

Chapter 1: Transaction Processing Concepts

Introduction

This chapter discusses the various aspects of transaction processing. It also studies the low-level tasks included in a transaction, the transaction states and properties of a transaction. In the last portion, it will look over schedules and Serializability of schedules.

Transaction and System Concepts

A **transaction** is an event which occurs on the database. Generally, a transaction **reads** a value from the database or **writes** a value to the database. Although a transaction can both read and write on the database, there are some fundamental differences between these two classes of operations. A transaction involving only data retrieval without any data update is called read-only transaction. A read operation does not change the image of the database in any way. But a write operation, whether performed without the intention or intention of inserting, updating or deleting data from the database, changes the image of the database. That is, these transactions bring the database from old image to new image, called the **Before Image** or **BFIM** and **After Image** or **AFIM**.

Database transaction is a collection of SQL queries which forms a logical one task. For a transaction to be completed successfully all SQL queries have to run successfully. It is **an atomic process** that is either performed into completion entirely or is not performed at all. Database transaction executes either **all or none**, so for example if your database transaction contains 4 SQL queries and one of them fails then change made by other 3 queries should be rolled back. This way your database always remains consistent. The transaction is implemented in the database using **SQL keyword transaction, commit, and rollback**. **Commit** writes the changes made by transaction into database and **rollback** removes temporary changes logged in transaction log by database transaction.

On database transactions, each high-level operation (**read ()** or **write ()**) can be divided into a number of low level tasks or operations. For example, a **data update operation** can be divided into three tasks –

- **read_item()** – reads data item from storage to main memory. Which includes getting the disk block location too?
- **modify_item()** – change value of item in the main memory. Manipulate the data and switch the old value with new value on buffer
- **write_item()** – write the modified value from main memory to storage.

Database access is restricted to **read_item()** and **write_item()** operations. Likewise, for all transactions, read and write forms the basic database operations.

Why transaction is required in database

Your database records need to exist in a consistent state. After an operation, the database records should move from **one consistent state** to **another consistent state**. That is why we need a transaction. The database is used to store data required by real life application e.g. Banking, Healthcare, Finance etc. All your money stored in banks is stored in the database, all your account is stored in the database and many applications constantly work on these data. In order to protect data and keep it consistent, any changes in this data need to be done in a transaction so that even in the case of failure data remain in the previous state before the start of a transaction. Consider a Classical example of ATM (Automated Tailor Machine); we all use to withdraw and transfer money by using ATM. If you break withdrawal operation into individual steps you will find:

- 1) Verify account details.
- 2) Accept withdrawal request
- 3) Check balance
- 4) Update balance
- 5) Dispense money

Suppose your account balance is 1000Birr and you make a withdrawal request of 900Birr. At fourth step, your balance is updated to 900Birr and ATM machine stops working due to power outage. **What will happen?**

Once power comes back and you again tried to withdraw money you surprised by seeing your balance just 100Birr instead of 1000Birr. This is not acceptable by any person in the world. So, we need a transaction to perform such task. If SQL statements would have been executed inside a transaction in database balance would be either 100Birr until money has been dispensed or 1000Birr if money has not been dispensed.

Transaction Operations

The low-level operations performed in a transaction are –

- **Begin transaction** – A marker that specifies **start** of transaction execution.
- **read_item or write_item** – Database operations that may be **interleaved with main memory** operations as a part of transaction.
- **End transaction** – A marker that specifies **end** of transaction.
- **Commit** – A signal to specify that the transaction has been successfully completed in its entirety and will **not be undone**.
- **Rollback** – A signal to specify that the transaction has been unsuccessful and so all temporary changes in the database are **undone**. A committed transaction cannot be rolled back.

Transaction States

A transaction may go through a subset of five states:

- Active
- Partially committed
- Committed
- Failed and
- Aborted

Active – the initial state where the transaction enters is the active state. The transaction remains in this state while it is executing read, write or other operations. This is the first state of transaction and here the transaction is being executed. For example, updating or inserting or deleting a record is done here. But it is still not saved to the database. Once the transaction starts executing from the first instruction begin_transaction, the transaction will be considered in active state. During this state, it performs operations READ and WRITE on some data items.

From active state, a transaction can go into one of two states, a partially committed state or a failed state.

Partially Committed – after the execution of final statement.

The transaction enters this state after the last statement of the transaction has been executed.

- This is also an **execution phase where last step in the transaction is executed.**

But data is still not saved to the database. If you calculate total marks, on final display the total marks step is executed in this state. This is the state of a transaction that successfully executing its last instruction. That means, if an active transaction reaches and executes the COMMIT statement, then the transaction is said to be in partially committed state.

From partially committed state, a transaction can go into one of two states, a committed state or a failed state.

Committed – After successful completion of transaction.

At partially committed state the database recovery system will perform certain actions to ensure that a failure at this stage should not cause loss of any updates made by the executing transaction. If the current transaction passed this check, then the transaction reaches committed state.

From committed state, a transaction can go into terminated state.

Failed – If any failure occurs.

The transaction goes from partially committed state or active state to failed state when it is discovered that normal execution can no longer proceed or system checks fail. If a

transaction cannot proceed to the execution state because of the failure of the system or database, then the transaction is said to be in failed state. In the total mark calculation example, if the database is not able fire a query to fetch the marks, i.e.; very first step of transaction, then the transaction will fail to execute. While a transaction is in the active state or in the partially committed state, the issues like transaction failure, user aborting the transaction, concurrency control issues, or any other failure, would happen. If any of these issues are raised, then the execution of the transaction can no longer proceed. At this stage a transaction will go into a failed state.

From failed state, a transaction can go into only aborted state.

Aborted – After rolled back to the old consistent state.

This is the state after the transaction has been rolled back after failure and the database has been restored to its state that was before the transaction began. If a transaction is failed to execute, then the database recovery system will make sure that the database is in its previous consistent state. It brings the database to consistent state by aborting or rolling back the transaction. If the transaction fails in the middle of the transaction, all the executed transactions are rolled back to it consistent state before executing the transaction. Once the transaction is aborted it is either restarted to execute again or fully killed by the DBMS. After the failed state, all the changes made by the transaction has to be rolled back and the database has to be restored to its state prior to the start of the

transaction. If these actions are completed by the DBMS then the transaction considered to be in aborted state.

From aborted state, a transaction can go into terminated state.

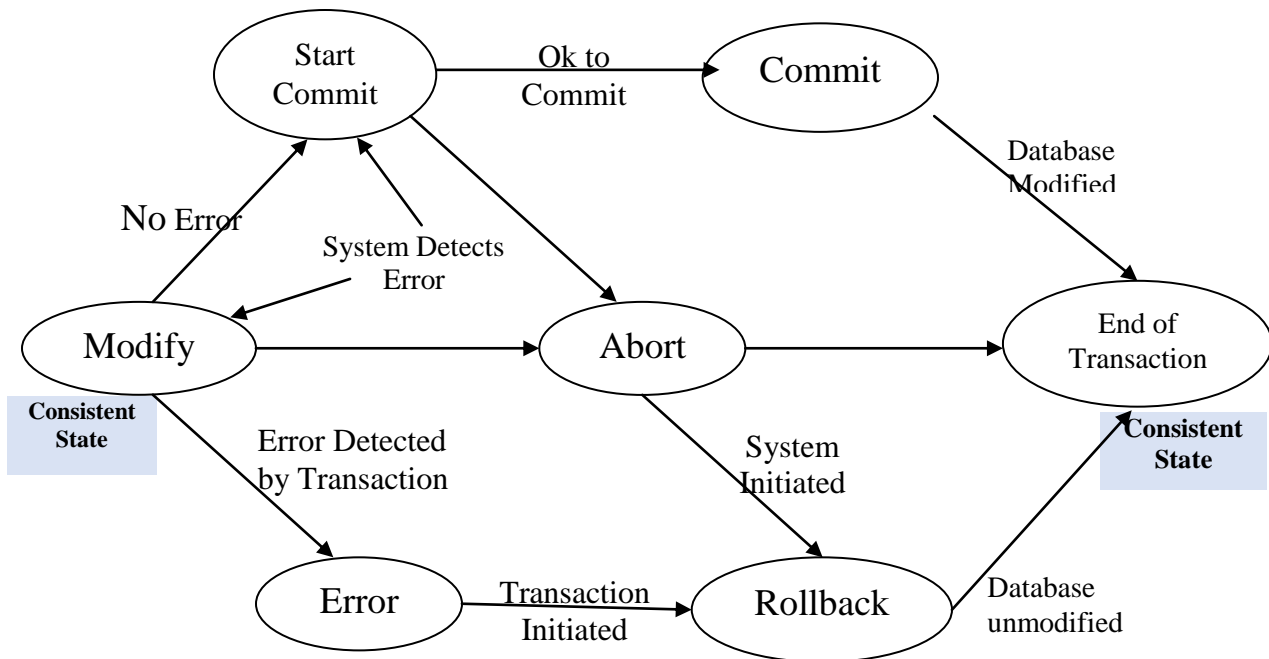
A transaction is an atomic operation from the users' perspective. But it has a collection of operations and it can have a number of states during its execution.

Therefore, A transaction can end/terminate in three possible ways.

1. **Successful Termination:** when a transaction completes the execution of all operations in it and reaches the COMMIT command.

Suicidal (ራስን ማጥፋት) Termination: when the transaction detects an error during its processing and decide to abrupt(በድንገት) itself before the end of the transaction and perform a ROLL BACK

2. **Murderous (ገዳይ) Termination:** When the DBMS or the system force the execution to abort for any reason.



Desirable Properties of Transactions

Every transaction, for whatever purpose it is being used, has the following four properties: Atomicity, Consistency, Isolation, and Durability. Taking the initial letters of these four properties collectively it is called the **ACID Properties**. Any transaction must maintain the ACID properties.

✓ **Atomicity:** A transaction is an atomic unit of processing; it is either performed in its own completeness or not performed at all.

✓ **Consistency protection:** A correct execution of the transaction must take the database from one consistent state to another.

✓ **Isolation:** A transaction should **not** make its updates visible to other transactions until it is committed.

✓ **Durability or permanency:** Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

Atomicity – This property states that a transaction is an atomic unit of processing, that is, either it is performed in its entirety or not performed at all. No partial update should exist. This property states that each transaction must be considered as a single unit. No transaction in the database is left half completed. Database should be in a state either before the transaction execution or after the transaction execution. It should not be in a state ‘executing’.

In our example above, the transaction should not be left at any one of the step above. All the 5 steps have to be either completed or none of the step has to be completed. If a transaction is failed to execute any step, then it has to rollback all the previous steps and come to the state before the transaction or it should try to complete the failed step and further steps to complete whole transaction.

Say for example, we have two accounts A and B, each containing Birr 1000. We now start a transaction to deposit Birr 100 from account A to Account B.

Read A;

$A = A - 100;$

Write A;

Read B;

$B = B + 100;$

Write B;

The transaction has 6 instructions to extract the amount from A and submit it to B. The AFIM will show Birr 900 in A and Birr 1100 in B.

Now, suppose there is a power failure just after instruction 3 (Write A) has been complete. What happens now? After the system recovers the AFIM will show Birr 900 in A, but the same Birr 1000 in B. It would be said that Birr 100 evaporated in the air for the power failure. Clearly such a situation is not acceptable.

The solution is to keep every value calculated by the instruction of the transaction not in any stable storage (hard disc) but in a volatile storage (RAM), until the transaction completes its last instruction. When we see that, there has not been any error we do something known as a **COMMIT** operation. Its job is to write every temporarily

calculated value from the volatile storage on to the stable storage. In this way, even if power fails at instruction 3, the post recovery image of the database will show accounts A and B both containing Birr 1000, as if the failed transaction had never occurred. The Atomicity property ensures that.

Consistency – A transaction should take the database from one consistent state to another consistent state. It should not adversely affect any data item in the database. Any transaction should not inject any incorrect or unwanted data into the database. It should maintain the consistency of the database.

In above example, while calculating the balance, it should not perform any other action like inserting or updating or delete. It should also not pick balance of other customers. It should be picking the amount for the A and B customers and adjust their balance. Hence it maintains the consistency of the database.

Isolation – A transaction should be executed as if it is the only one in the system. There should not be any interference from the other concurrent transactions that are simultaneously running. If there are multiple transactions executing simultaneously, then all the transaction should be processed as if they are single transaction. But individual transaction in it should not alter or affect the other transaction. That means each transaction should be executed as if they are independent.

There are several ways to achieve this and the most popular one is using some kind of **locking mechanism**. Locking states that a transaction must first lock the data item that it wishes to access, and release the lock when the accessing is no longer required. Once a transaction locks the data item, other transactions wishing to access the same data item must wait until the lock is released.

For example, account A is having a balance of 400Birr and it is transferring 100Birr to account B & C both. So, we have two transactions here. Let's say these transactions run concurrently and both the transactions read 400Birr balance; in that case the final balance of A would be 300Birr instead of 200Birr. This is wrong. If the transaction were to run in isolation, then the second transaction would have read the correct balance 300Birr (before debiting 100Birr) once the first transaction went successful.

Transactions are concurrency control mechanisms, and they deliver consistency even when being interleaved. Isolation brings us the benefit of hiding uncommitted state changes from the outside world, as failing transactions shouldn't ever corrupt the state of the system. Isolation is achieved through concurrency control using pessimistic or optimistic locking mechanisms.

Durability – If a committed transaction brings about a change, that change should be durable in the database and not lost in case of any failure. The database should be strong enough to handle any system failure. It should not be working for single transaction

alone. It should be able to handle multiple transactions too. If there is any set of insert /update, then it should be able to handle and commit to the database. If there is any failure, the database should be able to recover it to the consistent state.

As we have seen in the explanation of the Atomicity property, the transaction, if completes successfully, is committed. Once the COMMIT is done, the changes which the transaction has made to the database are immediately written into permanent storage. So, after the transaction has been committed successfully, there is no question of any loss of information even if the power fails. Committing a transaction guarantees that the AFIM has been reached.

There are several ways Atomicity and Durability can be implemented. One of them is called **Shadow Copy**. In this scheme a database pointer is used to point to the BFIM of the database. During the transaction, all the temporary changes are recorded into a Shadow Copy, which is an exact copy of the original database plus the changes made by the transaction, which is the AFIM. Now, if the transaction is required to COMMIT, then the database pointer is updated to point to the AFIM copy, and the BFIM copy is discarded. On the other hand, if the transaction is not committed, then the database pointer is not updated. It keeps pointing to the BFIM, and the AFIM is discarded. This is a simple scheme, but takes a lot of memory space and time to implement.

If you study carefully, you can understand that **Atomicity** and **Durability** is essentially the same thing, just as **Consistency** and **Isolation** is essentially the same thing.

A successful transaction must permanently change the state of a system, and before ending it, the **state changes are recorded in a persisted transaction log**. If our system is suddenly affected by a system crash or a power outage, then all unfinished committed transactions may be replayed.

Database users Environment

A DBMS can support many different types of databases. Databases can be classified according to:

- The number of users,
 - The database location, and
 - The expected type and extent of use.
- ✚ The number of users determines whether the database is classified as a
- single-user or
 - multiuser.

SINGLE-USER DBMS

A single-user can **access the database at one point of time.** If user A is using the database user B or C must **wait** until user A is through. These types of systems are optimized for a personal desktop experience, not for multiple users of the system at the same time.

❖ **Properties:-**

- All the resources are always available for the user to work.
- The architecture implemented is both One or Two tier(एक या दो).
● Both the application and physical layer are operated by user.
- For Ex: Standalone Personal Computers, Microsoft Access, etc.

In a single-user environment, the workspace repository resides on the local machine, and can be **accessed by the owner of the machine only.** Limited facilities exist for sharing work with other users.

MULTI USER DBMS

Multi user DBMS are **the systems that support two or more simultaneous users.** These type of database are familiar in an enterprise database and workgroup database environment. All **mainframes and minicomputers** are multi-user systems, but most personal computers and workstations are not.

- A multiuser database may exist on a single machine, such as a mainframe or other powerful computer, or it may be distributed and exist on multiple computers.
- Multiuser databases are accessible from multiple computers simultaneously
- Multiuser databases are accessible from multiple computers simultaneously.
- Many people can be working together to update information at the same time.
- All employees have access to the most up-to-date information all of the time.
- Customers have instant access to their personal information held by companies.

In a multiuser environment, the workspace repository resides on a database server, and can be accessed by any user with appropriate database privileges.

COMPARISION BETWEEN SINGLE USER AND MULTIPLE USER DATABASE.

Single-User	Multiuser
<ul style="list-style-type: none">• Access Restricted to single user at a time• Database Structure relatively simple• Switching between projects is difficult as single schemas repository is used	<ul style="list-style-type: none">• Access can share by Multiple user at a time• Complex Database Structure due to shared access Complexity Increases with the structure of database• Switching between projects is easy as different schemas repositories are used• Access sharing makes it difficult, sometimes causes deadlock

<ul style="list-style-type: none"> • Committing change in the database without causing deadlock changes • Infrastructure cost is minimum 	<ul style="list-style-type: none"> • Infrastructure cost is higher such as Servers, Networks etc • Maintenance is also overhead expense • Wastage of CPU and resource when Optimum usage/optimization of the user/application remain idle
--	--

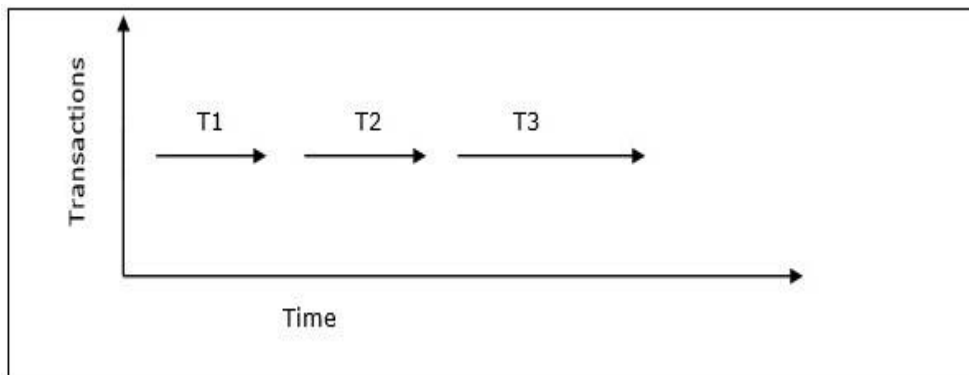
Schedules and Recoverability

When multiple transactions are being executed by the operating system in a multiprogramming environment, there are possibilities that instructions of one transactions are interleaved with some other transaction. A **schedule** is a collection of many transactions which is implemented as a unit.

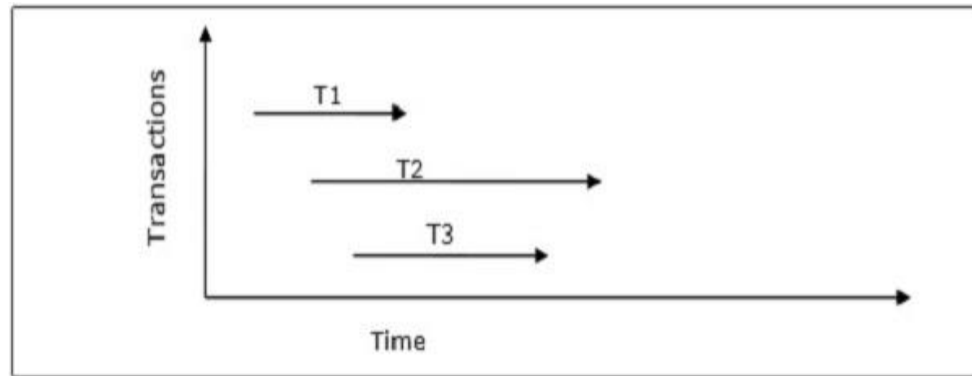
Schedule – A chronological execution sequence of a transaction is called a schedule. A schedule can have many transactions in it, each comprising of a number of instructions/tasks. Depending upon how these transactions are arranged in within a schedule, a **schedule can be of two types:**

- **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. This type of schedule is called a serial schedule, as transactions are executed in a serial manner. In a serial schedule, at any point of time, only one transaction is active, due to this, **there is no overlapping of transactions**. Therefore, **in a serial schedule, only one transaction at a time is active**—the commit (or abort) of the active transaction initiates execution of the next transaction. **No interleaving occurs in a serial schedule.** This is depicted in the following graph –



Parallel Schedules(non-serial schedule) – In parallel schedules, **more than one transactions are active simultaneously**, i.e. the transactions contain operations that **overlap at time**. This parallel execution brings a **Concurrent transaction**; the transactions are executed in a preemptive, time shared method. This is depicted in the following graph –



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.

In Serial schedule, there is **no question of sharing a single data item** among many transactions, because not more than a single transaction is executing at any point of time.

However, **a serial schedule** is inefficient in the sense that the transactions suffer for having a

a serial schedule_____:

- **longer waiting time** and response time,

- low amount of resource utilization.

In concurrent schedule,

- CPU time is shared among two or more transactions in order to run them concurrently.

However, this creates the possibility that more than one transaction may need to access a single data item for read/write purpose and the database could contain inconsistent value if such accesses are not handled properly. Let's explain with the help of an example.

Let's consider there are two transactions T1 and T2, whose instruction sets are given as following. T1 is the same as we have seen earlier, while T2 is a new transaction.

T1

Read A;

$A = A - 100;$

Write A;

Read B;

$B = B + 100;$

Write B;

T2

Read A;

Temp = A * 0.1;

Read C;

C = C + Temp;

Write C;

T2 is a new transaction which deposits to account C 10% of the amount in account A.

If we prepare a serial schedule, then either T1 will completely finish before T2 can begin, or T2 will completely finish before T1 can begin. However, if we want to create a concurrent schedule, then some Context Switching need to be made, so that some portion of T1 will be executed, then some portion of T2 will be executed and so on. For example say we have prepared the following concurrent schedule.

T1**T2**

Read A;

```
A = A - 100;
Write A;

Read A;
Temp = A * 0.1;
Read C;
C = C + Temp;
Write C;

Read B;
B = B + 100;
Write B;
```

No problem here. We have made some Context Switching in this Schedule, the first one after executing the third instruction of T1, and after executing the last statement of T2. T1 first deducts Birr 100 from A and writes the new value of Birr 900 into A. T2 reads the value of A, calculates the value of Temp to be Birr 90 and adds the value to C. The remaining part of T1 is executed and Birr 100 is added to B.

It is clear that a proper Context Switching is very important in order to maintain the Consistency and Isolation properties of the transactions. But let us take another example where a wrong Context Switching can bring about disaster. Consider the following example involving the same T1 and T2

T1**T2**

```
Read A;  
  
A = A - 100;  
  
Read A;  
  
Temp = A * 0.1;  
  
Read C;  
  
C = C + Temp;  
  
Write C;  
  
Write A;  
  
Read B;  
  
B = B + 100;  
  
Write B;
```

This schedule is wrong, because we have made the switching at the second instruction of T1. The result is very confusing. If we consider accounts A and B both containing Birr 1000 each, then the result of this schedule should have left Birr 900 in A, Birr 1100 in B and add Birr 90 in C (as C should be increased by 10% of the amount in A). But in this wrong schedule, the Context Switching is being performed before the new value of Birr 900/- has been updated in A. T2 reads the old value of A, which is still Birr 1000, and deposits Birr 100 in C. C makes an unjust gain of Birr 10 out of nowhere.

In the above example, we detected the error simple by examining the schedule and applying common sense. But there must be some well-formed rules regarding how to arrange instructions of the transactions to create error free concurrent schedules.

Problems Associated with Concurrent Transaction Processing

Although two transactions may be correct in themselves, interleaving of operations may produce an incorrect result which needs control over access. Having a concurrent transaction processing, one can enhance the *throughput* of the system. As reading and writing is performed from and on secondary storage, the system will not be idle during these operations, if there is a concurrent processing.

Every transaction should be correct by themselves, but this would not guarantee that the interleaving of these transactions will produce a correct result. The three potential problems caused by concurrency are:

- Lost Update Problem..... **ww conflict**
- Uncommitted Dependency Problem(dirty read)..... **Rw conflict**
- Inconsistent Analysis Problem..... **wR conflict**

Lost Update Problem

Successfully completed update on a data set by **one transaction is overridden by another transaction/user.**

E.g. Account with balance $A=100$.

- T1 reads the account A
- T1 withdraws 10 from A
- T1 makes the update in the Database
- T2 reads the account A
- T2 adds 100 on A
- T2 makes the update in the Database
- In the above case, if done one after the other (serially) then we have no problem.
 - If the execution is T1 followed by T2 then $A=190$
 - If the execution is T2 followed by T1 then $A=190$
- But if they start at the same time in the following sequence:
 - T1 reads the account $A=100$
 - T1 withdraws 10 making the balance $A=90$
 - T2 reads the account $A=100$
 - T2 adds 100 making $A=200$
 - T1 makes the update in the Database $A=90$

- T2 makes the update in the Database $A=200$
- After the successful completion of the operation in this schedule, the final value of A will be 200 which override the update made by the first transaction that changed the value from 100 to 90.

Uncommitted Dependency Problem – Temporary update Problem (RW conflict) or dirty read

- Occurs when one transaction can see intermediate results of another transaction before it is committed.

E.g.

- T2 increases 100 making it 200 but then aborts the transaction before it is committed. T1 gets 200, subtracts 10 and make it 190. But the actual balance should be 90

Inconsistent Analysis Problem – incorrect summery

Occurs when transaction reads several values but second transaction updates some of them during execution and before the completion of the first.

E.g.

- T2 would like to add the values of A=10, B=20 and C=30. after the values are read by T2 and before its completion, T1 updates the value of B to be 50. at the end of the execution of the two transactions T2 will come up with the sum of 60 while it should be 90 since B is updated to 50.

These concurrent transactions should be in such a way to avoid any interference between them. This demands a new principle in transaction processing, which is Serializability of the schedule of execution of multiple transactions.

Schedules and Conflicts

In a system with a number of simultaneous transactions, a **schedule** is the total order of **execution of operations**. Given a schedule S comprising of n transactions, say T1, T2, T3.....Tn; for any transaction Ti, the operations in Ti must execute as laid down in the schedule S.

Conflicts in Schedules

In a schedule comprising of multiple transactions, a **conflict** occurs when two active **transactions perform non-compatible operations**.

Two operations are said to be in conflict, when all of the following three conditions exists simultaneously –

- The two operations are parts of different transactions.
- Both the operations access the same data item.
- At least one of the operations is a `write_item()` operation, i.e. it tries to modify the data item.

Serializability

A **serializable schedule** of 'n' transactions is a parallel schedule which is equivalent to a serial schedule comprising of the same 'n' transactions. A serializable schedule contains the correctness of serial schedule while ascertaining better CPU utilization of parallel schedule. The definition of *serializable schedule* is as follows: A schedule *S* of *n* transactions is **serializable** if it is *equivalent to some serial schedule* of the same *n* transactions.



Chapter 3: concurrency control

Concurrency Control (CC) Techniques

Concurrency Control is the process of managing simultaneous operations on the database **without having them interfere** with one another. Prevents interference when two or more users are accessing database simultaneously and **at least one** is updating data. Although two transactions may be correct in themselves, interleaving of operations may produce an incorrect result.

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.

Why Concurrency Control is needed?

- The lost update problem
- The temporary update (dirty read) problem
- The incorrect summary problem
- The unrepeated read

Concurrency control **protocols** are (lock, time stamp, and optimistic) can be broadly divided into two categories:

- Pessimistic concurrency control
- Optimistic concurrency control

Pessimistic concurrency control

A system **of locks prevents users from modifying data in** a way that affects other users. After a user performs an action that causes a lock to be applied, other users cannot perform actions that would conflict with the lock until the owner releases it. This is called pessimistic control because it is mainly used in environments where there is high contention for data, **where the cost of protecting data with locks is less than the cost of rolling back transactions if concurrency conflicts occur**. Locking and Time stamping are conservative approaches: **delay transactions in case they conflict with other transactions.**

Optimistic concurrency control

In optimistic concurrency control, **users do not lock data** when they read it. When a user updates data, the system checks to see if another user changed the data after it was read. If another user updated the data, an error is raised. Typically, the user receiving the error rolls back the transaction and starts over. This is called

optimistic because it is mainly used in environments where **there is low contention for data, and where the cost of occasionally rolling back a transaction is lower than the cost of locking data when read.** The optimistic approach allows us to proceed and check conflicts at the end. **Optimistic methods assume conflict is rare and only check for conflicts at commit.**

The pros and cons of the pessimistic and optimistic concurrency control mechanisms

Both pessimistic and optimistic concurrency control mechanisms provide different performance, e.g., the different average transaction completion rates or throughput, depending on transaction type's mix, computing level of parallelism, and other events. Both have pros and cons as shown below:

For pessimistic concurrency control, the strength is:

- Guarantee that all transactions can be executed correctly.
- Data is properly consistent by either rolling back to the previous state (Abort operation) or new content (Commit operation) when the transaction conflict is cleared.
- Database is relatively stable and reliable.

Its weakness is:

- Transactions are slow due to the delay by locking or time-stamping event.
- Runtime is longer. Transaction latency increases significantly.
- Throughput or the amount of work (e.g. read/write, update, rollback operations, etc.) is reduced.

For optimistic concurrency control, the strength is:

- Transactions are executed more efficiently.
- Data content is relatively safe.
- Throughput is much higher.

Its weakness is:

- There is a risk of data interference among concurrent transactions since it transactions conflict may occur during execution. In this case, data is no longer correct.
- Database may have some hidden errors with inconsistent data; even conflict check is performed at the end of transactions.
- Transactions may be in deadlock that causes the system to hang.

Locking Techniques for Concurrency Control

A **LOCK** is a mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution. Locks are one way of enforcing concurrency control policies. Transaction uses locks to deny access to other transactions and so prevent incorrect updates.

Lock prevents another transaction from modifying item or even reading it, in the case of a write lock. Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

Types of a Lock

Binary Locks: Have only two states: locked (1) or unlocked (0)

- **Lock (X):** If a transaction T1 applies Lock on data item X, then X is locked and it is not available to any other transaction.
- **Unlock (X):** T1 Unlocks X. X is available to other transactions.

Shared/Exclusive Locks

Shared lock: A Read operation does not change the value of a data item. Hence a data item can be read by two different transactions simultaneously under share lock

mode. So only to read a data item T1 will do: *Share lock (X), then Read (X), and finally Unlock (X)*. shared lock is also called read lock.

Exclusive lock: A write operation changes the value of the data item. Hence two write operations from two different transactions or a write from T1 and a read from T2 are not allowed. A data item can be modified only under Exclusive lock. To modify a data item T1 will do: *Exclusive lock (X), then Write (X) and finally Unlock (X)*. Exclusive lock is also called write lock.

- ❖ The relationship between Shared and Exclusive Lock can be represented by the following table which is known as **Lock Matrix**.

		Locks already Existing	
Lock to be Granted		S	X
	S	True	False
	X	False	False

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions.
- Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted

When these locks are applied, then a transaction must behave in a special way.

This special behavior of a transaction is referred to as *well-formed*.

Well-formed: A transaction is well- formed if it does not lock a locked data item and it does not try to unlock an unlocked data item.

Locking - Basic Rules

- If transaction has shared lock on item, can read but not update item.
- If transaction has exclusive lock on item, can both read and update item.
- Reads cannot conflict, so more than one transaction can hold shared locks simultaneously on same item.
- Exclusive lock gives transaction exclusive access to that item.
- Some systems allow transaction to upgrade a shared lock to an exclusive lock, or vice-versa.

Examples: T1 and T2 are two transactions. They are executed under locking as follows. T1 locks A in exclusive mode. When T2 wants to lock A, it finds it locked by T1 so T2 waits for Unlock on A by T1. When A is released then T2 locks A and begins execution.

Example of a transaction performing locking:

T1:

lock-S(A);

read (A);

Locking as shown here is not sufficient to guarantee serializability — if A and B get updated in-between the

unlock(A);	read of A and B, the displayed sum would be wrong.
lock-S(B);	
read (B);	
unlock(B);	
display(A+B)	

Suppose a lock on a data item is applied, the data item is processed and it is unlocked immediately after reading/writing is completed as follows.

Locking methods: problems

Deadlock: Deadlock refers to a particular situation where two or more processes are each waiting for another to release a resource, or more than two processes are waiting for resources in a circular chain. Deadlock is a common problem in multiprocessing where many processes share a specific type of mutually exclusive resource. A deadlock that may result when two (or more) transactions are each waiting for locks held by the other to be released.

T1	T2
lock-X(B);	
read (B);	
B =B-50;	
WRITE(B);	

lock-S(A);

read (A);

lock-S(B);

lock-X(A);

Neither T1 nor T2 can make progress — executing lock-S(B) causes T2 to wait for T1 to release its lock on B, while executing lock-X(A) causes T1 to wait for T2 to release its lock on A.

Such a situation is called a deadlock. To handle a deadlock one of T1 or T2 must be rolled back and its locks released

Deadlock - possible solutions

Only one way to break deadlock: **abort one or more of the transactions** in the deadlock. Deadlock should be transparent to user, so DBMS should restart transaction(s).

T3	T4
lock - X(B)	
read(B)	
B := B-50	
Write(B)	
	lock -S(A)
	read(A)
	lock - S(B)
	wait...
	wait...
lock-X(A)	
wait...	
wait...	

Three general techniques for handling deadlock:

- Deadlock prevention.
- Timeout
- Deadlock detection and recovery.

Timeout

The deadlock detection could be done using the technique of TIMEOUT. Every transaction will be given a time to wait in case of deadlock. If a transaction waits for the predefined period of time in idle mode, the DBMS will assume that deadlock occurred and it will abort and restart the transaction.

Concurrency Control Based on Timestamp Ordering

Timestamp: a unique identifier created by DBMS that indicates relative starting time of a transaction. This protocol uses either system time or logical counter as a timestamp. Therefore, can be generated by:

- using system clock at time transaction started, or
- Incrementing a logical counter every time a new transaction starts.

Time-stamping → A concurrency control protocol that orders transactions in such a way that older transactions, transactions with smaller time stamps, get priority in the event of conflict.

- Transactions ordered globally base do their timestamp so that older transactions, transactions with earlier timestamps, get priority in the event of conflict.
- Conflict is resolved by rolling back and restarting transaction.
- Since there is no need to use lock there will be *No Deadlock*.

In timestamp ordering, the schedule is equivalent to the particular serial order that corresponds to the order of the transaction timestamps. To implement this scheme, every transaction will be given a timestamp which is a unique identifier of a transaction. If T_i came to processing prior to T_j then TS of T_j will be larger than TS of T_i . Again, each data item will have a timestamp for Read and Write.

- $WTS(A)$ which denotes the largest timestamp of any transaction that successfully executed $Write(A)$
- $RTS(A)$ which denotes the largest timestamp of any transaction that successfully executed $Read(A)$

These timestamps are updated whenever a new $Read(A)$ or $Write(A)$ instruction is executed.

Read/write proceeds only if last update on that data item was carried out by an older transaction. Otherwise, transaction requesting read/write is restarted and given a new 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 that any conflicting read and write operations are executed in the timestamp order.

- a transaction waits for a lock only for a specified amount of time. If the lock has not been granted within that time, the transaction is rolled back and restarted

- Thus, deadlocks are not possible
- simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

Cascading Rollback

Whenever some transaction T tries to issue a Read_Item(X) or a Write_Item(X) operation, the basic timestamp ordering algorithm compares the timestamp of T with the read timestamp and the write timestamp of X to ensure that the timestamp order of execution of the transactions is not violated. If the timestamp order is violated by the operation, then transaction T will violate the equivalent serial schedule, so T is aborted. Then T is resubmitted to the system as a new transaction with new timestamp. If T is aborted and rolled back, any transaction T_i that may have used a value written by T must also be rolled back. Similarly, any transaction T_j that may have used a value written by T_i must also be rolled back, and so on. This effect is known as ***cascading rollback***.

Validation (Optimistic) Concurrency Control Technique

Optimistic Technique

- Locking and assigning and checking timestamp values may be unnecessary for some transactions
- Assumes that conflict is rare.
- When transaction reaches the level of executing commit, a check is performed to determine whether conflict has occurred. If there is a conflict, transaction is rolled back and restarted.
- Based on assumption that conflict is rare and more efficient to let transactions proceed without delays to ensure serializability.
- At commit, check is made to determine whether conflict has occurred.
- If there is a conflict, transaction must be rolled back and restarted.
- Potentially allows greater concurrency than traditional protocols.
- Three phases:
 1. Read
 2. Validation
 3. Write

1. Optimistic Techniques - Read Phase

- Extends from start until immediately before commit. Also, called

Execution Phase – A transaction fetches data items to memory and performs operations upon them.

- Transaction reads values from database and stores them in local variables. Updates are applied to a local copy of the data.

2. Optimistic Techniques - Validation Phase

- Follows the read phase.
- For read-only transaction, checks that data read are still current values. If no interference, transaction is committed, else aborted and restarted.
- For update transaction, checks transaction leaves database in a consistent state, with serializability maintained.

3. Optimistic Techniques - Write Phase

- Follows successful validation phase for update transactions.
- Updates made to local copy are applied to the database. Also called **Commit Phase** – A transaction writes back modified data item in memory to the disk

Schedule		
T1	T2	Concurrency control Manger
Lock-X(B)		Gant-X(B,T1)
Read (B,b)		

b= b-50 Write(B,b) Unlock(B)		
	Lock-S(A) Read (A,a) Unlock (A) b= b-50	Grant-S(A,T2)
	Lock-S(B) Read (B,b) Unlock(B) display(a+b)	Grant-S(B,T2)
Lock-X(A) Read (A,a) a= a+50 Write(A,a) Unlock(A)		Gant-X(A,T1)

Chapter 4: Database Recovery Techniques

Database Recovery Concepts

Types of failures

There are various types of failure that may occur in a system, each of which needs to be dealt with in a different manner.

There are many different types of failure that can affect database processing, each of which has to be dealt with in a different manner. Some **failures affect main memory only**, while others involve nonvolatile (secondary) storage.

- **Transactions may fail because of:**
 - ▶ **Logical errors:** transaction cannot complete due to some internal error condition
 - ▶ **System errors:** the database system must terminate an active transaction due to an error condition (e.g., incorrect input, **deadlock**)

Among the causes of failure are:

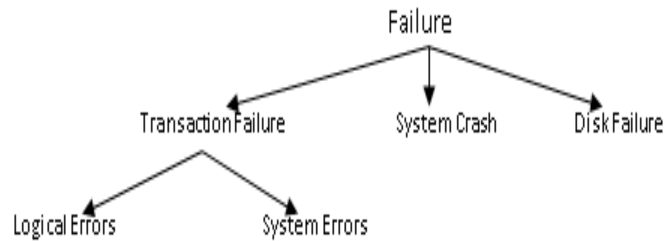
- **System crashes failures:** due to hardware or software errors, **resulting in loss of main memory;**
- **Media failures,** such as head crashes or unreadable media, **resulting in the loss of parts of secondary storage;**
- **Application software errors,** such as **logical** errors in the program that is accessing the database, that cause one or more transactions to fail;
- **Natural physical disasters,** such as fires, floods, earthquakes, or power failures;
- **Carelessness** or unintentional destruction of data or facilities by operators or users;
- **Sabotage,** or intentional **corruption** or destruction of data, hardware, or software facilities.

In databases, usually a failure can generally be categorized as one of the following major groups:

- **Transaction failure:**

- **System failure:**
- **Media failure:**

- **Transaction failure.** a transaction cannot continue with its execution; therefore, it is aborted and if desired it may be restarted at some other time. Reasons: Deadlock, timeout, protection violation, or system error. There are two types of errors that may cause a transaction to fail:



- **Logical error.** The transaction can no longer continue with its normal execution because of some **internal condition**, such as bad input, data not found, overflow, or resource limit exceeded.
 - **System error.** The system has entered an undesirable state (for example, deadlock), as a result of which a transaction cannot continue with its normal execution. The transaction, however, can be re-executed at a later time.
- **System crash.** There is a **hardware malfunction**, or a bug in the database software or the operating system, that causes the loss of the content of volatile storage, and brings transaction processing to a halt. The content of nonvolatile storage remains intact, and is not corrupted. The assumption that hardware errors and bugs in the software bring the system to a halt, but do not corrupt the nonvolatile storage contents, is known as the **fail-stop assumption**. Well-designed systems have numerous internal checks, at the hardware and the software level, that brings the system to a halt when there is an error. Hence, the fail-stop assumption is a reasonable one. The database system is unable to process any transactions. Some of the common reasons of system failure are: register overflow, addressing error, power failure, memory failure, etc.
- **Disk failure.** A disk block loses its content as a result of either a head crash or failure during a data-transfer operation. Copies of the data on other disks, or archival backups on tertiary media, such as DVD or tapes, are used to recover from the failure. Failure of non-volatile storage media (mainly disk). Some of the common reasons are: head crash, dust on the recording surfaces, fire, etc.

Whatever the cause of the failure, there are two principal effects that we need to consider: the loss of main memory, including the database buffers, and the loss of the disk copy of the database. All of these failures can and do occur in database applications. Since a database is shared by many people and since it is a key element of the organization's operations, it is important to recover it as soon as possible. To make the database secured, one should formulate a "plan of attack" in advance. The plan will be used in case of database insecurity that may range from minor inconsistency to total loss of the data due to hazardous events.

The basic steps in performing a recovery are

1. **Isolating the database from other users.** Occasionally, you may need to drop and re-create the database to continue the recovery.
2. **Restoring the database from the most recent useable dump.**
3. **Applying transaction log dumps, in the correct sequence,** to the database to make the data as current as possible.

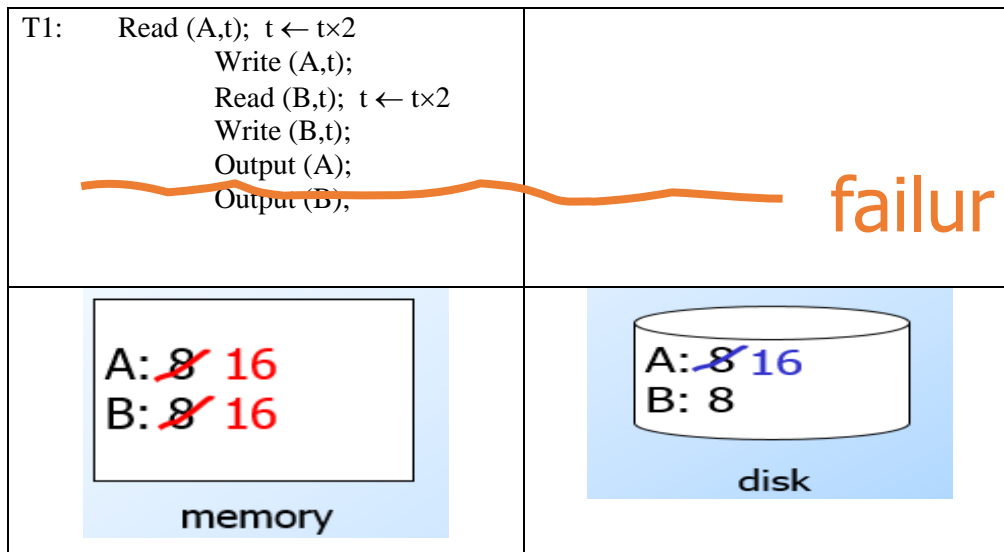
It is a good idea to test your backup and recovery plans periodically by loading the backups and transaction logs into a test database and verifying that your procedure really works.

One can recover databases after three basic types of **problems**:

- ☛ **User error,**
- ☛ **Software failure, and**
- ☛ **Hardware failure.**

Each type of failure requires a recovery mechanism. In a transaction recovery, the effect of failed transaction is removed from the database, if any. In a system failure, the effects of failed transactions have to be removed from the database and the effects of *completed* transactions have to be *installed* in the database. The database recovery manger is responsible to guarantee the **atomicity** and **durability** properties of the ACID property.

Example:

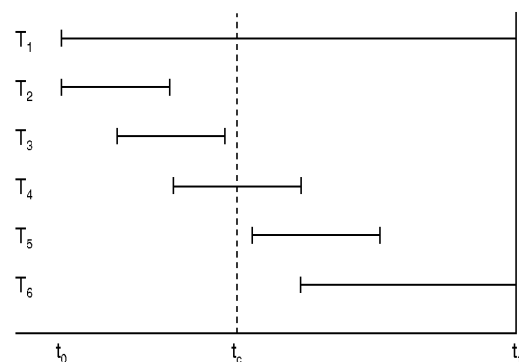


As you can see above on the picture the value of A and B is 16 on the memory. Whereas, the value of A is 16 and before the value of B which should be 16 is 8. The failure occurs before the value of B=16 is written to disk. There should be a recovery mechanism to return either both values to 16 or both values to 8.

Transaction Log

Execution history of concurrent transactions.

- DBMS starts at time t_0 , but fails at time t_f . Assume data for transactions T2 and T3 have been written to secondary storage.
- T1 and T6 have to be undone. In absence of any other information, recovery manager has to redo T2, T3, T4, and T5.
- t_c is the checkpoint time by the DBMS



A logging and recovery manager, responsible for the durability of transactions.

Log Records

The most widely used structure for **recording database modifications** is the **log**.

The log is a sequence of **log records**, recording all the update activities in the database.

There are several types of **log records**.

An **update log record** describes a single **database write**. It has these fields:

- **Transaction identifier**, which is the unique identifier of the transaction that performed the write operation.
- **Data-item identifier**, which is the unique identifier of the data item written. Typically, it is the location on disk of the data item, consisting of the block identifier of the block on which the data item resides, and an offset within the block.
- **Old value**, which is the value of the data item prior to the write.
- **New value**, which is the value that the data item will have after the write.

Transaction creates a log record prior to modifying the database. The log records allow the system to undo changes made by a transaction in the event that the transaction must be aborted; they allow the system also to redo changes made by a transaction if the transaction has committed but the system crashed before those changes could be stored in the database on disk.

We represent an update log record as $\langle T_i, X_j, V_1, V_2 \rangle$, indicating that transaction T_i has performed a write on data item X_j . X_j had value V_1 before the write, and has value V_2 after the write. Other special log records exist to record significant events during transaction processing, such as the start of a transaction and the commit or abort of a transaction. Among the types of log records are:

- $\langle T_i \text{ start} \rangle$. Transaction T_i has started.
- $\langle T_i \text{ commit} \rangle$. Transaction T_i has committed.
- $\langle T_i \text{ abort} \rangle$. Transaction T_i has aborted.

Recovery Techniques

A computer system, like any other device, is subject to failure from a variety of causes: disk crash, power outage, software error, a fire in the machine room, even sabotage. A failure is a state where data inconsistency is visible to transactions if they are scheduled for execution. In

any failure, information may be lost. Therefore, the database system must take actions in advance to ensure that the atomicity and durability properties of transactions are preserved.

Database recovery is the process of restoring database to a correct state in the event of a failure. A **database recovery** is the process of eliminating the effects of a failure from the database. **Recovery**, in database systems terminology, is called **restoring the last consistent state of the data items**.

Recovery Facilities

- DBMS should provide following facilities to assist with recovery:
 - **Backup mechanism:** that **makes** periodic backup copies of database.
 - **Logging facility:** that **keeps** track of current state of transactions and database changes.
 - **Checkpoint facility:** that enables updates to database in progress to be made permanent.
 - **Recovery manger:** This allows DBMS to restore the database to a consistent state following a failure.

Damage to the database could be either physical and relate which will result in the loss of the data stored or just inconsistency of the database state after the failure. For each we can have a recover mechanism:

1. If database has been damaged:
 - Need to restore last backup copy of database and reapply updates of committed transactions using log file.
 - Extensive damage/catastrophic failure: physical media failure; is restored by using the backup copy and by re executing the committed transactions from the log up to the time of failure.
2. If database is only inconsistent:
 - No physical damage/only inconsistent: the restoring is done by reversing the changes made on the database by consulting the transaction log.
 - Need to undo changes that caused inconsistency. May also need to redo some transactions to ensure updates reach secondary storage.

- Do not need backup, but can restore database using before- and after-images in the log file.

Recovery Techniques for Inconsistent Database State

Recovery is required if only the database is updated. The kind of recovery also depends on the kind of update made on the database.

Restoring the database means **transforming** the state of the database to the immediate good state before the failure. To do this, the change made on the database should be preserved. Such kind of information is stored in a *system log* or *transaction log* file.

The database buffers occupy an area in main memory from which data is transferred to and from secondary storage. Only once the buffers have been **flushed** to secondary storage can any update operations be regarded as permanent. This flushing of the buffers to the database can be triggered by a specific command (for example, transaction commit) or automatically when the buffers become full. The explicit writing of the buffers to secondary storage is known as **force-writing**.

If a failure occurs between writing to the buffers and flushing the buffers to secondary storage, the recovery manager must determine the status of the transaction that performed the write at the time of failure. If the transaction had issued its commit, then to ensure durability the recovery manager would have to **redo** that transaction's updates to the database (also known as **roll forward**).

On the other hand, if the transaction had not committed at the time of failure, then the recovery manager would have to **undo (rollback)** any effects of that transaction on the database to guarantee transaction atomicity.

If only one transaction has to be undone, this is referred to as **partial undo**. A partial undo can be triggered by the scheduler when a transaction is rolled back and restarted as a result of the concurrency control protocol, as described in the previous section. A transaction can also be aborted unilaterally, for example, by the user or by an exception condition in the application program. When all active transactions have to be undone, this is referred to as **global undo**.

Three main recovery techniques:

1. Deferred Update
2. Immediate Update
3. Shadow Paging:

The **Deferred Update** and **Immediate Update** are log based approaches of database recovery, due to this they serve **on non-catastrophic failure**.

- **Recovery from catastrophic**
 - If there is extensive damage to the wide portion of the database
 - This method restore a past copy of the database from the backup storage and reconstructs operation of a committed transaction from the back up log up to the time of failure
- **Recovery from non-catastrophic failure**
 - When the database is not physically damaged but has become inconsistent
 - The strategy uses *undoing and redoing* some operations in order to restore to a consistent state :

The **shadow paging** is an alternative approach to the deferred update. The modified version of a data item does not overwrite its disk copy but is written at a separate disk location.

Deferred Update

Updates are not written to the database until after a transaction has reached its commit point. If **transaction fails before commit**, it will not have modified database and **so no undoing** of changes required. May be necessary to redo updates of committed transactions as their effect may not have reached database.

A transaction first modifies all its data items and then writes all its updates to the final copy of the database. No change is going to be recorded on the database before commit. The changes will be made only on the local transaction workplace. **Update on the actual database is made after commit and after the change is recorded on the log**. Since there is no need to perform undo operation it is also called NO-UNDO/REDO Algorithm

Example transactions T_0 and T_1 (T_0 executes before T_1):

T_0 : read (A) $A := A - 50$ Write (A) read (B) $B := B + 50$ write (B)	T_1 : read (C) $C := C - 100$ write (C)
--	---

On these three scenarios initial value of A=1000, B=2000, and C=700

< T_0 start>	< T_0 start>	< T_0 start>
< T_0 , A, 950>	< T_0 , A, 950>	< T_0 , A, 950>
< T_0 , B, 2050>	< T_0 , B, 2050>	< T_0 , B, 2050>
	< T_0 commit>	< T_0 commit>
	< T_1 start>	< T_1 start>
	< T_1 , C, 600>	< T_1 , C, 600>
		< T_1 commit>
(a)	(b)	(c)

Based on the above log, if log on stable storage at time of crash is as in case:

- (a) No redo actions need to be taken
- (b) redo(T_0) must be performed since < T_0 commit> is present
- (c) **redo**(T_0) must be performed followed by redo(T_1) since
 < T_0 commit> and < T_1 commit> are present

Immediate Update/ Update-In-Place

Using the *immediate update* recovery protocol, updates are applied to the **databases they occur without waiting to reach the commit point.** As well as having to redo the updates of committed transactions following a failure, it may now be necessary to undo the effects of transactions that had not committed at the time of failure.

As soon as a transaction updates a data item, it updates the final copy of the database on the database disk. During making the update, the change will be recorded on the transaction log to permit rollback operation in case of failure. UNDO and REDO are required to make the transaction consistent. Thus, it is called **UNDO/REDO Algorithm**. This algorithm will undo all updates made in place before commit. The redo is required because some operations which are completed but not committed should go to the database. If we don't have the second scenario, then the other variation of this algorithm is called **UNDO/NO-REDO Algorithm**.

Shadow Paging

This scheme maintains two page tables during life of a transaction: current page and shadow page table. When transaction starts, two pages are the same. Shadow page table is never changed thereafter and is used to restore database in event of failure. During transaction, current page table records all updates to database. When transaction completes, current page table becomes shadow page table.

Shadow paging has several advantages over the log-based schemes: the overhead of maintaining the log file is eliminated and recoveries is significantly faster as there is no need for undo or redo operations. However, it has disadvantages as well, such as data fragmentation and the need for periodic garbage collection to reclaim inaccessible blocks.

The three styles of logging for database recovery are:

- **Undo**: Restore all BFIMs on to disk (Remove all AFIMs)
- **Redo**: Restore all AFIMs on to disk
- **Undo/Redo** restore both BFIM and AFIM

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

Summary:

Material associated with one transaction must be written to disk in the following order:

- U1 : Log records indicating changed DB elements

- U2 : Changed DB elements themselves
- U3 : COMMIT
log record

Example : Actions & log entries

Recovery using Undo Logging

Simple form of recovery assumes that, any time failure may occur. This failure may have brought certain changes written to disk, where as some changes that should occur may not be written to disk. T recover using undo the following are series of steps performed:

Step	Action	t	M-A	M-B	D-A	D-B	Log
1							<Start T>
2	READ(A, t)	8	8		8	8	
3	t:=t*2	16	8		8	8	
4	WRITE(A, t)	16	16		8	8	<T, A, 8>
5	READ(B, t)	8	16	8	8	8	
6	t:=t*2	16	16	8	8	8	
7	WRITE(B, t)	16	16	16	8	8	<T, B, 8>
8	FLUSH LOG						
9	OUTPUT(A)	16	16	16	16	8	
10	OUTPUT(B)	16	16	16	16	16	
11							<Commit T>
12	FLUSH LOG						

Flush Log : Log manager needs a FL command to tell the buffer manager to copy to disk any log blocks that have not previously copied to disk

1. Look at the entire log and Divide the transaction into committed and uncommitted (active) transactions
2. Scan the log end by recovery manager, if it sees <T, X, V>

If <Commit T> is found: Transaction could not have left DB in an inconsistent state

If <Commit T> is not found: Some changes may be in disk and other changes in MM, not copied to disk. This means T is incomplete or aborted transaction, the recovery manger must Use <T, X, v> to undo the changes, the value of X to be V.

3. After this write <ABORT T> for each incomplete transaction
4. Then flush log the transaction.

On the above example, what will happen if a crash occurs;

1. Immediately After step 12: we know the <COMMIT> record got to the disk before the crash. No need of undo, all records by the record manger are ignored

2. Immediately After step 11: it is seen that the <COMMIT> got flushed to the disk, so no need of undo. If commit is not flushed the action to be taken will be similar to the one mentioned on number 3 next.
3. At step 11: However if the <COMMIT> record never reached to disk the recovery manger considers T is incomplete it therefore stores 8 as the value of B, and continue on scanning, it will make the value of A to 8 again and <ABORT T> is written to the log, and the log is flushed.
4. Immediately After step 10: now the <COMMIT> record surely not written, so T is incomplete and it is undone like step three above..
5. Immediately After step 7: now it is not certain that whether any of the records reached to disk, therefore the recovery manger will undo the changes.
6. Immediately After step 4: no record is known about database element B. there is no <COMMIT> record observed. Due to this the recovery manger will undo the change to A and <ABORT T>. Note that no new value, 16, is mentioned on undo log.

Check pointing

Recovery require entire log to be examined due to this Recovery is very, very SLOW! The simplest way to untangle potential problems is to checkpoint the log periodically. If a transaction has its commit log recorded on the disk, log record of that transaction are no longer needed. The solution is a simple check pointing.

Check pointing

Checkpoint: is a point of synchronization between database and a transaction log file. All buffers are force-written to secondary storage. Checkpoint record is created containing identifiers of all active transactions. When failure occurs, redo all transactions that committed since the checkpoint and undo all transactions active at time of crash.

Nmk96 Advanced Database

Chapter 4: Query processing and Optimization

Query Processing

Query Processor: Group of components of a DBMS that turns user queries and data modification commands into a sequence of database operations and executes those operations. Query Processing Refers to the range of activities involved (parsing, validating, optimizing, and executing a query) in extracting data from a database. The aim of query processing is to find information in one or more databases and deliver it to the user quickly and efficiently. Traditional techniques work well for databases with standard, single-site relational structures, but databases containing more complex and diverse types of data demand new **query processing and optimization techniques**.

The aims of query processing are to **transform** a query written in a **high-level language**, typically SQL, into a correct and efficient execution strategy expressed **in a low-level language** (implementing the relational algebra), and to execute the strategy to retrieve the required data. A Query Processor is responsible to produce an execution plan that will guarantee an acceptable response time. The activities involved in.

Query Processing can be divided into the following main phases:

1. Query Decomposition
2. Optimization
3. Code generation and
4. Execution

Query Decomposition

- **Query decomposition** is the process of transforming a **high-level query into a relational algebra query**, and to check that the query is syntactically and semantically correct.
- **Query decomposition** consists of parsing and validation

Typical **stages** in query decomposition are:

1. **Analysis:** lexical and syntactical analysis of the query (correctness). Query tree will be built for the query containing leaf node for base relations, one or many non-leaf nodes for relations produced by relational algebra operations and root node for the result of the query. Sequence of operation is from the leaves to the root.
2. **Normalization:** convert the query into a normalized form. The predicate **WHERE** will be converted to **Conjunctive (\vee) or Disjunctive (\wedge)** Normal form.
3. **Semantic Analysis:** to **reject normalized queries** that is not correctly formulated or contradictory. Incorrect if components do not contribute to generate result. Contradictory if the predicate cannot be satisfied by any tuple.
4. **Simplification:** to **detect redundant qualifications**, eliminate common sub-expressions, and transform the query to a semantically equivalent but more easily and effectively computed form.
5. **Query Restructuring** More than one translation is possible Use transformation rules

Query Optimization

What is wrong with the ordinary query?

- ❖ Everyone wants the performance of their database to be optimal. In particular, there is often a requirement for a specific query or object that is query based, to run faster.
- ❖ Problem of query optimization is to find the sequence of steps that produces the answer to user request in the most efficient manner, given the database structure.
- ❖ The performance of a query is affected by the tables or queries that underlies the query and by the/ Choosing a query execution plan is called Query Optimization and it mainly means making decisions about data access methods. **Query Optimization strongly relies on File Organization techniques**

Query optimizers are one of the main means by which modern database systems achieve their performance advantages. Given a request for data manipulation or retrieval, **an optimizer will choose an optimal plan for evaluating the request from among the manifold alternative strategies.** i.e. there are many ways (access paths) for accessing desired file/record. **The optimizer tries to select the most efficient (cheapest) access path for accessing the data.** DBMS is responsible to pick the best execution strategy based various considerations.

Query optimizers were already among the largest and most complex modules of database systems. **The objective in query optimization is to select an efficient execution strategy.** As there are many equivalent

transformations from same high-level query, aim is to choose the one that **minimizes** resource usage. **Generally**, an efficient execution plan **reduces the total execution time** of a query; thereby **reducing the response time of a query**. The problem is computationally intractable with large number of relations, so the strategy adopted is reduced to finding a near optimum solution.

Most efficient processing: Least amount of I/O and CPU resources.

Selection of the best method:

In a **non-procedural language**, the system does the optimization at the time of execution. On the other hand, in a **procedural language**, programmers have some **flexibility** in selecting the best method.

For optimizing the execution of a query the programmer must know:

- File organization
- Record access mechanism and primary or secondary key.
- Data location on disk.
- Data access limitations.

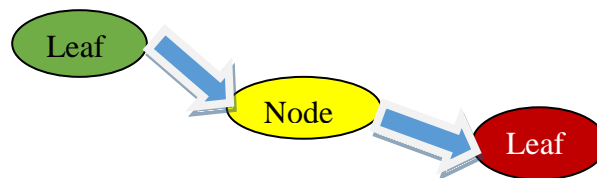
Approaches to Query Optimization

There are two approaches to Query Optimization,

1. **heuristic approach** which improves and refine relational Algebra tree to create equivalent logical query plan. It orders operations in a query.
 2. **cost based estimation** which use database statistics to estimate physical costs of logical operators. The second uses comparing different strategies based on relative **cost**, and selecting one that minimizes resource usage
- Heuristic (Logical Transformations)
 - Use Transformation Rules to convert one relational algebra expression into an equivalent form that is known to **be more efficient**
 - Heuristic Optimization Guidelines
 - Cost Based (Physical Execution Costs)
 - Data Storage/Access Refresher
 - Catalog & Costs

I. *Heuristics Approach*

- The **heuristic approach** uses the knowledge of the characteristics of the relational algebra operations and the relationship between the operators to optimize the query.
- Thus, the heuristic approach of optimization will make use of:
 - **Properties of individual operators**
 - **Association between operators**
 - **Query Tree**: a graphical representation of the operators, relations, attributes and predicates and processing sequence during query processing.
 - Query tree is composed of three main parts:
 - i. **The Leafs**: the base relations used for processing the query/ extracting the required information.
 - ii. **The Root**: the final result/relation as an output based on the operation on the relations used for query processing
 - iii. **Nodes**: intermediate results or relations before reaching the final result.
 - Sequence of execution of operation in a query tree will start from the leaves and continues to the intermediate nodes and ends at the root.



The properties of each operations and the association between operators is analyzed using set of rules called **TRANSFORMATION RULES**. Use of the transformation rules will transform the query to relatively good execution strategy. In using heuristics during query optimization, the following steps must be followed:

- Translate queries into query trees or query graphs
- Apply transformation rules for relational algebra expression.
- Perform heuristic optimization

Example

Example Join Query over R(A,B,C) and S(C,D,E): **Select** B, D**From** R, S**Where** R.A = "c" AND S.E = 2 AND R.C = S.C

We need transformation rule to address such query

Transformation Rules for Relational Algebra

1. **Cascade of SELECTION:** conjunctive SELECTION Operations can cascade into individual Selection Operations and Vice Versa

$$\sigma_{(c1 \wedge c2 \wedge c3)}(R) = \sigma_{c1}(\sigma_{c2}(\sigma_{c3}(R))) \text{ where } c_i \text{ is a predicate}$$

2. **Commutativity of SELECTION operations**

$$\sigma_{c1}(\sigma_{c2}(R)) = \sigma_{c2}(\sigma_{c1}(R)) \text{ where } c_i \text{ is a predicate}$$

3. **Cascade of PROJECTION:** in the sequence of PROJECTION Operations, only the last in the sequence is required

$$\pi_{L1}\pi_{L2}\pi_{L3}\pi_{L4}(R) = \pi_{L1}(R)$$

4. Commutativity of SELECTION with PROJECTION and Vice Versa
 - a. If the predicate c_i involves only the attributes in the projection list (L_i), then the selection and projection operations commute

$$\pi_{L1}(\sigma_{c1}(R)) = \sigma_{c1}(\pi_{L1}(R))$$

5. Commutativity of THETA JOIN/Cartesian Product

R X S is equivalent to S X R

Also holds for Equi-Join and Natural-Join

$$(R \bowtie_{c1} S) = (S \bowtie_{c1} R)$$

6. Commutativity of SELECTION with THETA JOIN
 - a. If the predicate c_i involves only attributes of one of the relations (R) being joined, then the Selection and Join operations commute

$$\sigma_{c1}(R \bowtie_c S) = (\sigma_{c1}(R)) \bowtie_c S$$

- b. If the predicate is in the form $c1 \wedge c2$ and $c1$ involves only attributes of R and $c2$ involves only attributes of S, then the Selection and Theta Join operations commute

$$\sigma_{c1 \wedge c2}(R \bowtie_c S) = (\sigma_{c1}(R)) \bowtie_c (\sigma_{c2}(S))$$

7. Commutativity of PROJECTION and THETA JOIN

δIf the projection list is of the form L_1, L_2 , where L_1 involves only attributes of R and L_2 involves only attributes of S being joined and the predicate c involves only attributes in the projection list, then the SELECTION and JOIN operations commute

$$\pi_{L_1, L_2} (R \bowtie_c S) = (\pi_{L_1, L_2}(R)) \bowtie_c (\pi_{L_1, L_2} (S))$$

8. Commutativity of the Set Operations: UNION and INTERSECTION but not SET DIFFERENCE

$$R \cap S = S \cap R \text{ and } R \cup S = S \cup R$$

9. Associativity of the THETA JOIN, CARTESIAN PRODUCT, UNION and INTERSECTION.

$$(R \theta S) \theta T = R \theta (S \theta T) \text{ where } \theta \text{ is one of the operations}$$

10. Commuting SELECTION with SET OPERATIONS

$$\sigma_c (R \theta S) = (\sigma_c(R) \theta \sigma_c(S)) \text{ where } \theta \text{ is one of the operations}$$

11. Commuting PROJECTION with UNION

$$\pi_{L_1} (S \cup R) = \pi_{L_1} (S) \cup \pi_{L_1} (R)$$

Heuristic Approach will be implemented by using the above transformation rules in the following sequence or steps.

Sequence for Applying Transformation Rules

1. Use
Rule-1 → Cascade SELECTION
2. Use
Rule-2: Commutativity of SELECTION
Rule-4: Commuting SELECTION with PROJECTION
Rule-6: Commuting SELECTION with JOIN and CARTESIAN
Rule-10: commuting SELECTION with SET OPERATIONS
3. Use
Rule-9: Associativity of Binary Operations (JOIN, CARTESIAN, UNION and INTERSECTION). Rearrange nodes by making the most restrictive operations to be performed first (moving it as far down the tree as possible)
4. Perform Cartesian Operations with the subsequent Selection Operation
5. Use
Rule-3: Cascade of PROJECTION
Rule-4: Commuting PROJECTION with SELECTION
Rule-7: Commuting PROJECTION with JOIN and CARTESIAN
Rule-11: commuting PROJECTION with UNION

Query compilation is divided into three steps: -

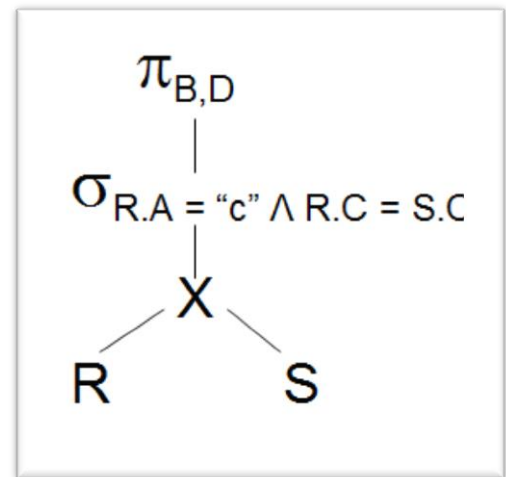
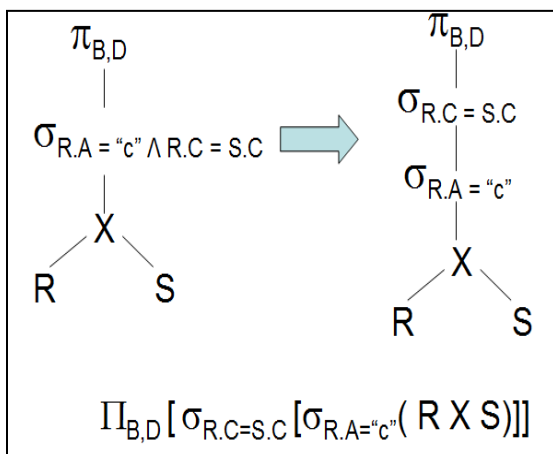
1. **Parsing:** Parse SQL query into parser tree. Parse tree, representing the query and its structure is constructed
2. **Logical query plan:** Transforms parse tree into expression tree of relational algebra. Parse tree – converted to an initial query plan – an algebraic representation of the query
3. **Physical query plan:** Transforms logical query plan into physical query plan. Abstract/initial/logical plan – turned into physical plan by selecting algorithms to implement each of the operations of the logical plan which includes:
 - ✓ Operation performed
 - ✓ Order of operation
 - ✓ Algorithm used
 - ✓ The way in which stored data is obtained and passed from one operation to another

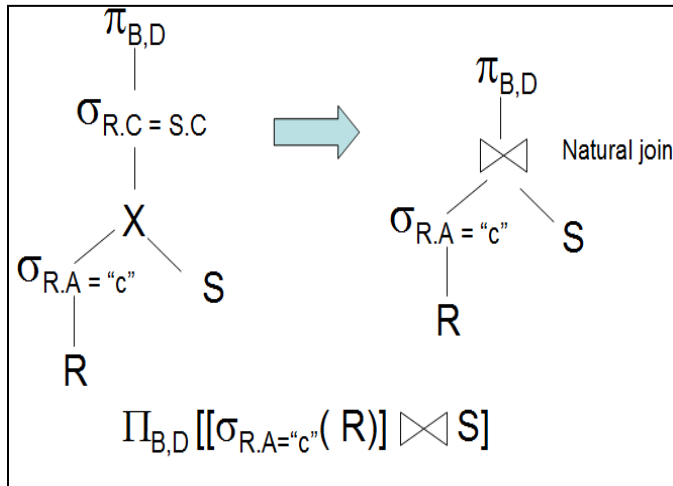
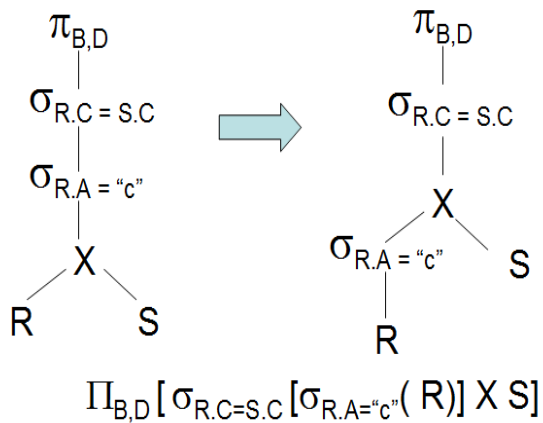
EXAMPLE:

SELECT B,D FROM R,S WHERE R.A = "C" \wedge R.C=S.C

Relational Algebra:

$\Pi_{B,D} [\sigma_{R.A="c" \wedge R.C = S.C} (R \times S)]$





Main Heuristic

The main heuristic is to first apply operations that **reduce the size** (the cardinality and/or the degree) of the **intermediate relation**. That is:

- Perform SELECTION as early as possible: that will reduce the **cardinality** (number of **tuples**) of the relation.
- Perform PROJECTION as early as possible: that will reduce the **degree** (number of **attributes**) of the relation.
 - Both a and b will be accomplished by placing the SELECT and PROJECT operations as far down the tree as possible.

- c. SELECT and JOIN operations with most restrictive conditions resulting with smallest absolute size should be executed before other similar operations. This is achieved by reordering the nodes with JOIN

Example: consider the following schemas and the query, where the EMPLOYEE and the PROJECT relations are related by the WORKS_ON relation.

EMPLOYEE (EEmpID, FName, LName, Salary, Dept, Sex, DoB)

PROJECT (PProjID, PName, PLocation, PFund, PManagerID)

WORKS_ON (WEmpID, WProjID)

WEmpID (refers to employee identification) and PProjID (refers to project identification) are foreign keys to WORKS_ON relation from EMPLOYEE and PROJECT relations respectively.

Query: The manager of the company working on road construction would like to view employees name born before January 1 1965 who are working on the project named Ring Road.

Relational Algebra representation of the query will be:

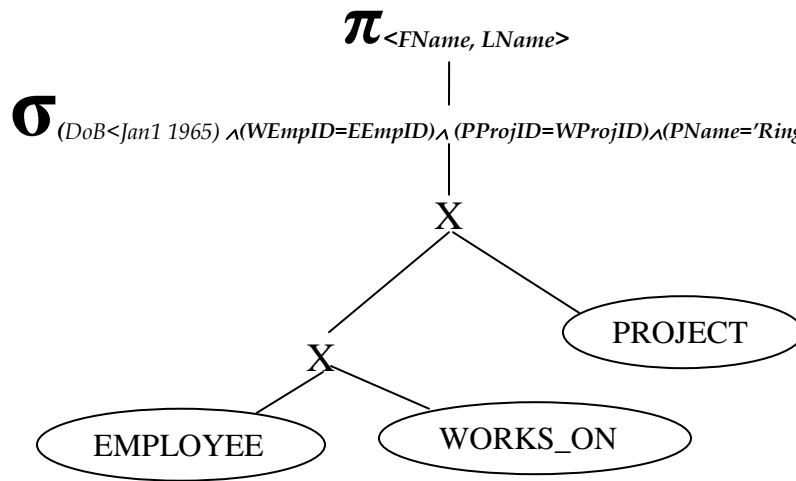
$\pi_{\langle FName, LName \rangle} (\sigma_{\langle DoB < Jan\ 1\ 1965 \wedge WEmpID = EEmpID \wedge PProjID = WProjID \wedge PName = 'Ring\ Road' \rangle} (EMPLOYEE \times WORKS_ON \times PROJECT))$

The SQL equivalence for the above query will be:

SELECT FName, LName **FROM** EMPLOYEE, WORKS_ON, PROJECT **WHERE** $DoB < Jan\ 1\ 1965$

$\wedge EEmpID = WEmpID \wedge WProjID = PProjID \wedge PName = 'Ring\ Road'$

The initial query tree will be:



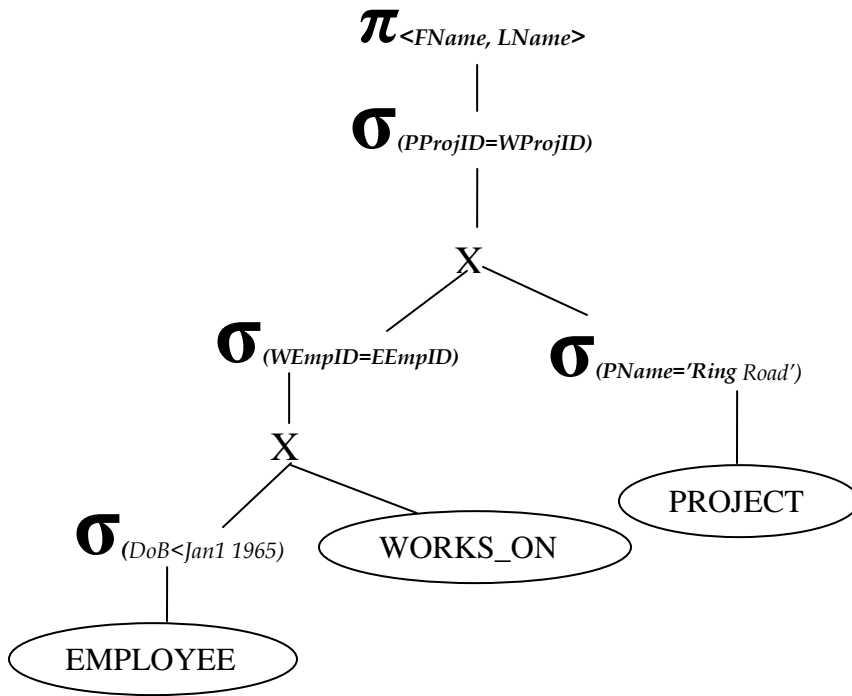
By applying the first step (cascading the selection) we will come up with the following structure.

$\sigma_{(DoB < Jan1\ 1965)} (\sigma_{(WEmpID = EEmpID)} (\sigma_{(PProjID = WProjID)} (\sigma_{(PName = 'Ring\ Road')} (EMPLOYEE \ X \ WORKS_ON \ X \ PROJECT))))$

By applying the second step it can be seen that some conditions have attribute that belong to a single relation (DoB belongs to EMPLOYEE and PName belongs to PROJECT) thus the selection operation can be commuted with Cartesian Operation. Then, since the condition $WEmpID = EEmpID$ base the employee and WORKS_ON relation the selection with this condition can be cascaded.

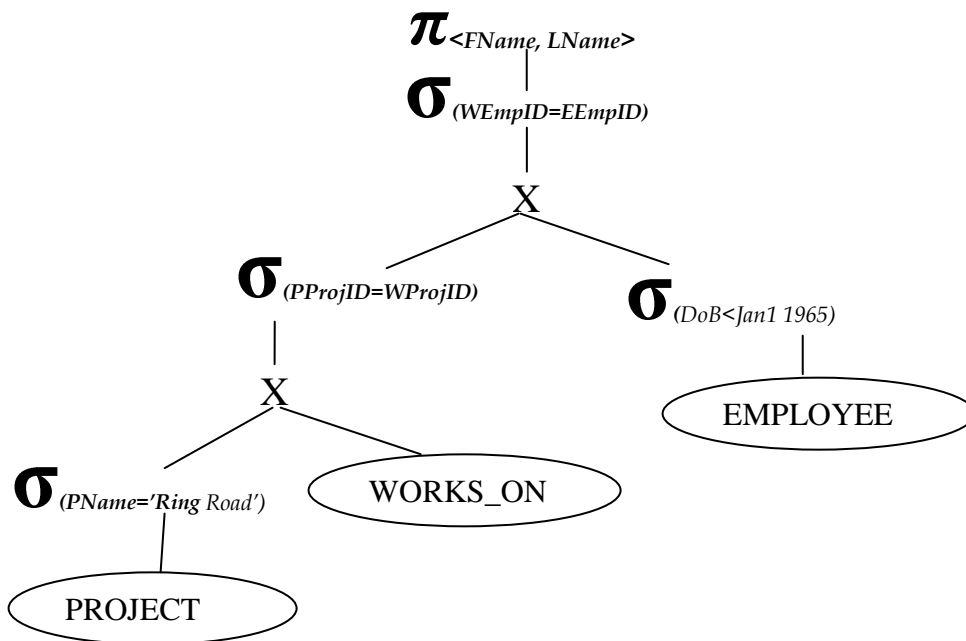
$(\sigma_{(PProjID = WProjID)} (\sigma_{(PName = 'Ring\ Road')} PROJECT)) \ X \ (\sigma_{(WEmpID = EEmpID)} (WORKS_ON \ X \ (\sigma_{(DoB < Jan1\ 1965)} EMPLOYEE)))$

The query tree after this modification will be:

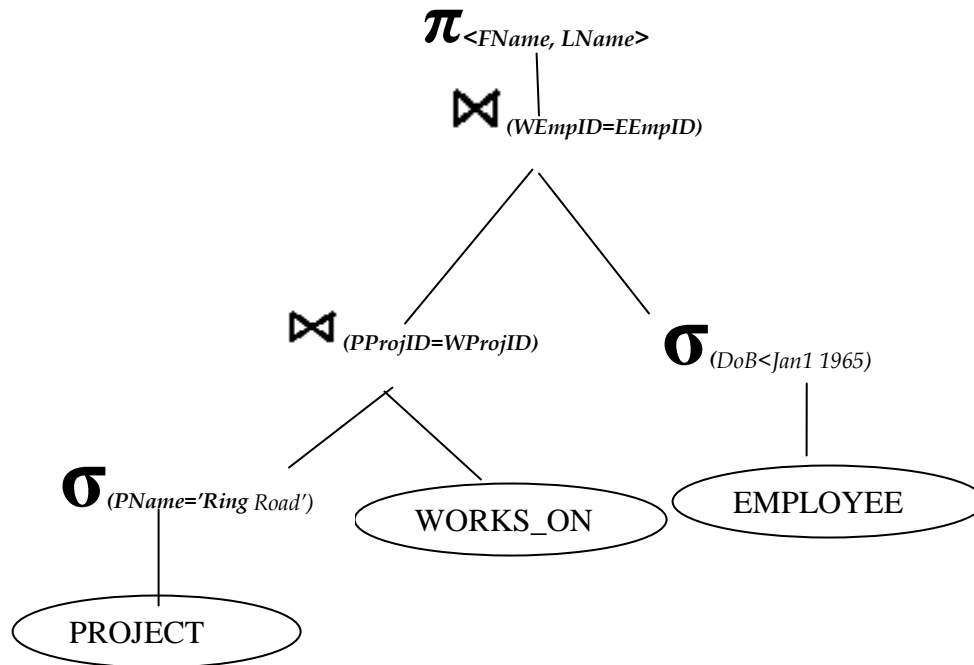


Using the third step, perform most restrictive operations first.

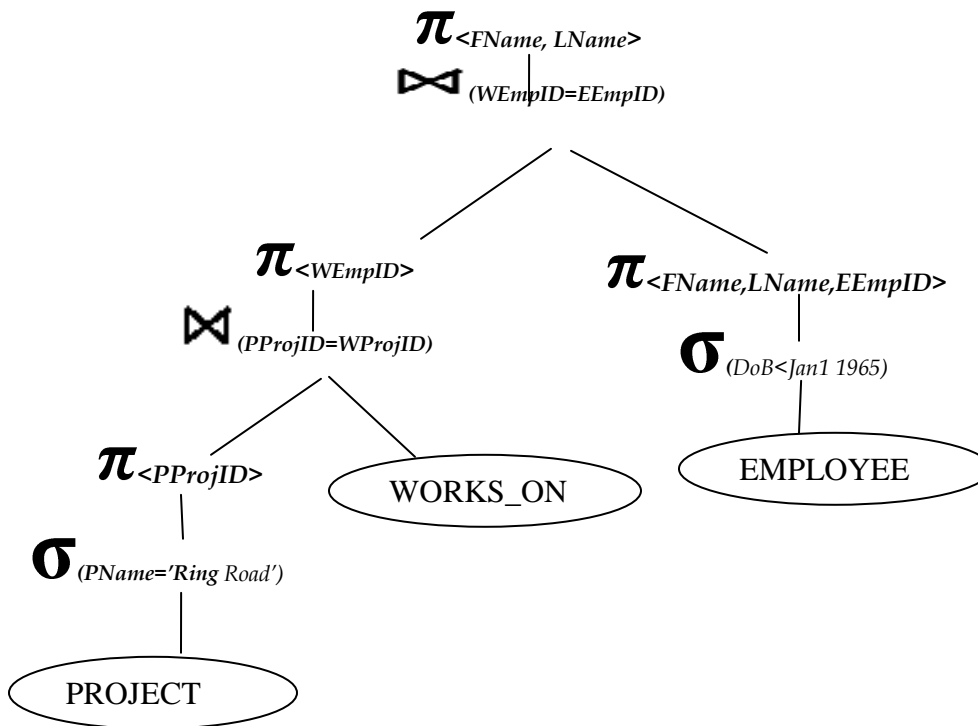
From the query given we can see that selection on **PROJECT** is most restrictive than selection on **EMPLOYEE**. Thus, it is better to perform selection on **PROJECT** BEFORE on **EMPLOYEE**. rearrange the nodes to achieve this.



Using the forth step, Perform Cartesian Operations with the subsequent Selection Operation.



Using the fifth step, Perform the projection as early as possible.



For every project located in ‘Stafford’, list the project number, the controlling department number and the department manager’s last name, address, and birth date.

II. Cost Estimation Approach to Query Optimization

The main idea is to minimize the cost of processing a query. The cost function is comprised of:

- I/O cost + CPU processing cost + communication cost + Storage cost

These components might have different weights in different processing environments

The DBMs will use information stored in the system catalogue for the purpose of estimating cost. The main target of query optimization is to minimize the size of the intermediate relation.

The size will have effect in the cost of:

- Disk Access
- Data Transpiration
- Storage space in the Primary Memory
- Writing on Disk

The statistics in the system catalogue used for cost estimation purpose are:

- Cardinality of a relation: the number of tuples contained in a relation currently (r)
- Degree of a relation: number of attributes of a relation
- Number of tuples on a relation that can be stored in one block of memory
- Total number of blocks used by a relation
- Number of distinct values of an attribute (d)
- Selection Cardinality of an attribute (S): that is average number of records that will satisfy an equality condition $S=r/d$

By using the above information one could calculate the cost of executing a query and selecting the best strategy, which is with the minimum cost of processing.

Cost Components for Query Optimization

The costs of query execution can be calculated for the following major process we have during processing.

1. *Access Cost of Secondary Storage*

Data is going to be accessed from secondary storage, as a query will be needing some part of the data stored in the database. The disk access cost can again be analyzed in terms of:

- Searching
- Reading, and
- Writing, data blocks used to store some portion of a relation.

The disk access cost will vary depending on the

- ✓ File organization used and the access method implemented for the file organization.
- ✓ The data allocation scheme, whether the data is stored contiguously or in scattered manner, will affect the disk access cost.

2. **Storage Cost** While processing a query, as any query would be composed of many database operations, there could be one or more intermediate results before reaching the final output. These intermediate results should be stored in primary memory for further processing. The bigger the intermediate relation, the larger the memory requirement, which will have impact on the limited available space. This will be considered as a cost of storage.

3. **Computation Cost** Query is composed of many operations. The operations could be database operations like reading and writing to a disk, or mathematical and other operations like:

- Searching
- Sorting
- Merging
- Computation on field values

4. **Communication Cost** In most database systems the database resides in on station and various queries originate from different terminals. This will have impact on the performance of the system adding cost for query processing. Thus, the cost of transporting data between the database site and the terminal from where the query originate should be analyzed.

Chapter 6: Database Security and Authorization

Introduction to DB Security Issues

Database Security

Database security is the mechanisms that **protect** the database against intentional(hone bilo) or accidental threats. Database security encompasses hardware, software, people, and data.

It is Protection of information contained in the database against **unauthorized access**, modification or destruction.

Database security aims to minimize losses caused by anticipated events in a cost-effective manner without unduly constraining the users.

In recent times, computer based criminal activities have significantly increased and are forecast to continue to rise over the next few years.

A good database security management system has not only the following characteristics:

- ✓ data independence,
 - ✓ shared access,
 - ✓ minimal redundancy,
 - ✓ data consistency, and data integrity but,
also
 - ✓ privacy, integrity, and availability.
-
- **Privacy** signifies that an **unauthorized user cannot disclose (open) data**. Ethical and legal rights that individuals have with regard to control over the dissemination and user of their personal information
 - **Integrity** ensures that an **unauthorized user cannot modify data**. Mechanism that is applied to ensure that the data in the database is correct and consistent
 - **Availability** ensures that data be made available to the authorized user unfailingly
 - **Copyright** ensures the **native rights of individuals** as a creator of information.
 - **Validity** ensures **activities to be accountable by law**.

When we talk about the **levels of security protection**, it may start from organization & administrative security, physical & personnel security, communication security and Information systems security. **Database security and integrity** is about protecting the database from **being inconsistent and being disrupted**(betebete). We can also call it **database misuse**.

Database misuse could be *Intentional* or *Accidental*, where accidental misuse is **easier** to cope(control) with than intentional misuse.

Accidental inconsistency occurs due to:

- System crash during transaction processing
- Anomalies due to concurrent access
- Anomalies due to redundancy
- Logical errors

Intentional misuse could be:

- Unauthorized reading of data
- Unauthorized modification of data or
- Unauthorized destruction of data

Most systems **implement** good **Database Integrity** to protect the system from **accidental misuse**, whereas, there are many computer based **measures** to protect the system from intentional misuse, which is termed as **Database Security** measures.

Threats to Databases

Threats to databases can result in the loss or degradation of some or all of the following commonly accepted security goals:

- Theft and Fraud
- Loss of Confidentiality
- Loss of Privacy
- Loss of Integrity
- Loss of Availability

These situations broadly represent areas in which the organization should seek to reduce risk, that is, the possibility of incurring loss or damage. In some situations, these areas are closely related such that an activity that leads to loss in one area may also lead to loss in another. In addition, events such as fraud or loss of privacy may arise because of either intentional or unintentional acts, and do not necessarily result in any detectable changes to the database or the computer system.

Theft and fraud: Theft and fraud affect not only the database environment but also the entire organization. As it is people who perpetrate such activities, attention should focus on reducing the opportunities for this occurring.

Theft and fraud do not necessarily alter data, as is the case for activities that **result in** either

- ✓ **loss** of confidentiality or
- ✓ loss of privacy.

Loss of integrity: Database integrity refers to the requirement that information be protected from improper **modification**. Modification of data includes creating, inserting, and updating data; changing the status of data; and deleting data.

Integrity is lost if **unauthorized** changes are made to the data by either **intentional or accidental** acts. If the loss of system or data integrity isn't corrected, continued use of the contaminated system or corrupted data could result in **inaccuracy, fraud, or erroneous decisions**.

Loss of data integrity results in

- **invalid or**
- **corrupted data**, which may seriously affect the operation of an organization.

Database Integrity constraints contribute to maintaining a secure database system by preventing data from becoming invalid and hence giving misleading or incorrect results.

- **Domain Integrity** means that each column in any table will have set of allowed values and cannot **assume** any value other than the one specified in the domain.
- **Entity Integrity** means that in each table the primary key (which may be composite) satisfies both of two conditions:
 - That the primary key is **unique** within the table and
 - That the primary key column(s) contains **no null values**.
- **Referential Integrity** means that in the database as a whole, things are set up in such a way **that if a column exists in two or more tables in the database** (typically as a primary key in one table and as a foreign key in one or more other tables), then any change to a

value in that column in any one table will be reflected in corresponding changes to that value where it occurs in other tables. This means that the RDBMS must be set up so as to take appropriate actions to spread a change—in one table—from that table to the other tables where the change must also occur.

The effect of the existence and maintenance of referential integrity is, in short, that if a **column exists in two or more tables in the database**, every occurrence of the column will contain only values that are **consistent** across the database.

- **Key constraints** in a relational database, there should be **some collection of attributes** with a special feature used to maintain the integrity of the database. These attributes will be named as Primary Key, Candidate Key, Foreign Key, and etc. These Key(s) should obey some rules set by the relational data model.
- **Enterprise Constraint** means some **business rules set by** the enterprise on how to use, manage and control the database

Loss of availability Database availability refers to making objects available to a human user or a program who/which has a legitimate(**authorized**) right to those data objects.

✚ *Loss of availability* occurs when the user or program cannot **access** these objects

Loss of confidentiality Database confidentiality refers to the protection of data from **unauthorized disclosure**. Whereas **privacy** refers to the need to protect data about **individuals**. The impact of unauthorized disclosure of confidential information can range from violation of the Data Privacy Act to the jeopardization of national security. Unauthorized, unanticipated, or unintentional disclosure could result in loss of public confidence, embarrassment, or legal action against the organization. Breaches of security resulting in loss of confidentiality could, for instance, lead to loss of competitiveness, and loss of privacy could lead to legal action being taken against the organization.

Threat may be any **situation or event, whether intentional or accidental**, that may adversely affect a system and consequently the organization. A threat may be caused by a situation or event

involving a person, action, or circumstance that is likely to bring harm to an organization. The harm to an organization may be *tangible* or *intangible*.

- **Tangible** – loss of hardware, software, or data
 - **Intangible** – loss of credibility(hakeгна) or client confidence
-

Countermeasures: Computer Based Controls

An organization deploying a database system needs to identify the types of threat it may be subjected to and initiate appropriate plans and *countermeasures*, bearing in mind the *costs* of implementing each.

Database Management Systems supporting multi-user database system must provide a database security and authorization subsystem to enforce limits on individual and group access rights and privileges.

Security Issues and general considerations

- **Legal, ethical** and **social** issues regarding the right to access information
- **Physical control** issues regarding how to keep the database physically secured.
- **Policy** issues regarding privacy of individual level at enterprise and national level
- **Operational** consideration on the techniques used (password, etc) to access and manipulate the database
- **System** level security including operating system and hardware control
- Security levels and security policies in enterprise level

The designer and the administrator of a database should first identify *the possible threat* that might be faced by the system in order *to take counter measures*.

Levels of Security Measures

Security measures can be implemented at several levels and for different components of the system. These levels are:

1. **Physical Level:** concerned with **securing the site containing the computer system**. The backup systems should also be physically protected from access except for authorized users. In other words, the site or sites containing the computer systems must be physically secured **against armed or sneaky(ashimur) entry by intruders.(talika geb)**
2. **Human Level:** concerned with **authorization of database users** for access the content at different levels and privileges.
3. **Operating System:** concerned with the weakness and strength of the operating system security on data files. Weakness may serve as a means of unauthorized access to the database. **No matter how secure the database system is**, weakness in operating system security may serve as a means of **unauthorized access to the database**. This also includes **@protection of data in primary and secondary memory from unauthorized access.**
4. **Database System:** concerned with **data access limit** enforced by the database system. Access limit like password, isolated transaction and etc. Some database system users may be authorized to access only a limited portion of the database. Other users may be allowed to issues queries, but may be forbidden to modify the data. It is the responsibility of the database system to ensure that these authorization restrictions are not violated.
5. **Application Level:** Since almost all database systems allow **remote** access through terminals or networks, software-level security with the network software is as important as physical

The following are computer-based security controls for a multi-user environment:

Authorization (give permission)

Granting a right privilege enables a subject to have legitimate access to a system object. Authorization controls can be built into the software, and govern not only what system or object a specified user can access, but also what the user may do with it.

Authorization controls are sometimes referred to as *access controls*. The process of authorization involves authentication of *subjects* (i.e. a user or program) requesting access to *objects* (i.e. a database table, view, procedure, trigger, or any other object that can be created within the system)

Authentication (check the reality)

All users of the database will have different access levels and permission for different data objects, and

authentication is the process of checking whether the user is the one with the privilege for the access level. Is the process of checking the users are who they say they are. Each user is given a unique identifier, which is used by the operating system to determine who they are. Thus, the system will check whether the user with a specific username and password is trying to use the resource. Associated with each identifier is a password, chosen by the user and known to the operation system, which must be supplied to enable the operating system to authenticate who the user claims to be.

Authorization/Privilege

Authorization refers to the process that determines the mode in which a particular (**previously authenticated**) client is allowed to access a specific resource controlled by a server.

Most of the time, authorization is implemented by using Views.

- Views are unnamed relations containing part of one or more base relations creating a customized/personalized view for different users.
- Views are used to hide data that a user needs not to see.

Forms of user authorization

There are different forms of user authorization on the resource of the database. These forms are privileges on what operations are allowed on a specific data object.

User authorization on the data/extension

1. **Read Authorization:** the user with this privilege is allowed only to read the content of the data object.
 2. **Insert Authorization:** the user with this privilege is allowed only to insert new records or items to the data object.
 3. **Update Authorization:** users with this privilege are allowed to modify content of attributes but are not authorized to delete the records.
 4. **Delete Authorization:** users with this privilege are only allowed to delete a record and not anything else.
- Different users, depending on the power of the user, can have one or the combination of the above forms of authorization on different data objects.

User authorization on the database schema

1. **Index Authorization:** deals with permission to create as well as delete an index table for relation.
2. **Resource Authorization:** deals with permission to add/create a new relation in the database.
3. **Alteration Authorization:** deals with permission to add as well as delete attribute.
4. **Drop Authorization:** deals with permission to delete and existing relation.

Access Controls

The typical way to provide access controls for a database system is based on the granting and revoking of privileges. A **privilege** allows a user to create or access (that is read, write, or modify) some database object (such as a relation, view, or index) or to run certain DBMS utilities.

Privileges are granted to users to accomplish the tasks required for their jobs. As excessive granting of unnecessary privileges can compromise security: a privilege should be granted to a user only if that user cannot accomplish his or her work without that privilege. A user who creates database object such as a relation or a view automatically gets all privileges on that

object. The DBMS subsequently keeps track of how these privileges are granted to other users, and possibly revoked, and ensures that at all times only users with necessary privileges can access an object.

Types of access control

- Discretionary and,
- Mandatory

Discretionary Access Control (DAC)

These are used to **grant** privileges to users, including the **capability** to access specific data files, records, or fields in a specified mode (such as read, insert, delete, or update). Grant different privileges to different users and user groups on various data objects. **The privilege is to access different data objects.** The mode of the privilege could be: Read, Insert, Delete, Update files, records or fields. Is more **flexible**. One user can have A but not B and another user can have B but not A

Mandatory Access Control (MAC)

These are used to enforce multilevel security by **classifying the data and users into various security classes (levels)** and then implementing the appropriate security policy of the organization. For example, a typical security policy is to permit users at a certain classification (or clearance) level to see only the data items classified at the user's own (or lower) classification level.

✚ **Classifying data and users into various security classes (or levels)** and implementing the appropriate security policy of the organization.

- Each **data object** will have certain **classification** level
- Each **user** is given certain **clearance level**
- Only users who can pass the clearance level can access the data object
- Is comparatively not-flexible/rigid
- If one user can have A but not B then B is accessed by users with higher privilege and we cannot have B but not A

The ability to classify user into a hierarchy of groups provide a powerful tool for administering large systems with thousands of users and objects.

A database system can support one or both of the security mechanisms to protect the data. In most systems, **it is better to filter those that are allowed** rather than identifying the not allowed. **Since if some object is authorized then it means it is not constrained.**

Views

A **view** is the dynamic result of one or more relational operations on the base relations to produce another relation. **A view is a virtual relation that does not actually exist in the database,** but is produced upon request by a particular user. The view mechanism provides a powerful and flexible security mechanism by **hiding** parts of the database from certain users. Using a view is more restrictive than simply having certain privileges granted to a user on the base relation(s)

Backup and recovery

Backup is the process of periodically taking a copy of the database and log file (and possibly programs) on to **offline** storage media. A DBMS should provide backup facilities to assist with the recovery of a database following failure. **Database recovery** is the process of **restoring** the database to a correct state in the event of a failure.

Journaling is the process of keeping and maintaining a log file (or journal) of all changes made to the database to enable recovery to be undertaken effectively in the event of a failure.

The advantage of journaling is that, in the event of a failure, the database can be recovered to its last known consistent state using a backup copy of the database and the information contained in the log file. If no journaling is enabled on a failed system, the only means of recovery is to restore the database using the latest backup version of the database. However, without a log file, any changes made after the last backup to the database will be lost

Integrity

Integrity constraints contribute to maintaining a secure database system by preventing data from becoming

- invalid and hence giving misleading or incorrect results

Encryption

Authorization may not be sufficient to protect data in database systems, especially when there is a situation where data should be moved from one location to the other using network facilities. **Encryption** is used to protect information stored at a particular site or transmitted between sites from being accessed by unauthorized users.

Encryption is the encoding of the data by a special algorithm that renders **the data unreadable** by any program without the decryption key. It is not possible for encrypted data to be read unless the reader knows how to decipher/decrypt the encrypted data.

Security at different Levels of Data

Almost all RDBMSs provide security at different levels and formats of data. This includes:

1. **Relation Level:** permission to have access to a specific relation.
2. **View Level:** permission to data included in the view and not in the named relations
3. **Hybrid (Relation/View):** the case where only part of a single relation is made available to users through View.

Any **database access request** will have the following three major components

1. **Requested Operation:** what kind of operation is requested by a specific query?
2. **Requested Object:** on which resource or data of the database is the operation sought to be applied?
3. **Requesting User:** who is the user requesting the operation on the specified object?

The database should be able to check for all the three components before processing any request. The checking is performed by the security subsystem of the DBMS.

Statistical Database Security

Statistical databases contain information **about individuals** which may not be permitted to be seen by others as individual records. Such databases may contain information about various populations. Example: Medical Records, Personal Data like address, salary, etc

Such kind of databases should have special security mechanisms so that confidential information about people will not be disclosed for many users. Thus, statistical databases should have additional security techniques which will protect the retrieval of individual records. Only queries with statistical aggregate functions like Average, Sum, Min, Max, Standard Deviation, Mid, Count, etc should be executed. Queries retrieving confidential attributes should be prohibited. Not to let the user make inference on the retrieved data, one can also implement constraint on the minimum number of records or tuple in the resulting relation by setting a threshold.

Role of DBA in Database Security

The database administrator is responsible to make the database to be as secure as possible. For this the DBA should have the most powerful privilege than every other user. The DBA provides capability for database users while accessing the content of the database.

The major responsibilities of DBA in relation to authorization of users are:

1. **Account Creation:** involves creating different accounts for different **USERS** as well as **USER GROUPS**. This action creates a new account and password for a user or a group of users to enable access to the DBMS.
2. **Security Level Assignment:** involves in assigning different users at different categories of access levels. This action consists of assigning user accounts to the appropriate security clearance level.
3. **Privilege Grant:** involves giving different levels of privileges for different users and user groups. This action permits the DBA to grant certain privileges to certain accounts.

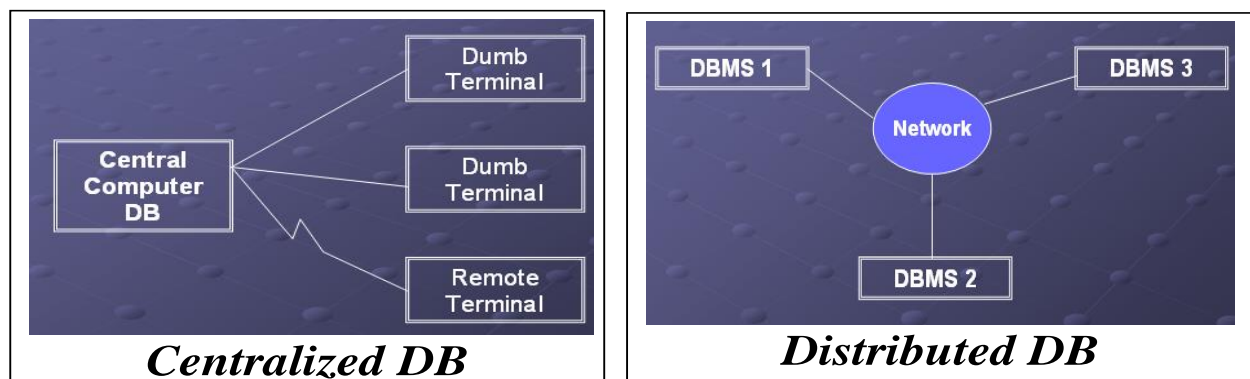
4. **Privilege Revocation:** involves denying or canceling previously granted privileges for users due to various reasons. This action permits the DBA to revoke (cancel) certain privileges that were previously given to certain accounts
5. **Account Deletion:** involves in deleting an existing account of users or user groups. Is similar with denying all privileges of users on the database.

Chapter 5: Distributed Database System

Distributed Database Concepts

Before the development of database technology, every application used to have its own data in the application logic. Database development facilitates the integration of data available in an organization from a number of applications and enforces security on data access on a single local site. But it is not always the case that organizational data reside in one central site. This demand databases at different sites to be integrated and synchronized with all the facilities of database approach. Parallel processing is the opportunity created for computers speedy performance, besides the system should not stop processing while a site stops working. This will be made possible by computer networks and data communication optimized by internet, mobile and wireless computing and intelligent devices.

This leads to **Distributed Database Systems**. The decentralization approach to data base mirrors the natural organizational structure of companies which are logically distributed in to divisions, departments, projects and so on and physically distributed in to offices, plants, and factories each of which maintains its own operational data. **Distributed Database is not a centralized database**. Distributed DBMSs should help resolve **the islands of information problem**



Data Distribution Strategies

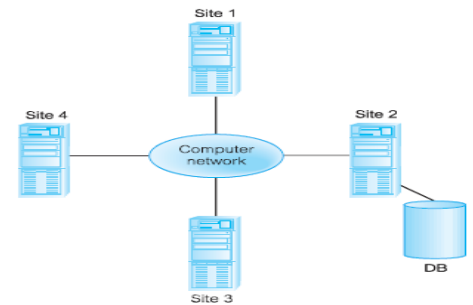
- **Distributed DB** stores logically related shared data and metadata at several physically independent sites connected via network

- **Distributed DBMS** is the software system that permits the management of a Distributed DB and makes the distribution transparent to the user.
- **Data allocation** is the process of deciding where to allocate/**store** particular data item.

It is important to distinguish between a distributed DBMS and distributed processing.

There are 3 data allocation strategies:

- Centralized,
- Partitioned,
- Replicated (Selective replication, Complete replication).



In a **distributed database system**, the database is **logically** stored as single database but **physically** fragmented on several computers. The computers in a distributed system communicate with each other through various communication media, such as high speed buses or telephone line.

A distributed database system consists of a collection of sites, each of which maintains a local database system (Local DBMS) but each local DBMS also participates in **at least one global transaction** where different databases are integrated together.

- **Local Transaction**: transactions that access data only in that **single site**
- **Global Transaction**: transactions that access data **in several sites**.

Data Allocation

There are four alternative strategies regarding the placement of data:

- Centralized,
- Fragmented,
- Complete replication, and
- Selective replication.

Centralized: This strategy consists of a single database and DBMS stored at **one site** with users distributed across the network (we referred to this previously as distributed processing). Locality of reference is at its lowest as all sites, except the central site, have to use the network for all data accesses. This also means that

- ✓ **communication costs are high.**
- ✓ **Reliability and availability are low,**
- ✓ as a failure of the central site results in the **loss of the entire database system.**

Fragmented (or partitioned): This strategy **partitions the database into disjoint fragments**, with each fragment assigned to one site. If data items are located at the site where they are used most frequently, locality of reference is high.

As there is

- ✓ no replication,
- ✓ storage costs are low;
- ✓ similarly, reliability and availability are low,
- ✓ although they are higher than in the centralized case,
- ✓ as the failure of a site results in the loss of only that site's data.
- ✓ **Performance** should be good and communications **costs** low if the distribution is designed properly.

Complete replication: This strategy consists of **maintaining a complete copy of the database at each site.**

Therefore,

- ✓ locality of reference, reliability and availability, and performance are maximized.
- ✓ However, storage costs and communication costs for updates are the **most expensive.**

To overcome some of these problems, **snapshots** are sometimes used. **A snapshot is a copy of the data at a given time.** The copies are updated periodically—for example, hourly or weekly—

so they may not be always up to date. Snapshots are also sometimes used to implement views in a distributed database to improve the time it takes to perform a database operation on a view.

Selective replication: This strategy is a combination of fragmentation, replication, and centralization. Some data items are fragmented to achieve high locality of reference, and others that are used at many sites and are not frequently updated are replicated; otherwise, the data items are centralized. The objective of this strategy is to have all the advantages of the other approaches but none of the disadvantages. This is the most commonly used strategy, because of its flexibility.

Comparison of strategies for data allocation

	Locality of Reference	Reliability and Availability	Performance	Storage Costs	Communication Costs
Centralized	Lowest	Lowest	Unsatisfactory	Lowest	Highest
Fragmented	High	Low for item; high for system	Satisfactory	Lowest	Low
Complete Replication	Highest	Highest	Best for read	Highest	High for update; low for read
Selective replication	High	Low for item; high for system	Satisfactory	Average	Low

Fragmentation

Why fragment? Before we discuss fragmentation in detail,

we list four reasons for fragmenting a relation:

- **Usage.** In general, applications work with views rather than entire relations. Therefore, for data distribution, it seems appropriate to work with subsets of relations as the unit of distribution.
- **Efficiency.** Data is stored close to where it is most frequently used. In addition, data that is not needed by local applications is not stored.

- **Parallelism.** With fragments as the unit of distribution, a transaction can be divided into several sub queries that operate on fragments. This should increase the degree of concurrency, or parallelism, in the system, thereby allowing transactions that can do so safely to execute in parallel.
- **Security.** Data not required by local applications is not stored and consequently not available to unauthorized users.

attributes must be repeated to allow reconstruction. This rule ensures minimal data redundancy.

Types of fragmentation

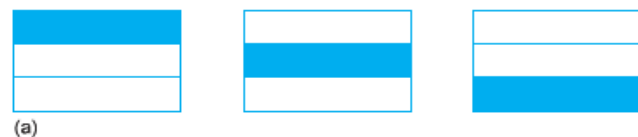
There are two main types of fragmentation:

- ✓ **Horizontal fragments** and
 - ✓ **Vertical fragments**
- + Horizontal fragments are subsets of tuples and
 - + vertical fragments are subsets of attributes

Relation is partitioned into several fragments stored in distinct sites. The partitioning could be *vertical*, *horizontal* or *both*.

Horizontal Fragmentation

- Systems can share the responsibility of storing information from a single table with individual systems storing groups of rows
- Performed by the *Selection Operation*
- The whole content of the relation is reconstructed using the *UNION* operation



(a)



(b)

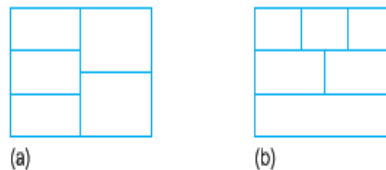
(a) Horizontal and (b) vertical fragmentation.

Vertical Fragmentation

- Systems can share the responsibility of storing particular attributes of a table.
- *Needs attribute with tuple number (the primary key value be repeated.)*
- Performed by the **Projection Operation**
- The whole content of the relation is reconstructed using the **Natural JOIN** operation using the attribute with **Tuple number (primary key values)**

Both (hybrid fragmentation)

- A system can share the responsibility of storing **particular attributes** of a **subset of records** in a given **relation**.
- Performed by projection then selection or selection then projection relational algebra operators.
- Reconstruction is made by combined effect of **Union** and **natural join** operators



Mixed fragmentation: (a) vertical fragments, horizontally fragmented; (b) horizontal fragments, vertically fragmented.

Replication:

- System maintains multiple copies of similar data (identical data)
- Stored in different sites, for faster retrieval and fault tolerance.
- Duplicate copies of the tables can be kept on each system (replicated). With this option, updates to the tables can become involved (of course the copies of the tables can be read-only).
- Advantage: Availability, Increased parallelism (if only reading)
- Disadvantage: increased overhead of update

Advantages and Disadvantages of DDBMSs

The distribution of data and applications has potential advantages over traditional centralized database systems. Unfortunately, there are also disadvantages. In this section, we review the advantages and disadvantages of the DDBMS.

Advantages of DDBMS

1. **Reflects organizational structure** many organizations are naturally distributed over several locations.
2. **Many existing systems:** Maybe you have no choice. Possibly there are many different existing system, with possible different kinds of systems (Oracle, Informix, others) that need to be used together.
3. **Data sharing and distributed control: Improved sharing and local autonomy:** User at one site may be able access data that is available at another site. Each site can retain some degree of control over local data. We will have local as well as global database administrator
4. **Improved Reliability and availability of data:** If one site fails the rest can continue operation as long as transaction does not demand data from the failed system and the data is not replicated in other sites
5. **Improved performance - Speedup of query processing:** If a query involves data from several sites, it may be possible to split the query into sub-queries that can be executed at several sites which is parallel processing. Query can be sent to the least heavily loaded sites
6. **Economics:** However, it is now generally. Accepted that it costs much less to create a system of smaller computers with the equivalent power of a single large computer. This makes it more cost-effective for corporate divisions and departments to obtain separate computers. It is also much more cost-effective to add workstations to a network than to update a mainframe system. The second potential cost saving occurs where databases are geographically remote and the applications require access to distributed data. In such cases, owing to the relative expense of data being transmitted across the network as opposed to the cost of local access, it may be much more economical to partition the application and perform the processing locally at each site.
7. **Expansion (Scalability):** In a distributed environment, you can easily expand by adding more machines to the network. In a distributed environment, it is much easier to handle expansion. New sites can be added to the network without affecting the operations of other sites. This flexibility allows an organization to expand relatively easily. Increasing

database size can usually be handled by adding processing and storage power to the network.

8. **Integration** At the start of this section we noted that integration was a key advantage of the DBMS approach, not centralization. The integration of legacy systems is one particular example that demonstrates how some organizations are forced to rely on distributed data processing to allow their legacy systems to coexist with their more modern systems. At the same time, no one package can provide all the functionality that an organization requires nowadays. Thus, it is important for organizations to be able to integrate software components from different vendors to meet their specific requirements.
9. **Remaining competitive** There are a number of relatively recent developments that rely heavily on distributed database technology such as e-business, computer supported collaborative work, and workflow management. Many enterprises have had to reorganize their businesses and use distributed database technology to remain competitive.

Disadvantages of DDBMS

1. **Greater Potential for Bugs:** Parallel processing may endanger correctness of algorithms
2. **Increased Processing Overhead:** Exchange of message between sites – high communication latency. Due to communication jargons
3. **Complexity** A distributed DBMS that hides the distributed nature from the user and provides an acceptable level of performance, reliability, and availability is inherently more complex than a centralized DBMS. The fact that data can be replicated also adds an extra level of complexity to the distributed DBMS. If the software does not handle data replication adequately, there will be degradation in availability, reliability, and performance compared with the centralized system, and the advantages we cited earlier will become disadvantages.
4. **Cost** Increased complexity means that we can expect the procurement and maintenance costs for a DDBMS to be higher than those for a centralized DBMS. Furthermore, a distributed DBMS requires additional hardware to establish a network between sites. There are ongoing communication costs incurred with the use of this network. There are also additional labor costs to manage and maintain the local DBMSs and the underlying network.

5. **Security** In a centralized system, access to the data can be easily controlled. However, in a distributed DBMS not only does access to replicated data has to be controlled in multiple locations, but the network itself has to be made secure. In the past, networks were regarded as an insecure communication medium. Although this is still partially true, significant developments have been made to make networks more secure.
6. **Integrity control more difficult** Database integrity refers to the validity and consistency of stored data. Integrity is usually expressed in terms of constraints, which are consistency rules that the database is not permitted to violate. Enforcing integrity constraints generally requires access to a large amount of data that defines the constraint but that is not involved in the actual update operation itself. In a distributed DBMS, the communication and processing costs that are required to enforce integrity constraints may be prohibitive.
7. **Lack of standards** Although distributed DBMSs depend on effective communication, we are only now starting to see the appearance of standard communication and data access protocols. This lack of standards has significantly limited the potential of distributed DBMSs. There are also no tools or methodologies to help users convert a centralized DBMS into a distributed DBMS.
8. **Lack of experience** General-purpose distributed DBMSs have not been widely accepted, although many of the protocols and problems are well understood. Consequently, we do not yet have the same level of experience in industry as we have with centralized DBMSs. For a prospective adopter of this technology, this may be a significant deterrent.
9. **Database design more complex** Besides the normal difficulties of designing centralized database, the design of a distributed database has to take account of fragmentation of data, allocation of fragments to specific sites, and data replication.

Homogeneous and Heterogeneous Distributed Databases

A DDBMS may be classified as homogeneous or heterogeneous. In a **homogeneous** system, all sites use the same DBMS product. In a **heterogeneous** system, sites may run different DBMS products, which need not be based on the same underlying data model, and so the system may be composed of relational, network, hierarchical, and object-oriented DBMSs.

■ In a homogeneous distributed database

- All sites have identical software (DBMS)
- Are aware of each other and agree to cooperate in processing user requests.
- Each site surrenders part of its autonomy in terms of right to change schemas or software
- Appears to the user as a single system

■ In a heterogeneous distributed database

- Different sites may use different schemas and software (DBMS)
 - Difference in schema is a major problem for query processing
 - Difference in software is a major problem for transaction processing
- Sites may not be aware of each other and may provide only limited facilities for cooperation in transaction processing.
- May need *gateways* to interface one another.

Transparencies in a DDBMS

Transparency hides implementation details from the user. For example, in a centralized DBMS data independence is a form of transparency—it hides changes in the definition and organization of the data from the user. A DDBMS may provide various levels of transparency. However, they all participate in the same overall objective: to make the use of the distributed database equivalent to that of a centralized database. We can identify **four** main types of transparency in a DDBMS:

- Distribution transparency;
 - Transaction transparency;
 - Performance transparency;
 - DBMS transparency.
- **Data transparency:** The degree to which system user may remain unaware of the details of how and where the data items are stored in a distributed system.

- **Distribution transparency** Even though there are many systems they appear as one- seen as a single, logical entity.
- **Replication transparency** Copies of data floating around everywhere also seem like just one copy to the developers and users
- **Fragmentation transparency** A table that is actually stored in parts everywhere across sites may seem like just a single table in a single
- **Location Transparency-** the user doesn't need to know where a data item is physically located.

Relational Query Languages

- **Query languages:** Allow manipulation and retrieval of data from a database.
- Query Languages is not a programming languages
 - QLS not intended to be used for complex calculations.
 - QLS support easy, efficient access to large data sets.
- Relational model supports simple, powerful query languages.

Formal Relational Query Languages

- There are varieties of Query languages used by relational DBMS for manipulating relations.
- Some of them are procedural
 - User tells the system exactly *what* and *how* to manipulate the data
- Others are non-procedural
 - User states *what data is needed* rather than *how* it is to be retrieved.

Two mathematical Query Languages form the basis for Relational languages

- Relational Algebra:
- Relational Calculus:
- We may describe the relational algebra as procedural language: it can be used to tell the DBMS how to build a new relation from one or more relations in the database.
- We may describe relational calculus as a non procedural language: it can be used to formulate the definition of a relation in terms of one or more database relations.
- Formally the relational algebra and relational calculus are equivalent to each other.
- *For every expression in the algebra, there is an equivalent expression in the calculus.*
- Both are non-user friendly languages. They have been used as the basis for other, higher-level data manipulation languages for relational databases.

A query is applied to relation instances, and the result of a query is also a relation instance.

- **Schemas** of input relations for a query are **fixed**
- The schema for the *result* of a given query is also fixed! Determined by definition of query language constructs.

Relational Algebra

The basic set of operations for the relational model is known as the relational algebra. These operations enable a user to specify basic retrieval requests.

The result of the retrieval is a new relation, which may have been formed from one or more relations. The **algebra operations** thus produce new relations, which can be further manipulated using operations of the same algebra.

A sequence of relational algebra operations forms a **relational algebra expression**, whose result will also be a relation that represents the result of a database query (or retrieval request).

- Relational algebra is a theoretical language with operations that work on one or more relations to define another relation without changing the original relation.
- The output from one operation can become the input to another operation (nesting is possible)
- **There are different basic operations that could be applied on relations on a database based on the requirement.**
 - **Selection** (σ) Selects a subset of rows from a relation.

- **Projection (π)** Deletes unwanted columns from a relation.
 - **Renaming:** assigning intermediate relation for a single operation
 - **Cross-Product (\times)** Allows us to combine two relations.
 - **Set-Difference ($-$)** Tuples in relation1, but not in relation2.
 - **Union (\cup)** Tuples in relation1 or in relation2.
 - **Intersection (\cap)** Tuples in relation1 and in relation2
 - **Join** \bowtie Tuples joined from two relations based on a condition
- Using these we can build up sophisticated database queries.

Table1:

Sample table used to illustrate different kinds of relational operations. The relation contains information about employees, IT skills they have and the school where they attend each skill. The primary key for this table is EmpId and Skill ID since a single employee can have multiple skills and a single skill be acquired by many employees.

School address is the address of a school for which the address of the main office will be considered in cases where a single school has many branches at different locations.

Employee

<u>EmpID</u>	FName	LName	<u>SkillID</u>	Skill	SkillType	School	SchoolAdd	SkillLevel
12	Abebe	Mekuria	2	SQL	Database	AAU	Sidist_Kilo	5
16	Lemma	Alemu	5	C++	Programming	Unity	Gerji	6
28	Chane	Kebede	2	SQL	Database	AAU	Sidist_Kilo	10
25	Abera	Taye	6	VB6	Programming	Helico	Piazza	8
65	Almaz	Belay	2	SQL	Database	Helico	Piazza	9
24	Dereje	Tamiru	8	Oracle	Database	Unity	Gerji	5
51	Selam	Belay	4	Prolog	Programming	Jimma	Jimma City	8
94	Alem	Kebede	3	Cisco	Networking	AAU	Sidist_Kilo	7
18	Girma	Dereje	1	IP	Programming	Jimma	Jimma City	4
13	Yared	Gizaw	7	Java	Programming	AAU	Sidist_Kilo	6

1. Selection

- Selects subset of tuples/rows in a relation that satisfy *selection condition*.
- Selection operation is a unary operator (it is applied to a single relation)
- The Selection operation is applied to each tuple individually
- The degree of the resulting relation is the same as the original relation but the cardinality (no. of tuples) is less than or equal to the original relation.
- The Selection operator is commutative.
- Set of conditions can be combined using Boolean operations (\wedge (AND), \vee (OR), and \sim (NOT))
- No duplicates in result!
- *Schema* of result identical to schema of (only) input relation.
- *Result* relation can be the *input* for another relational algebra operation! (*Operator composition*.)
- It is a filter that keeps only those tuples that satisfy a qualifying condition (those satisfying the condition are selected while others are discarded.)

Notation:

$\sigma_{\langle \textit{Selection Condition} \rangle} \langle \textit{Relation Name} \rangle$

Example: Find all Employees with skill type of Database.

$\sigma_{\langle \textit{SkillType} = \textit{"Database"} \rangle} (\textit{Employee})$

This query will extract every tuple from a relation called Employee with all the attributes where the SkillType attribute with a value of “Database”.

The resulting relation will be the following.

<i>EmpID</i>	<i>FName</i>	<i>LName</i>	<i>SkillID</i>	<i>Skill</i>	<i>SkillType</i>	<i>School</i>	<i>SchoolAdd</i>	<i>SkillLevel</i>
12	Abebe	Mekuria	2	SQL	Database	AAU	Sidist_Kilo	5
28	Chane	Kebede	2	SQL	Database	AAU	Sidist_Kilo	10

65	Almaz	Belay	2	SQL	Database	Helico	Piazza	9
24	Dereje	Tamiru	8	Oracle	Database	Unity	Gerji	5

If the query is all employees with a SkillType *Database* and School *Unity* the relational algebra operation and the resulting relation will be as follows.


$\sigma_{\langle SkillType = "Database" \text{ AND } School = "Unity" \rangle} (Employee)$

<i>EmpID</i>	<i>FName</i>	<i>LName</i>	<i>SkillID</i>	<i>Skill</i>	<i>SkillType</i>	<i>School</i>	<i>SchoolAdd</i>	<i>SkillLevel</i>
24	Dereje	Tamiru	8	Oracle	Database	Unity	Gerji	5


2. Projection

- Selects certain attributes while discarding the other from the base relation.
- The PROJECT creates a vertical partitioning – one with the needed columns (attributes) containing results of the operation and other containing the discarded Columns.
- Deletes attributes that are not in *projection list*.
- *Schema* of result contains exactly the fields in the projection list, with the same names that they had in the (only) input relation.
- Projection operator has to eliminate *duplicates*!
 - Note: real systems typically don't do duplicate elimination unless the user explicitly asks for it.
- If the Primary Key is in the *projection list*, then duplication will not occur
- Duplication removal is necessary to insure that the resulting table is also a relation.

Notation:

 *<Selected Attributes>* *<Relation Name>*

Example: To display Name, Skill, and Skill Level of an employee, the query and the resulting relation will be:

 *<FName, LName, Skill, Skill_Level>* (*Employee*)

<i>FName</i>	<i>LName</i>	<i>Skill</i>	<i>SkillLevel</i>
Abebe	Mekuria	SQL	5
Lemma	Alemu	C++	6
Chane	Kebede	SQL	10
Abera	Taye	VB6	8
Almaz	Belay	SQL	9
Dereje	Tamiru	Oracle	5
Selam	Belay	Prolog	8
Alem	Kebede	Cisco	7

Girma	Dereje	IP	4
Yared	Gizaw	Java	6

If we want to have the Name, Skill, and Skill Level of an employee with Skill SQL and SkillLevel greater than 5 the query will be:

$\langle FName, LName, Skill, Skill_Level \rangle$ ($\langle Skill = "SQL" \wedge SkillLevel > 5 \rangle$ (***Employee***))

<i>FName</i>	<i>LName</i>	<i>Skill</i>	<i>SkillLevel</i>
Chane	Kebede	SQL	10
Almaz	Belay	SQL	9

3. Rename Operation

- We may want to apply several relational algebra operations one after the other. The query could be written in two different forms:

1. Write the operations as a single relational algebra expression by nesting the operations.
2. Apply one operation at a time and create intermediate result relations. In the latter case, we must give names to the relations that hold the intermediate results → Rename Operation

If we want to have the Name, Skill, and Skill Level of an employee with salary greater than 1500 and working for department 5, we can write the expression for this query using the two alternatives:

1. A single algebraic expression:

The above used query is using a single algebra operation, which is:

$$\sigma_{\langle FName, LName, Skill, Skill_Level \rangle} \left(\sigma_{\langle Skill="SQL" \wedge SkillLevel > 5 \rangle} (Employee) \right)$$

2. Using an intermediate relation by the Rename Operation:

$$\text{Step1: } Result1 \leftarrow \sigma_{\langle DeptNo=5 \wedge Salary > 1500 \rangle} (Employee)$$

$$\text{Step2: } Result \leftarrow \sigma_{\langle FName, LName, Skill, Skill_Level \rangle} (Result1)$$

Then Result will be equivalent with the relation we get using the first alternative.

4. Set Operations

The three main set operations are the Union, Intersection and Set Difference. The properties of these set operations are similar with the concept we have in mathematical set theory. The difference is that, in database context, the elements of each set, which is a Relation in Database, will be tuples. The set operations are Binary operations which demand the two operand Relations to have type compatibility feature.

Type Compatibility

Two relations R_1 and R_2 are said to be Type Compatible if:

1. The operand relations $R_1(A_1, A_2, \dots, A_n)$ and $R_2(B_1, B_2, \dots, B_n)$ have the same number of attributes, and
2. The domains of corresponding attributes must be compatible; that is, $\text{Dom}(A_i) = \text{Dom}(B_i)$ for $i=1, 2, \dots, n$.

To illustrate the three set operations, we will make use of the following two tables:

Employee

<u>EmpID</u>	<i>FName</i>	<i>LName</i>	<u>SkillID</u>	<i>Skill</i>	<i>SkillType</i>	<i>School</i>	<i>SkillLevel</i>
12	Abebe	Mekuria	2	SQL	Database	AAU	5
16	Lemma	Alemu	5	C++	Programming	Unity	6
28	Chane	Kebede	2	SQL	Database	AAU	10
25	Abera	Taye	6	VB6	Programming	Helico	8
65	Almaz	Belay	2	SQL	Database	Helico	9
24	Dereje	Tamiru	8	Oracle	Database	Unity	5
51	Selam	Belay	4	Prolog	Programming	Jimma	8
94	Alem	Kebede	3	Cisco	Networking	AAU	7
18	Girma	Dereje	1	IP	Programming	Jimma	4
13	Yared	Gizaw	7	Java	Programming	AAU	6

RelationOne: Employees who attend Database Course

<u>EmpID</u>	<i>FName</i>	<i>LName</i>	<u>SkillID</u>	<i>Skill</i>	<i>SkillType</i>	<i>School</i>	<i>SkillLevel</i>
--------------	--------------	--------------	----------------	--------------	------------------	---------------	-------------------

12	Abebe	Mekuria	2	SQL	Database	AAU	5
28	Chane	Kebede	2	SQL	Database	AAU	10
65	Almaz	Belay	2	SQL	Database	Helico	9
24	Dereje	Tamiru	8	Oracle	Database	Unity	5

RelationTwo : Employees who attend a course in AAU

<u>EmpID</u>	<i>FName</i>	<i>LName</i>	<u>SkillID</u>	<i>Skill</i>	<i>SkillType</i>	<i>School</i>	<i>SkillLevel</i>
12	Abebe	Mekuria	2	SQL	Database	AAU	5
94	Alem	Kebede	3	Cisco	Networking	AAU	7
28	Chane	Kebede	2	SQL	Database	AAU	10
13	Yared	Gizaw	7	Java	Programming	AAU	6

a. UNION Operation

The result of this operation, denoted by $R \cup S$, is a relation that includes all tuples that are either in R or in S or in both R and S. Duplicate tuple is eliminated.

The two operands must be "type compatible"

Eg: RelationOne \cup RelationTwo

Employees who attend Database in any School or who attend any course at AAU

<u>EmpID</u>	<i>FName</i>	<i>LName</i>	<u>SkillID</u>	<i>Skill</i>	<i>SkillType</i>	<i>School</i>	<i>SkillLevel</i>
12	Abebe	Mekuria	2	SQL	Database	AAU	5
28	Chane	Kebede	2	SQL	Database	AAU	10
65	Almaz	Belay	2	SQL	Database	Helico	9
24	Dereje	Tamiru	8	Oracle	Database	Unity	5
94	Alem	Kebede	3	Cisco	Networking	AAU	7
13	Yared	Gizaw	7	Java	Programming	AAU	6

b. INTERSECTION Operation

The result of this operation, denoted by $R \cap S$, is a relation that includes all tuples that are in both R and S. The two operands must be "type compatible"

Eg: $RelationOne \cap RelationTwo$

Employees who attend Database Course at AAU

<u>EmpID</u>	<u>FName</u>	<u>LName</u>	<u>SkillID</u>	<u>Skill</u>	<u>SkillType</u>	<u>School</u>	<u>SkillLevel</u>
12	Abebe	Mekuria	2	SQL	Database	AAU	5
28	Chane	Kebede	2	SQL	Database	AAU	10

c. Set Difference (or MINUS) Operation

The result of this operation, denoted by $R - S$, is a relation that includes all tuples that are in R but not in S.

The two operands must be "type compatible"

Eg: $RelationOne - RelationTwo$

Employees who attend Database Course but didn't take any course at AAU

<u>EmpID</u>	<u>FName</u>	<u>LName</u>	<u>SkillID</u>	<u>Skill</u>	<u>SkillType</u>	<u>School</u>	<u>SkillLevel</u>
65	Almaz	Belay	2	SQL	Database	Helico	9
24	Dereje	Tamiru	8	Oracle	Database	Unity	5

Eg: $RelationTwo - RelationOne$

Employees who attend Database Course but didn't take any course at AAU

<u>EmpID</u>	<u>FName</u>	<u>LName</u>	<u>SkillID</u>	<u>Skill</u>	<u>SkillType</u>	<u>School</u>	<u>SkillLevel</u>
12	Abebe	Mekuria	2	SQL	Database	AAU	5
94	Alem	Kebede	3	Cisco	Networking	AAU	7
28	Chane	Kebede	2	SQL	Database	AAU	10
13	Yared	Gizaw	7	Java	Programming	AAU	6

The resulting relation for; $R_1 \cup R_2$, $R_1 \cap R_2$, or $R_1 - R_2$ has the same attribute names as the first operand relation R_1 (by convention).

Some Properties of the Set Operators

Notice that both union and intersection are commutative operations; that is

$$\mathbf{R \cup S = S \cup R, and R \cap S = S \cap R}$$

Both union and intersection can be treated as n-ary operations applicable to any number of relations as both are associative operations; that is

$$\mathbf{R \cup (S \cup T) = (R \cup S) \cup T, and (R \cap S) \cap T = R \cap (S \cap T)}$$

The minus operation is not commutative; that is, in general

$$\mathbf{R - S \neq S - R}$$

5. CARTESIAN Operation (Cross Product)

This operation is used to combine tuples from two relations in a combinatorial fashion. That means, every tuple in Relation1(R) one will be related with every other tuple in Relation2 (S).

- In general, the result of $\mathbf{R(A_1, A_2, \dots, A_n) \times S(B_1, B_2, \dots, B_m)}$ is a relation \mathbf{Q} with degree $\mathbf{n + m}$ attributes $\mathbf{Q(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m)}$, in that order.
- Where \mathbf{R} has \mathbf{n} attributes and \mathbf{S} has \mathbf{m} attributes.
- The resulting relation \mathbf{Q} has one tuple for each combination of tuples—one from \mathbf{R} and one from \mathbf{S} .
- Hence, if \mathbf{R} has \mathbf{n} tuples, and \mathbf{S} has \mathbf{m} tuples, then $\mathbf{| R \times S |}$ will have $\mathbf{n * m}$ tuples.

Example:

Employee

ID	FName	LName
123	Abebe	Lemma
567	Belay	Taye
822	Kefle	Kebede

Dept

DeptID	DeptName	MangID
2	Finance	567
3	Personnel	123

Then the Cartesian product between Employee and Dept relations will be of the form:

Employee X Dept:

ID	FName	LName	DeptID	DeptName	MangID
123	Abebe	Lemma	2	Finance	567
123	Abebe	Lemma	3	Personnel	123
567	Belay	Taye	2	Finance	567
567	Belay	Taye	3	Personnel	123
822	Kefle	Kebede	2	Finance	567
822	Kefle	Kebede	3	Personnel	123

Basically, even though it is very important in query processing, the Cartesian Product is not useful by itself since it relates every tuple in the First Relation with every other tuple in the Second Relation. Thus, to make use of the Cartesian Product, one has to use it with the Selection Operation, which discriminate tuples of a relation by testing whether each will satisfy the selection condition.

In our example, to extract employee information about managers of the departments (Managers of each department), the algebra query and the resulting relation will be.

$\square_{<ID, FName, LName, DeptName>} (\square_{<ID=MangID>} (Employee \ X \ Dept))$

ID	FName	LName	DeptName
123	Abebe	Lemma	Personnel
567	Belay	Taye	Finance

6. JOIN Operation

The sequence of Cartesian product followed by select is used quite commonly to identify and select related tuples from two relations, a special operation, called **JOIN**. Thus in JOIN operation, the Cartesian Operation and the Selection Operations are used together.

JOIN Operation is denoted by a \bowtie symbol.

This operation is very important for any relational database with more than a single relation, because it allows us to process relationships among relations.

The general form of a join operation on two relations

$R(A_1, A_2, \dots, A_n)$ and $S(B_1, B_2, \dots, B_m)$ is:

$R \bowtie_{<join\ condition>} S$ is equivalent to $\square_{<selection\ condition>} (R \ X \ S)$

where $<join\ condition>$ and $<selection\ condition>$ are the same

Where, R and S can be any relation that results from general relational algebra expressions.

Since JOIN is an operation that needs two relation, it is a Binary operation.

This type of JOIN is called a **THETA JOIN (θ - JOIN)**

Where θ is the logical operator used in the join condition.

θ Could be $\{ <, \leq, >, \geq, \neq, = \}$

Example:

Thus in the above example we want to extract employee information about managers of the departments, the algebra query using the JOIN operation will be.

$Employee \bowtie_{< ID=MangID >} Dept$

a. EQUIJOIN Operation

The most common use of join involves join conditions with equality comparisons only (=). Such a join, where the only comparison operator used is called an EQUIJOIN. In the result of an EQUIJOIN we always have one or more pairs of attributes (whose names need not be identical) that have identical values in every tuple since we used the equality logical operator.

For example, the above JOIN expression is an EQUIJOIN since the logical operator used is the equal to operator (=).

b. NATURAL JOIN Operation

We have seen that in EQUIJOIN one of each pair of attributes with identical values is extra, a new operation called **natural join** was created to get rid of the second (or extra) attribute that we will have in the result of an EQUIJOIN condition.

The standard definition of natural join requires that the two join attributes, or each pair of corresponding join attributes, have the same name in both relations. If this is not the case, a renaming operation on the attributes is applied first.

c. OUTER JOIN Operation

OUTER JOIN is another version of the JOIN operation where non matching tuples from a relation are also included in the result with NULL values for attributes in the other relation.

There are two major types of OUTER JOIN.

1. **RIGHT OUTER JOIN**: where non matching tuples from the second (Right) relation are included in the result with NULL value for attributes of the first (Left) relation.
2. **LEFT OUTER JOIN**: where non matching tuples from the first (Left) relation are included in the result with NULL value for attributes of the second (Right) relation.

Notation for Left Outer Join:

$$R \bowtie \triangleright_{\langle \text{Join Condition} \rangle} S$$

When two relations are joined by a JOIN operator, there could be some tuples in the first relation not having a matching tuple from the second relation, and the query is interested to display these non matching tuples from the first or second relation. Such query is represented by the OUTER JOIN.

d. SEMIJOIN Operation

SEMI JOIN is another version of the JOIN operation where the resulting Relation will contain those attributes of only one of the Relations that are related with tuples in the other Relation. The following notation depicts the inclusion of only the attributes form the first relation (R) in the result which are actually participating in the relationship.

Let's create after triggers. First of all, let's create a table and insert some sample data. Then, on this table, I will be attaching several triggers.

```
CREATE TABLE Employee_Test
(
Emp_ID INT Identity,
Emp_name Varchar(100),
Emp_Sal Decimal (10,2)
)

INSERT INTO Employee_Test VALUES ('Anees',1000);
INSERT INTO Employee_Test VALUES ('Rick',1200);
INSERT INTO Employee_Test VALUES ('John',1100);
INSERT INTO Employee_Test VALUES ('Stephen',1300);
INSERT INTO Employee_Test VALUES ('Maria',1400);
```

Now, create the audit table as:-

```
CREATE TABLE Employee_Test_Audit
(
Emp_ID int,
Emp_name varchar(100),
Emp_Sal decimal (10,2),
Audit_Action varchar(100),
Audit_Timestamp datetime
)
```

Insert Trigger

This trigger is fired after an INSERT on the table. Let's create the trigger as:

```
CREATE TRIGGER trgAfterInsert ON [dbo].[Employee_Test]
FOR INSERT
AS
    declare @empid int;
    declare @empname varchar(100);
    declare @empsal decimal(10,2);
    declare @audit_action varchar(100);

    select @empid=i.Emp_ID from inserted i;
    select @empname=i.Emp_Name from inserted i;
    select @empsal=i.Emp_Sal from inserted i;
    set @audit_action='Inserted Record -- After Insert Trigger.';

    insert into Employee_Test_Audit
        (Emp_ID,Emp_Name,Emp_Sal,Audit_Action,Audit_Timestamp)
    values(@empid,@empname,@empsal,@audit_action,getdate());

    PRINT 'AFTER INSERT trigger fired.'
```

GO

UPDATE Trigger

This trigger is fired after an update on the table. Let's create the trigger as:

```
CREATE TRIGGER trgAfterUpdate ON [dbo].[Employee_Test]
FOR UPDATE
AS
    declare @empid int;
    declare @empname varchar(100);
    declare @empsal decimal(10,2);
    declare @audit_action varchar(100);

    select @empid=i.Emp_ID from inserted i;
    select @empname=i.Emp_Name from inserted i;
    select @empsal=i.Emp_Sal from inserted i;

    if update(Emp_Name)
        set @audit_action='Updated Record -- After Update Trigger.';
    if update(Emp_Sal)
        set @audit_action='Updated Record -- After Update Trigger.';

    insert into Employee_Test_Audit(Emp_ID,Emp_Name,Emp_Sal,Audit_Action,Audit_Timestamp)
    values(@empid,@empname,@empsal,@audit_action,getdate());

    PRINT 'AFTER UPDATE Trigger fired.'
```

GO

DELETE Trigger

This trigger is fired after a delete on the table. Let's create the trigger as:

```
CREATE TRIGGER trgAfterDelete ON [dbo].[Employee_Test]
AFTER DELETE
AS
    declare @empid int;
    declare @empname varchar(100);
    declare @empsal decimal(10,2);
    declare @audit_action varchar(100);

    select @empid=d.Emp_ID from deleted d;
    select @empname=d.Emp_Name from deleted d;
    select @empsal=d.Emp_Sal from deleted d;
    set @audit_action='Deleted -- After Delete Trigger.';

    insert into Employee_Test_Audit
    (Emp_ID,Emp_Name,Emp_Sal,Audit_Action,Audit_Timestamp)
    values(@empid,@empname,@empsal,@audit_action,getdate());

    PRINT 'AFTER DELETE TRIGGER fired.'
```

GO