# Database Management System

Transaction Processing

Dr. Balasundaram A

VIT-Chennai

## BCSE302L Syllabus

**Module-5**

- Introduction to Transaction Processing
- Transaction and System concepts
- Desirable properties of Transactions
- Characterizing schedules based on recoverability
- Characterizing schedules based on serializability

# Text Books and References

**Text Books**

- R. Elmasri & S. B. Navathe, Fundamentals of Database Systems, Addison Wesley, 7 th Edition, 2015
- Raghu Ramakrishnan,Database Management Systems,Mcgraw-Hill,4 th edition,2015

**References**

- A. Silberschatz, H. F. Korth & S. Sudershan, Database System Concepts, McGraw Hill, 6 th Edition 2010
- Thomas Connolly, Carolyn Begg, Database Systems : A Practical Approach to Design, Implementation and Management,6 th Edition,2012
- Pramod J. Sadalage and Marin Fowler, NoSQL Distilled: A brief guide to merging world of Polyglot persistence, Addison Wesley, 2012.
- Shashank Tiwari, Professional NoSql, Wiley, 2011.

# Database Management System

## Introduction to Transaction Processing

Dr. Balasundaram A

VIT-Chennai

## Introduction to Transaction Processing

- It is a logical unit of work that represents real-world events of any organization or an enterprise.
- Transaction processing systems execute database transactions with large databases and hundreds of concurrent users.
  For Eg : Railway/Air reservations systems, Banking system, Credit card processing, Stock market monitoring, Super market inventory and checkouts and so on.

**Transaction:**

- It is a logical unit of work of database processing that includes one or more database access operations.
- It is defined as an action or series of actions that is carried out by a single user or application program to perform operations (retrieval (Read), insertion (Write), deletion and modification) for accessing the contents of the database.
- A transaction must be either completed or aborted.
- It can either be embedded within an application program or can be specified interactively via a high-level query language such as SQL.
- Each transaction should access shared data without interfering with the other transactions(i.e., Permanent Effect).

## Introduction to Transaction Processing

- A transaction is a sequence of READ and WRITE actions that are grouped together to from a database access.
- A transaction may consist of a simple SELECT operation to generate a list of table contents, or it may consist of a series of related UPDATE command sequences.

This basic abstraction frees the database application programmer from the following concerns :

- Inconsistencies caused by conflicting updates from concurrent users.
- Partially completed transactions in the event of systems failure.
- User-directed undoing of transactions.

## Operations in Transaction

A transaction can include the following basic database access operations:

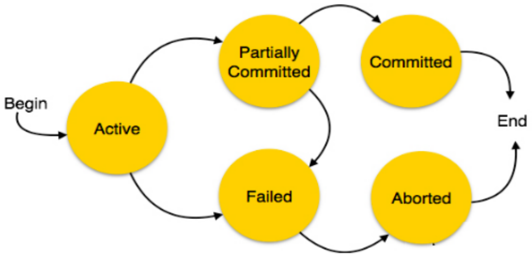
| **Operations** | **Descriptions** |
|---|---|
| Retrieve | To retrive data stored ina database. |
| Insert | To store new data in database. |
| Delete | To delete existing data from database. |
| Update | To modify existing data in database. |
| Commit | To save the work done permanently. |
| Rollback | To undo the work done. |

- Transaction that changes the contents of the 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. To ensure database consistency, every transaction must begin with the database in a known consistent state.

## Transaction Execution and Problems

- A transaction which successfully completes its execution is said to have been Committed. Otherwise, the transaction is Aborted.
- Thus, if a Committed transaction performs any update operation on the database, its effect must be reflected on the database even if there is a failure.

**Transaction States :**



## Transaction States

| State | Description |
|---|---|
| Active state | A transaction goes into an active state immediately after it starts execution, where it can issue READ and WRITE operations. A transaction may be aborted, when the transaction itself detects an error during execution which it cannot recover from |
| Partially committed | When the transaction ends, it moves to the partially committed state. When the last state is reached. To this point, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the transaction permanently. Once this check is successful, the transaction is said to have reached its commit point and enters the committed state. |
| Aborted | When the normal execution can no longer be performed. Failed or aborted transactions may be restarted later, either automatically or after being resubmitted by the user as new transactions. |
| Committed | After successful completion of transaction. A transaction is said to be in a committed state if it has partially committed and it can be ensured that it will never be aborted. |

## Transaction Execution with SQL

The American National Standards Institute (ANSI) has defined standards that govern SQL database transactions.Transaction support is provided by two SQL statements namely COMMIT and ROLLBACK. The ANSI standards require that, when a transaction sequence is initiated by a user or an application program, it must continue through all succeeding SQL statements until one of the following four events occur :

- A COMMIT statement is reached, in which case all changes are permanently recorded within the database. The COMMIT statement automatically ends the SQL transaction. The COMMIT operations indicates successful end-of-transaction.
- A ROLLBACK statement is reached, in which case all the changes are aborted and the database is rolled back to its previous consistent state. The ROLLBACK operation indicates unsuccessful end-of-transaction.
- The end of the program is successfully reached, in which case all changes are permanently. This action is equivalent to COMMIT.
- The program is abnormally terminated, in which case the changes made in the database are aborted and the database is rolled back to its previous consistent state. This action is equivalent to ROLLBACK.

## Desirable Properties of Transaction

A transaction must have the following four properties, called ACID properties (also called ACIDITY of a transaction), to ensure that a database remains stable state after the transaction is executed:

- Atomicity(A): A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
- Consistency preservation(C): A correct execution of the transaction must take the database from one consistent state to another.
- Isolation(I): 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(D): Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

## Transaction Log or Journal

- To support transaction processing, DBMSs maintain a transaction record of every change made to the database into a log (also called journal).
- Log is a record of all transactions and the corresponding changes to the database.
- The information stored in the log is used by the DBMS for a recovery requirement triggered by a ROLLBACK statement, which is program's abnormal termination, a system (power or network) failure, or disk crash.
- Some relational database management systems (RDBMSs) use the transaction log to recover a database forward to a currently consistent state.
- The DBMS automatically update the transaction log while executing transactions that modify the database. The transaction log stores before-and-after data about the database and any of the tables, rows and attribute values that participated in the transaction.
- The beginning and the ending (COMMIT) of the transaction are also recorded in the transaction log.

## Transaction Log or Journal

For each transaction, the following data is recorded on the log:

- A start-of-transaction marker.
- The transaction identifier which could include who and where information.
- The record identifiers which include the identifiers for the record occurrences.
- The operation(s) performed on the records (for example, insert, delete, modify).
- The previous value(s) of the modified data. This information is required for undoing the changes made by a partially completed transaction. It is called the undo log. Where the modification made by the transaction is the insertion of a new record, the previous values can be assumed to be null.
- The updated value(s) of the modified record(s). This information is required for making sure that the changes made by a committed transaction are in fact reflected in the database and can be used to redo these modifications. This information is called the redo part of the log. In case the modification made by the transaction is the deletion of a record, the updated values can be assumed to be null.
- A commit transaction marker if the transaction is committed, otherwise an abort or rollback transaction marker.

## Transaction Log or Journal

- The log is written before any updates are made to the database. This is called write-ahead log strategy. In this strategy, a transaction is not allowed to modify the physical database until the undo portion of the log is written to stable database.
- In case of a system failure, the DBMS examines the transaction log for all uncommitted or incomplete transactions and restores (ROLLBACK) the database to its previous state based on the information in the transaction log.

When the recovery process is completed, the DBMS writes in the transaction log all committed transactions that were not physically written to the physical database before the failure occurred. If a ROLLBACK is issued before the termination of a transaction, the DBMS restores the database only for that particular transaction, rather than for all transactions, in order to maintain the durability of the previous transactions. In other words, committed transactions are not rolled back.

## Transaction Log or Journal Example

Example : Consider the Transaction 'T' refers to a unique transaction-id, generated automatically by the system and is used to identify each transaction:

- $[start\_transaction, T]$: Records that transaction T has started execution.
- $[write\_item, T, X, old\_value, new\_value]$: Records that transaction T has changed the value of database item X from old_value to new_value.
- $[read\_item, T, X]$: Records that transaction T has read the value of database item X.
- $[commit, T]$: Records that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
- $[abort, T]$: Records that transaction T has been aborted.

## Degree of Inconsistency

Following four levels of transaction consistency have been defined by Gray (1976):

| Consistency | Description |
|---|---|
| Level-0 | They are not recoverable since they may have interactions with the external world which cannot be undone. i.e The transaction T does not overwrite other transaction's dirty (or uncommitted) data |
| Level-1 | It is the minimum consistency requirement that allows a transaction to be recovered in the event of system failure. The transaction T does not overwrite other transaction's dirty (or uncommitted) data. The transaction T does not make any of its updates visible before it commits. |
| Level-2 | It's consistency isolates from the updates of other transactions. The transaction T does not overwrite other transaction's dirty (or uncommitted) data. The transaction T does not make any of its updates visible before it commits. The transaction T does not read other transaction's dirty (or uncommitted) data |
| Level-3 | It adds consistent reads so that successive reads of a record will always give the same values. The transaction T does not overwrite other transaction's dirty (or uncommitted) data. The transaction T does not make any of its updates visible before it commits. The transaction T does not read other transaction's dirty (or uncommitted) data. The transaction T can perform consistent reads, i.e., no other transaction can update data read by the transaction T before T has committed. |

## Permutable Actions :

- An action is a unit of processing that is indivisible from the DBMS's perspective.
- The actions provided are determined by the system designers, but in all cases they are independent of side-effects and do not produce side-effects.
- A pair of actions is permutable if every execution of A, followed by Aj has the same result as the execution of $A_j$ followed by A, on the same granule. Actions on different granules are always permutable.
- For the actions read and write we have:
  - Read-Read: Permutable .
  - Read-write: Not permutable, since the result is different depending on whether read is first or write is first.
  - Write-Write: Not permutable, as the second write always nullifies the effects of the first write.

## Schedule

- A schedule (also called history) is a sequence of actions or operations (for example, reading writing, aborting or committing) that is constructed by merging the actions of a set of transactions, respecting the sequence of actions within each transaction.
- As long as two transactions $T_1$ and $T_2$ access unrelated data, there is no conflict and the order of execution is not relevant to the final result. Thus, DBMS has inbuilt software called scheduler, which determines the correct order of execution.
- The scheduler establishes the order in which the operations within concurrent transactions are executed.
- The scheduler interleaves the execution of database operations to ensure serialisability (as explained in next section).The scheduler bases its actions on concurrency control algorithms, such as locking or time stamping methods.
- The schedulers ensure the efficient utilization of central processing unit (CPU) of computer system.
- It can be observed that the schedule does not contain an ABORT or COMMIT action for either transaction.

## Schedule (Cont....)

- Schedules which contain either an ABORT or COMMIT action for each transaction whose actions are listed in it are called a complete schedule.
- If the actions of different transactions are not interleaved, that is, transactions are executed one by one from start to finish, the schedule is called a serial schedule.
- A non-serial schedule is a schedule where the operations from a group of concurrent transactions are interleaved.
- A serial schedule gives the benefits of concurrent execution without giving up any correctness.
- The disadvantage of a serial schedule is that it represents inefficient processing because no interleaving of operations form different transactions is permitted.
- This can lead to low CPU utilization while a transaction waits for disk input/output (I/O), or for another transaction to terminate, thus slowing down processing considerably.

## Schedule Serializability

- When multiple transactions are running concurrently then there is a possibility that the database may be left in an inconsistent state.
- Serializability helps us to check which schedules are serializable and always the database in consistent state.

**What is a serializable schedule?**

- A serializable schedule always leaves the database in consistent state.
- A serial schedule is always a serializable schedule because in serial schedule a transaction only starts when the other transaction finished execution.
- However a non-serial schedule needs to be checked for Serializability.
- A non-serial schedule of n number of transactions is said to be serializable schedule, if it is equivalent to the serial schedule of those n transactions.
- A serial schedule doesn't allow concurrency, only one transaction executes at a time and the other starts when the already running transaction finished.

## Testing of Serializable

Serialization Graph is used to test the Serializability of a schedule

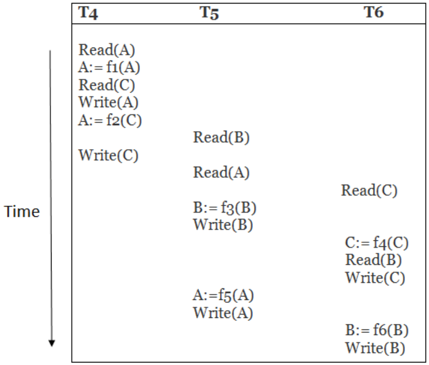
For Schedule S, we construct a precedence graph has a pair G = (V, E), where V consists a set of vertices, and E consists a set of edges. The set of vertices is used to contain all the transactions participating in the schedule.

The set of edges is used to contain all edges $T_i \rightarrow T_j$ for which one of the three conditions holds:

- Create a node $T_i \rightarrow T_j$, if $T_i$ executes write (Q) before $T_j$ executes read (Q).
- Create a node $T_i \rightarrow T_j$, if $T_i$ executes read (Q) before $T_j$ executes write (Q).
- Create a node $T_i \rightarrow T_j$, if $T_i$ executes write (Q) before $T_j$ executes write (Q).

# Example....

- Read(A): In $T_4$,no subsequent writes to A, so no new edges
- Read(C): In $T_4$, no subsequent writes to C, so no new edges
- Write(A): A is subsequently read by T5, so add edge $T_4 \rightarrow$ T5
- Read(B): In $T_5$,no subsequent writes to B, so no new edges
- Write(C): C is subsequently read by $T_6$, so add edge $T_4 \rightarrow T_6$
- Write(B): A is subsequently read by $T_6$, so add edge T5$\rightarrow T_6$
- Write(C): In $T_6$, no subsequent reads to C, so no new edges
- Write(A): In $T_5$, no subsequent reads to A, so no new edges
- Write(B): In $T_6$, no subsequent reads to B, so no new edges



| T4 | T5 | T6 |
|---|---|---|
| Read(A) | | |
| A:= f1(A) | | |
| Read(C) | | |
| Write(A) | | |
| A:= f2(C) | | |
| | Read(B) | |
| Write(C) | | |
| | Read(A) | |
| | | Read(C) |
| | B:= f3(B) | |
| | Write(B) | |
| | | C:= f4(C) |
| | | Read(B) |
| | | Write(C) |
| | A:=f5(A) | |
| | Write(A) | |
| | | B:= f6(B) |
| | | Write(B) |

## Precedence Graph

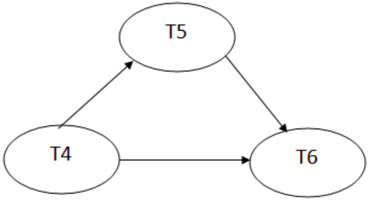
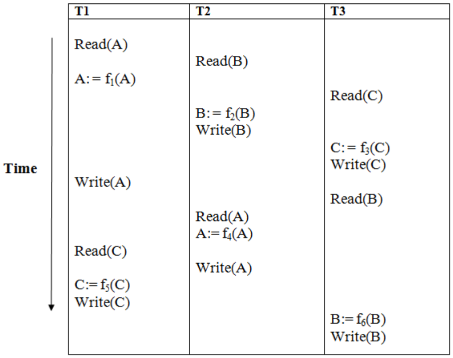

$$\begin{tikzcd} & T5 \arrow[dr] & \\ T4 \arrow[ur] \arrow[rr] & & T6 \end{tikzcd}$$

Figure: Precedence Graph

The precedence graph for schedule S contains no cycle that's why Schedule S is serializable.

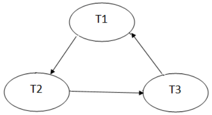
# Example-2

- Read(A): In T1, no subsequent writes to A, so no new edges
- Read(B): In T2, no subsequent writes to B, so no new edges
- Read(C): In T3, no subsequent writes to C, so no new edges
- Write(B): B is subsequently read by T3, so add edge T2 $\rightarrow$ T3
- Write(C): C is subsequently read by T1, so add edge T3 $\rightarrow$ T1
- Write(A): A is subsequently read by T2, so add edge T1 $\rightarrow$ T2
- Write(A): In T2, no subsequent reads to A, so no new edges
- Write(C): In T1, no subsequent reads to C, so no new edges
- Write(B): In T3, no subsequent reads to B, so no new edges



| T1 | T2 | T3 |
|---|---|---|
| Read(A) | | |
| | Read(B) | |
| A:= $f_1$(A) | | |
| | | Read(C) |
| | B:= $f_2$(B) | |
| | Write(B) | |
| | | C:= $f_3$(C) |
| | | Write(C) |
| Write(A) | | |
| | | Read(B) |
| | Read(A) | |
| | A:= $f_4$(A) | |
| Read(C) | | |
| | Write(A) | |
| C:= $f_5$(C) | | |
| Write(C) | | |
| | | B:= $f_6$(B) |
| | | Write(B) |

**Schedule S1**



The precedence graph for schedule S contains a cycle that's why Schedule S is non-serializable.

## Types of Serializability

1. Conflict Serializable
2. View Serialziable

**Types of Schedule** :

- Serial Schedule : Doesn't support concurrent execution
- Non-Serial Schedule : Support's concurrency

## Conflict Serializable

**Conflict Serialziability**

- A schedule is called conflict serializable if we can convert it into a serial schedule after swapping its non-conflicting operations.
- Conflict Serializability is one of the type of Serializability, which can be used to check whether a non-serial schedule is conflict serializable or not.
- Two operations are said to be in conflict, if they satisfy all the following three conditions:
  - Both the operations should belong to different transactions.
  - Both the operations are working on same data item.
  - At least one of the operation is a write operation.

## Examples of Conflict Serializable

- **Example 1**: Operation W(X) of transaction $T_1$ and operation R(X) of transaction $T_2$ are conflicting operations, because they satisfy all the three conditions mentioned above. They belong to different transactions, they are working on same data item X, one of the operation in write operation.
- **Example 2**: Similarly Operations W(X) of $T_1$ and W(X) of $T_2$ are conflicting operations.
- **Example 3**: Operations W(X) of $T_1$ and W(Y) of $T_2$ are non-conflicting operations because both the write operations are not working on same data item so these operations don't satisfy the second condition.
- **Example 4**: Similarly R(X) of $T_1$ and R(X) of $T_2$ are non-conflicting operations because none of them is write operation.

  Example 5: Similarly W(X) of $T_1$ and R(X) of $T_1$ are non-conflicting operations because both the operations belong to same transaction $T_1$.

## Conflict Equivalent Schedule

Two schedules are said to be conflict Equivalent if one schedule can be converted into other schedule after swapping non-conflicting operations.

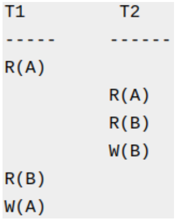
**Conflict Serializable check** : Lets consider this schedule:

```
T1         T2
-----     ------
R(A)
R(B)
          R(A)
          R(B)
          W(B)
W(A)
```

To convert this schedule into a serial schedule we must have to swap the R(A) operation of transaction $T_2$ with the W(A) operation of transaction $T_1$. However we cannot swap these two operations because they are conflicting operations, thus we can say that this given schedule is not Conflict Serializable.

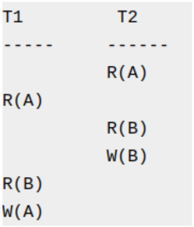
## Examples...

Let Us have an Example of Transaction Schedule



```
T1          T2
-----      ------
R(A)
          R(A)
          R(B)
          W(B)
R(B)
W(A)
```

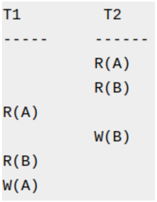
Figure: After swapping R(A) of T1 and R(A) of T2



```
T1          T2
-----      ------
          R(A)
R(A)
          R(B)
          W(B)
R(B)
W(A)
```

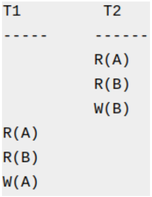
## Examples...

After swapping R(A) of T1 and R(B) of T2 we get:



```
T1            T2
-----        ------
             R(A)
             R(B)
R(A)
             W(B)
R(B)
W(A)
```

Figure: After swapping R(A) of T1 and W(B) of T2



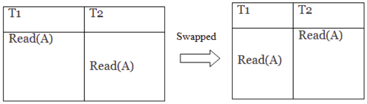
```
T1            T2
-----        ------
             R(A)
             R(B)
             W(B)
R(A)
R(B)
W(A)
```

Now the schedule is serial after swapping all the non-conflicting operations and is Conflict Serializable.

# Examples....

Swapping is possible only if $S_1$ and $S_2$ are logically equal.

**1. T1: Read(A)  T2: Read(A)**



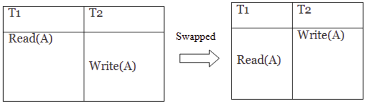
| T1 | T2 |
|---|---|
| Read(A) | |
| | Read(A) |

| T1 | T2 |
|---|---|
| | Read(A) |
| Read(A) | |

**Schedule S1**  **Schedule S2**

Figure: Here, $S_1 = S_2$. That means it is non-conflict.

**2. T1: Read(A)  T2: Write(A)**



| T1 | T2 |
|---|---|
| Read(A) | |
| | Write(A) |

| T1 | T2 |
|---|---|
| | Write(A) |
| Read(A) | |

**Schedule S1**  **Schedule S2**

Figure: Here, $S_1 \,!= S_2$. That means it is Conflict.

## Examples...

Table: Schedule $S_1$

| $T_1$ | $T_2$ |
|---|---|
| Read(A) | |
| Write(A) | |
| | Read(A) |
| | Write(A) |
| Read(B) | |
| Write(B) | |
| | Read(B) |
| | Write(B) |

Table: Schedule $S_2$

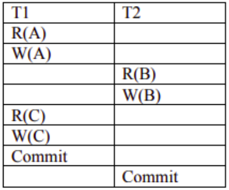
| T1 | T2 |
|---|---|
| Read(A) | |
| Write(A) | |
| Read(B) | |
| Write(B) | |
| | Read(A) |
| | Write(A) |
| | Read(B) |
| | Write(B) |

Schedule $S_2$ is a serial schedule because, in this, all operations of $T_1$ are performed before starting any operation of $T_2$. Schedule $S_1$ can be transformed into a serial schedule by swapping non-conflicting operations of $S_1$.

## Solve if U Can....!

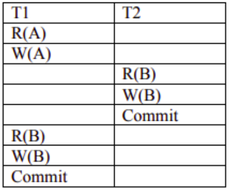
Check for Serialize and its type

Example 1



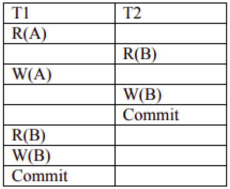
| T1 | T2 |
|---|---|
| R(A) | |
| W(A) | |
| | R(B) |
| | W(B) |
| R(C) | |
| W(C) | |
| Commit | |
| | Commit |

Example 2.



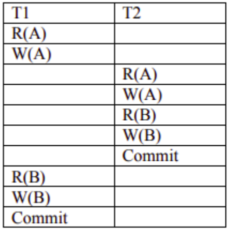
| T1 | T2 |
|---|---|
| R(A) | |
| W(A) | |
| | R(B) |
| | W(B) |
| | Commit |
| R(B) | |
| W(B) | |
| Commit | |

Example 3



| T1 | T2 |
|---|---|
| R(A) | |
| | R(B) |
| W(A) | |
| | W(B) |
| | Commit |
| R(B) | |
| W(B) | |
| Commit | |

Example 4.



| T1 | T2 |
|---|---|
| R(A) | |
| W(A) | |
| | R(A) |
| | W(A) |
| | R(B) |
| | W(B) |
| | Commit |
| R(B) | |
| W(B) | |
| Commit | |

## Thanks