Database Management Systems II (DBMS II)

BCSE1-730

Database System Concepts And
Architecture

Topics to be covered

- 1 Data Models
  - 1A. History of data Models
  - 1B. Network Data Model
  - 1C. Hierarchical Data Model
- 2 Schemas versus Instances
- 3 Database Schema vs. Database State
- 4 Three-Schema Architecture
- 5 Data Independence
- 6 DBMS Languages
- 7 DBMS Interfaces
- 8 Database System Environment
- 9 Classification of DBMSs

Data Models

Data Model: A set of concepts to describe the *structure* of a database, and certain *constraints* that the database should obey.

Data Model Operations: Operations for specifying database retrievals and updates by referring to the concepts of the data model. Operations on the data model may include *basic operations* and *user-defined operations*.

Categories of data models:

- **Conceptual** (**high-level**, **semantic**) data models: Provide concepts that are close to the way many users *perceive* data. (Also called **entity-based** or **object-based** data models.)

- **Physical** (**low-level**, **internal**) data models: Provide concepts that describe details of how data is stored in the computer.

- **Implementation** (**representational**) data models: Provide concepts that fall between the above two, balancing user views with some computer storage details.

HISTORY OF DATA MODELS

- Relational Model: proposed in 1970 by E.F. Codd (IBM), first commercial system in 1981-82.
- Network Model: the first one to be implemented by Honeywell in 1964-65 (IDS System). Later implemented in a large variety of systems -DMS 1100 (Unisys), IMAGE (H.P.), VAX -DBMS (Digital Equipment Corp.).
- Hierarchical Data Model: a joint effort by IBM and North American Rockwell around 1965. Resulted in the IMS family of systems. The most popular model.
- Object-oriented Data Model(s): comprises models of persistent O-O Programming Languages such as C++
- Object-Relational Models: Most Recent Trend. Started with Informix Universal Server. Exemplified in the latest versions of Oracle-10i, DB2, and SQL Server etc. systems.

Figure 2.1 Schema diagram for the database of Figure 1.2.

STUDENT

Name	StudentNumber	Class	Major

COURSE

CourseName	CourseNumber	CreditHours	Department

PREREQUISITE

CourseNumber	PrerequisiteNumber

SECTION

SectionIdentifier	CourseNumber	Semester	Year	Instructor

GRADE_REPORT

StudentNumber	SectionIdentifier	Grade

© Addison Wesley Longman, Inc. 2000, Elmasri/Navathe, Fundamentals of Database Systems, Third Edition

HIERARCHICAL MODEL

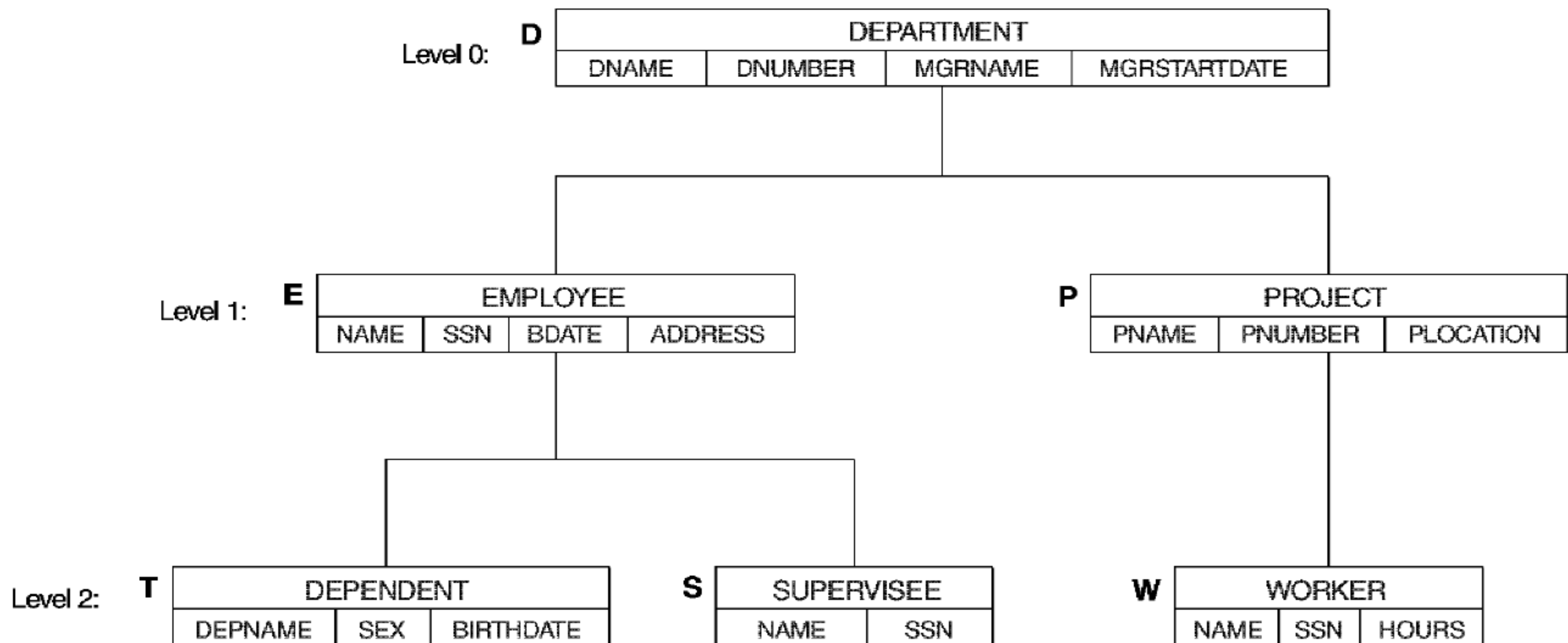
ADVANTAGES:

- Hierarchical Model is simple to construct and operate on
- Corresponds to a number of natural hierarchically organized domains - e.g., assemblies in manufacturing, personnel organization in companies
- Language is simple

DISADVANTAGES:

- Navigational and procedural nature of processing
- Database is visualized as a linear arrangement of records
- Little scope for "query optimization"

Figure D.4 A hierarchical schema for part of the COMPANY database.



© Addison Wesley Longman, Inc. 2000, Elmasri/Navathe, Fundamentals of Database Systems, Third Edition

NETWORK MODEL

ADVANTAGES:

• Network Model is able to model complex relationships and represents semantics of add/delete on the relationships.

• Can handle most situations for modeling using record types and relationship types.

• Language is navigational; uses constructs like FIND, FIND member, FIND owner, FIND NEXT within set, GET etc. Programmers can do optimal navigation through the database.

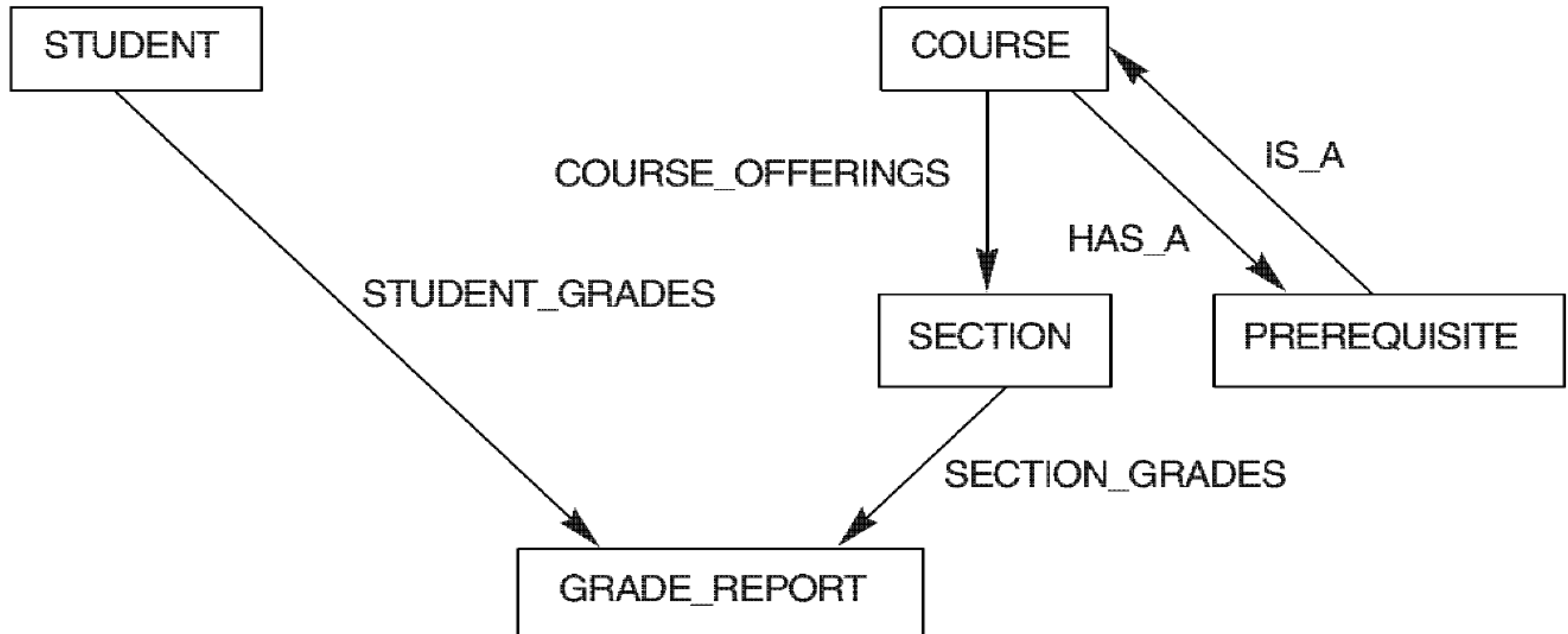
DISADVANTAGES:

• Navigational and procedural nature of processing

• Database contains a complex array of pointers that thread through a set of records.

• Little scope for automated "query optimization"

Figure 2.4 The schema of Figure 2.1 in the notation of the network data model.



© Addison Wesley Longman, Inc. 2000, Elmasri/Navathe, Fundamentals of Database Systems, Third Edition

Transaction Processing
Concepts

Topics to be covered

- Introduction to Transaction Processing
- Transaction and System Concepts
- Desirable Properties of Transactions
- Schedules and Recoverability
- Serializability of Schedules.

Introduction to Transaction Processing

Single-User System: At most one user at a time can use the system.

Multiuser System: Many users can access the system concurrently.

– **Interleaved processing**: concurrent execution of processes is interleave in a single CPU

– **Parallel processing**: processes are concurrently executed in multip CPUs

•**A Transaction:** logical unit of database processing that includes one more access operations (read -retrieval, write - insert or update, delete).

⑩ **A transaction (set of operations)** may be standalone specified in high level language like SQL submitted interactively, or may be embedde within a program.

⑩ **Transaction boundaries**: Begin and End transactio An **application program** may contain several transactions separated by th Begin and End transaction boundaries.

Introduction to Transaction Processing

SIMPLE MODEL OF A DATABASE (for purposes of discussing transactions):

- **A database -** collection of named data items
- **Granularity of data** - a field, a record , or a whole disk block (Concepts are independent of granularity)

•Basic operations are **read** and **write**

– **read_item(X)**: Reads a database item named X into a program variable. To simplify our notation, we assume that *the program variable is also named X.*

– **write_item(X)**: Writes the value of program variable X into the database item named X.

READ AND WRITE OPERATIONS:

•Basic unit of data transfer from the disk to the computer main memory is one block. In general, a data item (what is read or written) will be the field of some record in the database, although it may be a larger unit such as a record or even a whole block.

⑩ **read_item(X) command includes the following steps:**

1. Find the address of the disk block that contains item X.

2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).

3. Copy item X from the buffer to the program variable named X.

READ AND WRITE OPERATIONS (cont.):

- **write_item(X) command includes the following steps:**

1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
3. Copy item X from the program variable named X into its correct location in the buffer.
4. Store the updated block from the buffer back to disk (either immediately or at some later point in time).

FIGURE 17.2

Two sample transactions. (a) Transaction T_1. (b) Transaction T_2.

(a)

T_1
read_item (X);
$X:=X-N$;
write_item (X);
read_item (Y);
$Y:=Y+N$;
write_item (Y);

(b)

T_2
read_item (X);
$X:=X+M$;
write_item (X);

Why Concurrency Control is needed:

•The Lost Update Problem.

This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.

•The Temporary Update (or Dirty Read) Problem.

This occurs when one transaction updates a database item and then the transaction fails for some reason. The updated item is accessed by another transaction before it is changed back to its original value.

•The Incorrect Summary Problem .

If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.

FIGURE 17.3

Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem.

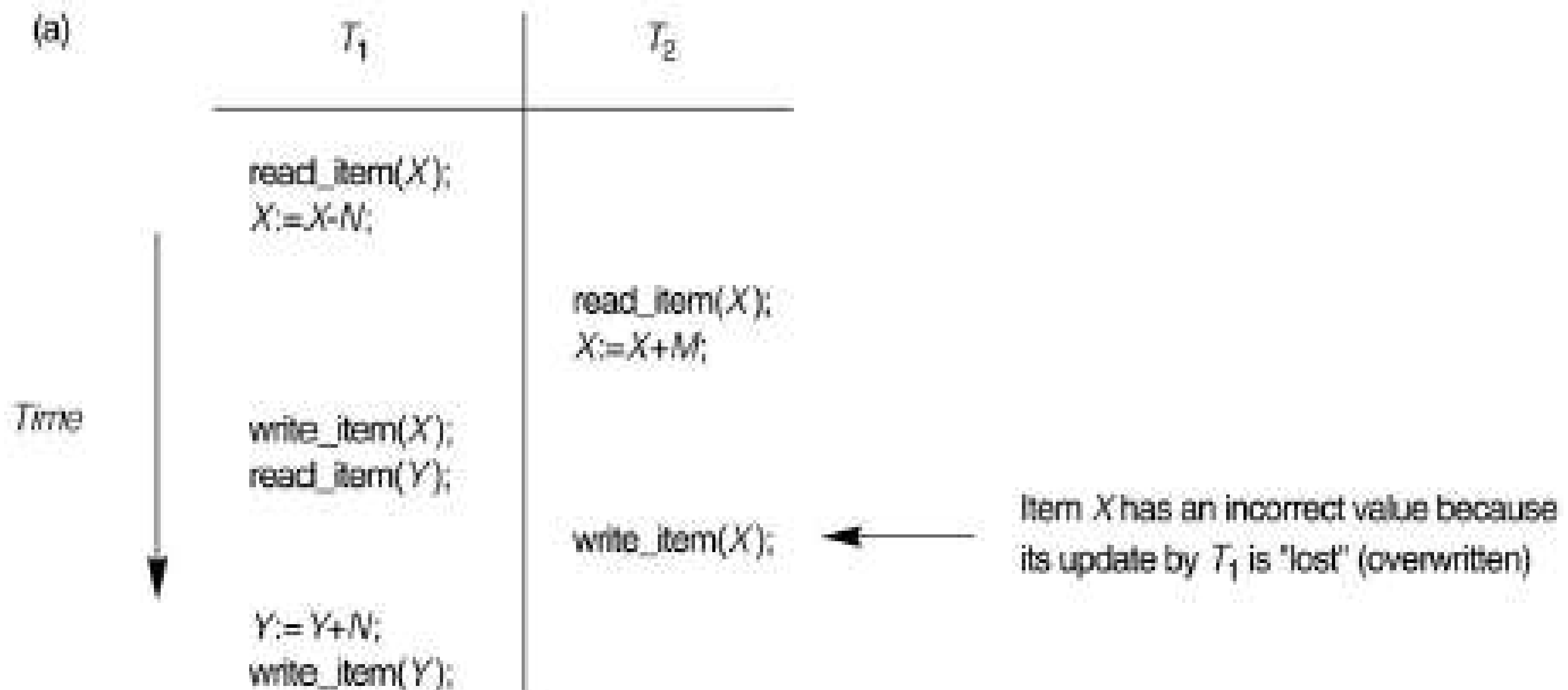


FIGURE 17.3 (continued)

Some problems that occur when concurrent execution is uncontrolled. (b) The temporary update problem.

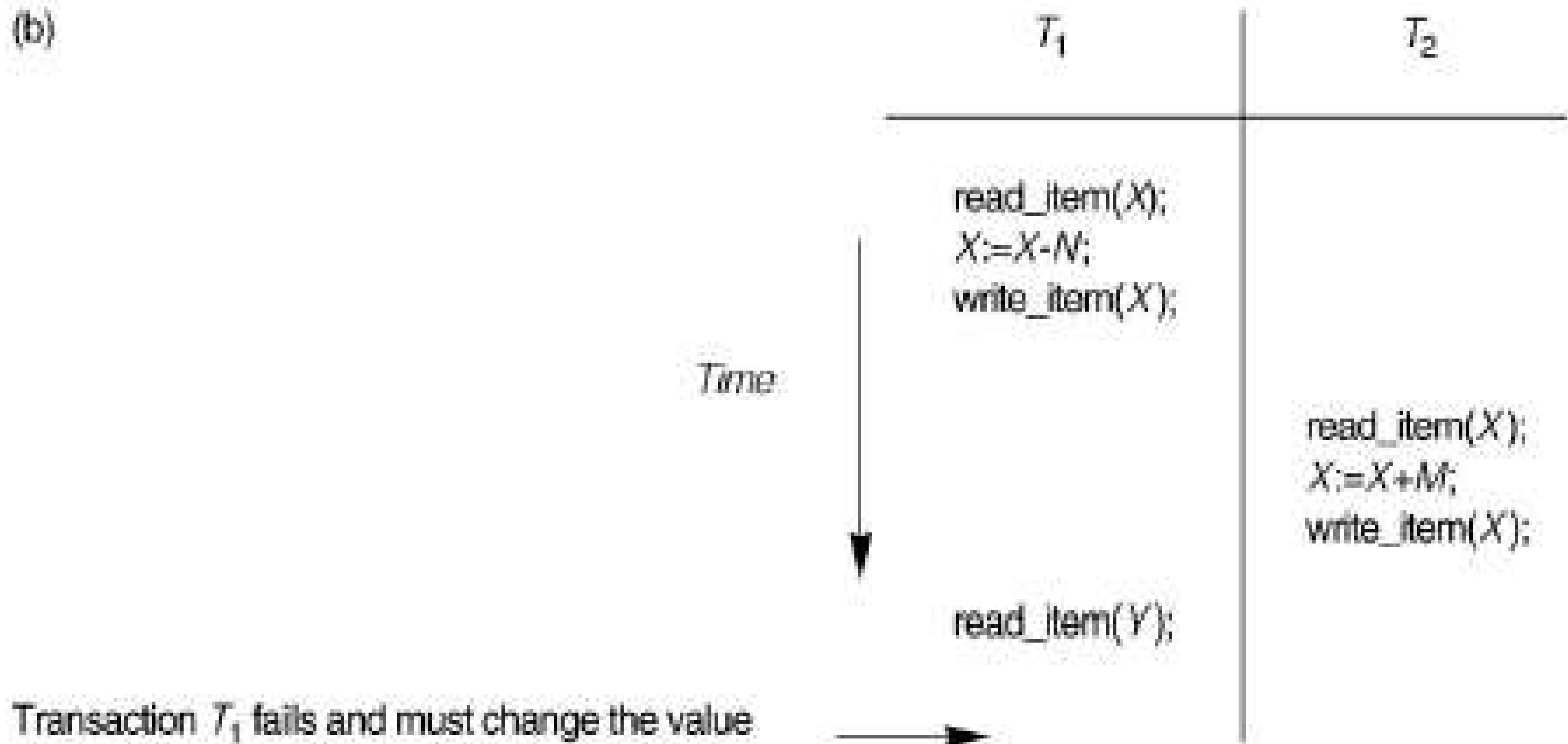
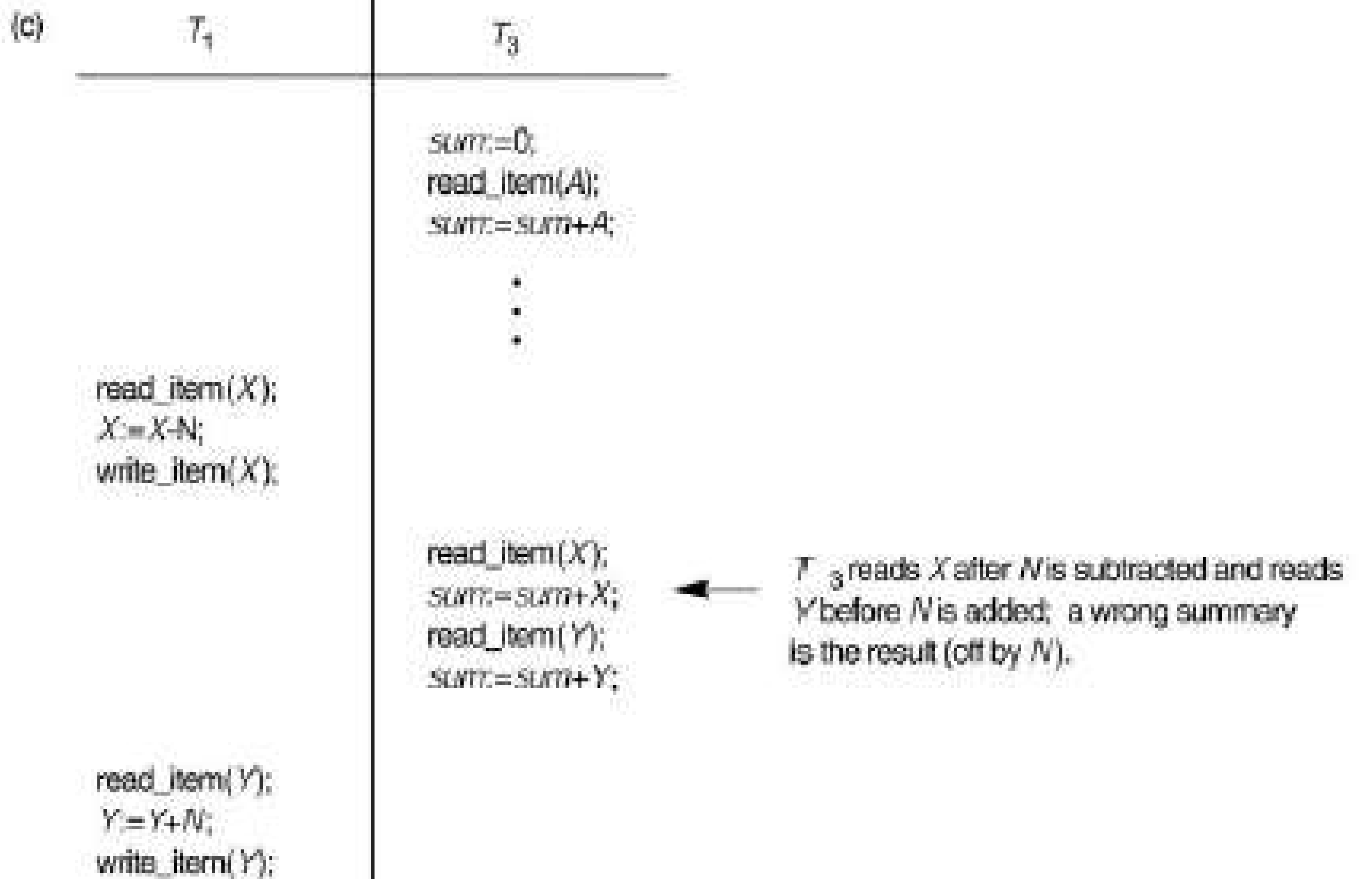


FIGURE 17.3 (continued)

Some problems that occur when concurrent execution is uncontrolled. (c) The incorrect summary problem.



Why recovery is needed:

1. **A computer failure (system crash):** A hardware or software error occurs in the computer system during transaction execution. If the hardware crashes, the contents of the computer's internal memory may be lost.

2. **A transaction or system error :** Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. In addition, the user may interrupt the transaction during its execution.

3. **Local errors or exception conditions** detected by the transaction:

- certain conditions necessitate cancellation of the transaction. For example, data for the transaction may not be found. A condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal from that account, to be canceled.

- a programmed abort in the transaction causes it to fail.

4. **Concurrency control enforcement:** The concurrency control method may decide to abort the transaction, to be restarted later, because it violates serializability or because several transactions are in a state of deadlock.

5. **Disk failure:** Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.

6. **Physical problems and catastrophes:** This refers to an endless list of problems that includes power or airconditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

Transaction and System Concepts

A **transaction** is an atomic unit of work that is either completed in its entirety or not done at all. For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts.

Transaction states:

- Active state
- Partially committed state
- Committed state
- Failed state
- Terminated State

Recovery manager keeps track of the following operations:

⑩ **begin_transaction:** This marks the beginning of transaction execution.

⑩ **read or write:** These specify read or write operations on the database items that are executed as part of a transaction.

⑩ **end_transaction:** This specifies that read and write transaction operations have ended and marks the end limit of transaction execution. At this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database or whether the transaction has to be aborted because it violates concurrency control or for some other reason.

•**commit_transaction:** This signals a *successful end* of the transaction so that any changes (updates) executed by the transaction can be safely **committed** to the database and will not be undone.

⑩ **rollback (or abort):** This signals that the transaction has *ended unsuccessfully,* so that any changes or effects that the transaction may

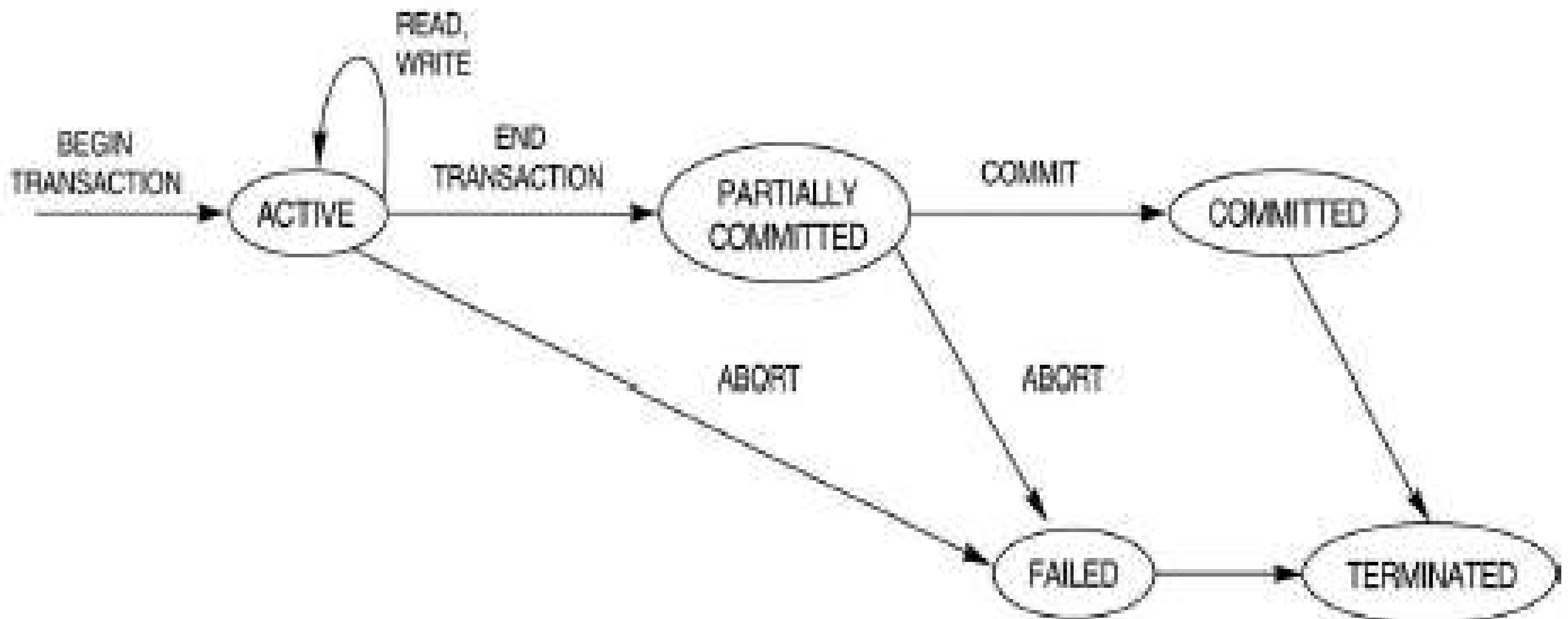
Recovery techniques use the following operators:

- **undo:** Similar to rollback except that it applies to a single operation rather than to a whole transaction.

- **redo:** This specifies that certain *transaction operations* must be *redone* to ensure that all the operations of a committed transaction have been applied successfully to the database.

FIGURE 17.4

State transition diagram illustrating the states for transaction execution.



The System Log

• **Log or Journal** : The log keeps track of all transaction operations that affect the values of database items. This information may be needed to permit recovery from transaction failures. The log is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure. In addition, the log is periodically backed up to archival storage (tape) to guard against such catastrophic failures.

➓ T in the following discussion refers to a unique **transactionid** that is generated automatically by the system and is used to identify each transaction:

Types of log record:

1. [start_transaction,T]: Records that transaction T has started execution.

2. [write_item,T,X,old_value,new_value]: Records that transaction T has changed the value of database item X from old_value to new_value.

3. [read_item,T,X]: Records that transaction T has read the value of database item X.

4. [commit,T]: Records that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.

5. [abort,T]: Records that transaction T has been aborted

⑩ protocols for recovery that avoid cascading rollbacks do not require that read operations be written to the system log, whereas other protocols require these entries for recovery.

•strict protocols require simpler write entries that do not include new_value.

Recovery using log records:

If the system crashes, we can recover to a consistent database state by examining the log and using some techniques.

1. Because the log contains a record of every write operation that changes the value of some database item, it is possible to **undo** the effect of these write operations of a transaction T by tracing backward through the log and resetting all items changed by a write operation of T to their old_values.

2. We can also **redo** the effect of the write operations of a transaction T by tracing forward through the log and setting all items changed by a write operation of T (that did not get done permanently) to their new_values.

Commit Point of a Transaction:

•**Definition:** A transaction T reaches its **commit point** when all its operations that access the database have been executed successfully *and* the effect of all the transaction operations on the database has been recorded in the log. Beyond the commit point, the transaction is said to be **committed,** and its effect is assumed to be *permanently recorded* in the database. The transaction then writes an entry [commit,T] into the log.

⑩ **Roll Back of transactions:** Needed for transactions that have a [start_transaction,T] entry into the log but no commit entry [commit,T] into the log.

Commit Point of a Transaction (cont):

- **Redoing transactions:** Transactions that have written their commit entry in the log must also have recorded all their write operations in the log; otherwise they would not be committed, so their effect on the database can be *redone* from the log entries. (Notice that the log file must be kept on disk. At the time of a system crash, only the log entries that have been *written back to disk* are considered in the recovery process because the contents of main memory may be lost.)

- **Force writing a log:** *before* a transaction reaches its commit point, any portion of the log that has not been written to the disk yet must now be written to the disk. This process is called force-writing the log file before committing a transaction.

Desirable Properties of Transactions

ACID properties:

- **Atomicity**: A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
- **Consistency preservation**: A correct execution of the transaction must take the database from one consistent state to another.
- **Isolation**: A transaction should not make its updates visible to other transactions until it is committed; this property, when enforced strictly, solves the temporary update problem and makes cascading rollbacks of transactions unnecessary .
- **Durability or permanency**: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

Characterizing Schedules based on Recoverability

- **Transaction schedule or history:** When transactions are executing concurrently in an interleaved fashion, the order of execution of operations from the various transactions forms what is known as a transaction schedule (or history).

- A **schedule** (or **history**) S of n transactions T1, T2, ..., Tn :

It is an ordering of the operations of the transactions subject to the constraint that, for each transaction Ti that participates in S, the operations of T1 in S must appear in the same order in which they occur in T1. Note, however, that operations from other transactions Tj can be interleaved with the operations of Ti in S.

Schedules classified on recoverability:

- **Recoverable schedule:** One where no transaction needs to be rolled back.

A schedule S is **recoverable** if no transaction T in S commits until all transactions T' that have written an item that T reads have committed.

- **Cascadeless schedule:** One where every transaction reads only the items that are written by committed transactions.

Schedules requiring cascaded rollback: A schedule in which uncommitted transactions that read an item from a failed transaction must be rolled back.

- **Strict Schedules:** A schedule in which a transaction can neither read or write an item X until the last transaction that wrote X has committed.

Characterizing Schedules based on Serializability

- **Serial schedule**: A schedule S is **serial** if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule. Otherwise, the schedule is called **nonserial schedule.**
- **Serializable schedule**: A schedule S is **serializable** if it is equivalent to some serial schedule of the same n transactions.
- **Result equivalent**: Two schedules are called result equivalent if they produce the same final state of the database.
- **Conflict equivalent**: Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules.
- **Conflict serializable**: A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule S'.

- Being serializable is not the same as being serial
- Being serializable implies that the schedule is a correct schedule.

– It will leave the database in a consistent state.

– The interleaving is appropriate and will result in a state as if the transactions were serially executed, yet will achieve efficiency due to concurrent execution.

- Serializability is hard to check.

– Interleaving of operations occurs in an operating system through some scheduler

– Difficult to determine beforehand how the operations in a schedule will be interleaved.

Practical approach:

- Come up with methods (protocols) to ensure serializability.
- It's not possible to determine when a schedule begins and when it ends. Hence, we reduce the problem of checking the whole schedule to checking only a *committed project* of the schedule (i.e. operations from only the committed transactions.)
- Current approach used in most DBMSs:

– Use of locks with two phase locking

- **View equivalence**: A less restrictive definition of equivalence of schedules
- **View serializability:** definition of serializability based on view equivalence. A schedule is *view serializable* if it is *view equivalent* to a serial schedule.

Two schedules are said to be **view equivalent** if the following three conditions hold:

1. The same set of transactions participates in S and S', and S and S' include the same operations of those transactions.

2. For any operation Ri(X) of Ti in S, if the value of X read by the operation has been written by an operation Wj(X) of Tj (or if it is the original value of X before the schedule started), the same condition must hold for the value of X read by operation Ri(X) of Ti in S'.

3. If the operation Wk(Y) of Tk is the last operation to write item Y in S, then Wk(Y) of Tk must also be the last operation to write item Y in S'.

The premise behind view equivalence:

- As long as each read operation of a transaction reads the result of *the same write operation* in both schedules, the write operations of each transaction must produce the same results.

- **"The view"**: the read operations are said to see the *the same view* in both schedules.

Characterizing Schedules based on Serializability

Relationship between view and conflict equivalence:

•The two are same under **constrained write assumption** which assumes that if T writes X, it is constrained by the value of X it read; i.e., new X = f(old X)

⓾ Conflict serializability is **stricter** than view serializability. With unconstrained write (or blind write), a schedule that is view serializable is not necessarily conflict serialiable.

⓾ Any conflict serializable schedule is also view serializable, but not vice versa.

Consider the following schedule of three transactions

T1: r1(X), w1(X); T2: w2(X); and T3: w3(X):

Schedule Sa: r1(X); w2(X); w1(X); w3(X); c1; c2; c3;

In Sa, the operations w2(X) and w3(X) are blind writes, since T1 and T3 do not read the value of X.

Sa is **view serializable**, since it is view equivalent to the serial schedule T1, T2, T3. However, Sa is **not conflict serializable**, since it is not conflict equivalent to any serial schedule.

Testing for conflict serializability

Algorithm 17.1:

1. Looks at only read_Item (X) and write_Item (X) operations

2. Constructs a precedence graph (serialization graph) - a graph with directed edges

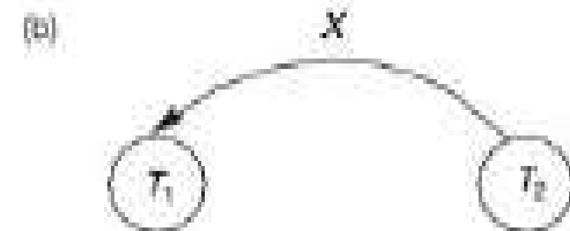
3. An edge is created from Ti to Tj if one of the operations in Ti appears before a conflicting operation in Tj

4. The schedule is serializable if and only if the precedence graph has no cycles.

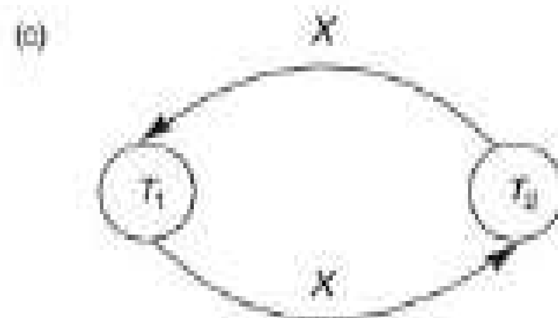
Figure 19.7 Constructing the precedence graphs for schedules *A* to *D* from Figure 19.5 to test for conflict serializability. (a) Precedence graph for serial schedule *A*. (b) Precedence graph for serial schedule *B*. (c) Precedence graph for schedule *C* (not serializable). (d) Precedence graph for schedule *D* (serializable, equivalent to schedule *A*).

(a)

(b)



(c)



(d)

Other Types of Equivalence of Schedules

⑩ Under special **semantic constraints**, schedules that are otherwise not conflict serializable may work correctly. Using commutative operations of addition and subtraction (which can be done in any order) certain non-serializable transactions may work correctly

Example: bank credit / debit transactions on a given item are **separable** and **commutative.** Consider the following schedule S for the two transactions:

Sh : r1(X); w1(X); r2(Y); w2(Y); r1(Y); w1(Y); r2(X); w2(X);

Using conflict serializability, it is not **serializable.** However, if it came from a (read,update, write) sequence as follows:

r1(X); X := X – 10; w1(X); r2(Y); Y := Y – 20;r1(Y);

Y := Y + 10; w1(Y); r2(X); X := X + 20; (X);

Sequence explanation: debit, debit, credit, credit. It is a **correct schedule for the given semantics**

Concurrency Control Techniques

Topics to be covered

- Locking Techniques for Concurrency Control
- Concurrency Control Based on Timestamp Ordering
- Validation Concurrency Control Techniques
- Granularity of Data Items
- Multiple Granularity Locking.

Database Concurrency Control

Purpose of Concurrency Control

- To enforce Isolation (through mutual exclusion) among conflicting transactions.

- To preserve database consistency through consistency preserving execution of transactions.

- To resolve read-write and write-write conflicts.

Example: In concurrent execution environment if T1 conflicts with T2 over a data item A, then the existing concurrency control decides if T1 or T2 should get the A and if the other transaction is rolled-back or waits.

Two-Phase Locking Techniques

Locking is an operation which secures

(a) permission to Read or

(b) (b) permission to Write a data item for a transaction.

Example: Lock (X). Data item X is locked in behalf of the requesting transaction.

Unlocking is an operation which removes these permissions from the data item. Example: Unlock (X). Data item X is made available to all other transactions.

Lock and Unlock are Atomic operations.

Two-Phase Locking Techniques: Essential components

Two locks modes (a) shared (read) and (b) exclusive (write).

Shared mode: shared lock (X). More than one transaction can apply share lock on X for reading its value but no write lock can be applied on X by any other transaction.

Exclusive mode: Write lock (X). Only one write lock on X can exist at any time and no shared lock can be applied by any other transaction on X.

Conflict matrix

	Read	Write
Read	Y	N
Write	N	N

Lock Manager: Managing locks on data items.

Lock table: Lock manager uses it to store the identify of transaction locking a data item, the data item, lock mode and pointer to the next data item locked. One simple way to implement a lock table is through linked list.

Transaction ID	Data item id	lock mode	Ptr to next data item
T1	X1	Read	Next

Database requires that all transactions should be well formed.

A transaction is well-formed if:

- It must lock the data item before it reads or writes to it.
- It must not lock an already locked data items and it must not try to unlock a free data item.

The following code performs the lock operation:

```
B: if LOCK (X) = 0 (*item is unlocked*)
then LOCK (X) ← 1 (*lock the item*)
else begin
        wait (until lock (X) = 0) and
        the lock manager wakes up the transaction);
goto B
end;
```

The following code performs the unlock operation:

LOCK (X) ← 0 (*unlock the item*)

if any transactions are waiting then

wake up one of the waiting the transactions;

The following code performs the read operation:

```
B: if LOCK (X) = "unlocked" then
        begin LOCK (X) ← "read-locked";
                no_of_reads (X) ← 1;
        end
        else if LOCK (X) ← "read-locked" then
                no_of_reads (X) ← no_of_reads (X) +1
        else begin wait (until LOCK (X) = "unlocked" and
                the lock manager wakes up the transaction);
                go to B
                end;
```

The following code performs the write lock operation:

```
B: if LOCK (X) = "unlocked" then
        begin LOCK (X) ← "read-locked";
                no_of_reads (X) ← 1;
        end
        else if LOCK (X) ← "read-locked" then
                no_of_reads (X) ← no_of_reads (X) +1
                else begin wait (until LOCK (X) = "unlocked" and
                        the lock manager wakes up the transaction);
                        go to B
                end;
```

The following code performs the unlock operation:

```
if LOCK (X) = "write-locked" then
        begin LOCK (X) ← "unlocked";
                wakes up one of the transactions, if any
        end
        else if LOCK (X) ← "read-locked" then
        begin
                        no_of_reads (X) ← no_of_reads (X) -1
                if no_of_reads (X) = 0 then
                begin
                        LOCK (X) = "unlocked";
                        wake up one of the transactions, if any
                        end
                        end;
```

Lock conversion

Lock upgrade: existing read lock to write lock

if Ti has a read-lock (X) and Tj has no read-lock (X) ($i \neq j$) then

convert read-lock (X) to write-lock (X)

else

force Ti to wait until Tj unlocks X

Lock downgrade: existing write lock to read lock

Ti has a write-lock (X) (*no transaction can have any lock on X*)

convert write-lock (X) to read-lock (X)

Two-Phase Locking Techniques: The algorithm

Two Phases: (a) Locking (Growing) (b) Unlocking (Shrinking).

Locking (Growing) Phase: A transaction applies locks (read or write) on desired data items one at a time.

Unlocking (Shrinking) Phase: A transaction unlocks its locked data items one at a time.

Requirement: For a transaction these two phases must be mutually

exclusively, that is, during locking phase unlocking phase must not start and during unlocking phase locking phase must not begin.

Two-Phase Locking Techniques: The algorithm

T1	T2	Result
read_lock (Y);	read_lock (X);	Initial values: X=20; Y=30
read_item (Y);	read_item (X);	Result of serial execution
unlock (Y);	unlock (X);	T1 followed by T2
write_lock (X);	Write_lock (Y);	X=50, Y=80.
read_item (X);	read_item (Y);	Result of serial execution
X:=X+Y;	Y:=X+Y;	T2 followed by T1
write_item (X);	write_item (Y);	X=70, Y=50
unlock (X);	unlock (Y);	

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

T1	T2	Result
read_lock (Y); read_item (Y); **unlock (Y);**		X=50; Y=50 Nonserializable because it. violated two-phase policy.
	read_lock (X); read_item (X); **unlock (X);** **write_lock (Y);** read_item (Y); Y:=X+Y; write_item (Y); unlock (Y);	
write_lock (X); read_item (X); X:=X+Y; write_item (X); unlock (X);		

Time ↓

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

T'1	T'2
read_lock (Y);	read_lock (X);
read_item (Y);	read_item (X);
write_lock (X);	Write_lock (Y);
unlock (Y);	unlock (X);
read_item (X);	read_item (Y);
X:=X+Y;	Y:=X+Y;
write_item (X);	write_item (Y);
unlock (X);	unlock (Y);

T1 and T2 follow two-phase policy but they are subject to deadlock, which must be dealt with.

Two-phase policy generates two locking algorithms (a) Basic and (b)Conservative.

Conservative: Prevents deadlock by locking all desired data items before transaction begins execution.

Basic: Transaction locks data items incrementally. This may cause

deadlock which is dealt with.

Strict: A more stricter version of Basic algorithm where unlocking is performed after a transaction terminates (commits or aborts and rolledback).

This is the most commonly used two-phase locking algorithm.

Database Concurrency Control

Dealing with Deadlock and Starvation

Deadlock

T'1	T'2
read_lock (Y);	
read_item (Y);	
	read_lock (X);
	read_item (Y);
write_lock (X);	
(waits for X)	write_lock (Y);
	(waits for Y)

Deadlock (T'1 and T'2)

T1 and T2 did follow two-phase policy but they are deadlock

Dealing with Deadlock and Starvation

Deadlock prevention

A transaction locks all data items it refers to before it begins execution. This way of locking prevents deadlock since a transaction never waits for a data item. The conservative two-phase locking uses this approach.

Deadlock detection and resolution

In this approach, deadlocks are allowed to happen. The scheduler maintains a wait-for-graph for detecting cycle. If a cycle exists, then one transaction involved in the cycle is selected (victim) and rolled back.

A wait-for-graph is created using the lock table. As soon as a transaction is blocked, it is added to the graph. When a chain like: Ti waits for Tj waits for Tk waits for Ti or Tj occurs, then this creates a cycle. One of the transaction of the cycle is selected and rolled back.

Deadlock avoidance

There are many variations of two-phase locking algorithm. Some avoid deadlock by not letting the cycle to complete. That is as soon as the algorithm discovers that blocking a transaction is likely to create a cycle, it rolls back the transaction. Wound-Wait and Wait-Die algorithms use timestamps to avoid deadlocks by rolling-back victim.

Starvation

Starvation occurs when a particular transaction consistently waits or restarted and never gets a chance to proceed further. In a deadlock resolution it is possible that the same transaction may consistently be selected as victim and rolled-back. This limitation is inherent in all priority based scheduling mechanisms. In Wound-Wait scheme a younger transaction may always be wounded (aborted) by a long running older transaction which may create starvation.

Timestamp based concurrency control algorithm

Timestamp

A monotonically increasing variable (integer) indicating the age of an operation or a transaction. A larger timestamp value indicates a more recent event or operation.

Timestamp based algorithm uses timestamp to serialize the execution of concurrent transactions.

Basic Timestamp Ordering

1. Transaction T issues a write_item(X) operation:

a. If read_TS(X) > TS(T) or if write_TS(X) > TS(T), then an younger transaction has already read the data item so abort and roll-back T and reject the operation.

b. If the condition in part (a) does not exist, then execute write_item(X) of T and set write_TS(X) to TS(T).

2. Transaction T issues a read_item(X) operation:

a. If write_TS(X) > TS(T), then an younger transaction has already written to the data item so abort and roll-back T and reject the operation.

b. If write_TS(X) ≤ TS(T), then execute read_item(X) of T and set read_TS(X) to the larger of TS(T) and the current read_TS(X).

Strict Timestamp Ordering

1. Transaction T issues a write_item(X) operation:

a. If TS(T) > read_TS(X), then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).

2. Transaction T issues a read_item(X) operation:

a. If TS(T) > write_TS(X), then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).

Thomas's Write Rule

1. If read_TS(X) > TS(T) then abort and roll-back T and reject the operation.
2. If write_TS(X) > TS(T), then just ignore the write operation and continue execution. This is because the most recent writes counts in case of two consecutive writes.
3. If the conditions given in 1 and 2 above do not occur, then execute write_item(X) of T and set write_TS(X) to TS(T).

Multiversion concurrency control techniques

Concept

This approach maintains a number of versions of a data item and allocates the right version to a read operation of a transaction. Thus unlike other mechanisms a read operation in this mechanism is never rejected.

Side effect: Significantly more storage (RAM and disk) is required to maintain multiple versions. To check unlimited growth of versions, a garbage collection is run when some criteria is satisfied.

Validation (Optimistic) Concurrency Control Schemes

In this technique only at the time of commit serializability is checked and transactions are aborted in case of non-serializable schedules.

Three phases:

Read phase: A transaction can read values of committed data items. However, updates are applied only to local copies (versions) of the data items (in database cache).

Validation phase: Serializability is checked before transactions write their updates to the database.

This phase for Ti checks that, for each transaction Tj that is either committed or is in its validation phase, one of the following conditions holds:

1. Tj completes its write phase before Ti starts its read phase.

2. Ti starts its write phase after Tj completes its write phase, and the read_set of Ti has no items in common with the write_set of Tj

3. Both the read_set and write_set of Ti have no items in common with the write_set of Tj, and Tj completes its read phase.

When validating Ti, the first condition is checked first for each transaction Tj, since (1) is the simplest condition to check. If (1) is false then (2) is checked and if (2) is false then (3) is checked. If none of these conditions holds, the validation fails and Ti is aborted.

Write phase: On a successful validation transactions' updates are applied to the database; otherwise, transactions are restarted.

Granularity of data items and Multiple Granularity Locking

A lockable unit of data defines its granularity. Granularity can be coarse (entire database) or it can be fine (a tuple or an attribute of a relation). Data item granularity significantly affects concurrency control performance. Thus, the degree of concurrency is low for coarse granularity and high for fine granularity. Example of data item granularity:

1. A field of a database record (an attribute of a tuple).

2. A database record (a tuple or a relation).

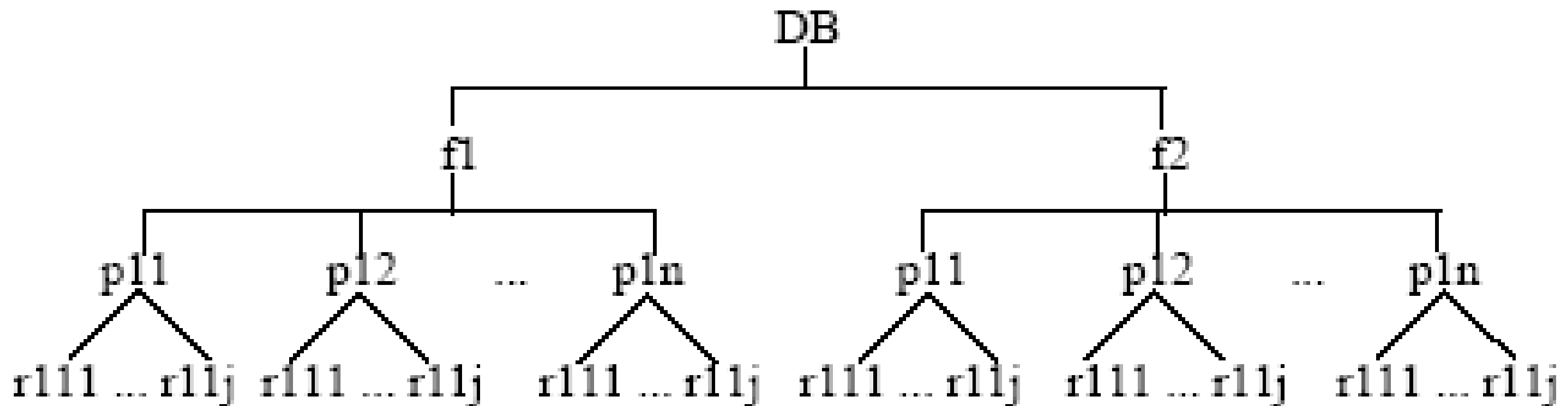
3. A disk block.

4. An entire file.

5. The entire atabase.

Granularity of data items and Multiple Granularity Locking

The following diagram illustrates a hierarchy of granularity from coarse (database) to fine (record).



To manage such hierarchy, in addition to read and write, three additional locking modes, called intention lock modes are defined:

Intention-shared (IS): indicates that a shared lock(s) will be requested on some descendent nodes(s).

Intention-exclusive (IX): indicates that an exclusive lock(s) will be requested on some descendent nodes(s).

Shared-intention-exclusive (SIX): indicates that the current node is locked in shared mode but an exclusive lock(s) will be requested on some descendent nodes(s).

Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

These locks are applied using the following compatibility matrix:

	IS	IX	S	SIX	X
IS	yes	yes	yes	yes	no
IX	yes	yes	no	no	no
S	yes	no	yes	no	no
SIX	yes	no	no	no	no
X	no	no	no	no	no

The set of rules which must be followed for producing serializable schedule are

1. The lock compatibility must adhered to.

2. The root of the tree must be locked first, in any mode..

3. A node N can be locked by a transaction T in S or IX mode only if the parent node is already locked by T in either IS or IX mode.

4. A node N can be locked by T in X, IX, or SIX mode only if the parent of N is already locked by T in either IX or SIX mode.

5. T can lock a node only if it has not unlocked any node (to enforce 2PL policy).

6. T can unlock a node, N, only if none of the children of N are currently locked by T.

Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

An example of a serializable execution:

T1	T2	T3
IX(db)		
IX(f1)		
	IX(db)	
		IS(db)
		IS(f1)
		IS(p11)
IX(p11)		
X(r111)		
	IX(f1)	
	X(p12)	
		S(r11j)
IX(f2)		
IX(p21)		
IX(r211)		
Unlock (r211)		
Unlock (p21)		
Unlock (f2)		
		S(f2)

Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

An example of a serializable execution (continued):

T1	T2	T3
	unlock(p12)	
	unlock(f1)	
	unlock(db)	
unlock(r111)		
unlock(p11)		
unlock(f1)		
unlock(db)		
		unlock (r111j)
		unlock (p11)
		unlock (f1)
		unlock(f2)
		unlock(db)

Database Recovery Techniques

Topics to be covered

- Recovery Concepts
- Recovery Techniques Based on Deferred Update
- Recovery Techniques Based on Immediate Update
- Shadow Paging.

Purpose of Database Recovery

- To bring the database into the last consistent state, which existed prior to the failure.
- To preserve transaction properties (Atomicity, Consistency, Isolation and Durability).

Example: If the system crashes before a fund transfer transaction completes its execution, then either one or both accounts may have incorrect value. Thus, the database must be restored to the state before the transaction modified any of the accounts.

Types of Failure

The database may become unavailable for use due to

- Transaction failure: Transactions may fail because of incorrect input, deadlock, incorrect synchronization.
- System failure: System may fail because of addressing error, application error, operating system fault, RAM failure, etc.
- Media failure: Disk head crash, power disruption, etc.

Transaction Log

For recovery from any type of failure data values prior to modification (BFIM - BeFore Image) and the new value after modification (AFIM – AFter Image) are required. These values and other information is stored in a sequential file called Transaction log.

A sample log is given below. **Back P** and **Next P** point to the previous and next log records of the same transaction.

T ID	Back P	Next P	Operation	Data item	BFIM	AFIM
T1	0	1	Begin			
T1	1	4	Write	X	X = 100	X = 200
T2	0	8	Begin			
T1	2	5	W	Y	Y = 50	Y = 100
T1	4	7	R	M	M = 200	M = 200
T3	0	9	R	N	N = 400	N = 400
T1	5	nil	End			

Data Update

- **Immediate Update**: As soon as a data item is modified in cache, the disk copy is updated.
- **Deferred Update**: All modified data items in the cache is written either after a transaction ends its execution or after a fixed number of transactions have completed their execution.
- **Shadow update**: The modified version of a data item does not overwrite its disk copy but is written at a separate disk location.
- **In-place update**: The disk version of the data item is overwritten by the cache version.

Data Caching

Data items to be modified are first stored into database cache by the Cache Manager (CM) and after modification they are flushed (written) to the disk. The flushing is controlled by **Modified** and **Pin-Unpin** bits.

Pin-Unpin: Instructs the operating system not to flush the data item.

Modified: Indicates the AFIM of the data item.

Transaction Roll-back (Undo) and Roll-Forward (Redo)

To maintain atomicity, a transaction's operations are **redone** or **undone**.

Undo: Restore all BFIMs on to disk (Remove all AFIMs).

Redo: Restore all AFIMs on to disk.

Database recovery is achieved either by performing only Undos or only Redos or by a combination of the two. These operations are recorded in the log as they happen.

Roll-back

We show the process of roll-back with the help of the following three transactions T1, and T2 and T3.

T1	T2	T3
read_item (A)	read_item (B)	read_item (C)
read_item (D)	write_item (B)	write_item (B)
write_item (D)	read_item (D)	read_item (A)
	write_item (A)	write_item (A)

Database Recovery

Roll-back: One execution of T1, T2 and T3 as recorded in the log.

		A	B	C	D
		30	15	40	20
	[start_transaction, T3]				
	[read_item, T3, C]				
*	[write_item, T3, B, 15, 12]		12		
	[start_transaction,T2]				
	[read_item, T2, B]				
**	[write_item, T2, B, 12, 18]		18		
	[start_transaction,T1]				
	[read_item, T1, A]				
	[read_item, T1, D]				
	[write_item, T1, D, 20, 25]				25
	[read_item, T2, D]				
**	[write_item, T2, D, 25, 26]				26
	[read_item, T3, A]				

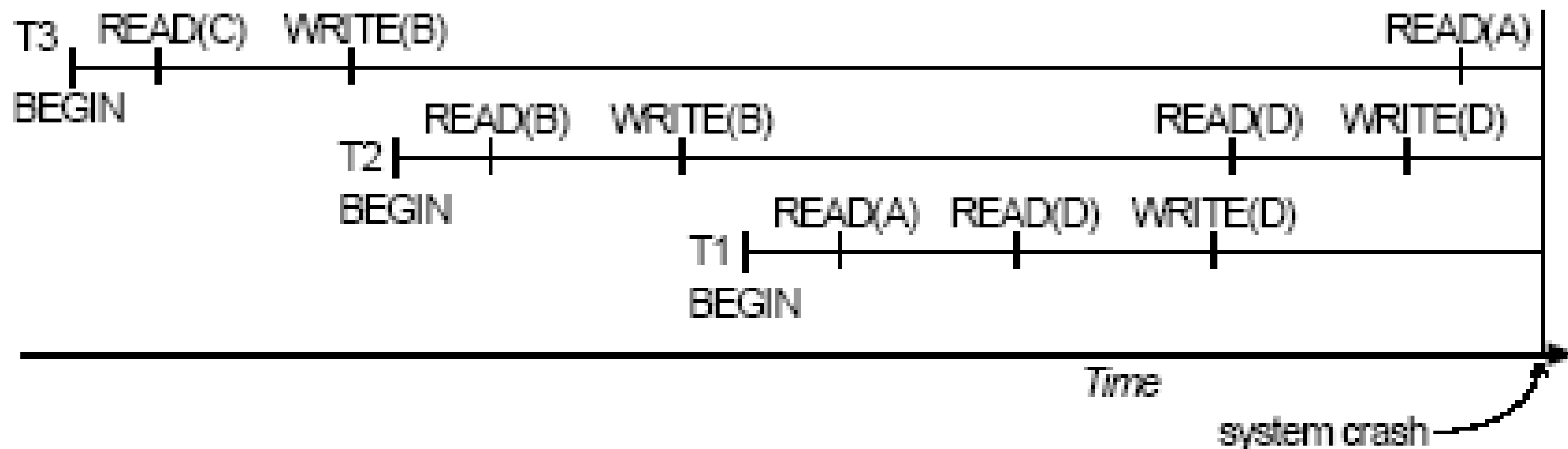
---- system crash ----

* T3 is rolled back because it did not reach its commit point.

** T2 is rolled back because it reads the value of item B written by T3.

Database Recovery

Roll-back: One execution of T1, T2 and T3 as recorded in the log.



Illustrating cascading roll-back

Write-Ahead Logging

When **in-place** update (immediate or deferred) is used then log is necessary for recovery and it must be available to recovery manager. This is achieved by **Write-Ahead Logging** (WAL) protocol. WAL states that

For Undo: Before a data item's AFIM is flushed to the database disk (overwriting the BFIM) its BFIM must be written to the log and the log must be saved on a stable store (log disk).

For Redo: Before a transaction executes its commit operation, all its AFIMs must be written to the log and the log must be saved on a stable store.

Checkpointing

Time to time (randomly or under some criteria) the database flushes its buffer to database disk to minimize the task of recovery. The following steps defines a checkpoint operation:

1. Suspend execution of transactions temporarily.

2. Force write modified buffer data to disk.

3. Write a [checkpoint] record to the log, save the log to disk.

4. Resume normal transaction execution.

During recovery **redo** or **undo** is required to transactions appearing after [checkpoint] record.

Steal/No-Steal and Force/No-Force

Possible ways for flushing database cache to database disk:

Steal: Cache can be flushed before transaction commits.

No-Steal: Cache cannot be flushed before transaction commit.

Force: Cache is immediately flushed (forced) to disk.

No-Force: Cache is deferred until transaction commits.

These give rise to four different ways for handling recovery:

Steal/No-Force (Undo/Redo),

Steal/Force (Undo/No-redo),

No-Steal/No-Force (Redo/No-undo) and

No-Steal/Force (Noundo/ No-redo).

Recovery Scheme

Deferred Update (No Undo/Redo)

The data update goes as follows:

1. A set of transactions records their updates in the log.
2. At commit point under WAL scheme these updates are saved on database disk.

After reboot from a failure the log is used to redo all the transactions affected by this failure. No undo is required because no AFIM is flushed to the disk before a transaction commits.

Deferred Update in a single-user system

There is no concurrent data sharing in a single user system. The data update goes as follows:

1. A set of transactions records their updates in the log.
2. At commit point under WAL scheme these updates are saved on database disk.

After reboot from a failure the log is used to redo all the transactions affected by this failure. No undo is required because no AFIM is flushed to the disk before a transaction commits.

Database Recovery

Deferred Update in a single-user system

(a)

T1	T2
read_item (A)	read_item (B)
read_item (D)	write_item (B)
write_item (D)	read_item (D)
	write_item (A)

(b)

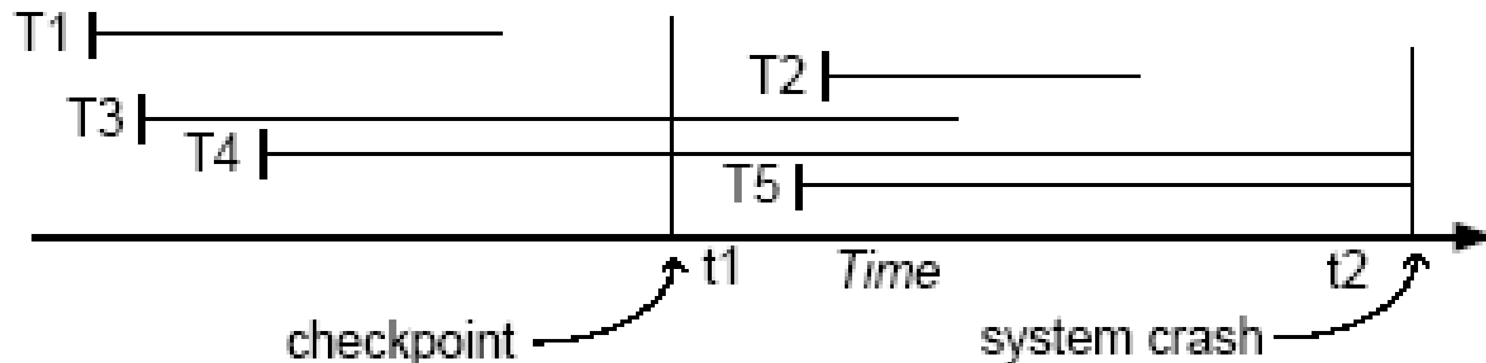
[start_transaction, T1]
[write_item, T1, D, 20]
[commit T1]
[start_transaction, T1]
[write_item, T2, B, 10]
[write_item, T2, D, 25] ? system crash

The [write_item, …] operations of T1 are redone.
T2 log entries are ignored by the recovery manager.

Database Recovery

Deferred Update with concurrent users

This environment requires some concurrency control mechanism to guarantee **isolation** property of transactions. In a system recovery transactions which were recorded in the log after the last checkpoint were **redone**. The recovery manager may scan some of the transactions recorded before the checkpoint to get the AFIMs.



Recovery in a concurrent users environment.

Database Recovery

Deferred Update with concurrent users

(a)

T1	T2	T3	T4
read_item (A)	read_item (B)	read_item (A)	read_item (B)
read_item (D)	write_item (B)	write_item (A)	write_item (B)
write_item (D)	read_item (D)	read_item (C)	read_item (A)
	write_item (D)	write_item (C)	write_item (A)

(b) [start_transaction, T1]
[write_item, T1, D, 20]
[commit, T1]
[checkpoint]
[start_transaction, T4]
[write_item, T4, B, 15]
[write_item, T4, A, 20]
[commit, T4]
[start_transaction T2]
[write_item, T2, B, 12]
[start_transaction, T3]
[write_item, T3, A, 30]
[write_item, T2, D, 25] ? system crash

T2 and T3 are ignored because they did not reach their commit points.
T4 is redone because its commit point is after the last checkpoint.

Deferred Update with concurrent users

Two tables are required for implementing this protocol:

Active table: All active transactions are entered in this table.

Commit table: Transactions to be committed are entered in this table. During recovery, all transactions of the **commit** table are redone and all transactions of **active** tables are ignored since none of their AFIMs reached the database. It is possible that a **commit** table transaction may be **redone** twice but this does not create any inconsistency because of a redone is "**idempotent**", that is, one redone for an AFIM is equivalent to multiple redone for the same AFIM.

Recovery Techniques Based on Immediate Update

Undo/No-redo Algorithm

In this algorithm AFIMs of a transaction are flushed to the database disk under WAL before it commits. For this reason the recovery manager **undoes** all transactions during recovery. No transaction is **redone**. It is possible that a transaction might have completed execution and ready to commit but this transaction is also **undone**.

Undo/Redo Algorithm (Single-user environment)

Recovery schemes of this category apply **undo** and also **redo** for recovery. In a single-user environment no concurrency control is required but a log is maintained under WAL. Note that at any time there will be one transaction in the system and it will be either in the commit table or in the active table. The recovery manager performs:

1. **Undo** of a transaction if it is in the **active** table.

2. **Redo** of a transaction if it is in the **commit** table.

Undo/Redo Algorithm (Concurrent execution)

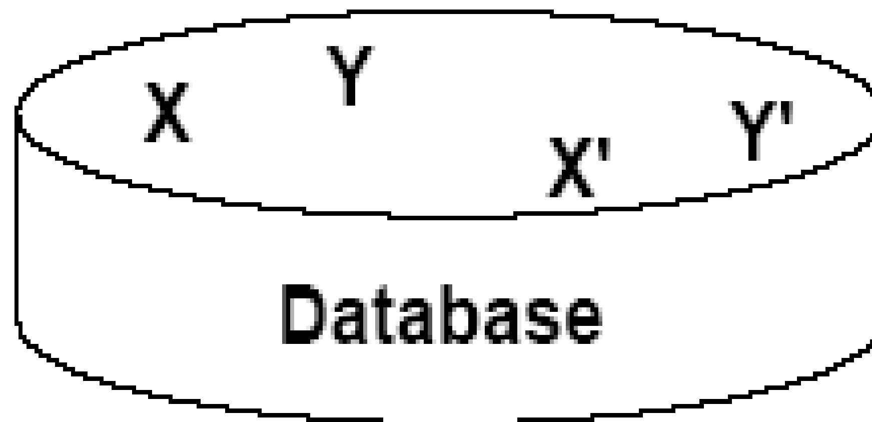
Recovery schemes of this category applies undo and also redo to recover the database from failure. In concurrent execution environment a concurrency control is required and log is maintained under WAL.

Commit table records transactions to be committed and active table records active transactions. To minimize the work of the recovery manager checkpointing is used. The recovery performs:

1. **Undo** of a transaction if it is in the active table.
2. **Redo** of a transaction if it is in the commit table.

Shadow Paging

The AFIM does not overwrite its BFIM but recorded at another place on the disk. Thus, at any time a data item has AFIM and BFIM (Shadow copy of the data item) at two different places on the disk.

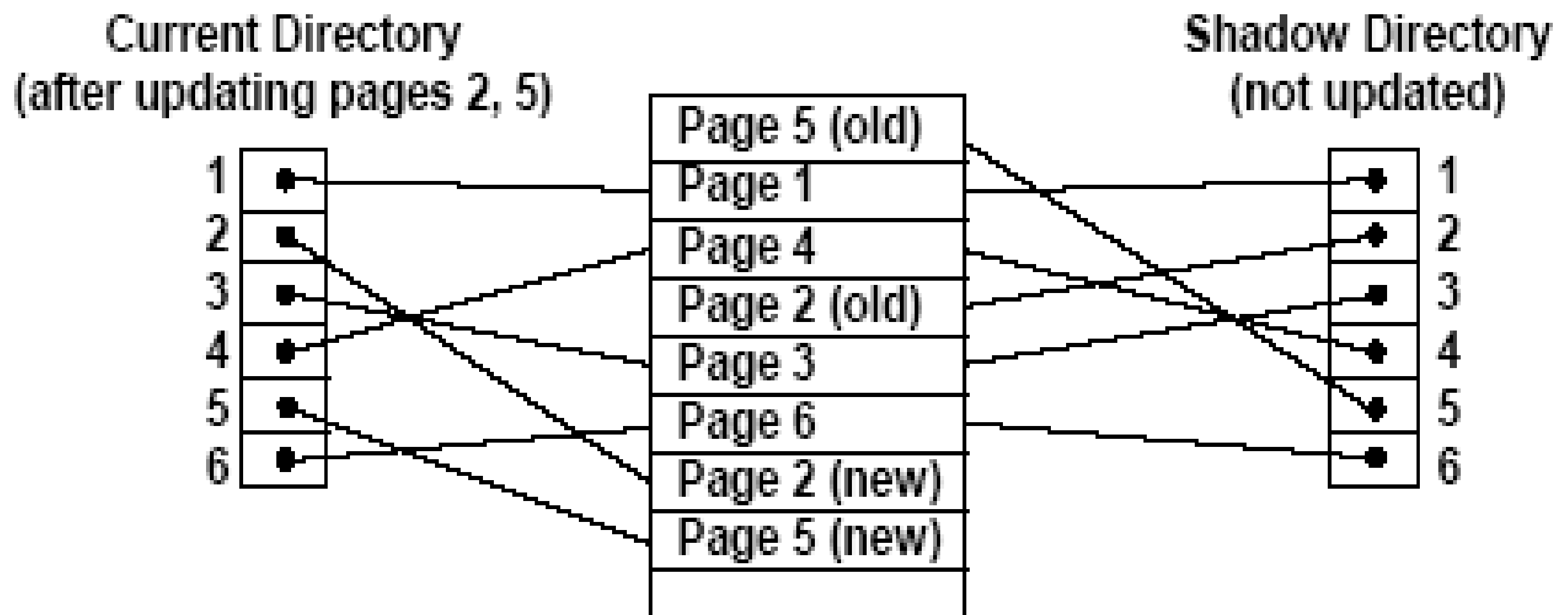


X and Y: Shadow copies of data items

X` and Y`: Current copies of data items

Shadow Paging

To manage access of data items by concurrent transactions two directories (current and shadow) are used. The directory arrangement is illustrated below. Here a page is a data item.



Object Database Standards,
Languages, and Design

Overview of the Object Model ODMG

The Object Definition Language DDL

The Object Query Language OQL

Object Database Conceptual Model

Summary

- Discuss the importance of standards (e.g., portability, interoperability)
- Introduce Object Data Management Group (ODMG): object model, object definition language (ODL), object query language (OQL)
- Present ODMG object binding to programming languages (e.g., C++)
- Present Object Database Conceptual Design

The Object Model of ODMG

- Provides a standard model for object databases
- Supports object definition via ODL
- Supports object querying via OQL
- Supports a variety of data types and type constructors

ODMG Objects and Literals

- The basic building blocks of the object model are
  - **Objects**
  - **Literals**
- An object has four characteristics
  1. **Identifier**: unique system-wide identifier
  2. **Name**: unique within a particular database and/or program; it is optional
  3. **Lifetime**: persistent vs. transient
  4. **Structure**: specifies how object is constructed by the type constructor and whether it is an atomic object

ODMG Literals

- A literal has a current value but not an identifier
- Three types of literals
  1. **atomic**: predefined; basic data type values (e.g., **`short, float, boolean, char`**)
  2. **structured**: values that are constructed by type constructors (e.g., date, **`struct`** variables)
  3. **collection**: a collection (e.g., array) of values
  4. or objects

ODMG Interface Definition: An Example

- Note: interface is ODMG's keyword for class/type

```
interface Date:Object {
  enum
  weekday{sun,mon,tue,wed,thu,fri,sat};
  enum Month{jan,feb,mar,…,dec};
  unsigned short year();
  unsigned short month();
  unsigned short day();
  …
  boolean is_equal(in Date other_date);
};
```

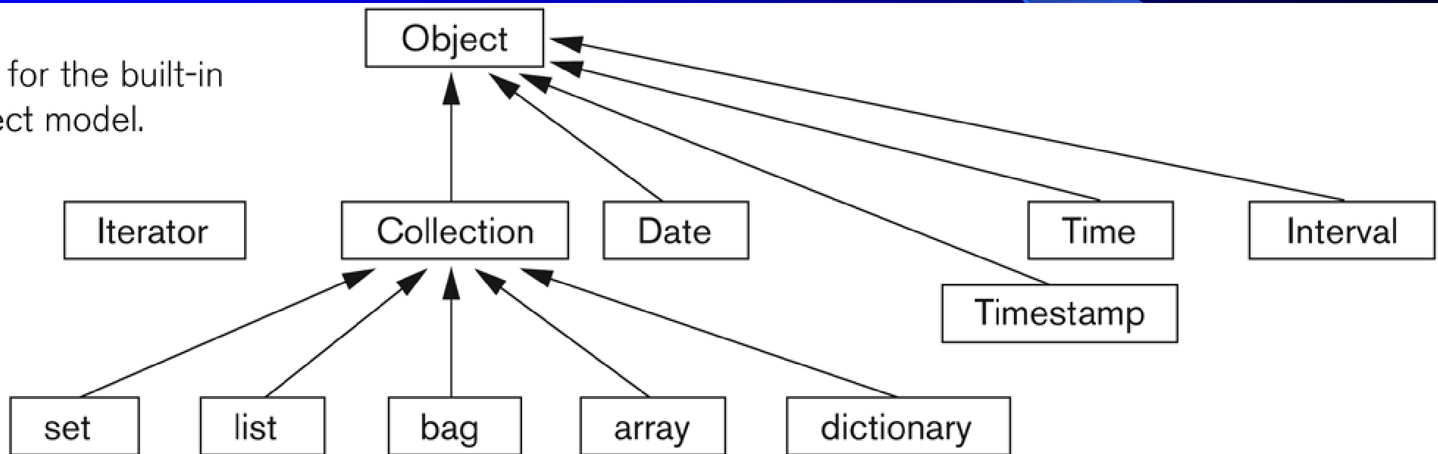
Built-in Interfaces for Collection Objects

- A **`collection`** object inherits the basic **`collection`** interface, for example:
  - **`cardinality()`**
  - **`is_empty()`**
  - **`insert_element()`**
  - **`remove_element()`**
  - **`contains_element()`**
  - **`create_iterator()`**

Collection Types

- Collection objects are further specialized into types like a set, list, bag, array, and dictionary
- Each collection type may provide additional interfaces, for example, a set provides:
  - **`create_union()`**
  - **`create_difference()`**
  - **`is_subset_of(`**
  - **`is_superset_of()`**
  - **`is_proper_subset_of()`**

Object Inheritance Hierarchy



Atomic Objects

- **Atomic objects** are user-defined objects and are defined via keyword **`class`**
- An example:

```
class Employee (extent all_emplyees key
  ssn) {
   attribute string name;
   attribute string ssn;
   attribute short age;
   relationship Dept works_for;
   void reassign(in string new_name);
}
```

Class Extents

- An ODMG object can have an **`extent`** defined via a class declaration
  - Each `extent` is given a name and will contain all persistent objects of that class
  - For `Employee` class, for example, the `extent` is called `all_employees`
  - This is similar to creating an object of type `Set<Employee>` and making it persistent

Class Key

- A class key consists of one or more unique attributes
- For the `Employee` class, the key is `ssn`
  - Thus each employee is expected to have a unique `ssn`
- Keys can be composite, e.g.,
  - `(`**`key`**` dnumber, dname)`

Object Factory

- An object factory is used to generate individual objects via its operations
- An example:

```
interface ObjectFactory {
  Object new ();
};
```

- **new()** returns new objects with an `object_id`
- One can create their own factory interface by inheriting the above interface

Interface and Class Definition

- ODMG supports two concepts for specifying object types:
  - **Interface**
  - **Class**
- There are similarities and differences between interfaces and classes
- Both have behaviors (operations) and state (attributes and relationships)

ODMG Interface

- An interface is a specification of the abstract behavior of an object type
  - State properties of an interface (i.e., its attributes and relationships) cannot be inherited from
  - Objects cannot be instantiated from an interface

ODMG Class

- A class is a specification of abstract behavior and **state** of an object type
  - A class is **Instantiable**
  - **Supports “extends”** inheritance to allow both state and behavior inheritance among classes
  - **Multiple inheritance** via “extends” is not allowed

21.2 Object Definition Language

- ODL supports semantics constructs of ODMG
- ODL is independent of any programming language
- ODL is used to create object specification (classes and interfaces)
- ODL is not used for database manipulation

ODL Examples (1) A Very Simple Class

- A very simple, straightforward class definition
  - (all examples are based on the university schema presented in Chapter 4):

```
class Degree {
   attribute string college;
   attribute string degree;
   attribute string year;
};
```

ODL Examples (2)
A Class With Key and Extent

- A class definition with “extent”, “key”, and more elaborate attributes; still relatively straightforward

```
class Person (extent persons key ssn) {
  attribute struct Pname {string fname …}
  name;
  attribute string ssn;
  attribute date birthdate;
  …
  short age();
}
```

ODL Examples (3)
A Class With Relationships

- Note extends (inheritance) relationship
- Also note “inverse” relationship

```
class Faculty extends Person (extent faculty)
  {
  attribute string rank;
  attribute float  salary;
  attribute string phone;
  …
  relationship Dept works_in inverse
  Dept::has_faculty;
  relationship set<GradStu> advises inverse
  GradStu::advisor;
  void give_raise (in float raise);
  void promote (in string new_rank);
  };
```

Inheritance via ":" – An Example

```
interface Shape {
   attribute struct point {…}
   reference_point;
   float perimeter ();
   …
};

class Triangle: Shape (extent triangles)
   {
   attribute short side_1;
   attribute short side_2;
   …
};
```

Object Query Language

- OQL is DMG’s query language
- OQL works closely with programming languages such as C++
- Embedded OQL statements return objects that are compatible with the type system of the host language
- OQL’s syntax is similar to SQL with additional features for objects

Simple OQL Queries

- Basic syntax: select…from…where…
  - SELECT d.name
  - FROM d in departments
  - WHERE d.college = 'Engineering';
- An **entry point** to the database is needed for each query
- An `extent` name (e.g., departments in the above example) may serve as an entry point

Iterator Variables

- Iterator variables are defined whenever a collection is referenced in an OQL query
- Iterator `d` in the previous example serves as an iterator and ranges over each object in the collection
- Syntactical options for specifying an iterator:
  - `d in departments`
  - `departments d`
  - `departments as d`

Data Type of Query Results

- The data type of a query result can be any type defined in the ODMG model
- A query does not have to follow the `select…from…where…` format
- A persistent name on its own can serve as a query whose result is a reference to the persistent object. For example,
  - departments; whose type is set<Departments>

Path Expressions

- A **path expression** is used to specify a path to attributes and objects in an entry point
- A path expression starts at a persistent object name (or its iterator variable)
- The name will be followed by zero or more dot connected relationship or attribute names
  - E.g., departments.chair;

Views as Named Objects

- The **define** keyword in OQL is used to specify an identifier for a **named query**
- The name should be unique; if not, the results will replace an existing named query
- Once a query definition is created, it will persist until deleted or redefined
- A view definition can include parameters

An Example of OQL View

- A view to include students in a department who have a minor:

```
define has_minor(dept_name) as
select   s
from     s in students
where
 s.minor_in.dname=dept_name
```

- has_minor can now be used in queries

Single Elements from Collections

- An OQL query returns a collection
- OQL's `element` operator can be used to return a single element from a singleton collection that contains one element:

```
element (select d from d in departments
where d.dname = 'Software Engineering');
```

- If `d` is empty or has more than one elements, an **exception** is raised

Collection Operators

- OQL supports a number of aggregate operators that can be applied to query results
- The aggregate operators and operate over a collection and include
  - `min, max, count, sum, avg`
- `count` returns an integer; others return the same type as the collection type

An Example of an OQL Aggregate Operator

- To compute the average GPA of all seniors majoring in Business:

```
avg (select s.gpa from s in
  students
  where s.class = 'senior' and
  s.majors_in.dname ='Business');
```

Membership and Quantification

- OQL provides membership and quantification operators:
  - (`e` **`in`** `c`) is true if `e` is in the collection `c`
  - (**`for all`** `e` **`in`** `c:` `b`) is true if all `e` elements of collection `c` satisfy `b`
  - (**`exists`** `e` **`in`** `c:` `b`) is true if at least one `e` in collection `c` satisfies `b`

An Example of Membership

- To retrieve the names of all students who completed CS101:

```
select s.name.fname s.name.lname
from    s in students
where   'CS101' in
  (select c.name
  from c
  in
  s.completed_sections.section.of_course);
```

Ordered Collections

- Collections that are lists or arrays allow retrieving their **first**, **last**, and **ith** elements
- OQL provides additional operators for extracting a sub-collection and concatenating two lists
- OQL also provides operators for ordering the results

An Example of Ordered Operation

- To retrieve the last name of the faculty member who earns the highest salary:

```
first (select struct
        (faculty: f.name.lastname,
            salary f.salary)
from f in faculty
ordered by f.salary desc);
```

Grouping Operator

- OQL also supports a grouping operator called **group by**
- To retrieve average GPA of majors in each department having >100 majors:

```
select deptname, avg_gpa:
  avg (select p.s.gpa from p in partition)
from s in students
group by deptname: s.majors_in.dname
having count (partition) > 100
```

4. C++ Language Binding

- C++ language binding specifies how ODL constructs are mapped to C++ statements and include:
  - a C++ class library
  - a Data Manipulation Language (ODL/OML)
  - a set of constructs called **physical pragmas** (to allow programmers some control over the physical storage concerns)

Class Library

- The class library added to C++ for the ODMG standards uses the prefix `d_` for class declarations
- `d_Ref<T>` is defined for each database class T
- To utilize ODMG's collection types, various templates are defined, e.g., `d_Object<T>` specifies the operations to be inherited by all objects

Template Classes

- A template class is provided for each type of ODMG collections:
  - `d_Set<T>`
  - `d_List<T>`
  - `d_Bag<t>`
  - `d_Varray<t>`
  - `d_Dictionary<T>`
- Thus a programmer can declare:

```
d_Set<d_Ref<Student>>
```

Data Types of Attributes

- The data types of ODMG database attributes are also available to the C++ programmers via the d_ prefix, e.g., `d_Short, d_Long, d_Float`
- Certain structured literals are also available, e.g., `d_Date, d_Time, d_Intreval`

Specifying Relationships

- To specify relationships, the prefix Rel_ is used within the prefix of type names
  - E.g., `d_Rel_Ref<Dept, has_majors> majors_in;`
- The C++ binding also allows the creation of extents via using the library class d_Extent:
  - `d_Extent<Person> All_Persons(CS101)`

Object Database Conceptual Design

- Object Database (ODB) vs. Relational Database (RDB)
  - Relationships are handled differently
  - Inheritance is handled differently
  - Operations in OBD are expressed early on since they are a part of the class specification

Relationships: ODB vs. RDB (1)

- Relationships in ODB:
  - relationships are handled by reference attributes that include OIDs of related objects
  - single and collection of references are allowed
  - references for binary relationships can be expressed in single direction or both directions via inverse operator

Relationships: ODB vs.. RDB (2)

- Relationships in RDB:
  - Relationships among tuples are specified by attributes with matching values (via **foreign keys**)
  - Foreign keys are single-valued
  - **M:N** relationships must be presented via a separate relation (table)

Inheritance Relationship in ODB vs. RDB

- Inheritance structures are built in ODB (and achieved via “:” and `extends` operators)
- RDB has no built-in support for inheritance relationships; there are several options for mapping inheritance relationships in an RDB (see Chapter 7)

Early Specification of Operations

- Another major difference between ODB and RDB is the specification of operations
  - ODB:
    - Operations specified during design (as part of class specification)
  - RDB:
    - Operations specification may be delayed until implementation

Mapping EER Schemas to ODB Schemas

- Mapping EER schemas into ODB schemas is relatively simple especially since ODB schemas provide support for inheritance relationships
- Once mapping has been completed, operations must be added to ODB schemas since EER schemas do not include an specification of operations

Mapping EER to ODB Schemas Step 1

- Create an ODL class for each EER entity type or subclass
  - Multi-valued attributes are declared by sets, bags or lists constructors
  - Composite attributes are mapped into tuple constructors

Mapping EER to ODB Schemas Step 2

- Add relationship properties or reference attributes for each binary relationship into the ODL classes participating in the relationship
  - Relationship cardinality: single-valued for 1:1 and N:1 directions; set-valued for 1:N and M:N directions
  - Relationship attributes: create via tuple **constructors**

Mapping EER to ODB Schemas Step 3

- Add appropriate operations for each class
  - Operations are not available from the EER schemas; original requirements must be reviewed
  - Corresponding **constructor** and **destructor** operations must also be added

Mapping EER to ODB Schemas Step 4

- Specify inheritance relationships via `extends` clause
  - An ODL class that corresponds to a sub-class in the EER schema inherits the types and methods of its super-class in the ODL schemas
  - Other attributes of a sub-class are added by following Steps 1-3

Mapping EER to ODB Schemas Step 5

- Map weak entity types in the same way as regular entities
  - Weak entities that do not participate in any relationships may alternatively be presented as **composite multi-valued attribute** of the owner entity type

Mapping EER to ODB Schemas Step 6

- Map categories (union types) to ODL
  - The process is not straightforward
  - May follow the same mapping used for EER-to-relational mapping:
    - Declare a class to represent the category
    - Define 1:1 relationships between the category and each of its super-classes

Mapping EER to ODB Schemas Step 7

- Map **n-ary relationships** whose degree is greater than 2
  - Each relationship is mapped into a separate class with appropriate reference to each participating class

Summary

- Proposed standards for object databases presented
- Various constructs and built-in types of the ODMG model presented
- ODL and OQL languages were presented
- An overview of the C++ language binding was given
- Conceptual design of object-oriented database discussed

References : Elmasri & Navathe “ Fundamentals of Database Systems”, Fifth edition