Concurrency Control and Recovery[*]

Michael J. Franklin

Department of Computer Science and UMIACS

University of Maryland

College Park, MD

1 Introduction

Many service-oriented businesses and organizations, such as banks, airlines, catalog retailers, hospitals, etc. have grown to depend on fast, reliable, and correct access to their "mission-critical" data on a constant basis. In many cases, particularly for global enterprises, 7x24 access is required; that is, the data must be available seven days a week, twenty-four hours a day. Data Base Management Systems (DBMS) are often employed to meet these stringent performance, availability, and reliability demands. As a result, two of the core functions of a DBMS are: 1) to protect the data stored in the database and 2) to provide correct and highly available access to that data in the presence of concurrent access by large and diverse user populations, and despite various software and hardware failures. The responsibility for these functions resides in the **concurrency control** and **recovery** components of the DBMS software. Concurrency control ensures that individual users see consistent states of the database even though operations on behalf of many users may be interleaved by the database system. Recovery ensures that the database is fault tolerant; that is, that the database state is not corrupted as the result of a software, system, or media failure. The existence of this functionality in the DBMS allows applications to be written without explicit concern for concurrency and fault tolerance. This freedom provides a tremendous increase in pro-

[*] Portions of this chapter are reprinted with permission from: M. Franklin, M. Zwilling, C. Tan, M. Carey, and D. DeWitt, "Crash Recovery in Client-Server EXODUS", *Proc. ACM International Conference on Management of Data (SIGMOD'92)*, San Diego, June, 1992,(c) 1992 by the Association for Computing Machinery, Inc. (ACM).

grammer productivity and allows new applications to be added more easily and safely to an existing system.

For database systems, correctness in the presence of concurrent access and/or failures is tied to the notion of a **transaction**. A transaction is a unit of work, possibly consisting of multiple data accesses and updates, that must **commit** or **abort** as a single atomic unit. When a transaction commits, all updates it performed on the database are made permanent and visible to other transactions. In contrast, when a transaction aborts, all of its updates are removed from the database and the database is restored (if necessary) to the state it would have been in if the aborting transaction had never been executed. Informally, transaction executions are said to respect the **ACID properties**[Gray93]:

Atomicity: This is the "all-or-nothing" aspect of transactions discussed above — either all operations of a transaction complete successfully, or none of them do. Therefore, after a transaction has completed (i.e., committed or aborted), the database will not reflect a partial result of that transaction.

Consistency: Transactions preserve the consistency of the data — a transaction performed on a database that is internally consistent will leave the database in an internally consistent state. Consistency is typically expressed as a set of declarative integrity constraints. For example, a constraint may be that the salary of an employee cannot be higher than that of his or her manager.

Isolation: A transaction's behavior is not impacted by the presence of other transactions that may be accessing the same database concurrently. That is, a transaction sees only a state of the database that could occur if that transaction were the only one running against the database and produces only results that it could produce if it was running alone.

Durability: The effects of committed transactions survive failures. Once a transaction commits, its updates are guaranteed to be reflected in the database even if the contents of volatile (e.g., main memory) or non-volatile (e.g., disk) storage are lost or corrupted.

Of these four transaction properties, the concurrency control and recovery components of a

DBMS are primarily concerned with preserving Atomicity, Isolation, and Durability. The preservation of the Consistency property typically requires additional mechanisms such as compile-time analysis or run-time triggers in order to check adherence to integrity constraints.[1] For this reason, this chapter focuses primarily on the A,I, and D, of the ACID transaction properties.

Transactions are used to structure complex processing tasks which consist of multiple data accesses and updates. A traditional example of a transaction is a money transfer from one bank account (say account A) to another (say B). This transaction consists of a withdrawal from A and a deposit into B and requires four accesses to account information stored in the database: a read and write of A and a read and write of B. The data accesses of this transaction are as follows:

```
TRANSFER()
    01 A_bal := Read(A)
    02 A_bal := A_bal - $50
    03 Write(A,A_bal)
    04 B_bal := Read(B)
    05 B_bal := B_bal + $50
    06 Write(B,B_bal)
```

The value of A in the database is read and decremented by \$50, then the value of B in the database is read and incremented by \$50. Thus, TRANSFER preserves the invariant that the sum of the balances of A and B prior to its execution must equal the sum of the balances after its execution, regardless of whether the transaction commits or aborts. Consider the importance of the atomicity property. At several points in during the TRANSFER transaction, the database is in a temporarily inconsistent state. For example, between the time that account A is updated (statement 3) and the time that account B is updated (statement 6) the database reflects the decrement of A but not the increment of B, so it appears as if \$50 has disappeared from the database. If the transaction reaches such a point and then is unable to complete, (e.g., due to a failure or an unresolvable conflict, etc.) then the system must ensure that the effects of the partial results of the transaction (i.e., the update to A) are removed from the database — otherwise the database state will be incorrect. The durability property, in contrast, only comes into play in the event that the transaction successfully commits. Once the user is notified that the transfer has taken place he or she will assume that

[1] In the case of triggers, the recovery mechanism is typically invoked to abort an offending transaction.

account B contains the transferred funds and may attempt to use those funds from that point on. Therefore, the DBMS must ensure that the results of the transaction (i.e., the transfer of the $50) remain reflected in the database state even if the system crashes.

Atomicity, consistency, and durability address correctness for **serial execution** of transactions, where only a single transaction at a time is allowed to be in progress. In practice, however, database management systems typically support **concurrent execution**, in which the operations of multiple transactions can be executed in an interleaved fashion. The motivation for concurrent execution in a DBMS is similar to that for multiprogramming in operating systems, namely, to improve the utilization of system hardware resources and to provide multiple users a degree of fairness in access to those resources. The isolation property of transactions comes into play when concurrent execution is allowed.

Consider a second transaction that computes the sum of the balances of accounts A and B:

```
ReportSum()
    A_bal := Read(A)
    B_bal := Read(B)
    Print(A_bal + B_bal)
```

Assume that initially, the balance of account A is $300 and the balance of account B is $200. If a ReportSum transaction is executed on this state of the database, it will print a result of $500. In a database system restricted to serial execution of transactions, ReportSum will also produce the same result if it is executed after a Transfer transaction. The atomicity property of transactions ensures that if the Transfer aborts, all of its effects are removed from the database (so ReportSum would see A = $300 and B = $200), and the durability property ensures that if it commits then all of its effects remain in the database state (so ReportSum would see A = $250 and B = $250).

Under concurrent execution, however, a problem could arise if the isolation property is not enforced. As shown in Figure 1, if ReportSum were to execute after Transfer has updated account A but before it has updated account B, then ReportSum could see an inconsistent state of the database. In this case, the execution of ReportSum sees a state of the database in which

$50 has been withdrawn from account A but has not yet been deposited in account B, resulting in a total of $450 — it seems that $50 has disappeared from the database. This result is not one that could be obtained in any serial execution of Transfer and ReportSum transactions. It occurs because in this example, ReportSum accessed the database when it was in a temporarily inconsistent state. This problem is sometimes referred to as the inconsistent retrieval problem. To preserve the isolation property of transactions the DBMS must prevent the occurrence of this and other potential anomalies that could arise due to concurrent execution. The formal notion of correctness for concurrent execution in database systems is known as **serializability** and is described in the "Basic Principles" section of this chapter.

Transfer	ReportSum	
01 A_bal := **Read**(A)		
02 A_bal := A_bal - $50		
03 **Write**(A,A_bal)		
	01 A_bal := **Read**(A)	/* value is $250 */
	02 B_bal := **Read**(B)	/* value is $200 */
	03 Print(A_bal + B_bal)	/* result = $450 */
04 B_bal := **Read**(B)		
05 B_bal := B_bal + $50		
06 **Write**(B,B_bal)		

Figure 1: An Incorrect Interleaving of Transfer and ReportSum

Although the transaction processing literature often traces the history of transactions back to antiquity (such as Sumerian tax records) or to early contract law[Gray81, Gray93, Kort95], the roots of the transaction concept in information systems are typically traced back to the early 1970's, and the work of Bjork and Davies[Bjor73, Davi73]. Early systems such as IBM's IMS addressed related issues but a systematic treatment and understanding of ACID transactions was developed several years later by members of the IBM System R group [Gray75, Eswa76] and others (e.g., [Rose77, Lome77]). Since that time, many techniques for implementing ACID transactions have been proposed and a fairly well accepted set of techniques has emerged. The remainder of this chapter contains an overview of the basic theory that has been developed as well as a survey of the more widely-known implementation techniques for concurrency control and recovery. A brief discussion of work on extending the simple transaction model is presented at the end of the chapter.

It should be noted that issues related to those addressed by concurrency control and recovery in database systems arise in other areas of computing systems as well, such as file systems and memory systems. There are, however, two salient aspects of the ACID model that distinguish transactions from other approaches. First is the incorporation of both isolation (concurrency control) and fault tolerance (recovery) issues. Second is the concern with treating multiple write and/or read operations on multiple data items as an atomic, isolated unit of work. While these aspects of the ACID model provide powerful guarantees for the protection of data, they also can induce significant systems implementation complexity and performance overhead. For this reason, the notion of ACID transactions and their associated implementation techniques have remained largely within the DBMS domain where the provision of highly-available and reliable access to "mission critical" data is a primary concern.

2 Underlying Principles

2.1 Concurrency Control

2.1.1 Serializability

As stated in the previous section, the responsibility for maintaining the isolation property of ACID transactions resides in the concurrency control portion of the DBMS software. The most widely accepted notion of correctness for concurrent execution of transactions is serializability. Serializability is the property that a (possibly interleaved) execution of a group transactions has the same effect on the database, and produces the same output as some serial (i.e., non-interleaved) execution of those transactions. It is important to note that serializability does not specify any particular serial order, but rather, only that the execution is equivalent to some serial order. This distinction makes serializability a slightly less intuitive notion of correctness compared to transaction initiation time or commit order but it provides the DBMS with significant additional flexibility in the scheduling of operations. This flexibility can translate into increased responsiveness for end users.

A rich theory of database concurrency control has been developed over the years (see [Papa86, Bern87, Gray93]) and serializability lies at the heart of much of this theory. In this chapter we focus on the simplest models of concurrency control, where the operations that can be performed by transactions are restricted to: *read*(x), *write*(x), *commit*, and *abort*. The operation *read*(x) retrieves the value of a data item from the database, *write*(x) modifies the value of a data item in the database, and *commit* and *abort* indicate successful or unsuccessful transaction completion respectively (with the concomitant guarantees provided by the ACID properties). We also focus on a specific variant of serializability called conflict serializability. Conflict serializability is the most widely accepted notion of correctness for concurrent transactions because there are efficient, easily implementable techniques for detecting and/or enforcing it. Another well-known variant is called view serializability. View serializability is less restrictive (i.e., it allows more legal schedules) than conflict serializability, but it and other variants are primarily of theoretical interest because they are impractical to implement. The reader is referred to [Papa86] for a detailed treatment of alternative serializability models.

2.1.2 Transaction Schedules

Conflict serializability is based on the notion of a **schedule** of transaction operations. A schedule for a set of transaction executions is a partial ordering of the operations performed by those transactions, which shows how the operations are interleaved. The ordering defined by a schedule can be partial in the sense that it is only required to specify two types of dependencies:

- All operations of a given transaction for which an order is specified by that transaction must appear in that order in the schedule. For example, the definition of ReportSum above specifies that account A is read before account B.

- The ordering of all **conflicting operations** from different transactions must be specified. Two operations are said to conflict if they both operate on the same data item and at least one of them is a *write*().

The concept of a schedule provides a mechanism to express and reason about the (possibly) concurrent execution of transactions. A serial schedule is one in which all the operations of each transaction appear consecutively. For example, the serial execution of TRANSFER followed by REPORTSUM is represented by the following schedule:

$$r_0[A] \rightarrow w_0[A] \rightarrow r_0[B] \rightarrow w_0[B] \rightarrow c_0 \rightarrow r_1[A] \rightarrow r_1[B] \rightarrow c_1 \tag{1}$$

In this notation, each operation is represented by its initial letter, the subscript of the operation indicates the transaction number of the transaction on whose behalf the operation was performed, and a capital letter in brackets indicates a specific data item from the database (for read and write operations). A transaction number (tn) is a unique identifier that is assigned by the DBMS to an execution of a transaction. In the example above, the execution of TRANSFER was assigned tn 0 and the execution of REPORTSUM was assigned tn 1. A right arrow ($\rightarrow$) between two operations indicates that the lefthand operation is ordered before the righthand one. The ordering relationship is transitive; the orderings implied by transitivity are not explicitly drawn.

For example, the interleaved execution of TRANSFER and REPORTSUM shown in Figure 1 would produce the following schedule:

$$r_0[A] \rightarrow w_0[A] \rightarrow r_1[A] \rightarrow r_1[B] \rightarrow c_1 \rightarrow r_0[B] \rightarrow w_0[B] \rightarrow c_0 \tag{2}$$

The formal definition of serializability is based on the concept of equivalent schedules. Two schedules are said to be equivalent ($\equiv$) if:

- They contain the same transactions and operations; and
- They order all conflicting operations of non-aborting transactions in the same way.

Given this notion of equivalent schedules, a schedule is said to be serializable if and only if it is equivalent to some serial schedule. For example, the following concurrent schedule is serializable because it is equivalent to schedule (1):

$$r_0[A] \rightarrow w_0[A] \rightarrow r_1[A] \rightarrow r_0[B] \rightarrow w_0[B] \rightarrow c_0 \rightarrow r_1[B] \rightarrow c_1 \tag{3}$$

In contrast, the interleaved execution of schedule (2) is not serializable. To see why, notice that in any serial execution of Transfer and ReportSum either both writes of Transfer will precede both reads of ReportSum or vice versa. However, in schedule (2) $w_0[A] \rightarrow r_1[A]$ but $r_1[B] \rightarrow w_0[b]$. Schedule (2), therefore, is not equivalent to any possible serial schedule of the two transactions so it is not serializable. This result agrees with our intuitive notion of correctness, because recall that schedule (2) resulted in the apparent loss of $50.

2.1.3 Testing for Serializability

A schedule can easily be tested for serializability through the use of a precedence graph. A precedence graph is a directed graph that contains a vertex for each committed transaction execution in a schedule (non-committed executions can be ignored). The graph contains an edge from transaction execution T_i to transaction execution T_j $(i \neq j)$ if there is an operation in T_i that is constrained to precede an operation of T_j in the schedule. A schedule is serializable if and only if its precedence graph is acyclic. Figure 2(a) shows the precedence graph for schedule (2). That graph has an edge $T_0 \rightarrow T_1$ because the schedule contains $w_0[A] \rightarrow r_1[A]$ and an edge $T_1 \rightarrow T_0$ because the schedule contains $r_1[B] \rightarrow w_0[b]$. The cycle in the graph shows that the schedule is non-serializable. In contrast, Figure 2(b) shows the precedence graph for schedule (1). In this case, all ordering constraints are from T_0 to T_1 so the precedence graph is acyclic, indicating that the schedule is serializable.

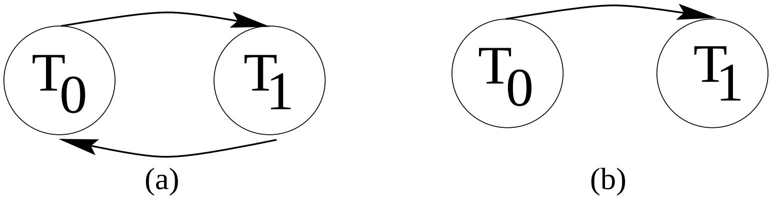

Figure 2: Precedence Graphs for (a) Non-Serializable and (b) Serializable Schedules

There are a number of practical ways to implement conflict serializability. These and other implementation issues are addressed in the "Best Practices" section of this chapter. Before discussing implementation issues, however, we first survey the the basic principles underlying database recovery.

2.2 Recovery

2.2.1 Coping With Failures

Recall that the responsibility for the atomicity and durability properties of ACID transactions lies in the recovery component of the DBMS. For recovery purposes it is necessary to distinguish between two types of storage: 1) **volatile storage**, such as main memory, whose state is lost in the event of a system crash or power outage, and 2) **non-volatile storage**, such as magnetic disks or tapes, whose contents persist across such events. The recovery subsystem is relied upon to ensure correct operation in the presence of three different types of failures (listed in order of likelihood):

- Transaction Failure - When a transaction that is in-progress reaches a state from which it cannot successfully commit, all updates that it made must be removed from the database in order to preserve the atomicity property. This is known as transaction rollback.

- System Failure - If the system fails in a way that causes the loss of volatile memory contents, recovery must ensure that: 1) the updates of all transactions that had committed prior to the crash are reflected in the database and 2) that all updates of other transactions (aborted or in-progress at the time of the crash) are removed from the database.

- Media Failure - In the event that data is lost or corrupted on the non-volatile storage (e.g., due to a disk-head crash) then the on-line version of data is lost. In this case, the database must be restored from an archival version of the database and brought up to date using operation logs.

In this chapter we focus on the issues of rollback and crash recovery, the most frequent uses of the DBMS recovery subsystem. Recovery from media crashes requires substantial additional mechanisms and complexity beyond what is covered here. Media recovery is addressed in the recovery-related references listed at the end of this chapter.

2.2.2 Buffer Management Issues

The process of removing the effects of an incomplete or aborted transaction for preserving atomicity is known as UNDO. The process of re-instating the effects of a committed transaction for durability is known as *REDO*. The amount of work that a recovery subsystem must perform for either of these functions depends on how the DBMS buffer manager handles data that is updated by in-progress and/or committing transactions [Haer83, Bern87]. Recall that the buffer manager is the DBMS component that is responsible for coordinating the transfer of data between main memory (i.e., volatile storage) and disk (i.e., non-volatile storage). The unit of storage that can be written atomically to non-volatile storage is called a *page*. Updates are made to copies of pages in the (volatile) buffer pool and those copies are written out to non-volatile storage at a later time. If the buffer manager allows an update made by an *uncommitted* transaction to overwrite the most recent committed value of a data item on non-volatile storage, it is said to support a *STEAL* policy (the opposite is called *NO-STEAL*). If the buffer manager ensures that all updates made by a transaction are reflected on non-volatile storage before the transaction is allowed to commit, then it is said to support a *FORCE* policy (the opposite is *NO-FORCE*).

Support for the STEAL policy implies that in the event that a transaction needs to be rolled-back (due to transaction failure or system crash), UNDOing the transaction will involve restoring the values of any non-volatile copies of data that were updated by that transaction back to their previous committed state. In contrast, a NO-STEAL policy guarantees that the data values on non-volatile storage are valid, so they do not need to be restored. A NO-FORCE policy raises the possibility that some committed data values may be lost during a system crash because there is no guarantee that they have been placed on non-volatile storage. This means that substantial REDO work may be required to preserve the durability of committed updates. In contrast, a FORCE policy ensures that the committed updates *are* placed on non-volatile storage, so that in the event of a system crash, the updates will still be reflected in the copy of the database on non-volatile storage.

From the above discussion, it should be apparent that a buffer manager that supports the combination of NO-STEAL and FORCE would place the fewest demands on UNDO and REDO recovery. However, these policies may negatively impact the performance of the DBMS during normal operation (i.e., when there are no crashes or rollbacks) because they restrict the flexibility of the buffer manager. NO-STEAL obligates the buffer manager to retain updated data in memory until a transaction commits or to write that data to a temporary location on non-volatile storage (e.g., a swap area). The problem with a FORCE policy is that it can impose significant disk write overhead during the critical path of a committing transaction. For these reasons, many buffer managers support the STEAL and NO-FORCE (**STEAL/NO-FORCE**) policies.

2.2.3 Logging

In order to deal with the UNDO and REDO requirements imposed by the STEAL and NO-FORCE policies respectively, database systems typically rely on the use of a **log**. A log is a sequential file that stores information about transactions and the state of the system at certain instances. Each entry in the log is called a **log record**. One or more log records are written for each update performed by a transaction. When a log record is created, it is assigned a **Log Sequence Number** (LSN) which serves to uniquely identify that record in the log. LSNs are typically assigned in a monotonically increasing fashion so that they provide an indication of relative position in the log. When an update is made to a data item in the buffer, a log record is created for that update. Many systems write the LSN of this new log record into the page containing the updated data item. Recording LSNs in this fashion allows the recovery system to relate the state of a data page to logged updates in order to tell if a given log record is reflected in a given state of a page.

Log records are also written for transaction management activities such as the commit or abort of a transaction. In addition, log records are sometimes written to describe the state of the system at certain periods of time. For example, such log records are written as part of the **checkpointing** process. Checkpoints are taken periodically during normal operation to help bound the amount of recovery work that would be required in the event of a crash. Part of the checkpointing process involves the writing of one or more checkpoint records. These records can include information about the contents of the buffer pool and the transactions that are currently active, etc. The particular contents of these records depends on the method of checkpointing that is used. Many different checkpointing methods have been developed, some of which involve quiescing the system to a consistent state, while others are less intrusive. A particularly non-intrusive type of checkpointing is used by the ARIES recovery method [Moha92b] that is described in the “Best Practices” section of this chapter.

For transaction update operations there are two basic types of logging: physical and logical [Gray93]. Physical log records typically indicate location (e.g., position on a particular page) of modified data in the database. If support for UNDO is provided (i.e., a STEAL policy is used) then the value of the item prior to the update is recorded in the log record. This is known as the before image of the item. Similarly the after image, (i.e., the new value of the item after the update), is logged if REDO support is provided. Thus, physical log records in a DBMS with STEAL/NO-FORCE buffer management contain both the old and new data values of items. Recovery using physical log records has the property that recovery actions (i.e., UNDOs or REDOs) are idempotent, meaning that they have the same effect no matter how many times they are applied. This property is important if recovery is invoked multiple times as will occur if a system fails repeatedly (e.g., due to a power problem or a faulty device).

Logical logging (sometimes referred to as operational logging) records only high-level information about operations that are performed, rather than recording the actual changes to items (or storage locations) in the database. For example, the insert of a new tuple into a relation might require many physical changes to the database such as space allocation, index updates, and reorganization, etc. Physical logging would require log records to be written for all of these changes. In contrast, logical logging would simply log the fact that the insertion had taken place, along with the value of the inserted tuple. The REDO process for a logical logging system must determine the set of actions that are required to fully reinstate the insert. Likewise, the UNDO logic must determine the set of actions that make up the inverse of the logged operation.

Logical logging has the advantage that it minimizes the amount of data that must be written to the log. Furthermore, it is inherently appealing because it allows many of the implementation details of complex operations to be hidden in the UNDO/REDO logic. In practice however, recovery based on logical logging is difficult to implement because the actions that make up the logged operation are not performed atomically. That is, when a system is restarted after a crash, the database may not be in an action consistent state with respect to a complex operation — it is possible that only a subset of the updates made by the action had been placed on non-volatile storage prior to the crash. As a result, it is difficult for the recovery system to determine which portions of a logical update are reflected in the database state upon recovery from a system crash. In contrast, physical logging does not suffer from this problem but it can require substantially higher logging activity.

In practice, systems often implement a compromise between physical and logical approaches that has been referred to as physiological logging [Gray93]. In this approach log records are constrained to refer to a single page, but may reflect logical operations on that page. For example, a physiological log record for an insert on a page would specify the value of the new tuple that is added to the page, but would not specify any free-space manipulation or reorganization of data on the page resulting from the insertion; the REDO and UNDO logic for insert would be required to infer the necessary operations. If a tuple insert required updates to multiple pages (e.g., data pages plus multiple index pages), then a separate physiological log record would be written for each page updated. Physiological logging avoids the action consistency problem of logical logging, while reducing, to some extent, the amount of logging that would be incurred by physical logging. The ARIES recovery method is one example of a recovery method that uses physiological logging.

2.2.4 Write Ahead Logging (WAL)

A final recovery principle to be addressed in this section is the **Write Ahead Logging** (WAL) protocol. Recall that the contents of volatile storage are lost in the event of a system crash. As a result, any log records that are not reflected on non-volatile storage will also be lost during a crash. WAL is a protocol that ensures that in the event of a system crash, the recovery log contains sufficient information to perform the necessary UNDO and REDO work when a STEAL/NO-FORCE buffer management policy is used. The WAL protocol ensures that:

1. All log records pertaining to an updated page are written to non-volatile storage before the page itself is allowed to be over-written in non-volatile storage.

2. A transaction is not considered to be committed until all of its log records (including its commit record) have been written to stable storage.

The first point ensures that UNDO information required due to the STEAL policy will be present in the log in the event of a crash. Similarly, the second point ensures that any REDO information required to due to the NO-FORCE policy will be present in the non-volatile log. The WAL protocol is typically enforced with special support provided by the DBMS buffer manager.

3 Best Practices

3.1 Concurrency Control

3.1.1 Two-phase locking

The most prevalent implementation technique for concurrency control is locking. Typically, two types of locks are supported, shared (S) locks and exclusive (X) locks. The compatibility of these locks is defined by the compatibility matrix shown in Table 1. The compatibility matrix shows that two different transactions are allowed to hold S locks simultaneously on the same data item, but that X locks cannot be held on an item simultaneously with any other locks (by other transactions) on that item. S locks are used for protecting read access to data (i.e., multiple concurrent readers are allowed) and X locks are used for protecting write access to data. As long as a transaction is holding a lock, no other transaction is allowed to obtain a conflicting lock. If a transaction requests a lock that cannot be granted (due to a lock conflict), that transaction is blocked (i.e., prohibited from proceeding) until all the conflicting locks held by other transactions are released.

	S	X
S	y	n
X	n	n

Table 1: Compatibility Matrix for S and X Locks

S and X locks as defined in Table 1 directly model the semantics of conflicts used in the definition of conflict serializability. Therefore, locking can be used to enforce serializability. Rather than testing for serializability after a schedule has been produced (as was done in the previous section), the blocking of transactions due to lock conflicts can be used to prevent non-serializable schedules from ever being produced.

A transaction is said to be **well-formed** with respect to reads if it always holds an S or an X lock on an item while reading it, and well-formed with respect to writes if it always holds an X lock on an item while writing it. Unfortunately, restricting all transactions to be well-formed is not sufficient to guarantee serializability. For example, a non-serializable execution such as that of schedule (2) is still possible using well-formed transactions. Serializability can be enforced, however, through the use of **two-phase locking** (2PL). Two phase locking requires that all transactions be well-formed and that they respect the following rule:

Once a transaction has released a lock, it is not allowed to obtain any additional locks.

This rule results in transactions that have two phases:

1. A growing phase in which the transaction is acquiring locks; and

2. A shrinking phase in which locks are released.

The two-phase rule dictates that the transaction shifts from the growing phase to the shrinking phase at the instant it first releases a lock. To see how 2PL enforces serializability, consider again schedule (2). Recall that the problem arises in this schedule because $w_0[A] \rightarrow r_1[A]$ but $r_1[B] \rightarrow w_0[b]$. This schedule could not be produced under 2PL, because transaction 1 (Report-Sum) would be blocked when it attempted to read the value of A because transaction 0 would be holding an X lock on it. Transaction 0 would not be allowed to release this X lock before obtaining its X lock on B, and thus it would either abort or perform its update of B before transaction 1 is allowed to progress. In contrast, note that schedule (1) (the serial schedule) would be allowed in 2PL. 2PL would also allow the following (serializable) interleaved schedule:

$$r_1[A] \rightarrow r_0[A] \rightarrow r_1[B] \rightarrow c_1 \rightarrow w_0[A] \rightarrow r_0[B] \rightarrow w_0[B] \rightarrow c_0 \tag{4}$$

It is important to note, however, that two-phase locking is sufficient but not necessary for implementing serializability. In other words, there are schedules that are serializable but would not be allowed by two-phase locking. Schedule (3) is an example of such a schedule.

In order to implement 2PL, the DBMS contains a component called a lock manager. The lock manager is responsible for granting or blocking lock requests, for managing queues of blocked transactions, and for un-blocking transactions when locks are released. In addition, the lock manager is also responsible for dealing with **deadlock** situations. A deadlock arises when a set of transactions is blocked, each waiting for another member of the set to release a lock. In a deadlock situation, none of the transactions involved can make progress. Database systems deal with deadlocks using one of two general techniques: avoidance or detection. Deadlock avoidance can be achieved by imposing an order in which locks can be obtained on data items, by requiring transactions to pre-declare their locking needs, or by aborting transactions rather than blocking them in certain situations.

Deadlock detection, on the other hand, can be implemented using timeouts or explicit checking. Timeouts are the simplest technique; if a transaction is blocked beyond a certain amount of time, it is assumed that a deadlock has occurred. The choice of a timeout interval can be problematic, however. If it is too short, then the system may infer the presence of a deadlock that does not truly exist. If it is too long, then deadlocks may go undetected for too long a time. Alternatively the system can explicitly check for deadlocks using a structure called a waits-for graph. A waits-for graph is a directed graph with a vertex for each active transaction. The lock manager constructs the graph by placing an edge from a transaction T_i to a transaction T_j $(i \neq j)$ if T_i is blocked waiting for a lock held by T_j. If the waits-for graph contains a cycle, all of the transactions involved in the cycle are waiting for each other, and thus, they are deadlocked. When a deadlock is detected, one or more of the transactions involved is rolled-back. When a transaction is rolled-back its locks are automatically released, so the deadlock will be broken.

3.1.2 Isolation Levels

As should be apparent from the previous discussion, transaction isolation comes at a cost in potential concurrency. Transaction blocking can add significantly to transaction response time.[2] As stated previously, serializability is typically implemented using two-phase locking, which requires locks to be held at least until all necessary locks have been obtained. Prolonging the holding-time of locks increases the likelihood of blocking due to data contention.

In some applications, however, serializability is not strictly necessary. For example, a data analysis program that computes aggregates over large numbers of tuples may be able to tolerate

[2] Note that other non-blocking approaches discussed later in this section also suffer from similar problems.

some inconsistent access to the database in exchange for improved performance. The concept of degrees of isolation or isolation levels has been developed to allow transactions to trade concurrency for consistency in a controlled manner [Gray75, Gray93, Bere95]. In their 1975 paper, Gray et al. defined four degrees of consistency using characterizations based on locking, dependencies, and anomalies (i.e., results that could not arise in a serial schedule). The degrees were named degree 0-3, with degree 0 being the least consistent, and degree 3 intended to be equivalent to serializable execution.

The original presentation has served as the basis for understanding relaxed consistency in many current systems but it has become apparent over time that the different characterizations in that paper were not specified to an equal degree of detail. As pointed out in a recent paper by Berenson et al. [Bere95], the SQL-92 standard suffers from a similar lack of specificity. Berenson et al. have attempted to clarify the issue, but it is too early to determine if they have been successful. In this section we focus on the locking-based definitions of the isolation levels, as they are generally acknowledged to have "stood the test of time" [Bere95]. However, the definition of the degrees of consistency requires an extension to the previous description of locking in order to address the phantom problem.

An example of the phantom problem is the following: assume a transaction T_i reads a set of tuples that satisfy a query predicate. A second transaction T_j inserts a new tuple that satisfies the predicate. If T_i then executes the query again, it will see the new item, so that its second answer differs from the first. This behavior could never occur in a serial schedule as a "phantom" tuple appears in the midst of a transaction, thus, this execution is anomalous. The phantom problem is an artifact of the transaction model consisting of reads and writes to individual data items that we have used so far. In practice, transactions include queries that dynamically define sets of items based on predicates. When a query is executed, all of the tuples that satisfy the predicate at that time can be locked as they are accessed. Such individual locks, however, do not protect against the later addition of further tuples that satisfy the predicate.

One obvious solution to the phantom problem is to lock predicates instead of (or in addition to) individual items [Eswa76]. This solution is impractical to implement, however, due to the complexity of detecting the overlap of a set of arbitrary predicates. Predicate locking can be approximated using techniques based on locking clusters of data or ranges of index values. Such techniques, however, are beyond the scope of this chapter. In this discussion we will assume that predicates can be locked without specifying the technical details of how this can be accomplished (see [Gray93, Moha92a] for detailed treatments of this topic).

The locking-oriented definitions of the isolation levels are based on whether or not read and/or write operations are well-formed (i.e., protected by the appropriate lock), and if so, whether those locks are long duration or short duration. Long duration locks are held until the end of a transaction (EOT) (i.e., when it commits or aborts); short duration locks can be released earlier. Long duration write locks on data items have important benefits for recovery, namely, they allow recovery to be performed using before images. If long duration write locks are not used, then the following scenario could arise:

$$w_0[A] \rightarrow w_1[A] \rightarrow a_0 \tag{5}$$

In this case restoring A with T_0's before image of it will be incorrect because it would overwrite T_1's update. Simply ignoring the abort of T_0 is also incorrect. In that case, if T_1 were to subsequently abort, installing its before image would reinstate the value written by T_0. For this reason and for simplicity, locking systems typically hold long duration locks on data items. This is sometimes referred to as strict locking [Bern87].

Given these notions of locks, the degrees of isolation presented in the SQL-92 standard can be obtained using different lock protocols. In the following, all levels are assumed to be well-formed with respect to writes and to hold long duration write (i.e., exclusive) locks on updated data items. Four levels are defined (from weakest to strongest) [3]:

READ UNCOMMITTED - This level, which provides the weakest consistency guarantees, allows transactions to read data that has been written by other transactions that have not committed. In a locking implementation this level is achieved by being ill-formed with respect to reads

[3] It should be noted that two-phase locks can be substituted for the long-duration locks in these definitions without impacting the consistency provided. Long-duration locks are typically used, however, to avoid the recovery-related problems described previously.

(i.e., not obtaining read locks). The risks of operating at this level include (in addition to the risks incurred at the more restrictive levels) the possibility of seeing updates that will eventually be rolled-back and the possibility of seeing some of the updates made by another transaction but missing others made by that transaction.

READ COMMITTED - This level ensures that transactions only see updates that have been made by transactions that have committed. This level is achieved by being well-formed with respect to reads on individual data items, but holding the read locks only as short duration locks. Transactions operating at this level run the risk of seeing non-repeatable reads (in addition to the risks of the more restrictive levels). That is, a transaction T_0 could read a data item twice and see two different values. This anomaly could occur if a second transaction were to update the item and commit in between the two reads by T_0.

REPEATABLE READ - This level ensures that reads to individual data items are repeatable, but does not protect against the phantom problem described previously. This level is achieved by being well-formed with respect to reads on individual data items, and holding those locks for long duration.

SERIALIZABLE - This level protects against all of the problems of the less restrictive levels, including the phantom problem. It is achieved by being well-formed with respect to reads on predicates as well as on individual data items and holding all locks for long duration.

A key aspect of this definition of degrees of isolation is that as long as all transactions execute at the READ UNCOMMITTED level or higher, they are able to obtain at least the degree of isolation they desire without interference from any transactions running at lower degrees. Thus, these degrees of isolation provide a powerful tool that allows application writers or users to trade off consistency for improved concurrency. As stated earlier, the definition of these isolation levels for concurrency control methods that are not based on locking has been problematic. This issue is addressed in depth in [Bere95].

It should be noted, that the discussion of locking so far has ignored an important class of data that is typically present in databases, namely, indexes. Because indexes are auxiliary information, they can be accessed in a non-two-phase manner without sacrificing serializability. Furthermore, the hierarchical structure of many indexes (e.g., B-trees) makes them potential concurrency bottlenecks due to high contention at the upper levels of the structure. For this reason, significant effort has gone into developing methods for providing highly-concurrent access to indexes. Pointers to some of this work can be found in the "For Further Information" section at the end of this chapter.

3.1.3 Hierarchical Locking

The examples in the preceeding discussions of concurrency control primarily dealt with operations on a single granularity of data items (e.g., tuples). In practice, however, the notions of conflicts and locks can be applied at many different granularities. For example, it is possible to perform locking at the granularity of a page, relation, or even an entire database. In choosing the proper granularity at which to perform locking there is a fundamental tradeoff between potential concurrency and locking overhead. Locking at a fine granularity, such as an individual tuple, allows for maximum concurrency as only transactions that are truly accessing the same tuple have the potential to conflict. The downside of such fine-grained locking, however, is that a transaction that accesses a large number of tuples will have to acquire a large number of locks. Each lock request requires a call to the lock manager. This overhead can be reduced by locking at a coarser granularity but coarse granularity raises the potential for false conflicts. For example, two transactions that update different tuples residing on the same page would conflict under page-level locking but not under tuple-level locking.

The notion of hierarchical or multi-granular locking was introduced to allow concurrent transactions to obtain locks at different granularities in order to optimize the above trade-off [Gray75]. In hierarchical locking, a lock on a granule at a particular level of the granularity hierarchy implicitly locks all items included in that granule. For example, an S lock on a relation implicitly locks all pages and tuples in that relation. Thus a transaction with such a lock can read any tuple in the relation without requesting additional locks. Hierarchical locking introduces additional lock modes beyond S and X. These additional modes allow transactions to declare their intention to perform an operation on objects at lower levels of the granularity hierarchy. The new modes are IS, IX, and

SIX for Intention Shared, Intention Exclusive and Shared with Intention Exclusive. An IS (or IX) lock on a granule provides no privileges on that granule, but indicates that the holder intends to obtain S (or X) locks on one or more finer granules. An SIX lock combines an S lock on the entire granule with an IX lock. SIX locks support the common access pattern of scanning the items in a granule (e.g., tuples in a relation) and choosing to update a fraction of them based on their values.

Similarly to S and X locks, these lock modes can be described using a compatibility matrix. The compatibility matrix for these modes is shown in Table 2. In order for transactions locking at different granularities to coexist, all transactions must follow the same hierarchical locking protocol starting from the root of the granularity hierarchy. This protocol is shown in Table 3. For example, to read a single record, a transaction would obtain IS locks on the database, relation, and page, followed by an S lock on the specific tuple. If a transaction wanted to read all or most tuples on a page, then it could obtain IS locks on the database and relation, followed by an S lock on the entire page. By following this uniform protocol, potential conflicts between transactions that ultimately obtain S and/or X locks at different granularities can be detected. A useful extension to hierarchical locking is known as lock escalation. Lock escalation allows the DBMS to automatically adjust the granularity at which transactions obtain locks based on their behavior. If the system detects that a transaction is obtaining locks on a large percentage of the granules that make up a larger granule, it can attempt to grant the transaction a lock on the larger granule so that no additional locks will be required for subsequent accesses to other objects in that granule. Automatic

	IS	IX	S	SIX	X
IS	y	y	y	y	n
IX	y	y	n	n	n
S	y	n	y	n	n
SIX	y	n	n	n	n
X	n	n	n	n	n

Table 2: Compatibility Matrix for Regular and Intention Locks

To Get	Must Have on all Ancestors
IS or S	IS or IX
IX,SIX, or X	IX or SIX

Table 3: Hierarchical Locking Rules

escalation is useful because the access pattern that a transaction will produce is often not known until runtime.

3.1.4 Other Concurrency Control Methods

As stated previously, two-phase locking is the most generally accepted technique for ensuring serializability. Locking is considered to be a pessimistic technique because it is based on the assumption that transactions are likely to interfere with each other and takes measures (e.g., blocking) to ensure that such interference does not occur. An important alternative to locking is **optimistic concurrency control**. Optimistic methods (e.g., [Kung81]) allow transactions to perform their operations without obtaining any locks. To ensure that concurrent executions do not violate serializability, transactions must perform a validation phase before they are allowed to commit. Many optimistic protocols have been proposed. In the algorithm of [Kung81], the validation process ensures that the reads and writes performed by a validating transaction did not conflict with any other transactions with which it ran concurrently. If during validation it is determined a conflict had occurred, the validating transaction is aborted and restarted.

Unlike locking, which depends on blocking transactions to ensure isolation, optimistic policies depend on transaction restart. As a result, although they don't perform any blocking, the performance of optimistic policies can be negatively impacted by data contention (as are pessimistic schemes) — a high degree of data contention will result in a large number of unsuccessful transaction executions. The performance tradeoffs between optimistic and pessimistic have been addressed in numerous studies (see [Agra87]). In general, locking is likely to be superior in resource-limited environments because blocking does not consume cpu or disk resources. In contrast, optimistic techniques may have performance advantages in situations where resources are abundant, because they allow more executions to proceed concurrently. If resources are abundant, then the resource consumption of restarted transactions will not significantly hurt performance. In practice, however resources are typically limited and thus, concurrency control in virtually all commercial database systems is based on locking.

Another class of concurrency control techniques is known as **multiversion concurrency con-**

trol (e.g., [Reed83]). As updating transactions modify data items, these techniques retain the previous versions of the items on-line. Read-only transactions (i.e., transactions that perform no updates) can then be provided with access to these older versions, allowing them to see a consistent (although possibly somewhat out-of-date) snapshot of the database. Optimistic, multiversion, and other concurrency control techniques (e.g., timestamping) are addressed in further detail in [Bern87].

3.2 Recovery

The recovery subsystem is generally considered to be one of the more difficult parts of a DBMS to design for two reasons: First, recovery is required to function in failure situations and must correctly cope with a huge number of possible system and database states. Second the recovery system depends on the behavior of many other components of the DBMS, such as concurrency control, buffer management, disk management, and query processing. As a result, few recovery methods have been described in the literature in detail. One exception is the ARIES recovery system developed at IBM [Moha92b]. Many details about the ARIES method have been published, and the method has been included in a number of DBMSs. Furthermore, the ARIES method involves only a small number of basic concepts. For these reasons, we focus on the ARIES method in the remainder of this section. The ARIES method is related to many other recovery methods such as those described in [Bern87, Gray93]. A comparison with other techniques appears in [Moha92b].

3.2.1 Overview of ARIES

ARIES is a fairly recent refinement of the Write-Ahead-Logging (WAL) protocol. Recall that the WAL protocol enables the use of a STEAL/NO FORCE buffer management policy, which means that pages on stable storage can be overwritten at any time and that data pages do not need to be forced to disk in order to commit a transaction. As with other WAL implementations, each page in the database contains a Log Sequence Number (LSN) which uniquely identifies the log record for the latest update that was applied to the page. This LSN (referred to as the pageLSN) is used during recovery to determine whether or not an update for a page must be redone. LSN information is also used to determine the point in the log from which the Redo pass must commence during restart from a system crash. LSNs are often implemented using the physical address of the log record in the log to enable the efficient location of a log record given its LSN.

Much of the power and relative simplicity of the ARIES algorithm is due to its REDO paradigm of repeating history, in which it redoes updates for all transactions — including those that will eventually be undone. Repeating history enables ARIES to employ a variant of the physiological logging technique described earlier: it uses page-oriented REDO and a form of logical UNDO. Page-oriented REDO means that REDO operations involve only a single page and that the affected page is specified in the log record. This is part of physiological logging. In the context of ARIES, logical UNDO means that the operations performed to undo an update do not need to be the exact inverses of the operations of the original update.

In ARIES, logical UNDO is used to support fine-grained (i.e., tuple-level) locking and high-concurrency index management. For an example of the latter issue, consider a case in which a transaction T1 updates an index entry on a given page P1. Before T1 completes, a second transaction T2 could split P1, causing the index entry to be moved to a new page (P2). If T1 must be undone, a physical, page-oriented approach would fail because it would erroneously attempt to perform the UNDO operation on P1. Logical UNDO solves this problem by using the index structure to find the index entry, and then applying the UNDO operation to it in its new location. In contrast to UNDO, page-oriented REDO can be used because the repeating history paradigm ensures that REDO operations will always find the index entry on the page referenced in the log record — any operations that had affected the location of the index operation at the time the log record was created will be replayed before that log record is redone.

ARIES uses a three pass algorithm for restart recovery. The first pass is the Analysis pass, which processes the log forward from the most recent checkpoint. This pass determines information about dirty pages and active transactions that is used in the subsequent passes. The second pass is the REDO pass, in which history is repeated by processing the log forward from the earliest log record that could require redo, thus insuring that all logged operations have been applied. The third pass is the UNDO pass. This pass proceeds backwards from the end of the log, removing from the database the effects of all transactions that had not committed at the time of the crash. These passes are shown in Figure 3. (Note that the relative ordering of the starting point for the REDO pass, the endpoint for the UNDO pass, and the checkpoint can be different than that shown in the figure.) The three passes are described in more detail below.

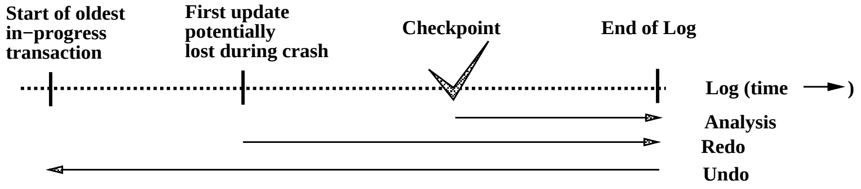
Start of oldest in−progress transaction
First update potentially lost during crash
Checkpoint
End of Log
Log (time ➝)
Analysis
Redo
Undo

Figure 3: The Three Passes of ARIES Restart

ARIES maintains two important data structures during normal operation. The first is the Transaction Table, which contains status information for each transaction that is currently running. This information includes a field called the lastLSN, which is the LSN of the most recent log record written by the transaction. The second data structure, called the Dirty Page Table, contains an entry for each "dirty" page. A page is considered to be dirty if it contains updates that are not reflected on stable storage. Each entry in the Dirty Page Table includes a field called the recoveryLSN, which is the LSN of the log record that caused the associated page to become dirty. Therefore, the recoveryLSN is the LSN of the earliest log record that might need to be redone for the page during restart. Log records belonging to the same transaction are linked backwards in time using a field in each log record called the prevLSN field. When a new log record is written for a transaction, the value of the lastLSN field in the Transaction Table entry is placed in the prevLSN field of the new record and the new record's LSN is entered as the lastLSN in the Transaction Table entry.

During normal operation, checkpoints are taken periodically. ARIES uses a form of fuzzy checkpoints which are extremely inexpensive. When a checkpoint is taken, a checkpoint record is constructed which includes the contents of the Transaction Table and the Dirty Page Table. Checkpoints are efficient since no operations need be quiesced and no database pages are flushed to perform a checkpoint. However, the effectiveness of checkpoints in reducing the amount of the log that must be maintained is limited in part by the earliest recoveryLSN of the dirty pages at checkpoint time. Therefore, it is helpful to have a background process that periodically writes dirty pages to non-volatile storage.

3.2.2 Analysis

The job of the Analysis pass of restart recovery is threefold: 1) It determines the point in the log at which to start the REDO pass, 2) It determines which pages could have been dirty at the time of the crash in order to avoid unnecessary I/O during the REDO pass, and 3) It determines which transactions had not committed at the time of the crash and will therefore need to be undone.

The Analysis pass begins at the most recent checkpoint and scans forward to the end of the log. It reconstructs the Transaction Table and Dirty Page Table to determine the state of the system as of the time of the crash. It begins with the copies of those structures that were logged in the checkpoint record. Then, the contents of the tables are modified according to the log records that are encountered during the forward scan. When a log record for a transaction that does not appear in the Transaction Table is encountered, that transaction is added to the table. When a log record for the commit or the abort of a transaction is encountered, the corresponding transaction is removed from the Transaction Table. When a log record for an update to a page that is not in the Dirty Page Table is encountered, that page is added to the Dirty Page Table, and the LSN of the record which caused the page to be entered into the table is recorded as the recoveryLSN for that page. At the end of the Analysis pass, the Dirty Page Table is a conservative (since some pages may have been flushed to non-volatile storage) list of all database pages that could have been dirty at the time of the crash, and the Transaction Table contains entries for those transactions that will actually require undo processing during the UNDO phase. The earliest recoveryLSN of all the entries in the Dirty Page Table, called the firstLSN, is used as the spot in the log from which to begin the REDO phase.

3.2.3 REDO

As stated earlier, ARIES employs a redo paradigm called repeating history. That is, it redoes updates for all transactions, committed or otherwise. The effect of repeating history is that at the end of the REDO pass, the database is in the same state with respect to the logged updates that it was in at the time that the crash occurred. The REDO pass begins at the log record whose LSN is the firstLSN determined by Analysis and scans forward from there. To redo an update, the logged action is re-applied and the pageLSN on the page is set to the LSN of the redone log record. No logging is performed as the result of a redo. For each log record the following algorithm is used to determine if the logged update must be redone:

- If the affected page is not in the Dirty Page Table then the update does NOT require redo.
- If the affected page is in the Dirty Page Table, then if the recoveryLSN in the page's table entry is greater than the LSN of the record being checked, the update does NOT require redo.
- Otherwise, the LSN stored on the page (the pageLSN) must be checked. This may require that the page be read in from disk. If the pageLSN is greater than or equal to the LSN of the record being checked, then the update does NOT require redo. Otherwise, the update MUST be redone.

3.2.4 UNDO

The UNDO pass scans backwards from the end of the log. During the UNDO pass, all transactions that had not committed by the time of the crash must be undone. In ARIES, undo is an unconditional operation. That is, the pageLSN of an affected page is not checked because it is always the case that the undo must be performed. This is due to the fact that the repeating of history in the REDO pass insures that all logged updates have been applied to the page.

When an update is undone, the undo operation is applied to the page and is logged using a special type of log record called a Compensation Log Record (CLR). In addition to the undo information, a CLR contains a field called the UndoNxtLSN. The UndoNxtLSN is the LSN of the next log record that must be undone for the transaction. It is set to the value of the prevLSN field of the log record being undone. The logging of CLRs in this fashion enables ARIES to avoid ever having to undo the effects of an undo (e.g., as the result of a system crash during an abort) thereby limiting the amount of work that must be undone and bounding the amount of logging done in the event of multiple crashes. When a CLR is encountered during the backwards scan, no operation is performed on the page, and the backwards scan continues at the log record referenced by the UndoNxtLSN field of the CLR, thereby jumping over the undone update and all other updates for the transaction that have already been undone (the case of multiple transactions will be discussed shortly). An example execution is shown in Figure 4.

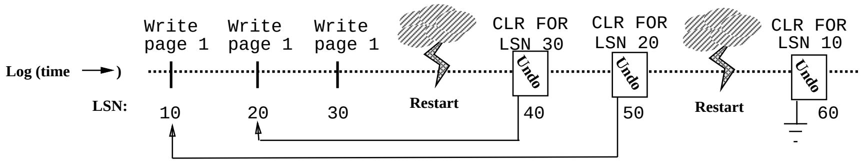

Figure 4: The Use of CLRs for UNDO

In Figure 4, a transaction logged three updates (LSNs 10, 20, and 30) before the system crashed for the first time. During REDO, the database was brought up to date with respect to the log (i.e., 10, 20, and/or 30 were redone if they weren't on non-volatile storage), but since the transaction was in progress at the time of the crash, they must be undone. During the UNDO pass, update 30 was undone, resulting in the writing of a CLR with LSN 40, which contains an UndoNxtLSN value that points to 20. Then, 20 was undone, resulting in the writing of a CLR (LSN 50) with an UndoNxtLSN value that points to 10. However, the system then crashed for a second time before 10 was undone. Once again, history is repeated during REDO which brings the database back to the state it was in after the application of LSN 50 (the CLR for 20). When UNDO begins during this second restart, it will first examine the log record 50. Since the record is a CLR, no modification will be performed on the page, and UNDO will skip to the record whose LSN is stored in the UndoNxtLSN field of the CLR (i.e., LSN 10). Therefore, it will continue by undoing the update whose log record has LSN 10. This is where the UNDO pass was interrupted at the time of the second crash. Note that no extra logging was performed as a result of the second crash.

In order to undo multiple transactions, restart UNDO keeps a list containing the next LSN to be undone for each transaction being undone. When a log record is processed during UNDO, the prevLSN (or UndoNxtLSN, in the case of a CLR) is entered as the next LSN to be undone for that transaction. Then the UNDO pass moves on to the log record whose LSN is the most recent of the next LSNs to be redone. UNDO continues backward in the log until all of the transactions in the list have been undone up to and including their first log record. UNDO for transaction rollback works similarly to the UNDO pass of the restart algorithm as described above. The only difference is that during transaction rollback, only a single transaction (or part of a transaction) must be undone. Therefore, rather than keeping a list of LSNs to be undone for multiple transactions, rollback can simply follow the backward chain of log records for the transaction to be rolled back.

4 Research Issues and Summary

The model of ACID transactions that has been described in this chapter has proven to be quite durable in its own right, and serves as the underpinnings for the current generation of database and transaction processing systems. This chapter has focused on the issues of concurrency control and recovery in a centralized environment. It is important to note, however, that the basic model is used in many types of distributed and parallel DBMS environments and the mechanisms described here have been successfully adapted for use in more complex systems. Additional techniques, however, are needed in such environments. One important technique is two-phase commit, which is a protocol for ensuring that all participants in a distributed transaction agree on the decision to commit or abort that transaction.

While the basic transaction model has been a clear success, it's limitations have also been apparent for quite some time (e.g., [Gray81]). Much of the ongoing research related to concurrency control and recovery is aimed at addressing some of these limitations. This research includes the development of new implementation techniques, as well as the investigation of new and extended transaction models.

The ACID transaction model suffers from a lack of flexibility and the inability to model many types of interactions that arise in complex systems and organizations. For example, in collaborative work environments, strict isolation is not possible nor even desirable [Kort95]. Workflow management systems are another example where the ACID model, which works best for relatively simple and short transactions, is not directly appropriate. For these types of applications, a richer, multi-level notion of transactions is required.

In addition to the problems raised by complex application environments, there are also many computing environments for which the ACID model is not fully appropriate. These include environments such as mobile wireless networks, where large periods of disconnection are expected, and loosely-coupled wide-area networks (the Internet is an extreme example) in which the availability of systems is relatively low. The techniques that have been developed for supporting ACID transactions must be adjusted to cope with such highly-variable situations. New techniques must also be developed to provide concurrency control and recovery in non-traditional environments such as heterogeneous systems, dissemination-oriented environments, and others.

A final limitation of ACID transactions in their simplest form is that they are a general mechanism, and hence, do not exploit the semantics of data and/or applications. Such knowledge could be used to significantly improve system performance. Therefore, the development of concurrency control and recovery techniques that can exploit application-specific properties is another area of active research.

As should be obvious from the preceding discussion, there is still a significant amount of work that remains to be done in the areas of concurrency control and recovery for database systems. The basic concepts, however, such as serializability theory, two-phase locking, write ahead logging, etc. will continue to be a fundamental technology, both in their own right, and as building blocks for the development of more sophisticated and flexible information systems.

Defining Terms

ACID properties: The transaction properties of Atomicity, Consistency, Isolation, and Durability that are upheld by the DBMS.

abort: The process of rolling back an uncommitted transaction. All changes to the database state made by that transaction are removed.

checkpointing: An action taken during normal system operation that can help limit the amount of recovery work required in the event of a system crash.

commit: The process of successfully completing a transaction. Upon commit, all changes to the database state made by a transaction are made permanent and visible to other transactions.

concurrency control: The mechanism that ensures that individual users see consistent states of the database even though operations on behalf of many users may be interleaved by the database system.

concurrent execution: The (possibly) interleaved execution of multiple transactions simultaneously.

conflicting operations: Two operations are said to conflict if they both operate on the same data item and at least one of them is a $write()$.

deadlock: A situation in which a set of transactions is blocked, each waiting for another member of the set to release a lock. In such a case none of the transactions involved can make progress.

log: A sequential file that stores information about transactions and the state of the system at certain instances.

log record: An entry in the log. One or more log records are written for each update performed by a transaction.

Log Sequence Number (LSN): A number assigned to a log record, which serves to uniquely identify that record in the log. LSNs are typically assigned in a monotonically increasing fashion so that they provide an indication of relative position.

multiversion concurrency control: A concurrency control technique that provides read-only transactions with conflict-free access to previous versions of data items.

non-volatile storage: Storage, such as magnetic disks or tapes, whose contents persist across power failures and system crashes.

optimistic concurrency control: A concurrency control technique that allows transactions to proceed without obtaining locks and ensures correctness by validating transactions upon their completion.

recovery: The mechanism that ensures that the database is fault tolerant; that is, that the database state is not corrupted as the result of a software, system, or media failure.

schedule: A schedule for a set of transaction executions is a partial ordering of the operations performed by those transactions, which shows how the operations are interleaved.

serial execution: The execution of a single transaction at-a-time.

serializability: The property that a (possibly interleaved) execution of a group transactions has the same effect on the database, and produces the same output as some serial (i.e., non-interleaved) execution of those transactions.

STEAL/NO-FORCE: A buffer management policy that allows committed data values to be overwritten on non-volatile storage and does not require committed values to be written to non-volatile storage. This policy provides flexibility for the buffer manager at the cost of increased demands on the recovery sub-system.

transaction: A unit of work, possibly consisting of multiple data accesses and updates, that must commit or abort as a single atomic unit. Transactions have the ACID properties of Atomicity, Consistency, Isolation, and Durability.

two-phase locking (2PL): A locking protocol that is a sufficient but not a necessary condition for serializability. Two phase locking requires that all transactions be well-formed and that once a transaction has released a lock, it is not allowed to obtain any additional locks.

volatile storage: Storage, such as main memory, whose state is lost in the event of a system crash or power outage.

well-formed: A transaction is said to be well-formed with respect to reads if it always holds a shared or an exclusive lock on an item while reading it, and well-formed with respect to writes if it always holds an exclusive lock on an item while writing it.

Write Ahead Logging: A protocol that ensures all log records required to correctly perform recovery in the event of a crash a placed on non-volatile storage.

References

[Agra87] Agrawal, R., Carey, M., Livny, M., “Concurrency Control Performance Modeling: Alternatives and Implications”, *ACM Transactions on Database Systems*, 12(4), December, 1987.

[Bere95] Berenson, H., Bernstein, P., Gray, J., Melton, J., Oneil, B., Oneil, P., “A Critique of ANSI SQL Isolation Levels”, *Proc. of the ACM SIGMOD International Conference on the Management of Data*, San Jose, CA., June, 1995.

[Bern87] Bernstein, P., Hadzilacos, V., Goodman, N., *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[Bjor73] Bjork, L., “Recovery Scenario for a DB/DC System”, *Proc. of the ACM Annual Conference*, Atlanta, 1973.

[Davi73] Davies, C., “Recovery Semantics for a DB/DC System”, *Proc. of the ACM Annual Conference*, Atlanta, 1973.

[Eswa76] Eswaran, L, Gray, J., Lorie, R., Traiger, I., “The Notion of Consistency and Predicate Locks in a Database System”, *Communications of the ACM*, 19(11), November, 1976.

[Gray75] Gray, J., Lorie, R., Putzolu, G., Traiger, I., “Granularity of Locks and Degrees of Consistency in a Shared Database”, *IFIP Working Conference on Modelling of Database Management Systems*, 1975.

[Gray81] Gray, J., “The Transaction Concept: Virtues and Limitations”, *Proc. of the Seventh International Conference on Very Large Databases*, Cannes, 1981.

[Gray93] Gray, J., Reuter, A., *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, San Mateo, CA, 1993.

[Haer83] Haerder, T., Reuter, A., “Principles of Transaction-Oriented Database Recovery” *ACM Computing Surveys*, 15(4), 1983.

[Kort95] Korth, H., "The Double Life of the Transaction Abstraction: Fundamental Principle and Evolving System Concept", *Proc. of the Twenty-First International Conference on Very Large Databases*, Zurich, 1995.

[Kung81] Kung, H, and Robinson, J., "On Optimistic Methods for Concurrency Control", *ACM Transactions on Database Systems*, 6(2), 1981.

[Lome77] Lomet, D., "Process Structuring, Synchronization and Recovery Using Atomic Actions" *SIGPLAN Notices* 12(3), March, 1977.

[Moha92a] Mohan, C., "ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes", , *Proc. of the 16th International Conference on Very Large Data Bases*, Brisbane, August, 1990.

[Moha92b] Mohan, C., Haderle, D. Lindsay, B., Pirahesh, H., Schwarz, P., "ARIES: A Transaction Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", *ACM Transactions on Database Systems*, 17(1), March, 1992.

[Papa86] Papdimitriou, C., *The Theory of Database Concurrency Control*, Computer Science Press, Rockville, MD, 1986.

[Reed83] Reed, D., "Implementing Atomic Actions on Decentralized Data" *ACM Transactions on Computer Systems* 1(1) February, 1983.

[Rose77] Rosenkrantz, D., Sterns, R., Lewis, P., "System Level Concurrency Control for Distributed Database Systems", *ACM Transactions on Database Systems*, 3 (2), 1977.

For Further Information

For many years, what knowledge that existed in the public domain about concurrency control and recovery was passed-on primarily though the use of multiple-generation copies of a set of lecture notes written by Jim Gray in the late seventies ("Notes on Database Operating Systems" in *Operating Systems: An Advanced Course* published by Springer-Verlag, Berlin, 1978). Fortunately, this state of affairs has been supplanted by the publication of *Transaction Processing: Concepts and Techniques* by Jim Gray and Andreas Reuter (Morgan Kaufmann, San Mateo, 1993). This latter book contains a detailed treatment of all of the topics covered in this chapter, plus many others that are crucial for implementing transaction processing systems.

An excellent treatment of concurrency control and recovery theory and algorithms can be found in *Concurrency Control and Recovery in Database Systems* by Phil Bernstein, Vassos Hadzilacos, and Nathan Goodman, (Addison-Wesley, Reading MA, 1987). Another source of valuable information on concurrency control and recovery implementation is the series of papers on the ARIES method by C. Mohan and others at IBM, some of which are referenced in this chapter. The book *The Theory of Database Concurrency Control* by Christos Papadimitriou (Computer Science Press, Rockville MD, 1986) covers a number of serializability models.

The performance aspects of concurrency control and recovery techniques have been only briefly addressed in this chapter. More information can be found in the recent book *Performance of Concurrency Control Mechanisms in Centralized Database Systems* edited by Vijay Kumar (Prentice Hall, Englewood Cliffs, NJ, 1996). Also, the performance aspects of transactions are addressed in *The Benchmark Handbook: For Database and Transaction Processing Systems (second edition)* edited by Jim Gray (Morgan Kaufmann, San Mateo, 1993).

Finally, extensions to the ACID transaction model are discussed in *Database Transaction Models* edited by Ahmed Elmagarmid (Morgan Kaufmann, San Mateo, 1993). Papers containing the most recent work on related topics appear regularly in the ACM SIGMOD Conference and the International Conference on Very Large DataBases (VLDB), among others.