CHAPTER 15

DATA INTEGRITY

CHAPTER OBJECTIVES

- Learn what data integrity is and why it is extremely important
- Note the nature and significance of a database transaction
- Study the meaning of ACID properties of transactions and appreciate why a database system must preserve these
- Understand concurrent transactions and the types of problems they could cause
- Discuss serializability and recoverability of transactions
- Examine various methods of concurrency control
- Classify the potential types of failures in a database environment and review recovery concepts
- Survey logging and log-based recovery techniques
- Inspect shadow paging and its role as a recovery method

Take the case of a banking database system. Consider this scenario. Assume that one of the bank's customers wants to transfer $50,000 from his or her savings account to the his or her checking account to cover a check issued for an important deal. The check bounces, and on investigation the bank discovers that although $50,000 was deducted from the customer's savings account the amount was not added to the checking account. The bank loses an important but irate customer. Why? What has happened?

That database transaction transferring the amount from savings to checking accounts did not leave the database in a consistent state. Only one part of the trans-

Database Design and Development: An Essential Guide for IT Professionals by Paulraj Ponniah
ISBN 0-471-21877-4 Copyright © 2003 by John Wiley and Sons, Inc.

action updated the savings account database table; the other part meant to update the checking account database table failed. The database became inconsistent as far this customer's accounts were concerned.

Data integrity means absence of inconsistent data. The data in the database must be correct, valid, and accurate at all times. None of the transactions that update the database can leave the database in an inconsistent state. Transactions must preserve consistency; they must not corrupt the correctness and validity of the database.

In a typical database environment, thousands of transactions interact with the database and many of these update data. Concurrent transactions may attempt to update the same data element simultaneously. A database is also vulnerable to several types of failures. In such unsettling circumstances, the database system must maintain data consistency, correctness, and validity.

First, we will explore the nature of a proper transaction and its requisite properties for data integrity. We will examine how improper executions of transactions could cause problems, and we will review solution options. We will also discuss failures of the database caused by external elements and present methods to recover from such failures.

TRANSACTION PROCESSING

Consider the database systems that support businesses like airlines, banking, finance, and retail. These are large transaction processing systems. Transactions in the tens of thousands interact with databases for airline reservations, banking operations, supermarket sales, credit card processing, stock buying and selling, and so on. Such critical database systems must be available at all times and must provide fast response times for hundreds of concurrent users. Some of these transactions just read the database and display data, but most of the transactions perform some type of updating.

A database system is susceptible to two major types of problems. First, concurrent transactions attempting to update the same data item may affect one another adversely and produce incorrect and inconsistent results. Second, hardware or software malfunctions could cause failures of the database and ruin its content. Concurrency control and recovery from failures form the foundation for maintaining data integrity.

Transaction Concepts

Let us get back to the example of transfer of funds from a savings account to a checking account. Figure 15-1 illustrates the execution of this transaction.

First, note that this transaction would be part of an application program usually written in a high-level language. For a transaction, the terminal points are marked with BEGIN and COMMIT statements. Consider a transaction as a unit of an application program. A program consists of one or more transactions. When you execute a program, you may be executing many transactions. Next, observe that in the example shown of a transaction, although the transaction is considered as one unit, it has two database updates—one to update the savings account record and the other to update the checking account record. Before the execution of the

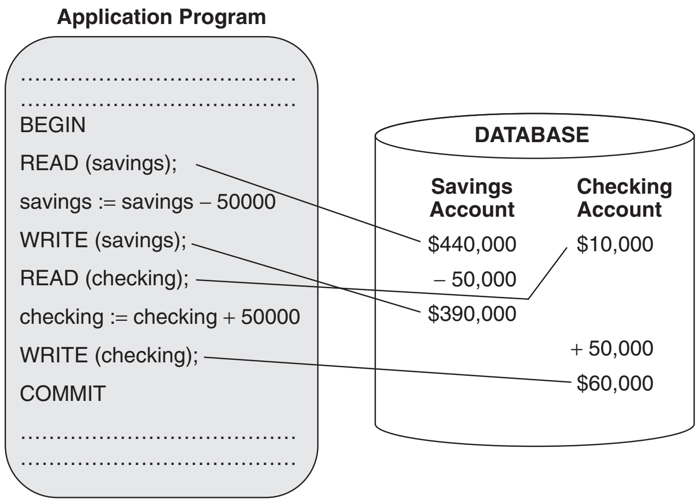

Figure 15-1 Example of a successful transaction.

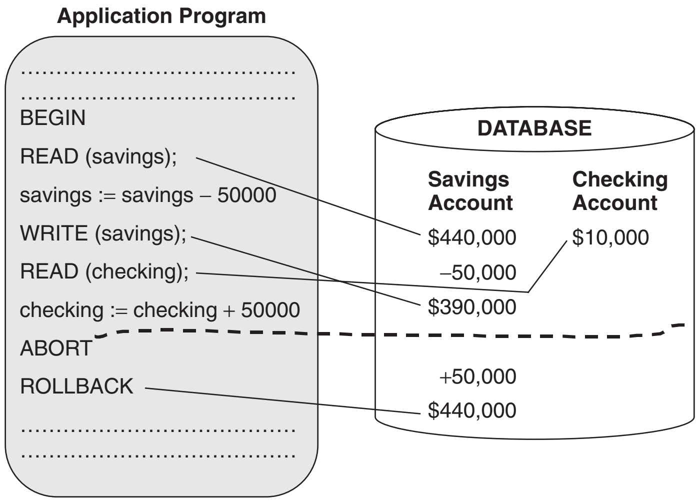

Figure 15-2 Example of an aborted transaction.

transaction, the combined total amount in both accounts is $450,000; after the completion of the transaction, the combined total amount is still the same, preserving data consistency.

Now assume that some program malfunction occurs immediately after the update to the savings record but before the update to the checking record. The program detects the malfunction and rolls back the effect of the update to the savings record. Figure 15-2 shows the action of the transaction in this case. Note the BEGIN and ROLLBACK statements demarking the start and end of the transaction.

In the successful execution of the transaction in the first case, the database is taken from an old consistent state to a new consistent state. When program aborts and the effects of incomplete updates are rolled back, the database is restored to its old consistent state. In either case, the transaction leaves the database in a consistent state.

Let us now summarize the key points of a database transaction:

- A transaction is an atomic unit of work.
- A transaction must leave the database in a consistent state; it must either be completed in its entirety or not all.
- The database management system cannot detect the beginning and end points of each individual transaction within an application unless the boundary points are marked by specific statements.
- Data manipulation languages provide specific statements of BEGIN, COMMIT, ABORT, and ROLLBACK that can be included in the application program. The BEGIN statement signifies the start of a transaction; COMMIT the end of a transaction after successful database updates; ROLLBACK the end of a transaction after rolling back the effects of unsuccessful database updates.
- When a transaction completes, two outcomes are possible—a new consistent state of the database or partial database updates rolled back and the database restored to its old consistent state.
- Transactions of the same program cannot be nested. Only while a transaction is in progress can COMMIT and ROLLBACK commands be performed.
- Each transaction must be executed only once as written, not repeatedly.
- If the start and end of transactions are not marked with specific statements, the database management assumes the beginning and end of the entire program as the start and end points of the transactions within the program.

Properties of Transactions

You must already have noticed that database transactions need to behave in a certain way to preserve data consistency, correctness, and validity. Unless transactions execute properly, users cannot trust the data in the database. They cannot depend on the database system to provide them with correct and consistent information.

Analyzing the required behavior of database transactions, we come up with four basic, essential properties that each database transaction must possess. The acronym ACID refers to these fundamental properties: A for atomicity, C for consistency, I for isolation, and D for durability. When a transaction has these four properties, it will maintain the consistency, correctness, and validity of a database. Let us examine and describe these properties.

Atomicity A transaction is said to be atomic if it executes all of its database updates or none at all. An atomic transaction is considered to be one complete unit of work—either the entire work in the unit gets done or the bad effects of no part of the work are left behind in the database.

The transaction management component of the database management system (DBMS) handles atomicity.

Consistency This property of a transaction preserves database consistency. If a transaction executes successfully and performs its database updates properly, the database is transformed from one consistent state to another consistent state.

Preservation of database consistency also results from the other three properties of a transaction.

Isolation For the user, a transaction must appear as though it executed by itself without interference from any other concurrent transaction. In practice, however, many concurrent database transactions execute simultaneously. For performance reasons, operations of one transaction are interleaved with operations of other transactions. However, the database system must guarantee the effects of database updates for every pair of concurrent transactions to be as though one transaction executed from beginning to end and then the second started, or vice versa. Transactions must execute independently, and intermediary or partial results of incomplete transactions must be transparent to other transactions.

The concurrency control component of the DBMS handles isolation.

Durability The effects of database updates by successfully completed transactions must be permanent in the database. Even if and when there are subsequent failures, the effects must persist in the database.

The recovery management component of the DBMS handles durability.

Let us revisit the example of the transfer of funds from savings to checking accounts. Look at Figures 15-1 and 15-2 again. If you name the transaction T1, then the coding shown below may represent the transaction:

```
T1:   BEGIN
      READ (savings);
      savings := savings – 50000;
      WRITE (savings);
      READ (checking);
      checking := checking + 50000;
      WRITE (checking);
      COMMIT
```

Walk through the ACID requirements for transaction T1 and note how database integrity is preserved.

Atomicity. As shown in Figures 15-1 and 15-2, the initial values of *savings* and *checking* before T1 started are 440000 and 10000, respectively. Figure 15-1, which presents the successful completion of T1, represents the case in which all database updates are performed and the transaction is executed in its entirety. Figure 15-2 presents a database failure after WRITE (*savings*) operation. In this case, the transaction management component ensures atomicity by removing the effect of WRITE (*savings*). The effects of all operations of a transaction are left in the database or none at all. This is the requirement of the property of atomicity.

Consistency. The consistency requirement in this case is that the combined total of savings and checking balances must be $450,000. From Figure 15-1, you will note that the combined total is the same at the start as well as at the end of the transaction. Figure 15-2 similarly shows the combined total to be same at the start and end of the transaction. Whether the transaction reaches a successful end or is aborted, consistency of the database is maintained.

Isolation. To understand the property of isolation, consider another transaction, T2, that simply makes a deposit of $30,000 to the savings account. Now assume that T1 and T2 execute concurrently and that the database operations of T1 and T2 are interleaved. If T1 executes first, the balance in the savings account is $390,000. If T2 follows, at the end of T2 the balance in the savings account is $420,000. Now reverse the order. If T2 executes first, the savings balance is $470,000. If T1 follows, at the end of T1 the savings balance is $420,000. The property of isolation requires that even if the database operations of T1 and T2 are interleaved, the final savings balance is still $420,000 as if the two transactions executed independently without any interference between them.

Durability. Look again to Figures 15-1 or 15-2. At the successful end of the transaction shown in Figure 15-1, the checking balance is $60,000 and the savings balance is $390,000. Figure 15-2 shows these amounts as $10,000 and $440,000 respectively. If transaction T1 executed successfully, the values of $60,000 and $390,000 must persist in the database. Even after a hardware failure or a system failure, if there are no intervening transactions affecting these balances, these values must persist.

Transaction States

As a transaction proceeds through its various operations, it passes through several states. Each state marks a transition point where some specific action takes place or which signals a specific stage in the processing. You know that the database system is a transaction-processing environment. To understand and appreciate the types of data integrity problems and possible solution options, you need a good understanding of transaction processing. This means that you need to trace through the different states of a transaction, clearly comprehend the significance of each state and also what happens between two consecutive states, and understand the branch points. The recovery manager of the DBMS keeps track of the operations of each transaction.

Figure 15-3 illustrates the various operations and the state of the transaction at the end of each operation.

Here is a list of operations and states as shown in the figure:

BEGIN	Marks the beginning of the transaction. Transaction enters ***active*** state.
READ	Transaction performs read operation. Transaction remains in ***active*** state.
WRITE	Transaction performs write (insertion or update or deletion) operation. Transaction remains in ***active*** state.

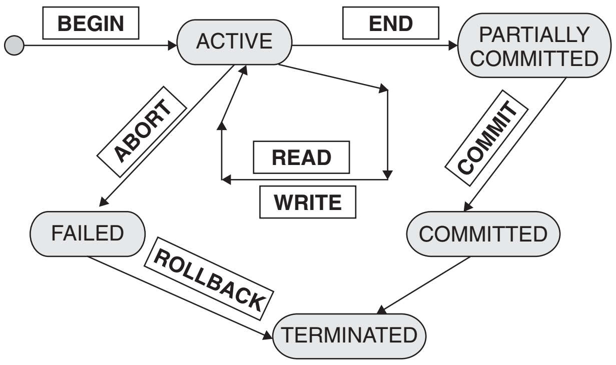

Figure 15-3 Operations and states of a transaction.

END	Indicates end of transaction read or write operations. Transaction proceeds to ***partially committed*** state.
COMMIT	Signifies successful end of transaction so that any database updates can be safely confirmed or committed to the database. Updates cannot be undone. Transaction moves to ***committed*** state and finally reaches the ***terminated*** state.
ABORT	Indicates failure of transaction so that effects of any database updates must be undone. Transaction proceeds to ***failed*** state from either ***active*** state or ***partially committed*** state depending on the nature of the failure.
ROLLBACK	Indicates that the recovery manager needs to roll back database updates and restore database to prior consistent state. After rollback, transaction reaches the ***terminated*** state.

Processing of Transactions

According to C. J. Date, an eminent database expert, "The fundamental purpose of the database system is to carry out *transactions*." As already mentioned, a database system is a transaction-processing environment. The database is there to support processing of transactions.

In the process of supporting transactions, what is the impact of transactions on the database itself? How do the users consider transactions? What purpose do transactions serve for reading and changing database content? Let us go over a few important aspects of transactions.

Transaction Structure Typically, a transaction consists of three major phases:

Input phase	Receive messages and parameters to carry out the necessary interaction with the database.
Database interaction	Perform database read or write.

Output phase	Provide messages to be passed along regarding the successful or unsuccessful completion of the database interaction. Retrieve and provide data for completion of the application program. or update database as needed.

Transaction Types From the point of view of database interactions, transactions fall into two broad types:

Read-only transactions. These do not change the database content. Therefore, any number of read-only transactions can share the database content simultaneously.

Update transactions. These change the database content. Therefore, while processing update transactions, conflicts could arise among simultaneous transactions attempting to update the same data items.

Single-User Environment Personal database systems are single-user environments. Concurrent transactions and resulting concurrency problems are typically absent in such an environment. However, different types of hardware and software failures could cause data integrity problems.

Multi-User Environment Most database systems are multiuser environments. In such environments, concurrent transactions are the norm. Concurrent transactions could give rise to transaction conflicts.

Concurrency Control A multiuser environment requires effective measures and techniques for concurrency control. Each transaction must perform independently as though it were executing in isolation. Concurrent transactions must employ locking mechanisms to prohibit other transactions attempting to update the same data items.

Data Granularity A database is a collection of named data items intended to be shared among multiple users. When multiple users need to update the same item, the data item must be locked by one transaction while it goes through the database interaction phase. How many data items do you lock and prevent other users from updating? Just specific fields, specified rows, or the whole table? Data granularity, in this context, refers to the level and scope of database content that needs to be locked at a given time.

Dependability When user requests are processed as transactions, the users expect the transactions to preserve the integrity of the database. Processing of concurrent transactions in parallel must be transparent to the users.

Availability The database must be kept available to the users as and when needed. Even after system and hardware failures, the database must be quickly restored to a stable and consistent state.

Integrity Considerations

As you have seen, the transaction-processing database system is subject to potential integrity problems. Users may be exposed to inconsistent, incorrect, and invalid data. Integrity functions in the database management system must be able to preserve the integrity of the database by performing the following actions:

- Monitor and scrutinize all transactions, especially those with database updates
- Detect any data integrity violations by transactions
- If a violation is detected, take appropriate action such as
  - Rejection of the operation
  - Reporting of the violation
  - Correction of error condition, if possible

Specific components exist in the DBMS for preserving database integrity. Figure 15-4 presents an overview of integrity components. Each component performs a particular set of functions.

While processing concurrent transactions in a database environment, the operations of different transactions interleave with one another. Integrity problems might arise because of such interleaved processing of transactions. We will discuss concurrency control in detail in subsequent sections.

Database failures result in potential integrity problems. The transactions that are in flight during a disk crash may not complete all the database updates. They may not have a chance to roll back the incomplete updates. We will examine database failures in depth in another subsequent section.

Here, let us turn our attention to a few other types of integrity problems and look at the solution options. For our discussion, we will adopt a very common example of a suppliers and parts database schema. Figure 15-5 presents the relational tables of the suppliers and parts database.

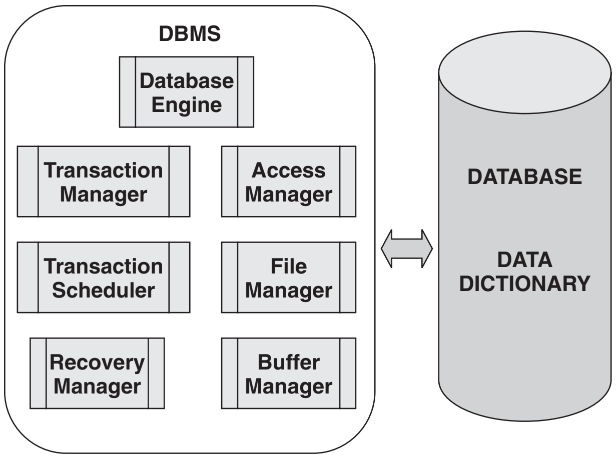

Figure 15-4 DBMS integrity components.

SUPPLIER

SupplierID	SupplierName	SupplierStatus	SupplierCity	ShipCode

PART

PartNo	PartName	PartColor	PartWeight

SUPPLY

SupplierID	PartNo	Quantity

SHIP-METHOD

ShipCode	ShipDesc	UnitCost

Figure 15-5 Suppliers and parts database.

Domain Integrity Recall the DDL statements defining a relational schema. Each attribute has an underlying domain; the attribute can take values only from a specific domain of values and can only be of a particular data type. When a transaction inserts or updates a row in a particular table, the values of the attributes added or changed must conform to the domain values of the attributes. If they do not, then the transaction attempts to violate domain integrity rules. Violations of domain integrity rules are more common than you might imagine, and they must be addressed to preserve database integrity.

Figure 15-6 shows the DDL statements for defining the relational database schema for the suppliers and parts database.

Note the following examples of domain integrity rules as indicated in the schema definition. The transaction manager of the DBMS must detect any attempted violation and reject the transaction with a message. In some cases, the system may substitute default values and allow the transaction to proceed.

Data types. Each attribute must conform to the defined data type.

Null values. Null values are not allowed for the attribute SupplierName.

Allowable values. Only certain values are allowed for the attribute SupplierStatus.

Range values. Values for PartWeight must fall within the given range.

Missing values. Substitute a given color if value of PartColor is missing.

Referential Integrity Refer to Figure 15-6 and note the ON DELETE clauses. These specify conditions for enforcing referential integrity constraints. Also, note the NOT NULL conditions for foreign keys. These specify actions to preserve

```
CREATE SCHEMA PARTS-SUPPLY
          AUTHORIZATION McMillan;
CREATE DOMAIN ItemIdentifier CHARACTER (4) DEFAULT “ZZZZ”
          CHECK (VALUE IS NOT NULL);
CREATE TABLE SUPPLIER(
          SupplierID                    ItemIdentifier,
          SupplierName                  CHARACTER (35) NOT NULL,
          SupplierStatus                CHARACTER (2),
                    CHECK (SupplierStatus IN ’10’, ’20’, ’30’, ’40’),
          SupplierCity                  CHARACTER (25),
          ShipCode                      CHARACTER (6),
          PRIMARY KEY (SupplierID),
          FOREIGN KEY (ShipCode) REFERENCES
          SHIP-METHOD(ShipCode)
                     ON DELETE SET NULL);
CREATE TABLE PART (
          PartNo                        CHARACTER (8),
          PartName                      CHARACTER (25),
          PartColor                     CHARACTER (15)
                                        DEFAULT  ‘RED’,
          PartWeight                    DECIMAL (5, 2)  NOT NULL,
          CHECK (PartWeight < 900.00 AND PartWeight > 99.00),
          PRIMARY KEY (PartNo));

CREATE TABLE SHIP-METHOD (
                ShipCode                          CHARACTER (6),
                ShipDesc                          CHARACTER (25),
                UnitCost                          DECIMAL (5, 2)
                                                  NOT NULL,
                PRIMARY KEY (ShipCode) );
CREATE TABLE SUPPLY (
                SupplierID                        ItemIdentifier,
                PartNo                            CHARACTER (8),
                Quantity                          INTEGER,
                PRIMARY KEY (SupplierID, PartNo),
                FOREIGN KEY (SupplierID) REFERENCES SUPPLIER (SupplierID)
                                  ON DELETE CASCADE,
                FOREIGN KEY (PartNo) REFERENCES PART (PartNo)
                                  ON DELETE CASCADE);
```

Figure 15-6 Schema definition: suppliers and parts.

referential integrity. The integrity subsystem of the DBMS handles attempted referential integrity violations.

Business Rules The integrity subsystem can also enforce business rules through stored procedures and triggers. A stored procedure is a named module of code defining a business rule and specifying the action to be taken in case of attempted violation. A trigger is a special type of stored procedure. A trigger module can be made to fire when a particular type of insert, update, or delete takes place. Figure 15-7 presents a sample trigger.

CONCURRENT TRANSACTIONS

The database environment supports processing of transactions, several of which may interact with the database at the same time. Transaction T1 may start at a certain

```
CREATE      TRIGGER   PARTWEIGHT
BEFORE                           INSERT, UPDATE (PartWeight)
ON                                         PART
FOR EACH ROW
DECLARE
                    BigPartWeight                    PART.PartWeight%type   ;
BEGIN
                    SELECT PartWeight        INTO        BigPartWeight
                    FROM PART
                    WHERE PartName = 'BigPart'     ;

                    IF          :new.PartWeight > BigPartWeight         AND
                                :old.PartName < > 'BigPart'             THEN
                                RAISE APPLICATION ERROR (-20200,
                                'Part cannot have overweight'   ) ;
                    ENDIF  ;
 END  ;
```

Figure 15-7 Sample trigger module.

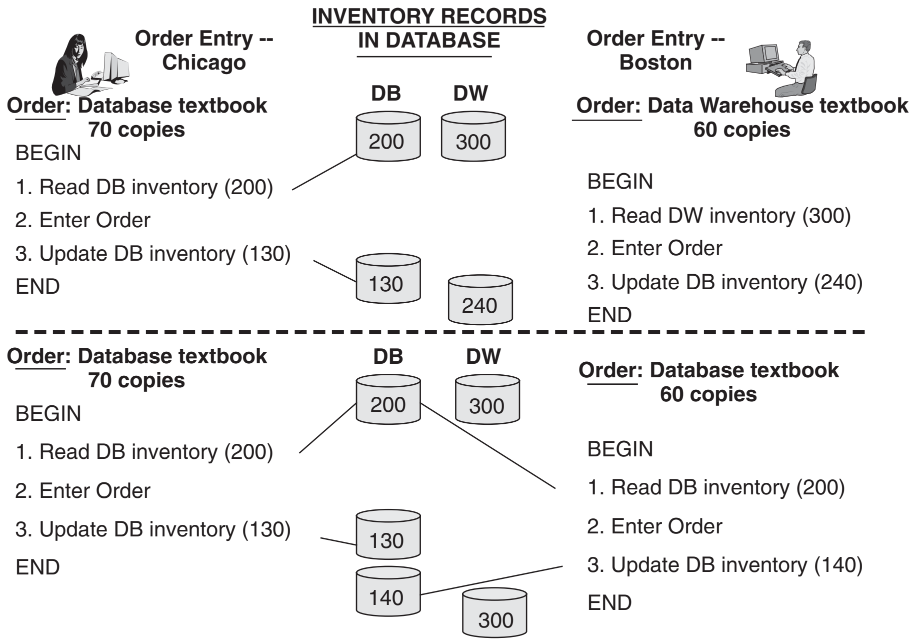
INVENTORY RECORDS IN DATABASE
Order Entry -- Chicago
Order Entry -- Boston
Order: Database textbook 70 copies
BEGIN
1. Read DB inventory (200)
2. Enter Order
3. Update DB inventory (130)
END
DB
DW
200
300
130
240
Order: Data Warehouse textbook 60 copies
BEGIN
1. Read DW inventory (300)
2. Enter Order
3. Update DB inventory (240)
END
Order: Database textbook 70 copies
BEGIN
1. Read DB inventory (200)
2. Enter Order
3. Update DB inventory (130)
END
DB
DW
200
300
130
140
300
Order: Database textbook 60 copies
BEGIN
1. Read DB inventory (200)
2. Enter Order
3. Update DB inventory (140)
END

Figure 15-8 Concurrent transactions.

time and start executing. Within microseconds, a second transaction, T2, may start execution from another place and interact with the database. Transaction T2, with fewer reads and updates, may finish sooner, and transaction T1, although started earlier, may end later. These are concurrent transactions or transactions that interact with the database simultaneously. Transaction T1 may read and update a particular record in the database, while T2 may read and update an altogether different record. On the other hand, both may need to read and update the same record.

Figure 15-8 presents these two cases: In the first case, T1 reads and updates the inventory record for database textbook and T2 reads and updates the inventory record for data warehousing textbook; in the second case, both transactions need to read and update the inventory record for database textbook.

Note the two cases carefully. In both cases, transactions T1 and T2 execute simultaneously. Observe the second case and see why this case gives rise to a potential problem. In the second case, both transactions need to read and update the same record, namely, the inventory record of database textbook. Instead of allowing transactions T1 and T2 to execute concurrently, let the system allow transaction T1 to execute independently first from beginning to end and then start transaction T2 and allow T2 also to execute from beginning to end. If this is done, you will not have an integrity problem even though both transactions read and update the same record.

Motivation for Concurrency

The question naturally arises: Why not execute transactions sequentially and avoid potential integrity problems? Note that in the above two cases, because the two transactions are permitted to execute concurrently, the read and write operations within the two transactions can be interleaved. How does this help? Does not this interleaving of operations cause integrity problems? Let us examine possible reasons for allowing transactions to execute concurrently.

Remember that a database system is a transaction-processing environment and any method to speed up the overall processing is highly welcome in such an environment. A database system exists to support the processing of transactions—every effort to accelerate transaction processing deserves top priority.

A transaction consists of CPU processing and I/O operations. While one transaction waits for I/O, CPU can process another transaction. In this manner, parallelism of the overall process can be achieved. You have noted this type of parallelism in the above example. This is the primary motivation for concurrent transactions.

Here is a list of major benefits of transaction concurrency:

- Overall increase in system throughput
- Reduced response times
- Simultaneous sharing of data
- Executing a short transaction and a long transaction concurrently allows the shorter transaction to finish quickly without waiting for the longer transaction to end
- Avoids unpredictable delays in response times

Concurrency Problems

Perhaps you are now ready to accept that there are potential benefits of allowing the concurrent processing of transactions. Especially in a large transaction-processing environment and in an environment with long-running transactions, concurrent transaction processing seems to be inevitable.

You are already aware of potential integrity problems when two or more concurrent transactions seek to read and write the same database item. In this case, we say that the two concurrent transactions are in conflict. Two operations on

TIME	T1 -- Chicago Order:	T2 -- Boston Order:	INVENTORY
	DB 70 copies	DB 60 copies	
t1	BEGIN T1		200
t2	READ inventory	BEGIN T2	200
t3	inventory = inventory – 70	READ inventory	200
t4	WRITE inventory	inventory = inventory – 60	130
t5	COMMIT	WRITE inventory	140
t6	END	COMMIT	140
t7		END	140

Figure 15-9 Concurrent transactions: lost update.

the same database item conflict with each other if at least one of them is a write operation.

When two transactions execute concurrently, the operations of one transaction interleave with the operations of the other transaction. So it becomes obvious that, to avoid integrity problems, all types of interleaving of operations may not be allowed.

Three potential problems arise when operations of concurrent transactions are interleaved: the lost update problem, the problem resulting from dependency on uncommitted update, and the inconsistent summary problem. We will discuss these problems individually. If concurrency has to be allowed in transaction processing, then these problems must be resolved. Before considering solution options, try to get a clear understanding of the nature of the three types of problems.

Lost Update In this scenario, an apparently successful database update by one transaction can be overridden by another transaction. Figure 15-9 repeats the second case presented in the Figure 15-8. Here, both of transactions T1 and T2 read and write the inventory record for database textbook.

Note the apparent successful update by transaction T1 writing out the inventory balance as 130. Transaction T2, however, overwrites the inventory balance as 140, which is incorrect.

Dependency on Uncommitted Update Just as in the previous case, the operations of transactions T1 and T2 interleave. Transaction T2 starts slightly later after both read and write operations of T1 are completed but before the update is committed to the database. However, transaction T1 aborts for whatever reason and rolls back. Figure 15-10 presents this case.

Why does this case present an integrity problem? Transaction T2 is allowed to see the intermediate result of transaction T1 before the update is committed. So transaction T2 starts out with the uncommitted value of 130 and comes up with the final result of 70. That would be correct if transaction T1 had not aborted and rolled back its update.

Inconsistent Summary In this case, each transaction does not both read and write. One transaction reads several records while the second transaction, interleaved with the other transaction, updates some of the records being read by the first transaction. Figure 15-11 illustrates this situation.

TIME	T1 -- Chicago Order:	T2 -- Boston Order:	INVENTORY
	DB 70 copies	DB 60 copies	
t1	BEGIN T1		200
t2	READ inventory		200
t3	inventory = inventory – 70		200
t4	WRITE inventory	BEGIN T2	130
t5		READ inventory	130
t6	ROLLBACK	inventory = inventory – 60	70
t7	END	WRITE inventory	70
t8		COMMIT	70
t9		END	200

Figure 15-10 Concurrent transactions: uncommitted dependency.

TIME	T1 -- Count books on DB, DW, VB	T2 -- update inventory for DB, DW	DB	DW	VB	SUM
t1	BEGIN T1		200	250	300	
t2	sum = 0	BEGIN T2	200	250	300	0
t3	READ DB inventory	READ DB inventory	200	250	300	0
t4	sum = sum + DB inventory	DB inventory = DB inventory – 70	200	250	300	200
t5	READ DW inventory	WRITE DB inventory	130	250	300	200
t6	sum = sum + DW inventory	READ VB inventory	130	250	300	450
t7		VB inventory = VB inventory – 60	130	250	300	450
t8		WRITE VB inventory	130	250	240	450
t9	READ VB inventory	COMMIT	130	250	240	450
t10	sum = sum + VB inventory	END	130	250	240	690
t11	COMMIT		130	250	240	690
t12	END		130	250	240	690

Figure 15-11 Concurrent transactions: inconsistent summary.

Transaction T1 simply reads the inventory records of the books on the database, data warehousing, and Visual Basic to find the total number of these books in the warehouse. In the meantime, T2 reads and updates two of these records—one after it is read by T1 and another before it is read by T1. Note the error in the summary reads and the final sum arrived at the end of transaction T1.

Transactions and Schedules

The foregoing discussions clearly establish the need for concurrency control techniques that can preserve data integrity even while allowing interleaving of operations belonging to different transactions. But before you can appreciate the concurrency control techniques, you need to study the operations of concurrent transactions as they impact the database.

From the point of view of the database, a transaction is simply a list of operations such as BEGIN, READ, WRITE, END, COMMIT, and ABORT. In concurrent executions of different transactions, these operations are interleaved. The READ of one transaction may be followed by the READ of another transaction. After that, you may have the WRITE of the first transaction, and so on. These operations interact with the database. If you observe the operations over a certain time interval, you note a series of operations belonging to various concurrent transactions interacting with the database. How these operations of different transactions are scheduled—in what sequence and at which intervals—has a direct bearing on data integrity. Proper scheduling would result in preservation of integrity.

Let us review a few concepts about scheduling.

Schedule. Consists of a sequence of operations in a set of concurrent transactions where the order of the operations of each transaction is preserved.

Serial schedule. Operations of different transactions are not interleaved. Leaves database in a consistent state.

Nonserial schedule. Operations of different transactions are interleaved.

Complete schedule. A schedule that contains a COMMIT or ABORT for each participating transaction.

Cascadeless schedule. A schedule is said to be cascadeless, that is, to be avoiding cascading rollbacks, if every transaction in the schedule reads only data items written by committed transactions.

Figure 15-12 explains the definitions using the operations of two transactions, T1 and T2. Note each example and understand the significance of the variations in the schedule.

Now you will readily conclude that, given the operations of even four transactions, you can arrange the operations in different ways and come up with many

T1	T2
READ (A)	
A := A + 100	READ (A)
WRITE (A)	A := A – 50
READ (B)	WRITE (A)
B := B + 200	READ (B)
WRITE (B)	B := B – 100
	WRITE (B)

SCHEDULE

T1	T2
BEGIN	
READ (A)	BEGIN
A := A + 100	READ (A)
WRITE (A)	A := A – 50
READ (B)	WRITE (A)
B := B + 200	READ (B)
WRITE (B)	B := B – 100
COMMIT	ABORT
END	ROLLBACK
	END

COMPLETE NON-SERIAL SCHEDULE

T1	T2
BEGIN	
READ (A)	
A := A + 100	
WRITE (A)	
READ (B)	
B := B + 200	
WRITE (B)	
COMMIT	
END	
	BEGIN
	READ (A)
	A := A – 50
	WRITE (A)
	READ (B)
	B := B – 100
	WRITE (B)
	COMMIT
	END

COMPLETE SERIAL SCHEDULE

T1	T2
BEGIN	
READ (A)	
A := A + 100	
WRITE (A)	
COMMIT	BEGIN
END	READ (A)
	A := A – 50
	WRITE (A)
	COMMIT
	END

CASCADELESS SCHEDULE

Figure 15-12 Transactions and schedules.

different schedules. In fact, the number of possible schedules for a set of n transactions is much larger than $n(n - 1)(n - 2)\ldots\ldots 3.2.1$. With just five transactions, you can come up with more than 120 different schedules. Some of these possible schedules preserve data integrity—many do not.

Obviously, all the possible serial schedules preserve data integrity. In a serial schedule, the operations of one transaction are all executed before the next transaction starts. However, serial schedules are unacceptable, because they do not promote concurrency. Concurrent processing of transactions is highly desirable. So, what types of nonserial schedules are safe to preserve data integrity? In what types of schedules is there no interference among the participating transactions? This leads us to the discussion of serializability and recoverability.

Serializability

A serial schedule, although leaving the database in a consistent state, is inefficient. However, a nonserial schedule, although improving efficiency, increasing throughput, and reducing response times, still has the potential of leaving the database in an inconsistent state. So the question is whether you can find an equivalent schedule that will produce the same result as a serial schedule. This new schedule must still be nonserial, to allow for concurrency of transaction execution, but behave like a serial schedule. Such a schedule is known as a serializable schedule.

Serializability finds nonserial schedules that allow transactions to execute concurrently without interference and leave the database in a consistent state as though the transactions executed serially.

Serializable Schedule Consider a schedule S of n transactions with interleaving operations. Schedule S is defined to be serializable if it is equivalent to some serial schedule of the same n transactions. Two schedules are equivalent if they produce the same final state of the database. A serializable, complete schedule leaves the database in a consistent state.

Figure 15-13 illustrates the equivalence between a given serializable schedule and a serial schedule.

Ordering of Operations In serializability, ordering of read and write operations is important. Consider the following cases for two concurrent transactions, T1 and T2:

- If both T1 and T2 only read a specific data item, there is no conflict; therefore, order of the operations is not important.
- If T1 and T2 read or write completely different data items, there is no conflict; therefore, order of the operations is not important.
- If T1 writes a specific data item and T2 reads or writes the same data item, conflict arises; therefore, order of the operations becomes important.

Forms of Equivalence When two consecutive operations in a schedule belonging to different transactions both operate on the same data item, and if at least one of the two operations is a write, then the two operations are in conflict. Therefore,

T1	T2
BEGIN	
READ (A)	
A := A + 100	
WRITE (A)	
READ (B)	
B := B + 200	
WRITE (B)	
COMMIT	
END	
	BEGIN
	READ (A)
	A := A – 50
	WRITE (A)
	READ (B)
	B := B – 100
	WRITE (B)
	COMMIT
	END

SERIAL SCHEDULE

T1	T2
BEGIN	
READ (A)	
A := A + 100	
WRITE (A)	
	BEGIN
	READ (A)
	A := A – 50
	WRITE (A)
READ (B)	
B := B + 200	
WRITE (B)	
COMMIT	
END	READ (B)
	B := B – 100
	WRITE (B)
	COMMIT
	END

EQUIVALENT SERIALIZABLE SCHEDULE

Figure 15-13 Equivalence of a serial and a serializable schedule.

it follows that, if two operations do not conflict, we can swap the order of the operations to produce another schedule that is equivalent to the original schedule.

Consider Schedule-1 as shown Figure 15-14.

Look for sets of two operations that do not conflict and swap the operations within each set. After swapping we arrive at Schedule-2 as shown in Figure 15-15.

Closely examine Schedule-2. You can see that Schedule-2 is a serial schedule obtained by swapping nonconflicting operations. Because Schedule-2 is obtained by swapping nonconflicting operations in Schedule-1, Schedule-1 and Schedule-2 are said to be conflict equivalent.

Conflict serializability

A schedule is conflict-serializable if it is conflict equivalent to a serial schedule.

Let us relax the equivalence conditions slightly and establish the concept of two schedules being view equivalent. Two schedules, Schedule-1 and Schedule-2, are view equivalent if the following conditions are satisfied:

- For a particular data item, if T1 reads the initial value of the data item in Schedule-1, then T1 must also read the initial value of the same data item in Schedule-2.
- For a particular data item, if T1 reads that data item in Schedule-1 and that value was produced by T2, then T1 must also read the data item in Schedule-2 that was produced by T2.

T1	T2
BEGIN	
READ (A)	
A := A + 100	
WRITE (A)	
	BEGIN
	READ (A)
	A := A – 50
	WRITE (A)
READ (B)	
B := B + 200	
WRITE (B)	
COMMIT	
END	READ (B)
	B := B – 100
	WRITE (B)
	COMMIT
	END

SCHEDULE-1
NON-SERIAL SCHEDULE

Figure 15-14 Example of a nonserial schedule.

T1	T2
BEGIN	
READ (A)	
A := A + 100	
WRITE (A)	
	BEGIN
	READ (A)
	A := A – 50
READ (B)	
B := B + 200	
	WRITE (A)
WRITE (B)	
COMMIT	READ (B)
END	B := B – 100
	WRITE (B)
	COMMIT
	END

WRITE (A) of T2 does not conflict with READ (B) of T1. Therefore, these two operations are swapped.

SCHEDULE-2
OBTAINED BY
TRANSACTION SWAPPING

Figure 15-15 Schedule obtained by operation swapping.

T1	T2
BEGIN	
READ (A)	
A := A + 100	
WRITE (A)	
READ (B)	
B := B + 200	
WRITE (B)	
COMMIT	
END	
	BEGIN
	READ (A)
	A := A – 50
	WRITE (A)
	READ (B)
	B := B – 100
	WRITE (B)
	COMMIT
	END

SCHEDULE-1
SERIAL SCHEDULE

T1	T2
BEGIN	
READ (A)	
A := A + 100	
WRITE (A)	
	BEGIN
	READ (A)
	A := A – 50
	WRITE (A)
READ (B)	
B := B + 200	
WRITE (B)	
COMMIT	
END	READ (B)
	B := B – 100
	WRITE (B)
	COMMIT
	END

SCHEDULE-2
VIEW EQUIVALENT TO
SCHEDULE-1

T1	T2
	BEGIN
	READ (A)
	A := A – 50
	WRITE (A)
	READ (B)
	B := B – 100
	WRITE (B)
	COMMIT
	END
BEGIN	
READ (A)	
A := A + 100	
WRITE (A)	
READ (B)	
B := B + 200	
WRITE (B)	
COMMIT	
END	

SCHEDULE NOT
VIEW EQUIVALENT TO
SCHEDULE-1

Figure 15-16 View equivalent schedules.

- For a particular data item, the transaction that performs the final write of that data item in Schedule-1 must be the same transaction that performs the final write of that data item in Schedule-2.

The above conditions ensure that each transaction reads the same values for the particular data item in both schedules and that both schedules produce the same final state of the database.

Figure 15-16 presents two schedules that are view equivalent.

View serializability

A schedule is view-serializable if it is view equivalent to a serial schedule.

In the next section, we will examine concurrency control options that establish serializability of schedules. When serializable schedules execute, we derive both benefits: (1) concurrent processing of transactions by interleaving operations and (2) preservation of database integrity.

Recoverability

In our discussion on serializability, we considered schedules that are serializable and therefore acceptable for preserving database consistency. The assumption had been that no transaction failures occur during concurrent executions of transactions. You know that this is not quite true. Concurrent transactions are as vulnerable to failures as serial executions. So we need to take into account the effect of transaction failures during execution of concurrent transactions.

T1	T2
BEGIN	
READ (A)	
A := A + 100	
WRITE (A)	
	BEGIN
	READ (A)
	COMMIT
READ (B)	END
B := B + 200	
WRITE (B)	
…………………	
………………..	

NON-RECOVERABLE SCHEDULE

T1	T2
BEGIN	
READ (A)	
A := A + 100	
WRITE (A)	
	BEGIN
	READ (A)
READ (B)	
B := B + 200	
WRITE (B)	
COMMIT	
END	COMMIT
	END

RECOVERABLE SCHEDULE

Figure 15-17 Nonrecoverable and recoverable schedules.

Suppose transactions T1 and T2 are executing concurrently. Assume that T1 fails for some reason. Then the effects of its interaction with the database must be rolled back. If T2 had read a data item updated by T1 and used the value for some processing, then T2 must also be aborted and rolled back. Only then is the schedule containing the operations of T1 and T2 a recoverable schedule.

Figure 15-17 shows an example of a schedule that is nonrecoverable and another equivalent schedule that is recoverable.

Here are some ways of describing the concept of a recoverable schedule:

- In a recoverable schedule, once a transaction is committed it should never be rolled back
- A schedule S is recoverable if no transaction T in S commits until all transactions that have written an item that T reads have committed.
- In a recoverable schedule with transactions T1 and T2, if T2 reads a data item previously written by T1, then the commit of T1 must precede the commit of T2.

CONCURRENCY CONTROL

We have discussed the need for concurrent processing of transactions. You have understood that allowing transactions to process concurrently with interleaving operations could result in conflicts between transactions interacting with the same data item in the database. We covered transaction schedules in sufficient detail. You have also realized that serializable and recoverable schedules would preserve database consistency and enable recovery from transaction failures.

Now let us consider solution options for addressing problems resulting from the concurrent processing of transactions. Concurrency control is the process of allowing and managing simultaneous operations of transactions without conflicts or interference. Without concurrency control techniques, simultaneous processing of transactions is not possible.

Broadly, there are two basic approaches to concurrency control: one is a pessimistic or conservative approach and the other is an optimistic approach. The pessimistic approach takes the view that you must take every precaution to prevent interference and conflicts among concurrent transactions. Such preventive measures involve heavy overhead in transaction processing. In the optimistic approach, you assume that interference and conflicts among concurrent transactions are rare and that therefore enormous system overhead for preventing conflicts is not warranted. Instead, you take action as and when such infrequent conflicts do arise.

The pessimistic or conservative method makes use of concurrency control protocols that ensure serializability. One set of protocols relate to locking of data items; the other set relate to timestamping of transactions. We will discuss both concurrency control techniques. We will also cover the optimistic approach to concurrency control. You will learn about the applicability of these techniques to specific database environments.

Lock-Based Resolution

Why do conflicts arise while transactions process concurrently? Think of the underlying reason. The operations in a schedule interact with the same data item; they need to read or write or do both operations on the same data item. We reviewed the three types of problems that can be caused by concurrent processing above.

An obvious method to ensure serializability is to require that operations interact with a particular data item in a mutually exclusive manner. While one transaction is accessing a data item, no other transaction must be allowed to modify that data item. Allow a transaction to perform an operation on a data item only if the transaction holds a lock on that data item.

Locking and Locks Locking is a preventive procedure intended to control concurrent database operations. When one transaction performs database operations on a data item, a lock prevents another transaction from performing database operations on that data item. Locking is meant to ensure database consistency and correctness. Unlocking is the function of releasing a lock on a data item so that other transactions may now perform database operations on that data item.

A lock is simply a variable associated with a data item. This variable indicates the status of a data item as to whether or not it is open for any operation to be performed on it. A binary lock constitutes a simple locking mechanism. It can have two values representing the locked or unlocked states of a data item. For example, if the value of the lock variable for a data item is 1, the data item is locked and may not be accessed; if the value is 0, the data item is unlocked and available for access. A binary lock enforces mutual exclusion of transactions. In the database environment, the lock manager component of the DBMS keeps track of locks and exercises control over data items.

Locking Schemes Binary locking is a straightforward locking scheme. The locking rules are simple: At most only one transaction can hold a lock on a particular data item; if a transaction holds a binary lock on a data item, it has exclusive control over the data item until the lock is released. But a transaction-processing database environment needs more sophisticated locking schemes. If transaction T1 just needs to read a data item, it does not have to lock the data item from another transaction T2 that also just wants to read the data item. Let us explore the locking schemes commonly available in database management systems.

Two basic types of locks are available:

Exclusive lock. Also referred to as a write lock and indicated as an X-lock, this locking mechanism provides exclusive control over the data item. When a transaction holds an X-lock on a data item, no other transaction can obtain a lock of any type on the data item until the first transaction releases the X-lock.

Shared lock. Also referred to as a read lock and indicated as an S-lock, this lock type allows other transactions to share the data item for access. While a transaction T1 holds an S-lock on a data item, another transaction T2 can also acquire an S-lock on that data item, but T2 cannot obtain an X-lock on that data item until all shared locks on the data item are released. When a transaction acquires an S-lock on a data item, it is allowed to read the data item but not to write to that data item.

Let us consider locking and unlocking of a data item D by a transaction T using X- and S-locks. The locking scheme provided by the DBMS works as follows:

1. T must acquire an S-lock(D) or X-lock(D) before any read(D) operation in T.
2. T must acquire an X-lock(D) before any write(D) operation in T.
3. T need not request an S-lock(D) if it already holds an X-lock (D) or an S-lock(D).
4. T need not request an X-lock(D) if it already holds an X-lock (D).
5. T must issue an unlock(D) after all read(D) and write(D) operations are completed in T.

Locking Levels In our discussion on locking, we have mentioned locking a data item. What is a data item? What is the unit of data we need to lock and unlock? Naturally, the granularity size of data a transaction needs to lock depends on the intention and scope of the transaction. DBMSs support multiple levels of granularity for locking. Figure 15-18 shows a hierarchy of locking levels provided for a database system.

When multiple transactions execute concurrently, some may need to lock only at the record level and a few at the field level. Transactions very rarely need to lock at the entire table level in normal transaction processing. The entire database is hardly ever locked by a single transaction. Multiple granularity level protocols are flexible to allow locking at different levels even for transactions in a single schedule.

Lock Management The DBMS must ensure that only serializable and recoverable schedules are permitted while processing concurrent transactions. No operations of committed transactions can be lost while undoing aborted transactions. A

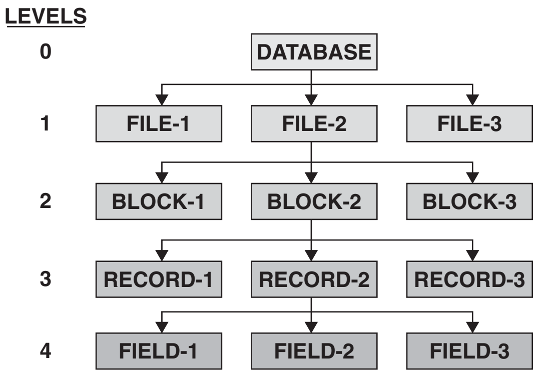

Figure 15-18 Hierarchy of locking levels.

locking protocol is a set of rules to be followed by transactions and enforced by the DBMS to enable interleaving of database operations.

The lock manager component of the DBMS keeps track of all active locks by maintaining a lock table. Usually, a lock table is a hash table with the identifier of the data item as the key. The DBMS keeps a descriptive record for each active transaction in a transaction table, and this record in the transaction table contains a pointer to the list of locks currently held by the transaction. Generally, the lock table record for a data item contains the following: the number of transactions currently holding a lock on the data item (more than one possible in shared mode), the nature of the lock (S or X), and a pointer to a queue of lock requests for the data item. The lock manager responds to lock and unlock requests by using the lock table records and the queues of lock requests.

Let us briefly trace how the lock manager handles lock and unlock requests. Assume a transaction T making lock and unlock requests on a data item D.

- If T requests an S-lock(D), the queue of lock requests is empty, and D is not currently in X-mode, the lock manager grants S-lock(D) and updates the lock table record for D.
- If T requests an X-lock(D) and no transaction currently holds a lock on D indicated by an empty queue of lock requests, the lock manager grants X-lock(D) and updates the lock table record for D.
- Otherwise, the lock manager does not grant the request immediately and adds the request to the queue for D (T is suspended.) When a transaction that was holding D aborts or commits, it releases all locks. When a lock is released, the lock manager updates the lock table record for D and inspects the lock request at the head of the queue for D. If the lock request at the head of the queue can be granted, the lock manager wakes up the transaction (T) at the head of the queue and grants the lock.

When a transaction requests a lock on a data item D in a particular mode and no other transaction has a lock on the same data item in a conflicting mode, the lock manager grants the lock. Now, follow the sequence of a set of lock requests on data item D and observe the predicament of one of the transactions:

T1 has S-lock(D)—T2 requests X-lock(D)—T2 waits—T3 requests S-lock(D)—T3 acquires S-lock(D)—T2 still waits—T4 requests S-lock(D)—T4 acquires S-lock(D)—T2 still waits

In this situation, we say that T2 is in starvation mode, unable to acquire a lock and proceed with its operations. Starvation of transactions can be avoided if lock requests are handled with the following rules:

- When a transaction T requests a lock on a data item D in a particular mode, grant the lock provided
  - There are no other transactions holding a lock on D in a mode that conflicts with the mode of the lock request by T, and
  - There is no other transaction Tn waiting for a lock on D, where Tn made its lock request earlier than T.

Application of Lock-Based Techniques

Lock-based concurrency control techniques depend on proper management and control of locking and unlocking of data items. Before a transaction attempts to perform a database operation, the transaction must request a lock on the data item. A management facility in the DBMS must grant the lock if the data item is not already locked by another transaction. Only after receiving a lock on a data item, may a transaction proceed to perform a database operation. After completion of the database operation, the lock on the data item must be released.

Simple Locking Let us now turn our attention to the application of locking and unlocking in a few examples and appreciate the significance of concurrency control protocols. We will begin with a simple binary locking scheme. Revisit the example of concurrent transactions for accessing inventory record of database textbooks. Figure 15-19 illustrates a simple locking protocol for two concurrent transaction accessing inventory record of database textbooks.

Let us refer to the inventory record accessed as data item D. In the simple locking scheme shown in the figure, every transaction T must obey the following plain set of rules:

1. T must issue lock(D) before any read(D) or write(D) in T.
2. T must issue unlock(D) after all read(D) and write(D) are completed in T.
3. T will not issue unlock(D) unless it already holds a lock on D.

The lock manager will enforce these rules. If you look at the figure closely, you note that a transaction T holds the lock on D between the initial locking and the final unlocking. In this simple concurrency control protocol, only one transaction can access D during any period. Two transactions cannot access D concurrently. In practice, this protocol is too restrictive. What if two concurrent transactions just want to read D? Even then, this protocol will delay one transaction until the other completes. Therefore, we need to employ both S-locks and X-locks. With S-locks, you can allow multiple transactions with only read operations to proceed concurrently.

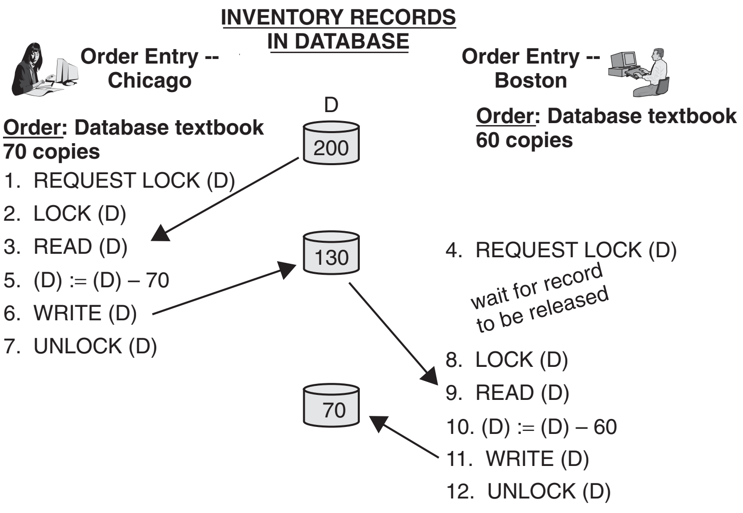

Figure 15-19 Example of simple locking with binary locks.

Initial Values
(D1) 100, (D2) 150

Operations in T1 and T2

T1	T2
LOCK-S (D2)	LOCK-S (D1)
READ (D2)	READ (D1)
UNLOCK (D2)	UNLOCK (D1)
LOCK-X (D1)	LOCK-X (D2)
READ (D1)	READ (D2)
(D1) := (D1) + (D2)	(D2) := (D2) + (D1)
WRITE (D1)	WRITE (D2)
UNLOCK (D1)	UNLOCK (D2)

Serial Schedule: T1, T2

T1	T2
LOCK-S (D2)	
READ (D2)	
UNLOCK (D2)	
LOCK-X (D1)	
READ (D1)	
(D1) := (D1) + (D2)	
WRITE (D1)	
UNLOCK (D1)	
	LOCK-S (D1)
	READ (D1)
	UNLOCK (D1)
	LOCK-X (D2)
	READ (D2)
	(D2) := (D2) + (D1)
	WRITE (D2)
	UNLOCK (D2)

Final Values
(D1) 250, (D2) 400

Interleaved Schedule with S and X Locks

T1	T2
LOCK-S (D2)	
READ (D2)	
UNLOCK (D2)	
	LOCK-S (D1)
	READ (D1)
	UNLOCK (D1)
	LOCK-X (D2)
	READ (D2)
	(D2) := (D2) + (D1)
	WRITE (D2)
	UNLOCK (D2)
LOCK-X (D1)	
READ (D1)	
(D1) := (D1) + (D2)	
WRITE (D1)	
UNLOCK (D1)	

Final Values
(D1) 250, (D2) 250

Serial Schedule: T2, T1

Final Values
(D1) 350, (D2) 250

Figure 15-20 Example of locking with S- and X-locks.

Locking with S- and X-Locks Let us consider another example, already mentioned in earlier sections, about two transactions accessing the inventory records of database and data warehousing textbooks. In this example, each of the two transactions makes adjustments to inventory numbers. Figure 15-20 presents the application of S- and X-locks to this example.

The first part of the figure lists the operations of the two transactions T1 and T2. The inventory records of database and data warehousing textbooks are indicated as D1 and D2 in the illustration. Note the initial values for D1 and D2. The second part of the figures shows a serial schedule with T1 followed by T2. Note the final values for D1 and D2 produced by this serial schedule. The other serial schedule with T2 followed by T1 produces a different set of values for D1 and D2. The last part of the figure shows an interleaved execution of T1 and T2 with S- and X-locks. Note the set of values for D1 and D2 produced by this interleaved schedule.

Review the execution of T1 and T2 as shown in the figure. Locking and unlocking as shown in the figure follow the rules stipulated earlier about S- and X-locks. Nevertheless, the execution of T1 and T2 as illustrated in the figure does not produce the correct final results. You can easily see that the schedule consisting of T1 and T2 as presented in the figure is nonserializable. Binary locks or S- and X-locks, just by themselves, do not guarantee serializability. We need to adopt a specific set of rules or protocol to ensure serializability. We need to adopt a two-phase locking protocol to obtain the correct final results.

Two-Phase Locking (2PL) Two-phase locking guarantees serializability. This protocol is based on the assumption of all transactions being made to obey the following rules:

1. Before performing any operation on any database object the transaction must first acquire a lock on that object.
2. After releasing a lock, the transaction never acquires any more locks.

Adherence to these rules ensures that all interleaved executions of those transactions are serializable. In the two-phase locking protocol, all locking operations precede the first unlock operation in a transaction. Once a transaction issues an unlock, it cannot obtain any more locks.

According to the concepts of this protocol, every transaction may be divided into two phases: first a growing or expanding phase and then a second shrinking phase.

- In the growing phase, the transaction acquires all the necessary locks but does not release any of them. Each lock may be an S- or X-lock, depending on the type of database access. If upgrading of locks is permitted, upgrading can take place only in this phase. (Upgrading means changing an S-lock to an X-lock.)
- In the shrinking phase, the transaction releases all the locks. Once the transaction releases the first lock, it cannot acquire any new lock. If downgrading of locks is allowed, downgrading can take place only in this phase. (Downgrading means changing an X-lock to an S-lock.)

Now, let us get back to our example of concurrent transactions accessing inventory records of database and data warehouse textbooks. Figure 15-21 presents the operations in T1 and T2 rearranged so that each transaction follows the two-phase locking protocol. Any schedule with T1 and T2 will be serializable. When every transaction in a schedule follows 2PL, the schedule is serializable. 2PL enforces seri-

T1 and T2 following 2PL

T1	T2
LOCK-S (D2)	LOCK-S (D1)
READ (D2)	READ (D1)
LOCK-X (D1)	LOCK-X (D2)
UNLOCK (D2)	UNLOCK (D1)
READ (D1)	READ (D2)
(D1) := (D1) + (D2)	(D2) := (D2) + (D1)
WRITE (D1)	WRITE (D2)
UNLOCK (D1)	UNLOCK (D2)

Figure 15-21 Transactions following 2PL.

TIME	T1 -- Chicago Order:	T2 -- Boston Order:	DB
	DB 70 copies	DB 60 copies	
t1	BEGIN T1		200
t2	LOCK-X (DB)	BEGIN T2	200
t3	READ (DB)	LOCK-X (DB)	200
t4	(DB) := (DB) – 70	…. wait ….	200
t5	WRITE (DB)	…. wait ….	130
t6	UNLOCK (DB)	…. wait ….	130
t7	COMMIT	READ (DB)	130
t8	END	(DB) := (DB) – 60	130
t9		WRITE (DB)	70
t10		UNLOCK (DB)	70
t11		COMMIT	70
t12		END	70

Figure 15-22 Two-phase locking: resolution of lost update problem.

alizability. Carefully inspect the lock and unlock operations in T1 and T2 to ensure that each of these transactions follows 2PL.

Before we proceed further, go back and review Figures 15-9, 15-10, and 15-11, which presented the three major problems of concurrent transactions—lost update, uncommitted dependency, and inconsistent summary. You can address these three problems by applying the two-phase locking protocol. Figure 15-22 illustrates how 2PL resolves the lost update problem presented earlier in Figure 15-9. 2PL also resolves the other two problems of uncommitted dependency and inconsistent summary.

Deadlock: Prevention and Detection

Review lock-based concurrency control protocols including 2PL. Although 2PL ensures serializability, it is still subject to a different problem. Refer back to Figure 15-21 showing the two transactions T1 and T2, each of which follows 2PL. In the first phase, each transaction acquires all the locks it needs and performs its database operations. While each transaction is acquiring the necessary locks, it is possible that the first transaction may have to wait for a lock held by a second transaction to be released and vice versa. Refer to Figure 15-23 portraying a situation in which transactions T1 and T2 are deadlocked although they attempt to conform to 2PL.

TIME	T1 -- Chicago Order:	T2 -- Boston Order:	DB	DW
	DB 70 copies	DW 45 copies		
	DW 65 copies	DB 60 copies		
t1	BEGIN T1		200	150
t2	LOCK-X (DB)	BEGIN T2	200	150
t3	READ (DB)	LOCK-X (DW)	200	150
t4	(DB) := (DB) – 70	READ (DW)	200	150
t5	WRITE (DB)	(DW) := (DW) – 45	130	150
t6	LOCK-X (DW)	WRITE (DW)	130	105
t7	…. wait ….	LOCK-X (DB)	130	105
t8	…. wait ….	…. wait ….	130	105
t9	…. wait ….	…. wait ….	130	105
t10		…. wait ….	130	105

Figure 15-23 Transactions in deadlock.

A cycle of transactions waiting for locks to be released is known as a deadlock. Two or more deadlocked transactions are in a perpetual wait state. 2PL or any other lock-based protocol, by itself, cannot avoid deadlock problems. In fact, 2PL is prone to cause deadlocks whenever the first phases of two or more transactions attempt to execute simultaneously.

First, let us look at two obvious solutions to resolve deadlocks:

Deadlock prevention protocol. Modify 2PL to include deadlock prevention. In the growing phase of 2PL, lock all required data items in advance. If any data item is not available for locking, lock none of the data items. Either lock everyone of the needed data items or none at all. If locking is not possible, the transaction waits and tries again. The transaction will be able to lock all data items as and when all become available. The solution, although workable, limits concurrency a great deal.

Ordering of data items. Order all lockable data items in the database and ensure that when a transaction needs to lock several data items, it locks them in the given order. This is not a solution that can be implemented through the DBMS. The programmers need to know the published order of lockable items. It is not a practical solution at all.

In practice, these two solutions are not feasible. One solution seriously limits the advantages of concurrent transaction processing. The other is impractical and too hard to implement. We have to explore other possibilities. Ask a few questions about your current or proposed database environment. How frequently do you expect concurrent transactions to process the same data item simultaneously? What are the chances? How large is your transaction volume? What kind of overhead can your environment tolerate for resolving deadlocks?

If deadlocks are expected to be very rare, then it is unwise to take a sophisticated approach to prevent deadlocks. You might do well to have a method for detecting the rare deadlocks and deal with them as and when they occur. Deadlock detection and deadlock prevention are commonly used schemes for resolving deadlocks. Deadlock detection has wider acceptance and appeal.

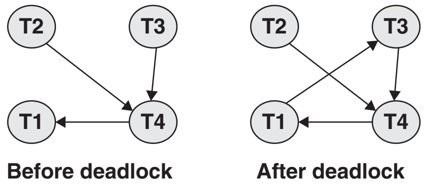

Figure 15-24 Wait-for graph: before and after deadlock.

Deadlock Detection If transactions are generally short and each transaction usually locks only a few data items, you can assume that deadlocks are quite infrequent in such environments. Then you can let deadlocks happen, detect them, and take action. What is the resolution when a deadlock is detected? On the basis of a chosen set of criteria, the system is allowed to abort one of the contending transactions. Of course, before the DBMS can take this action, it must first be able to detect that two or more transactions are in a deadlock.

Here are the common methods for detecting deadlocks:

Wait-for graph. The DBMS maintains a wait-for graph with a node for each currently executing transaction. If transaction T1 is waiting to lock data item D that is currently locked by T2, draw a directed line from node T1 to node T2. When T2 releases the locks on the data items T1 is waiting for, drop the line from T1 to T2 in the graph. Figure 15-24 shows a sample wait-for graph before and after deadlock. When the wait-for graph has a cycle, the DBMS detects a deadlock. Two major issues arise with this deadlock detection approach. First, how often should the DBMS check the wait-for graph? Common methods are based on the number of currently executing transactions or based on the waiting times of several transactions. The second issue relates to choosing the transactions for aborting. Tip: Avoid aborting long-running transactions or transactions that have already performed many write operations.

Timeouts. The DBMS assumes a deadlock if a transaction is waiting too long for a lock and aborts the transaction. A predefined period is used as the waiting time after which a transaction is subject to be aborted. This is a simpler scheme with very little overhead.

Deadlock Prevention If most of the transactions in a database environment run long and update many data items, deliberately allowing deadlocks and aborting the long-running transactions frequently may not be the acceptable approach. In such environments, you must adopt deadlock prevention schemes and avoid deadlocks.

Deadlock prevention or avoidance schemes are based on the answer to the question of what should be done to a transaction proceeding toward a possible deadlock. Should the transaction be put on hold and made to wait, or should it be aborted? Should the transaction preempt and abort another possible contending transaction?

Each transaction is given a logical timestamp, a unique identifier assigned to each transaction in chronological sequence of the start times of transactions. If transaction T1 starts earlier than transaction T2, Timestamp (T1) < Timestamp (T2). The earlier transaction has the smaller value of the timestamp.

Two schemes are available for deadlock prevention, both using transaction timestamps for taking appropriate action. Assume that transaction T1 is attempting to lock data item D that is locked by transaction T2 with a conflicting lock.

Wait-die. (1) T1 is older than T2. Timestamp (T1) < Timestamp (T2). T1 is permitted to wait. T1 waits. (2) T1 is younger than T2. Timestamp (T1) > Timestamp (T2). Abort T1 and restart it later with the same timestamp. T1 dies.

Wound-wait. (1) T1 is older than T2. Timestamp (T1) < Timestamp (T2). Abort T2 and restart it later with the same timestamp. T1 wounds T2. (2) T1 is younger than T2. Timestamp (T1) > Timestamp (T2). Allow T1 to wait. T1 waits.

Both schemes end up aborting the younger transaction. It can be easily shown that both techniques are deadlock-free. However, both schemes may abort some younger transactions and restart them unnecessarily even though these younger transactions may never cause a deadlock.

In wait-die, the older transaction waits on the younger transaction. A younger transaction requesting a lock on a data item locked by the older transaction is aborted and restarted. Because transactions only wait on younger transactions, no cycle is generated on the wait-for graph. Therefore, this scheme prevents deadlock.

In wound-wait, the younger transaction is allowed to wait on the older one. An older transaction requesting a lock on a data item locked by the younger transaction preempts the younger transaction and aborts it. Because transactions only wait on older transactions, no cycle is generated on the wait-for graph. Therefore, this scheme prevents deadlock.

Timestamp-Based Resolution

From our discussion on lock-based concurrency control, you have perceived that use of locks on the basis of 2PL ensures serializability of schedules. Locking of data items according to 2PL determines the order in which transactions are made to execute. A transaction waits if the data item it needs is already locked by another transaction. The order of execution of transactions is implicitly determined by such waits for locks to be released. Essentially, the ordering of transaction execution guarantees serializability.

There is another method in use to produce the same effect. This method does not use locks. Right away, because of the absence of locks, the problem of deadlocks does not occur. This method orders transaction execution by means of timestamps. Lock-based protocols prevent conflicts by forcing transactions to wait for locks to be released by other transactions. However, transactions do not have to wait in the timestamp method. When a transaction runs into a conflict, it is aborted and restarted. The timestamp-ordering protocol also guarantees serializability.

How are timestamps used to determine the serializability order? If Timestamp(T1) < Timestamp(T2), then the DBMS must ensure that the schedule produced by timestamp ordering is equivalent to a serial schedule where T1 precedes T2.

Timestamps The DBMS assigns a unique identifier to each transaction as soon as it is submitted for execution. This identifier is used as the timestamp for that transaction to be used for ordering transaction executions. If transaction T2 is submitted after transaction T1, Timestamp(T1) < Timestamp(T2).

DBMSs use two common methods for assigning timestamps:

- Use a logical counter that is incremented each time its value is assigned to a transaction when the transaction arrives at the system. The logical counter is reset periodically to an initial value.
- Use the current date and time value of the system while making sure that no two transactions are given the same timestamp value.

Besides timestamps for transactions, timestamp ordering also needs timestamps for data items. Two timestamp values are associated with a data item D as indicated below:

Read-timestamp(D)—	Indicates the timestamp of the largest timestamp of the transactions that successfully read D, that is, the timestamp of the youngest transaction that read D successfully.
Write-timestamp(D)—	Indicates the timestamp of the largest timestamp of the transactions that successfully wrote D, that is, the timestamp of the youngest transaction that wrote D successfully.

Read- and write-timestamps for data item D are updated whenever newer transactions read or write D, respectively.

Basic Timestamp Ordering Recall transaction executions using 2PL. The protocol makes a schedule serializable by making it equivalent to some serial schedule. The equivalent serial schedule is not necessarily representative of the order in which transactions entered the system. However, in the case of timestamp ordering, the equivalent schedule preserves the specific order of the transactions.

The timestamp ordering protocol examines each read and write operation to ensure that they are executed in strict timestamp order of the transactions. Here is how the protocol works for a transaction T in relation to a data item D:

(1) T issues read(D)

(a) Timestamp(T) < Write-timestamp(D)

T asks to read D that has already been updated by a younger or later transaction. This means that an earlier transaction T is asking to read D that has updated by a later transaction. T is too late to read the previous outdated value. Values of any other data items acquired by T are likely to be incompatible with the value of D.

Reject read operation. Abort T and restart it with a new timestamp.

(b) Timestamp(T) > = Write-timestamp(D)

T asks to read D that has been updated not by a later transaction.

Execute read operation. Set Read-timestamp(D) to the larger of Read-timestamp(D) and Timestamp(T).

(2) T issues write(D)
   (a) Timestamp(T) < Read-timestamp(D)
      T asks to write D that has already been read by a younger or later transaction. This means that a later transaction is already using the value of D and it would be incorrect to update D now. T is too late to attempt to write D.
      Reject write operation. Abort T and restart it with a new timestamp.
   (b) Timestamp(T) > Write-timestamp(D)
      T asks to write D that has been updated by a younger or later transaction. This means that T is attempting to override the value of D with an obsolete value.
      Reject write operation. Abort T and restart it with a new timestamp.
   (c) Otherwise, when (a) and (b) are not true
      Execute write operation. Set Write-timestamp(D) to Timestamp(T).

To illustrate the basic timestamp-ordering protocol, let us consider two transactions T1 and T2 concurrently accessing savings and checking account records. T1 wants to read the two records, add the balances, and display the total. T2 intends to transfer $1,000 from a savings account to a checking account. Assume that T2 is submitted after T1 so that Timestamp(T1) < Timestamp(T2). Figure 15-25 presents the functioning of the timestamp-ordering protocol.

The basic timestamp-ordering protocol ensures conflict serializability. It produces results that are equivalent to a serial schedule where transactions are executed in their chronological order. In other words, the executions of operations would be as though all operations of one transaction are executed first, followed by all operations of another transaction that are executed next, and so on, with no interleaving of operations. A modification of the basic timestamp-ordering protocol affords greater potential for concurrency. We discuss this in the next subsection.

Thomas's Write Rule Consider concurrent transactions T1 and T2 accessing data item D. Assume that T1 starts earlier than T2 so that Timestamp(T1) < Time-

Transaction T1

T1
READ (SAV)
READ (CHK)
SUM := (CHK) + (SAV)
DISPLAY SUM

Transaction T2

T2
READ (SAV)
(SAV) := (SAV) – 1000
WRITE (SAV)
READ (CHK)
(CHK) := (CHK) + 1000
WRITE (CHK)
SUM := (CHK) + (SAV)
DISPLAY SUM

Execution under timestamp ordering protocol

TIME	T1	T2	CHK	SAV	SUM
t1	READ (SAV)		2000	15000	
t2		READ (SAV)	2000	15000	
t3		(SAV) := (SAV) – 1000	2000	15000	
t4		WRITE (SAV)	2000	14000	
t5	READ (CHK)		2000	14000	
t6		READ (CHK)	2000	14000	
t7	SUM := (CHK) + (SAV)				17000
t8	DISPLAY SUM				
t9		(CHK) := (CHK) + 1000	2000	14000	
t10		WRITE (CHK)	3000	14000	
t11		SUM := (CHK) + (SAV)			17000
t12		DISPLAY SUM			

Figure 15-25 Timestamp-ordering protocol.

stamp(T2). The schedule consisting of the operations of T1 and T2 is shown below in time sequence:

```
T1:     Read(D)                    Write(D)
T2:                 Write(D)
          ──────────────────────────────►
```

Let us apply the timestamp-ordering protocol to the schedule:

Read(D) of T1 succeeds.
Write(D) of T2 succeeds.
When T1 attempts to perform Write(D), Timestamp(T1) < Write-timestamp(D) because Write-timestamp(D) was set to Timestamp(T2).
Therefore, Write(D) of T1 is rejected and the transaction must be aborted and restarted with a new timestamp.

Examine the application of the protocol carefully. The timestamp-ordering protocol requires the rejection of T2 although it is unnecessary. See what has happened to D. T2 had already written D; the value T1 wants to write to D will not be required to be read. Any transaction Tn earlier than T2 attempting to read D will be aborted because Timestamp(Tn) < Write-timestamp(D). Any transaction Tn with Timestamp(Tn) > Timestamp(T2) must read D as written by T2 rather than the value of D written by T1. Therefore, the write operation of T1 may be ignored.

What you have reviewed here leads us to a modified version of the timestamp-ordering protocol in which certain obsolete write operations may be ignored. The modified version has no changes for read operation. The modified version for write operation of the timestamp-ordering protocol incorporates Thomas's Write Rule as follows:

T issues write(D)

(a) Timestamp(T) < Read-timestamp(D)
T asks to write D that has already been read by a younger or later transaction. This means that a later transaction is already using the value of D and it would be incorrect to update D now. T is too late to attempt to write D.
Reject write operation. Abort T and restart it with a new timestamp.

(b) Timestamp(T) > Write-timestamp(D)
T asks to write D that has been updated by a younger or later transaction. This means that T is attempting to override the value of D with an obsolete value.
Ignore write operation.

(c) Otherwise, when (a) and (b) are not true
Execute write operation. Set Write-timestamp(D) to Timestamp(T).

Optimistic Techniques

Review the types of processing done while using lock-based or timestamp-ordering protocols. Before each database operation gets performed, some verification has to be done to ensure that the operation will not cause a conflict. This results in system overhead during transaction execution through locking or time-stamping and slows the transaction down. In database environments where conflicts among transactions are rare, such additional overhead is unnecessary. In these environments, we can adopt different techniques based on this optimistic assumption. Because no locking is involved optimistic techniques allow greater concurrency.

In optimistic concurrency control techniques, no checking is done during transaction execution; verification is put off until a validation phase. In this later phase, a check determines whether a conflict has occurred. If so, the transaction is aborted and restarted. Restarting may be time-consuming, but it is assumed that in database environments where conflicts are rare, aborting and restarting will also be sufficiently infrequent. Restarting is still tolerable compared to the overhead of locks and timestamps.

The system keeps information for verification in the validation phase. During execution, all updates are made to local copies. If serializability is not violated, a transaction wishing to commit is allowed to commit and the database is updated from the local copy; otherwise, the transaction is aborted and restarted later. The optimistic protocol we are studying here uses timestamps and also maintains Write-sets and Read-sets for transactions. The Write-set or Read-set for a transaction is the set of data items the transaction writes or reads.

Let us study the three phases of an optimistic concurrency control technique:

Read Phase. Extends from the start of a transaction until just before it is ready to commit. The transaction performs all the necessary read operations, stores values in local variables, and stores values in local copies of data items kept in transaction workspace.

Validation Phase. Extends from the end of the read phase to when the transaction is allowed to commit or is aborted. The system checks to ensure that serializability will not be violated if the transaction is allowed to commit. For a read-only transaction, the system verifies whether data values read are still the current values. If so, the transaction is allowed to commit. If not, the transaction is aborted and restarted. For an update transaction, the system determines whether the transaction will leave the database in a consistent state and ensure serializability. If not, the transaction is aborted and restarted.

Write Phase. Follows a successful verification in the validation phase. Updates are made to the database from the local copies.

The system examines reads and writes of the transactions in the validation phase using the following timestamps for each transaction T:

Start(T)	Start of execution of T
Validation(T)	Beginning of validation phase of T
Finish (T)	End of T, including write phase, if any

For a transaction T to pass the verification in the validation phase, one of the following must be true:

(1) Any transaction Tn with earlier timestamp must have ended before T started, that is, Finish(Tn) < Start(T).

(2) If T starts before an earlier Tn ends, then
  (a) Write-set(Tn) is not the same as Read-set(T), and
  (b) Earlier transaction Tn completes its write phase before T enters its validation phase, that is, Start(T) < Finish(Tn) < Validation(T).

Rule 2 (a) above ensures that values written by an earlier transaction are not read by the current transaction. Rule 2 (b) guarantees that the write operations are performed serially, not causing conflicts.

DATABASE FAILURES AND RECOVERY

So far, we have concentrated on problems emanating from concurrent processing of transactions and how concurrent processing could cause data integrity problems. If concurrent transactions are not processed according to specific concurrency control protocols, they are likely to leave the database in an inconsistent state.

We now turn our attention to data integrity problems that arise because of various types of failures. Recall the definition of a transaction as a unit of work that must be completed successfully to maintain database consistency. You know that in a large database environment a continuous stream of transactions is accessing the database all the time. What happens in such an environment if there is a sudden crash of the disk where the database resides? Just at the moment when disaster strikes, several transactions could be in flight, unable to complete successfully and resulting in serious data inconsistency problems. The effects of the partially executed transactions must be cleaned up, and the database should be restored to a consistent state.

We begin by surveying the types of failures that could cause data integrity problems and examine some ideas on how to recover from these types of failures. Specific recovery techniques address particular types of failures and how to recover from them. We will explore these recovery techniques. Recovering from failures and keeping the database system correct and consistent are primary requirements in a modern organization.

Classification of Failures

A database system is subject to a variety of failures. When a database system fails for one reason or another, the processing of transactions stops and the database is rendered unusable. The action that follows to recover from the interruption and make the database functional depends on the type of failure. The database and operating system software automatically handle some error conditions. Other error conditions need special actions. Depending on the severity of the failure, it may take a while for the recovery process to complete.

Let us list the types of failures and try to classify them. The list describes each type of failure and also indicates the nature of the recovery process.

Transaction error. Malfunction of a transaction caused by exception conditions such as integer overflow, divide by zero, bad input, data not found, resource limit exceeded, and so on. These are usually handled in the application program. The program is coded to detect such conditions and take appropriate action. Most of the time, the transaction aborts and rolls back partial database updates. Usually, these errors are not considered database system failures; these are local exceptions at the transaction level.

System error. In this type of failure, the database system enters an undesirable state such as a deadlock or overflow on some system file or resource. The database system is unable to continue processing. In such cases, the database may or may not be corrupted. The cause for the error must be determined and corrected before restarting the database.

System failure. This includes failure of hardware, data communication links, network components, operating system software, or the DBMS itself. Hardware failure is usually failure of main memory losing volatile storage. The cause for the failure must be ascertained and corrected. The effects of partial database updates of transactions in process at the instant of failure must be resolved before restarting the database system.

Disk failure. This is usually a head crash caused by the disk head crashing into the writing surface of the disk. The extent of the damage must be ascertained, and the disk has to be discarded if the damage is extensive. The database must be restored on another disk from a backup copy and recovered from copies of transactions up to the point of failure. Also, the effects of partial database updates of transactions in process at the instant of failure must be resolved before restarting the database system.

Physical failure. This type comprises a variety of problems including power loss, power surge, fire, disk overwrites, and so on. Full recovery from such a failure depends on the exact nature of the cause.

Human error. You may include a number of problems in this group such as operator error, carelessness, sabotage, and theft. Recovery options vary and are difficult.

Recovery Concepts

One of the major concerns about recovering from failures relates to the transactions in progress at the time of a failure. These in-flight transactions do not have a chance to complete, and the recovery mechanism must address the partial updates to the database by these transactions. Let us consider one such transaction in progress at a time of failure. Figure 15-26 shows the transaction in progress at the instant of failure.

To preserve the consistency of the database, either the effects of the partial updates must be removed or the updates must be completed. How can we achieve this? If the system keeps information about all the intended updates of the transaction, then the system can use that information either to back out the effects of the partial updates or to complete the remaining updates. If the partial updates are backed out, the database is restored to the prior consistent state. On the other hand, if the remaining updates are completed correctly, the database is taken to a new

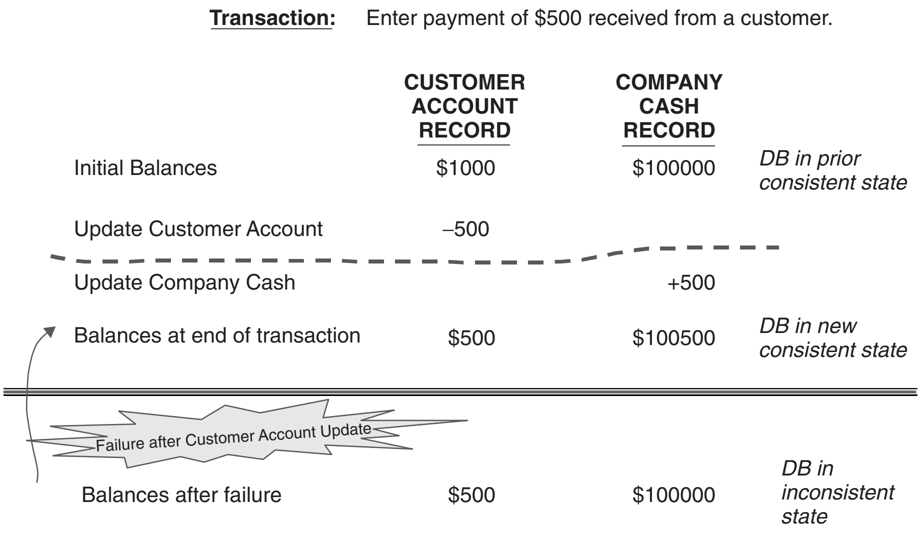

Figure 15-26 Transaction in progress at time of failure.

consistent state. Again, remember that in this example we have considered only one transaction in progress. In practice, many transactions are normally in progress at the time of a failure. The effects of partial database updates by all of the in-flight transactions must be addressed.

The underlying principle for recovery is redundancy. Keep enough redundant data to enable recovery. This includes keeping backup copies of the database and a file of copies of the updates since the last backup. A transaction is the basic unit of recovery in case of failures. The recovery system must be able to recover the effects of transactions in progress—for all of them, one by one. The recovery manager of the DBMS must be able to guarantee the properties of atomicity and durability.

To summarize, a DBMS must provide the following facilities for recovery from failures:

- A backup mechanism to create periodic backup copies of the entire database
- Journalizing or logging facilities to maintain an audit trail of transactions and database changes
- A checkpoint facility for the DBMS to suspend all processing, from time to time, and synchronize its files and journals
- A recovery manager module to bring the database to a consistent state and resume transaction processing

Logging

Please go back to the example shown in Figure 15-26. You note that the failure interrupts the transaction and stops it before it can complete successfully. You quickly realize that you could easily bring the database to a consistent state if only you had information about the updates intended by the transaction. Using the information

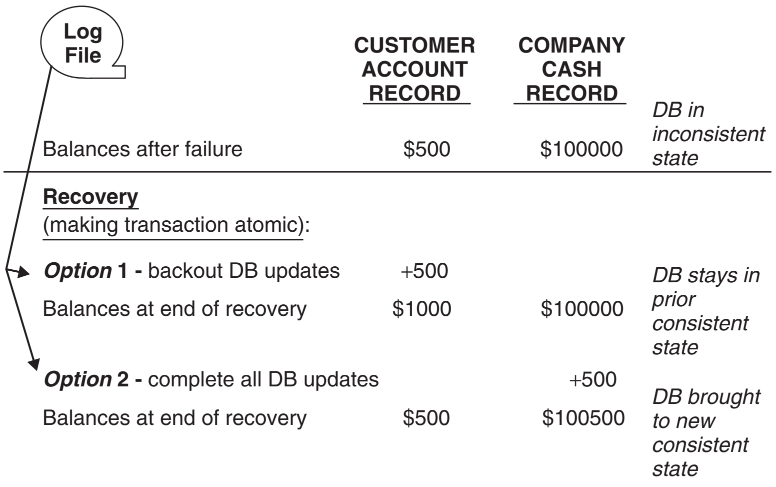
Log File
CUSTOMER ACCOUNT RECORD
COMPANY CASH RECORD
Balances after failure $500 $100000
DB in inconsistent state
Recovery (making transaction atomic):
Option 1 - backout DB updates +500
Balances at end of recovery $1000 $100000
DB stays in prior consistent state
Option 2 - complete all DB updates +500
Balances at end of recovery $500 $100500
DB brought to new consistent state

Figure 15-27 Log file.

about the intended updates, you can do one of two things. You can either back out the effects of partial update and restore the database to its prior consistent state or you can apply the incomplete update and move the database forward to a new consistent state. But where can you get the information about the intended updates? What if you store this information in another file apart from the database itself? The transaction can write information about the update on this file in addition to updating the database. Such a file is called a log, journal, or trail file. Figure 15-27 demonstrates the use of a log file for the transaction shown in Figure 15-26.

Logging is a principal component in the database recovery process. Most database systems adopt log-based techniques for recovery from failures. The log file keeps track of all operations of transactions that change values of data items in a database. The file is kept on secondary storage such as a disk and used for recovery from failures. Initially, the recent log records are kept in main memory and then forced out to secondary storage. By this approach, processing of each transaction need not wait on the I/O to write log records to disk. Periodically, the contents of the log file on disk are transferred to a tape or other archival medium for safekeeping. Let us examine the types of records written on the log file.

Writing Log File Records Every log record receives a unique identifier called the log sequence number to identify the specific transaction. Usually, log sequence identifiers are in increasing order, as required by some recovery techniques. Each record relating to a transaction has this unique identifier. The DBMS writes a log record for each of the following actions:

- Start of a transaction
- Read of a data item (not required in all recovery techniques)
- Update of a data item
- Insertion of a data item
- Deletion of a data item

- Commit
- Abort

Types of Log File Records Recovery requires a few pieces of important information about each transaction and the data items the transaction intends to update. Therefore, a log file keeps different types of records to trace the operations of each transaction. Each record type is identified in the log file. A log record contains a transaction identifier and a record type indicator. Each record is also date and timestamped. Here is a list of the record types. The record types shown are for a transaction T performing operations on data item D.

(T, BEGIN)
Marks the start of T.

(T, D, value)
Indicates the read operation and contains the value of D read from database.

(T, D, old value, new value)
Indicates an update, insertion, or deletion operation. A record indicating an insertion contains the new value (after image field). For update, the old and new values (before and after image fields) are present in the log record. For deletion, the log record has the old value (before image field).

(T, COMMIT)
Marks the commit point of T.

(T, ABORT)
Marks the abort operation for T.

Checkpoint

By now, you have realized that the recovery manager of the DBMS needs to read through the log file, first to identify the transactions that were in progress at the time of a failure and then to use the information from the log records for actually performing the recovery process. The recovery manager has to search through the log records. In a large database environment, there are thousands of transactions daily and many of these perform update operations. So in the event of a recovery, the recovery manager is compelled to go through thousands of log records to redo or undo partial database updates and restore the database to a consistent state. Two difficulties arise:

- The recovery process will be too long because of the time taken for the recovery manager to process thousands of log records.
- On the basis of the recovery technique adopted, the recovery manager is likely to unnecessarily redo many updates to the database that had already taken place. This will further extend the recovery time.

What if the DBMS periodically marks points of synchronization between the database and the log file? Checkpoint records may be written on the log file with date- and timestamps. These points of time indicate that completion of all database updates up to that point are confirmed. In this scenario, the recovery manager is relieved of the task of reading and processing the entire log file; it only needs to process log records from the latest checkpoint found on the log file.

Through the use of the recovery manager you can determine how often to take a checkpoint. While taking a checkpoint, the system has to be paused to perform the necessary actions of synchronization. Too frequent checkpoints, therefore, interrupt transaction processing. On the other hand, if the checkpoints are too far apart, the recovery process is likely to become longer. So a proper balance is necessary. The recovery manager can be made to take a checkpoint at the end of a certain number of minutes or at the end of a given number of transactions.

Actions at a Checkpoint At each checkpoint, the DBMS performs the following actions:

- Temporarily suspend executions of transactions.
- Force-write all modified main memory buffers to the database on secondary storage.
- Write a checkpoint record on the log file. A checkpoint record indicates the transactions that were active at that time.
- Force-write the log entries in main memory to secondary storage.
- Resume transaction executions.

Log-Based Recovery Techniques

Assume that a failure has not destroyed the database but has caused inconsistency problems in the database. Transactions in progress at the time of the failure could not complete their database updates and commit. These partial updates rendered the database inconsistent. In such a failure, the recovery process has to undo the partial updates that have caused inconsistency. Sometimes, it is also necessary to redo some updates to ensure that the changes have been completed in the database. The recovery manager utilizes the before and after images of the update operations as found in the log to restore the database to a consistent state.

Two common methods available for log-based recovery are referred to as the deferred updates technique and the immediate updates technique. Both techniques depend on the log records for recovery. They differ in the way updates are actually made to the database in secondary storage.

Deferred Updates Technique In this technique, a transaction does not update the database until after the transaction commits. This means that the transaction defers or postpones updating the database until after all log entries are completed. The writing of log records precedes any database update.

Consider a transaction T, performing database operations, and the DBMS writing log records for the transaction. Let us trace the actions of T.

1. When T starts, write a (T, BEGIN) record on the log file.
2. When T issues a write operation to data item D, write a (T, D, old value, new value) record on the log file. For the deferred updates technique, the old value

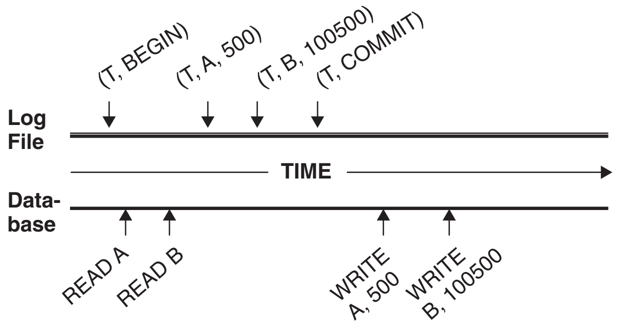

Figure 15-28 Logging with deferred updates.

or before image is not necessary. Do not write to the data item D in the database memory buffers or to the database itself yet.

3. When the transaction is about to commit, write (T, COMMIT) record on the log file. Write all log records for the transaction to secondary storage and then commit the transaction.
4. Perform actual updates to the database from the log records.
5. If T aborts, write (T, ABORT) record on the log file. Do not perform any writes to the database. (Recovery process will ignore all log records.)

Figure 15-28 is a simple illustration of the deferred updates technique for transaction T updating two data items, A and B. Note the chronological order of the log records and database updates.

If a failure occurs, the recovery system scrutinizes the log file to identify the transactions in progress at the time of failure. Beginning from the last record in the log file, the recovery system goes back to the most recent checkpoint record. The recovery system needs to examine only the log records in the interval from that checkpoint and to take action to restore the database to a consistent state.

Assume that T is possibly one of the transactions in progress. So the log records for T will be on the log file in the interval from the most recent checkpoint to the end of the file. The log records for T will be among the records the recovery system will examine for recovery. It will take action based on the log records for T using the logic indicated below:

CASE 1: Both (T,BEGIN) and (T,COMMIT) present on the log.
Failure occurred after all log records were written.
Cannot be sure whether all database updates were completed.
However, the log file contains all records for T.
So the log records for T can be used to update the database.
USE LOG RECORDS TO UPDATE DATABASE.
DATABASE BROUGHT TO NEW CONSISTENT STATE.

CASE 2: (T,BEGIN) present but (T,COMMIT) not present on the log.
Failure occurred before any database updates were made.
No updates were completed on database.

Cannot be sure whether all log records are written.
So, the log records cannot be used to update the database.
IGNORE LOG RECORDS.
DATABASE STAYS IN PRIOR CONSISTENT STATE.

Immediate Updates Technique The deferred updates technique holds off all database updates until a transaction reaches its commit point. The immediate updates technique speeds up the processing by applying database updates as they occur without waiting for the entire transaction to arrive at the commit point. This may make a significant difference in a long transaction with many updates. The writing of log records is interleaved with database updates.

Again, consider a transaction T, performing database operations, and the DBMS writing log records for the transaction. Let us trace the actions of T.

1. When T starts, write a (T, BEGIN) record on the log file.
2. When T issues a write operation to data item D, write a (T, D, old value, new value) record on the log file. For the immediate updates technique, both old and new values or before image and after image are necessary.
3. Write the update to the database. (In practice, the update is made in the database buffers in main memory and the actual update to the database takes place when the buffers are next flushed to secondary storage.)
4. When the transaction commits, write (T,COMMIT) record on the log file.
5. If T aborts, write (T, ABORT) record on the log file. (Recovery process will use log records to undo partial updates.)

Figure 15-29 is a simple illustration of the immediate updates technique for transaction T updating two data items, A and B. Note the chronological order of the log records and database updates.

If a failure occurs, the recovery system scrutinizes the log file to identify the transactions in progress at the time of failure. Beginning from the last record in the log file, the recovery system goes back to the most recent checkpoint record. The recovery system needs to examine only the log records in the interval from that checkpoint and to take action to restore the database to a consistent state.

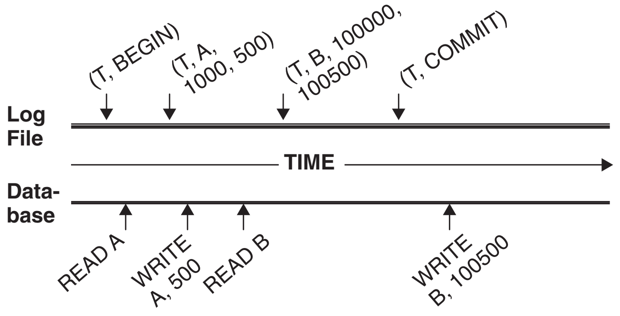

Figure 15-29 Logging with immediate updates.

Assume that T is possibly one of the transactions in progress. So the log records for T will be on the log file in the interval from the most recent checkpoint to the end of the file. The log records for T will be among the records the recovery system will examine for recovery. It will take action based on the log records for T using the logic indicated below:

CASE 1: Both (T,BEGIN) and (T,COMMIT) are present on the log
Failure occurred after all log records were written.
Cannot be sure whether all database updates were completed.
However, the log file contains all records for T.
So the log records for T can be used to update the database.
USE LOG RECORDS TO UPDATE DATABASE WITH AFTER IMAGE VALUES. DATABASE BROUGHT TO NEW CONSIS TENT STATE.

CASE 2: (T,BEGIN) is present but (T,COMMIT) is not present on the log.
Cannot be sure exactly when failure occurred.
Failure could have occurred before updating B or both B and A.
Cannot be sure whether all log records are written.
However, available log records may be used to undo any database updates.
USE AVAILABLE LOG RECORDS TO UNDO DATABASE UPDATES WITH BEFORE-IMAGE VALUES.
DATABASE STAYS IN PRIOR CONSISTENT STATE.

Shadow Paging

The log-based recovery techniques, however effective, come with some substantial overhead. The DBMS must maintain log files. Depending on the circumstances of the failure, the recovery process can be long and difficult. Shadow paging is an alternative technique for database recovery. This technique does not require log files, nor is this recovery process as long and difficult as that of log-based techniques. However, it is not easy to extend shadow paging to an environment with concurrent transaction processing. Still, shadow paging has its use under proper circumstances. We will quickly review the concepts of shadow paging. A detailed discussion is not within our scope here.

The underlying principle in shadow paging is the consideration of the database as a set of fixed-size disk pages or blocks. A directory or page table is created to point to database pages. If the database has *n* pages, the directory has *n* entries. Directory entry number 1 points to the first database page, entry number 30 points to the 30th database page, entry number 75 points to the 75th database page, and so on. Unless the directory is too large, the directory is kept in main memory during transaction processing. All database reads and writes pass through the directory. During the life of a transaction, two directories are maintained, the current page table or current directory and the shadow page table or shadow directory. The current directory points to the current set of database pages.

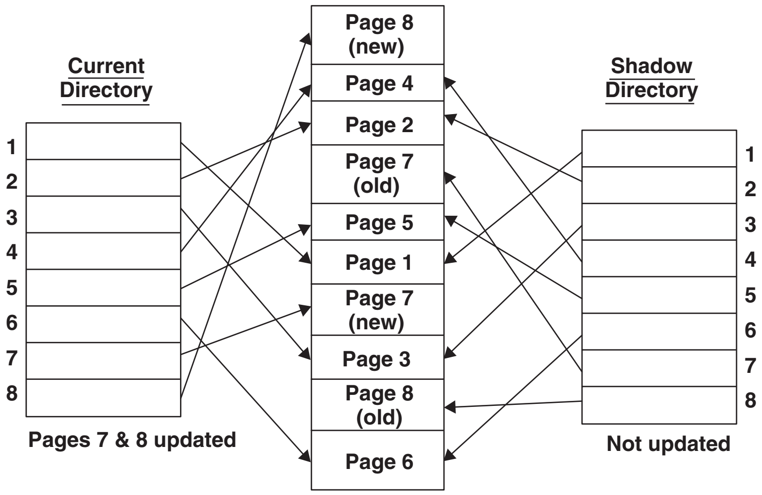

Figure 15-30 Example of shadow paging.

Transaction Execution Let us illustrate the concept of database pages and the two directories. Figure 15-30 shows an example of shadow paging. Refer to the figure and follow the actions during transaction execution.

Here is the list of actions:

Transaction starts execution.
  Current directory is copied to shadow directory.
  During transaction execution shadow directory is not modified.
Transaction performs a write operation (say to page 7 and 8 of the database).
Disk pages (page 7 and 8) are copied to unused disk pages.
  Update is performed in the new disk pages.
  Current directory is modified to point to new disk pages.
  Shadow directory still points to the old disk pages.
To commit the transaction:
  Ensure that all buffer pages in main memory changed by the transaction are written to disk.
  Write current page table to disk.
  Change disk address of the shadow directory to that of the current directory.

Recovery Let us say that a failure occurred during the execution of the above transaction. During execution, the shadow directory still points to the prior status of the database pages. The following actions are taken for recovery:

- Discard the current directory.
- Free the modified pages.
- Reinstate the shadow directory as the current directory.

Although shadow paging is simple and straightforward, and avoids the overhead associated with log-based recovery, it has a few disadvantages as listed below:

- The simple version of shadow paging does not work in database environments with concurrent transaction processing. In such environments, the technique must be augmented by logs and checkpoints.
- The shadow paging technique scatters around the updated database pages and changes page locations. If pages contained related data that must be kept together in a particular database system, then shadow paging is not the proper technique for that system.
- When a transaction commits, the old pages must be released and added to the pool of unused pages. In a volatile environment with numerous updates, constant reorganization becomes necessary to collect the dispersed free blocks and keep them together.
- The process of migration between current and shadow directories itself may be vulnerable to failures.

A Recovery Example

We have reviewed the types of failures that can cause problems in a database environment—some serious enough to damage the entire database and some resulting in database inconsistencies for a few data items. Serious failures take a lot of effort and time to recover and restore the database. We examined a few major techniques for recovery from failures. You have understood the use of logging and how log-based recovery techniques work. You now know the significance of taking checkpoints and the purpose of the checkpoint records on a log file. You have learned the role and importance of database backups.

Let us look at an example of how to recover and restore a database to the latest consistent state after a system crash that has damaged the database contents on disk. You have to begin the recovery from the latest available full backup of the database and do forward recovery. Figure 15-31 presents a comprehensive example of the process.

Observe the timeline in the top half of the figure and note the backup and logging activities shown. Note the points on the timeline when different events take place. The bottom half of the figure indicates how the various log files are used to recover from the crash. Observe how each file is used to bring the status of the database forward up to the point of the crash.

CHAPTER SUMMARY

- Data integrity in a database includes data consistency, correctness, and validity.
- A database system is prone to two kinds of problems: Concurrent transactions may cause data inconsistency, and hardware or software failures may ruin its contents.

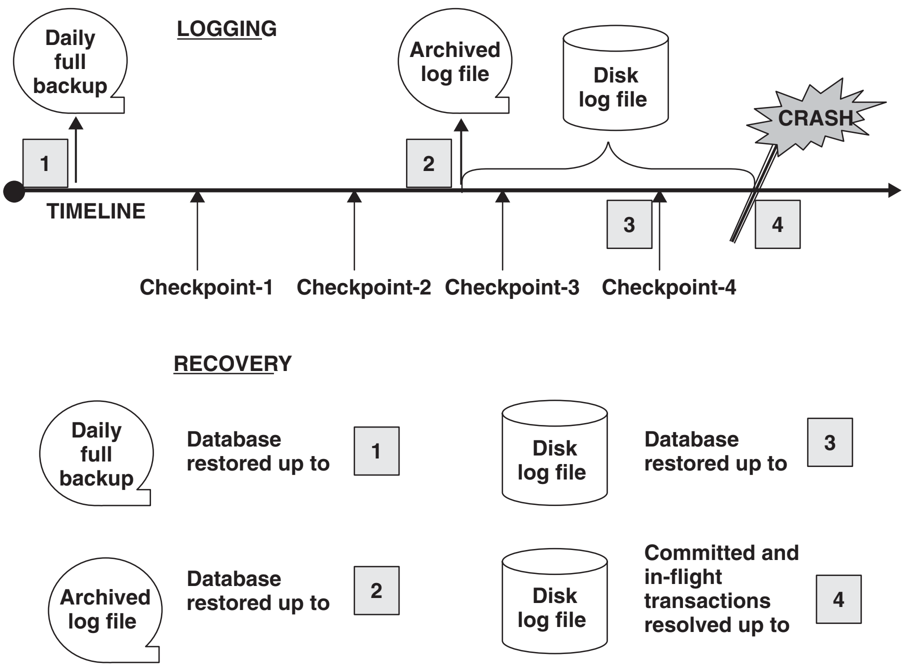

Figure 15-31 A database recovery example.

- A proper transaction must have the properties of atomicity, consistency, isolation, and durability(ACID).
- Operations or actions performed by a transaction: BEGIN, READ, WRITE, END, COMMIT, ABORT, ROLLBACK.
- Major benefits of transaction concurrency: increased system throughput, reduced response times, simultaneous sharing.
- Three primary types of concurrency problems: lost update, dependency on uncommitted update, inconsistent summary.
- A schedule is a set of interleaved database operations of concurrent transactions. A serializable schedule leaves the database in a consistent state.
- Lock-based techniques resolve concurrency problems. A transaction may hold a shared or exclusive lock on a data item.
- The two-phase locking protocol guarantees serializability. However, two-phased locking or any other lock-based scheme cannot avoid deadlock situations.
- Deadlock prevention and deadlock detection are two methods for deadlock resolution.
- The timestamp-ordering protocol also resolves transaction concurrency problems. This scheme does not depend on locks.
- Common types of database failures: system error, system failure, disk failure, physical failure, human error.

- A log file keeps information necessary for recovery from failures. Two log-based recovery techniques: deferred updates technique and immediate updates technique.
- Shadow paging, which does not utilize logging, is a less common method for recovery from database failures.

REVIEW QUESTIONS

1. What are the four essential properties of a database transaction? Describe any one.
2. List the major operations of a transaction.
3. What is domain integrity? State a few domain integrity constraints or rules.
4. Why is concurrent processing of transactions important in a database environment? Describe any one type of concurrency problem.
5. Distinguish between a serial schedule and a serializable schedule. Give an example.
6. What are S-locks and X-locks? Describe their purposes in transaction processing.
7. How is timestamp-based concurrency control different from lock-based techniques?
8. What are the major types of database failures? Briefly describe each.
9. What are the two common log-based database recovery protocols? List their similarities and differences.
10. What are the advantages and disadvantages of shadow paging?

EXERCISES

1. Match the columns:

1. transaction commit	A. no rollback of committed transaction
2. trigger	B. database and log synchronization
3. concurrent transactions	C. used for deadlock prevention
4. serial schedule	D. leaves database in consistent state
5. recoverable schedule	E. used for deadlock detection
6. wound-wait scheme	F. special type of stored procedure
7. wait-for graph	G. desired property of database transaction
8. checkpoint	H. confirm database updates
9. isolation	I. no interleaved operations
10. atomic transaction	J. increased system throughput

2. Define lock-based concurrency control. Why does simple locking not establish serializability? Describe the two-phase locking protocol and show how this technique ensures serializability.

3. As the database administrator for a local bank, which method of deadlock resolution will you adopt? Describe the two common methods and explain the reasons for your choice.
4. What are optimistic concurrency control techniques? In what types of database environments are these techniques applicable?
5. As the database administrator for an international distributor of electronic components, write a general plan for recovery from potential database failures. Describe the type of facilities you will look for in a DBMS for recovery.