Chapter 4

Transaction and Concurrency

LEARNING OBJECTIVES

- *Transactions and concurrency control*
- *Transaction*
- *Transaction properties*
- *Uncommitted data*
- *Transaction processing systems*
- *Concurrency control with locking methods*
- *Two-phase locking to ensure serializability*
- *Concurrency control with time stamping methods*
- *Concurrency control with optimistic methods*
- *Recoverability*
- *Equivalence of schedules*
- *Testing for conflict serializability*

Introduction

A transaction is a logical unit of work. It begins, with the execution of a BEGIN TRANSACTION operation, and ends with the execution of a COMMIT or ROLLBACK operation. The logical unit of work that is, a transaction does not necessarily involve just a single database operation. Rather, it involves a sequence of several such operations as follows:

1. Database updates are kept in buffers in main memory and not physically written to disk until the transaction commits. That way, if the transaction terminates unsuccessfully, there will be no need to undo any disk updates.
2. Database updates are physically written to disk as part of the process of honouring the transaction's COMMIT request. That way if the system subsequently crashes, we can be sure that there will be no need to redo any disk updates.

Transactions and Concurrency Control

Database transactions reflect real-world transactions that are triggered by events, such as buying a product, registering for a course, or making a deposit in your checking account. Transactions are likely to contain many parts, for example, a sales transaction consists of at least two parts.

UPDATE inventory by subtracting number of units sold from the PRODUCT table's available quantity on hand and UPDATE the ACCOUNTS RECEIVABLE table in order to bill the CUSTOMER. All parts of a transaction must be completed to prevent data integrity problems. Therefore, executing and managing transactions are important database system activities.

Concurrency control is the management of concurrent transactions execution. When many users are able to access the database, the number of concurrent transactions tends to grow rapidly; as a result, concurrency control is especially important in multiuser database environments.

Transaction

A transaction is a logical unit of work that must be either entirely completed or aborted, no intermediate states are acceptable, that is, multicomponent transactions like the previously mentioned sale, must not be partially completed. If you read from and/or write to (update) the database, you create a transaction. Another example is using SELECT, to generate a list of table contents. Many real-world database transactions are formed by two or more database requests. A database request is the equivalent of a single SQL statement in an application program or transaction. Each database request generates several input/output operations. A transaction that changes the contents of a database must alter the database from one consistent state to another. A consistent database state is one in which all data integrity constraints are satisfied.

Example:

1. Checking an account balance:
   SELECT ACC_NUM, ACC_BALANCE
   FROM CHECKACC
   WHERE ACC_NUM = '0908110638';
   Even though we did not make any changes to the CHECKACC table, the SQL code represents a transaction, because we accessed the database.

2. Registering a credit sale of 100 units of product *X* to customer *Y* in the amount of $500.00 first, product X's quantity on hand (QOH) needs to be reduced by 100.

   UPDATE PRODUCT
   SET PROD_QOH = PROD_QOH_100
   WHERE PROD_CODE = '*x*';
   Then, $500 needs to be added to customer Y's accounts receivable
   UPDATE ACCT_RECEIVABLE
   SET ACCT_RECEIVABLE = ACCT_BALANCE + 500
   WHERE ACCT_NUM = '*Y*';

In Example 2, both the SQL, transactions must be completed in order to represent the real-world sales transaction. If both transactions are not completely executed, the transaction yields an inconsistent database.

If a transaction yields an inconsistent database, the DBMS must be able to recover the database to a previous consistent state.

Transaction Properties

All transactions must display atomicity, consistency, isolation and durability. These are known as ACID properties of transactions.

Atomicity

It requires that all operations of a transaction be completed; if not, the transaction is aborted. Therefore, a transaction is treated as a single, logical unit of work.

Consistency

It describes the result of the concurrent execution of several transactions. The concurrent transactions are treated as though they were executed in serial order. This property is important in multiuser and distributed database, where several transactions are likely to be executed concurrently.

Isolation

It means that the data used during the execution of a transaction cannot be used by a second transaction until the first one is completed. Therefore, if a transaction T_1 is being executed and is using the data item X_1, that data item cannot be accessed by any other transaction (T_2… T_n) until T_1 ends. This property is particularly useful in multiuser database environment, because several different users can access and update the database at the same time.

Durability

It indicates the permanence of the database's consistent state. When a transaction is completed, the database reaches a consistent state, and that state cannot be lost, even in the event of the system's failure.

Transaction Management with SQL

The ISO standard defines a transaction model based on two SQL statements: COMMIT and ROLLBACK. The standard specifies that an SQL transaction automatically begins with a transaction-initiating SQL statement executed by a user or program (e.g., SELECT, INSERT, UPDATE). Changes made by a transaction are not visible to other concurrently executing transactions until the transaction completes. When a transaction sequence is initiated, it must continue through all succeeding SQL, statements until one of the following four events occur:

1. A COMMIT statement ends the transaction successfully, making the database changes permanent.
   A new transaction starts after COMMIT with the next transaction initiating statement.
2. For programmatic SQL, successful program termination ends the final transaction successfully, even if a commit statement has not been executed (equivalent to COMMIT)
3. For programmatic SQL, abnormal program termination aborts the transaction (equivalent to ROLLBACK)

Example:

```
UPDATE PRODUCT
SET PROD_QOH = PROD_QOH_100
WHERE PROD_CODE = '345TYX';

UPDATE ACCREC
SET AR_BALANCE = AR_BALANCE + 3500
WHERE AR_NUM = '60120010';
COMMIT;
```

Concurrency Control

The coordination of simultaneous execution of transactions in a multiprocessing database system is known as *concurrency control*. The objective of concurrency control is to ensure the serializability of transaction in a multiuser database environment. Concurrency is important, because the simultaneous execution of transactions over a shared database can create several data integrity and consistency problems. Three main problems are lost updates, uncommitted data and inconsistent retrievals.

Lost Updates

Consider the following two concurrent transactions where PROD_QOH represents a particular PRODUCT's quantity on hand. (PROD_QOH is an attribute in the Product table)

Assume the current PROD_QOH value for the product concerned is 35.

Table 1

Transaction	Computation
T_1: purchase 100 units	PROD_QOH = PROD_QOH + 100
T_2: sell 30 units	PROD_QOH = PROD_QOH – 30

Table 1 shows the serial execution of these transactions under normal circumstances, yielding the correct answer: PROD_QOH = 105. But suppose that a transaction is able to read a product's PROD_QOH value from the table before a previous transaction (using the same product) has been committed. The sequence depicted in Table 2 shows how the cost update problem can arise. Note that the first transaction (T_1) has not yet been committed when the second transaction (T_2) is executed. Therefore T_2 still operates on the value 135 to disk which is promptly over written by T_2. As a result, the addition of 100 units is "lost" during the process.

Uncommitted Data

Data are not committed when two transactions, T_1 and T_2 are executed concurrently and the first transaction (T_1) is rolled back after the second transaction (T_2) has already accessed the uncommitted data, thus violating the isolation property of transaction. Consider the same transactions from T_1 and T_2, from above. However, this time T_1 is rolled back to eliminate the addition of the 100 units. Because T_2 subtracts 30 from the original 35 units, the correct answer should be 5.

Table 2

	Computation
T_1: purchase 100 units	PROD_QOH = PROD_QOH + 100 (Rolled Back)
T_2: sell 30 units	PROD_QOH = PROD_QOH_30

Table 2 shows how, under normal circumstances, the serial execution of these transactions yield the correct answer. The uncommitted data problem can arise when the ROLLBACK is completed after T_2 has begun its execution.

Inconsistent Retrievals

Inconsistent retrievals occur when a transaction calculates some summary (aggregate) functions over a set of data, while other transactions are updating the data. The problem is that the transaction might read some data before they are changed and other data after they are changed, thereby yielding inconsistent results.

Example:

1. T_1 calculates the total PROD_QOH of the products stored in the PRODUCT table
2. At the same time, T_2 updates the PROD_QOH for two of the PRODUCT table's products (T_2 represents the correction of a typing error: the user added 30 units to product 345TYX's PROD_QOH but meant to add the 30 units to 125TYZ's PROD_QOH to correct the problem, the user subtracts 30 from product 345TYX's PROD_QOH and adds 30 to product 125TYZ's PROD_QOH).

The computed answer 485 is obviously wrong, because we know the correct answer to be 455.

Transaction Processing Systems

Transaction processing systems are systems with large databases and hundreds of concurrent users that are executing database transactions. For example, banking, credit card processing, stock markets, supermarket checkout, etc.

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

1. A transaction includes one or more database access operations. These can include insertion, deletion, modification, or retrieval operations.
2. *Basic operations*: 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.
   - Write_item (*X*): Writes the value of program variable *X* into the database item named *X*

Executing a 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 in main memory buffer).
3. Copy item *X* from the buffer to the program variable named x

Executing a 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 in 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.

Step 4 actually updates the database on disk.

The decision about when to store back a modified disk block that is in a main memory buffer is handled by the recovery manager of the DBMS in cooperation with the underlying operating system.

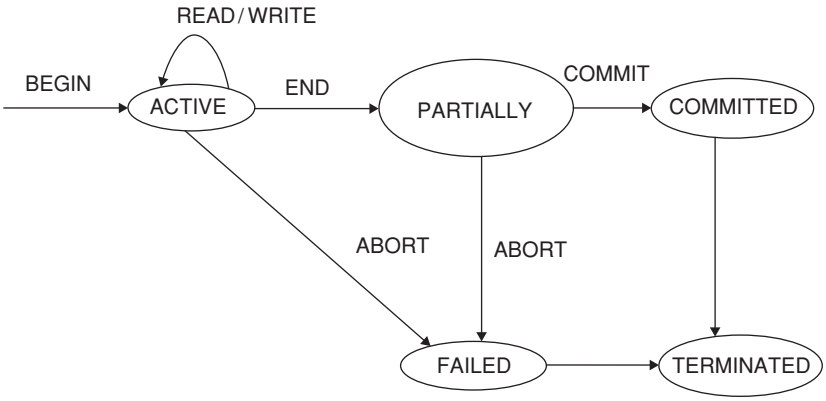


Figure 1 Transactions execution state transition diagram.

For the purpose of recovery, the system needs to keep track of when the transaction starts, terminates, and commits or aborts. Hence, the recovery manager keeps track of the following operations:

1. BEGIN_TRANSACTION: It shows the beginning of Execution of a transaction.
2. READ/WRITE: These specify read or write operations on the database items.
3. END_TRANSACTION: This specifies that READ and WRITE operations have ended and marks the end of transaction execution.
4. COMMIT_TRANSACTION: This shows a successful end of the transaction so that any changes executed by the transaction can be safely committed to the database and will not be undone.
5. ROLL BACK OR ABORT: This shows that the transaction has ended unsuccessfully, so that any changes or effects that the transaction may have applied to the database must be undone.
6. ACTIVE STATE: A transaction goes into an active state immediately after it starts execution where it can issue READ and WRITE operations.
7. PARTIALLY COMMITTED: When the transaction ends, it moves to the partially committed state
8. COMMIT: A transaction 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 have 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.
9. FAILED_STATE: A transaction can go to the failed state if the transaction is aborted during its active state. The transaction may then have to be rolled back to undo the effect of its WRITE operations on the database.
10. TERMINATIED: The terminated state corresponds to the transactions leaving the system.

Concurrency Control with Locking Methods

A lock guarantees exclusive use of a data item to a transaction. In general, if transaction T_1 holds a lock on a data item (e.g., an employee's salary) then transaction T_2 does not have access to that data item. A transaction acquires a lock prior to data access; the lock is released (unlocked) when the transaction is completed, so that another transaction can lock the data item for its exclusive use. All lock information is managed by a lock manager, which is responsible for assigning and policing the locks used by the transactions.

Lock Granularity

Lock granularity indicates the level of lock use. Locking can take place at the following levels: database level, table level, page level, row level and field (or attribute) level.

Database Level

In a database-level lock, the entire database is locked, thus preventing the use of any tables in the database by transaction T_2 while transaction T_1 is being executed. Transaction T_1 and T_2 cannot access the database concurrently, even if they use different tables. This level of locking is suitable for batch processes, but it is not unsuitable for online multiuser DBMSs.

Table Level

In a table-level lock, the entire table is locked, preventing access to any row by transaction T_2 while transaction T_1 is using Table 2 transactions can access the same database, as long as they access different tables. Transactions T_1 and T_2 cannot access the same table even if they try to use different rows, T_2 must wait until T_1 unlocks the table.

Page level

In a page level lock, the DBMS will lock an entire disk page (a disk page or page is the equivalent of a disk block, which can

be described as a (referenced) section of a disk). Transactions T_1 and T_2 access the same table while locking different disk pages. If T_2 requires the use of a row located on a page that is locked by T_1, T_2 must wait until the page is unlocked by T_1.

Row level

The row-level lock is much less restrictive than the locks discussed earlier. The DBMS allows concurrent transactions to access different rows of the same table, even if the rows are located on the same page. A lock exists for each row in each table of the database.

Field level

The field-level lock allows concurrent transactions to access same row as long as they require the use of different fields (attributes) within the row. Although, field-level locking clearly yields the most flexible multi user data access, it requires a high level of computer over head.

Lock Types

1. Binary locks
2. Shared/Exclusive locks

Binary Locks

A binary lock has only two states: locked (1) or unlocked (0). If an object, that is, a database, table, page, or row is locked by a transaction, no other transaction can use that object. If an object is unlocked, any transaction can lock the object for its use. As a rule, a transaction must unlock the object after its termination. Every database operation requires that the affected object be locked. Therefore, every transaction requires a lock and unlock operation for each data item that is accessed. Such operations are automatically scheduled by the DBMS, the user need not concerned about locking or unlocking data items. Binary locks are now considered too restrictive to yield optimal concurrency conditions. For example if two transaction want to read the same database object, the DBMS will not allow this to happen, even though neither transaction updates the database (and therefore, no concurrency problems can occur) concurrency conflicts occur only when two transactions execute concurrently and one of them updates the database.

Shared/Exclusive locks

The tables "shared' and "exclusive" indicate the nature of the lock. The following table comparatively explains both locks.

Exclusive Locks	Shared Locks
An *exclusive lock* exists when access is specifically reserved for the transaction that locked the object.	A *shared lock* exists when concurrent transactions are granted READ access on the basis of a common lock.
The exclusive lock must be used when the potential for conflict exists.	A shared lock produces no conflict as long as the concurrent transactions are read only.
(An exclusive lock is issued when a transaction wants to write (update) a data item and no locks are currently held on that data item by any other transaction.	A shared lock is issued when a transaction wants to read data from the database and no exclusive lock is held on that data item.

Using the shared/exclusive locking concept, a lock can have three states: unlocked, shared (READ) and exclusive (WRITE). 2 READ transactions can be safely executed and shared locks allow several READ transactions to concurrently read the same data item. For example, if transaction T_1 has a shared lock on data item X, and transaction T_2 wants to read data item X, T_2 may also obtain a shared lock on data item X.

If transaction T_2 updates data item X, then an exclusive lock is required by T_2 over data item X. The exclusive lock is granted if and only if no other locks are held on the data item. Therefore, if a shared or exclusive lock is already held on data item X by transaction T_1, an exclusive lock cannot be granted to transaction T_2.

Potential problems with locks

Although locks prevent serious data inconsistencies, their use may lead to two major problems:

1. The resulting transaction schedule may not be serializable.
2. The schedule may create deadlocks. Database *deadlocks* are the equivalent of a traffic gridlock in a big city and are caused when *two transactions wait for each other to unlock data.*

Both problems can be solved. Serializability is guaranteed through a locking protocol known as two-phase locking and deadlocks can be eliminated by using deadlock detection, and prevention techniques. We shall examine these techniques next.

Two-phase Locking to Ensure Serializability

The *two-phase locking protocol* defines how transactions acquire and relinquish locks. It guarantees serializability, but it does NOT prevent deadlocks. The two phases are as follows:

1. A *growing phase*, in which a transaction acquires all the required locks without unlocking any data. Once all locks have been acquired, the transaction is in its locked point.
2. A *shrinking phase*, in which a transaction releases all locks and cannot obtain any new lock.

The two-phase locking protocol is governed by the following rules"

1. Two transactions cannot have conflicting locks.
2. No unlock operation can precede a lock operation in the same transaction.
3. No data are affected until all locks are obtained, that is, until the transaction is in its locked point.

Deadlocks

Deadlocks exist when two transactions, T_1 and T_2, exist in the following mode:

1. T_1 would like to access data item X and then data item Y. (So far, T_1 has locked data item X and T_1 is in progress, it will eventually require to lock data item Y.)
2. T_2 needs to access data items X and Y, to begin. (So far, T_2 has locked data item Y.)

If T_1 has not unlocked data item X, T_2 cannot begin; if T_2 has not unlocked data item Y, T_1 cannot continue. Consequently, T_1 and T_2 wait indefinitely, each waiting for the other to unlock the required data item. Such in a real-world DBMS, many transactions can be executed simultaneously, thereby increasing the probability of generating deadlocks. Note that deadlocks are possible only if one of the transactions wants to obtain an exclusive lock on a data item; no deadlock condition can exist among shared locks.

Three basic techniques exist to control deadlocks:

1. *Deadlock prevention*: A transaction requesting a new lock is aborted if there is a possibility that a deadlock can occur. If the transaction is aborted, all the changes made by this transaction are ROLLED BACK, and all locks obtained by the transaction are released. The transaction is then rescheduled for execution. Deadlock prevention works because it avoids the conditions that lead to deadlocking.
2. *Deadlock detection*: The DBMS periodically tests the database for deadlocks. If a deadlock is found, one of the transactions (the "victim") is aborted (ROLLED BACK and restarted), and the other transaction continues.
3. *Deadlock avoidance*: The transaction must obtain all the locks it needs before it can be executed.

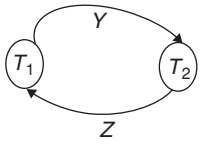
The best deadlock control method depends on the database environment. For example, if the probability of deadlocks is low, deadlock detection is recommended. However, if the probability of deadlocks is high, deadlock prevention is recommended. If response time is not high on the system priority list, deadlock avoidance might be employed.

Deadlock occurs when each transaction T in a set of two or more tractions is waiting for some item that is locked by some other transaction T^1 in the set. Each transaction in the set is on a waiting queue, waiting for one of the other transactions in the set to release the lock on an item

Example:

T_1	T_2
Write lock (z)	
	Read lock (z)
	Read lock (y)
Write lock (y)	

Transaction T_1 is waiting for Y which is locked by Transaction T_2 and transaction T_2 is waiting for z which is locked by transaction T_1. The below graph is called *wait for graph*.



Deadlock Prevention

There are number of deadlock prevention schemes that make a decision about what to do with a transaction involved in a possible deadlock situation:

1. Should it be blocked and made to wait
2. Should it be aborted
3. Should the transaction pre-empt and abort another transaction.

Concurrency Control with Time Stamping Methods

The *time stamping* approach to scheduling concurrent transactions assigns a global unique time stamp to each transaction. The time stamp value produces an explicit order in which transactions are submitted to the DBMS. Time stamps must have two properties: uniqueness and monotonicity. *Uniqueness* ensures that no equal time stamp values can exist, and *monotonicity* ensures that time increases.

All database operations (READ and WRITE) within the same transaction must have the same time stamp. The DBMS executes conflicting operations in time stamp order, thereby ensuring serializability of the transactions. If two transactions conflict, one often is stopped, rescheduled, and assigned a new time stamp value.

The concept of transaction time stamp $TS(T)$, which is a unique identifier assigned to each transaction. The time stamps are based on the order in which transactions start. If transaction T_1 starts before transaction T_2, then $TS(T_1) < TS(T_2)$

1. Older transaction will have the smaller time stamp value
2. Two schemes that prevent deadlock are
   - Wait-die
   - Wound-wait

Suppose that transaction T_k tries to lock an item x but is not able to because x is locked by some other transaction T_L with a conflicting lock.

The rules followed by these schemes are as follows:

1. Wait-die: If T_s (T_k) < T_s (T_L), then (T_k older than T_L) T_k is allowed to wait; otherwise (T_k younger than T_L) abort T_k and restart it later with the same time stamp
2. Wound-wait: If T_s (T_k) < T_s (T_L) then (T_k older than T_L) abort T_L and restart it later with the same time stamp; otherwise (T_k younger than T_L) T_k is allowed to wait

Both schemes end up aborting the younger of the two transactions that may be involved in a deadlock

Concurrency Control with Optimistic Methods

Optimistic methods are based on the assumption that the majority of the database operations do not conflict. A transaction is executed without restrictions until it is committed. Each transaction moves through *two* or *three* phases:

Read phase: The transaction reads the database, executes the needed computations, and makes the updates to a private copy of the database values.

Validation phase: The transaction is validated to assure that the changes made will not affect the integrity and consistency of the database.

If the validation test is positive, transaction goes to the Write Phase.

If the validation test is negative, transaction is restarted, and changes are discarded.

Write phase: The changes are permanently applied to the database.

Serializability

Serializabilty is accepted as 'criterion for correctness' for the interleaved execution of a set of transactions, such an execution is considered to be correct if and only if it is serializable.

1. A set of transactions is serializable if and only if it is equivalent to some serial execution of the same transactions
2. A serial execution is one in which the transactions are run one at a time in some sequence

Schedule: Given a set of transactions, any execution of those transactions interleaved or otherwise is called a *schedule*.

1. Executing the transactions one at a time, with no interleaving constitutes a serial schedule. A schedule that is not serial is an interleaved schedule (or) non-serial schedule.
2. Two schedules are said to be equivalent if and only if they are guaranteed to produce the same result as each other. Thus, a schedule is serializable, and correct, if and only if it is equivalent to some serial schedule.

Two-phase Locking Theorem

If all transactions obey the two phase locking protocol, then all possible interleaved schedule are serializable.

1. Before operating on any object (it could be a database tuple), a transaction must acquire a lock on the object
2. After releasing a lock, a transaction must never go on to acquire any more locks.

A transaction that obeys this protocol thus has two phases: a lock acquisition or "growing phase and a lock releasing or "shrinking" phase

Let '*I*' be an interleaved schedule involving some set of transactions T, T_2, T_3, … T_n.

If '*I*' is serializable, then there exists at least one serial schedule '*S*' involving T_1, T_2, … T_n such that '*I*' is equivalent to '*S*' is said to be a serialization of '*I*'

Let T_i and T_j be any two distinct transactions in the set T_1, T_2, T_3, … T_n. Let T_i precede T_j in the serialization '*S*'. In the interleaved schedule *I*, then the effect must be as if T_i really did execute before T_j. In other words, if *A* and *B* are any two transactions involved is some serializable schedule, then either A logically precedes *B* or *B* logically precedes *A* in that schedule, that is, either *B* can see A's output or *A* can see B's. If the effect is not as if either A ran before *B* or *B* ran before A, then the schedule is not serializable and not correct.

1. A schedule '*S*' of '*n*' transactions T_1, T_2 … T_n is an ordering of the operations of the transactions subject to the constraint that, for each transaction T_i that participates in '*S*', the same order in which they occur in T_i.
2. 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, '*R*', '*W*', '*C*' and 'A' for the operations read item, write item, commit, and abort respectively, and appends as subscript the transition-id (transaction number) to each operation in the schedule

Example: The schedule of the given set of transactions can be written as follows:

T_1	T_2
Read item (*x*);	
X = *X* – *N*;	
	Read item (*x*);
	X = *X* + *M*;
Write item (*x*)	
Read item (*y*)	
	Write item (*x*)
	Write item (*y*)
Y = *Y* + *N*	
Write item (*y*)	Commit

Schedule:

$$S: R_1(X); R_2(X); W_1(X); R_1(Y); W_2(X); W_2(Y); W_1(Y); C_2$$

Conflicts: Two operations in a schedule are said to have conflict if they satisfy all three conditions, if

1. they belong to different transactions
2. they access the same data item
3. at least one of the operations is a write item

For example, In the schedule '*S*' given above, the operations $r_1(x)$ and $w_2(x)$ conflict, as do

The operations $r_2(x)$ and $w_1(x)$ and the operations $w_1(x)$ and $w_2(x)$. However the operations $r_1(x)$ and $r_2(x)$ do not conflict, since they are both read operations;

The operations $w_1(x)$ and $w_2(y)$ do not conflict because they operate on distinct data items x and y. The operations $r_1(x)$ and $w_1(x)$ do not conflict, because they belong to the same transaction.

Complete schedule: A schedule *S* of '*n*' transactions T_1, T_2, T_3 … T_n is said to be a complete schedule if the following conditions hold.

1. The operations is '*S*' are exactly those operations in T_1, T_2, … T_n including a commit or abort operation as the last operation for each transaction in the schedule.
2. For any pair of operations from the same transaction T_i, their order of appearance in '*S*' is the same as their order of appearance in T_i
3. For any two conflicting operations, one of the two must occur before the other in the schedule.

The preceding condition (3) allows for two non-conflicting operations to occur in the schedule without defining which occurs first, thus leading to the definition of a schedule as a partial order of the operations in the '*n*' transactions.

It is difficult to encounter complete schedules in a transaction processing system, because new transactions are continually being submitted to the system. Hence, it is useful to define the concept of the 'committed projection *C*(*S*) of schedule *S*; which include only the operations in *S* that belong to committed transactions, that is, transaction T_i whose commit operation is C_i.

Recoverability

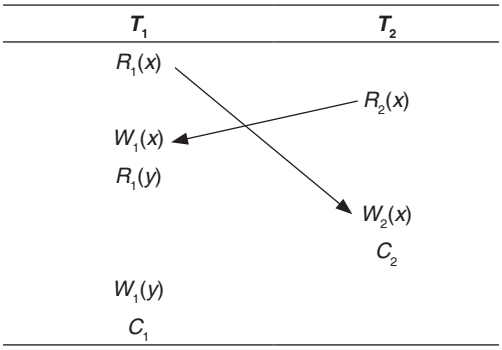
Recoverability ensures that once a transaction *T* is committed, it should never be necessary to roll back *T*. The schedules that theoretically meet this criterion are called *recoverable schedules* and those that do not are called *non-recoverable*, and hence should not be permitted

A schedule '*S*' is recoverable if no transaction *T* in '*S*' commits until all transactions T^1 that have written an item that *T* reads have committed

A transaction *T* reads from transaction T^1 in a schedule *S* if some item *x* is first written by T^1 and later read by *T*. In addition, T^1 should not have been aborted before *T* reads item *x*, and there should be no transactions that write *x* after T^1 writes it and before *T* reads it (unless those transactions, if any, have aborted before *T* reads *x*)

Example:
Consider the given schedule, check whether it is recoverable or not:
S: $R_1(X); R_2(X); W_1(X); R_1(Y); W_2(X); C_2, W_1(Y); C_1$?

Solution:
The given schedule can also be represented as follows:



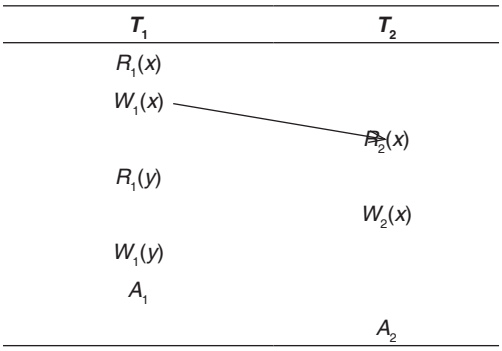
There are two *WR* conflicts, if the schedule consists of *RW* conflict, then we may say that the schedule is not recoverable (if the transaction which is performing read operation commits first)

Cascadeless Schedule

In a recoverable schedule, no committed transaction ever needs to be rolled back. It is possible for a phenomenon known as cascading rollback to occur, when an uncommitted transaction has to be rolled back because it read an item from a transaction that failed.

This is illustrated in the following schedule:

Example:
S: $R_1(X); W_1(X); R_2(X); R_1(Y); W_2(X); W_1(Y); A_1, A_2$
The above schedule is represented as follows:



Transaction T_2 has to be rolled back because it reads item *x* from T_1, and T_1 is then aborted, because cascading rollback

can be quite time consuming since numerous transactions can be rolled back. It is important to characterize the schedules where this phenomenon is guaranteed not to occur.

A scheduled is said to be cascadeless if every transaction in the schedule reads only items that were written by committed transaction. In this case, all items read will not be discarded, so no cascading rollback will occur.

Strict Schedule

A schedule is called *strict schedule*, in which transactions can neither read nor write an item x until the last transaction that wrote x has committed or aborted

1. All strict schedules are cascadeless
2. All cascadeless schedules are recoverable

Equivalence of Schedules

There are several ways to define equivalence of schedules as follows:

1. Result equivalent
2. Conflict equivalent
3. View equivalent

Result Equivalent

Two schedules are called *result equivalent* if they produce the same final state of the database. However, two different schedules may accidentally produce the same final state.

Example: Check whether the two schedules are result equivalent or not:

S_1	S_2
Read item (x);	Read item (x)
$X = X + 20$;	$X = X * 1.1$;
Write item (x);	Write item (x);

Solution:

Schedules S_1 and S_2 will produce the same final database state if they execute on a database with an initial value of $x = 200$; but for other initial values of x, the schedules are not result equivalent

For two schedules to be equivalent, the operations applied to each data item affected by the schedules should be applied to that item in both schedules in the same order. The other two definitions of equivalence of schedules generally used are conflict equivalence and view equivalence

Conflict Equivalence

Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules. If two conflicting operations are applied in different orders in two schedules, the effect can be different on the database or on other transactions in the schedule, and hence the schedules are not conflict equivalent.

Example:

S_1	
$r_1(x)$	
	$W_2(x)$

S_3	
$W_1(x)$	
	$W_2(x)$

S_2	
	$W_2(x)$
$r_1(x)$	

S_4	
	$W_2(x)$
$W_1(x)$	

The value read by $r_1(x)$ can be different in the two schedules. Similarly, if two write operations occur in the order $w_1(x)$, $w_2(x)$ in s_3, and in the reverse order $w_2(x)$, $w_1(x)$ in s_4, the next $r(x)$ operation in the two schedules will read potentially different values.

Testing for Conflict Serializability

The following algorithm can be used to test a schedule for conflict serializability. The algorithm takes read item and write item operations in a schedule to construct a precedence graph or serialization graph, which is a directed graph $G(N, E)$ here N is a set of Nodes $N = \{T_1, T_2, \ldots. T_n\}$ and E is a set of directed edges $E = \{e_1, e_2, \ldots.. E_m\}$ There is one node in the graph for each transaction. T_i in the schedule. Each edge e_i in the graph is of the form $(T_j \rightarrow T_k)$, $1 \text{″} j \text{″} n$, $1 \text{″} k \text{″} n$,

Where T_j is the starting node of e_i and T_k is the ending node of e_i.

Edge is created if one of the operations in T_j appears in the schedule before some conflicting operation in T_k

Algorithm

1. For each transaction T_i participating in schedule S, create a node labelled T_i in the precedence graph
2. For each case in S where T_j executes a read item (x) after T_i executes a write item (x), create an edge $(T_i \rightarrow T_j)$ in the graph
3. For each case in S where T_j executes a write item (x) after T_i executes a read item (x), create an edge $(T_i \rightarrow T_j)$ in the graph
4. For each case in S where T_j executes a write item (x) after T_i executes a write item (x), create an edge $(T_i \rightarrow T_j)$ in the precedence graph
5. The schedule S is serializable, if the precedence graph contains no cycles

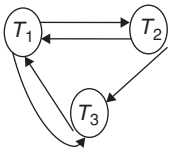
A cycle in a directed graph is a sequence of edges $C = ((T_j \rightarrow T_k), (T_k \rightarrow T_p), \ldots (T_i \rightarrow T_j))$

With the property that the starting node of each edge, except the first edge is the same as the ending node of the previous edge, and the starting node of the first edge is the same as the ending node of the last edge.

Example: Check whether the given schedule is conflict serializable or not by drawing precedence graph:

T_1	T_2	T_3
$R_1(x)$		
	$W_2(x)$	
		$R_3(x)$
$W_1(x)$		
		$W_3(x)$
$R_1(x)$		

Solution:
First identify the conflicts:
$(T_1 \rightarrow T_2)$ WR conflict
$(T_2 \rightarrow T_1)$ WW conflict
$(T_2 \rightarrow T_3)$ RW conflict
$(T_3 \rightarrow T_1)$ WR conflict
$(T_1 \rightarrow T_3)$ WW conflict
Take transactions as nodes in the precedence graph:



The precedence graph has cycle, which says that the schedule is not serializable.

View Equivalence and View Serializability

View equivalence is less restrictive compared to conflict equivalence. Two schedules S and S' are said to be view equivalent if the following three conditions hold:

1. The same set of transactions participate in S and S', and S and S' include the same operations of those transactions
2. For any operation $r_i(x)$ of T_i in S, if the value of x read by the operation has been written by an operation $w_j(x)$ of T_j, the same condition must hold for the value of x read by operation $r_i(x)$ of T_j in S'
3. If the operation $w_k(y)$ of T_k is the last operation to write Y in S, then $W_k(y)$ of T_k must also be the last operation to write item Y in S'

The idea behind view equivalence is that 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 result. Hence the read operation is said to see the same view in both schedules:

1. A schedule S is said to be view serializable if it is view equivalent to a serial schedule.
2. All conflict serializable schedules are view serializable, but vice versa is not true.

Exercises

Practice Problem 1

Directions for questions 1 to 20: Select the correct alternative from the given choices.

1. Consider the given schedules S_1 and S_2
S_1: $r_1(x), r_1(y), r_2(x), r_2(y), w_2(y), w_1(x)$
S_2: $r_1(x), r_2(x), r_2(y), w_2(y), r_1(y), w_1(x)$
Which schedule is conflict serializable?
(A) S_1 (B) S_2
(C) S_1 and S_2 (D) None of these

2. Consider the given schedule with three transactions T_1, T_2 and T_3:

T_1	T_2	T_3
$r_1(x)$		
	$r_2(y)$	
		$r_3(y)$
	$w_2(y)$	
$w_1(x)$		
		$w_3(x)$
	$r_2(x)$	
	$w_2(x)$	

Which of the following is correct serialization?
(A) $T_2 \rightarrow T_1 \rightarrow T_3$ (B) $T_1 \rightarrow T_3 \rightarrow T_2$
(C) $T_3 \rightarrow T_1 \rightarrow T_2$ (D) None of these

3. Consider the three data items D_1, D_2 and D_3 and the following execution of schedules of transactions T_1, T_2 and T_3:

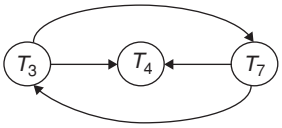
T_1	T_2	T_3
	$R(D_2)$	
	$R(D_2)$	
	$W(D_2)$	
		$R(D_2)$
		$R(D_3)$
$R(D_1)$		
$W(D_1)$		
		$W(D_2)$
		$W(D_3)$
	$R(D_1)$	
$R(D_2)$		
$W(D_2)$		
	$W(D_1)$	

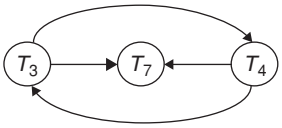
Which of the following is true?
(A) The schedule is conflict serializable
(B) The schedule is not conflict serializable
(C) The schedule has deadlock
(D) Both (A) and (C)

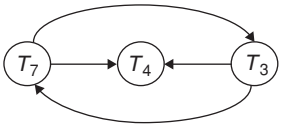
4. Consider the given schedule

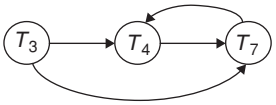
T_3	T_4	T_7
$R(Q)$		
	$W(Q)$	
$W(Q)$		
		$R(Q)$
		$W(Q)$

Which of the following is the correct precedence graph for the above schedule?

(A) 

(B) 

(C) 

(D) 

5. Consider two Transactions T_1 and T_2 and four schedules: S_1, S_2, S_3 and S_4 of T_1 and T_2:

T_1: $r_1(x), w_1(x), w_1(y)$

T_2: $r_2(x), r_2(y), w_2(y)$

S_1: $r_1(x), r_2(x), r_2(y), w_1(x), w_1(y), w_2(y)$

S_2: $r_1(x), r_2(x), r_2(y), w_1(x), w_2(y), w_1(y)$

S_3: $r_1(x), w_1(x), r_2(x), w_1(y), r_2(y), w_2(y)$

S_4: $r_2(x), r_2(y), r_1(x), w_1(x), w_1(y), w_2(y)$

Which schedules are conflict serializable in the given schedules?

(A) S_1 and S_2
(B) S_1 and S_3
(C) S_2 and S_3
(D) S_1 and S_4

6. Consider the following transactions with data items P and Q initialized to '0':

T_1: read(P)
Read(Q)
if $p = 0$ then $Q = Q + 1$
Write(Q)

T_2: read(Q)
Read(P)
if $Q = 0$ then $p = p + 1$
Write (P)

Any non-serial interleaving of T_1 and T_2 for concurrent execution leads to

(A) a serializable schedule
(B) a schedule that is not conflict serializable
(C) a conflict serializable schedule
(D) a schedule for which a precedence graph cannot be drawn

7. Consider the concurrent execution of two transactions T_1 and T_2, if the initial values of x, y, M and N are 200, 100, 10, 20 respectively. What are the final values of x and y?

T_1	T_2
read-item(x)	
$x = x - N$	
	read-item(x)
	$x = x + M$
Write-item(x)	
read-item(y)	
	Write-item(x)
$y = y + N$	
Write-item(y)	

(A) 220, 110 (B) 210, 120
(C) 220, 120 (D) 210, 110

8. For the above data, if the transactions are executed in serial manner, what would be the values of X and Y at the end of the serial execution of T_1 and T_2?

T_1	T_2
Read-item(x)	
$X = X - N$	
Write-item(x)	
Read-item(y)	
$Y = Y + N$	
Write-item(y)	
	Read-item(x)
	$X = X + M$
	Write-item(x)

(A) 190, 120
(B) 180, 120
(C) 190, 110
(D) 180, 110

9. Consider the given two transactions T_1 and T_2:

T_1: $r_1(x), w_1(x), r_1(y)$

T_2: $r_2(x), r_2(y), w_2(x), w_2(y)$

Which of the following schedules are complete schedules?

(A) $r_1(x), r_2(x), w_1(x), r_1(y), r_2(y), w_2(x), w_2(y)$
(B) $r_2(x), r_1(x), r_2(y), w_1(x), w_2(x), r_1(y), w_2(y)$
(C) $r_1(x), r_1(y), r_2(x), r_2(y), w_1(x), w_2(x), w_2(y)$
(D) All the above

10. Consider the given schedule with data-locks on data-items, check whether it has dead-lock or not. The locks are shared-lock(S) and Exclusive-lock(X). Shared-lock is also called Read-lock, Exclusive-lock is also called Write-lock. Read and Write operations are denoted by *R* and *W*, respectively.

T_1	T_2	T_3	T_4
S(A)			
R(A)			
	X(B)		
	W(B)		
S(B)			
		S(C)	
		R(C)	
	X(C)		
			X(B)
		X(A)	

Which of the following is incorrect?

(A) $T_1 \rightarrow T_2$ (B) $T_3 \rightarrow T_1$
(C) $T_2 \rightarrow T_3$ (D) $T_4 \rightarrow T_3$

11. Consider the three transactions T_1, T_2 and T_3 and the schedule S_1 as given below. Draw the serializability (precedence) graph for S_1, and state whether the schedule is serializable or not. If a schedule is serializable, which one of the following is equivalent serial schedule?

T_1: $r_1(x), r_1(z), w_1(x)$
T_2: $r_2(z), r_2(y), w_2(z), w_2(y)$
T_3: $r_3(x), r_3(y), w_3(y)$
S_1: $r_1(x), r_2(z), r_1(z), r_3(x), r_3(y), w_1(x), w_3(y), r_2(y), w_2(z), w_2(y)$?

(A) $r_3(x), r_3(y), w_3(y), r_1(x), r_1(z), w_1(x), r_2(z), r_2(y), w_2(z), w_2(y)$
(B) $r_1(x), r_1(z), w_1(x), r_2(z), r_2(y), w_2(z), w_2(y), r_3(x), r_3(y), w_3(y)$
(C) $r_2(z), r_2(y), w_2(z), w_2(y), r_3(x), r_3(y), w_3(y), r_1(x), r_1(z), w_1(x)$
(D) $r_2(z), r_2(y), w_2(z), w_2(y), r_1(x), r_1(z), w_1(x), r_3(x), r_3(y), w_3(y)$

12. Consider the data given in the above question. Draw the precedence graph for S_2 and state whether each schedule is serializable or not. If a schedule is serializable, which of the following is equivalent serial schedule?

S_2: $r_1(x), r_2(z), r_3(x), r_1(z), r_2(y), r_3(y), w_1(x), w_2(z), w_3(y), w_2(y)$

(A) $r_3(x), r_3(y), w_3(y), r_1(x), r_1(z), w_1(x), r_2(z), r_2(y), w_2(z), w_2(y)$
(B) $r_1(x), r_1(z), w_1(x), r_2(z), r_2(y), w_2(z), w_2(y), r_3(x), r_3(y), w_3(y)$
(C) $r_2(z), r_2(y), w_2(z), w_2(y), r_3(x), r_3(y), w_3(y), r_1(x), r_1(z), w_1(x)$
(D) $r_2(z), r_2(y), w_2(z), w_2(y), r_1(x), r_1(z), w_1(x), r_3(x), r_3(y), w_3(y)$

13. Consider schedule S_3, which is a combination of transactions T_1, T_2 and T_3 from Q. No.11.

S_3: $r_1(x), r_2(z), r_1(z), r_3(x), r_3(y), w_1(x), c_1, w_3(y), c_3, r_2(y), w_2(z), w_2(y), c_2$?

Which of the following is true?

(A) Recoverable and conflict serializable
(B) Recoverable but not conflict serializable
(C) Conflict serializable but not Recoverable
(D) Not recoverable and not conflict serializable

14. Consider the given schedule:

S_4: $r_1(x), r_2(z), r_1(z), r_3(x), r_3(y), w_1(x), w_3(y), r_2(y), w_2(z), w_2(y), c_1, c_2, c_3$:

Which of the following is true?

(A) Recoverable and conflict serializable
(B) Recoverable but not conflict serializable
(C) Conflict serializable but not Recoverable
(D) Not recoverable and not conflict serializable

15. Which of the following is correct for the below compatibility matrix?

Mode of Locks Currently held by other transactions		**Shared-Lock**	**Exclusive-Lock**
	S		
	X		

S – shared - Lock, *X* – Exclusive - Lock

(A)

	S	*X*
S	No	No
X	Yes	No

(B)

	S	*X*
S	Yes	No
X	No	No

(C)

	S	*X*
S	Yes	Yes
X	No	No

(D)

	S	*X*
S	No	Yes
X	No	No

16. Consider the following schedule with locking:

T_1	T_2
Lock – $X(A)$	
$R(A)$	
$W(A)$	
	Lock – $X(B)$
	$R(B)$
	$W(B)$
	Lock – $X(A)$
Lock – $X(B)$	

Which of the following is true?
(A) schedule is in Dead–Lock state
(B) schedule is conflict serializable
(C) schedule is not conflict serializable
(D) Both A and B

17. Consider the given set of transactions:

T_1	T_2
	SELECT AVG (balance)
	FROM Account
INSERT INTO Account	
VALUES (487, 2000);	
COMMIT	
	SELECT AVG (balance)
	FROM Account
	COMMIT

The above problem is a case of
(A) READ UNCOMMITTED
(B) RAD COMMITTED
(C) REPEATABLE READ
(D) DIRTY READ

18. Consider the given set of transactions

T_1	T_2
UPDATE ACCOUNT	
SET balance = balance – 1000	
WHERE number = 586;	
	SELECT AVG (balance)
	FROM Account
ROLL BACK	
	COMMIT

The above problem is a case of
(A) READ UNCOMMITTED
(B) READ COMMITTED
(C) DIRTY READ
(D) BOTH A and C

19. Consider the following set of transactions

T_1	T_2
	SELECT AVG (balance)
	FROM Account
UPDATE Account	
SET balance = balance – 4000	
WHERE number = 586;	
COMMIT	
	SELECT AVG (balance)
	FROM Account
	COMMIT

The above problem is a case of
(A) READ UNCOMMITTED
(B) READ COMMITTED
(C) REPEATABLE READ
(D) DIRTY READ

20. Consider the following schedule with locks on data items:

T_1	T_2	T_3
$X(A)$		
	$X(A)$	
		$X(A)$
$S(B)$		
	$S(B)$	

Which of the following is incorrect?
(A) $T_2 \rightarrow T_1$
(B) $T_3 \rightarrow T_2$
(C) $T_3 \rightarrow T_1$
(D) $T_1 \rightarrow T_3$

Practice Problem 2

Directions for questions 1 to 20: Select the correct alternative from the given choices.

1. Which of the following is false with respect to B^+ - trees of order p?
(A) Each internal node has at most p tree pointers.
(B) Each leaf node has at most $\left\lceil \left(\frac{p}{2} \right) \right\rceil$ values.
(C) Each internal node, except the root, has at least $\left\lceil \left(\frac{p}{2} \right) \right\rceil$ tree pointers.
(D) All leaf nodes are at same level.

2. Consider below transactions:

T_1	T_2
Read – item(X);	
X : = X – N;	
	Read – item (X);
	X: = X + M;
Write – item(X);	
Read – item (Y);	
	Write – item (X);
Y: = Y + N;	
Write – item (Y);	

Which of the following problem will occur during the concurrent execution of the above transactions?

(A) Lost update problem because of incorrect X.
(B) Lost update problem because of incorrect Y.
(C) Dirty read problem because of incorrect X.
(D) Dirty read problem because of incorrect Y.

3. Consider the scheduled:

S: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; C_2; $w_1(Y)$; C_1;

This schedule is

(A) Recoverable
(B) Non-recoverable
(C) Strict schedule
(D) Both (A) and (C)

4. Consider below schedule:

T_1	T_2
Read – item(X);	
X : = X – N;	
	Read – item (X);
	X: = X + M;
Write – item(X);	
Read – item (Y);	
	Write – item (X);
Y: = Y + N;	
Write – item (Y);	

This schedule is

(A) Serializable
(B) Not serializable
(C) Under dead lock
(D) Both (B) and (C)

5. Let, current number of file records = r
maximum number of records = bfr
current number of file buckets = N
Then what will be the file load factor?

(A) $\frac{r}{(bfr * N)}$
(B) $r + (bfr * N)$
(C) $r * (bfr * N)$
(D) $r * (bfr + N)$

6. Match the following:

	LIST I		LIST II
1.	Primary index	A.	Ordered key field
2.	Clustering index	B.	Non-ordered field
3.	Secondary index	C.	Ordered non-key field

(A) 1 – A, 2 – B, 3 – C
(B) 1 – A, 2 – C, 3 – B
(C) 1 – C, 2 – B, 3 – A
(D) 1 – C, 2 – A, 3 – B

7. Consider a file with 30,000 fixed length records of size 100 bytes stored on a disk with block size 1024 bytes. Suppose that a secondary index on a non-ordering key field is constructed with key field size 9 bytes and block pointer 6 bytes. What will be the number of blocks needed for the index?

(A) 68
(B) 442
(C) 1500
(D) 3000

8. Match the following:

	Index type		Number of Index entries
1.	Primary Index	A.	Blocks in data file
2.	Clustering index	B.	Record in data file
3.	Secondary key index	C.	Distinct index filed values

(A) 1 – A, 2 – B, 3 – C
(B) 1 – A, 2 – C, 3 – B
(C) 1 – C, 2 – B, 3 – A
(D) 1 – C, 2 – A, 3 – B

9. Which of the following is true with respect to B – Tree of order p?

(A) Each node has at most p tree pointers.
(B) Each node, except the root and leaf nodes, has at least $\left\lceil \frac{p}{2} \right\rceil$ tree pointers.
(C) All leaf nodes are at the same level.
(D) All of these.

10. What is the amount of unused space in allocation of unspanned fixed records of size R on a block of size B bytes?

(A) $B - R$
(B) $B - \left\lfloor \frac{B}{R} \right\rfloor$
(C) $B - \left(\left\lfloor \frac{B}{R} \right\rfloor * R \right) 4$
(D) $\left(\left\lfloor \frac{B}{R} \right\rfloor * R \right) - B$

11. What is the average time required to access a record in a file consisting of b blocks using unordered heap linear search?

(A) b
(B) $b/2$
(C) $\log_2^b$
(D) b^2

12. Consider a file of fixed length records of size R bytes. If the block size is B bytes, then the blocking factor will be

(A) $B \times R$ records (B) $\left\lfloor \frac{B}{R} \right\rfloor$ records

(C) $\left\lceil \frac{B}{R} \right\rceil$ records (D) $B + R$ records

13. Consider the following relation instance:

P	Q	R
1	4	2
1	5	3
1	6	3
3	2	2

Which of the following FDs are satisfied by the instance?

(A) $PQ \rightarrow R$ and $R \rightarrow Q$ (B) $QR \rightarrow P$ and $Q \rightarrow R$

(C) $QR \rightarrow P$ and $P \rightarrow R$ (D) $PR \rightarrow Q$ and $Q \rightarrow P$

14. Consider an ordered file with 30,000 records stored on a disk with block size of 1024 bytes. The records are of fixed size and are of unspanned, with record length 100 bytes. What is the number of accesses required to access a data file using binary search?

(A) 10 (B) 12

(C) 1500 (D) 3000

15. What is the blocking factor for an index if the ordering key field size is 9 bytes and block pointer is 6 bytes long, and the disk block size is 1024 bytes?

(A) 114 (B) 171

(C) 341 (D) 68

16. For a set of n transactions, there exist _____ different valid serial schedules

(A) n (B) n^2

(C) $n/2$ (D) $n!$

17. The number of possible schedules for a set of n transactions is

(A) lesser than $n!$ (B) much larger than $n!$

(C) $n!$ (D) None

18. Which one of the following is conflict operation?

(A) Reads and writes from the same transaction

(B) Reads and writes from different transaction

(C) Reads and writes from different transactions on different data items.

(D) Reads and writes from different transaction on same data.

19. The following schedule S: $r_3(x), r_2(x), w_3(x), r_1(x), w_1(x)$ is conflict equivalent to serial schedule

(A) $T_1 \rightarrow T_3 \rightarrow T_1$ (B) $T_2 \rightarrow T_1 \rightarrow T_3$

(C) $T_1 \rightarrow T_2 \rightarrow T_3$ (D) None

20. The following schedule S: $R_1(x), R_2(x), W_1(x), W_2(x)$ is

(A) Conflict serializable (B) View serializable

(C) Both (D) None

Previous Years' Questions

1. Consider the following four schedules due to three transactions (indicated by the subscript) using *read* and *write* on a data item x, denoted by $r(x)$ and $w(x)$, respectively. Which one of the them is conflict serializable? **[2014]**

(A) $r_1(x); r_2(x); w_1(x); r_3(x); w_2(x)$

(B) $r_2(x); r_1(x); w_2(x); r_3(x); w_1(x)$

(C) $r_3(x); r_2(x); r_1(x); w_2(x); w_1(x)$

(D) $r_2(x); w_2(x); r_3(x); r_1(x); w_1(x)$

2. Consider the following schedule S of transactions T_1, T_2, T_3, T_4:

T_1	T_2	T_3	T_4
	Reads (X)		
		Writes (X)	
		Commit	
Writes (X)			
Commit			
	Writes (Y)		
	Reads (Z)		
	Commit		
			Reads (X)
			Reads (Y)
			Commit

Which one of the following statements is correct? **[2014]**

(A) S is conflict-serializable but not recoverable

(B) S is not conflict-serializable but is recoverable

(C) S is both conflict-serializable and recoverable

(D) S is neither conflict-serializable nor it is recoverable

3. Consider the following transaction involving two bank accounts x and y.

read (x) ; $x := x - 50$; write (x) ; read(y); $y := y + 50$; write(y)

The constraint that the sum of the accounts x and y should remain constant is that of **[2015]**

(A) Atomicity (B) Consistency

(C) Isolation (D) Durability

4. Consider a simple checkpointing protocol and the following set of operations in the log.

(start, T_4); (write, T_4, y, 2, 3); (start, T_1); (commit, T_4); (write, T_1, z, 5, 7);

(checkpoint);

(start, T_2); (write, T_2, x, 1, 9); (commit, T_2); (start, T_3), (write, T_3, z, 7, 2);

If a crash happens now and the system tries to recover using both undo and redo operations. What are the contents of the undo list and the redo list? **[2015]**

(A) Undo: T_3, T_1; Redo: T_2
(B) Undo: T_3, T_1; Redo: T_2, T_4
(C) Undo: none; Redo: T_2, T_4, T_3, T_1
(D) Undo: T_3, T_1, T_4; Redo: T_2

5. Consider the following partial schedule S involving two transactions T_1 and T_2. Only the read and the write operations have been shown. The read operation on data item P is denoted by read(P) and the write operation on data item P is denoted by write(P)

Time Instance	Transaction – id	
	T_1	T_2
1	read(A)	
2	write(A)	
3		read(C)
4		write(C)
5		read(B)
6		write(B)
7		read(A)
8		commit
9	read(B)	

Schedule S

Suppose that the transaction T_1 fails immediately after time instance 9. Which one of the following statements is correct? **[2015]**

(A) T_2 must be aborted and then both T_1 and T_2 must be re-started to ensure transaction atomicity.
(B) Schedule S is non-recoverable and cannot ensure transaction atomicity.
(C) Only T_2 must be aborted and then re-started to ensure transaction atomicity.
(D) Schedule S is recoverable and can ensure atomicity and nothing else needs to be done.

6. Which one of the following is **NOT** a part of the ACID properties of database transactions? **[2016]**

(A) Atomicity
(B) Consistency
(C) Isolation
(D) Deadlock - freedom

7. Consider the following two phase locking protocol. Suppose a transaction T accesses (for read or write operations), a certain set of objects $\{O_1, \ldots, O_k\}$. This is done in the following manner: **[2016]**

Step 1. T acquires exclusive locks to $O_1, \ldots O_k$ in increasing order of their addresses.

Step 2. The required operations are performed.

Step 3. All locks are released.

This protocol will

(A) guarantee serializability and deadlock-freedom.
(B) guarantee neither serializability nor deadlock-freedom.
(C) guarantee serializability but not deadlock-freedom.
(D) guarantee deadlock-freedom but not serializabilty.

8. Suppose a database schedule S involves transactions $T_1, \ldots T_n$. Construct the precedence graph of S with vertices representing the transactions and edges representing the conflicts. If S is serializable, which one of the following orderings of the vertices of the precedence graph is guaranteed to yield a serial schedule? **[2016]**

(A) Topological order
(B) Depth - first order
(C) Breadth - first order
(D) Ascending order of transaction indices

9. Consider the following database schedule with two transactions T_1 and T_2.

$S = r_2(X); r_1(X); r_2(Y); w_1(X); r_1(Y); w_2(X); a_1; a_2$

Where $r_i(Z)$ denotes a *read* operation by transaction T_i on a variable Z, $w_i(Z)$ denotes a *write* operation by T_i on a variable Z and *ai* denotes an *abort* by transaction T_i.

Which one of the following statements about the above schedule is **TRUE**? **[2016]**

(A) S is non - recoverable
(B) S is recoverable, but has a cascading abort
(C) S does not have a cascading abort
(D) S is strict.

10. In a database system, unique timestamps are assigned to each transaction using Lamport's logical clock. Let $TS(T_1)$ and $TS(T_2)$ be the timestamps of transactions T_1 and T_2 respectively. Besides, T_1 holds a lock on the resource R, and T_2 has requested a conflicting lock on the same resource R. The following algorithm is used to prevent deadlocks in the database system assuming that a killed transaction is restarted with the same timestamp.

if $TS(T_2) < TS(T_1)$ then
    T_1 is killed
else T_2 waits.

Assume any transaction that is not killed terminates eventually. Which of the following is TRUE about the database system that uses the above algorithm to prevent deadlocks? **[2017]**

(A) The database system is both deadlock-free and starvation-free.
(B) The database system is deadlock-free, but not starvation-free.

(C) The database system is starvation-free, but not deadlock-free.

(D) The database system is neither deadlock-free nor starvation-free.

11. Two transactions T_1 and T_2 are given as

$$T_1 : r_1(X)w_1(X)r_1(Y)w_1(Y)$$

$$T_2 : r_2(Y)w_2(Y)r_2(Z)w_2(Z)$$

where $r_i(V)$ denotes a *read* operation by transaction T_i on a variable V and $w_i(V)$ denotes a *write* operation by transaction T_i on a variable V. The total number of conflict serializable schedules that can be formed by T_1 and T_2 is _____. **[2017]**

Answer Keys

Practice Problem I

1. B	**2.** B	**3.** B	**4.** B	**5.** C	**6.** B	**7.** B	**8.** A	**9.** A	**10.** D
11. A	**12.** A	**13.** A	**14.** C	**15.** B	**16.** D	**17.** C	**18.** D	**19.** B	**20.** D

Practice Problem I

1. B	**2.** A	**3.** A	**4.** D	**5.** A	**6.** B	**7.** B	**8.** B	**9.** D	**10.** C
11. B	**12.** B	**13.** B	**14.** B	**15.** D	**16.** D	**17.** C	**18.** D	**19.** A	**20.** D

Previous Years' Questions

1. D	**2.** C	**3.** B	**4.** A	**5.** B	**6.** D	**7.** A	**8.** A	**9.** C	**10.** A
11. 54									