Chapter-1- Transaction Management and Concurrency Control

1.1. Transaction

A transaction can be defined as a group of tasks. A single task is the minimum processing unit which cannot be divided further. It is an action, or a series of actions, carried out by a single user or an application program, which reads or updates the contents of a database.

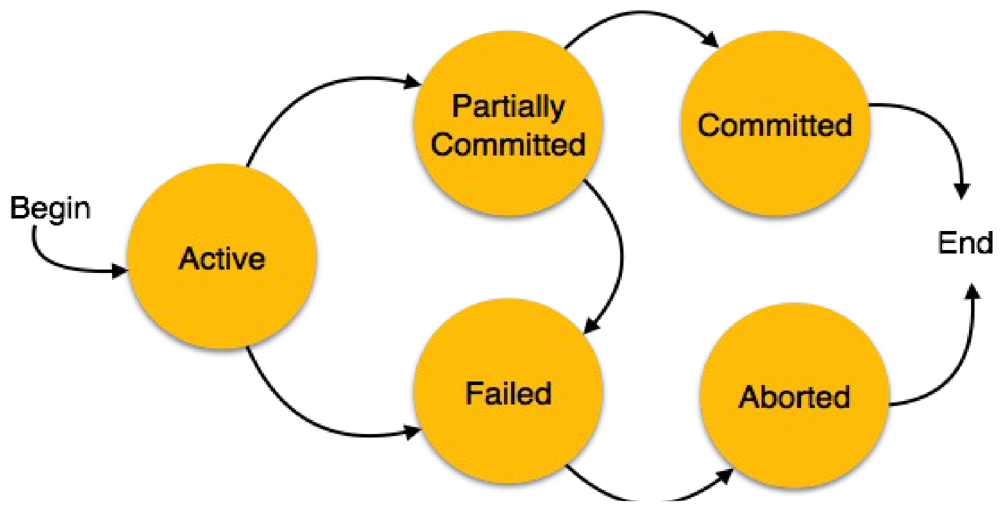
A transaction is a collection of operations that performs a single logical function in a database application. The transaction-management component ensures that the database remains in a consistent (correct) state despite system failures (e.g., power failures and operating system crashes) and transaction failures.

1.1.1. Possible States of Transactions

A transaction in a database can be in one of the following states:

- **Active** − In this state, the transaction is being executed. This is the initial state of every transaction.
- **Partially Committed** − When a transaction executes its final operation, it is said to be in a partially committed state.
- **Failed** − A transaction is said to be in a failed state if any of the checks made by the database recovery system fails. A failed transaction can no longer proceed further.
- **Aborted** − If any of the checks fails and the transaction has reached a failed state, then the recovery manager rolls back all its write operations on the database to bring the database back to its original state where it was prior to the execution of the transaction. Transactions in this state are called aborted. The database recovery module can select one of the two operations after a transaction aborts −
  - Re-start the transaction
  - Kill the transaction

- **Committed** – If a transaction executes all its operations successfully, it is said to be committed. All its effects are now permanently established on the database system.



1.1.2. Properties of Transactions

All transactions should satisfy the following four properties, abbreviated as, ACID properties.

- **Atomicity**: Transactions are atomic – they don't have parts (conceptually)

- **Consistency**: Transactions take the database from one consistent state into another. In the middle of a transaction the database might not be consistent.
- **Isolation/Independence**: The effects of a transaction are not visible to other transactions until it has completed, i.e. no partial effects of incomplete transactions are visible.
- **Durability**: Once a transaction has completed, its changes are made permanent. Even if the system crashes, the effects of a transaction must remain in place.

Example of transaction

Transfer 50 birr from account A to account B

$$\left.\begin{array}{l}\text{Read(A)}\\ \text{A = A - 50}\\ \text{Write(A)}\\ \text{Read(B)}\\ \text{B = B+50}\end{array}\right\}\text{A single transaction}$$

Write(B)

This transaction can be described according to the four properties as follows:

Atomicity - shouldn’t take money from A without giving it to B

Consistency - money isn’t lost or gained

Isolation - other queries shouldn’t see A or B

change until completion

Durability - the money does not go back to A

1.2. Transaction Support

In a database system, it is the task of the transaction manager to control/manage transactions so that all transactions can satisfy the four transaction properties. The transaction manager follows different techniques for this purpose.

- For atomicity: COMMIT and ROLLBACK techniques are used.
- For Consistency and Isolation: Locks or timestamps are used.
- For Durability: A log is kept in the event of system failure.

The COMMIT process signals the successful completion of a transaction.

Any changes made by the transaction should be saved. And these changes are now visible to other transactions.

In contrast to the COMMIT process, the ROLLBACK signals the unsuccessful end of a transaction. And in this case, any changes made by the transaction should be undone. It is now as if the transaction never existed.

The transaction log records the details of all transactions.

The transaction details include:

- Any changes the transaction makes to the database
- How to undo these changes
- When transactions complete and how

The log is stored on disk, not in memory and if the system crashes, it is preserved.

1.3. Concurrency Control

Concurrency control is the part of transaction handling that deals with how multiple users access the shared database without running into each other.

1.3.1. Concurrency Problems

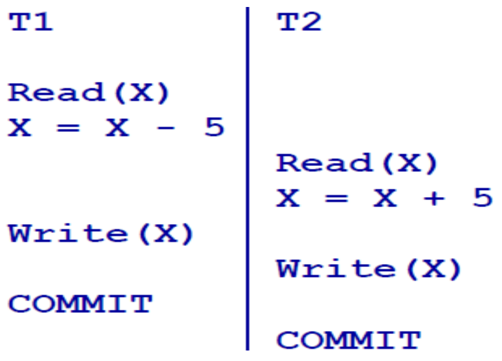
Concurrency can result in the following problems:

- Lost update problem
- Uncommitted update
- Incorrect analysis

Lost Update Problem:

It can be briefed by the example bellow.

There are two transactions T1 and T2. T1 is expected to subtract 5 from X and T2 is expected to add 5 to X. And it is obvious that the expected final result is no change to X. But, because of the lost update problem, the result is different.



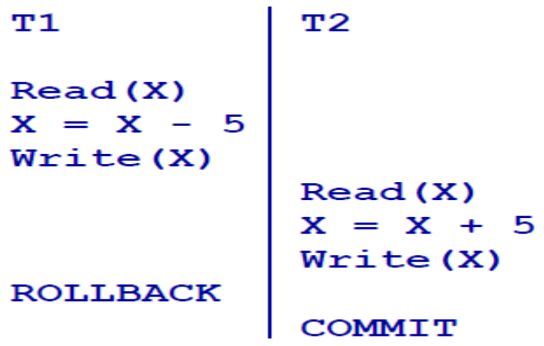
```
T1            | T2
              |
Read(X)       |
X = X - 5     |
              | Read(X)
              | X = X + 5
Write(X)      |
              | Write(X)
COMMIT        |
              | COMMIT
```

Both transactions, T1 and T2, read X. T1 subtracts 5 from X and updates the database and T1 commits. T2 also adds 5 to X (which is read before T1 updates its value) and again updates the database and commits. At the end of both transactions, the value of X is increased by 5.

As you can observe, the update T1 has made to X is meaningless and this is how a lost update problem may occur.

Uncommitted Update Problem:

Let's take the above two transactions for demonstration. What if T2 reads X after T1 updated it and if T1 is rolled back? That is T2 sees the change T1 made to X but T1 rolled back.



```
T1            | T2
              |
Read(X)       |
X = X - 5     |
Write(X)      |
              | Read(X)
              | X = X + 5
              | Write(X)
ROLLBACK      |
              | COMMIT
```

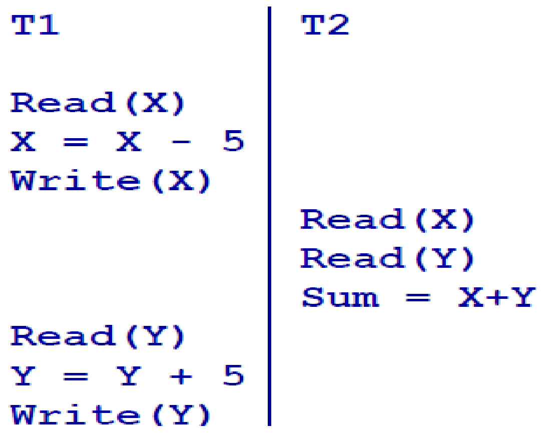
The change made by T1 is undone because it rolls back. The expected result is a final addition of 5 to the old value of X because already what T1 is made is undone. But, according to the above scenario, no change is made to the value of X though T1 is rolled back. This kind of problem is called uncommitted update problem.

Incorrect Analysis

Observe the following transaction processing, T1 and T2.

T1 takes 5 from X and adds it to Y. But, before T1 updates Y, T2 reads X and Y.

What would you expect on the sum, X+Y? According to T1, it has got no change; but, according to T2, the sum X+Y is lesser by 5 than the sum T1 expects. This kind of problem is called incorrect analysis problem.



```
T1            | T2
              |
Read(X)       |
X = X - 5     |
Write(X)      |
              | Read(X)
              | Read(Y)
              | Sum = X+Y
Read(Y)       |
Y = Y + 5     |
Write(Y)      |
```

In a multiprogramming environment where multiple transactions can be executed simultaneously, it is highly important to control the concurrency of transactions. We have concurrency control protocols to ensure atomicity, isolation, and serializability of concurrent transactions.

Concurrency control protocols can be broadly divided into two categories −

- Lock based protocols
- Time stamp based protocols

1.3.2. Lock-based Protocols

Database systems equipped with lock-based protocols use a mechanism by which any transaction cannot read or write data until it acquires an appropriate lock on it. Locks are of two kinds −

- **Binary Locks** – In which case a lock on a data item can be in two states; it is either locked or unlocked.
- **Shared/exclusive** − This type of locking mechanism differentiates the locks based on their uses. If a lock is acquired on a data item to perform a write operation, it is an exclusive lock. Allowing more than one transaction to write on the same data item would lead the database into an inconsistent state. Read locks are shared because no data value is being changed.

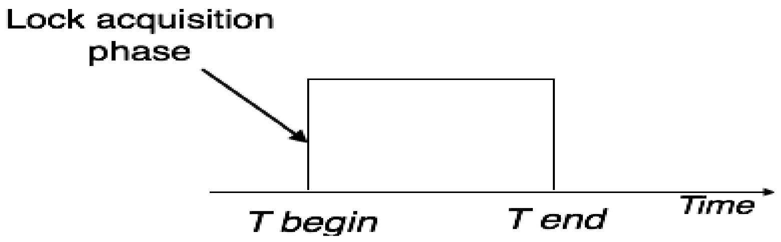
There are four types of lock protocols available – *simplistic lock, pre-claiming lock, two phase lock (2PL), and strict two phase lock (strict 2PL).*

Simplistic Lock Protocol

Simplistic lock-based protocols allow transactions to obtain a lock on every object before a 'write' operation is performed. Transactions may unlock the data item after completing the ‘write’ operation.

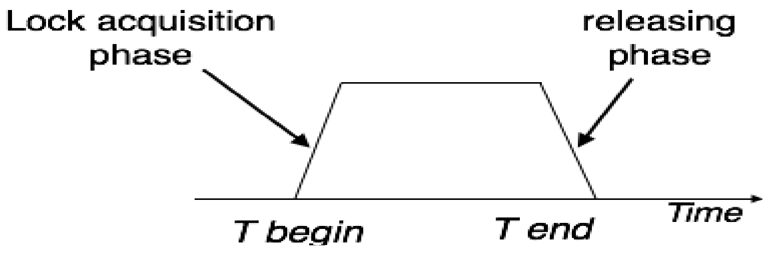
Pre-claiming Lock Protocol

Pre-claiming protocols evaluate their operations and create a list of data items on which they need locks. Before initiating an execution, the transaction requests the system for all the locks it needs beforehand. If all the locks are granted, the transaction executes and releases all the locks when all its operations are over. If all the locks are not granted, the transaction rolls back and waits until all the locks are granted.



Two-Phase Locking 2PL

This locking protocol divides the execution phase of a transaction into three parts. In the first part, when the transaction starts executing, it seeks permission for the locks it requires. The second part is where the transaction acquires all the locks. As soon as the transaction releases its first lock, the third phase starts. In this phase, the transaction cannot demand any new locks; it only releases the acquired locks.

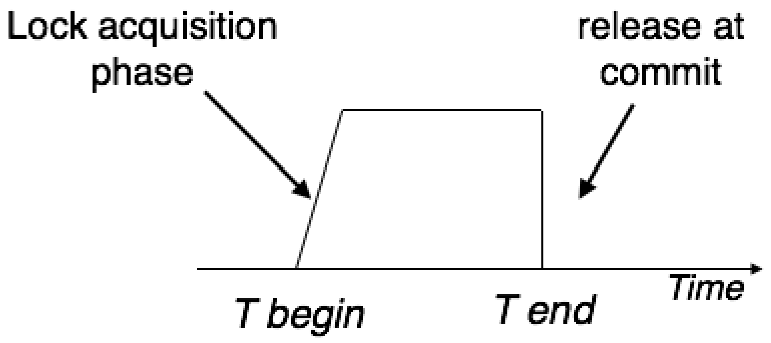


Two-phase locking has two phases, one is growing, where all the locks are being acquired by the transaction; and the second phase is shrinking, where the locks held by the transaction are being released.

To claim an exclusive (write) lock, a transaction must first acquire a shared (read) lock and then upgrade it to an exclusive lock.

Strict Two-Phase Locking

The first phase of Strict-2PL is same as 2PL. After acquiring all the locks in the first phase, the transaction continues to execute normally. But in contrast to 2PL, Strict-2PL does not release a lock after using it. Strict-2PL holds all the locks until the commit point and releases all the locks at a time.



1.3.3. Time Stamp-Based Protocols

The most commonly used concurrency protocol is the timestamp based protocol. This protocol uses either system time or logical counter as a timestamp.

Lock-based protocols manage the order between the conflicting pairs among transactions at the time of execution, whereas timestamp-based protocols start working as soon as a transaction is created.

Every transaction has a timestamp associated with it, and the ordering is determined by the age of the transaction. A transaction created at 0002 clock time would be older than all other transactions that come after it. For example, any transaction 'y' entering the system at 0004 is two seconds younger and the priority would be given to the older one.

In addition, every data item is given the latest read and write-timestamp. This lets the system know when the last ‘read and write’ operation was performed on the data item.

The timestamp-ordering protocol ensures serializability among transactions in their conflicting read and writes operations. This is the responsibility of the protocol system that the conflicting pair of tasks should be executed according to the timestamp values of the transactions.

- The timestamp of transaction T_i is denoted as $TS(T_i)$.
- Read time-stamp of data-item X is denoted by R-timestamp(X).
- Write time-stamp of data-item X is denoted by W-timestamp(X).

Timestamp ordering protocol works as follows −

- If a transaction T_i issues a read(X) operation −
  - If $TS(T_i)$ < W-timestamp(X)

    Operation rejected.
  - If $TS(T_i)$ >= W-timestamp(X)

    Operation executed.
  - All data-item timestamps updated.
- If a transaction T_i issues a write(X) operation −
  - If $TS(T_i)$ < R-timestamp(X)

    Operation rejected.
  - If $TS(T_i)$ < W-timestamp(X)

    Operation rejected and T_i rolled back.
  - Otherwise, operation executed.

1.4. Schedule and Concept of Serializability

When multiple transactions are being executed by the operating system in a multiprogramming environment, there are possibilities that instructions of one transaction are interleaved with some other transaction.

A chronological execution sequence of a transaction is called a **schedule**. It is a process of grouping transactions into one large transaction and executing them in a predefined order. It is a sequence of execution of operation from various transactions. It must consist of all the instruction of those transactions and must preserve the order in which the instructions appear in each individual transaction.

Schedule is required in database because when multiple transactions execute in parallel, they may affect the result of each other, resulting in concurrency access problems. So, to resolve this, the order of execution of instructions in the transactions is changed by creating a schedule.

A schedule can be serial schedule or non-serial schedule when there is more than one transaction in a system.

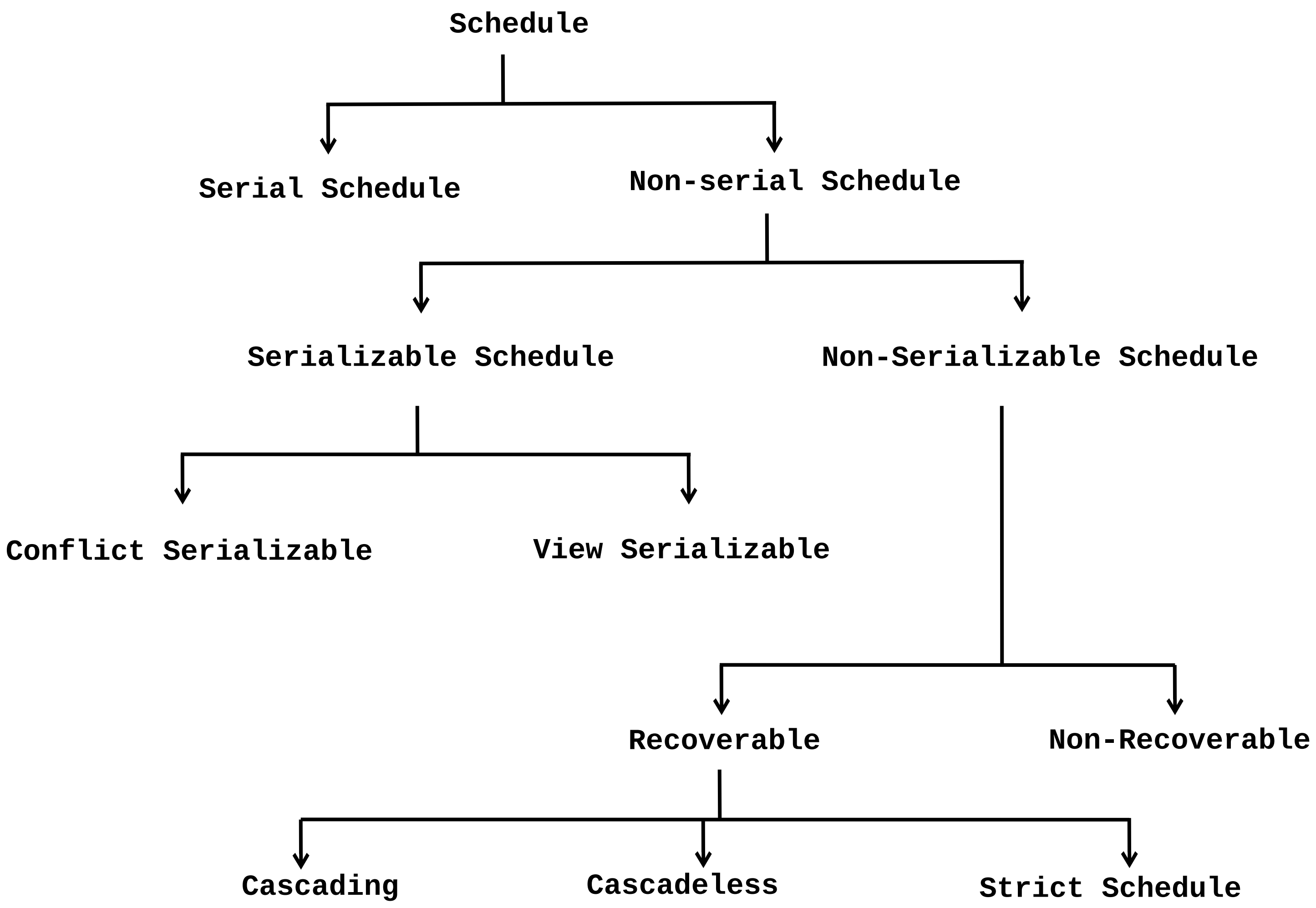
Serial Schedule

It is a schedule in which transactions are aligned in such a way that one transaction is executed first. When the first transaction completes its cycle, then the next transaction is executed. Transactions are ordered one after the other and because of this a serial schedule is also called non-interleaved execution/schedule. This type of schedule is called a serial schedule, as transactions are executed in a serial manner. This type of schedule is ALWAYS consistent; but some non-serial schedules may lead to inconsistency of databases. In non serial schedule, transactions are interleaved together to form one big transaction and for this they are called interleaved schedule.

In a multi-transaction environment, serial schedules are considered as a benchmark. The execution sequence of an instruction in a transaction cannot be changed, but two transactions can have their instructions executed in a random fashion. This execution does no harm if two transactions are mutually independent and working on different segments of data; but in case these two transactions are working on the same data, then the results may vary. This ever-varying result may bring the database to an inconsistent state.

To resolve this problem, we allow parallel execution of a transaction schedule, if its transactions are either serializable or have some equivalence relation among them.

Serializability is a concept that helps to identify which non-serial schedules are correct and will maintain the consistency of the database.



Serial Schedule and Serializable Schedule:

Both schedules are consistent, i.e. they keep the database consistent. A non-serial schedule is called serializable if it has an equivalent serial schedule, i.e. if it has the same result as if it was executed in a serial schedule. Below is some distinction between serial and serializable schedules:

Serial Schedule:	**Serializable Schedule:**
• Concurrency access is not possible, i.e. not allowed. This is because the transactions execute one after the other	• Concurrency access is possible, i.e. allowed. Multiple transactions can execute concurrently
• poorer efficiency, i.e. will result in lesser CPU throughput and higher resource consumption	• Better efficiency because they have better resource utilization and higher CPU throughput

Conflict Serializable Schedule:

If a given non-serial schedule can be converted into a serial schedule by swapping its non-conflicting operations, then it is called as a **conflict serializable schedule**.

Two operations are called conflicting operations if all the following conditions hold true:

- Both the operations belong to different transactions
- Both the operations are on the same data item
- At least one of the two operations is a write operation

Example: Assume two transactions T1 (Deposits 500 birr to account A) and transaction T2 (Withdraws 300 birr from account A) and a non-serial schedule S which is given as follows:

Schedule S	
T1	**T2**
Read1(A)	...
A=A+500	...
Write1(A)	...
...	**Read2(A)**
...	A=A-300
...	Write2(A)

In the above schedule, the two operations **`Write1(A)`** and **`Read2(A)`** are conflicting operations because they fulfill all the above three conditions.

Steps on how to check if a serializable schedule is conflict serializable or not:

i) Identify all conflicting operations

ii) Start creating a precedence graph, also called serialization graph, by drawing one node for each transaction

iii) Draw an edge for each conflicting pair directing from the first operation to the second operation in the conflicting pair and this ensures the order of execution of the operations

iv) Checking the graph, if there is cycle then the schedule is NOT conflict serialiazable because a cycle between transactions shows that one is dependent on the other which means the schedule is NOT serializable and so it is non-serializable; otherwise, i.e. if there is no cycle, the schedule is conflict serializable.

If the schedule is conflict serializable, you can get at least one serial schedule from the directed acyclic graph.

View Serializable Schedule:

A schedule S is view serializable if it is view equivalent to a serial schedule. And two schedules S1 and S2 are view equivalent if and only if all the following conditions are met for every data item D:

- S1 and S2 must have the same set of transactions
- If in schedule S1, transaction T_i reads the initial value of D, then in schedule S2 also transaction T_i must read the initial value of D.
- If in schedule S1 transaction T_i executes **`Read(D)`**, and that value was produced by transaction T_k (if there was any), then in schedule S2 also transaction T_i must read the value of D that was produced by the same **`Write(D)`** operation of transaction T_k.
- A transaction T_i (if any) that performs the final **`Write(D)`** operation in schedule S1 must also perform the final **`Write(D)`** operation in schedule S2.

To check a schedule for view serializability in short, do the following:

i) Check for conflict serializability (if it is conflict serializable, then is view serializable; otherwise, proceed to the next step)

ii) Check for blind write (a blind write is a write operation on data item without reading it). If there is a blind write, there is a possibility for view serializability and it must be checked by the above techniques; but if there is no blind write, we are dead sure that the schedule is not view serializable.

Note that every conflict serializable schedule is also view serializable; but every view serializable schedule is NOT conflict serializable.

For example, the following schedule is view serializable but not conflict serializable:

Schedule S		
T1	**T2**	**T3**
Read1(D)		
	…	…
…	Write2(D)	…
Write1(D)	…	…
…	…	Write3(D)

The above schedule S is view serializable because it is view equivalent with the serial schedule **T1->T2 -> T3**. It is NOT conflict serializable because there exists a cycle between **T1** and **T2** in its precedence graph as shown below:

$$\begin{tikzcd} T1 \arrow[r, bend left] & T2 \arrow[l, bend left] & T3 \end{tikzcd}$$

Recoverable and Non-Recoverable Schedules:

A schedule *S* is recoverable if **no transaction *T* in *S* commits** until all transactions T_i that have written some item *X* that *T* reads have committed. Otherwise, it is non-recoverable.

The following schedule is recoverable because T2 has read the data (A) that T1 updated previously and committed after T1 committed.

Schedule S	
T1	**T2**
Read1(A) A=A+500 **Write1(A)** ... ... COMMIT	... ... **Read2(A)** A=A-300 Write2(A) ... COMMIT

Cascading, Cascadeless, and Strict Schedule:

A schedule is said to be cascadeless if every transaction in the schedule reads only items that were written by committed transactions. It avoids the problem of **cascading rollback**. Cascading rollback is a condition where an *uncommitted* transaction has to be rolled back because it reads an item from a transaction that failed. For example, in the above schedule S, if T1 failed rather than committed, then T2 must be rolled back to avoid uncommitted update problem. And for this, it is called cascading recoverable schedule.

An example of cascadeless schedule is as follows:

Schedule S	
T1	**T2**
Read1(A)	...
A=A+500	...
Write1(A)	...
COMMIT	
...	**Read2(A)**
...	A=A-300
...	Write2(A)
	COMMIT

A more restrictive form of recoverable schedule is strict schedule. In such a schedule, transactions can *neither read nor write* an item *X* until the last transaction that wrote *X* has committed (or aborted). Strict schedules simplify the recovery process.

Below is an example of cascadeless schedule which is NOT strict schedule; It is not strict schedule because T2 wrote data item A which was first written by T1 before T1 commits. But if T2 write data item A after T1 committed, it will be strict schedule.

Schedule S	
T1	**T2**
Read1(A)	
A=A+500	...
Write1(A)	...
	Write2(A)
...	...
COMMIT	
...	COMMIT

The following is an example of strict schedule:

Schedule S	
T1	**T2**
Read1(A)	...
A=A+500	...
Write1(A)	...
COMMIT	...
...	Write2(A)
...	...
	COMMIT

1.5. Concurrency Control Mechanism

One way to avoid any problems regarding concurrency database access is to allow only one user in the database at a time. The only problem with that solution is that the other users are going to get lousy response time. Can you seriously imagine doing that with a bank teller machine system or an airline reservation system where tens of thousands of users are waiting to get into the system at the same time?

It is the task of the concurrency-control manager to control the interaction among the concurrent transactions, to ensure the consistency of the database.

Concurrency control mechanisms can be pessimistic, optimistic, or logical control types.

Pessimistic Concurrency Control

Pessimistic concurrency control is based on the idea that transactions are expected to conflict with each other, so we need to design a system to avoid the problems before they start.

All pessimistic concurrency control schemes use locks, which are flags placed in the database that give exclusive access to a schema object to a user.

The differences are the level of locking they use; setting those flags on and off costs time and resources. If you lock the whole database, then you have a serial batch processing system since only one transaction at a time is active. In practice, you would do this only for system maintenance work—there are no other transactions that involve the whole database.

If you lock at the table level, then performance can suffer because users must wait for the most common tables to become available. However, there are transactions that do involve the whole table and this will use only one flag.

If you lock the table at the row level, then other users can get to the rest of the table and you will have the best possible shared access. You will also have a huge number of flags to process and performance will suffer. This approach is generally not practical.

Page locking is in between table and row locking. This approach puts a lock on subsets of rows within the table that include the desired values. The name comes from the fact that this is usually implemented with pages of physical disk storage. Performance depends on the statistical distribution of data in physical storage, but it is generally the best compromise.

Optimistic Concurrency Control

Optimistic concurrency control is based on the idea that transactions are not very likely to conflict with each other, so we need to design a system to handle the problems as exceptions after they actually occur.

Most optimistic concurrency control uses a timestamp to track copies of the data.

Logical Concurrency Control

Logical concurrency control is based on the idea that the machine can analyze the predicates in the queue of waiting queries and processes on a purely logical level and then determine which of the statements can be allowed to operate on the database at the same time.

1.6. Database Recovery

Database recovery is important whenever there is a media failure. As compared to system failure, media failure is severe. This is because, when a system failure occurs, the DBMS will recover transactions from a transaction log file; but, if the media fails, the log file itself may fail in which case another technique should be used to make a database recovery. This is possible if you have a backup. So, backup is a technique which is useful when media failure occurs.

When you backup a database, the transaction log and entire contents of the database are written to secondary storage (often tape). A backup is taken on a regular basis so that it is possible to restore the database from the last backup.

The transaction log is used to redo any changes made since the last backup.

But if the transaction log file is also damaged, there is no means to recover the database.

To reduce the risk of losing both the log file and your data, it is preferable to backup the log file and the data on a separate backup device.

1.7. Transaction and Recovery

Transactions should be durable, but we cannot prevent all sorts of failures.

In a database system, there are a number of types of failure including the following:

- System failures (such as hardware like RAM, software, network errors etc)
- Transaction/system error: Instructions in a transaction might cause the transaction to fail, such as division by zero or programming logic error or erroneous parameter values etc
- Concurrency control enforcement: The concurrency control method may decide to abort a transaction because it violates serializability, or to resolve a state of deadlock among several transactions
- Power failures
- Disk crashes
- User mistakes such as user interrupting transactions during their execution
- Sabotage
- Natural disasters

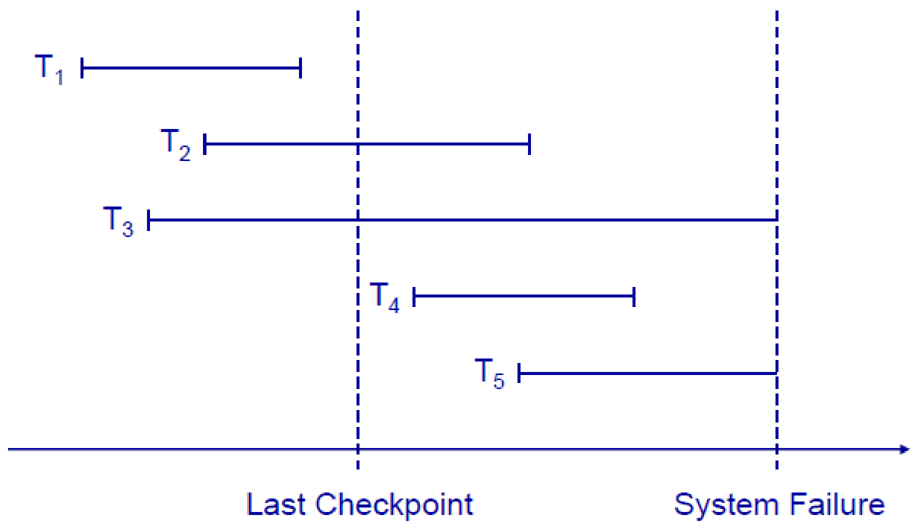
And there are different techniques that prevent failures. Following are some common techniques:

- Installing reliable operating system
- Implementing strong security systems
- Using UPS and surge protectors
- RAID arrays

If a system failure occurs, all running transactions are affected.

The DBMS takes checkpoints at various times. And at a given checkpoint, there can be a record of committed transactions and running transactions which are not yet completed.

Following are different possible states of transactions when a system failure occurs.



After the system failure, the DBMS should be able to recover the transactions follows:

- Any transaction that was running at the time of failure needs to be undone and restarted.
- Any transaction that committed since the last checkpoint need to be redone.

Using the above logic, the following can be done to the above five transaction conditions:

- Transactions of type T1 need no recovery, because it is a committed transaction.
- Transactions of type T3 or T5 need to be undone and restarted, because they are running transactions at the time of failure.
- Transactions of type T2 and T4 need to be redone because they are committed after the last checkpoint and before the system fails.

1.8. Recovery Techniques

Transaction recovery techniques work as follows.

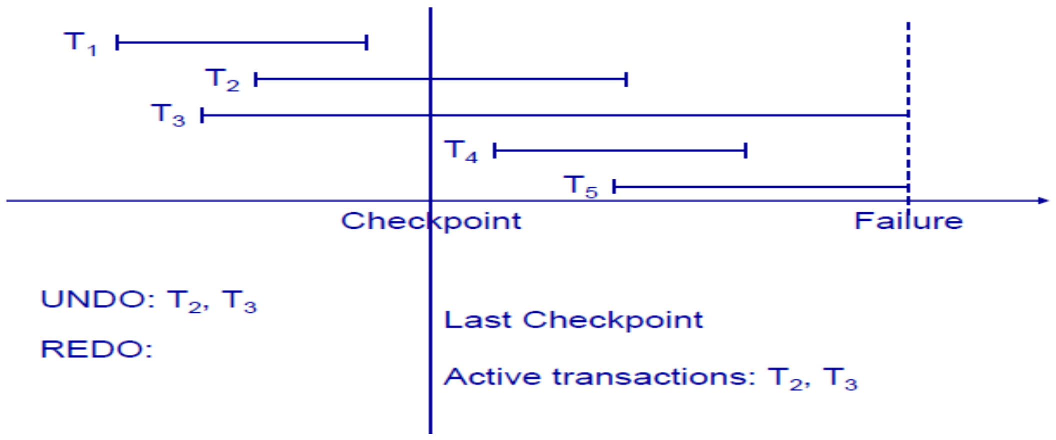
They use two lists of transactions, UNDO transaction lists and REDO transaction lists.

In the beginning of the recovery process, both lists are initialized. The recovery algorithm is given below:

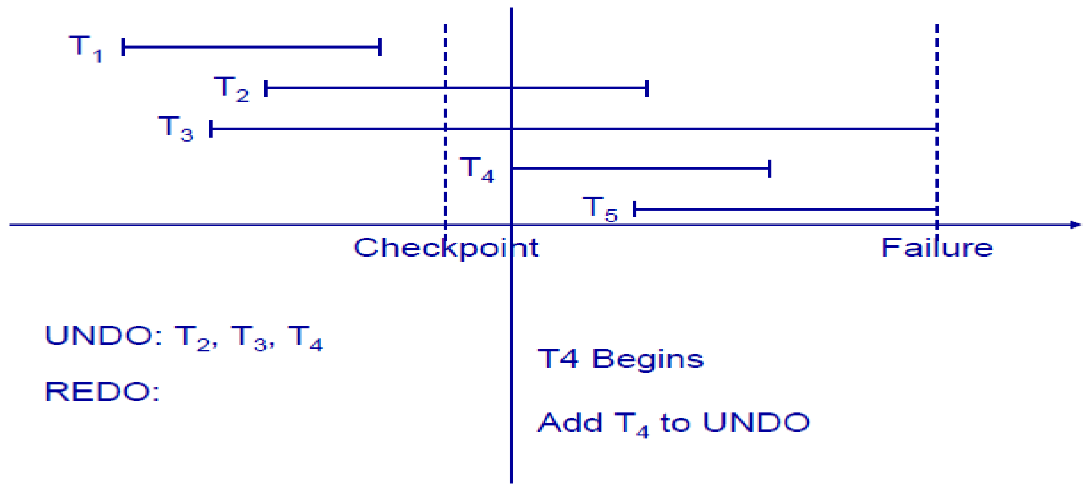
```
UNDO = all transactions running at the last checkpoint
REDO = empty
For each entry in the log, starting at the last checkpoint
If a BEGIN TRANSACTION entry is found for T
Add T to UNDO
If a COMMIT entry is found for T
Move T from UNDO to REDO
```

To demonstrate the recovery algorithm, let’s take the previous five transactions above.

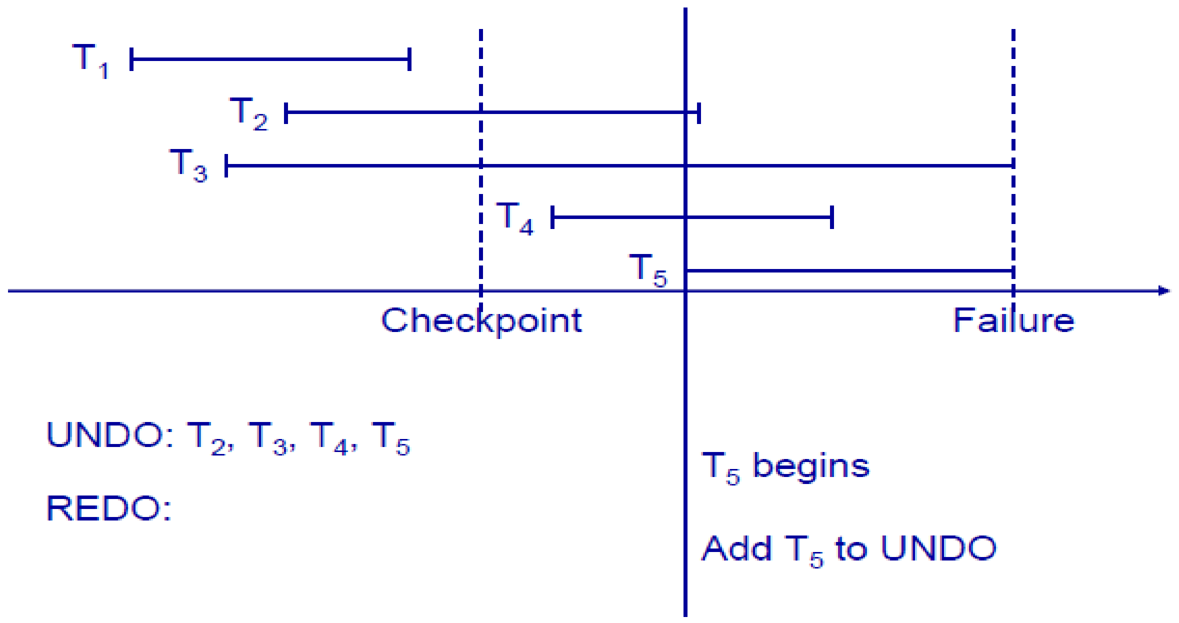
Observe that, at the last checkpoint only transaction T2 and T3 were in a running state. So, the UNDO list contains only the two transactions and the REDO list is empty.



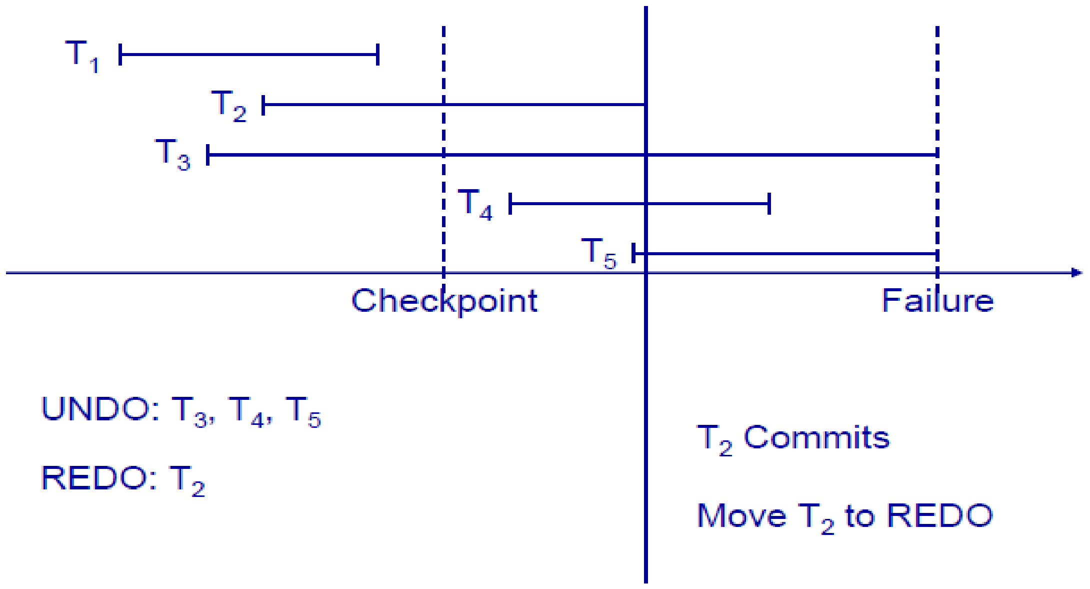
The next entry in the log says ‘T4’ begins, and therefore it should be added to the UNDO list:



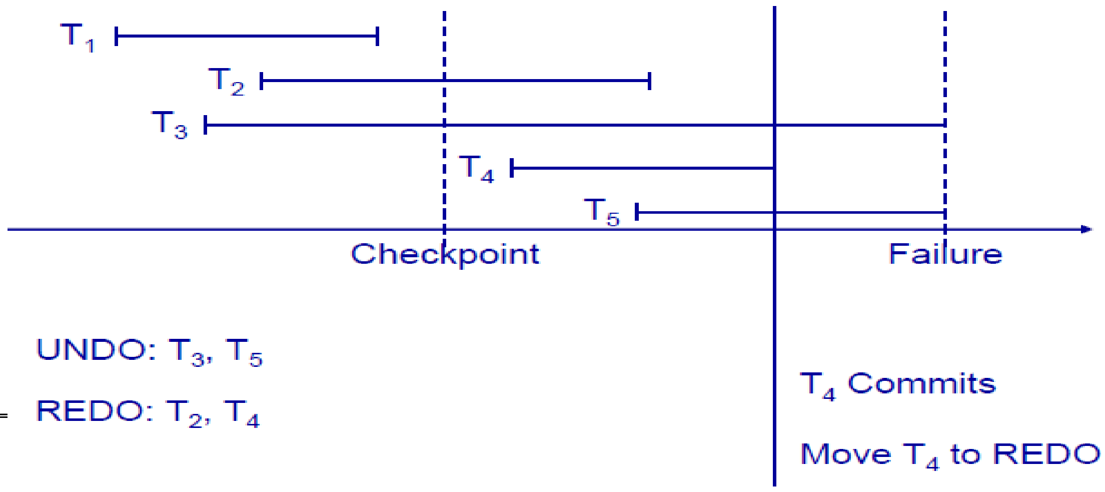
And again, the next entry in the log says ‘T5’ begins, and therefore it should be added to the UNDO list:



Accordingly, the next entry says ‘T2 committed’, and therefore it should be transferred from the UNDO list to the REDO list.



Accordingly, the next entry says ‘T4 committed’, and therefore it should be transferred from the UNDO list to the REDO list.



And next no entry is found in the log because of the failure.

In conclusion, T2 and T4 are redone by the recovery process whereas T3 and T5 are restarted.

Chapter-2- Query Processing and Optimization

2.1. Overview

Relational database systems are expected to be equipped with a query language that can assist its users to query the database instances. One type of such query language is relational algebra.

Relational algebra is a procedural query language, which takes instances of relations as input and yields instances of relations as output. It is used internally in a DBMS to represent a query **evaluation plan.** It uses operators to perform queries. An operator can be either unary or binary. They accept relations as their input and yield relations as their output.

Relational algebra is performed recursively on a relation and intermediate results are also considered relations.

The fundamental operations of relational algebra are as follows −

- Select
- Project
- Union
- Set different
- Cartesian product

- Rename

The Select Operation, σ

It selects tuples that satisfy the given predicate from a relation. The **σ** operator is a unary operator, i.e. it is always applied to a single relation. It follows the notation

$$\sigma_p r$$

Where **σ** stands for selection predicate and **r** stands for relation. **p** is a logic formula which may use connectors like **and**, **or**, and **not**. These terms may use relational operators like – =, ≠, ≥, <, >, ≤.

Look at the examples in the table below.

Relational algebra expression:	An equivalent sql statement:	What it does:
$\sigma_{fname=”Abebe”}$(student)	SELECT * FROM student WHERE fname=”Abebe”;	Selects all tuples from **student** relation/table where the first name, fname, of a student is ‘Abebe’.
$\sigma_{fname=”Abebe” \text{ AND } dep = “IT”}$(student)	SELECT * FROM student WHERE fname=”Abebe” AND dep=”IT”;	Selects all students whose first name is ‘Abebe’ from IT department.
$\sigma_{fname=”Abebe” \text{ OR } dep = “IT”}$(student)	SELECT * FROM student WHERE fname=”Abebe” OR	Selects all students whose first name is ‘Abebe’ regardless of their department and also selects all students in IT

	dep=”IT”;	department.

The Project Operation, Π

It projects columns that satisfy a given predicate. The project, **Π,** operator is a unary operator, i.e. it is always applied to a single relation. It follows the notation

$$\Pi_{A1,\ A2\ldots\ An}\ r$$

Where $A_1, A_2 \ldots A_n$ are attribute names of relation **r**.

Duplicate rows are automatically eliminated, as relation is a set.

For example,

Relational algebra expression:	**An equivalent sql statement:**	**What it does:**

Π_{fname}**(student)**	SELECT fname
	FROM student
		Selects the first name of all students; but, the difference between the projection and the sql statement is that the projection selects only unique first names, i.e. no duplication of first names, though there might be several students with the same first name. But, the sql statement doesn’t avoid any duplication.

The Union Operation, ∪

It performs binary union between two given relations. The union operation is defined as:

$$r \cup s = \{ t \mid t \in r \text{ or } t \in s\}$$

Where, **r** and **s** are relations.

For a union operation to be valid, the following conditions must hold −

- **r** and **s** must have the same number of attributes.

- Attribute domains must be compatible.
- Duplicate tuples are automatically eliminated.

The Set Difference Operation, -

The result of set difference operation is tuples, which are present in the first relation but are not in the second one. It uses the notation:

r - s

It finds all the tuples that are present in **r** but not in **s**.

Example: Suppose we have two relations **Books** and **Articles**, and they have ‘***a_Name’*** as a common attribute that represents the name of the author.

Relational algebra expression:	**What it does:**
Π_{a_Name} (Books) ∪ Π_{a_Name} (Articles)	Projects the names of the authors who have either written a book or an article or both.
Π_{a_Name} (Books) - Π_{a_Name} (Articles)	Provides the name of authors who have written books but not articles.

Cartesian Product (Cross Product), X

The Cartesian product combines information of two different relations into one. It follows the notation:

r X s

Where **r** and **s** are relations and their output will be defined as −

$$r \times s = \{\, q\,t \mid q \in r \text{ and } t \in s \}$$

q and **t** are tuples of the relations **r** and **s**, respectively.

If the relations **r** and **s** contain the tuples as follow:

relation **r**

T_Id	T_name
001	Ayele
002	Bahru
003	Adem

relation **s**

C_Id	C_title
C101	Introduction to ICT
C102	Basics of XML Technology

r X s is given by

T_Id	T_name	C_Id	C_title
001	Ayele	C101	Introduction to ICT
001	Ayele	C102	Basics of XML Technology
002	Bahru	C101	Introduction to ICT
002	Bahru	C102	Basics of XML Technology
003	Adem	C101	Introduction to ICT
003	Adem	C102	Basics of XML Technology

Join operator (⋈)

The relational algebra operator join is a binary operator, i.e. it operates between two relations and returns a single relation. It uses the notation

$$R_1 \bowtie_{a1 = a2} R_2$$

Where, R1 and R2 are relations and a1 and a2 are attributes of R1 and R2, respectively.

The above join operation is exactly equivalent to the following combination of select and Cartesian product operations:

$$\sigma_{a1 = a2} (R_1 \times R_2)$$

Example:

Let '**Teacher'** and '**Course'** be two relations defined in the following way:

T_Id ….. teacher ID

C_Num … course number

CT_Id … teacher Id, which is a foreign key in the course relation

Teacher:

T_Id	**T_name**
001	Ayele
002	Bahru
003	Adem

Course:

C_Num	**CT_Id**	**C_title**
C101	002	Introduction to ICT
C102	001	Basics of XML Technology
C103	001	Internet Programming II
C104	002	Basics of Application Management

Therefore, the join operation

$$\text{Teacher} \bowtie_{T\_Id = CT\_Id} \text{Course}$$

outputs the same result as

$$\sigma_{T\_Id = CT\_Id}(\text{Teacher} \times \text{Course})$$

T_Id	T_name	C_Num	CT_Id	C_title
002	Bahru	C101	002	Introduction to ICT
001	Ayele	C102	001	Basics of XML Technology
001	Ayele	C103	001	Internet Programming II
002	Bahru	C104	002	Basics of Application Management

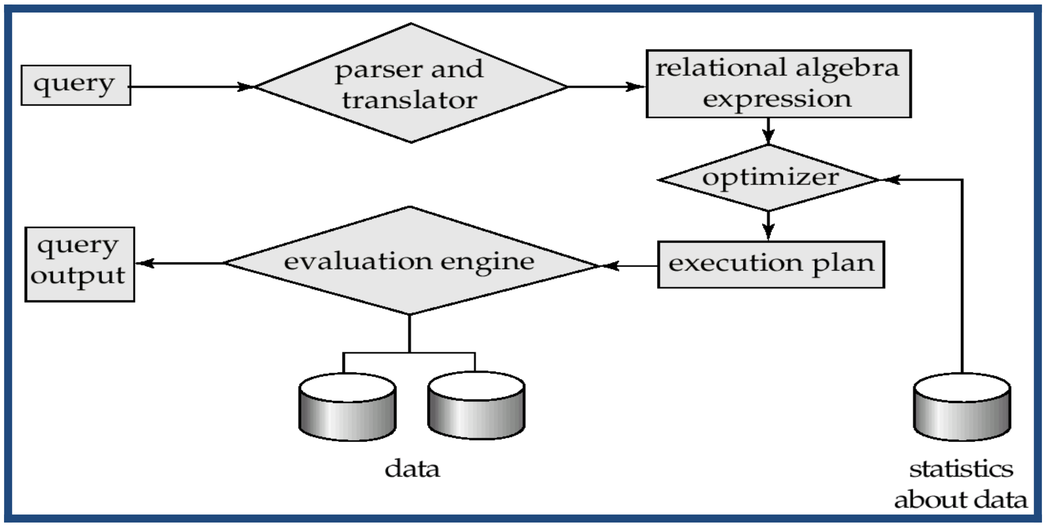
Any query with a join can always be rewritten into cross product followed by selection.

2.2. Query Processing

Query processing involves mainly the execution of the following activities:

- *Parsing and translation*: the query is translated into relational algebra and the parser then checks for syntax errors.
- *Optimization*: making the query as efficient as possible.
- *Evaluation*: The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query. Relational algebra is used to represent a plan internally in the DBMS.

The above activities are illustrated by the figure bellow:



Two key components of the query evaluation component of a SQL database system are the query optimizer and the query execution engine.

The execution engine is responsible for the execution of a query plan that results in generating answers to the query.

The query optimizer is responsible for generating the input for the execution engine.

2.3. Optimization Process

Query optimization is a process of determining an **evaluation plan** with the **lowest cost** amongst all equivalent evaluation plans. Cost is estimated using statistical information from the database catalog, which for example, can be number of tuples in each relation, size of tuples, etc.

Cost is generally measured by the total elapsed time for answering a query.

Many factors such as *disk accesses*, *CPU*, or even *network communication* contribute to time cost.

A relational algebra expression may have many equivalent expressions.

Each relational algebra operation can be evaluated using one of several different algorithms. And correspondingly, a relational-algebra expression can be evaluated in many ways.

Generally speaking, the query optimizer takes a parsed representation of a SQL query as input and is responsible for generating an efficient execution plan for the given SQL query from the space of possible execution plans.

To find an efficient execution plan for a given SQL query, the optimizer performs two things:

- The algebraic representation of the given query can be transformed into many other logically equivalent algebraic representations:

  e.g. `Join(Join(A,B),C)= Join(Join(B,C),A)`

Where, A, B, and C are relations.

- For a given algebraic representation, there may be many operator trees that implement the algebraic expression. So, it is the task of the optimizer to select the best tree.

The query optimizer is then a critical component in reducing the response time of the execution plan.

2.4. Basic Steps in Query Optimization

As seen previously, the query optimization process performs the two basic tasks: representing the input query statement into a proper relational algebra expression and then building a number of operator trees. It is from those possible evaluation plans that the most efficient one will be provided to the query execution engine.

Converting query statements into a relational algebra expression:

The following table summarizes basic query statements and their corresponding relational algebra expressions:

Let a relational schema for a ***student*** is given as follows:

<u>St_Id</u>	F_Name	L_Name	Department	Division	Batch_Year

Query statement:	Relational algebra expression:

SELECT *
 FROM *student*
 WHERE *Batch_Year>1;*
	$\sigma_{Batch\_Year > 1}(\text{student})$

SELECT *F_Name, L_Name*
 FROM *student*
	$\Pi_{F\_Name,\ L\_Name}(\text{student})$

SELECT *F_Name, L_Name*
 FROM *student*
 WHERE *Batch_Year>1;*
	$\Pi_{F\_Name,\ L\_Name}(\sigma_{Batch\_Year > 1}\ (\text{student}))$

Exercise: Convert the following query statements into a relational algebra expression that corresponds each.

(1) **SELECT** *

FROM *student*

WHERE *Batch_Year>1* AND*F_Name="Abebe" ;*

(2) **SELECT** *F_Name, L_Name*

FROM *student*

WHERE *Batch_Year>1* AND *Department="IT" ;*

The query optimizer has a number of relational algebra expression options which are equivalent. Following are equivalent expressions which results similar information, but with different efficiency:

$$\sigma_{Batch\_Year > 1}\ (\prod_{F\_Name,\ Batch\_Year} (\text{student})) \equiv \prod_{F\_Name,\ Batch\_Year} (\sigma_{Batch\_Year > 1}\ (\text{student}))$$

2.5. Pipelining

Pipelining is one way of evaluating relational operators from the relational expression tree. In this technique, several operations are evaluated simultaneously. The result of one operation is sent to the parent operation while other tuples are operated.

As an example, consider the following two relations: ***department*** and ***student***.

student

<u>St_Id</u>	**F_Name**	**L_Name**	**Dep_code**	**CGPA**

department

<u>Dep_code</u>	**Dep_title**	**Dep_faculty**

The following relational algebra expression tree extracts title/name of departments that contains students that have a **CGPA** of **4.0**.

The pipelining technique selects the first student that scored 4.0, then it passes the tuple of that student to the join operation, i.e. prior to selecting the rest students that scored 4.0, and then the join operation is performed on that student and the result is sent to the projection operator. The selection for the next students that have CGPA of 4.0 also continues in parallel to the join and projection operations.

Pipelining may not always be applicable. For example, if the final output of the expression tree is to be sorted in some order pipelining can't be applied.

Dep_title

$\bowtie$

The second technique of evaluating relational algebra expression tree, other than pipelining, is materialization. It is always applicable.

To execute the above tree, it first selects all students that scored 4.0, then it sends those students to the operation, and finally the result of the join operation is passed to the projection operation.

Chapter-3- Database Integrity, Security and Recovery

3.1. Integrity

3.1.1. Integrity Concept & Subsystem

Following are some concepts in relational database.

Tables − in relational data model, relations are saved in the format of Tables. This format stores the relation among entities. A table has rows and columns, where rows represent records and columns represent the attributes.

Tuple − A single row of a table, which contains a single record for that relation is called a tuple.

Relation instance − A finite set of tuples in the relational database system represents relation instance. Relation instances do not have duplicate tuples.

Relation schema − A relation schema describes the relation name table name, attributes, and their names.

Relation key − each row has one or more attributes, known as relation key, which can identify the row in the relation table uniquely.

Attribute domain − every attribute has some pre-defined value scope, known as attribute domain.

Keys and Candidate Keys - There must be at least one minimal subset of attributes in the relation, which can identify a tuple uniquely. This minimal subset of attributes is called key for that relation. If there are more than one such minimal subset, these are called candidate keys.

Foreign key - is a key attribute of a relation that can be referred in other relation.

3.1.2.Integrity Constraints

Every relation has some conditions that must hold for it to be a valid relation. These conditions are called **Relational Integrity Constraints**. There are three main integrity constraints –

Key Constraints:

Key constraints force the following two conditions:

- In a relation with a key attribute, no two tuples/rows can have identical values for key attributes.
- A key attribute cannot have NULL values.

Key constraints are also referred to as Entity Constraints.

Domain constraints:

Attributes have specific values in real-world scenario. For example, age can only be a positive integer. Every attribute is bound to have a specific range of values. For example, age cannot be less than zero and telephone numbers cannot contain a digit outside 0-9.

Referential integrity constraints:

Referential integrity constraints work on the concept of *Foreign Keys*.

Referential integrity constraint states that if a relation refers to a key attribute of a different or same relation, then that key element must exist.

It ensures the integrity of referential relationships between tables as defined by primary and foreign keys. In a relation between two tables, one table has a primary key and the other a foreign key. The primary key uniquely identifies each record in the first table. In other words, there can be only one record in the first table with the same primary key value.

The foreign key is placed into the second table in the relationship such that the foreign key contains a copy of the primary key value from the record in the related table.

So, referential Integrity ensures the integrity of relationships between primary and foreign key values in related tables. Most relational database engines use what are often called constraints. Primary and foreign keys are both constraints.

There are some specific circumstances to consider in terms of how Referential Integrity is generally enforced:

A primary key table is assumed to be a parent table and a foreign key table a child table.

- When adding a new record to a child table, if a foreign key value is entered, it must exist in the related primary key field of the parent table.

Foreign key fields can contain NULL values. Primary key field values can never contain NULL values as they are required to be unique.

- When changing a record in a parent table if the primary key is changed, the change must be cascaded to all foreign key valued records in any related child tables. Otherwise, the change to the parent table must be prohibited.
- When changing a record in a child table, a change to a foreign key requires that a related primary key must be checked for existence, or changed first. If a foreign key is changed to NULL, no primary key is required. If the foreign key is changed to a non-NULL value, the foreign key value must exist as a primary key value in the related parent table.
- When deleting a parent table record then related foreign key records in child tables must either be cascade deleted or deleted from child tables first.

3.2. Security

When you think of securing your database, the following issues should be considered:

- Who gets the DBA role?
- How many users will need access to the database?
- Which users will need which privileges and which roles?
- How will you remove users who no longer need access to the database?

3.2.1. Database threats

Database security is all about securing the data in the database, the network, system environment on which the database is running (the machine and the operating system), the database clients (that includes the applications that makes use of the database and browser software), the server etc.

The goal of ***database access security*** is to determine precisely the data that each database user needs to conduct their business, and what they are permitted to do with the data (that is, select, insert, update, or delete). Each database user should be given exactly the privileges they need—nothing more and nothing less.

3.2.2. Identification and Authentication

Every database user who connects to the database must supply appropriate credentials to establish the connection. Typically, this is in the form of a user ID (or login ID) and a password.

Database security can be formed from two levels: *system level* and *object level.*

System privileges

System privileges are general permissions to perform functions in managing the server and the database(s). Hundreds of permissions are supported by each database vendor, with most of those being system privileges.

Permit the grantee to perform a general database function, such as creating new user accounts or connecting to the database.

Here are some commonly used Microsoft SQL Server system privileges:

- **SHUTDOWN**: Provides the ability to issue the server shutdown command

- **CREATE DATABASE**: Provides the ability to create new databases on the SQL server

- **BACKUP DATABASE**: Provides the ability to run backups of the databases on the SQL server

Object Privileges

Object privileges are granted to users with the SQL GRANT statement and revoked with the REVOKE statement. The database user (login) who receives the privileges is called the grantee.

Permit the grantee to perform specific actions on specific objects, such as selecting from the EMPLOYEES table or updating the DEPARTMENTS table.

To reduce the burden of managing privileges, most RDBMSs support storing a group of privilege definitions as a single named object called a **role**. Roles may then be granted to individual users, who then inherit all the privileges contained in the role. RDBMSs that support roles also typically come with a number of predefined roles. Oracle, for example, has a role called DBA that contains all the high-powered system and object privileges a database user needs in administering a database.

3.2.3. Categories of Control

One function that is performed by a DBMS is access control. The DBMS identifies and controls the issues such as who accesses what data, to do what, when, from where, etc.

Access control is mandatory in a multiuser database, e.g., for confidentiality of information.

There are various access modes to data that include read only, read and update, delete etc.

DBMS subsystem enforces security and authorization levels.

The restrictions that the subsystem of a DBMS concerned with can be program related or data related. Program restriction can be, for example, who can create new bank accounts and data restriction can be which bank accounts an individual user can see.

The DBMS stores information regarding the users of the DBMS and their access privileges (name and password) in the data dictionary.

The access privileges can have the following several levels:

- To create a database
- to authorize (grant) additional users to
  - ✓ access the database
  - ✓ access some relations
  - ✓ create new relations
  - ✓ update the database

- to revoke privileges

3.2.4. Data Encryption

Encryption is the translation of data into a secret code that cannot be read with the use of a password or secret key. Unencrypted data is called plain text, whereas encrypted data is called cipher text.

Some encryption schemes use a symmetric key, which means that a single key is used to both encrypt plain text and to decrypt cipher text. This form is considered less secure compared with the use of asymmetric keys, where a pair of keys is used— one called the public key and the other the private key. What the public key encrypts, the private key can decrypt, and vice versa. The names come from the expected use of the keys—the public key is given to anyone with whom an enterprise does business, and the private key remains confidential and internal to the enterprise.

Chapter-4- Distributed Database Systems

4.1. Concepts of Distributed Databases

A distributed database system consists of loosely coupled sites that share no physical component. The database systems that run on each site are independent of each other. But, transactions may access data at one or more sites.

Distributed databases can be homogeneous or heterogeneous.

Homogeneous Distributed Database

In this database, all database sites have identical software. Each site knows each other and they cooperate in processing user requests. But for the user, it seems to be a single-site database.

Heterogeneous Distributed Database

In these types of databases, different sites may use different schemas and software. The difference in schema is a major problem for query processing and also the difference in software is a major problem for transaction processing. The different sites may not be aware of each other and may provide only limited facilities for cooperation in transaction processing.

Generally, in heterogeneous distributed databases, difference in the following issues can exist:

- Data models (relational, object, hierarchical, network etc)
- Transaction commit protocols may be incompatible
- Concurrency control techniques may vary (locking and timestamp based)
- System level issues such as character set (ASCII and EBCDIC), data types, precisions etc may vary

4.2. Distributed Database Design

Distributed databases are designed in such a way that they support replicated data, i.e. they keep multiple copies of data for faster data access and better fault tolerance and fragmented relations which are stored in distinct sites.

In some distributed systems, both replication and fragmentation can co-exist, which means different sites might store identical replicas of each of the fragmented relations.

A relation is said to be replicated if it is stored in at least two sites redundantly.

If all sites store identical relations, then there exists **full replication**. In addition, if all sites contain a copy of the whole database, then the system is referred to be **fully redundant** system.

Replication:

Advantage of Data Replication

- **Increases Availability**: failure of site containing relation r does not result in unavailability of r because of the existence of other copies of the relation (replicas).
- **Parallelism**: queries on r may be processed by several nodes in parallel.
- **Reduced data transfer**: because replica of relation r is available locally at each site, there is no need of transfer of relations from site to site.

Disadvantage of Data Replication

- **Increased cost of updates**: each replica of relation r must be updated.
- **Increased complexity of concurrency control**: concurrent updates to distinct replicas may lead to inconsistent data unless special concurrency control mechanisms are implemented. One solution for this can be choosing one copy as primary copy and apply concurrency control operations on primary copy.

Fragmentation:

Fragmentation of relation r are divisions such as r_1, r_2... r_n which contain sufficient information to reconstruct the original relation r.

Relation fragmentation can be horizontal or vertical fragmentation.

In horizontal fragmentation, each tuple of a relation r is assigned to one or more fragments.

And in vertical fragmentation, the schema for relation r is split into several smaller schemas. All schemas must contain a common candidate key (or superkey) to ensure lossless join property. A special attribute, the tuple-id attribute may be added to each schema to serve as a candidate key.

st_id	**fname**	**lname**	**dep_name**	**division**
S001	Abera	Tadese	IT	Regular
S002	Zinash	Demere	COTM	Weekend
S003	Mohammed	Ahmed	CS	Regular
S004	Adem	Ali	Accounting	Extension
S005	Hilina	Sitota	IT	Extension
S006	Zahra	Muktar	CS	Weekend

Following is an example student relation schema to illustrate how a horizontal and vertical schema can be formed.

student schema

Two possible horizontal fragmentation of the student relation can be: (let's name them ***stdent1*** and ***student2***)

First fragmentation: ***student1***

$\sigma_{division='Extension' \vee division='Weekend'}$***(student) = student1***

<u>**st_id**</u>	**fname**	**lname**	**dep_name**	**division**
S002	Zinash	Demere	COTM	Weekend
S004	Adem	Ali	Accounting	Extension
S005	Hilina	Sitota	IT	Extension
S006	Zahra	Muktar	CS	Weekend

Second fragmentation: ***student2***=

$\sigma_{division='Regular'}$***(student) = student2***

<u>**st_id**</u>	**fname**	**lname**	**dep_name**	**division**
S001	Abera	Tadese	IT	Regular
S003	Mohammed	Ahmed	CS	Regular

Two possible vertical fragmentation of the student relation can be: (let's name them ***list1*** and ***list2***)

list1= $\Pi_{tuple\_id,\ fname,\ lname}$(***student***)

tuple_id	**fname**	**lname**
1	Abera	Tadese
2	Zinash	Demere
3	Mohammed	Ahmed

4	Adem	Ali
5	Hilina	Sitota
6	Zahra	Muktar

$$\boldsymbol{List2} = \Pi_{tuple\_id,\ dep\_name,\ division}(\boldsymbol{student})$$

tuple_id	**dep_name**	**division**
1	IT	Regular
2	COTM	Weekend
3	CS	Regular
4	Accounting	Extension
5	IT	Extension
6	CS	Weekend

Advantages of horizontal fragmentation:

- allows parallel processing on fragments of a relation
- allows a relation to be split so that tuples are located where they are most frequently accessed

Advantages of vertical fragmentation:

- allows tuples to be split so that each part of the tuple is stored where it is most frequently accessed
- the tuple-id attribute allows efficient joining of vertical fragments
- allows parallel processing on a relation

Horizontal and vertical fragmentation can be mixed.

4.3. Distributed Query Processing

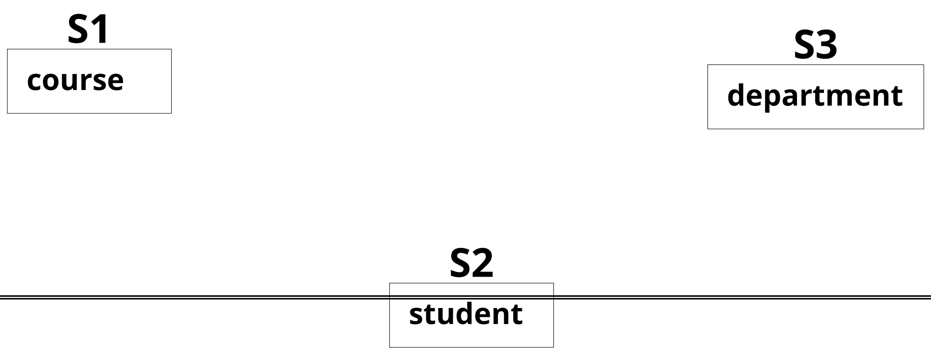
For centralized systems, the primary criterion for measuring the cost of a particular strategy is the number of disk accesses.

In a distributed system, other issues such as the following must be taken into account:

- The cost of a data transmission over the network.
- The potential gain in performance from having several sites process parts of the query in parallel.

Processing JOIN statements:

Consider the following relational algebra expression in which the three relations (***course, student, and department***) are neither replicated nor fragmented. Let's assume that the three relations are located in the three sites: S1, S2, and S3 as follows.



To process the algebraic expression:

For a query issued at site ***S1***, the system needs to produce the result at site ***S1***.

Possible processing strategies:

Strategy 1:

- Transfer copies of all three relations to site ***S1*** and choose a strategy for processing the entire locally at site ***S1***.

Strategy 2:

Transfer a copy of the ***course*** relation to site ***S2*** and compute

temp1 = course   student

at **S2**.

Then transfer **temp1** from **S2** to **S3**, and compute

temp2 = temp1   department

at **S3**. Then transfer the result **temp2** to **S1**.

Strategy 3:

- Devise similar strategies, exchanging the roles ***S1, S2, S3***

In selecting a strategy, one must consider the following factors:

- amount of data being transferred
- cost of transmitting a data block between sites
- relative processing speed at each site

4.4. Distributed Transaction Management and Recovery

In a distributed database system, each site has its own transaction manager. And a transaction may access data from several sites. The local transaction manager of each site has two responsibilities:

- Maintaining a log for recovery purposes
- Participating in coordinating the concurrent execution of the transactions executing at that site.

In addition to the local transaction manager, there is also transaction coordinator at each site performing the following activities:

- Starting the execution of transactions that originate at the site.
- Distributing subtransactions at appropriate sites for execution.
- Coordinating the termination of each transaction that originates at the site, which may result in the transaction being committed at all sites or aborted at all sites.

Atomicity of distributed transactions is assured by a commit protocol. A distributed transaction can either be committed at all sites or aborted at all the sites; it is not acceptable for it to be committed at some site and aborted in another site.

To recover transactions, the log information at each site is used.

Chapter-5- Object Oriented Database

5.1. Object Oriented Concepts

An object corresponds to an entity in entity-relational data model. Like ER models, object oriented data model is conceptual data modeling technique.

Objects in object oriented data models have the following things associated with them:

- A set of variables that contain the data for the object. The value of each variable is itself an object.
- A set of messages to which the object responds; each message may have zero, one, or more parameters.
- A set of methods, each of which is a body of code to implement a message; a method returns a value as the response to the message.

The physical representation of data is visible only to the implementer of the object. Messages and responses provide the only external interface to an object.

5.2. OODBMS Definitions

Following are some common terminologies regarding the design of OODBMS.

Messages and Methods:

The term message does not necessarily imply physical message passing. Messages can be implemented as procedure invocations.

Methods are programs written in general-purpose language with the following features:

- only variables in the object itself may be referenced directly
- data in other objects are referenced only by sending messages

Methods can be read-only or update methods. A read-only method does not change the value of the object.

Strictly speaking, every attribute of an entity must be represented by:

- a variable and
- two methods, one to read and the other to update the attribute
    - e.g., the attribute address is represented by a variable ***address*** and two messages ***get_address*** and ***set_address***.

For convenience, many object-oriented data models permit direct access to variables of other objects.

Objects and Classes

Similar objects are grouped into a class. Each object is referred as an instance of its class. All objects of a class have the same variable (with same data type), the same message interface, and the same method. Those objects differ by the value of their variables.

Classes are analogous to entities in relational databases.

A ***person*** class, for example, can be defined to have a number of people as object instances.

Following is a class definition for an employee class:

```
class employee {
    /*Variables */
         string   name;
         string   address;
         date     start-date;
         int      salary;
        /* Messages */
         int      annual-salary();
         string   get-name();
         string   get-address();
         int      set-address(string new-address);
         int      employment-length();
};
```

Methods to read and set the other variables (*start-date, salary…*) can also be added with strict encapsulation.

Methods are defined separately, i.e. outside the class definition. Following are, for example, two of the method definitions of the above class.

```
int employment-length() {
return today() – start-date;
}
```

```
int set-address(string new-address) {
address = new-address;
}
```

Inheritance:

Inheritance is the concept when a subclass inherits the definition of another general class. This meant that an object of a subclass need not carry its own definition of data and methods that are generic to the class of which it is a part; it can use/inherit the general class's data and methods. Doing so has advantages. It speeds up program development and also reduces program size. It also reduces the burden of the programmer by promoting code reusability.

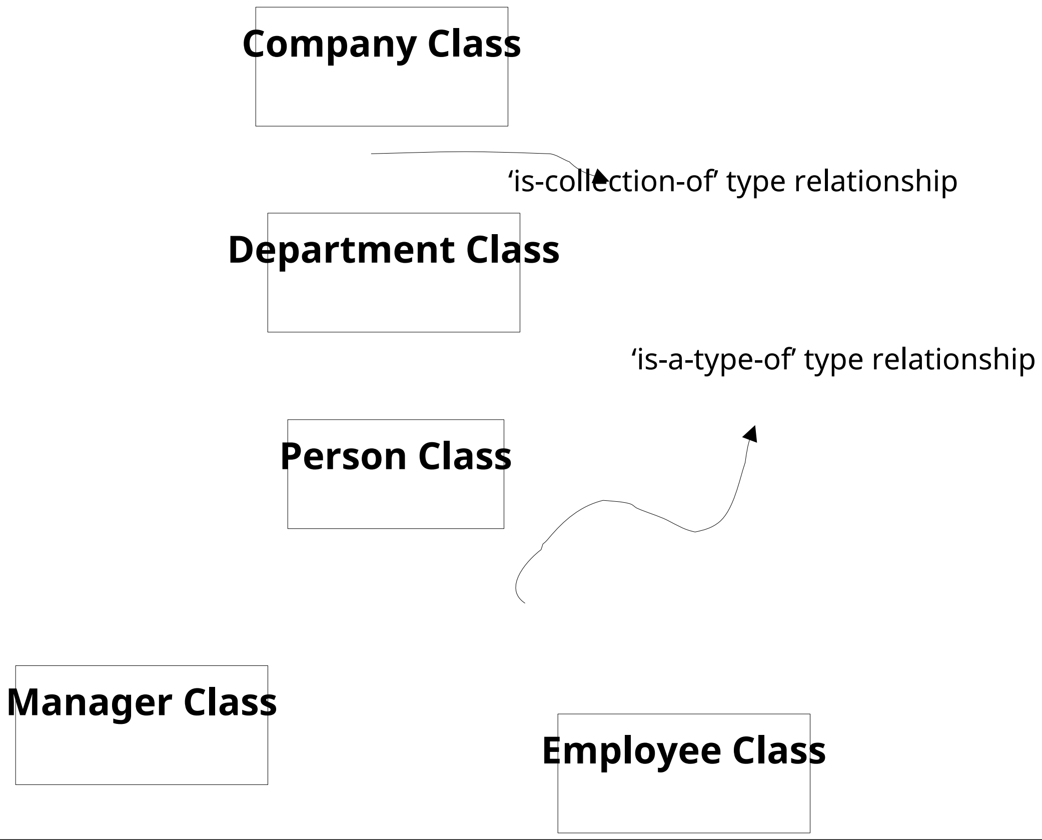
For example: You can define a ***car*** class and then you can define different types of cars such as ***minibus, midi-bus, bus, lorry*** etc by inheriting common variables and methods from car class.

5.3. OO Data modeling and E-R diagramming

An object database model provides a three-dimensional structure to data where any item in a database can be retrieved from any point very rapidly. Whereas the relational database model lends itself to retrieval of groups of records in two dimensions, the object database model is efficient for finding unique items. Consequently, the object database model performs poorly when retrieving more than a single item, at which the relational database model is proficient.

In object oriented data modeling, the classes are first identified. A class is just like entities in relational data modeling. After identifying those classes, their relationship is formed. But, in entity relationship modeling (ERD), the entities are identified first, and then the attributes, and finally the relationship between those entities are identified.

Following is a sample object oriented model for a company.


Company Class
'is-collection-of' type relationship
Department Class
'is-a-type-of' type relationship
Person Class
Manager Class
Employee Class

5.4. Object Identity

In object oriented database design, object data items are managed in terms of objects. Each object can have the same attribute value. But this doesn't mean that the two objects are the same. There is a mechanism, known as object identity, to identify each object from other objects. The object identity acts like primary key in relational databases. The object identity is viewed as a reference or pointer to the object.

5.5. Object Relational Databases

Object-relational databases are constructed based on an object-relational data model.

This model extends the relational model by providing a rich data type for handling complex objects and object orientation. Because most sophisticated database applications need to handle complex objects and structures, object-relational databases are becoming increasingly popular in industry and applications.

Conceptually, the object-relational data model inherits the essential concepts of object-oriented databases, where, in general terms, each entity is considered as an object.