**Unit – 8 Transaction and Recovery**

**Transaction concepts:**

**Transaction processing systems** are systems with large databases and hundreds of concurrent users executing database transactions. Examples of such systems include airline reservations, banking, credit card processing, online retail purchasing, stock markets, supermarket checkouts, and many other applications. These systems

require high availability and fast response time for hundreds of concurrent users.

It is the atomic unit of work that is either completed in its entirety or not at all

It may also be defined as the unit or piece of program execution that accesses and possibly updates the various data items.

**Transaction Operations :-**

A **transaction** is an executing program that forms a logical unit of database processing.A transaction includes one or more database access operations—these can include insertion, deletion, modification, or retrieval operations. The database

operations that form a transaction can either be embedded within an application

program or they can be specified interactively via a high-level query language such

as SQL. One way of specifying the transaction boundaries is by specifying explicit

**begin transaction** and **end transaction** statements in an application program; in

this case, all database access operations between the two are considered as forming one transaction. A single application program may contain more than one transaction if it contains several transaction boundaries. If the database operations in a transaction do not update the database but only retrieve data, the transaction is

called a **read-only transaction**; otherwise it is known as a **read-write transaction**.

The *database model* that is used to present transaction processing concepts is quite

Simple A **database** is basically represented as a collection of *named data items.* The size of a data item is called its **granularity**. A **data item** can be a *database record*, but it can also be a larger unit such as a whole *disk block*, or even a smaller unit such as an individual *field (attribute) value* of some record in the database. The transaction processing concepts we discuss are independent of the data item granularity (size) and apply to data items in general. Each data item has a *unique name*, but this name is not typically used by the programmer; rather, it is just a means to *uniquely identify each data item*. For example, if the data item granularity is one disk block, then the disk block address can be used as the data item name. Using this simplified database model, the basic database access operations that a transaction can include are as follows:

■ **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*.

**Desirable Properties of Transactions :-**

Transactions should possess several properties, often called the **ACID** properties;

they should be enforced by the concurrency control and recovery methods of the

DBMS. The following are the ACID properties:

■ **Atomicity.** A transaction is an atomic unit of processing; it should either be

performed in its entirety or not performed at all.

■ **Consistency preservation.** A transaction should be consistency preserving,

meaning that if it is completely executed from beginning to end without

interference from other transactions, it should take the database from one

consistent state to another.

■ **Isolation.** A transaction should appear as though it is being executed in isolation

from other transactions, even though many transactions are executing concurrently. That is, the execution of a transaction should not be interfered

with by any other transactions executing concurrently.

■ **Durability or permanency.** The changes applied to the database by a committed

transaction must persist in the database. These changes must not be

lost because of any failure.

|  |
| --- |
|  |
| Transaction states |

The *atomicity property* requires that we execute a transaction to completion. It is the

responsibility of the *transaction recovery subsystem* of a DBMS to ensure atomicity.

If a transaction fails to complete for some reason, such as a system crash in the

midst of transaction execution, the recovery technique must undo any effects of the

transaction on the database. On the other hand, write operations of a committed

transaction must be eventually written to disk.

**Schedules (Histories) of Transactions**

A **schedule** (or **history**) *S* of *n* transactions *T*1, *T*2, ..., *Tn* is an ordering of the operations

of the transactions. Operations from different transactions can be interleaved in the schedule *S*. However, for each transaction *Ti* that participates in the schedule *S*, the operations of *Ti* in *S* must appear in the same order in which they occur in *Ti*. The order of operations in *S* is considered to be a *total ordering*, meaning *that for any two operations* in the schedule, one must occur before the other. It is possible theoretically to deal with schedules whose operations form *partial orders*), but we will assume for now total ordering of the operations in a schedule.

For the purpose of recovery and concurrency control, we are mainly interested in

the **read\_item** and **write\_item** operations of the transactions, as well as the commit and

abort operations. A shorthand notation for describing a schedule uses the symbols *b*,

*r*, *w*, *e*, *c*, and *a* for the operations begin\_transaction, read\_item, write\_item, end\_transaction,

commit, and abort, respectively, and appends as a *subscript* the transaction id

(transaction number) to each operation in the schedule.

For example, the schedule in Figure21.3(a), which we shall call *Sa*, can be written as follows in this notation:

*Sa*: *r*1(*X*); *r*2(*X*); *w*1(*X*); *r*1(*Y*); *w*2(*X*); *w*1(*Y*);

Similarly, the schedule for Figure 21.3(b),which we call *Sb*, can be written as follows,

if we assume that transaction *T*1 aborted after its read\_item(*Y*) operation:

*Sb*: *r*1(*X*); *w*1(*X*); *r*2(*X*); *w*2(*X*); *r*1(*Y*); *a*1;

Two operations in a schedule are said to **conflict** if they satisfy all three of the following conditions: (1) they belong to *different transactions*; (2) they access the *same item X*; and (3) *at least one* of the operations is a write\_item(*X*).

**Characterizing Schedules Based on Serializability**

In the previous section, we characterized schedules based on their recoverability

properties. Now we characterize the types of schedules that are always considered to be *correct* when concurrent transactions are executing. Such schedules are known as *serializable schedules*. Suppose that two users—for example, two airline reservations agents—submit to the DBMS transactions *T*1 and *T*2 in Figure 21.2 at approximately the same time. If no interleaving of operations is permitted, there are only two possible outcomes:

**1.** Execute all the operations of transaction *T*1 (in sequence) followed by all the

operations of transaction *T*2 (in sequence).

**2.** Execute all the operations of transaction *T*2 (in sequence) followed by all the

operations of transaction *T*1 (in sequence).

These two schedules—called *serial schedules*—are shown in Figure 21.5(a) and (b),

respectively. If interleaving of operations is allowed, there will be many possible

orders in which the system can execute the individual operations of the transactions.

Two possible schedules are shown in Figure 21.5(c). The concept of

**serializability of schedules** is used to identify which schedules are correct when

transaction executions have interleaving of their operations in the schedules

|  |
| --- |
|  |
|  |

**Serial and Nonserial,**

Schedules A and B in Figure 21.5(a) and (b) are called ***serial***because the operations of each transaction are executed consecutively, without any interleaved operations from the other transaction. In a serial schedule, entire transactions are performed in serial order: *T*1 and then *T*2 in Figure 21.5(a), and *T*2 and then *T*1 in Figure 21.5(b).

Schedules C and D in Figure 21.5(c) are called ***nonserial*** because each sequence

interleaves operations from the two transactions.

Formally, 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**. Therefore, in a serial schedule, only one transaction at a time is active—the commit (or abort) of the active transaction initiates execution of the next transaction. No interleaving occurs in a serial schedule. One reasonable assumption we can make, if we consider the transactions to be *independent*, is that *every serial schedule is considered correct*.We can assume this because every transaction is assumed to be correct if executed on its own. Hence, it does not matter which transaction is executed first. As long as every transaction is executed from beginning to end in isolation from the operations of other transactions, we get a correct end result on the database.

The problem with serial schedules is that they limit concurrency by prohibiting

interleaving of operations. In a serial schedule, if a transaction waits for an I/O

operation to complete, we cannot switch the CPU processor to another transaction,

thus wasting valuable CPU processing time. Additionally, if some transaction *T* is

quite long, the other transactions must wait for *T* to complete all its operations

before starting. Hence, serial schedules are *considered unacceptable* in practice.

However, if we can determine which other schedules are *equivalent* to a serial schedule, we can allow these schedules to occur.

**Why Concurrency Control Is Needed:**

If transactions are executed serially, i.e., sequentially with no overlap in time, no transaction concurrency exists. However, if concurrent transactions with interleaving operations are allowed in an uncontrolled manner, some unexpected, undesirable result may occur. Here are some typical **examples:**

1. **The Lost Update Problem.** This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database items incorrect. Suppose that transactions T1 and T2 are submitted at approximately the same time, and suppose that their operations are interleaved as shown in Figure 21.3(a); then the final value of item X is incorrect because T2 reads the value of X before T1 changes it in the database, and hence the updated value resulting from T1 is lost.

|  |
| --- |
|  |

1. **The Temporary Update (or Dirty Read) Problem.** This problem occurs when one transaction updates a database item and then the transaction fails for some. Meanwhile, the updated item is accessed (read) by another transaction before it is changed back to its original value. Example where T1 updates item X and then fails before completion, so the system must change X back to its original value. Before it can do so, however, transaction T2 reads the temporary value of X, which will not be recorded permanently in the database because of the failure of T1. The value of item X that is read by T2 is called dirty data.
2. **The Incorrect Summary Problem**. If one transaction is calculating an aggregate summary function on a number of database items while other transactions are updating some of these items, the aggregate function may calculate some values before they are updated and others after they are updated.

**Concurrency control lock based control**

**Two-Phase Locking Techniques for Concurrency Control**

Some of the main techniques used to control concurrent execution of transactions

are based on the concept of locking data items. A **lock** is a variable associated with a data item that describes the status of the item with respect to possible operations

that can be applied to it. Generally, there is one lock for each data item in the database. Locks are used as a means of synchronizing the access by concurrent transactions to the database items.

**1. Binary lock:-**

A binary lock can have two states i.e. locked and unlocked (1 for locked state and 0 for unlocked state). A transaction requests access to an item Q by first issuing LOCK (Q) operations. If LOCK (Q) =1, the transaction is forced to wait and if LOCK (Q) = 0, it is set to 1 and the transaction is allowed to access item Q. When transaction finishes with the operation, it finally issues unlock operation by setting LOCK (Q) to 0. The rule followed by binary locking scheme is as follows:

A transaction T must issue the operation lock (Q) before any read (Q) or write (Q) operations.

A transaction T must issue unlock (Q) after all read (Q) and write (Q) operations are completed.

A transaction T will not issue any lock (Q) if it already holds lock on item Q.

A transaction T will issue unlock (Q) if it already holds a lock on data item Q.

|  |
| --- |
|  |

**2. Shared/Exclusive or Read/Write lock:-**

Binary lock as discussed earlier is too restrictive for database items. But we should allow multiple transactions to access the same item if they all access Q for reading purpose only. Hence shared/exclusive lock is desired.

Shared: - if Ti holds shared mode lock on Q, it can only read and cannot write. Multiple shared mode lock can exist for the same data item Q.

Exclusive: - if Ti holds exclusive mode lock on Q, then it can both read as well as write. There can exist only one exclusive mode lock on a data item Q. Let A and B are the two arbitrary lock modes. If transaction Ti can be granted a lock on Q immediately inspire of the presence of the mode B lock on Q, then we say mode A is compatible with mode B than one shared mode lock can exist on the same data item.

To access a data item, transaction Ti must first lock that item. If data item is already locked in an incompatible mode, then the concurrency control manager do not grant the lock until all incompatible locks held by other transactions have been released. Thus Ti is made to wait until all incompatible locks held by other transactions have been released. Any transaction Ti may unlock the data item that it had locked in earlier point but unlocking data item immediately may not ensure serializability. For example consider the schedule as shown below(A=1000 & B=2000):

|  |
| --- |
|  |

As we can see that unlocking B too early has caused the database in inconsistent state. i.e. transaction T2 should display 3000 as a result but it is now showing 2950. Hence from here we can conclude that unlocking data items too early is not desirable. Hence delaying data unlocking can solve this problem. i.e.

|  |
| --- |
|  |

**Database Recovery:**

Recovery from transaction failures usually means that the database is *restored* to the most recent consistent state just before the time of failure. To do this, the system must keep information about the changes that were applied to data items by the various transactions. This information is typically kept in the **system log**.

A typical strategy for recovery may be summarized informally

as follows:

**1.** If there is extensive damage to a wide portion of the database due to catastrophic

failure, such as a disk crash, the recovery method restores a past copy of the database that was *backed up* to archival storage (typically tape or other large capacity offline storage media) and reconstructs a more current state by reapplying or *redoing* the operations of committed transactions from the *backed up* log, up to the time of failure.

**2.** When the database on disk is not physically damaged, and a noncatastrophic Failure. For example, a transaction that has updated some database items on disk but has not been committed needs to have its changes reversed by *undoing* its write operations. It may also be necessary to *redo* some operations in order to restore a consistent state of the database; for example, if a transaction has committed but some of its write operations have not yet

been written to disk. For noncatastrophic failure, the recovery protocol does not need a complete archival copy of the database. Rather, the entries kept in the online system log on disk are analyzed to determine the appropriate actions for recovery.

Conceptually, we can distinguish two main techniques for recovery from noncatastrophic

transaction failures: deferred update and immediate update. The **deferred update** techniques do not physically update the database on disk until *after* a transaction reaches its commit point; then the updates are recorded in the database. Before reaching commit, all transaction updates are recorded in the local transaction workspace or in the main memory buffers that the DBMS maintains (the DBMS main memory cache). Before commit, the updates are recorded persistently in the log, and then after commit, the updates are written to the database on disk. If a transaction fails before reaching its commit point, it will not have changed the database in any way, so UNDO is not needed. It may be necessary to REDO the effect of the operations of a committed transaction from the log, because their

effect may not yet have been recorded in the database on disk. Hence, deferred update is also known as the **NO-UNDO/REDO algorithm**.

In the **immediate update** techniques, the database *may be updated* by some operations

of a transaction *before* the transaction reaches its commit point. However, these operations must also be recorded in the log *on disk* by force-writing *before* they are applied to the database on disk, making recovery still possible. If a transaction fails after recording some changes in the database on disk but before reaching its commit point, the effect of its operations on the database must be undone; that is, the transaction must be rolled back. In the general case of immediate update, both *undo* and *redo* may be required during recovery. This technique, known as the **UNDO/REDO algorithm**.

# Concept of log-based recovery

The most widely used structure for recording database modifications is the *log*. The log is a sequence of *log records* and maintains a history of all update activities in the database. There are several types of log records.

An *update log record* describes a single database write:

* Transactions identifier.
* Data-item identifier.
* Old value.
* New value.

Whenever a transaction performs a write, it is essential that the log record for that write be created before the database is modified. Once a log record exists, we can output the modification that has already been output to the database. Also we have the ability to *undo* a modification that has already been output to the database, by using the old-value field in the log records.

For log records to be useful for recovery from system and disk failures, the log must reside on stable storage. However, since the log contains a complete record of all database activity, the volume of data stored in the log may become unreasonable large.

**Deferred Database Modification**

The deferred-modification technique ensures transaction atomicity by recording all database modifications in the log, but deferring all write operations of a transaction until the transaction partially commits (i.e., once the final action of the transaction has been executed). Then the information in the logs is used to execute the deferred writes. If the system crashes or if the transaction aborts, then the information in the logs is ignored.

**Write ahead loging**

(In general, the old value of the data item before updating is called the **before image**

**(BFIM**), and the new value after updating is called the **after image (AFIM**).)

When in-place updating is used, it is necessary to use a log for recovery . In this case, the recovery mechanism must ensure that the BFIM of the data item is recorded in the appropriate log entry and that the log entry is flushed to disk before the BFIM is overwritten with the AFIM in the database on disk. This process is generally known as **write-ahead logging**, and is necessary to be able to UNDO the operation if this is required during recovery. Before we can describe a protocol for write-ahead logging, we need to distinguish between two types of log entry information included for a write command: the information needed for UNDO and the information needed for REDO. A **REDO-type log entry** includes the **new value** (AFIM) of the item written by the operation since this is needed to *redo* the effect of the operation from the log (by setting the item value in the database on disk to its AFIM). The **UNDO-type log entries** include the **old value** (BFIM) of the item since this is needed to *undo* the effect of the operation from the log (by setting the item value in the database back to its BFIM). In an UNDO/REDO algorithm, both types of log entries are combined. Additionally, when cascading rollback is possible, read\_item entries in the log are considered to be UNDO-type entries .

**Checkpoints in the System Log:**

When a system failure occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone. Rather than reprocessing the entire log, which is time-consuming and much of it unnecessary, we can use *checkpoints*:

● Redoing/undoing all transactions recorded in the log can be very slow

☺ processing the entire log is time-consuming if the system has run for a long time

☺ we might unnecessarily redo transactions which have already output their updates to the database.

● Streamline recovery procedure by periodically performing check pointing

☺ Output all log records currently residing in main memory onto stable storage.

☺ Output all modified buffer blocks to the disk.

☺ Write a log record < checkpoint *L*> onto stable storage where *L* is a list of all transactions active at the time of checkpoint.

☺ All updates are stopped while doing check pointing

● During recovery we need to consider only the most recent transaction Ti that started before the checkpoint, and transactions that started after *Ti*.

☺ Scan backwards from end of log to find the most recent <checkpoint *L*> record

☺ Only transactions that are in *L* or started after the checkpoint need to be redone or undone

☺ Transactions that committed or aborted before the checkpoint already have all their updates output to stable storage.

● Some earlier part of the log may be needed for undo operations

☺ Continue scanning backwards till a record *<Ti* start> is found for every transaction *Ti* in *L*. Parts of log prior to earliest *<Ti* start> record above are not needed for recovery, and can be erased whenever desired

