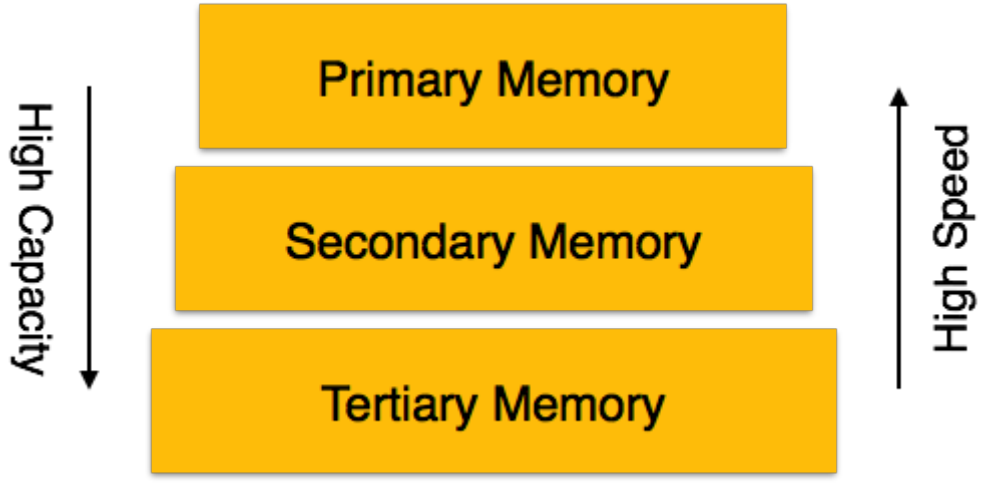
Storage and File Structure

- Press `Space` to navigate through the slides
- Use `Shift+Space` to go back

Storage System

- Databases are stored in **file formats**, which contain **records**.
- At **physical level**, the actual data is stored in electromagnetic format on a device.
- These storage devices can be broadly categorised into **three types**:



Primary Storage 

* Primary Storage is the memory storage that is directly accessible to the CPU comes under this category.

- CPU's **internal memory** (registers), **fast memory** (cache) and **main memory** (RAM) are directly accessible to the **CPU**.
- These components are all placed on the **motherboard** or **CPU chipset**.
- This storage is typically **very small**, **ultra-fast** and **volatile**.

- **Primary storage** requires **continuous power supply** in order to maintain its state.
- In case of a **power failure**, all its **data is lost**.

Secondary Storage 

- **Secondary storage** devices are used to store data for **future use** or as **backup**.
- **Secondary storage** includes **memory devices** that are not a part of the **CPU chipset** or **motherboard**.
- For example:
    - **magnetic disks**
    - **optical disks (DVD, CD, etc.)**
    - **hard disks**
    - **flash drives**
    - **magnetic tapes**

Tertiary Storage

- **Tertiary storage** (pronounced as **tuh·shuh·ree**) is used to store huge volumes of data.
- Since such storage devices are **external** to the computer system, they are the **slowest in speed**.
- These storage devices are mostly used to take the **back up of an entire system**.
- **Optical disks** and **magnetic tapes** are widely used as **tertiary storage**.

Memory Hierarchy 

- A computer system has a well-defined **hierarchy** of memory.
- A **CPU** has direct access to it **main memory** as well as its **inbuilt registers**.
- The **access time** of the **main memory** is obviously **less than the CPU speed**.
- To minimise this speed mismatch, **cache memory** is introduced.
- **Cache memory** provides the fastest access time and it contains data that is most frequently accessed by the **CPU**.
- The **memory with the fastest access** is the **costliest one**.
- **Larger storage** devices offer **slow speed** and they **are less expensive**, however they can store huge volumes of data as compared to CPU registers or cache memory.

Magnetic Disks

- **Hard disk** drives are the most common secondary storage devices in present computer systems.
- These are called **magnetic disks** because they use the concept of magnetisation to store information. Hard disks consist of metal disks coated with magnetisable material.
- These disks are placed vertically on a spindle.
- A **read/write** head moves in between the disks and is used to magnetise or de-magnetise the spot under it.
- A magnetised spot can be recognised as **0 (zero)** or **1 (one)**.

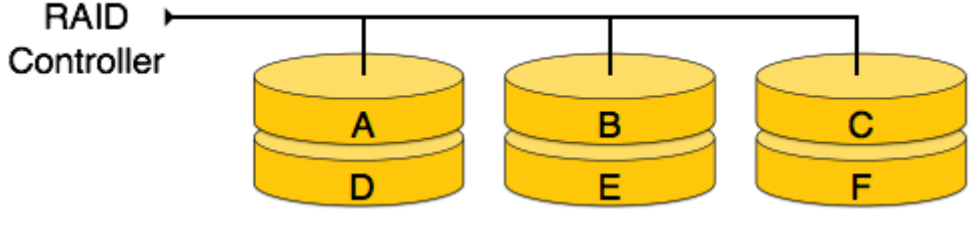
- **Hard disks** are formatted in a well-defined order to store data efficiently.
- A hard disk plate has many concentric circles on it, called **tracks**.
- Every track is further divided into **sectors**.
- A sector on a hard disk typically stores **512 bytes** of data.

Redundant Array of Independent Disks 

- **RAID** or **R**edundant **A**rray of **I**ndependent **D**isks, is a technology to connect multiple secondary storage devices and use them as a single storage media.
- **RAID** consists of an array of disks in which multiple disks are connected together to achieve different goals.
- **RAID** levels define the use of disk arrays:
    - **RAID 0 (striping)**
    - **RAID 1 (mirroring)**
    - **RAID 2-4**
    - **RAID 5 (distributed parity)**
    - **RAID 6 (dual parity)**

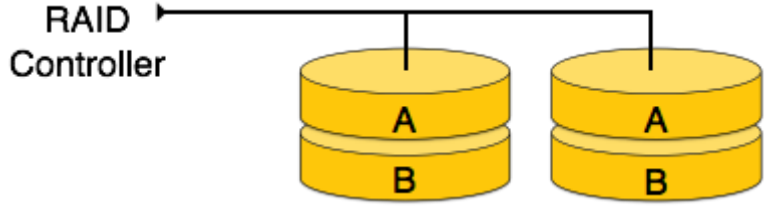
RAID 0 

- In this level, a **striped array of disks** is implemented.
- The data is broken down into blocks and the blocks are distributed among disks.
- Each disk receives a block of data to write/read in parallel.
- It enhances the speed and performance of the storage device.
- There is no parity and backup in **Level 0**.



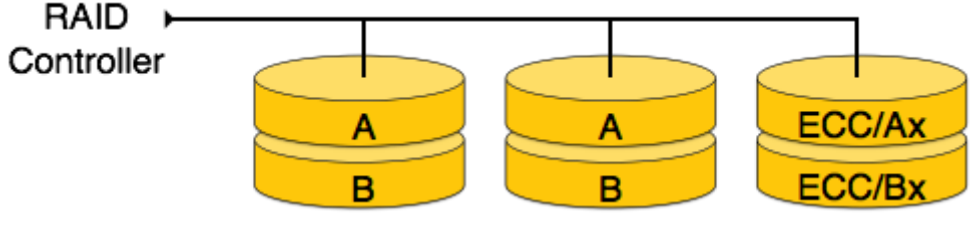
RAID 1 

- **RAID 1** uses **mirroring techniques**.
- When data is sent to a **RAID controller**, it sends a copy of data to all the disks in the array.
- **RAID Level 1** is also called mirroring and provides 100% redundancy in case of a failure.



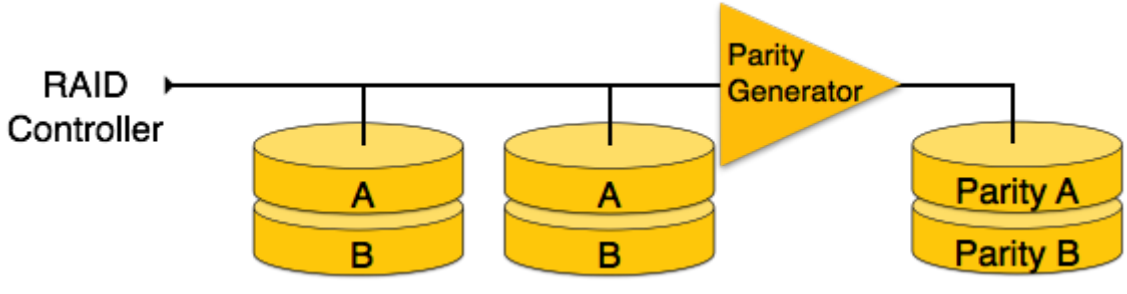
RAID 2 

- **RAID 2** records **Error Correction Code** using **Hamming distance** for its data, striped on different disks.
- Like **level 0**, each data bit in a word is recorded on a separate disk and ECC codes of the data words are stored on a different set disks.
- Due to its complex structure and high cost, **RAID 2 is not commercially available**.



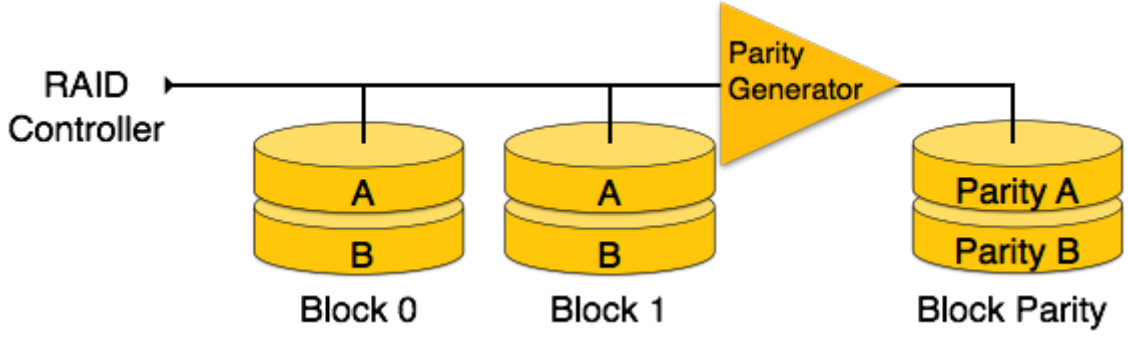
RAID 3 

- **RAID 3** stripes the data onto multiple disks.
- The **parity bit** generated for data word is stored on a different disk.
- This technique makes it to **overcome single disk failures**.



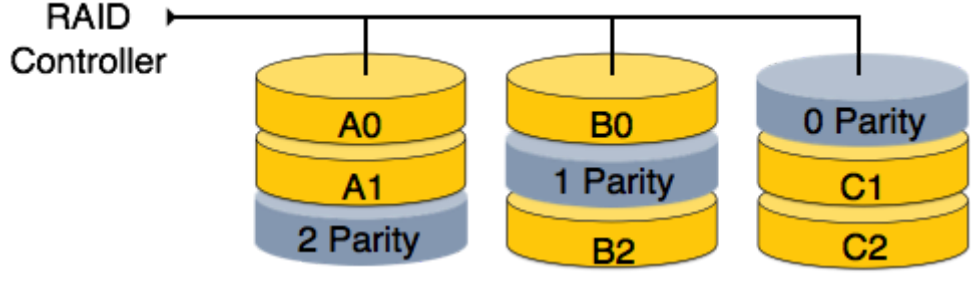
RAID 4 

- In this level, an **entire block of data** is written onto data disks and then the **parity is generated and stored** on a different disk.
- Note that **level 3** uses **byte-level striping**, whereas **level 4** uses **block-level striping**.
- Both **level 3** and **level 4** require at least three disks to implement **RAID**.



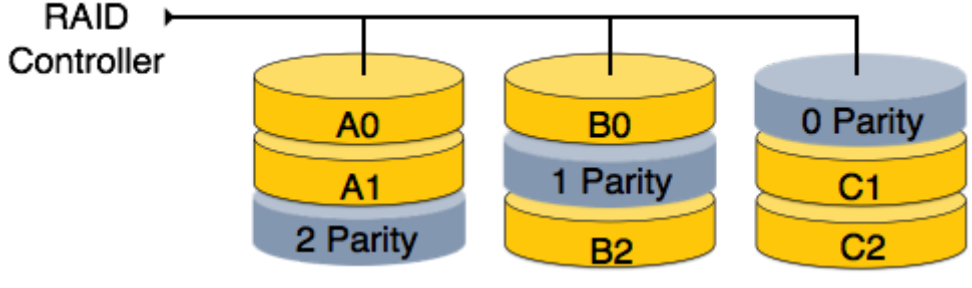
RAID 5 

- **RAID 5** writes whole data blocks onto different disks, but the parity bits generated for data block stripe are distributed among all the data disks rather than storing them on a different dedicated disk.



RAID 6 

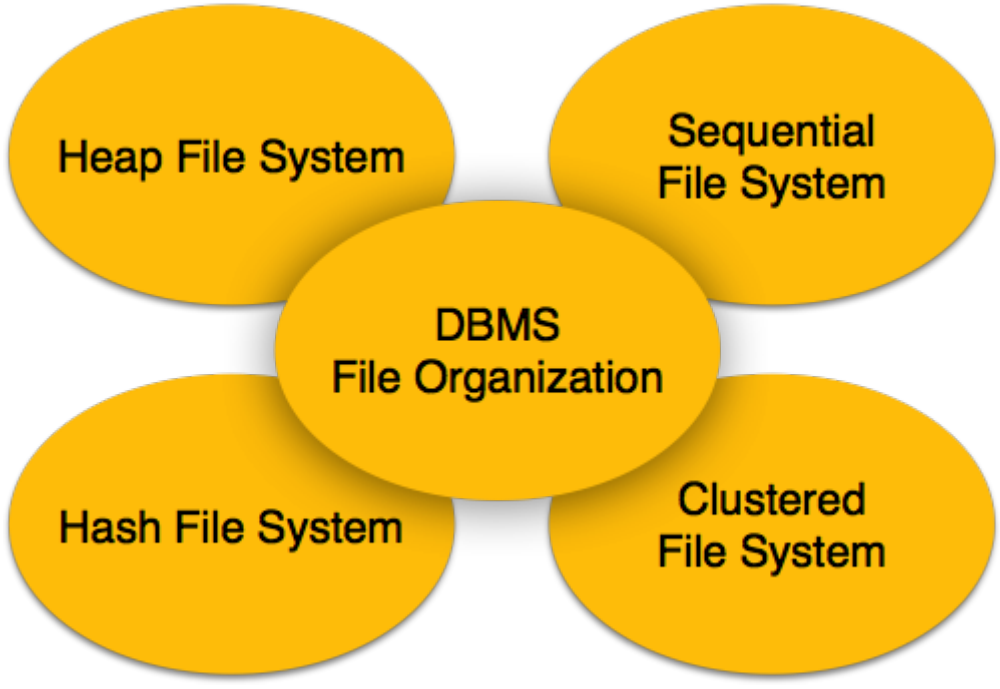
- **RAID 6** is an extension of **level 5**.
- In this level, **two independent parities** are generated and stored in distributed fashion among multiple disks.
- **Two parities** provide additional **fault tolerance**.
- This level requires **at least four disk drives** to implement **RAID**.



File Organisation 

- **Relative data** and **information** is stored collectively in **file formats**.
- **A file** is a sequence of records stored in **binary format**.
- **A disk drive** is formatted into **several blocks that can store records**.
- **File records** are mapped onto those **disk blocks**.

- **File organisation** defines how file records are mapped onto disk blocks.
- We have **four types** of **file organisation** to organise file records:


Heap File System
Sequential
File System
DBMS
File Organization
Hash File System
Clustered
File System

Sequential File Organisation

- Every file record contains a **data field (attribute)** to uniquely identify that record
- In sequential file organisation, records are placed in the file in some sequential order based on the **unique key field** or **search key**
- Practically, it is not possible to store all the records sequentially in physical form

Hash File Organisation

- **Hash File Organisation** uses **Hash function** computation on some fields of the records
- The **output** of the **hash function** determines the location of disk block where the records are to be placed

Clustered File Organisation

- **Clustered file organisation** is not considered good for large databases
- In this mechanism, **related records from one or more relations are kept in the same disk block**, that is, the ordering of records is not based on primary key or search key

File Operations

- **Operations** on database files can be broadly classified into **two categories**:
    - **Update Operations**
        - Change the data values by insertion, deletion or update
    - **Retrieval Operations**
        - Do not alter the data but retrieve them after optional conditional filtering
- In both types of operations, **selection plays a significant role**

File Operations 

- Other than **creation** and **deletion** of a file, there could be several operations, which can be done on files:
    - **Open**
        - A file can be opened in one of the two modes, read mode or write mode
        - In read mode, the operating system does not allow anyone to alter data
        - In other words, data is read only
        - Files opened in read mode can be shared among several entities
        - Write mode allows data modification
        - Files opened in write mode can be read but cannot be shared
    - **Locate**
        - Every file has a file pointer, which tells the current position where the data is to be read or written
        - This pointer can be adjusted accordingly
        - Using find (seek) operation, it can be moved forward or backward

File Operations

- **Read**
    - By default, when files are opened in read mode, the file pointer points to the beginning of the file
    - There are options where the user can tell the operating system where to locate the file pointer at the time of opening a file
    - The very next data to the file pointer is read
- **Write**
    - User can select to open a file in write mode, which enables them to edit its contents
    - It can be deletion, insertion, or modification
    - The file pointer can be located at the time of opening or can be dynamically changed if the operating system allows to do so

File Operations 

- **Close**
    - This is the most important operation from the operating system's point of view
    - When a request to close a file is generated, the operating system:
        - removes all the locks (if in shared mode)
        - saves the data (if altered) to the secondary storage media
        - releases all the buffers and file handlers associated with the file

---

- **The organisation of data inside a file plays a major role**

Indexing 

- Press `Space` to navigate through the slides
- Use `Shift+Space` to go back

Indexing 

- We know that data is stored in the form of records.
- Every record has a key field, which helps it to be recognised uniquely.

- **Indexing** is a data structure technique to efficiently retrieve records from the database files based on some attributes on which the indexing has been done.
- **Indexing** in database systems is similar to what we see in books.

- **Indexing** is defined based on its indexing attributes.
- **Indexing** can be of the following types:
    - **Primary Index** − Primary index is defined on an ordered data file. The data file is ordered on a **key field**. The key field is generally the primary key of the relation.
    - **Secondary Index** − Secondary index may be generated from a field which is a candidate key and has a unique value in every record, or a non-key with duplicate values.
    - **Clustering Index** − Clustering index is defined on an ordered data file. The data file is ordered on a non-key field.

Ordered Indexing 

Ordered Indexing is of two types:

- **Dense Index**
- **Sparse Index**

Dense Index

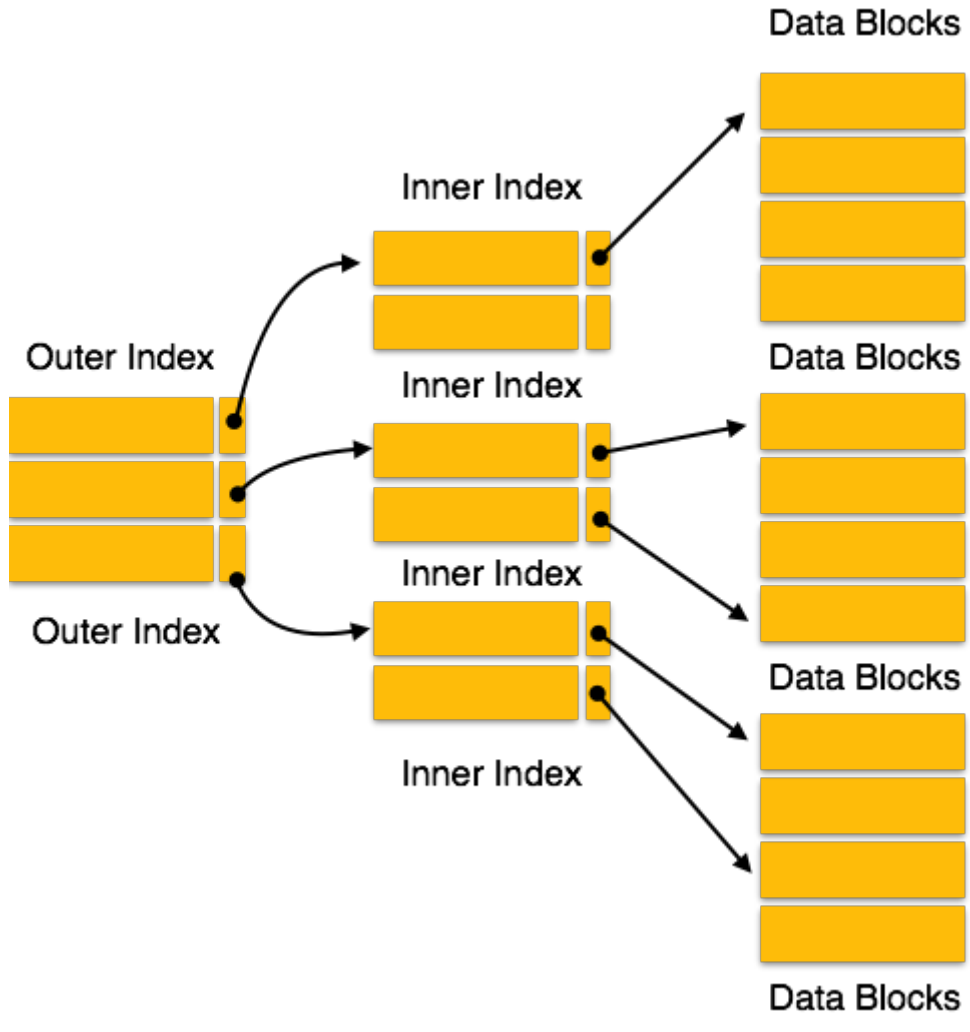
- In **dense index**, there is an index record for every search key value in the database.
- This makes **searching faster**, but **requires more space** to store index records itself.
- Index records contain **search key value** and **a pointer to the actual record** on the disk.

Sparse Index

- In **sparse index**, index records are not created for every search key.
- An **index record** here contains **a search key** and an actual **pointer to the data on the disk**.
- To search a record, we first proceed by index record and reach at the actual location of the data.
- If the data we are looking for is not where we directly reach by following the index, then the system starts sequential search until the desired data is found.

Multilevel Index

- Index records comprise search-key values and data pointers.
- **Multilevel index** is stored on the disk along with the actual database files.
- As the size of the database grows, so does the size of the indices.
- There is an immense need to keep the index records in the main memory so as to speed up the search operations.
- If **single-level index** is used, then a large size index cannot be kept in memory which leads to multiple disk accesses.



- **Multi-level Index** helps in breaking down the index into several smaller indices in order to make the outermost level so small that it can be saved in a single disk block, which can easily be accommodated anywhere in the main memory.

Transactions Management & Concurrency Control

- Press `Space` to navigate through the slides
- Use `Shift+Space` to go back

What is a Database Transaction?

- A transaction is a logical unit of processing in a **DBMS** which entails one or more database access operation.
- In a nutshell, database transactions represent **real-world events of any enterprise**.
- All types of database access operation which are held between the beginning and end transaction statements are considered as a single logical transaction.
- During the transaction the database is **inconsistent**.
- Only once the database is **committed** the state is changed from **one consistent state to another**.

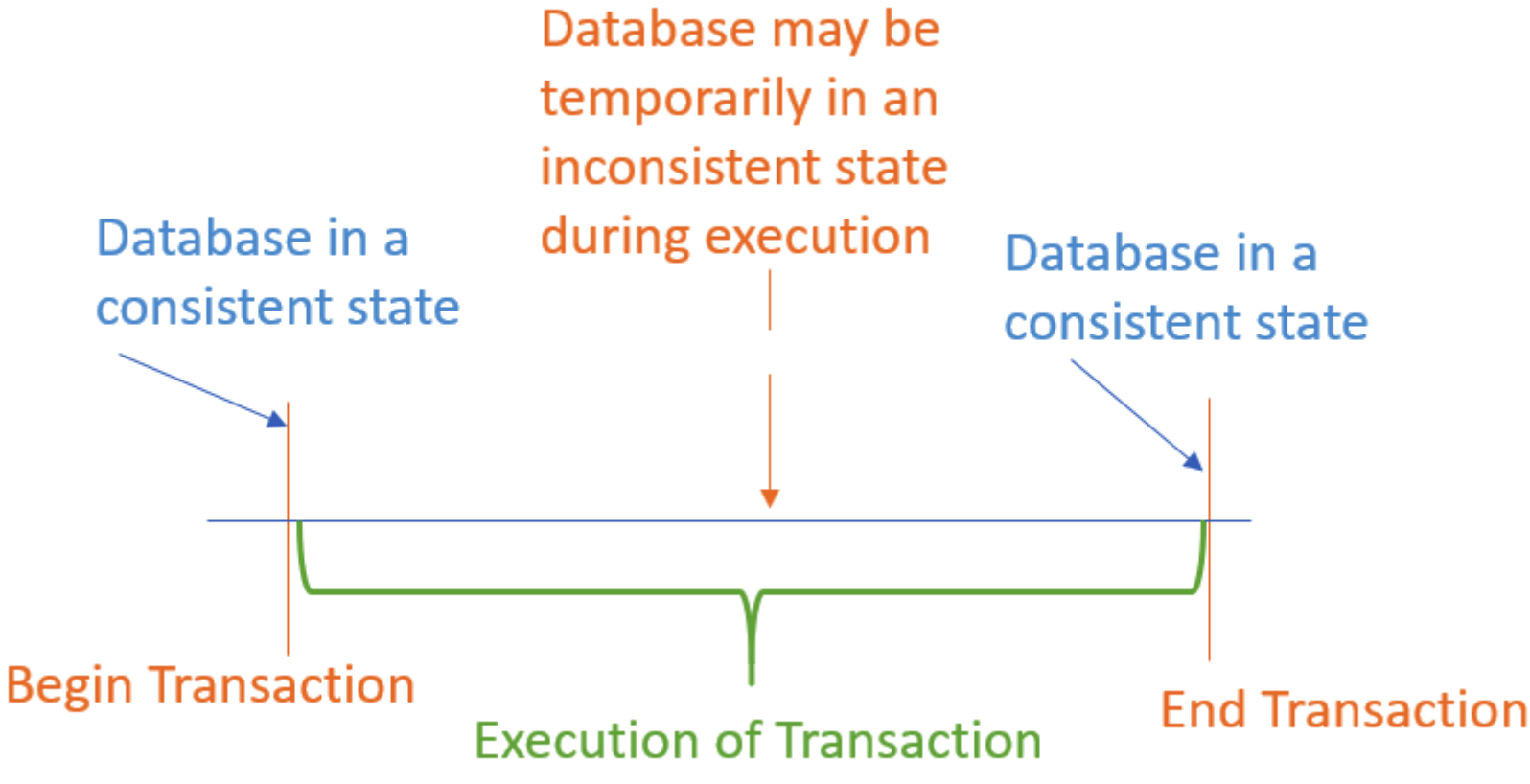
What is a Database Transaction?

- **A transaction is a unit of work that you want to treat as "a whole"**
- It has to either happen **in full** or **not at all**.

- A classical example is **transferring money from one bank account to another**.
- To do that you have first to withdraw the amount from the source account, and then deposit it to the destination account.
- The operation has to succeed in full.
- If you stop halfway, the money will be lost, and that is **very bad**.

- In modern databases transactions also do some other things - like ensure that you can't access data that another person has written halfway.
- The basic idea is the same - transactions are there to ensure, that no matter what happens, the data you work with will be in a sensible state.
- They guarantee that there will NOT be a situation where money is withdrawn from one account, but not deposited to another.

What is a Database Transaction?



What is a Database Transaction?

- A transaction is a program unit whose execution may or may not change the contents of a database.
- The transaction is executed as a single unit
- If the database operations do not update the database but only retrieve data, this type of transaction is called a read-only transaction.
- A successful transaction can change the database from one **CONSISTENT STATE** to another DBMS transactions must be **atomic, consistent, isolated and durable**
- If the database were in an inconsistent state before a transaction, it would remain in the inconsistent state after the transaction.

Concurrency Control in Transactions

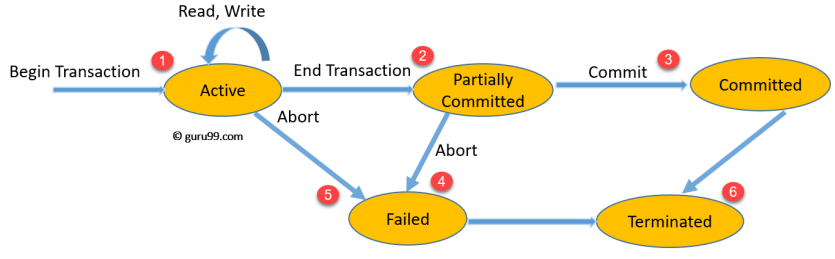
- A database is a shared resource accessed.
- It is used by **many users** and **processes concurrently**.
    - For example, the banking system, railway, and air reservations systems, stock market monitoring, supermarket inventory, and checkouts, etc.
- **Not managing concurrent access may create issues like**:
    - Hardware failure and system crashes
    - Concurrent execution of the same transaction, deadlock, or slow performance

States of Transactions

The various states of a Database Transaction are listed below:

State	Transaction types
Active State	A transaction enters into an active state when the execution process begins. During this state read or write operations can be performed.
Partially Committed	A transaction goes into the partially committed state after the end of a transaction.
Committed State	When the transaction is committed to state, it has already completed its execution successfully. Moreover, all of its changes are recorded to the database permanently.
Failed State	A transaction considers failed when any one of the checks fails or if the transaction is aborted while it is in the active state.
Terminated State	State of transaction reaches terminated state when certain transactions which are leaving the system can't be restarted.

States of Transactions

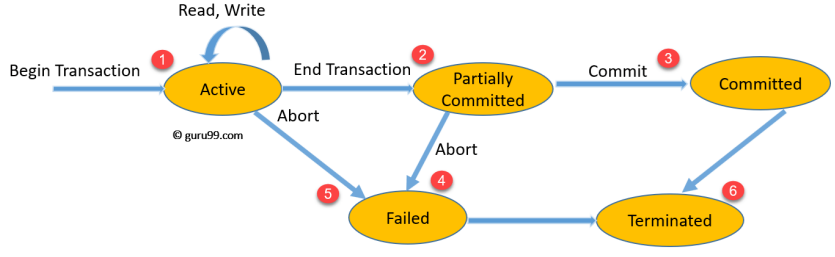


1. Once a transaction states execution, it becomes active. It can issue **READ** or **WRITE** operation.

2. Once the **READ** and **WRITE** operations complete, the transactions becomes partially committed state.

3. Next, some recovery protocols need to ensure that a system failure will not result in an inability to record changes in the transaction permanently. If this check is a **success**, **the transaction commits and enters into the committed state**.

States of Transactions



1. If the check is a fail, the transaction goes to the **Failed state**.

2. If the transaction is aborted while it's in the active state, it goes to the failed state. The transaction **should be rolled back** to undo the effect of its write operations on the database.

3. The **terminated state** refers to the transaction leaving the system.

What are ACID Properties?

- For maintaining the integrity of data, the DBMS system you have to ensure ACID properties.
- **ACID** stands for **A**tomicity, **C**onsistency, **I**solation and **D**urability.

ACID: Atomicity

- **Atomicity** states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none.
- There must be no state in a database where a transaction is left partially completed.
- States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction.

ACID: Consistency

- **Consistency**: the database must remain in a consistent state after any transaction.
- No transaction should have any adverse effect on the data residing in the database.
- If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.

ACID: Durability

- **Durability**: the database should be durable enough to hold all its latest updates even if the system fails or restarts.
- If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data.
- If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.

ACID: Isolation 

- **Isolation**: in a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system.
- No transaction will affect the existence of any other transaction.

ACID: Example 

- Example of **ACID**:

```
Transaction 1: Begin X=X+50, Y = Y-50 END
Transaction 2: Begin X=1.1*X, Y=1.1*Y END
```

  - **Transaction 1** is transferring **€50** from **account X** to **account Y**.
  - **Transaction 2** is crediting each account with a **10% interest payment**.

- If **both transactions are submitted together**, there is no guarantee that the **Transaction 1** will execute before **Transaction 2** or vice versa.
- Irrespective of the order, **the result must be** as if the transactions **take place serially one after the other**.

Serialisability 

* 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.

- **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.

- **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.

Serialisability 

- 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
- In case these two transactions are working on the same data, then the results may vary.
- This ever-varying result may bring the database to an **inconsistent state**.
- To resolve this problem, we allow **parallel execution of a transaction schedule**, if its transactions are either serialisable or have some equivalence relation among them.

Equivalence Schedules

- An **equivalence schedule** can be of **3 types**:
    - **Result Equivalence**
    - **View Equivalence**
    - **Conflict Equivalence**

Equivalence Schedules: Result

- **Result Equivalence**
    - If two schedules produce the same result after execution, they are said to be result equivalent.
    - They may yield the same result for some value and different results for another set of values.
    - That's why this equivalence is not generally considered significant.

Equivalence Schedules: View

- **View Equivalence**
    - Two schedules would be view equivalence if the transactions in both the schedules perform similar actions in a similar manner.
    - For example:
        - If **T** reads the initial data in **S1**, then it also reads the initial data in **S2**.
        - If **T** reads the value written by **J** in **S1**, then it also reads the value written by **J** in **S2**.
        - If **T** performs the final write on the data value in **S1**, then it also performs the final write on the data value in **S2**.

Equivalence Schedules: Conflict

- **Conflict Equivalence**
  - Two schedules would be **conflicting** if they have the following properties:
    - Both belong to separate transactions.
    - Both accesses the same data item.
    - At least one of them is "write" operation.
  - Two schedules having **multiple transactions with conflicting operations** are said to be conflict equivalent if and only if:
    - Both the schedules contain the same set of Transactions.
    - The order of conflicting pairs of operation is maintained in both the schedules.

Concurrency Control

- 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 **serialisability** of concurrent transactions.
- Concurrency control protocols can be broadly divided into two categories:
    - **Lock based protocols**
    - **Time stamp based protocols**

Lock-based Protocols

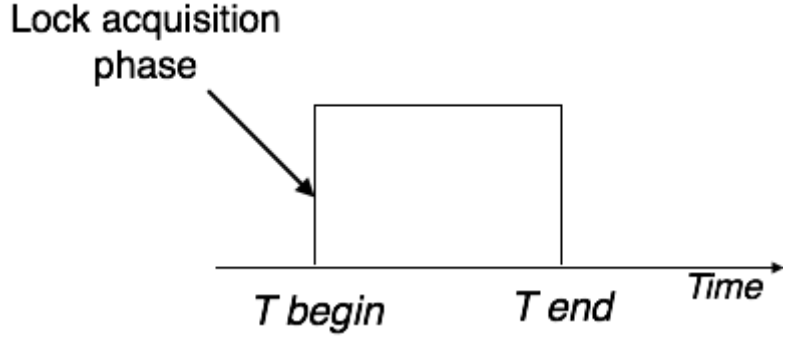
- Database systems equipped with lock-based protocols use a mechanism by which any transaction cannot read or write data until it acquires an appropriate lock on it. Locks are of two kinds:
    - **Binary Locks** – A lock on a data item can be in two states; it is either locked or unlocked.
    - **Shared/exclusive** – This type of locking mechanism differentiates the locks based on their uses. If a lock is acquired on a data item to perform a write operation, it is an exclusive lock. Allowing more than one transaction to write on the same data item would lead the database into an inconsistent state. Read locks are shared because no data value is being changed.
- Types:
    - **Simplistic**
    - **Pre-claiming**
    - **Two-Phase**
    - **Strict Two-Phase**

Simplistic Lock Protocol

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

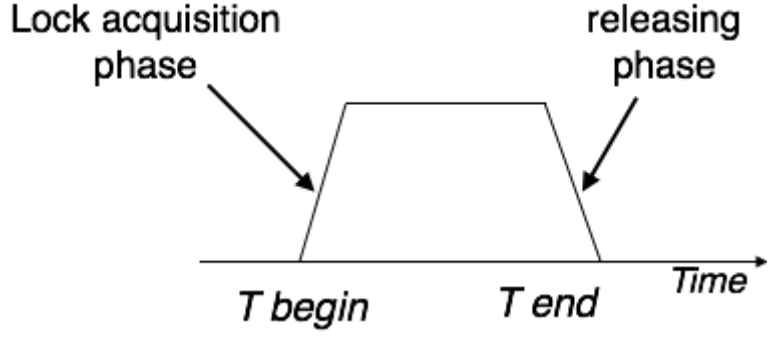
Pre-claiming Lock Protocol

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



Two-Phase Locking 2PL

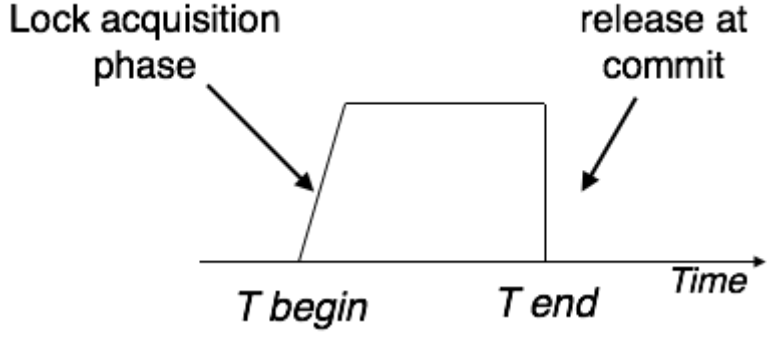
- **Two-Phase Locking 2PL**
  - This locking protocol divides the execution phase of a transaction into three parts.
  - In the first part, when the transaction starts executing, it seeks permission for the locks it requires.
  - The second part is where the transaction acquires all the locks.
  - As soon as the transaction releases its first lock, the third phase starts.
  - In this phase, the transaction cannot demand any new locks; it only releases the acquired locks.



- **Two-phase locking has** two phases:
    - One is **growing**, where all the locks are being acquired by the transaction
    - The second phase is **shrinking**, where the locks held by the transaction are being released
- To claim an exclusive (write) lock, a transaction must first acquire a shared (read) lock and then upgrade it to an exclusive lock.

Strict Two-Phase Locking

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



- **Strict-2PL does not have cascading abort as 2PL does**

Timestamp-based Protocols

- **Timestamp-based Protocol**
    - The most commonly used concurrency protocol
    - This protocol uses either system time or logical counter as a timestamp
    - Lock-based protocols manage the order between the conflicting pairs among transactions at the time of execution, whereas timestamp-based protocols start working as soon as a transaction is created
    - Every transaction has a timestamp associated with it, and the ordering is determined by the age of the transaction.
        - A transaction created at 0002 clock time would be older than all other transactions that come after it.
        - For example, any transaction 'y' entering the system at 0004 is two seconds younger and the priority would be given to the older one
    - In addition, every data item is given the latest read and write-timestamp. This lets the system know when the last ‘read and write’ operation was performed on the data item

Summary

- **Storage Systems**: Primary, Secondary, Tertiary
- **RAID**
- **File Structure**: Sequestial, Hash, Clustered
- **File Operations**
- **Indexing**: Dense, Sparse, Multilevel
- **Transactions**
- **ACID**
- **Serialisability**
- **Concurrency Control**