Otherwise, abort the purchase application.
2. The second transaction subtracts the value of the item from cash.

They find that the cash field occasionally becomes negative. Can you see what might have happened?

Consider the following scenario. There is $100 in cash available when the first application program begins to execute. An item to be purchased costs $75. So, the first transaction commits. Then some other execution of this application program causes $50 to be removed from cash. When the first execution of the program commits its second transaction, cash will be in deficit by $25.

So, dividing the application into two transactions can result in an inconsistent database state. Once you see it, the problem is obvious though no amount of sequential testing would have revealed it. Most concurrent testing would not have revealed it either because the problem occurs rarely.

So, cutting up transactions may jeopardize correctness even while it improves performance. This tension between performance and correctness extends throughout the space of tuning options for concurrency control. You can make the following choices:

- The number of locks each transaction obtains (*fewer tends to be better for performance, all else being equal*).
- The kinds of locks those are (*read locks are better for performance*).
- The length of time that the transaction holds them (*shorter is better for performance*).

Because a rational setting of these tuning options depends on an understanding of correctness goals, the next section describes those goals and the general strategies employed by database systems to achieve them.

Having done that, we will be in a position to suggest ways of rationally trading performance for correctness.

2.2.1 Correctness Considerations

Two transactions are said to be *concurrent* if their executions overlap in time. That is, there is some point in time in which both transactions have begun, and neither has completed. Notice that two transactions can be concurrent even on a uniprocessor. For example, transaction T may begin, then may suspend after issuing its first disk access instruction, and then transaction T' may begin. In such a case, T and T' would be concurrent. Figure 2.2 shows an example with four transactions.

Concurrency control is, as the name suggests, the activity of controlling the interaction among concurrent transactions. Concurrency control has a simple correctness goal: make it *appear* as if each transaction executes in isolation from all others. That is, the execution of the transaction collection must be *equivalent* to one in which

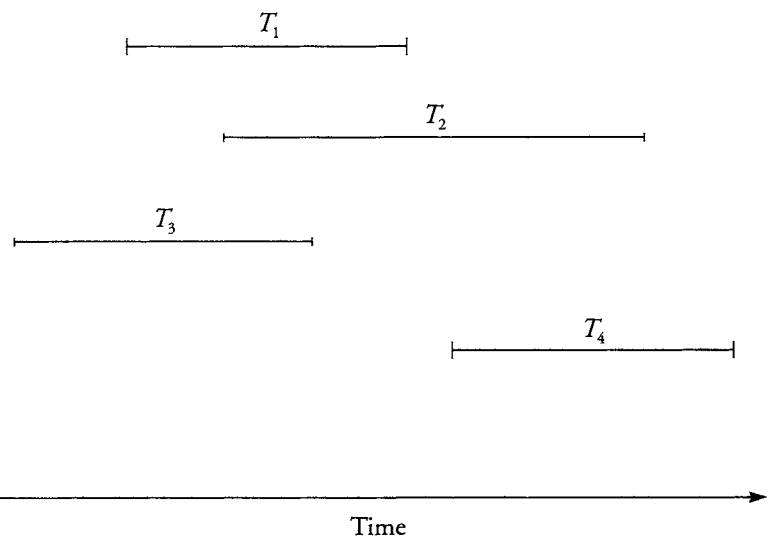

FIGURE 2.2 Example of concurrent transactions. T_1 is concurrent with T_2 and T_3. T_2 is concurrent with T_1, T_3, and T_4.

the transactions execute one at a time. Two executions are equivalent if one is a rearrangement of the other; and in the rearrangement, each read returns the same value, and each write stores the same value as in the original.

Notice that this correctness criterion says nothing about what transactions do. Concurrency control algorithms leave that up to the application programmer. That is, the application programmer must guarantee that the database will behave appropriately provided each transaction appears to execute in isolation. What is appropriate depends entirely on the application, so is outside the province of the concurrency control subsystem.

Concurrent correctness is usually achieved through mutual exclusion. Operating systems allow processes to use semaphores for this purpose. Using a semaphore S, a thread (or process) can access a resource R with the assurance that no other thread (or process) will access R concurrently.

A naive concurrency control mechanism for a database management system would be to have a single semaphore S. Every transaction would acquire S before accessing the database. Because only one transaction can hold S at any one time, only one transaction can access the database at that time. Such a mechanism would be correct but would perform badly at least for applications that access slower devices such as disks.

EXAMPLE: SEMAPHORE METHOD

Suppose that Bob and Alice stepped up to different automatic teller machines (ATMs) serving the same bank at about the same time. Suppose further that they both wanted to make deposits but into different accounts. Using the semaphore solution, Alice might hold the semaphore during her transaction during the several seconds that the database is updated and an envelope is fed into the ATM. This would prevent Bob (and any other person) from performing a bank transaction during those seconds. Modern banking would be completely infeasible.

Surprisingly, however, careful design could make the semaphore solution feasible. It all depends on when you consider the transaction to begin and end and on how big the database is relative to high-speed memory. Suppose the transaction begins after the user has made all decisions and ends once those decisions are recorded in the database, but before physical cash has moved. Suppose further that the database fits in memory

so the database update takes well under a millisecond. In such a case, the semaphore solution works nicely and is used in some main memory databases.

An overreaction to the Bob and Alice example would be to do away with the semaphore and to declare that no concurrency control at all is necessary. That can produce serious problems, however.

EXAMPLE: NO CONCURRENCY CONTROL

Imagine that Bob and Alice have decided to share an account. Suppose that Bob goes to a branch on the east end of town to deposit $100 in cash and Alice goes to a branch on the west end of town to deposit $500 in cash. Once again, they reach the automatic teller machines at about the same time. Before they begin, their account has a balance of 0. Here is the progression of events in time.

1. Bob selects the deposit option.
2. Alice selects the deposit option.
3. Alice puts the envelope with her money into the machine at her branch.
4. Bob does the same at his branch.
5. Alice's transaction begins and reads the current balance of $0.
6. Bob's transaction begins and reads the current balance of $0.
7. Alice's transaction writes a new balance of $500, then ends.
8. Bob's transaction writes a new balance of $100, then ends.

Naturally, Bob and Alice would be dismayed at this result. They expected to have $600 in their bank balance but have only $100 because Bob's transaction read the same original balance as Alice's. The bank might find some excuse ("excess static electricity" is one favorite), but the problem would be due to a lack of concurrency control.

So, semaphores on the entire database can give ruinous performance, and a complete lack of control gives manifestly incorrect results. Locking is a good compromise. There are two kinds of locks: *write* (also known as exclusive) locks and *read* (also known as shared) locks.

Write locks are like semaphores in that they give the right of exclusive access, except that they apply to only a portion of a database, for example, a page. Any such lockable portion is variously referred to as a *data item*, an *item*, or a *granule*.

Read locks allow shared access. Thus, many transactions may hold a read lock on a data item x at the same time, but only one transaction may hold a write lock on x at any given time.

Usually, database systems acquire and release locks implicitly using an algorithm known as *Two-Phase Locking*, invented at IBM Almaden research by K. P. Eswaran, J. Gray, R. Lorie, and I. Traiger in 1976. That algorithm follows two rules.

1. A transaction must hold a lock on x before accessing x. (That lock should be a write lock if the transaction writes x and a read lock otherwise.)
2. A transaction must not acquire a lock on any item y after releasing a lock on any item x. (In practice, locks are released when a transaction ends.)

The second rule may seem strange. After all, why should releasing a lock on, say, Ted's account prevent a transaction from later obtaining a lock on Carol's account? Consider the following example.

EXAMPLE: THE SECOND RULE AND THE PERILS OF RELEASING SHARED LOCKS

Suppose Figure 2.3 shows the original database state. Suppose further that there are two transactions. One transfers $1000 from Ted to Carol, and the other computes the sum of all deposits in the bank. Here is what happens.

1. The sum-of-deposits transaction obtains a read lock on Ted's account, reads the balance of $4000, then releases the lock.

ACCOUNT OWNER	BALANCE
Ted	4000
Bob and Alice	100
Carol	0

FIGURE 2.3 Original database state.

2. The transfer transaction obtains a lock on Ted's account, subtracts $1000, then obtains a lock on Carol's account and writes, establishing a balance of $1000 in her account.
3. The sum-of-deposits transaction obtains a read lock on Bob and Alice's account, reads the balance of $100, then releases the lock. Then that transaction obtains a read lock on Carol's account and reads the balance of $1000 (resulting from the transfer).

So, the sum-of-deposits transaction overestimates the amount of money that is deposited.

The previous example teaches two lessons.

- The two-phase condition should apply to reads as well as writes.
- Many systems (e.g., SQL Server, Sybase, etc.) give a default behavior in which write locks are held until the transaction completes, but read locks are released as soon as the data item is read (degree 2 isolation). This example shows that this can lead to faulty behavior. *This means that if you want concurrent correctness, you must sometimes request nondefault behavior.*

2.2.2 Lock Tuning

Lock tuning should proceed along several fronts.

1. Use special system facilities for long reads.
2. Eliminate locking when it is unnecessary.
3. Take advantage of transactional context to chop transactions into small pieces.
4. Weaken isolation guarantees when the application allows it.
5. Select the appropriate granularity of locking.
6. Change your data description data during quiet periods only. (Data Definition Language statements are considered harmful.)
7. Think about partitioning.
8. Circumvent hot spots.
9. Tune the deadlock interval.

You can apply each tuning suggestion independently of the others, but you must check that the appropriate isolation guarantees hold when you are done. The first

three suggestions require special care because they may jeopardize isolation guarantees if applied incorrectly.

For example, the first one offers full isolation (obviously) provided a transaction executes alone. It will give no isolation guarantee if there can be arbitrary concurrent transactions.

Use facilities for long reads

Some relational systems, such as Oracle, provide a facility whereby read-only queries hold no locks yet appear to execute serializably. The method they use is to re-create an old version of any data item that is changed after the read query begins. This gives the effect that a read-only transaction R reads the database *as the database appeared when R began* as illustrated in Figure 2.4.

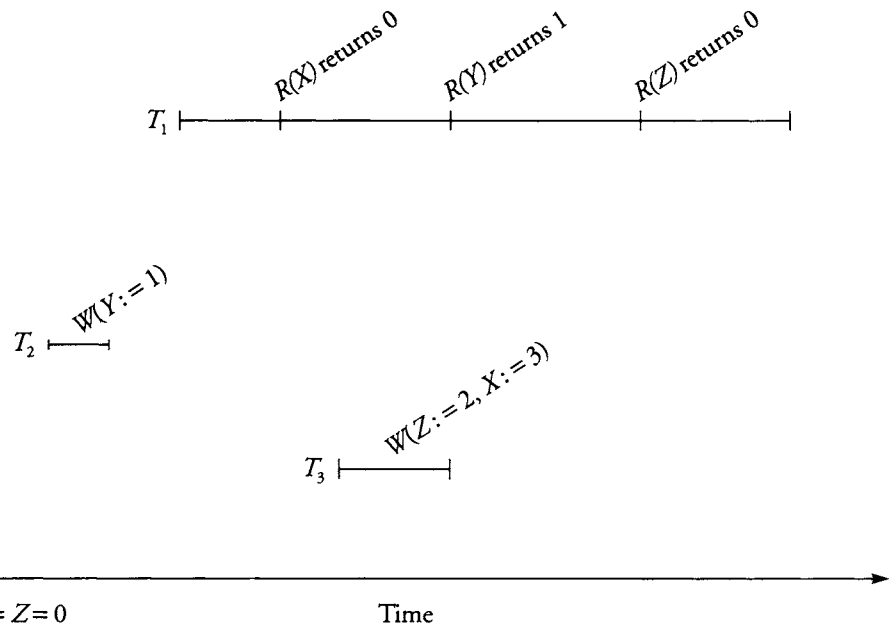

FIGURE 2.4 Multiversion read consistency. In this example, three transactions T_1, T_2, and T_3 access three data items X, Y, and Z. T_1 reads the values of X, Y, and Z (T_1: $R(X)$, $R(Y)$, $R(Z)$). T_2 sets the value of Y to 1 (T_2: $W(Y:=1)$). T_3 sets the value of Z to 2 and the value of X to 3 (T_3: $W(Z:=2)$, $W(X:=3)$). Initially X, Y, and Z are equal to 0. The diagram illustrates the fact that, using multiversion read consistency, T_1 returns the values that were set when it started.

Using this facility has the following implications:

- Read-only queries suffer no locking overhead.
- Read-only queries can execute in parallel and on the same data as short update transactions without causing blocking or deadlocks.
- There is some time and space overhead because the system must write and hold old versions of data that have been modified. The only data that will be saved, however, is that which is updated while the read-only query runs. Oracle snapshot isolation avoids this overhead by leveraging the before-image of modified data kept in the rollback segments.

Although this facility is very useful, please keep two caveats in mind.

1. When extended to read/write transactions, this method (then called snapshot isolation) does *not* guarantee correctness. To understand why, consider a pair of transactions

   $T_1 : x := y$
   $T_2 : y := x$

   Suppose that x has the initial value 3 and y has the initial value 17. In any serial execution in which T_1 precedes T_2, both x and y have the value 17 at the end. In any serial execution in which T_2 precedes T_1, both x and y have the value 3 at the end. But if we use snapshot isolation and T_1 and T_2 begin at the same time, then T_1 will read the value 17 from y without obtaining a read lock on y. Similarly, T_2 will read the value 3 from x without obtaining a read lock on x. T_1 will then write 17 into x. T_2 will write 3 into y. No serial execution would do this.

   What this example (and many similar examples) reveals is that database systems will, in the name of performance, allow bad things to happen. It is up to application programmers and database administrators to choose these performance optimizations with these risks in mind. A default recommendation is to use snapshot isolation for read-only transactions, but to ensure that read operations hold locks for transactions that perform updates.

2. In some cases, the space for the saved data may be too small. You may then face an unpleasant surprise in the form of a return code such as "snapshot too old." When automatic undo space management is activated, Oracle 9i allows you to configure how long before-images are kept for consistent read purpose. If this parameter is not set properly, there is a risk of "snapshot too old" failure.

Eliminate unnecessary locking

Locking is unnecessary in two cases.

1. When only one transaction runs at a time, for example, when loading the database
2. When all transactions are read-only, for example, when doing decision support queries on archival databases

In these cases, users should take advantage of options to reduce overhead by suppressing the acquisition of locks. (The overhead of lock acquisition consists of memory consumption for lock control blocks and processor time to process lock requests.) This may not provide an enormous performance gain, but the gain it does provide should be exploited.

Make your transaction short

The correctness guarantee that the concurrency control subsystem offers is given in units of transactions. At the highest degree of isolation, each transaction is guaranteed to appear as if it executed without being disturbed by concurrent transactions.

An important question is how long should a transaction be? This is important because transaction length has two effects on performance.

- The more locks a transaction requests, the more likely it is that it will have to wait for some other transaction to release a lock.
- The longer a transaction T executes, the more time another transaction will have to wait if it is blocked by T.

Thus, in situations in which blocking can occur (i.e., when there are concurrent transactions some of which update data), short transactions are better than long ones. Short transactions are generally better for logging reasons as well, as we will explain in the recovery section.

Sometimes, when you can characterize all the transactions that will occur during some time interval, you can "chop" transactions into shorter ones without losing isolation guarantees. Appendix B presents a systematic approach to achieve this goal. Here are some motivating examples along with some intuitive conclusions.

EXAMPLE: UPDATE BLOB WITH CREDIT CHECKS

A certain bank allows depositors to take out up to $1000 a day in cash. At night, one transaction (the "update blob" transaction) updates the balance

of every accessed account and then the appropriate branch balance. Further, throughout the night, there are occasional credit checks (read-only transactions) on individual depositors that touch only the depositor account (not the branch). The credit checks arrive at random times, so are not required to reflect the day's updates. The credit checks have extremely variable response times, depending on the progress of the updating transaction. What should the application designers do?

Solution 1: Divide the update blob transaction into many small update transactions, each of which updates one depositor's account and the appropriate branch balance. These can execute in parallel. The effect on the accounts will be the same because there will be no other updates at night. The credit check will still reflect the value of the account either before the day's update or after it.

Solution 2: You may have observed that the update transactions may now interfere with one another when they access the branch balances. If there are no other transactions, then those update transactions can be further subdivided into an update depositor account transaction and an update branch balance transaction.

Intuitively, each credit check acts independently in the previous example. Therefore, breaking up the update transaction causes no problem. Moreover, no transaction depends on the consistency between the account balance value and the branch balance, permitting the further subdivision cited in solution 2. Imagine now a variation of this example.

EXAMPLE: UPDATES AND BALANCES

Instead of including all updates to all accounts in one transaction, the application designers break them up into minitransactions, each of which updates a single depositor's account and the branch balance of the depositor's branch. The designers add an additional possible concurrent transaction that sums the account balances and the branch balances to see if they are equal. What should be done in this case?

Solution: The balance transaction can be broken up into several transactions. Each one would read the accounts in a single branch and the corresponding

branch balance. The updates to the account and to the branch balance may no longer be subdivided into two transactions, however. The reason is that the balance transaction may see the result of an update to a depositor account but not see the compensating update to the branch balance.

These examples teach a simple lesson: *whether or not a transaction T may be broken up into smaller transactions depends on what is concurrent with T.*

Informally, there are two questions to ask.

1. Will the transactions that are concurrent with T cause T to produce an inconsistent state or to observe an inconsistent value if T is broken up?

   That's what happened when the purchase transaction was broken up into two transactions in the example entitled "The Length of a Transaction" on page 11. Notice, however, that the purchase transaction in that example could have been broken up if it had been reorganized slightly. Suppose the purchase transaction first subtracted money from cash (rolling back if the subtraction made the balance negative) and then added the value of the item to inventory. Those two steps could become two transactions given the concurrent activity present in that example. The subtraction step can roll back the entire transaction before any other changes have been made to the database. Thus, rearranging a program to place the update that causes a rollback first may make a chopping possible.

2. Will the transactions that are concurrent with T be made inconsistent if T is broken up?

   That's what would happen if the balance transaction ran concurrently with the finest granularity of update transactions, that is, where each depositor-branch update transaction was divided into two transactions, one for the depositor and one for the branch balance.

Here is a rule of thumb that often helps when chopping transactions (it is a special case of the method to be presented in Appendix B):

Suppose transaction T accesses data X and Y, but any other transaction T' accesses at most one of X or Y and nothing else. Then T can be divided into two transactions, one of which accesses X and the other of which accesses Y.

This rule of thumb led us to break up the balance transaction into several transactions, each of which operates on the accounts of a single branch. This was possible because all the small update transactions worked on a depositor account and branch balance of the same branch.

➤CAVEAT Transaction chopping as advocated here and in Appendix B works correctly if properly applied. The important caution to keep in mind is that adding a new transaction to a set of existing transactions may invalidate all previously established choppings.

Weaken isolation guarantees carefully

Quite often, weakened isolation guarantees are sufficient. SQL offers the following options:

1. *Degree 0:* Reads may access *dirty data*, that is, data written by uncommitted transactions. If an uncommitted transaction aborts, then the read may have returned a value that was never installed in the database. Further, different reads by a single transaction to the same data will not be *repeatable*, that is, they may return different values. Writes may overwrite the dirty data of other transactions. A transaction holds a write lock on x while writing x, then releases the lock immediately thereafter.
2. *Degree 1—read uncommitted:* Reads may read dirty data and will not be repeatable. Writes may not overwrite other transactions' dirty data.
3. *Degree 2—read committed:* Reads may access only committed data, but reads are still not repeatable because an access to a data item x at time T_2 may read from a different committed transaction than the earlier access to x at T_1. In a classical locking implementation of degree 2 isolation, a transaction acquires and releases write locks according to two-phase locking, but releases the read lock immediately after reading it. Relational systems offer a slightly stronger guarantee known as *cursor stability*: during the time a transaction holds a cursor (a pointer into an array of rows returned by a query), it holds its read locks. This is normally the time that it takes a single SQL statement to execute.
4. *Degree 3—serializable:* Reads may access only committed data, and reads are repeatable. Writes may not overwrite other transactions' dirty data. The execution is equivalent to one in which each committed transaction executes in isolation, one at a time. Note that ANSI SQL makes a distinction between repeatable read and serializable isolation levels. Using repeatable read, a transaction T_1 can insert or delete tuples in a relation that is scanned by transaction T_2, but T_2 may see only some of those changes, and as a result the execution is not serializable. The serializable level ensures that transactions appear to execute in isolation and is the only truly safe condition.

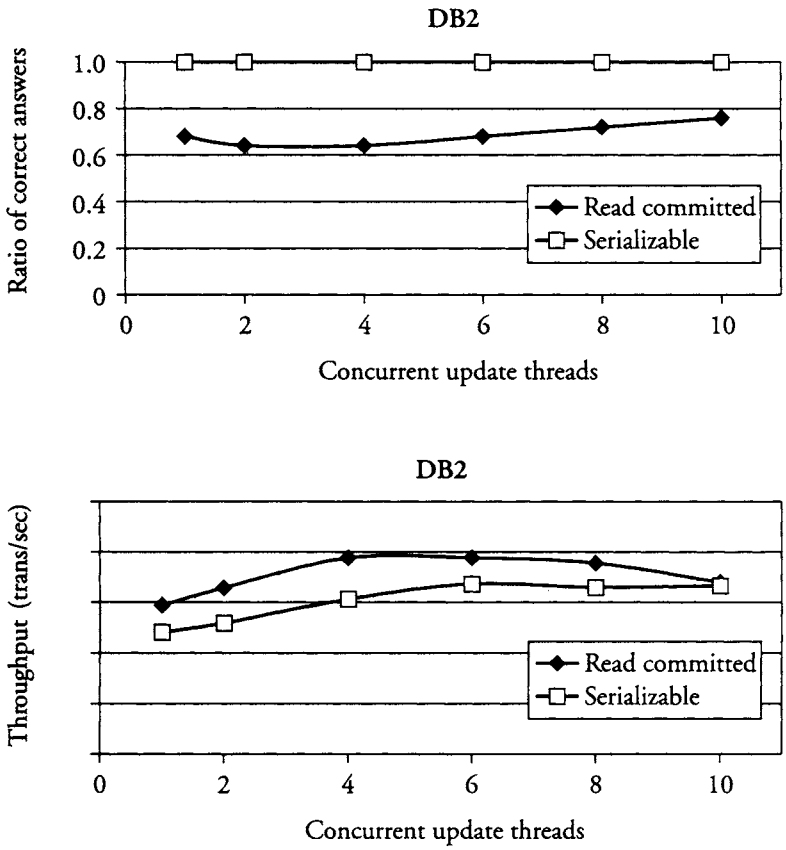

FIGURE 2.5 Value of serializability (DB2 UDB). A summation query is run concurrently with swapping transactions (a read followed by a write in each transaction). The read committed isolation level does not guarantee that the summation query returns correct answers. The serializable isolation level guarantees correct answers at the cost of decreased throughput. These graphs were obtained using DB2 UDB V7.1 on Windows 2000.

Figures 2.5, 2.6, and 2.7 illustrate the risks of a weak isolation level and the value of serializability. The experiment simply consists of a summation query repeatedly executed together with a fixed number of money transfer transactions. Each money transfer transaction subtracts (debits) a fixed amount from an account and adds (credits) the same amount to another account. This way, the sum of all account balances remains constant. The number of threads that execute these update transactions is our parameter. We use row locking throughout the experiment. The serializable isolation

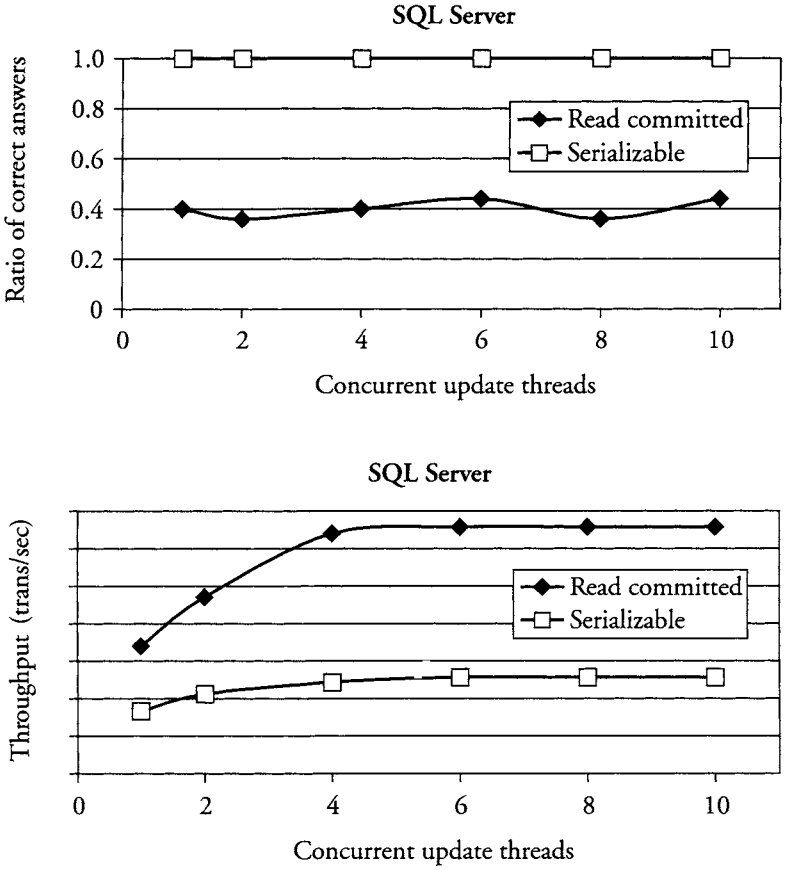

FIGURE 2.6 Value of serializability (SQL Server). A summation query is run concurrently with swapping transactions (a read followed by a write in each transaction). Using the read committed isolation level, the ratio of correct answers is low. In comparison, the serializable isolation level always returns a correct answer. The high throughput achieved with read committed thus comes at the cost of incorrect answers. These graphs were obtained using SQL Server 7 on Windows 2000.

level guarantees that the sum of the account balances is computed in isolation from the update transactions. By contrast, the read committed isolation allows the sum of the account balances to be computed after a debit operation has taken place but before the corresponding credit operation is performed. The bottom line is that applications that use any isolation level less strong than serializability can return incorrect answers.

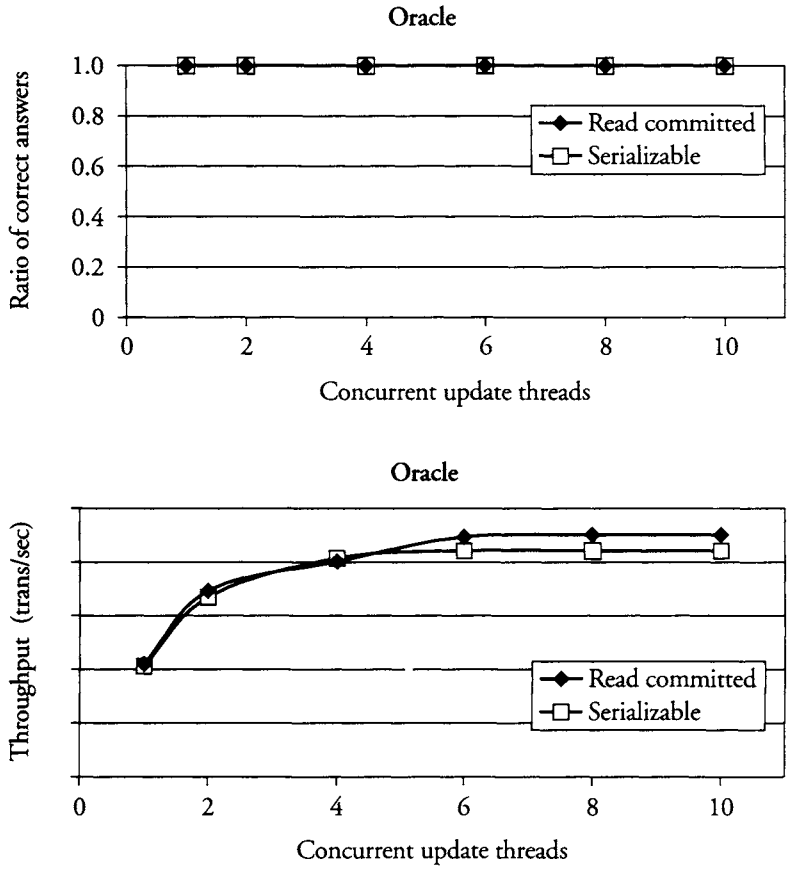

FIGURE 2.7 Value of serializability (Oracle). A summation query is run concurrently with swapping transactions (a read followed by a write in each transaction). In this case, Oracle's snapshot isolation protocol guarantees that the correct answer to the summation query is returned regardless of the isolation level because each update follows a read on the same data item. Snapshot isolation would have violated correctness had the writes been "blind." Snapshot isolation is further described in the next section on facilities for long reads. These graphs were obtained using Oracle 8i EE on Windows 2000.

Some transactions do not require exact answers and, hence, do not require degree 3 isolation. For example, consider a statistical study transaction that counts the number of depositor account balances that are over $1000. Because an exact answer is not required, such a transaction need not keep read locks. Degree 2 or even degree 1 isolation may be enough.

The lesson is this. *Begin with the highest degree of isolation (serializable in relational systems). If a given transaction (usually a long one) either suffers extensive deadlocks or causes significant blocking, consider weakening the degree of isolation, but do so with the awareness that the answers may be off slightly.*

Another situation in which correctness guarantees may be sacrificed occurs when a transaction includes human interaction and the transaction holds hot data.

EXAMPLE: AIRLINE RESERVATIONS

A reservation involves three steps.

1. Retrieve the list of seats available.
2. Determine which seat the customer wants.
3. Secure that seat.

If all this were encapsulated in a single transaction, then that transaction would hold the lock on the list of seats in a given plane while a reservations agent talks to a customer. At busy times, many other customers and agents might be made to wait.

To avoid this intolerable situation, airline reservation systems break up the act of booking a seat into two transactions and a nontransactional interlude. The first transaction reads the list of seats available. Then there is human interaction between the reservation agent and the customer. The second transaction secures a seat that was chosen by a customer. Because of concurrent activity, the customer may be told during step 2 that seat S is available and then be told after step 3 that seat S could not be secured. This happens rarely enough to be considered acceptable. Concurrency control provides the lesser guarantee that no two passengers will secure the same seat.

It turns out that many airlines go even farther, allocating different seats to different cities in order to reduce contention. So, if you want a window seat in your trip from New York to San Francisco and the New York office has none, consider calling the San Francisco office.

Control the granularity of locking

Most modern database management systems offer different "granularities" of locks. The default is normally record-level locking, also called row-level locking. A *page-level lock* will prevent concurrent transactions from accessing (if the page-level lock is a write lock) or modifying (if the page-level lock is a read lock) all records on that page. A *table-level lock* will prevent concurrent transactions from accessing or modifying (depending on the kind of lock) all pages that are part of that table and, by extension, all the records on those pages. Record-level locking is said to be *finer grained* than page-level locking, which, in turn, is finer grained than table-level locking.

If you were to ask the average application programmer on the street whether, say, record-level locking was better than page-level locking, he or she would probably say, "Yes, of course. Record-level locking will permit two different transactions to access different records on the same page. It must be better."

By and large this response is correct for online transaction environments where each transaction accesses only a few records spread on different pages.

Surprisingly, for most modern systems, the locking overhead is low even if many records are locked as the insert transaction in Figure 2.8 shows.

There are three reasons to ask for table locks. First, table locks can be used to avoid blocking long transactions. Figures 2.9, 2.10, and 2.11 illustrate the interaction of a long transaction (a summation query) with multiple short transactions (debit/credit transfers). Second, they can be used to avoid deadlocks. Finally, they reduce locking overhead in the case that there is no concurrency.

SQL Server 7 and DB2 UDB V7.1 provide a lock escalation mechanism that automatically upgrades row-level locks into a single table-level lock when the number of row-level locks reaches a predefined threshold. This mechanism can create a deadlock if a long transaction tries to upgrade its row-level locks into a table-level lock while concurrent update or even read transactions are waiting for locks on rows of that table. This is why Oracle does not support lock escalation. Explicit table-level locking by the user will prevent such deadlocks at the cost of blocking row locking transactions.

The conclusion is simple. *Long transactions should use table locks mostly to avoid deadlocks, and short transactions should use record locks to enhance concurrency.* Transaction length here is relative to the size of the table at hand: a long transaction is one that accesses nearly all the pages of the table.

There are basically three tuning knobs that the user can manipulate to control granule size.

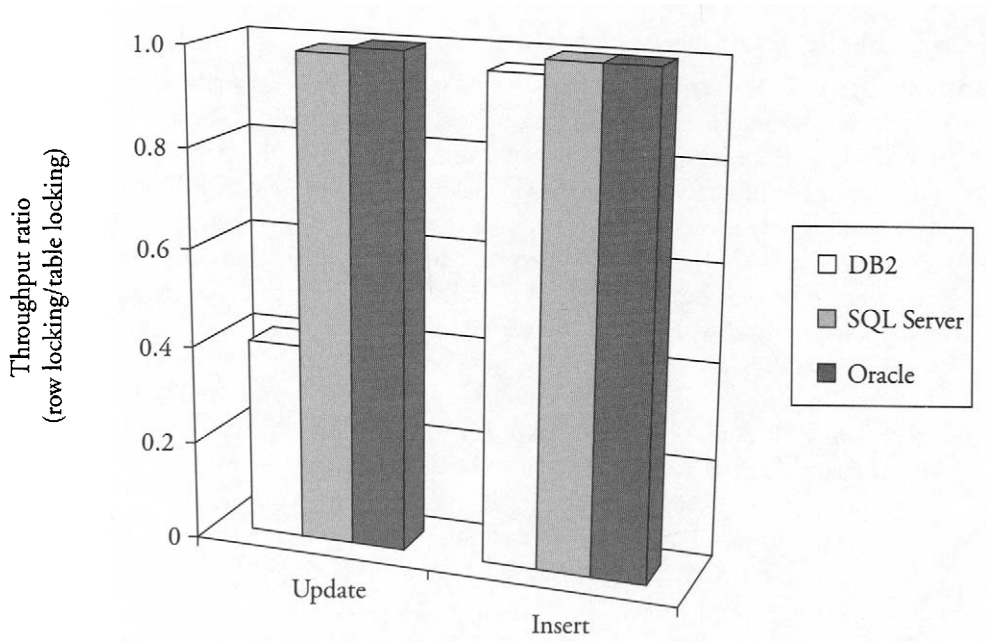

FIGURE 2.8 Locking overhead. We use two transactions to evaluate how locking overhead affects performance: an update transaction updates 100,000 rows in the accounts table while an insert transaction inserts 100,000 rows in this table. The transaction commits only after all updates or inserts have been performed. The intrinsic performance costs of row locking and table locking are negligible because recovery overhead (the logging of updates) is so much higher than locking overhead. The exception is DB2 on updates because that system does "logical logging" (instead of logging images of changed data, it logs the operation that caused the change). In that case, the recovery overhead is low and the locking overhead is perceptible. This graph was obtained using DB2 UDB V7.1, SQL Server 7, and Oracle 8i EE on Windows 2000.

1. *Explicit control of the granularity*
   - *Within a transaction:* A statement within a transaction explicitly requests a table-level lock in shared or exclusive mode (Oracle, DB2).
   - *Across transactions:* A command defines the lock granularity for a table or a group of tables (SQL Server). All transactions that access these tables use the same lock granularity.
2. *Setting the escalation point:* Systems that support lock escalation acquire the default (finest) granularity lock until the number of acquired locks exceeds some threshold set by the database administrator. At that point, the next coarser

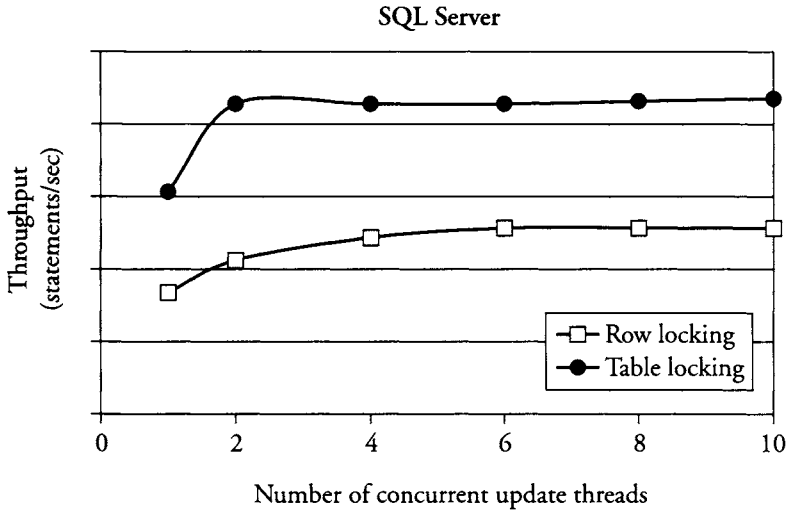

FIGURE 2.9 Fine-grained locking (SQL Server). A long transaction (a summation query) runs concurrently with multiple short transactions (debit/credit transfers). The serializable isolation level is used to guarantee that the summation query returns correct answers. In order to guarantee a serializable isolation level, row locking forces the use of key range locks (clustered indexes are sparse in SQL Server, thus key range locks involve multiple rows; see Chapter 3 for a description of sparse indexes). In this case, key range locks do not increase concurrency significantly compared to table locks while they force the execution of summation queries to be stopped and resumed. As a result, with this workload table locking performs better. Note that in SQL Server the granularity of locking is defined by configuring the table; that is, all transactions accessing a table use the same lock granularity. This graph was obtained using SQL Server 7 on Windows 2000.

granularity lock will be acquired. The general rule of thumb is to set the threshold high enough so that in an online environment of relatively short transactions, escalation will never take place.

3. *Size of the lock table:* If the administrator selects a small lock table size, the system will be forced to escalate the lock granularity even if all transactions are short.

Data definition language (DDL) statements are considered harmful

Data definition data (also known as the system catalog or metadata) is information about table names, column widths, and so on. DDL is the language used to access and manipulate that table data. Catalog data must be accessed by every transaction

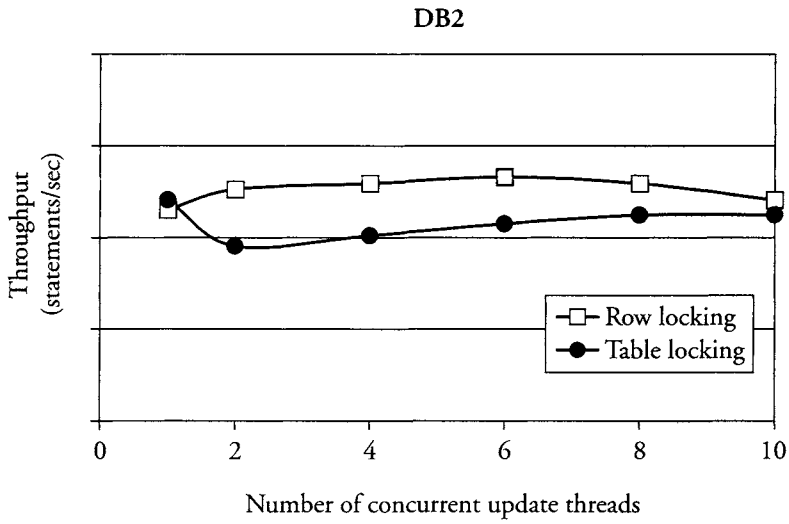

FIGURE 2.10 Fine-grained locking (DB2). A long transaction (a summation query) with multiple short transactions (debit/credit transfers). Row locking performs slightly better than table locking. Note that by default DB2 automatically selects the granularity of locking depending on the access method selected by the optimizer. For instance, when a table scan is performed (no index is used) in serializable mode, then a table lock is acquired. Here an index scan is performed and row locks are acquired unless table locking is forced using the LOCK TABLE command. This graph was obtained using DB2 UDB V7.1 on Windows 2000.

that performs a compilation, adds or removes a table, adds or removes an index, or changes an attribute description. As a result, the catalog can easily become a hot spot and therefore a bottleneck. A general recommendation therefore is to avoid updates to the system catalog during heavy system activity, especially if you are using dynamic SQL (which must read the catalog when it is parsed).

Think about partitioning

One of the principles from Chapter 1 held that partitioning breaks bottlenecks. Overcoming concurrent contention requires frequent application of this principle.

EXAMPLE: INSERTION TO HISTORY

If all insertions to a data collection go to the last page of the file containing that collection, then the last page may, in some cases, be a concurrency control bottleneck. This is often the case for history files and security files.

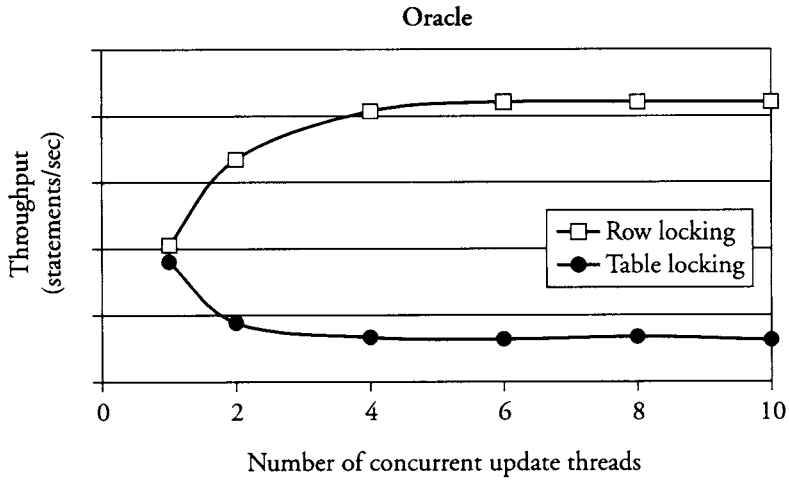

FIGURE 2.11 Fine-grained locking (Oracle). A long transaction (a summation query) with multiple short transactions (debit/credit transfers). Because snapshot isolation is used the summation query does not conflict with the debit/credit transfers. Table locking forces debit/credit transactions to wait, which is rare in the case of row locking. As a result, the throughput is significantly lower with table locking. This graph was obtained using Oracle 8i EE on Windows 2000.

A good strategy is to partition insertions to the file across different pages and possibly different disks.

This strategy requires some criteria for distributing the inserts. Here are some possibilities.

1. Set up many insertion points and insert into them randomly. This will work provided the file is essentially write-only (like a history file) or whose only readers are scans.
2. Set up a clustering index based on some attribute that is not correlated with the time of insertion. (If the attribute's values are correlated with the time of insertion, then use a hash data structure as the clustering index. If you have only a B-tree available, then hash the time of insertion and use that as the clustering key.) In that way, different inserted records will likely be put into different pages.

You might wonder how much partitioning to specify. A good rule of thumb is to specify at least $n/4$ insertion points, where n is the maximum number of concurrent transactions writing to the potential bottleneck.

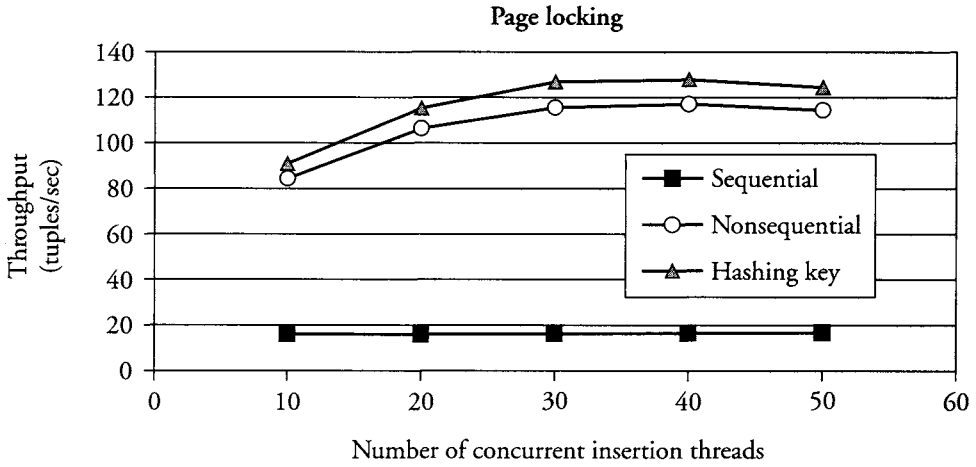

FIGURE 2.12 Multiple insertion points and page locking. There is contention when data is inserted in a heap or when there is a sequential key and the index is a B-tree: all insertions are performed on the same page. Use multiple insertion points to solve this problem. This graph was obtained using SQL Server 7 on Windows 2000.

Figures 2.12 and 2.13 illustrate the impact of the number of insertion points on performance. An insertion point is a position where a tuple may be inserted in a table. In the figures, "sequential" denotes the case where a clustered index is defined on an attribute whose value increases (or even decreases) with time. "Nonsequential" denotes the case where a clustered index is defined on an attribute whose values are independent of time. Hashing denotes the case where a composite clustered index is defined on a key composed of an integer generated in the range of 1 . . . k (k should be a prime number) and of the attribute on which the input data is sorted.

With page locking, the number of insertion points makes a difference: sequential keys cause all insertions to target the same page; as a result, contention is maximal. Row locking ensures that a new insertion point is assigned for each insert regardless of the method of insertion and hence eliminates the problem.

EXAMPLE: FREE LISTS

Free lists are data structures that govern the allocation and deallocation of real memory pages in buffers. Locks on free lists are held only as long as

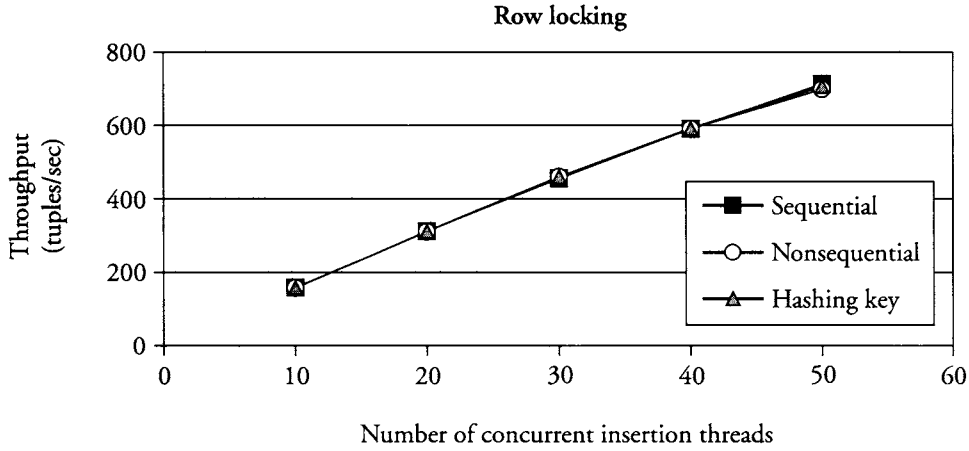

FIGURE 2.13 Multiple insertion points and row locking. Row locking avoids contention between successive insertions. The number of insertion points thus becomes irrelevant: it is equal to the number of inserted rows. This graph was obtained using SQL Server 7 on Windows 2000.

the allocation or deallocation takes place, but free lists can still become bottlenecks.

For example, in Oracle 8i, the number of free lists can be specified when creating a table (by default one free list is created). The rule of thumb is to specify a number of free lists equal to the maximum number of concurrent threads of control. Each thread running an insertion statement accesses one of these free lists. Oracle recommends to partition the master free list either when multiple user threads seek free blocks or when multiple instances run.[2]

Circumventing hot spots

A *hot spot* is a piece of data that is accessed by many transactions and is updated by some. Hot spots cause bottlenecks because each updating transaction must complete before any other transaction can obtain a lock on the hot data item. (You can usually identify hot spots easily for this reason: if transaction T takes far longer than normal at times when other transactions are executing at normal speeds, T is probably accessing a hot spot.) There are three techniques for circumventing hot spots.

2. Oracle 8i Parallel Server Concepts and Administration (8.1.5), Section 11.

1. Use partitioning to eliminate it, as discussed earlier.
2. Access the hot spot as late as possible in the transaction.

   Because transactions hold locks until they end, rewriting a transaction to obtain the lock on a hot data item as late as possible will minimize the time that the transaction holds the lock.
3. Use special database management facilities.

   Sometimes, the reason for a hot spot is surprising. For example, in some versions of Sybase Adaptive Server, performing a "select * into #temp. . ." locks the system catalog of the temporary database while the select takes place. This makes the system catalog a hot spot.

   Here is another example. In many applications, transactions that insert data associate a unique identifier with each new data item. When different insert transactions execute concurrently, they must somehow coordinate among themselves to avoid giving the same identifier to different data items.

   One way to do this is to associate a counter with the database. Each insert transaction increments the counter, performs its insert and whatever other processing it must perform, then commits. The problem is that the counter becomes a bottleneck because a transaction will (according to two-phase locking) release its lock on the counter only when the transaction commits.

   Some systems offer a facility (sequences in Oracle and identity in SQL Server, DB2 UDB, or Sybase Adaptive Server) that enables transactions to hold a latch[3] on the counter. This eliminates the counter as a bottleneck but may introduce a small problem.

   Consider an insert transaction I that increments the counter, then aborts. Before I aborts, a second transaction I' may increment the counter further. Thus, the counter value obtained by I will not be associated with any data item. That is, there may be gaps in the counter values. Most applications can tolerate such gaps but some cannot. For example, tax authorities do not like to see gaps in invoice numbers.

   Figure 2.14 compares the performance of transactions that rely on a system counter facility with the performance of transactions that increment unique identifiers.

---

3. A latch is a simple and efficient mechanism used to implement exclusive access to internal data structures. A latch is released immediately after access rather than being held until the end of a transaction like a lock. Also unlike a lock, a latch does not allow shared access (in read mode) and does not provide any support for queuing waiting threads.

APPENDIX B TRANSACTION CHOPPING

B.1 Assumptions

This appendix continues the discussion in Chapter 2 about making transactions smaller for the purpose of increasing available concurrency. It uses simple graph theoretical ideas to show how to cut up transactions in a safe way. If there are no control dependencies between the pieces that are cut up, then the pieces can be executed in parallel. Here are the assumptions that must hold for you to make use of this material.

- You can characterize all the transactions that will run in some interval. The characterization may be parametrized. For example, you may know that some transactions update account balances and branch balances, whereas others check account balances. However, you need not know exactly which accounts or branches will be updated.
- Your goal is to achieve the guarantees of serializability. (This appendix sets degree 3 isolation as its goal instead of full serializability because it makes the theory easier to understand. To apply chopping to systems desiring full serializability, the conflict graph should include semantic conflicts, for example, the conflict between a read of everyone whose last name begins with some letter between B and S and the insertion of a record about Bill Clinton.) You just don't want to pay for it.

  That is, you would like either to use degree 2 isolation (i.e., write locks are acquired in a two-phased manner, but read locks are released immediately after use), to use snapshot isolation, or to chop your transactions into smaller pieces.

The guarantee should be that the resulting execution be equivalent to one in which each original transaction executes in isolation.

- If a transaction makes one or more calls to rollback, you know when these occur.

  Suppose that you chop up the code for a transaction T into two pieces T_1 and T_2, where the T_1 part executes first. If the T_2 part executes a rollback statement in a given execution after T_1 commits, then the modifications done by T_1 will still be reflected in the database. This is not equivalent to an execution in which T executes a rollback statement and undoes all its modifications. Thus, you should rearrange the code so rollbacks occur early. We will formalize this intuition below with the notion of rollback safety.

- Suppose a transaction T modifies x, a program variable not in the database. If T aborts because of a concurrency control conflict and then executes properly to completion, variable x will be in a consistent state. (That is, we want the transaction code to be "reentrant.")
- If a failure occurs, it is possible to determine which transactions completed before the failure and which ones did not.

Suppose there are n transactions $T_1, T_2, \ldots, T_n$ that can execute within some interval. Let us assume, for now, that each such transaction results from a distinct program. Chopping a transaction will then consist of modifying the unique program that the transaction executes. Because of the form of the chopping algorithm, assuming this is possible will have no effect on the result.

A *chopping* partitions each T_i into *pieces* $c_{i_1}, c_{i_2}, \ldots, c_{i_k}$. Every database access performed by T_i is in exactly one piece.

A chopping of a transaction T is said to be *rollback safe* if either T has no rollback statements or all the rollback statements of T are in its first piece. The first piece must have the property that all its statements execute before any other statements of T. This will prevent a transaction from half-committing and then rolling back.

A chopping is said to be *rollback safe* if each of its transactions is rollback safe. Each piece will act like a transaction in the sense that each piece will acquire locks according to the two-phase locking algorithm and will release them when it ends. It will also commit its changes when it ends. Two cases are of particular interest.

- The transaction T is sequential and the pieces are nonoverlapping subsequences of that transaction.

  For example, suppose T updates an account balance and then updates a branch balance. Each update might become a separate piece, acting as a separate transaction.

- The transaction T operates at degree 2 isolation in which read locks are released as soon as reads complete.

  In this case, each read by itself constitutes a piece.[1] All writes together form a piece (because the locks for the writes are only released when T completes).

EXECUTION RULES

1. When pieces execute, they obey the order given by the transaction. For example, if the transaction updates account X first and branch balance B second, then the piece that updates account X should complete before the piece that updates branch balance B begins. By contrast, if the transaction performs the two steps in parallel, then the pieces can execute in parallel.
2. If a piece is aborted because of a lock conflict, then it will be resubmitted repeatedly until it commits.
3. If a piece is aborted because of a rollback, then no other pieces for that transaction will execute.

B.2 Correct Choppings

We will characterize the correctness of a chopping with the aid of an undirected graph having two kinds of edges.

1. *C edges.* C stands for *conflict.* Two pieces p and p' from different original transactions conflict if there is some data item x that both access and at least one modifies.[2] In this case, draw an edge between p and p' and label the edge C.
2. *S edges.* S stands for *sibling.* Two pieces p and p' are siblings if they come from the same transaction T. In this case, draw an edge between p and p' and label the edge S.

We call the resulting graph the *chopping graph.* (Note that no edge can have both an S and a C label.)

---

1. Technically, this needs some qualification. Each read that doesn't follow a write on the same data item constitutes a piece. The reason for the qualification is that if a write(x) precedes a read(x), then the transaction will continue to hold the lock on x after the write completes.
2. As has been observed repeatedly in the literature, this notion of conflict is too strong. For example, if the only data item in common between two transactions is one that is only incremented and whose exact value is insignificant, then such a conflict might be ignored. We assume the simpler read/write model only for the purposes of exposition.

We say that a chopping graph has an *SC-cycle* if it contains a simple cycle that includes at least one S edge and at least one C edge.[3]

We say that a chopping of $T_1, T_2, \ldots, T_n$ is *correct* if any execution of the chopping that obeys the execution rules is equivalent to some serial execution of the original transactions. "Equivalent" is in the sense of the textbook.[4] That is, every read (respectively, write) from every transaction returns (respectively, writes) the same value in the two executions and the same transactions roll back. Now, we can prove the following theorem.

Theorem 1: A chopping is correct if it is rollback safe and its chopping graph contains no SC-cycle.

Proof: The proof requires the properties of a serialization graph. Formally, a serialization graph is a directed graph whose nodes are transactions and whose directed edges represent ordered conflicts. That is, $T \rightarrow T'$ if T and T' both access some data item x, one of them modifies x, and T accessed x first. Following the Bernstein, Hadzilacos, and Goodman textbook, if the serialization graph resulting from an execution is acyclic, then the execution is equivalent to a serial one. The book also shows that if all transactions use two-phase locking, then all those who commit produce an acyclic serialization graph.

Call any execution of a chopping for which the chopping graph contains no SC-cycles a *chopped execution.* We must show that

1. any chopped execution yields an acyclic serialization graph on the given transactions $T_1, T_2, \ldots, T_n$ and hence is equivalent to a serial execution of those transactions.
2. the transactions that roll back in the chopped execution would also roll back if properly placed in the equivalent serial execution.

---

3. Recall that a simple cycle consists of (1) a sequence of nodes $n_1, n_2, \ldots, n_k$ such that no node is repeated and (2) a collection of associated edges: there is an edge between n_i and n_{i+1} for $1 \leq i < k$ and an edge between n_k and n_1; no edge is included twice.

4. See the following two references:

P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Reading, Mass., Addison-Wesley, 1987. *research.microsoft.com/pubs/ccontrol/.*

Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and Practice of Concurrency Control and Recovery.* Morgan Kaufmann, May 2001.

For point 1, we proceed by contradiction. Suppose there was a cycle in $T_1, T_2, \ldots, T_n$. That is $T \rightarrow T' \rightarrow \cdots \rightarrow T$. Consider three consecutive transactions T_i, T_{i+1}, and T_{i+2}. There must be pieces p_1, p_2, p_3, and p_4 such that p_1 comes from T_i, conflicts with p_2 from T_{i+1}, which is a sibling of or identical to p_3 from T_{i+1} and that conflicts with p_4 of T_{i+2}. Thus the transaction conflicts in the cycle correspond to a path P of directed conflict edges and possibly some sibling edges. Observe that there must be at least one sibling edge in P because otherwise there would be a pure conflict path of the form $p \rightarrow p' \rightarrow \cdots \rightarrow p$. Since the pieces are executed according to two-phase locking and since two-phase locking never allows cyclic serialization graphs, this cannot happen. (A piece may abort because of a concurrency control conflict, but then it will reexecute again and again until it commits.) Therefore, P corresponds to an SC-cycle, so there must be an SC-cycle in the chopping graph. By assumption, no such cycle exists.

For point 2, notice that any transaction T whose first piece p rolls back in the chopped execution will have no effect on the database, since the chopping is rollback safe. We want to show that T would also roll back if properly placed in the equivalent serial execution. Suppose that p conflicts with and follows pieces from the set of transactions $W_1, \ldots, W_k$. Then place T immediately after the last of those transactions in the equivalent serial execution. In that case, the first reads of T will be exactly those of the first reads of p. Because p rolls back, so will T. ■

Theorem 1 shows that the goal of any chopping of a set of transactions should be to obtain a rollback-safe chopping without an SC-cycle.

CHOPPING GRAPH EXAMPLE 1

Suppose there are three transactions that can abstractly be characterized as follows:

```
T1: R(x) W(x) R(y) W(y)
T2: R(x) W(x)
T3: R(y) W(y)
```

Breaking up T1 into

```
T11: R(x) W(x)
T12: R(y) W(y)
```

will result in a graph without an SC-cycle (Figure B.1). This verifies the rule of thumb from Chapter 2.

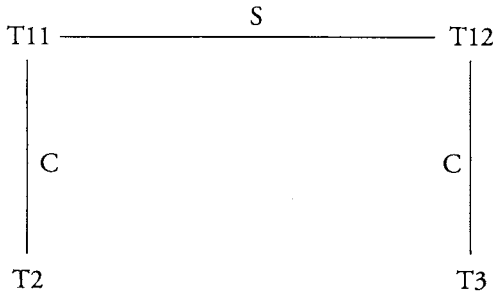

FIGURE B.1 No SC-cycle.

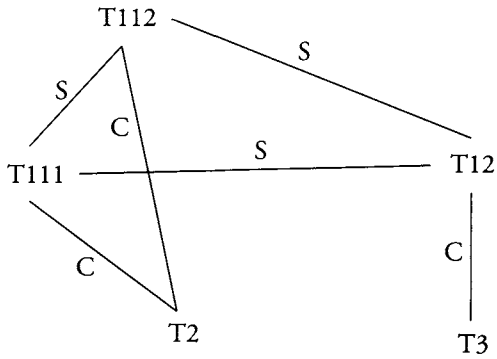

FIGURE B.2 SC-cycle.

CHOPPING GRAPH EXAMPLE 2

With the same T2 and T3 as in the first example, breaking up T11 further into

```
T111: R(x)
T112: W(x)
```

will result in an SC-cycle (Figure B.2).

CHOPPING GRAPH EXAMPLE 3

By contrast, if the three transactions were

```
T1: R(x) W(x) R(y) W(y)
T2: R(x)
T3: R(y) W(y)
```

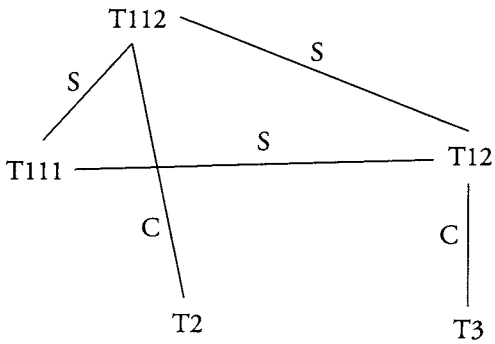

FIGURE B.3 No SC-cycle.

then T1 could be broken up into

```
T111: R(x)
T112: W(x)
T12: R(y) W(y)
```

There is no SC-cycle (Figure B.3). There is an S-cycle, but that doesn't matter.

CHOPPING GRAPH EXAMPLE 4

Now, let us take the example from Chapter 2 in which there are three types of transactions.

- A transaction that updates a single depositor's account and the depositor's corresponding branch balance.
- A transaction that reads a depositor's account balance.
- A transaction that compares the sum of the depositors' account balances with the sum of the branch balances.

For the sake of concreteness, suppose that depositor accounts D11, D12, and D13 all belong to branch B1; depositor accounts D21 and D22 both belong to B2. Here are the transactions.

```
T1 (update depositor): RW(D11) RW(B1)
T2 (update depositor): RW(D13) RW(B1)
T3 (update depositor): RW(D21) RW(B2)
T4 (read depositor): R(D12)
```

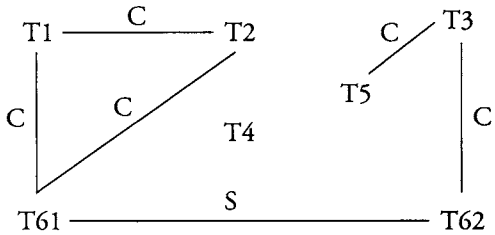

FIGURE B.4 No SC-cycle.

```
T5 (read depositor): R(D21)
T6 (compare balances): R(D11) R(D12) R(D13) R(B1) R(D21)
                       R(D22) R(B2)
```

Thus, T6 is the balance comparison transaction. Let us see first whether T6 can be broken up into two transactions.

```
T61: R(D11) R(D12) R(D13) R(B1)
T62: R(D21) R(D22) R(B2)
```

The absence of an SC-cycle shows that this is possible (Figure B.4).

CHOPPING GRAPH EXAMPLE 5

Taking the transaction population from the previous example, let us now consider dividing T1 into two transactions, giving the following transaction population (Figure B.5).

```
T11: RW(D11)
T12: RW(B1)
 T2: RW(D13) RW(B1)
 T3: RW(D21) RW(B2)
 T4: R(D12)
 T5: R(D21)
 T6: R(D11) R(D12) R(D13) R(B1) R(D21) R(D22) R(B2)
```

This results in an SC-cycle.

B.3 Finding the Finest Chopping

Now, you might wonder whether there is an algorithm to obtain a correct chopping. Two questions are especially worrisome.

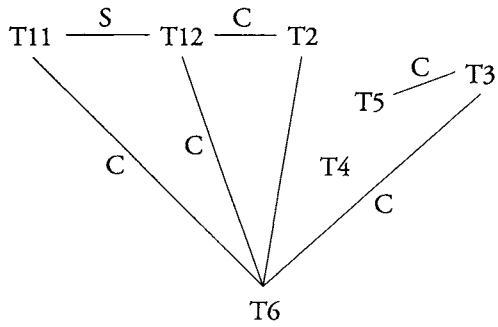

FIGURE B.5 SC-cycle.

1. Can chopping a piece into smaller pieces break an SC-cycle?
2. Can chopping one transaction prevent one from chopping another?

Remarkably, the answer to both questions is no.

Lemma 1: If a chopping is not correct, then any further chopping of any of the transactions will not render it correct.

Proof: Let the transaction to be chopped be called T and let the result of the chopping be called pieces(T). If T is not in an SC-cycle, then chopping T will have no effect on the cycle. If T is in an SC-cycle, then there are three cases.

1. If there are two C edges touching T from the cycle, then those edges will touch one or more pieces in pieces(T). (The reason is that the pieces partition the database accesses of T so the conflicts reflected by the C edges will still be present.) Those pieces (if there are more than one) will be connected by S edges after T is chopped, so the cycle will not be broken.
2. If there is one C edge touching T and one S edge leaving T (because T is already the result of a chopping), then the C edge will be connected to one piece p of pieces(T). If the S edge from T is connected to a transaction T', then p will also be connected by an S edge to T' because p and T' come from the same original transaction. So, the cycle will not be broken.
3. If there are two S edges touching T, then both of those S edges will be inherited by each piece in pieces(T), so again the cycle will not be broken. ■

Lemma 2: If two pieces of transaction T are in an SC-cycle as the result of some chopping, then they will be in a cycle even if no other transactions are chopped (Figure B.6).

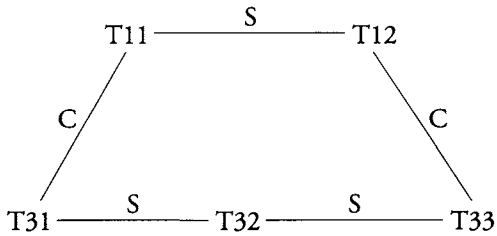

FIGURE B.6 Putting three pieces of T3 into one will not make chopping of T1 all right, nor will chopping T3 further.

Proof: Since two pieces, say, p and p', of T are in a cycle, there is an S edge between them and a C edge leading from each of them to pieces of other transactions. If only one piece of some other transaction T' is in the cycle, then combining all the pieces of T' will not affect the length of the cycle. If several pieces of T' are in the cycle, then combining them will simply shorten the cycle. ■

These two lemmas lead directly to a systematic method for chopping transactions as finely as possible. Consider again the set of transactions that can run in this interval $\{T_1, T_2, \ldots, T_n\}$. We will take each transaction T_i in turn. We call $\{c_1, c_2, \ldots, c_k\}$ a *private chopping* of T_i, denoted private(T_i), if both of the following hold:

1. $\{c_1, c_2, \ldots, c_k\}$ is a rollback-safe chopping of T_i.
2. There is no SC-cycle in the graph whose nodes are $\{T_1, \ldots, T_{i-1}, c_1, c_2, \ldots, c_k, T_{i+1}, \ldots, T_n\}$. (If $i = 1$, then the set is $\{c_1, c_2, \ldots, c_k, T_2, \ldots, T_n\}$. If $i = n$, then the set is $\{T_1, \ldots, T_{n-1}, c_1, c_2, \ldots, c_k\}$, i.e., the graph of all other transactions plus the chopping of T_i.)

Theorem 2: The chopping consisting of {*private* (T_1), *private* (T_2) . . . , *private* (T_n)} is rollback safe and has no SC-cycles.

Proof:

- *Rollback safe.* The "chopping" is rollback safe because all its constituents are rollback safe.
- *No SC-cycles.* If there were an SC-cycle that involved two pieces of private(T_i,) then lemma 2 would imply that the cycle would still be present even if all other transactions were not chopped. But that contradicts the definition of private (T_i). ■

B.4 Optimal Chopping Algorithm

Theorem 2 implies that if we can discover a fine-granularity private(T_i) for each T_i, then we can just take their union. Formally, the *finest chopping* of T_i, denoted FineChop (T_i) (whose existence we will prove) has two properties:

- FineChop(T_i) is a private chopping of T_i.
- If piece p is a member of FineChop(T_i), then there is no other private chopping of T_i containing p_1 and p_2 such that p_1 and p_2 partition p and neither is empty.

That is, we would have the following algorithm:

```
procedure chop(T_1,...,T_n)
 for each T_i
  Fine_i := finest chopping of T_i
 end for;
the finest chopping is
  {Fine_1, Fine_2,...,Fine_n}
```

We now give an algorithm to find the finest private chopping of T.

```
Algorithm FineChop(T)

initialization:

 if there are rollback statements then
  p_1 := all database writes
   of T that may occur
   before or concurrently with any rollback
   statement in T
 else
  p_1 := set consisting of
  the first database access;
end
P := {{x} | x is a database access not in p_1};
P := P union {p_1};

merging pieces:

 construct the connected components
 of the graph induced by C edges alone
 on all transactions besides T
 and on the pieces in P
```

```
update P based on the following rule:

if p_j and p_k are in the same connected
component and j < k, then
 add the accesses from p_k to p_j;
 delete p_k from P
end if
call the resulting partition FineChop(T)
```

Note on efficiency. The expensive part of the algorithm is finding the connected components of the graph induced by C on all transactions besides T and the pieces in P. We have assumed a naive implementation in which this is recomputed for each transaction T at a cost of $O(e + m)$ time in the worst case, where e is the number of C edges in the transaction graph and m is the size of P. Because there are n transactions, the total time is $O(n(e + m))$. Note that the only transactions relevant to FineChop(T) are those that are in the same connected component as T in the C graph. An improvement in the running time is possible if we avoid the total recomputation of the common parts of the connected components graph for the different transactions.

Note on shared code. Suppose that T_i and T_j result from the same program P. Since the chopping is implemented by changing P, T_i and T_j must be chopped in the same way. This may seem surprising at first, but in fact the preceding algorithm will give the result that FineChop(T_i) = FineChop(T_j). The reason is that the two transactions are treated symmetrically by the algorithm. When FineChop(T_i) runs, T_j is treated as unchopped and similarly for T_j. Thus, shared code does not change this result at all.

Theorem 3: FineChop(T) is the finest chopping of T.

Proof: We must prove two things: FineChop(T) is a private chopping of T and it is the finest one.

- FineChop(T) is a private chopping of T.
  1. *Rollback safety.* This holds by inspection of the algorithm. The initialization step creates a rollback-safe partition. The merging step can only cause p_1 to become larger.
  2. *No SC-cycles.* Any such cycle would involve a path through the conflict graph between two distinct pieces from FineChop(T). The merging step would have merged any two such pieces to a single one.

- No piece of FineChop(T) can be further chopped. Suppose p is a piece in FineChop(T). Suppose there is a private chopping *TooFine* of T that partitions p into two nonempty subsets q and r. Because p contains at least two accesses, the accesses of q and r could come from two different sources.

  1. Piece p is the first piece; that is, p_1, and q and r each contain accesses of p_1 as constructed in the initialization step. In that case, p_1 contains one or more rollback statements. So, one of q or r may commit before the other rolls back by construction of p_1. This would violate rollback safety.
  2. The accesses in q and r result from the merging step. In that case, there is a path consisting of C edges through the other transactions from q to r. This implies the existence of an SC-cycle for chopping *TooFine*. ■

B.5 Application to Typical Database Systems

For us, a typical database system will be one running SQL. Our main problem is to figure out what conflicts with what. Because of the existence of bind variables, it will be unclear whether a transaction that updates the account of customer :x will access the same record as a transaction that reads the account of customer :y. So, we will have to be conservative. Still, we can achi_ve substantial gains.

We can use the tricks of typical predicate locking schemes, however.[5] For example, if two statements on relation Account are both conjunctive (only AND's in the qualification) and one has the predicate

```
AND name LIKE 'T%'
```

whereas the other has the predicate

```
AND name LIKE 'S%'
```

they clearly will not conflict at the logical data item level. (This is the only level that matters because that is the only level that affects the return value to the user.) Detecting the absence of conflicts between two qualifications is the province of compiler writers. We offer nothing new.

---

5. K. C. Wong and M. Edelberg. "Interval Hierarchies and Their Application to Predicate Files." *ACM Transactions on Database Systems,* Vol. 2, No. 3, 223–232, September 1977.

The only new idea we have to offer is that we can make use of information in addition to simple conflict information. For example, if there is an update on the Account table with a conjunctive qualification and one of the predicates is

```
AND acctnum = :x
```

then, if acctnum is a key, we know that the update will access at most one record. This will mean that a concurrent reader of the form

```
SELECT ...
FROM Account
WHERE ...
```

will conflict with the update on at most one record, a single data item. We will therefore decorate this conflict edge with the label "1." If the update had not had an equality predicate on a key, then we would decorate this conflict edge with the label "many."

How does this decoration help? Well, suppose that the read and the update are the only two transactions in the system. Then if the label on the conflict is "1," the read can run at degree 2 isolation. This corresponds to breaking up the reader into n pieces, where n is the number of data items the reader accesses. A cycle in the SC-graph would imply that two pieces of the resulting chopping conflict with the update. This is impossible, however, since the reader and update conflict on only one piece altogether.

B.5.1 Chopping and Snapshot Isolation

Snapshot isolation (SI) is a concurrency control algorithm that never delays read operations, even for read/write transactions. SI has been implemented by Oracle (with certain variations), and it provides an isolation level that avoids many of the common concurrency anomalies. SI does not guarantee serializability, however. Like most protocols that fail to guarantee serializability, SI can lead to arbitrarily serious violations of integrity constraints. This results, we suspect, in thousands of errors per day, some of which may be quite damaging. Fortunately, there are many situations where snapshot isolation does give serializability, and chopping can help identify those situations.[6]

---

6. This subsection is drawn from joint work by Alan Fekete, Pat O'Neil, Betty O'Neil, Dimitrios Liarokapis, and Shasha.

In snapshot isolation, a transaction T1 reads data from the committed state of the database as of time start(T1) (the snapshot), and holds the results of its own writes in local memory store (so if it reads data a second time it will read its own output). Thus, writes performed by other transactions that were active after T1 started are not visible to T1. When T1 is ready to commit, it obeys the first-committer-wins rule: T1 will successfully commit if and only if no other concurrent transaction T2 has already committed writes to data that T1 intends to write. In the Oracle implementation of snapshot isolation, writes also take write locks, and a write is sometimes aborted immediately when a concurrent transaction has already written to that item, but the first-committer-wins certification check is still done at commit time.

Snapshot isolation does not guarantee serializability, however. For example, suppose X and Y are data items representing bank balances for a married couple, with the constraint that $X + Y > 0$ (the bank permits either account to overdraw as long as the sum of the account balances remains positive). Assume that initially $X_0 = 70$ and $Y_0 = 80$. Transaction T_1 reads X_0 and Y_0, then subtracts 100 from X, assuming it is safe because the two data items added up to 150. Transaction T_2 concurrently reads X_0 and Y_0, then subtracts 100 from Y_0, assuming it is safe for the same reason. Each update is safe by itself, but both are not, so no serial execution would allow them both to occur (since the second would be stopped by the constraint). Such an execution may take place on an Oracle DBMS ınder "set transaction isolation level serializable."

So snapshot isolation is not in general serializable, but observe that if we consider the read portion of each transaction to be a piece and the write portion to be another piece, then the execution on pieces is serializable. The reason is that the reads appear to execute when the transaction begins and the writes are serializable with respect to one another by the first-committer-wins rule. So, consider that decomposition to be the chopping. Snapshot serializability is serializable if the SC-graph on the pieces so constructed is acyclic. If a transaction program can run several times, it is sufficient to consider just two executions from the point of view of discovering whether a cycle is present.

B.6 Related Work

There is a rich body of work in the literature on the subject of chopping up transactions or reducing concurrency control constraints, some of which we review here. Such work nearly always proposes a new concurrency control algorithm and often proposes a weakening of isolation guarantees. (There is other work that proposes special-purpose concurrency control algorithms on data structures.)